CS 327E Final Project: Milestone 6

Jin Won Choi JWC2728

Jiajun Bao jb65634

Introduction

Goals

The goal of Elements of Databases CS327E is to understand and learn database and big data technologies whilst focusing on designing and building databases and data pipelines. Database was explored through peer learning as well as teaching guidance from the professor and projects/labs were completed with partners using Github to compile and store data to an initially existing repository.

Tools

Most of the tools used in this class was built around Amazon Web Services (AWS) including the following interfaces: SQL, Presto, Spark, Python, Postgres, QuickSight. The data sets that were provided were loaded in s3 buckets and downloaded via URL to be uploaded to our tables in the database. The Entity Data Relationships Diagram was constructed using the LucidChart services that were available to use whilst the data dictionary was constructed using Excel sheets. The queries were run and quality checked using SQL platforms like Athena and PSQL databases while visualizations were created using Amazon QuickSight. Github was used to compile and combine files for the projects and labs.

Data Sets

For projects and learning use, each group was given 5 main data set group folders - IMDB, Instacart, Movielens, Numbers with headers, and cinemalytics – to analyze and upload to database and integrate using various methods. IMDB was the dataset that was used for all of the labs and consisted of mainly movie title data and the people related to the certain titles in the movie industry including these relations: directors, person_basics, profession_basics, principals, stars, title_basics, title_episodes, title_genre, title_ratings, and writers. Instacart was used to code and learn in class and it consisted of mainly of online orders, products, aisles, departments, and order_products. Next data set provided was, movielens_ratings containing 7 different files most being CSV data files mainly dealing with ratings/scores of the movies and links. Then came numbers-with-headers data set with 36 text files that had the records of the movie titles and tags. The final data set was cinemalytics data set that contained 5 CSV files persons, singer_songs, songs, title_songs, and title.

