Final Project Report

Team Dadabass
Yujie Hou, Lucy Huang
Professor: Shirley Cohen

The University of Texas at Austin
Fall 2017

Chapter 1: Introduction

For the CS327E Elements of Databases class, the dadabass team created a database warehouse of cinema information. The team created ETL pipelines, designed queries to pull relevant information, and visualized results using several tools. This project showcases what the team learned from this class: SQL, relational database design, and utilizing ETL pipeline tools.

Chapter 2: Labs

2.1 Lab 1

2.1.1 Set Up

For the first lab of this class, the team installed a local git client, set up AWS accounts, and gained exposure to LucidChart. The team also set up a Github repository. The professor and TA's added team dadabass to the class's organization in Github in order to view progress and see individual contributions. Dadabass used git to share code and also enable version control. Dadabass used the "issues" feature on Github to identify problems which required collaboration from the other partner or a blocker that they temporarily could not overcome. Dadabass made AWS accounts, using education credits to fund the use of the technologies.

One of the most important parts of setting up AWS included setting up the IAM accounts. This would ensure that database information remained secure and so that personal credit cards would not be overcharged. IAM user credentials were shared, containing a special URL, using Stache which added another layer of security. Stache

used the UT Austin network to authenticate logins and ensure that only UT-associated users could access the IAM information.

The team also used Athena, a serverless querying service. Dadabass was able to easily load tables into S3 and use SQL to gather information in order to create an entity-relationship diagram (ERD). Athena is also part of AWS. Dadabass was given special education credits for the use of Athena as well.

To capture the schema of the data, dadabass used LucidChart. LucidChart is a diagramming and visualization app that allows its users to easily collaborate on projects. Dadabass mainly utilized the “Entity Relationship” library from the “Software” category of LucidChart’s library.

2.1.2 Data

The professor provided access to an imdb dataset which contained information about movies, television, and associated people such as director and principals. Dadabass created tables, such as directors, in Athena that referenced the data in the S3 buckets. Dadabass chose data types that matched, taking into account differences in syntax. For example, Athena refers to varchar and char as strings. Upon generating and populating the table, Athena also outputted DDL that it had used to create the external table.

2.1.3 Analysis

Since the imdb dataset was going to be used throughout the course of the semester, dadabass created 20 queries that would touch each table of the schema. This would allow dadabass to gain familiarity with the data elements and identify any interesting

trends in the data. To view the queries that dadabass created, please refer to Attachment A in the appendix.

2.1.4 Entity Relationship Diagram

To create the ERD in LucidChart, dadabass considered column names, primary keys and foreign keys. Primary keys were identified by using SQL code to check the following constraints: 1) If the column had a value for each row in the table since each row in a normalized table would have a primary key and 2) If each row in the column was a unique value since primary keys are unique for each element.

Special considerations were also made for concatenated primary keys, in which a table has at least two primary keys. Dadabass ran similar tests to count for whether concatenated primary keys existed by selecting the two columns, checking that counts matched and checking for unique values. To map out foreign keys, dadabass had to consider whether all the specified rows in a child table could be found in the parent table as well. This would ensure that there would be no orphan rows in the child table and satisfy the foreign key constraint. Please refer to Appendix B for a diagram of the ERD created.

2.2 Lab 2

2.2.1 Set Up

Dadabass set up an RDS instance in AWS. Dadabass also used PostgreSQL, an open source relational database system, and psql, a management tool for PostgreSQL.

Dadabass connected to the Postgres RDS instance from psql and began to set up the imdb schema using DDL.

Dadabass also had to keep in mind to whitelist IPs in order to access the database. This required IAM user credentials as well.

2.2.2 Table Creation

Dadabass used DDL to create the tables and properly map foreign keys. When mapping foreign keys, dadabass had to first create the parent table and then create the child table. This would ensure that the child table could properly reference the parent table.

