Final Project Report, CS 327E

Blake Schmidt, bcs2363

Kyle Gruber, kcg675

1. Introduction

1.1 Goal

The goal of the final project was to create a database schema comprising multiple databases' worth of information about a variety of TV and film related data, and to ask specific questions about the data through queries and then visualize the answers.

1.2 Data Sets

Four different datasets about television and movie-related data were given. These include: the IMDB dataset, made up of 10 csv files containing information on directors, stars, titles, genres, ratings, writers, and more, the Movielens dataset, made up of 6 csv files containing information on five-star rating and free-text tagging activity from a movie recommendation service, the numbers dataset, made up of 36 txt files containing information on the financials of movies, and the Cinemalytics dataset, made up of 5 csv's containing information on Bollywood's singers and songs.

For every dataset, we used given samples and complete portions of the data for reference and planning.

1.3 Tools

Various tools were used throughout this project to create and analyze our database. Amazon Web Services (AWS) was used in several aspects of this project. AWS served as the location in which all of our data was stored. Using access keys that we created in AWS using the Identity and Access Management Dashboard (IAM), we could ensure that all of our data could only be securely accessed by us without the threat of any sort of breach. We also used Amazon's EC2 dashboard to make sure that only our IP address could access any of our information.

Within AWS contained multiple services that we took advantage of. The Amazon Relational Database Service (RDS) gave us the ability to set up, operate and scale a relational database in the cloud with the database engine of our choosing, in this case PostgreSQL. With RDS, we were able to create a database instance that we could use to create a schema with our own tables filled with thousands of pieces of data that we obtained from various CSV and text files which included movie and TV show data from places such as IMDB, Movielens and Cinemalytics. Our RDS instance could be accessed by a terminal-based front-end for PostgreSQL called psql. From psql, we could connect to our database and run SQL commands to create and manage databases, views, indexes and more. The same RDS instance that we created from the beginning of Lab 1 was used all the way through our project.

AWS also contains Amazon Elastic MapReduce (EMR), a web service that uses Hadoop, an open source framework to quickly and cost effectively process large amounts of data. With EMR, we were able to create clusters with which we could process our scripts to load data onto our RDS instance. Because EMR isn't included in the AWS free-tier, we were required

to terminate each cluster after we were done using it for the day and then to clone it again later so that we wouldn't run over our free $100 student credit.

Another Amazon service called Quicksight is a data analytics service in which we could easily visualize and explore our data with various graphs and dashboards. This service made it easy for us to visualize several pieces of our databases such as the various views we created.

In order to create ER diagrams, we used a website called Lucidchart. This website let us easily share diagrams with our partner so that we could create a map of our schema. With the ER diagrams, we were able to see exactly how each table was related to one another and each tables primary keys, foreign keys and data types could be mapped out easily.

We also used a Python API called PySpark, which exposes the Apache Spark engine for big data processing to Python. With PySpark, we were able to write scripts using Python in which we could access our data, manipulate it and then modify the data with our RDS instance. To achieve this, we opened up an EMR cluster, securely SSH'd into it, uploaded our script and then ran the script to process our data.

For every assignment from our Labs or Project that was due, we would submit all of our files to a Git repo that was shared between both partners. From the command line, we could add, share and modify these files over the air as well as have a central place where all of our submissions were held. With our Git repo, we could simultaneously work on the same assignment and easily share our changes between each other without any problems.

Once our assignments were completed, we used UT's Stache service where any sensitive information such as encryption keys could be held and accessed by our professor and TAs so that our assignments could be securely graded.

2. Analysis

2.1 Setup

We started the project by creating a folder in our Git repository that had been made in a previous lab assignment. All of the files from the 3 labs and final project were placed in separate folders on the root of the repository, and GitHub was used to track issues and save our progress in small incremental commits.

After gathering the files, we planned out a strategy for effective teamwork. We decided that it would be best to use all of the assigned class time to get as far as possible in each lab and milestone of the final project. We would then let the amount of work leftover dictate the amount of further collaboration and the form of further collaboration that would take place. For most aspects, like writing queries, pair programming was an adequate and efficient use of our time. Working together in real time on the same task allowed us to quickly and exhaustively think through complex questions with speed and accuracy. For other aspects, like accessing the RDS or utilizing Quicksight, solo programming worked best because they were neither complex nor easily divisible tasks. Likewise, for a majority of tasks we found that working together in person made the most sense and was the most effective, but for others we found that a simple phone call or text conversation would be sufficient for exchanging the necessary information and provide a benefit of not wasting time en route to a meet-up location.

