Design, Load, and Explore a Movies Database

Perla Molina
Data Science Major
CS 333 Intro to Database Systems
Fall 2021

CHAPTER 1: PROJECT DESCRIPTION

a) Goal of the Project

The goal of this project is to practice the full circle of exploring a dataset. Exploring a dataset includes: designing, building, and using a database that reflects the description of the given dataset. The general steps can include but are not limited to: drawing an E/R diagram for the design of the database in order to perceive a visual of the database; writing SQL code in order to test, explore, query, and optimize the database; and summarizing the steps of the work with conclusions in order to report the entirety of the project. There are three phases to this project: 1) Design the database, 2) Build the database, 3) Use the database.

b) Data Exploration

The database is titled 'movies' with input files: movies.txt, size 490 KB; ratings.txt, size 229,596 KB; README.html, size 12 KB, and tags.txt, size 3,221 KB. 'Movies' has attributes: MovieID, Title, and Genres. 'Ratings' has attributes: UserID, MovieID, Rating, and Timestamp. And lastly, 'Tags' has attributes: UserID, MovieID, Tag, and Timestamp.

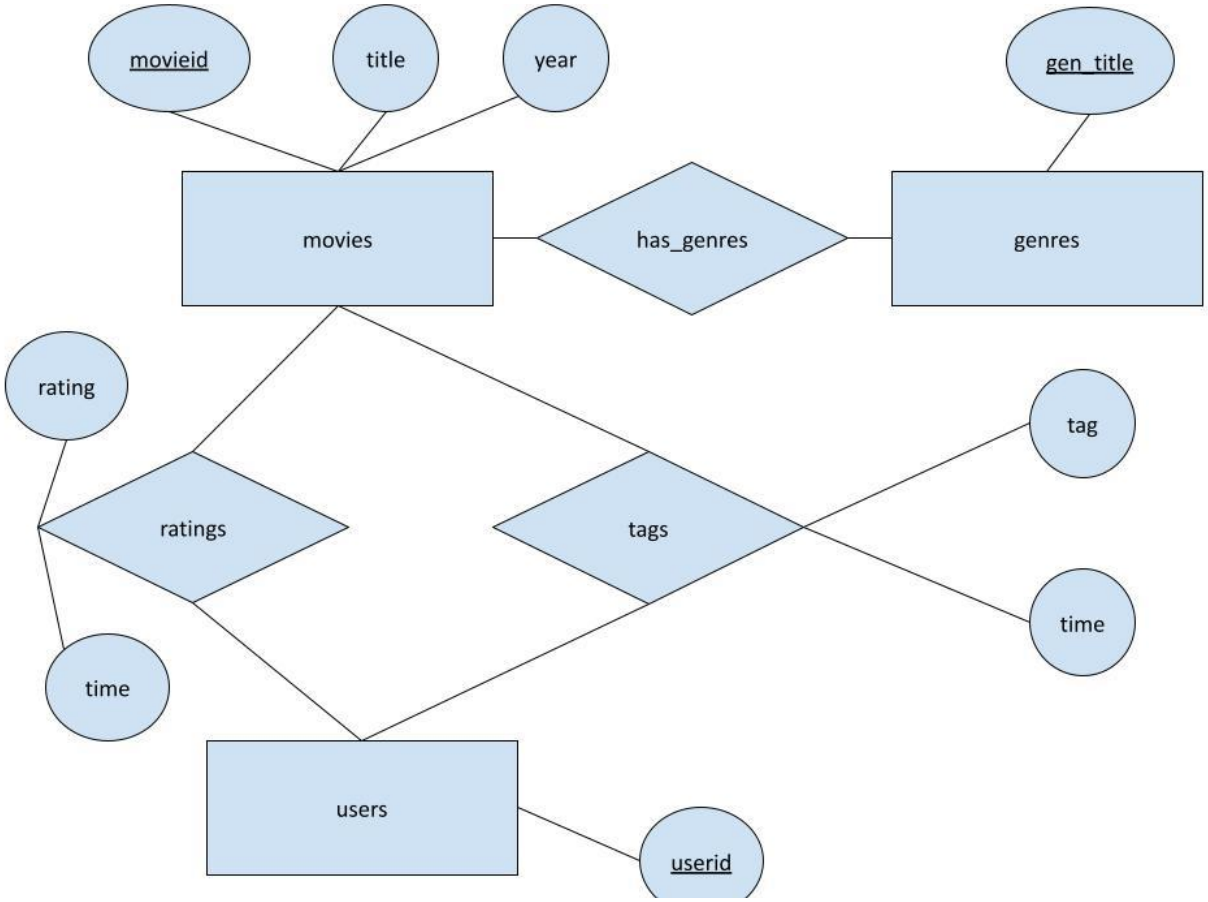
CHAPTER 2: DATABASE DESIGN

a) E/R Diagram

There are three entity sets: movies, genres, and users with three relationship sets: tags, ratings, and has_genres. The first entity set, 'movies' connects to the relationship sets 'ratings' and 'tags' because they all contain the attribute key of 'movieid'. The second entity set, 'genres' connects to the relationship set 'has_genres' because they both contain the attribute key of 'movieid'. The third entity set, 'users' also connects to the relationship sets 'ratings' and 'tags' because they all contain the attribute key of 'userid'.

The attributes of this database include: 'userid', which is the unique identification number of the user that either rated or tagged the movie; 'movieid', which is the unique and real MovieLens identification number of the movie; 'title', which by policy, should be entered identically to those found in IMDB; 'year' for the year the movie title was released just as shown in IMDB as well (however, both 'title' and 'year' have been entered manually from the original dataset, so errors and inconsistencies may exist); 'gen_title', which is all the genres that came from the Movies.txt file in a pipe-separated list, and are selected from the following: Action, Adventure, Animation, Children's, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir, Horror, Musical, Mystery, Romance, Sci-Fi, Thriller, War, Western; 'rating', which is made on a 5-star scale with half-star increments, given by a user; 'tag', which is a user generated metadata about movies, each typically a single word or short phrase, determined by each user; and lastly, 'time', which is a timestamp represented in seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

Here is the E/R diagram which I have come up with:



b) Logical Schema

Here is a list of the logical schemas for the entity sets from the E/R diagram.

- movies(movieid, title, year)
- genres(gen_title)
- users(userid)

Here is a list of the logical schemas for the relationship sets from the E/R diagram.

- ratings(userid, movieid, rating, time)
- tags(userid, movieid, tag, time)
- has_genre(movieid, gen_title)

Entity set ‘movies’ has attributes: movieid - the primary key, title, and year. Entity set ‘genres’ has only one attribute, which is ‘gen_title’ that is also the primary key. Entity set ‘users’ also contains only one attribute, which is ‘userid’ that is also its primary key. Relationship set ‘ratings’ has attributes: userid - the primary key, movieid - also the primary key, rating, and time. Relationship set ‘tags’ has attributes: userid - the primary key, movieid - also the primary key, tag, and time. The relationship set ‘has_genre’ has attributes movieid and gen_title, which are both the primary keys.

CHAPTER 3: LOAD DATA AND TEST THE DATABASE

a) Load Data

Now we are going to code SQL queries to load the data into a database and test it.

i) Create A New Database:

First, we create a new database titled, 'moviesdb' in the terminal.

```
createdb -U postgres moviesdb
```

After it has been created, it needs to be accessed in order to start loading the data.

```
psql -U postgres moviesdb
```

ii) Create the Tables From the Logical Schema

Now we are ready to start creating the tables of the schema and loading the data. They do not need to be created in this exact order nor all at once because loading the data from the .txt files into the relationship tables will sometimes require an extra step of dropping an entity set table using the **DROP TABLE** SQL command in order to properly load the data from these files into the table without errors. (After loading the data, we can re-create that previously dropped entity set table to continue loading data for it.) We first create the tables for the entity sets, movies, genres, and users.