2.2.3 Loading Tables

After creating the tables, dadabass downloaded the imdb data from the Amazon S3 bucket onto their laptops. Dadabass used the “copy” command in psql to load the data for each file into their respective table. Dadabass was able to customize the copy command according to the file type, adjusting for whether or not the CSV had a header, consider delimiters, and encode in UTF-8 as well. However there were complications for loading millions of records for each table. Dadabass was throttled by UT Austin’s wifi and experienced very slow loading. This can also be considered a downside to using the psql copy command when populating tables. However, it is also very straightforward and simple to execute which is also a positive.

2.2.4 Dimensional Schema

Dadabass also created a dimensional schema for the imdb leadership team to easily review ratings for titles. Dadabass created the dimensional schema in LucidChart. It is notably different from a regular ERD due to its denormalization. The dimensional schema for analyzing the ratings over time is denormalized because it contains redundant information that often is not related to the primary key. However, since the leadership team wants to analyze the change in ratings over time, a dimensional schema makes it easier for business analysts to examine the data.

2.3 Lab 3

2.3.1 Tools

Dadabass used Quicksight from AWS to visualize aggregate views created. Quicksight is a business analytics software that offers interactive and different forms of data visualization. It is easily customizable and can be securely shared between users.

2.3.2 Creating Views

After designing the SQL aggregate queries, dadabass created views. Views can be considered as virtual tables, containing results from a query which a user can query again and even join with other tables. Views are stored as objects in a database. Views can also be useful when you would like to store results without creating a new table or typing out the entire query again. Views also refresh and return the latest data to the user.

Dadabass created QuickSight visualizations in AWS from the views of aggregate queries. Using the “timer” function in psql, dadabass was able to measure the runtime of the aggregate query. Any query that took over 10 seconds turned into a materialized view. Any query that was under 10 seconds was stored as a regular view. Materialized views differ from regular views because they are stored as tables in the schema and are saved to disk. They must also be refreshed manually.

Chapter 3: Final Milestone Project

The final project in this class introduced us to Apache Spark which we used to perform ETL on the imdb and related dataset. While we were provided with a complete template in the first milestone, we transitioned to writing our own functions and eventually commanding Spark to perform specific transformations for MapReduce.

3.1 Milestone 1

For M1, we were concerned about the ratings of media that were stored in the Movielens database. In *ratings.csv*, we found many ratings that users had given for a movie at a certain time. The same user could have rated many different movies, so there was a concatenated primary key constraint on the user and movie id.

Reading through the template file, we identified that the challenge was combining the data from two sources together. IMDB and Movielens gave different identifiers to the same media, and our job was to add the ratings of the Movielens ids to the corresponding IMDB movies. Because this was the first time we had used Spark, we worked on understanding how each line of code affected the RDDs. We realized that the code was very formulaic in its approach to manipulating the dataset. We read in two

files, *ratings* and *links*, grabbed the appropriate fields by mapping them in key-value pairs, and returned them as an RDD. These two RDDs would then be joined on a common key-- the Movielens identifier. Afterwards, it would extract from tuples and remap the data that we cared about from that RDD: the IMDB id and the ratings. Finally, the Spark file connected to our RDS instance and executed the UPDATE query.

Our view for this query looked at the number of titles per genre that had a higher movielens rating than from imdb (see Appendix B.1). Looking at the Quicksight graphic, we were surprised at how poorly viewers on movielens rated media compared to on imdb. Only around 100 titles scored higher on movielens in total, with the most being in the “Action” genre. Because these genres may be overlapping, the number of higher-rated movies shrinks to even less.

Milestone 1 gave the team the most trouble overall, specifically with regards to the setup of the EMR cluster. Because our RDS instance was located in Ohio, we struggled to deploy our cluster. The cluster needed to be in a region that had the m3.xlarge master node type, which Ohio lacked. We considered two options-- (1) copy our RDS instance to Virginia or (2) continue the process on the Ohio server and implement the m4.xlarge fix. Ultimately, we decided that moving the database would be more work than it was worth, and the instructor had mentioned a possible error with IP that we did not want to run into. However, the m4 type fix did not work because there was another error with locating the setup script in the S3 bucket. Although it should not have been an issue, our cluster being setup in a different region than the bucket’s location caused a significant delay. To fix this, the instructor setup a new bucket in our region.