Analysis

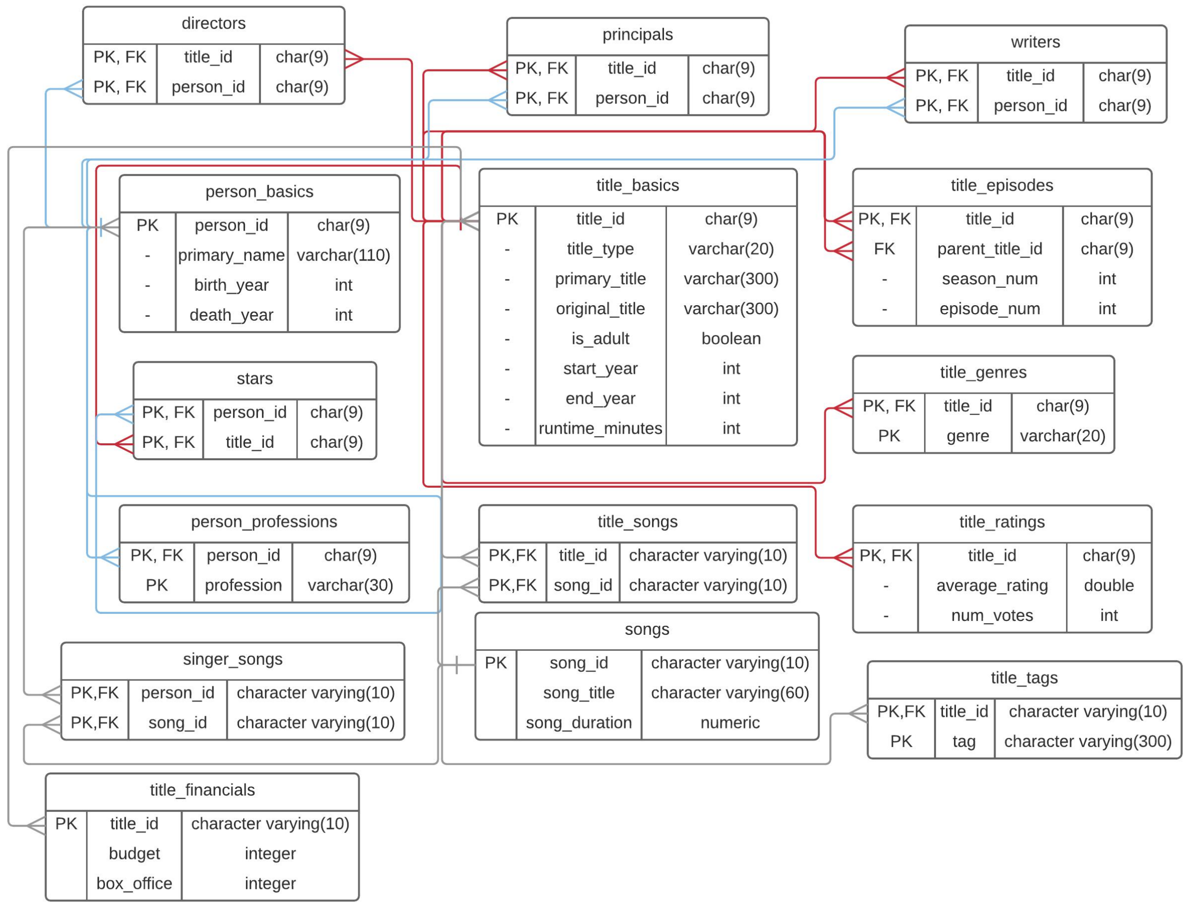
Setup

The projects were initially setup in a Github repository that had been created by the instructor and all our finished product of files would be placed in the folder and be turned in. Setting up the project and splitting up the work for each of the partners to work on was a tough challenge because the programing parts of the queries and the designing of the charts had to all be done together for any and all of the labs/projects. The charts were set up using the functions provided in the Lucidcharts and were designed by looking at the relations and references to all the tables. The tables were the one part of the assignments that were actually easy splitting up work for since it had individual tables and once they were done, all that had to be done was connecting the primary and foreign reference keys to all the tables. The last few labs dealing with the Amazon cluster was the worst when splitting work load since all the data had to be run on one cluster unless each were constantly sending the updated code to each other. Which meant that we couldn’t edit things in the cluster and Spark without expecting to get different entries in the tables and having to restart the whole process by dropping and re-creating the tables. So the set up had to be very precise and we had to make sure that we had set up the code perfectly before doing any ETL or uploading via UPDATE-ing the tables.

Data load

Data loading process was an easy but a time-consuming and confusing process for most of the students who have never used a database before. Data loading process had to be done once the files have been downloaded to a host uploading computer and once the tables were created. Throughout the labs, we created and dropped multiple tables due to constraints and field errors. Some of the attributes had to be declared with different parameters even after the tables were created and the loading processes had begun because some of the fields were throwing errors due to unavailable space in sizes of the dedicated fields. Some data were rejected and timed out because the foreign key constraints that were created before the data was uploaded and threw many error saying that the duplicate primary keys were already in existence. This could be overcome by removing the foreign key constraints from the relations then reloading the data to the database then altering the tables by recreating the foreign key constraints for proper setup. This meant that we had to rerun the copy commands several times due to the unexpected upload data errors which was a pain in everyone's behind because that meant another 3 hours we had to spend uploading a data set after having had spent over 12 hours waiting on the terminal to announce, "timed out." However, we were able to gather the data and upload properly despite all of the errors that we faced. Based on the experience we have of loading tons of data, we believe that it is usually the best that we create the foreign keys(reference) after uploaded the data. Doing so not only save us lots of time, but also give us the chance to deal with conflicts after we successfully uploaded the data.

Uploading the data usualy includes data integration that we will talk about in next few sections since the data we obtain is not alway in the way what want them to be presented in tables.


directors
PK, FK | title_id | char(9)
PK, FK | person_id | char(9)
principals
PK, FK | title_id | char(9)
PK, FK | person_id | char(9)
writers
PK, FK | title_id | char(9)
PK, FK | person_id | char(9)
person_basics
PK | person_id | char(9)
- | primary_name | varchar(110)
- | birth_year | int
- | death_year | int
title_basics
PK | title_id | char(9)
- | title_type | varchar(20)
- | primary_title | varchar(300)
- | original_title | varchar(300)
- | is_adult | boolean
- | start_year | int
- | end_year | int
- | runtime_minutes | int
title_episodes
PK, FK | title_id | char(9)
FK | parent_title_id | char(9)
- | season_num | int
- | episode_num | int
stars
PK, FK | person_id | char(9)
PK, FK | title_id | char(9)
title_genres
PK, FK | title_id | char(9)
PK | genre | varchar(20)
person_professions
PK, FK | person_id | char(9)
PK | profession | varchar(30)
title_songs
PK,FK | title_id | character varying(10)
PK,FK | song_id | character varying(10)
title_ratings
PK, FK | title_id | char(9)
- | average_rating | double
- | num_votes | int
songs
PK | song_id | character varying(10)
song_title | character varying(60)
song_duration | numeric
singer_songs
PK,FK | person_id | character varying(10)
PK,FK | song_id | character varying(10)
title_tags
PK,FK | title_id | character varying(10)
PK | tag | character varying(300)
title_financials
PK | title_id | character varying(10)
budget | integer
box_office | integer

Database design

The database was designed under the influence of the data sets that were given and later with additional data sets. The first set of tables that we received were the IMDB data sets which had many different CSV files that related to a table set up and we created tables according to the data provided by matching the primary keys to foreign keys and also their references. This also allowed us to design and compile the Entity Data Relationship diagram using Lucidchart by creating the relationship descriptions as one to many or one to one or many to many. The designing of the ERD helped in applying queries and joining tables to gather the proper tables and fields necessary to compute the interesting queries. This also helped in creating data constraints in the proper steps since it showed how the upload process was going to be interrupted in case of duplicates due to primary and foreign keys. The main uploading problems were usually caused by foreign keys, so by looking at the database design through the ERD, we were able to steer away from struggling with uploading the data and having to restart the process due to unnecessary interruptions. The tables were also created by looking at the fields of all the attributes and constraints that were needed to keep all the relations correct with each other on the pipelines where primary and foreign keys set the one to many or many to many relations.

