CS 327E Elements of Databases
Final Report

Authors:
Daniel Smarda (djs3745)
Changyong Peter Yi (cpy86)

Adjudicator:
Professor Shirley Cohen

11 December 2017

Abstract

The purpose of this report is to outline and analyze the most important skills and accomplishments of the CS 327E: Elements of Databases course and to critically evaluate the efficacy of the implemented solutions to the presented problems, which involved cleaning, merging, and visualizing media-related data from several popular, large entertainment databases. The structure of this paper follows the chronology of the course: We begin with an overview of the tools used to manipulate and analyze the data, continue through discussion of our approach to each of the challenges encountered and the advantages and disadvantages thereof, and finish with a discussion about the extensibility of the course concepts to "real-world" data challenges. We conclude that most of our solutions were appropriate for our dataset but would need to be adjusted for datasets with different characteristics, and that both the theoretical underpinnings and tools applied in this course have already proven useful in applications ranging from data mining to website development to industrial data processing.

1: Introduction

1.1: Tools

To store our final transformed data, our chosen Relational Database Management System (RDBMS) was PostgreSQL. Compared to the simpler but less-customizable and less-robust SQLite, PostgreSQL was better able to handle the large size of our dataset and the complexities of combining multiple databases worth of data. To the extent of the scope of this project, PostgreSQL is very similar to MySQL and MariaDB, the most similar RDBMS options to PostgreSQL. While PostgreSQL does have a GUI for management (pgAdmin), we used the more direct psql command line interface to execute queries and commands.

Several Amazon Web Services (AWS) programs were used, including but not limited to: S3 to store raw data, Identity Access Management (IAM) to share credentials, Athena to explore and query raw data, and Elastic MapReduce (EMR) in concurrency with Apache Spark in Python for cluster computing. As previously stated, we used Amazon Quicksight for data visualization, a quite limited visualization tool in terms of functionality and customizability when compared to

other data visualization packages like ggplot (R) or matplotlib/seaborn (Python), but one that generates aesthetically valuable visualizations very quickly.

Administratively, we used Github.com for source and workflow control, stache for secure credential storage, and LucidChart for Entity Relationship Diagram (ERD) creation.

1.2: Datasets

The data we used to populate our database was entertainment-related data. We started with an Interactive Movie Database (IMDB) database (see www.imdb.com), which contains information on a huge number of movies, documentaries, television episodes and similar works, as well as role, salary, birthday, and other information for the persons involved. We added matched tags, budget, and box office information from the Movielens dataset, a website which recommends movies based on a user's opinions of other films (https://movielens.org/). Finally, we imported data from the Cinemalytics database to incorporate more information about Bollywood films, songs, and performers in our initial database than what was available in the IMDB and Movielens datasets.

2: Concept Development

2.1: Data Exploration

Before populating the database, we needed first ensure that we understood our data and then give proper forethought to the schema that defines how the database is organized. Accessing the S3 buckets where the IMDB data was stored via Amazon Athena, we explored the IMDB data with increasingly complicated queries, beginning with simple COUNT(*) queries to view the size and columns of the tables, and cumulating with queries requiring the joining of up to four tables. All 20 of the questions that these queries answered can be found in Appendix A.

The most difficult part of both understanding the data and presenting it in such a way that a non-technical audience would be able to understand the nature of the data was creating an

effective Entity Relationship Diagram (ERD). The central questions that were difficult to answer were: How many column relationship should we include? Do we restrict relationships drawn on the diagram to those of a foreign key/primary key nature, or include all column overlaps? Initially, we included all relationships that made sense with respect to the dataset (i.e., all relationships between two columns which could realistically or meaningfully be joined on), but this made our ERD quite cluttered, so we later modified it to include only the most meaningful (e.g., primary key/foreign key) relationships. To create our ERD, we used LucidChart, a website which, when compared to other similar diagramming options such as draw.io, is quite slow but has a wider range of effects relevant to ERD aesthetics.

2.2: Database Design

Once we had an understanding of the data, we created and populated our local PostgreSQL database instance from the schema visualized in our ERD. While other methods for populating databases certainly exist, some of which are discussed later in this report, the initial data upload was executed using PostgreSQL \copy commands, which directly copied the tables from the csv files in the S3 bucket.