3.2 Milestone 2

The files for Milestone 2 came from from the same dataset as from M1. Instead of looking at the ratings, we focused on the tags that users on Movielens had given to digital media they watched. Our template file for this milestone contained some gaps in the code that we were expected to fill in. However, the process of loading the *tags.csv* file, mapping the tags to the title identifiers, and joining that RDD with the link data was the same as the previous milestone, so we borrowed and modified the code accordingly.

The most interesting aspect of this milestone came at the very end with our resulting view. This view grabbed the most popular tags at the turn of the 21st century, specifically in the year 2000. Appendix B.2 shows that the most popular tags during this year were “nudity (topless)” along with “drama”, “romance”, and “DVD”. Since these tags aren’t mutually exclusive, it made sense to assume that there was a proliferation of erotic scenes in romantic and dramatic films, which is a statement that still holds true today. What also puzzled us during this part was that many of the tags didn’t make much sense. For example, “not watched” and “watched” seemed useless. If a user had tagged a movie, we assumed that they had watched the movie.

3.3 Milestone 3

For Milestone 3, our goal was to load the financial information of movies such as their budget and box office revenue into the IMDB database. The first major task was parsing the .csv files in our Spark script. We did not have many issues during this phase of the milestone because we created a separate test script for the parsing function that would

print out to command prompt. By incrementally adding the rules of each field, we were able to ensure that all of the columns were properly parsed.

The most difficult part of M3 was the logic loops for the second phase. Because we only wanted to retrieve a single title_id for each row of financial data, we had to check for many conditions. If we didn't have a single row returned from the first SELECT query on primary_title and start_year only, we first checked if the box office was positive to make sure we were not grabbing tv episodes. Otherwise, we ran another query to make sure the genre also matched. When our query finally resulted in a single title_id being returned, we could finally insert the title_id and respective financial data into the title_financials table.

To speed up query times, dadabass also tested and implemented indexes (See Appendix B.3). To measure the impact of indexes, dadabass used the "timer" functionality in psql and also utilized the "explain" command to see if a query was using the index.

For a query that searched through the imdb dataset for a title's start year and the titile's primary title, dadabass found that that adding an index for start_year greatly improved the runtime, reducing runtime by around 16 seconds. Dadabas then removed the start_year index and independently tested the same query after adding a primary_title index. This reduced the runtime even more, resulting in less than a second to search through the table. As a result, dadabass added the start_year index back in and also kept the primary_title index to keep runtimes at a high speed.

Dadabass also created a partial index to determine whether a title was a TV episode or not. However, the partial index only reduced runtime by around 2 seconds, so dadabass

decided to remove the index in consideration of other tradeoffs. For example, including an index would mean that extra time would be added to inserting, altering, and deleting records from the table.

Lastly, dadabass tested if including a genres index would improve the runtime of a query that parsed through genres. Ultimately, this increased the runtime of the query by around half a second. This was probably due to the few unique genres in comparison to the millions of unique titles. Dadabass removed the genres index.

3.4 Milestone 4

M4 brought us to the Cinemalytics data source of Bollywood songs, titles, and singers. As we were given free reign, we needed to weigh the pros and cons of using Spark vs psql for different parts of the assignment. The benefits of using Spark are that it allows us to do transformations of the data in-place in the code, and the drawbacks are that the functions may be a bit difficult to write. Psql is great for loading large amounts of data with the “copy” command, but it is not very good at organizing or modifying the data because many step-by-step queries would have to be executed.