2.2 Database Design

From the four datasets given, we started with the IMDB dataset and implemented it as a good basis for relating the data from the other datasets. We were careful in selecting proper

primary keys and identifying all the foreign key references during the first stages of design to progressively work towards a level of database normalization that would ensure the schema worked as intended. Looking at the other datasets after the implementation of the IMDB dataset revealed several connections between the schemas, and from that we were able to either create new tables from combinations of cross-schema data or alter pre-existing tables from the IMDB dataset to include the new schema data in a new column. For example, the table Title_Tags was constructed by using title_id from IMDB as one column in the table and song_id from Cinemalytics as another, whereas the table Title_Ratings was only altered to include a new numeric field to hold the MovieLens ratings associated with each title_id.

Building the visual diagram of the unified schema revealed just how the deeply data was interconnected, and this helped us craft several queries that accessed multiple tables at a time over the data.

2.3 Data Load

Once our database was designed, it was then time to create the empty tables that we would use to load our data into. With our first lab, we used Amazon's Athena service to create our tables using the S3 paths that were provided. With Athena, the tables were created and loaded at the same time. We simply needed to create the columns with their correct data types and then the S3 path would load all of the data. This meant that we could immediately start to query our database without any extra effort.

For the rest of our labs and our project, we manually created our tables using SQL commands. These tables would be created without any data in them except for the column names and data types along with the primary key or keys. The foreign keys could also be added

in this step but we generally waited to add those until after our data was loaded because the load times would otherwise take too long.

For our IMDB data in Lab 2, we loaded the data into our tables using the \copy command. This command would load one CSV file per command and so we would have one \copy command for every CSV file we had. One problem that we had run into while working on this is that the load times for some of the very large CSV files took an extremely long amount of time. To solve this issue, we split up those large CSV files into several separate CSV files and ran them as separate commands so that we could speed up our load times. Although this took quite a while due to the vast amount of data we were processing, we eventually managed to load everything properly. After this, we could then add the foreign keys without that step taking too much time.

For most of the data in the our final project, we used PySpark scripts to load our data. To do this, we created a script which would essentially use the S3 paths we were given and loaded the CSV or text files into the script. From there, we would create parse functions that would extract the data from each line of the file and make any necessary transformations such as capitalizing words or encoding a string in UTF-8 so that we could process international and special characters. The variables that this parse function would return would then be loaded into a Resilient Distributed Dataset (RDD), which is a fundamental data structure of Spark. An RDD is a read-only partitioned collection of records and we would use these to store all of our data as it is processed in our script. From there, we could use these RDDs to make any additional joins or mappings before starting the loading process.

For each RDD who's data that we wanted to load into our tables, we would create a "save to database" function which would do all of this. The RDDs would save our data as tuples, so we would then use the PySpark partitionBy() function to access a list of the tuples. A for-loop

would then loop through all of these tuples and access the variables that we needed. After storing the data that we wanted to load as variables, we would then write out an insert or update SQL statement depending on our situation and would then plug our variables into the statement and run an execute() function which would execute this command into our RDD.

After completing these scripts, we could then SSH into our cluster and upload and run the script which would load our data into our database. Although we would frequently encounter bugs in our code, we were able to read the crash logs in our command line to see what went wrong and we could fix the issue and upload the script again to run. This was the method that we used to load data for all parts of our final project except for two of them which had no joins or transformations needed, so for those we used the \copy command. After each data load, we would also find the count of each table so that we were sure that all of the data was properly loaded, as well as printing out a number of rows to make sure that the data was displaying properly.

2.4 Data Integration

For several aspects of the integration of our data, many different transformations needed to take place so that we could correctly insert the data into our tables. For all of the data files that we input into our scripts, a parse function would need to be implemented to extract the data and create an RDD out of them. In order to accomplish this, many of the files were CSV files and would be split by commas and a split function would need to be used in order to split these rows up and put them into variables for each column. In some cases though, there may have been commas in movie titles or other columns depending on the data and so we would need to count how many commas there were and reconnect certain pieces after the split. Additionally,

some of these files, such as the movie finances files, were text files which were tab delimited. In this case, we would need to use the split function on tabs rather than commas to ensure that these were split up correctly.

After the splits were correctly achieved, some of the variables we had taken were already completely good to go while others needed some work done to them. For example, if there was a movie title, this would need to be uppercased and converted to UTF-8 in order to normalize the titles throughout the database and ensure that everything looked the same. For others such as the budget, the dollar signs and commas would need to be removed from those. There were also certain cases where there may have been no data in a particular column and because the database needs to see these as null, we would need to turn these into “None”. There may have also been dates where the year may have needed to be extracted from the dd/mm/yyyy format. After this was completed, the parse function would return the values and a map function would create an RDD based on these mappings. Because the map function will see these key/pair values as two item tuples, if there were more than two variables returned we would need to enclose the second and onward variables in parentheses so that these would be an n-sized tuple as the second item in the tuple.