Relational databases as a data storage paradigm have several advantages and disadvantages. While the pretty strict structure ensures data integrity, the relational algebra takes time to learn, and the computer processing of these constraints slows down RDBMSs compared to NoSQL databases that are more effective for large datasets. While the learning curve issue was solved through lectures and studies, the computer processing problem surfaced during the copying of the over eight million records for some tables, as execution of the \copy command was taking upwards of several hours for the largest tables. To solve this, the csv files were split into smaller

sizes to accommodate for the huge numbers and copy commands were implemented so that uploads can be completed in parts.

To facilitate business analysis, we also created a star schema with a centralized fact table and dimension tables. Specifically, the fact table was able to quickly answer the question: For a given combination of title-type, genre, and year, how many appalling, average, and outstanding titles were there? At first, the small schema that we used to organize the necessary information to answer this question was simply a child of the general schema of the database. It was not until we tried to write queries to acquire the required information from our schema that we realized that with our child schema, join operations were almost always necessary, at which point we reconfigured our schema to the correct star schema.

2.3: Data Visualization

To communicate meaningful conclusions mined from our data, we used PostgreSQL aggregate functions to process and Amazon Quicksight to visualize data. Our limited but improving knowledge of subqueries made attaining certain information (e.g., in each year, the maximum average movie length for a given genre and the corresponding genre) difficult. Additionally, while admittedly quick in creating deliverables, Quicksight has limited customization to visualize somewhat complicated queries, especially compared to code-based visualization tools such as matplotlib/seaborn (Python) and ggplot (R). Given our time frame and limited toolset, queries and visualizations had to be carefully planned to avoid queries that could not be visualized with Quicksight or formatted in the short time frame with any other tool. For faster future access to these conclusion tables, views were also created, with the decision of whether to materialize the given views decided by the processing time of the query.

3: ETL Application

3.1: Distributed Computing

Introducing the concept of the Amazon's EMR cluster was straightforward. Since uploading the data in a csv file directly to the database would take up valuable resources and generally too much time, we used clusters to separate the work into nodes, and seamlessly uploaded the necessary data into the database. Using Spark had the added benefits of allowing us the freedom to parse the file, trim or modify the data to our needs, and upload the data efficiently to the database. After analyzing the dataset and the given Python script, we would then turn on the EMR cluster, modify our credentials, and proceed to run our script to populate a new column in an existing table. We checked to see if the script returned any error messages and ran an aggregate query to check the number of non-null entries. After confirming the results, we wrote comments before every RDD function, causing this file to be an effective reference of RDD operation examples for the remainder of the project.

The most severe difficulty of this task was familiarizing ourselves with the structure of the EMR cluster. Learning how to navigate through the cluster; configuring our credentials according to the complex interactions of our personal computers, the EMR cluster nodes, and our PostgreSQL database; and uploading our Python files to run the script took some trial and error. Another valuable experience that we learned through accomplishing this task was the added responsibility to control our own cluster. Given that one day we are going to work in the programming field, using resources such as Amazon's EMR clusters gave us the responsibility to turn our clusters on when we need them and to remember to turn them off when we do not.

3.2: Data Transformation

Once that we had familiarized ourselves with using the EMR cluster and RDD functions, we had the skills to flexibly and creatively modify a given Python script according to our needs. After creating new tables and analyzing the csv files to formulate a method to parse the data, we modified the Python script template to effectively join two tuples from two csv files and delete any duplicates. Upon successfully uploading the required data into the table in the IMDB database, there were still rows that had to be deleted. Since there was a relationship between title_tags and title_basics, the title_tags table needed to have title_id entries that exist in title_basics' title_id column. A simple query in psql solved this problem and was able to delete the correct amount of rows from title_tags.

Understanding the RDD functions of a completed script was simple; however, knowing when to use certain functions in a template script proved to be somewhat difficult. After parsing the csv files, tuples were used to hold the parsed data and various RDD functions modified these tuples multiple times. Understanding exactly how each map or reduce function modified the RDD in each step of the transformation process was at first difficult without using output to troubleshoot, but a log file helped alleviate the problem by printing out the results of each RDD function which allowed us to analyze tuple after each modification.

