Final Report: Excelerate IMDB Database

Our project began with a search for datasets. The range of topics and the sheer number of data sets available publically on just Data.gov and Reddit alone was incredible. We came across many interesting data sets ranging from business to transportation. Notably, we discovered data on death rates due to auto accidents for each U.S State and admissions data from the University of California. Although data sets such as admissions statistics and auto mortality rates were interesting to some degree, we could not see ourselves committing to and working with them for the rest of the semester. After several days of searching data dumps, we got an idea when talking about movies during a study break. Both of us had watched The Shawshank Redemption (1994) and although we could not deny it was a great movie, I personally did not think it deserved an IMDB rating of 9.2, the same rating as The Godfather (1972). Because of our familiarity with IMDB ratings and the fact that subject movie ratings were something we argued about, we decided to use IMDB ratings as our topic.

Despite finally deciding on a data set to use, there was the problem of finding one. Unlike most data sets we encountered that were in the form of CSV (comma-separated values), JSON (JavaScript Object Notation) or XLM (Extensible Markup Language), we struggled to find IMDB movie data that were in these formats. The official IMDB website has a link to download data, but they were all in list format. We discovered (after installing MySQL 5.7) this would be very difficult to incorporate into our project since data fields in the file were not organized in columns. Perhaps IMDB did not have the license to share their files to the public in common formats such as CSV. Converting the list files to CSV was an option we considered, but that would be a big task on its own that was beyond the scope of the first lab. The solution that we came to was to build a "watch-list" from the IMDB website that could be exported as a CSV file.

Due to time constraints, we simply chose the top 250 films at the current time and saved them to our watch-list. Some of these movies include The Dark Knight (2008), Inception (2010), and Django Unchained (2012), all of which we had both watched. Additionally, we were able to find a DVD data set in CSV format that has all movies ever released as a DVD. This csv file was downloaded from hometheaterinfo.com and contained close to 300,000 titles and entries.

The next step was to build data models with the movie dataset that we had. Our watch-list had many columns of data, but the fields we were most interested in were title, IMDB rating, number of user votes, runtime, release year, director, and genre. The DVD data set also complimented our data from watch-list as it had similar fields and additional ones such as price and sound. Based on this, our conceptual model has five tables: Movie, Genre, Director, Dvd, and Studio. Movie and genre has a many to many relationship since a movie may have (optional) more than one genre and each genre such as action has many movies categorized under it. Director and movie also has a many to many relationship. Each movie could have co-directors (optional) in addition to the lead director and individual director commonly director more than one movie. In contrast, DVD and movie tables have a one to one relationship. Each DVD release can only contain one movie and each movie can only be on one DVD. The last relationship is a many to many between the DVD and studio tables. Each DVD can have (optional) multiple studios that worked on the DVD release and each studio commonly releases many DVDs. Building the conceptual model was fairly straightforward since it was covered in class and examples were given. The Lucid Chart application made it very simple to map out the relationships we wanted. We liked the wide array of options we had at hand to make graphs. It was also helpful that a Lucid Chart demo was shown in class to model students and professors in UT.

After completing the conceptual model, we adapted it to our logical model. All of the basic tables in our conceptual model translated into our logical model. At first we created tables for IMDB rating and number of votes however, this created problems since there were no fields to relate the rating and vote tables back to other tables. We found that it was much easier to just incorporate these fields into the movie table. The movie table consists of five fields: title, release year, runtime, IMDB rating, and user votes with the title being the primary key. All of the values for these fields are either varchar, integer, or float. The DVD table also consists of five fields: title, release date, sound, price, and rating with title being the primary and foreign key in this case. Values in the DVD table are either varchar or float. The studio, genre, and director tables all consist of only one field that is also the primary key which are studio, genre name, and director name respectfully. The main challenge we faced building the logical model was the need for junction tables to relate tables that had many to many relationships. These challenges are noted during the create table statements. The first junction table was created between genre and movie with genre_name and title being the primary keys and foreign keys. This was used to relate back to the primary keys title and genre_name to the parent tables movie and genre respectfully. Likewise, a second junction table was created for the tables director and movie. Title and director_name were used as foreign and primary keys to relate back to the movie and director tables. Our last junction table used title and studio_title as foreign and primary keys to relate back to the DVD and studio tables.