For the Movielens ratings data, several ratings may exist for the same title and we wanted to get the average rating for each title. To do this, we would use the mapValues() function on the RDD and create a tuple with the rating as the first item and a count of one as the second item. We would then need to use the reduceByKey() function on a method that was created that would group together all the title_id’s and for the value of that key we would add up all of the ratings as the first value in that tuple with the number of occurrences for that title as the second value. We would then use mapValues() on another method which would divide the total rating by the number of occurrences and replace the value to that key with the average rating

for that title. Another file which contained the links that connect the Movielens id to the imdb id was then joined to the previous RDD. Another map function was then used to create a prefix for all of these id's by adding the "tt" prefix plus however many 0's made it the length of every other imdb title_id. This ensured that the id for that title would be the same datatype as the table we were adding it to. Now that all of our data had been correctly manipulated, we could then use the forEachPartition() function with the RDD to run the save to db method discussed in the Data Load section to load our data into our tables.

The same approach to adding the imdb prefixes was used for our movie tags data that was to be added to our Title Tags table. We also used the same parsing and joining approaches for all of our other data as previously discussed. Although we ran into various roadblocks with the complexities that were discussed, we were eventually able to use the information from our crash logs as well as help from the professor and TAs to figure out the source of our bugs. We would also frequently need to use the truncate table SQL command in psql if something in our PySpark script went wrong so that it wouldn't be adding the same data over again.

2.5 Data Analysis (Queries)

After correctly designing our schema and loading all of the data onto our tables, we would write queries as select statements to select certain data based on parameters that we chose. The queries generally were based off of interesting questions that we felt someone might actually be curious about. For example, which TV shows have aired more than 100 episodes before? That's a pretty important accomplishment for a show and an interesting bit of information that someone may be curious to know.

Our queries were usually created by joining two or more tables together using an inner join. Sometimes, we would use outer joins, except those would usually contain semi-trivial pieces of information because we had realized that it was a bit more difficult to come up with an interesting query that was based off of an outer join that couldn't just be accomplished by an inner join. This usually meant that our outer join based queries were questions about people who were one thing but not something else, such as a director who has never been a writer. We could then use an outer join in this situation to see where they might have a null for the writer column.

Our queries would also frequently contain a "where" clause when we wanted to figure out if a certain variable had some sort of property. If we wanted to see if a title's release year or a person's birth day were in a certain range we could use the "between" clause. We also frequently used aggregate queries where we could use a "group by" clause and have our query give us a count for something. In the 100 episodes query that was mentioned earlier, we could group by the title of the TV show and if that count was greater than or equal to 100 then it would be selected. The "order by" clause was also frequently used to order our data in a way that made sense to us. If you wanted to see how many TV shows had more than 100 episodes, you would probably want to see the greatest number of episodes first, not last.

We would frequently run into issues where a query would output the data wrong because we made a query that did not match up with what we wanted. By running the query in psql, we could take a look at the data and see if the output is what we wanted and if not we could look back at the query and see what went wrong. Throughout the semester we got better and better at writing queries because we could familiarize ourselves with the syntax more.

2.6 Visualization

For visualizing our queries, we used Amazon Quicksight. This platform was fairly easy to use and worked quite reliably. Our only complaints were related to the relatively low control over how the data from our queries would be displayed and the very unintuitive sharing process.

Although there were several options for the type of graph to fit the data to, there weren't any settings for the types except for a slider that controlled the range of values to display. This was frustrating when there were a couple of times that Quicksight didn't give us back correct-looking graphs, and so we were unsure of whether our query was formulated incorrectly or Quicksight just interpreted the data incorrectly. If there had been more specific settings on displaying the data, maybe we could have gotten the data to display properly, or we could have at least confirmed the state of our queries. Regardless, new queries had to be formulated until we got ones that ended up looking correct on Quicksight.

When it came to share our data visualizations, we had to first create a dashboard, then share the dashboard with the Admin user, and then finally take a screenshot of the visualization. This process was repeated for each visualization, and for that we greatly wished that Amazon had some kind of group visualization feature where we could create a dashboard that held several of the visualizations at once. Furthermore, we wished that Amazon provided a feature of link generation where a link to the visualization could be automatically created which would have enabled us to share the visualization more easily.

3. Conclusion

Although we sometimes felt little freedom when being guided by instructions for a majority of the time, the last milestone that involved modifying our schema allowed us to tackle the extract, transform, and load process heads-on without step-by-step instructions. This final part felt like a true and fitting last challenge, and so it was very rewarding to see that we had learned enough throughout the semester to be able to complete the given task with grace.

Working our way through this final project has given us invaluable first-hand experience to the trials and tribulations that come with combining several datasets from different sources and running complex analyzations on them. Likewise, using a variety of tools, like AWS, Lucidchart, PySpark, and Git, has provided us a broad foundation for database implementation and manipulation that we will almost certainly use in our post-undergraduate careers.