3.3: Efficiency Improvements

After learning the basics of writing our own Spark program, we implemented a more sophisticated modification to our script. Similar to the process of creating the title_tags table, another table called movie_financials was created, analyzed the data set, and formulated an approach to populate the new table using Spark. For this table, we were given a data set that had to be modified slightly to suit our needs: The date had to be parsed so that only the year was added

to the tuple; the title had to be converted to uppercase and trimmed of any exterior whitespace as well as encoded to UTF-8 to handle foreign characters; certain genres had to be modified to match with the existing genre column in the IMDB database; and finally we removed any characters from box office so that we could enter a number into the column with integer as its data type constraint. Previously, when we had attempted a similar operation, we added unnecessary rows and later deleted them manually. This new script would prevent adding unnecessary rows by analyzing the data before uploading it to the table. This way, only the required data is added to the table, removing the additional labor of needing to manually delete rows after uploading them.

To decrease the upload time, we created indexes on tables and columns that were frequently used. Unfortunately, however, after running the script in the EMR cluster, the upload speed came to a crawl. Even when leaving the cluster on overnight, only about a couple hundred entries were added when tens of thousands were required. Given that indexes were supposed to solve the upload speed problem, we revisited our index assignment statements and realized that our understanding of indexes were premature. Due to what we were comfortable with from past experience, we had only used indexes on only one column. It was not until further research that we realized that we could create indexes on multiple columns. After fixing the indexes, the upload speed improved dramatically all of the data was successfully added to the new table in a reasonable time.

3.4: Tool Evaluation

For each step in the ETL process of our final task, we evaluated the advantages and disadvantages of our two main options: Spark or SQL scripts. Generally, Spark scripts are more efficient especially for large datasets and data that needs extensive modification. However, the cluster computing used for Spark costs money, and the several hundred line-long Spark scripts are

often much more difficult to both write and understand than the much shorter SQL scripts for simple data transformations.

To merge the Cinemalytics Bollywood data into our database, we extracted data from five csv files to populate three new tables (songs, singer_songs, title_songs) and added new rows to the person_basics table with a newly added column in the IMDB database. Through a brief examination of the data set, we realized the format of the songs.csv file matched exactly to the songs table we created. In order to keep our code and process simple, we used the copy command in psql to upload the data. In the singer_songs table, we populated the table from the singer_songs.csv file only when person_id existed in both singer_songs.csv and persons.csv. The process seemed too simple to use Spark so we created two temporary tables and used the copy command on psql to upload the data. We then wrote and executed a new query on the rows of the two temporary tables that we needed and subsequently dropped the temporary tables. A similar approach was used in populating the title_songs table by creating two temporary tables. Instead of checking if values existed in both tables, we joined the two temporary tables by its movie_id and disregarded any null entries and inserted the results into title_songs.

Finally, adding new rows into person_basics required Spark since we needed to modify the data that was parsed. When parsing person.csv, we had to modify the date by only returning the year, and return null if the date column was empty. In addition, we had no need to join any tuples since we are only checking to see if the person_id in the singer_songs tuple existed in the persons tuple. When going through each tuple in persons, we checked to see if the persons' person_id existed in the singer_songs. If so, then added that tuple to the person_basics table.

There was some debate on choosing which Python object to hold the person_id from singer_songs tuple. We decided the specific "in" operation that we were utilizing was too simple

to necessitate processing RDD objects like we normally do for Spark operations involving multiple datasets or data sources. Initially we used a Python list to hold the elements, but we realized that we had no need for the indexing that is normally the main advantage of using Python lists over other data structures. By using sets instead, we saved memory space excluding indexes and the process proved faster than using a list.

Finally, when populating the title_songs table, we ended up uploading a slightly greater number of entries than needed, due to entries with the same title_id but different song_ids. Adding foreign key constraints to the populated table allowed us to find the title_ids with this case, and enabled us to delete entries with those title_ids.

4: Conclusion

The Elements of Databases course and the projects described in this report have given us an extremely well-rounded understanding of database administration. We learned theories of relational algebra, delving deep into understanding why certain queries are processed the way they are and why foreign key and not-null constraints exist. However, we also learned the practical tools used by many industry leaders (PostgreSQL, Github, Spark), and how to effectively communicate conclusions drawn from data. After all, while the course was primarily focused on data integrity, data really has no value unless useful conclusions are drawn from it, and learning the entire ETL process from data acquisition to cleaning through analysis is invaluable.