After weighing the pros and cons of each method, we decided to use Spark to load Title_Songs and Singer_Songs and perform the INSERT on Person_Basics. Psql would then be used to clean up the Title_Songs and Singer_Songs tables in order to enforce the foreign key constraints. It would also be used to simply copy everything from the Songs data source into a corresponding table. The most notable ETL was performed in Spark on Title_Songs and Person_Basics. Like before, we needed to match the Cinemalytics title identifiers with the corresponding IMDB identifiers, and a join() did the job perfectly, returning a key-value pair of title_id and song_id. Person_Basics required

a different approach, as we needed to inner join persons and singer_songs on person_id and get the distinct list of person_ids to be added with their respective biographical information.

We ran into two substantial issues in M4. Firstly, our original DELETE statement was crashing psql when we ran it because we attempted to select every row in Person_Basics. By changing it to a RIGHT OUTER JOIN, we were able to optimize the statement and save a lot of time. Additionally, we were able to see a real application of the advantage of using joins vs. select *. The second issue was that too many duplicate person_ids were being inserted into Person_Basics. We fixed this issue by moving the distinct() from before we joined persons and singer_songs to afterwards. By placing the distinct() afterwards, we got rid of all duplicate person_id entries in the joined RDD.

Chapter 4: Conclusion

Overall, dadabass learned much about relational databases and utilizing industry-standard technologies such as PostgreSQL, AWS, and git. We were given detailed instructions on how to approach the labs and milestones, so this enabled us to quickly execute what we learned. However some of the set-up and commands were time-consuming and took up more time than expected, especially when populating tables. We also ran into surprises with the data, encountering obstacles such as UTF-8 encoding and decoding in addition to properly measuring the varchar lengths of title names. Using the RDD printlines() and psql tools such as the timer and explain functions, we were able to debug or optimize our work, which allows for a sense of independence in a classroom environment and gives us the confidence to continue to use these skills in the future.

Appendixes:

Appendix A:

1)SQL queries for Lab 1

```
--Query 1: list the stars who were in a movie that had the same original and primary title in order
SELECT DISTINCT person_id FROM stars a INNER JOIN title_basics b ON a.title_id = b.title_id WHERE primary_title=original_title ORDER BY person_id;

--Query 2: list the number of distinct stars
SELECT COUNT(DISTINCT person_id) FROM stars;

--Query 3: list the number of episodes that are in their second season
SELECT COUNT(title_id) FROM title_episodes WHERE season_num = 2;

--Query 4: list the unique parent titles that have episodes with an average rating greater than 7.0 in order
SELECT DISTINCT parent_title_id FROM title_episodes a INNER JOIN title_ratings b ON a.title_id=b.title_id WHERE average_rating > 7.0 ORDER BY parent_title_id;

--Query 5: list titles that classify as Dramas in order
SELECT title_id FROM title_genres WHERE genre = 'Drama' ORDER BY title_id;

--Query 6: list episodes of adult video that are in their second season in order
SELECT a.title_id FROM title_episodes a INNER JOIN title_basics b ON a.title_id = b.title_id WHERE a.season_num = 2 AND b.is_adult = TRUE ORDER BY a.title_id;

--Query 7: list the unique parent titles that have episodes with an average rating less than 3.0 in order
SELECT DISTINCT parent_title_id FROM title_episodes a INNER JOIN title_ratings b ON a.title_id=b.title_id WHERE average_rating < 3.0 ORDER BY parent_title_id;

--Query 8: list the number of unique parent titles that have episodes with more than 300 votes
SELECT COUNT(DISTINCT parent_title_id) FROM title_episodes a INNER JOIN title_ratings b ON a.title_id = b.title_id WHERE num_votes > 300;

--Query 9: return the runtime of the longest drama
SELECT MAX(runtime_minutes) FROM title_genres a INNER JOIN title_basics b ON a.title_id = b.title_id WHERE genre = 'Drama';
```

--Query 10: return the runtime of the shortest drama
SELECT MAX(runtime_minutes) FROM title_genres a INNER JOIN title_basics b ON a.title_id = b.title_id WHERE genre = 'Drama';