Data Integration

Sometimes we cannot just use the raw data that was given. For example in milestone 4, we want person_id in table singer_songs reference to the person_id in person_basics. However, when we actually tried to copy the data, we found out that some of the person_id in sinder_songs do not have a match to the person_id in person_basics. As a result, we had to delete the extra rows that are in singer_songs before we do the reference. we used "extract()" to only get the year from the raw data that has year-month-date, and we inserted data from two tables into title_songs to make it the way we want. Data integration is similar to data analysis since both of them are used to process and select the data we want. However, data integration is the method we use to organize our tables and records in them, but data analysis focus on selecting records from organized tables. Theoretically, we can use Psql to map all the datas, but using Spark can actually make the process a bit easiler. By creating a cluster and the Spark, we are able to connect different databases and run scripts directly. I think that it is a easiler way when we have lots of data integration to deal with comparing to Psql. However it is a little bit complicated to set up everything and giving the access.

Data Analysis(Queries)

Writing queries are the most interesting part of the database. Queries can be used to process datas in some tables that we select and give us the results we want to know. Queries usually include a select statement since we definitely want some results back. We usually use functions like count(), max(), sum() to get the datas we want and * means that we are selecting every row. We can also use distinct() to remove duplicates. The most common and simply querry we use is “select count(*) from” to see how many rows we have in selected table so that we can check if we imported the data correctly.

If we want get information by combining two tables together, we can use join method. “Join” or “inner join” only give the rows that both table have the same keys, and “full outer join” includes all the rows in both tables. For example, when we did “person_basics pb join writers w on pb.person_id=w.person_id” to combine two tables only on the rows that have same person_id. If we used “left join”, we should be able to get more data since we would be getting all the rows from person_basics even we cannot find a match. Though all the labs we did, I do realize that join is actually a very powerful tool since we are usually asked to get records cross tables. we need to actually think about the kind of join that we want to use. We need to have a picture of how the combined table will look like before we select data. There were so many cases that we used the wrong join so that we got results that are a little bit off. Sometimes it is really hard to realize that we used wrong join comparing to other mistakes that can give us an error message imediately when we did something wrong.

"Group by" is also a very interesting method we could use to group the rows with same sepcified record together so that we can know how many are them. For example, we did "group by primary_name" after we join title_basics and writers to see how many work each writers have done.

Based on our experience of working with all the queries, we notice that there is a order of organizing a query that we should use for most of our queries so that we can avoid lots of errors:

Select (records we want) as(name we want) from(table)

Join(table) on...

Where (condition)

Group by...

Having...

Order by...

Creating index and materialized views are very useful ways to speed up the queries when some queries take too long to run. We have to realize that when we create indexes and materialized views are basicly trading space for efficiency. Materialized view is basic to take a picture of the querry result and save it so that we do not spend extra time when we want to see the result again. One thing really important that I learned is that we have to update the views when we change the datas. Also, if the queries take 20 hours to run, you should expect that the materialized view will take 20 hours to create. By creating index, we are sorting the datas we have so that it would be much faster to search for certain record. It is important to determine the records that we create index of. When we created an index on (upper(primary_title),start_year), it makes the data mapping on Spark much faster. We tested many different ways of creating index using “explain” and “\timing”, and we finally got the correct one.

Data visualization

Data visualization is a way of offering the customer the graphic view of the data distribution that they want. By doing data visualization though the QuickSight, we have to make sure that we give QuickSight the access to our RDS and the regions are same. During the lab3 when I was trying to create a graph over the actors, I was not able to find anything under RDS because my QuickSight is on a different region. After the setup iss done, we need to create correct queries under a vew or materialized view if the queries take more than 10 seconds. By selecting the view in QuickSight, we can get a optimized graph of the data we get or we can visualize the data the way we want with tools in Quickight. Based on all the graphs I did, the optimized graph is usually the optimized way to present the datas. Those graphs can usually give us a better understanding of the data distribution. It is usually the final step of our assignments since we need to have a compeleted databases to do the visualization. QuickSight is a tool that can vissualize the datas, and it does not contain any data by itself.

Conclusion

We first learned how to use github as very powerful tool to share information, and set up a RDS as our first database instance. Then, by creating tables and RED graph, we learned how to build a organized database on RDS based only on the raw datas given. We have to be able to tell the relationship between records and data types to actually build tables correctly and avoid conflicts.

Then we studied the method to upload the table, which could get very complicated if we need to integrate the data. We can either use SQL to correctly mapping the data. Based on the size of the data, sometimes we have to set up the foreign keys after uploaded the data.

After we have the correct tables with records we want, we can do queries that can be run using Psql, and we can create views and index so that we can create graphs on QuickSight efficiently. By doing queries and views, we get to actually analysis the datas we have and get results from the analysis. It is the final goal of creating a database.

Finally, we tried to use the Spark to do more difficult mapping and analysis with Python script. With cluster, we are actually able to deal with multiple databases together.