As beginners with MySQL or SQL code before, writing code to create tables in MySQL was new to us. Thankfully the majority of SQL syntax was not to complex and difficult to understand. Tables from our logical model were translated and created successfully in MySQL using the “create table” command and following the correct syntax for creating the columns. One of the problems we encountered as a team was creating the junction tables. Initially we ran into key constrain issues when trying to run the code. Error messages appear stating that keys were not referenced correctly. We discovered that the composite keys in the junction table had to match the primary keys of the two tables it was referencing. For example, the composite keys genre_name and title of the genre junction table had to reference the primary keys title and genre_name of the movie and genre tables respectfully. Another technical difficulty we ran into was sourcing the create table code in MySQL. The create table statements were initially written in .txt file on notepad, but MySQL only runs files with .sql extensions. On one of our machines, sourcing the file with the conventional file path using backslashes resulted in an error stating the out-file being sourced did not exist. After trying different syntax, it was discovered that double slashes are required in the file path for the source command to work. For example, C://User//Desktop//CS327E-Database instead of C:\User\Desktop\CS327E-Desktop. Our only explanation would be that we worked on different Operating systems, Windows 10 and Windows 8.1.

Having worked previously with Python, the task of writing code to import data from our csv files into tables in MySQL appeared a less daunting task on lab 2. The only addition to python needed was the PyMySQL connector that was used to write import statements such as “insert” and the database connect that was used to connect python to our database. Regarding the database connect file that was provided, an inconvenience we faced was the need to change our login credentials for our individual MySQL accounts every time we committed changes or new files to our repository. This slight problem was likely due to using an older example of database connect from the Snippets example repository.

A total of eight import scripts were written, five scripts corresponding to the base tables and three scripts corresponding to the junction tables. Starting with the main movie table, we imported the records title, imdb_user_rating, runtime_in_minutes, release_year, and number_of_user_votes from our raw data file watchlist.csv. we used nested for loops and conditional statements along with built in functions such as “enumerate” and “replace” to go through each row in the raw file and extra the data we needed. Using the same methods, the genre, DVD, director, and studio base table scripts were created. Next, the scripts for the junction tables were written. Both genre_junction_table and director_junction_table retrieved raw data from watchlist.csv while the studio_junction_table retrieved data from dvd.csv using the same techniques as the scripts for the base tables. All import scripts included exceptions that printed out error messages when the scripts encountered problems when running. The critical error we made when running the import scripts was that we ran scripts out of order. This error was quickly addressed when we realized the base tables needed to be populated with data before any junction tables can be populated. Because the keys in the junction tables were looking to reference keys in base tables that did not exist yet, the exception conditionals in our scripts printed out error statements.

After our import scripts were able to populate our database without errors, rollback scripts were created for each table so the import scripts can be run multiple times. The import scripts were written using the PyMySQL connector when a simple statement “drop from” designated table. This allowed us the ability to alter the data in our tables without the need to manually clear each table. Having all of our import and rollback scripts completed, a populate database file was made to call/import all the scripts. We found it strange when the program ran smoothly to completion on one of our machines and failed miserably on the other despite the code being exactly the same. The error message read “access denied for user roor@localhost (using password: YES”. At first we checked the database connect file to make sure our login credentials were entered correctly, but the error still persisted. Next, we tried the test program for database connect to see if the connection to MySQL was working, which it was. After trying a myriad of fixes suggested on Stack Exchange such as using the command prompt shell to grant all user privileges to the IMDB database we created, we finally decided to reinstall MySQL. When the error persisted even after a reboot and reinstallation of MySQL server and all the extensions, we began to run out of ideas. It was only after going through the file and altering code bit by bit that we discovered the names of the import scripts being called in populate database must be capitalized for it to run. This was bizarre since the actual file name for the scripts were not capitalized. For example, the import script for the movie table is a python named import_movie.py, but to use the script in populate database it must be referenced as import_Movie. One especially frustrating problem we kept running across was simply connecting to the database to allow all this to work. We kept having to modify our code with code we found on websites like stackexchange.com. After the project was turned in it was discovered what the problem was. We had established a db_connect file based on the first code given to us in the snippets repo. The problem was we didn’t realize that in the days after that initial code was given, new and revised code was pushed. This new code made connecting much easier. In addition to making the connection process easier, it solved another annoying roadblock which was up until that point, we had to type each of our mysql login info on each script to connect. With the new db_connect code we only had to enter our login info on that file and not every import script. Despite these setback, we were still able to finish lab 2 before the deadline.

Using the material we have learned so far in class on SQL queries, we wanted to create some interesting queries involving our IMDB database for lab 3. Most of the requirements and criteria such as using group bys, order bys, where clauses, and aggregates were fairly straightforward. For instance, to show all movies in our database that received a particular rating, above a particular rating, or below a particular rating, a select statement was used along with an order by. To list all movies that were release in a certain year is also very similar. Queries became more complicated when we needed to join multiple tables. For example, to show DVD titles of an MPAA rating below a certain price required a full join between the movie and DVD tables. This was difficult because MySQL 5.7 does not support full joins. Instead we were able to achieve this with a union of left and right joins between the two tables. Several views were also created to display some interesting data. By joining the DVD and studio junction tables in a view, we were able to show which studio made the most DVDs and thus how profitable each studio is likely to be. With our queries completed, we developed a query interface program in python that allowed users to choose their own inputs in our IMDB database using the queries we had written. Lastly, we added SQL injection protection to prevent users from damaging our database. We found it both frightening and fascinating that a simple “drop table” command from a user input could potentially wipe a database if SQL injection protection is not implemented.