-- Query 11: List stars that were born between 1990 and 2000.
select distinct primary_name from stars inner join person_basics on stars.person_id = person_basics.person_id where birth_year between 1990 and 2000;

--- Query 12: List people that were both stars and directors.
select distinct primary_name from person_basics inner join directors on person_basics.person_id = directors.person_id inner join writers on person_basics.person_id = writers.person_id;

-- Query 13: List what genres that Jack Weiss starred in.
select title_genres.genre from principals inner join title_genres on principals.title_id = title_genres.title_id inner join person_basics on principals.person_id = person_basics.person_id where person_basics.primary_name = 'Jack Weiss';

--Query 14: List what people are involved in the costume profession.
select distinct primary_name from person_professions inner join person_basics on person_professions.person_id = person_basics.person_id where person_professions.profession like '%costume%' or person_professions.profession like '%Costume%';

-- Query 15: List the average rating of all the movies that starred Natalie Portman.
select avg(average_rating) from title_ratings inner join stars on title_ratings.title_id = stars.title_id inner join person_basics on stars.person_id= person_basics.person_id where person_basics.primary_name like 'Natalie Portman';

-- Query 16: List the birth year of all directors ordered from oldest (deceased) to youngest.
select distinct birth_year from person_basics inner join directors on person_basics.person_id = directors.person_id order by birth_year;

-- Query 17: Find the longest movie that Natalie Portman has ever been in.
select title_basics.primary_title from stars inner join person_basics on stars.person_id = person_basics.person_id inner join title_basics on stars.title_id = title_basics.title_id

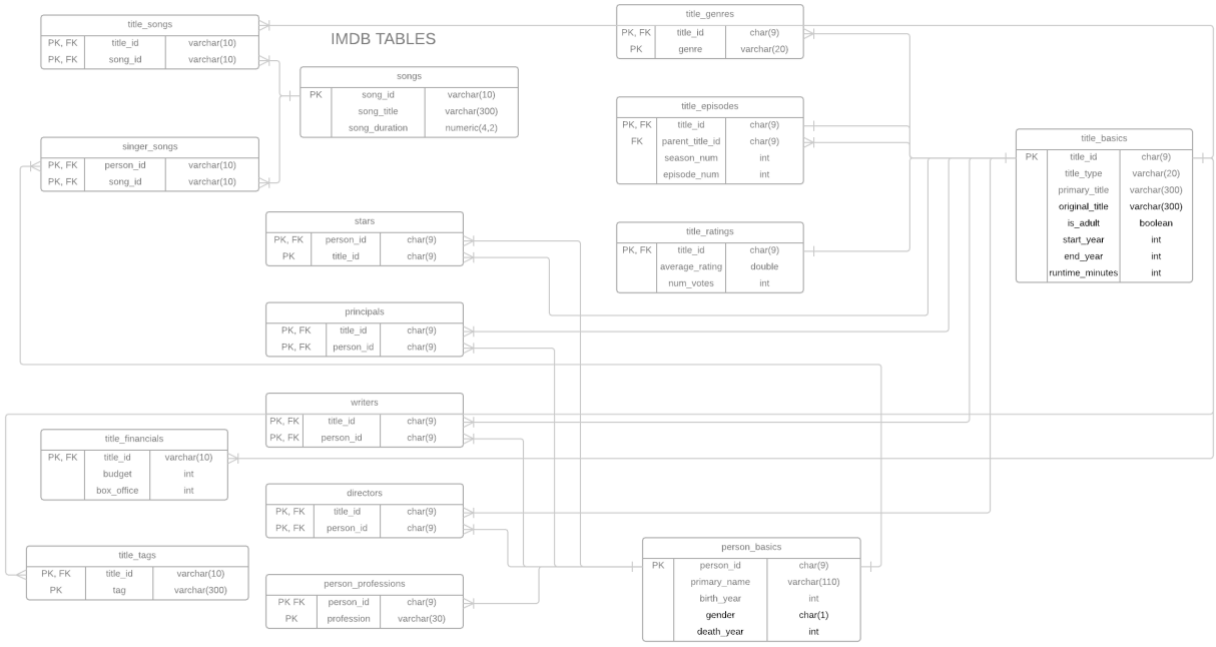
where person_basics.primary_name like 'Natalie Portman' order by runtime_minutes desc limit 1;