The skills and tools learned will expedite learning other software concepts and will be beneficial to a variety of software development projects in the future, an assertion which has already proven itself correct. Relational databases form the foundation of database administration and storage. The APIs and theories of NoSQL databases are primarily even taught with respect to relational databases, describing concepts in how they are similar to or different to their relational

equivalent. Web development, from models to url processing to formatting and style templates to backend database interfacing, is a wide, difficult discipline; this course ensures that in learning web development, like any other software that accesses or stores data in a database, we can focus on the new content and not have to simultaneously teach ourselves the relational database theory. In fact**,** one of the authors of this paper tried to teach himself web development over the summer but largely failed due to inability to understand Models and database design, which has been proven changed as he successfully set up a PostgreSQL backend database through Google Cloud Platform in another course. The tools used in the course, especially Spark, are obviously considered cutting-edge in industry, as one of the authors of this report received an offer for an ETL internship largely because he had experience with Spark.

One specific unsolved problem is that at one point in our final ETL project, a table was supposed to have ~15000 entries (specifically, movie_financials), but for some reason ours only ended up with ~13500. Due to our scrupulous revisiting of the logic of our code, there is a high probability that this was due to a misunderstanding of the definition of the task at hand. If we were to take the course over again, one of the main technical decisions we would have changed would have been to ensure that our EMR clusters and RDS instance are in the same region. While this did not strictly prohibit us from completing any part of the project, it added some unnecessarily complex configurations, especially with regard to security and authorization, and also appeared to slow down some data accessing operations that would have been faster if the EMR cluster and RDS instance were in the same region. Within the requested scope of the course, the previous two sentences describe the only main problem, but the topics that we believe would be useful and interesting to delve deeper into in the future include the following: advantages/disadvantages of data types, when and why indexes speed up vs. slow down queries, accessing our database with

more advanced visualization tools than Quicksight such as matplotlib, and taking advantage of Spark's Dataframe organizational capabilities and generally becoming more comfortable with the Spark environment.

Appendix A: Exploratory Query Questions

1. Find me the highest rated movie in the databases, but make sure it's kid-friendly so that I can watch it with my little cousin this evening.

2. This query searches for foreign movies by checking the primary title is different to its original title, since the original title uses the foreign. language and the primary title is the translation. It also returns the director's name. All of this will be ordered by the movie's starting year.

3. This query searches for people and the title they're most famous for, where the person has already deceased. It makes sure to pass living people by checking if death year is null or not, and is ordered by the title's starting year.

4. What is the oldest movie in the database that Tom Cruise starred in?

5. This query searches for people's name, the title they were starred in, and it's number of votes, by order of it's average rating from 10 to 0.

6. What movies do people hate so much they just NEED to rate it terribly?

7. Did the writers of any good TV Shows pass away in the last year or so?

8. This query searches for actors that were born after 2000 and prints their birth year, and orders it by actor's name.

9. I want to learn something but I don't have a lot of time. Show me the shortest good documentaries available, ideally made recently. I'm going to assume that any documentary with a length of less than 20 minutes is an erroneous entry in the database.

10. This query searches for the actor, title, and runtime minutes, where the
title is adult themed and is ordered from longest runtime to shortest.

11. Has George Clooney ever done a comedy movie? If so I'm quite curious, so curious in fact that I'd like to watch it tonight.

12. This query searches for the title, adult status, and average rating, where the adult status is true, and is ordered by highest rating to low.

13. Who directed the top-rated documentaries? Make sure ratings are validated by lots of people.

14. This query searches for the comedy title, and its average rating. It will be ordered by highest rating to low.

15. This query searches for actor name and number of votes, where the actor is still alive, and is ordered by the number of votes.

16. Show me old family movies so that I can educate my cousins on what Disney used to be back in the day. Though I don't think they'll have the patience for silent films.

17. How has the ratings of movies that James Franco has been in changed over time?

18. This query searches for the person's name if they wrote and/or directed. It's checked if the person is alive and is ordered by name.

19. This query searches for Wes Anderson Movies by its rating.

20. Generate a list of the oldest living actresses and actors.