Despite difficulties we encountered working on through the projects, one main reoccurring problem was glitches with the GitHub client. The most common issue is the amount of time it takes for files to be updated in our local repository. After a commit has been made, it would often take a few hours to see the changes and for them to take effect. This issue became a big setback in terms of submitting projects since the last commit id was required as part of the submission. A workaround of this technicality was instead of using the GitHub client, we used the GitHub shell. The shell console provided a much more broad range of options. We were able to commit just single changes as opposed to committing every change that occurred in every file. The shell console also provided feedback as to which file had been successfully committed and pushed onto the private repository.

The final project was a culmination of each of the previous labs. This made the final project seem somewhat lighter on the workload but this was just because we knew the process for solving the problems laid out before us. The project required us to pull data from twitter using a restful API and store the JSON twitter data in a table within our database. This is where the knowledge obtained from lab one came in handy. First, we had to extend our database to hold a new table entitled Tweet. Like in lab one, this involved making a new create table statement. One difficulty that we came across was that in this new table we were storing data types and making columns that we hadn't seen before. The first of these was the column name tweet_doc which held the tweet data in JSON format. The second was the tweet_id and screen_name columns whose data was pulled directly from the tweet. Additionally, the created_at column was created to hold a datetime data type which we hadn't used before. We then chose to link the new table Tweet to the parent table Movie using a foreign key 'title' in the Tweet table that links to Movie.title. This gave us the database setup necessary to begin populating the new table.

To begin populating the new table we had to make a twitter app to get multiple API keys. These keys were used in a python program called api_client.py. This program used the package tweepy to connect to twitter. Then, just like in lab 2, we connected to a data source (in this case twitter), pulled data, and committed that data into the Tweet table using the run_stmt program in db_connect. The difficulty we ran into here was the use of the API keys. If not copied exactly the program wouldn't connect to twitter and after a long while searching I realized I had copied the keys incorrectly. That quick fix solved our problem and we could connect to the database. Next, we decided to limit the number of tweets we would take. Because we were searching twitter for movie titles within our database, a database with 250 movies, we couldn't just let it pull every tweet. This is because in any given moment hundreds if not thousands of people are tweeting about the latest popular movies, some of which were in our database. At first, we didn't realize this and when we ran the api_client.py program we got over 2000 tweets for just one recent movie, which was hardly diverse enough to make queries on in the next part of the lab. To fix this we modified the code given in the snippets repo, specifically at the line that defines the twitter_cursor variable. We modified this variable with a .items(50) at its end which limits the number of tweets pulled for one movie to 50. It does this by dropping the twitter query for a movie (ex = '@deadpool or #deadpool') after 50 tweets are found and moving to the next query. This allowed us to get a range of tweets from over a hundred movies. The data, being stored in the table, was now ready to be queried.

This next part of the final project was the easiest because it used knowledge from the most recent lab. The data just needed to be queried to show we could present the data from our table using select statements. We needed five select statements and two of them had to have joins. Naturally, we joined the Tweet table with the parent table Movie on tweet.title = movie.title. After that it was just a matter of coming up with five select statements. One statement was made showing all the twitter screen names and how many tweets each posted. Another showed how many tweets were made for each movie. These were all relatively easy to make since they resembled the queries used in lab 3. Finally, we had to make a backup of our database to turn in. This part gave some problems. First, I had to figure out to run the cmd as an administrator to be able to use mysql dump. Secondly, our Tweet table was far too large to be included in the backup. Including it would have put our file size to over 6 gb. This meant we had to exclude the Tweet table from the backup.sql file. Without the Tweet table the file size was only 50 kb and perfect to turn in.

This final project was a great wrap up of all the lessons learned throughout the semester. We learned many important lessons not just about database design and implementation but also about datasets and data handling in general. We learned early in the semester that data is rarely available in a presentable way and it takes effort to polish the data. This was first learned when looking for a dataset. When we decided on the IMDB dataset we discovered that IMDB had no easily downloadable files in csv format. Instead, they had dozens of .list formatted files near impossible to read. We obfuscated this problem when I discovered that making a 'watchlist' on the IMDB website would allow you to export data as a csv file. Though this fix allowed us to continue, we only had 250 movies to work with instead of the millions available. When collecting and loading the data into the database we learned that, more often than not, you must reformat the data into a readable state. For example, in our DVD.csv file there were studio names listed for each movie but some movies had a studio "Warner Brothers" while others had "Warner Bros". These types of errors are annoying but easily fixed through some python string manipulation. All these lessons considered, the biggest lesson learned is that data isn't something easy to work with. It doesn't seem to want to cooperate but we learned that keeping an open mind and being flexible was the best way to solve problems.