-- Query 18: List the writers of movies, based on movies with the highest to lowest rating. Movies with the same rating should be listed in alphabetical order.
select primary_name, primary_title, average_rating from person_basics inner join writers on person_basics.person_id = writers.person_id inner join title_basics on writers.title_id = title_basics.title_id inner join title_ratings on writers.title_id = title_ratings.title_id order by title_ratings.average_rating desc, primary_title;

-- Query 19: List deceased stars ordered by how long they have been dead.
select distinct primary_name, death_year from person_basics inner join stars on person_basics.person_id = stars.person_id where death_year is not null order by death_year;

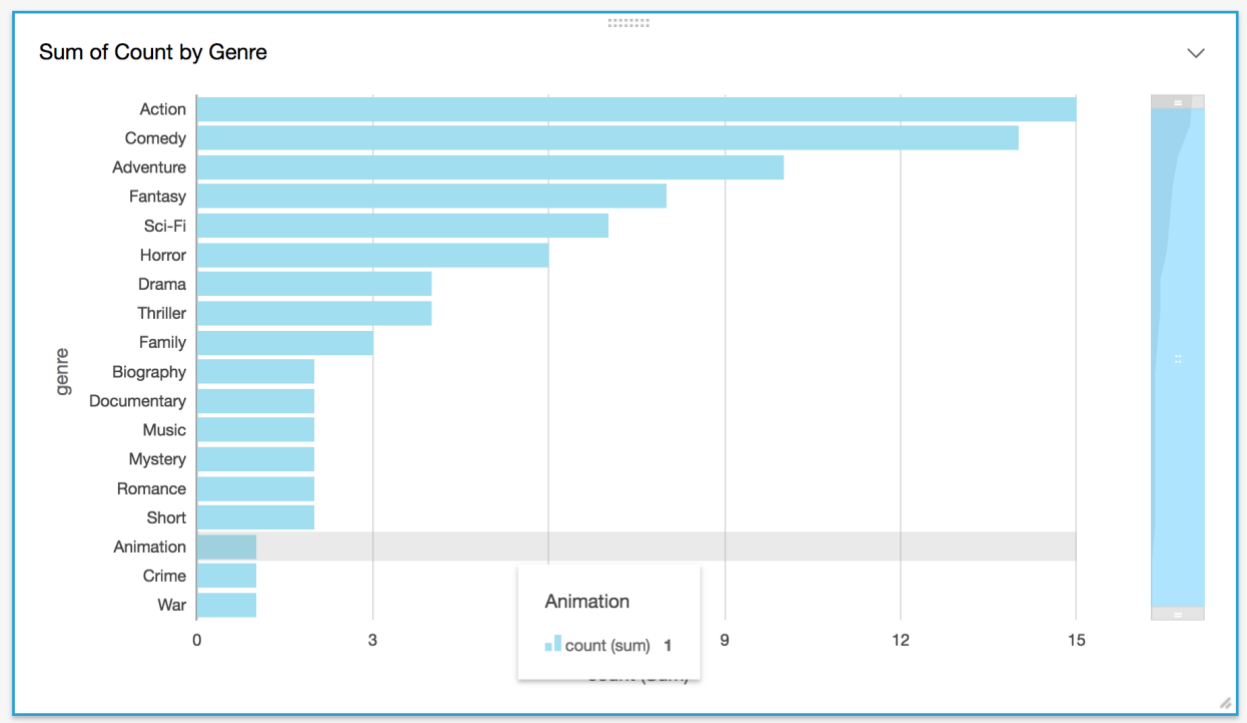
-- Query 20: List directors of top 10 longest adult movies.
select distinct primary_name, runtime_minutes from directors inner join person_basics on directors.person_id = person_basics.person_id inner join title_basics on directors.title_id = title_basics.title_id where is_adult = true order by runtime_minutes desc limit 10;