For the 'movies' table, this is the SQL query that will be used to create it.

```
CREATE TABLE movies (
movieid INTEGER PRIMARY KEY,
title VARCHAR(50),
year INTEGER
);
```

For the 'genres' table, this is the SQL query that will be used to create it.

```
CREATE TABLE genres (
gen_title VARCHAR(50) PRIMARY KEY
);
```

For the 'users' table, this is the SQL query that will be used to create it.

```
CREATE TABLE users (
userid INTEGER PRIMARY KEY
);
```

Now to create the relationship tables, it'll be slightly different from creating the entity set tables as we will need to be referencing primary key attributes from them. Similar to before, these tables do not need to be created in this exact order. However, there is no need to drop these relationship tables when loading the data.

To create the 'ratings' table, we will need to use this SQL query.

```
CREATE TABLE ratings (
userid INTEGER,
movieid INTEGER,
```

```
rating REAL,
time NUMERIC,
PRIMARY KEY (userid, movieid),
FOREIGN KEY (userid) REFERENCES users(userid),
FOREIGN KEY (movieid) REFERENCES movies(movieid)
);
```

To create the ‘tags’ table, we will need to use this SQL query.

```
CREATE TABLE tags (
userid INTEGER,
movieid INTEGER,
tag VARCHAR(1000),
time NUMERIC,
PRIMARY KEY (userid, movieid),
FOREIGN KEY (userid) REFERENCES users(userid),
FOREIGN KEY (movieid) REFERENCES movies(id)
);
```

To create the ‘has_genre’ table, we will need to use this SQL query.

```
CREATE TABLE has_genre (
movieid INTEGER,
gen_title VARCHAR(50),
PRIMARY KEY (movieid, gen_title),
FOREIGN KEY (movieid) REFERENCES movies(movieid),
FOREIGN KEY (gen_title) REFERENCES genres(gen_title)
);
```

iii) Load the Data From Input Files

Now that we have our tables ready, we can start loading the data into them. Before doing so, it’s important to make sure all files of this database are in the same folder (such as ‘project files’) where we will query from.

From all the tables, only 'ratings' is ready to be loaded from its corresponding .txt file right away with the **COPY** SQL command.

```
\COPY ratings(userid, movieid, rating, time) FROM
'project files/ratings.txt' WITH DELIMITER ':';
```

From this command, there should be 10000054 rows copied into table ‘ratings’. If an error pops up, dropping the tables ‘users’ and ‘movies’ might need to be done first and those tables will need to be re-created after ‘ratings’ has been loaded, otherwise, we can move forward with the rest.

For table ‘tags’, I used a python script that edited its corresponding text file and appended its data into a new tags.csv file. The purpose of this step is to help with loading the data using the **COPY** command because certain tags may contain the ':' symbol in their 'Tag' field. However, the ':' symbol is used as a delimiter in that file, too. Thus, the text file needs to

be edited where we'll have a row show like this: UserID::MovieID::Tag::Timestamp. This is so the **COPY** command can skip ':' symbols found in the 'Tag' field. Once that is cleared, we can load the data into the 'tags' table with the command.

```
\COPY tags(userid, movieid, tag, time) FROM 'project files/tags.csv' WITH DELIMITER ':';
```

From this command, there should be 95580 rows copied into table 'tags'. If an error pops up, dropping the tables 'users' and 'movies' might need to be done first and those tables will need to be re-created after 'tags' have been loaded, otherwise, we can move forward with the rest.