2) ERD of imdb created in LucidChart

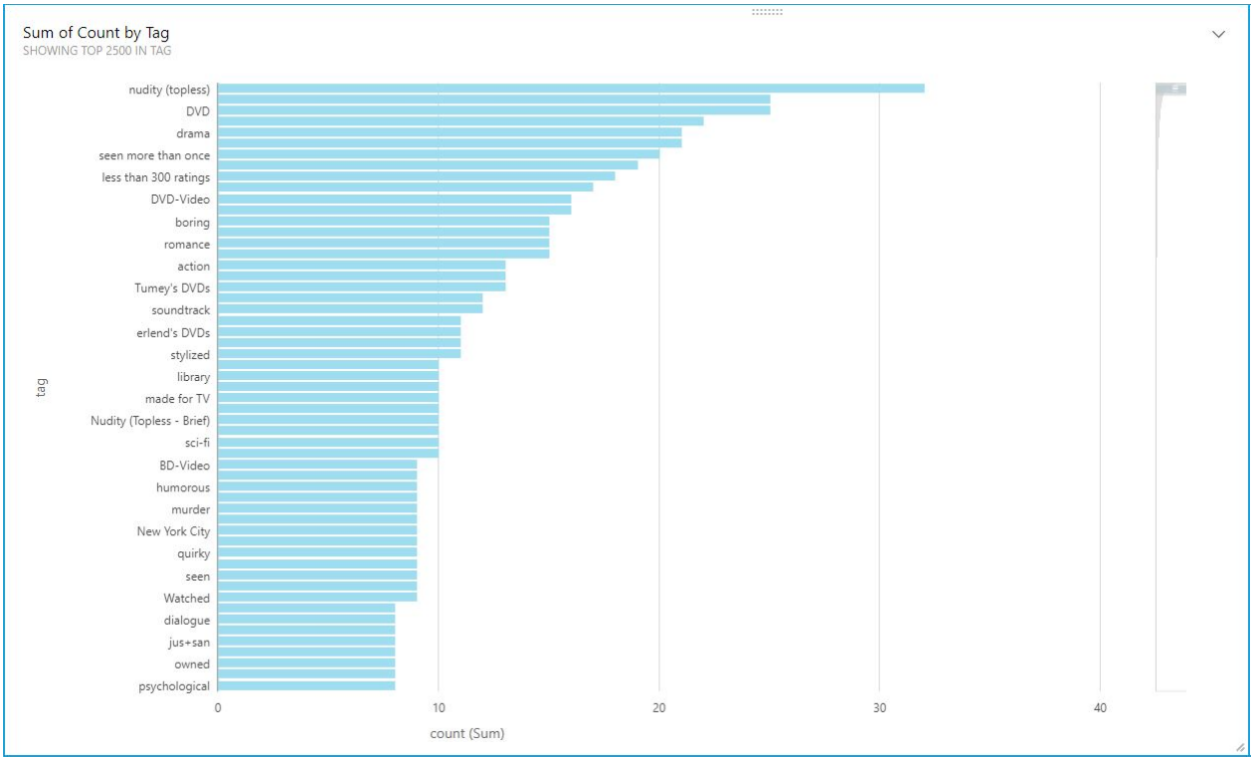


Appendix B

1: Movielens_ratings view



2: Movielens_tags view



3: Milestone 3 indexes

Index	Time	Result
None	18039.431 ms	Adding indexes reduced our our query time, so we decided to ***keep the indexes***.
start_year	2421.515 ms	
primary_title	61.506 ms	
None	16954.22 ms	The partial index added little value, so ***we ultimately removed it***.
non_tv_eps (partial)	14688.003 ms	
None	15192.6555 ms	There are few unique genres, so our ***search time increased***.
genres	15677.239 ms	