For table 'users', there's no need to write an external python script to load the data from a file. To populate the data, we do so by reading the unique users found in tables 'ratings' and 'tags'. To properly do this without complicated queries or commands, I first dropped the table 'users'.

```
DROP TABLE users CASCADE;
```

Then I used this SQL query to populate users from table 'ratings' into a table called 'users2'.

```
CREATE TABLE users2 AS SELECT DISTINCT userid FROM ratings;
```

From this query, 69878 users should be copied into table 'users2'. After this, I inserted unique users from table 'tags' into 'users2'.

```
INSERT INTO users2
SELECT DISTINCT userid FROM tags;
```

From this query, 4009 new users should be added to table 'users2'. From this, there will be duplicates in 'users2', which is why distinct users will be populated from this table into our actual 'users' table that will be used in this database.

```
CREATE TABLE users AS SELECT DISTINCT userid FROM users2;
```

Now there should be a total of 71567 unique users in table 'users' without any duplicates whatsoever. Now we are free to delete or drop the table 'users2'.

For table 'genres', I simply created a new text file title 'genres.txt' and manually typed all 20 genres into each line, including '(no genres listed)'. Then the text file was used to easily populate the data into its corresponding table using the **COPY** command.

```
\COPY genres(gen_title) FROM 'project files/genres.txt';
```

For table 'movies', I used a python script that edited its corresponding text file and appended its data into a new cleanedMovies.txt file. The purpose of this step is to help with

loading only the relevant data, which is 'movieid', 'title', and 'year' from the original text file that contained extra information and structure that is not needed for this table. I also decided to use the '%' symbol as a separator in the new cleanedMovies.txt file to make loading easier and visually less confusing. But before using the **COPY** command to load the data, I needed to set the encoding using this command '**SET CLIENT_ENCODING TO 'utf8';**' to enable Postgres to process data with accents or other language writings in movie titles.

```
SET CLIENT_ENCODING TO 'utf8';
\COPY movies(movieid, title, year) FROM 'project files/cleanedMovies.txt' WITH DELIMITER '%';
```

From these commands, there should be 10681 rows copied into table 'movies'. Make sure to run the **SET** command line first before running the **COPY** command, otherwise, an encoding error will occur.

Lastly, for table 'has_genre', I also had to use another python script that processed the necessary information from the original movies.txt file, which is 'movieid' and 'gen_title'. The '%' symbol was again used as a separator to make loading easier and visually less confusing. Once, that has been processed, the **COPY** command will be used again to load the data.

```
\COPY has_genre(movieid, gen_title) FROM 'project files/has_genres.txt' WITH DELIMITER '%';
```

After the data is loaded, there should be 21564 rows copied into the 'has_genre' table. And now we are done with loading the data into all the tables.

b) Test the Database

We move on to test our database. In order to verify that the data has been successfully loaded, we must run certain SQL queries to ensure that the database is up and ready to use for analysis.

A) First, we list our tables using the **\d** function.

```
\d
```

```
moviesdb=# \d
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | genres    | table | postgres
 public | has_genre | table | postgres
 public | movies    | table | postgres
 public | ratings   | table | postgres
 public | tags      | table | postgres
 public | users     | table | postgres
(6 rows)
```

B) Then, we list the data types of each table.

`\d genres`

```
moviesdb=# \d genres
                  Table "public.genres"
  Column   |         Type          | Collation | Nullable | Default
-----------+-----------------------+-----------+----------+---------
 gen_title | character varying(50) |           | not null |
Indexes:
    "genres_pkey" PRIMARY KEY, btree (gen_title)
Referenced by:
    TABLE "has_genre" CONSTRAINT "has_genre_gen_title_fkey" FOREIGN KEY (gen_title) REFERENCES genres(gen_title)
```

`\d has_genre`

```
moviesdb=# \d has_genre
                   Table "public.has_genre"
  Column   |         Type          | Collation | Nullable | Default
-----------+-----------------------+-----------+----------+---------
 movieid   | integer               |           | not null |
 gen_title | character varying(50) |           | not null |
Indexes:
    "has_genre_pkey" PRIMARY KEY, btree (movieid, gen_title)
Foreign-key constraints:
    "has_genre_gen_title_fkey" FOREIGN KEY (gen_title) REFERENCES genres(gen_title)
```

`\d movies`

```
moviesdb=# \d movies
                     Table "public.movies"
 Column  |          Type           | Collation | Nullable | Default
---------+-------------------------+-----------+----------+---------
 movieid | integer                 |           | not null |
 title   | character varying(1000) |           |          |
 year    | integer                 |           |          |
Indexes:
    "movies_pkey" PRIMARY KEY, btree (movieid)
```

`\d ratings`

```
moviesdb=# \d ratings
                  Table "public.ratings"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 userid  | integer |           | not null |
 movieid | integer |           | not null |
 rating  | real    |           |          |
 time    | numeric |           |          |
Indexes:
    "ratings_pkey" PRIMARY KEY, btree (userid, movieid)
```

```
\d tags
```

```
moviesdb=# \d tags
                    Table "public.tags"
 Column  |          Type           | Collation | Nullable | Default
---------+-------------------------+-----------+----------+---------
 userid  | integer                 |           |          |
 movieid | integer                 |           |          |
 tag     | character varying(1000) |           |          |
 time    | numeric                 |           |          |
```

```
\d users
```

```
moviesdb=# \d users
                Table "public.users"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 userid | integer |           |          |
```

C) We must also list the size of our tables. In other words, we will list how many rows are in each table.

```
SELECT COUNT(*)
FROM genres;
```

```
moviesdb=# SELECT COUNT(*)
moviesdb-# FROM genres;
 count
-------
    20
(1 row)
```

```
SELECT COUNT(*)
FROM has_genre;
```

```
moviesdb=# SELECT COUNT(*)
moviesdb-# FROM has_genre;
 count
-------
 21564
(1 row)
```

```
SELECT COUNT(*)
FROM movies;
```

```
moviesdb=# SELECT COUNT(*)
moviesdb-# FROM movies;
 count
-------
 10681
(1 row)
```

```
SELECT COUNT(*)
FROM ratings;
```

```
moviesdb=# SELECT COUNT(*)
moviesdb-# FROM ratings;
  count
----------
 10000054
(1 row)
```

```
SELECT COUNT(*)
FROM tags;
```

```
moviesdb=# SELECT COUNT(*)
moviesdb-# FROM tags;
 count
-------
 95580
(1 row)
```

```
SELECT COUNT(*)
```

```
FROM users;
```

```
moviesdb=# SELECT COUNT(*)
moviesdb-# FROM users;
 count
-------
 71567
(1 row)
```

D) Now we'll look at some data values. We'll take a look at the first 5 lines of the 'movies' table to make sure it looks alright.

```
SELECT *
FROM movies
LIMIT 5;
```

```
moviesdb=# SELECT *
moviesdb-# FROM movies
moviesdb-# LIMIT 5;
 movieid |            title            | year
---------+-----------------------------+------
       1 | Toy Story                   | 1995
       2 | Jumanji                     | 1995
       3 | Grumpier Old Men            | 1995
       4 | Waiting to Exhale           | 1995
       5 | Father of the Bride Part II | 1995
(5 rows)
```

Let's perform a query that checks for the number of non **NULL** titles to ensure it is the same number as the size of the table from the previous query involving the **COUNT()** function.

```
SELECT COUNT(title)
FROM movies;
```

```
moviesdb=# SELECT COUNT(title)
moviesdb-# FROM movies;
 count
-------
 10681
(1 row)
```

Now let's look at the last 5 lines of the ‘movies’ table as well to ensure it also looks good.

```
SELECT *
FROM movies
ORDER BY year
DESC LIMIT 5;
```

```
moviesdb=# SELECT *
moviesdb-# FROM movies
moviesdb-# ORDER BY year
moviesdb-# DESC LIMIT 5;
 movieid |                    title                    | year
---------+---------------------------------------------+------
   55830 | Be Kind Rewind                              | 2008
   56949 | 27 Dresses                                  | 2008
   53207 | 88 Minutes                                  | 2008
   55603 | My Mom's New Boyfriend                      | 2008
   57326 | In the Name of the King: A Dungeon Siege Tale | 2008
(5 rows)
```

We will also test some simple sorting by the attribute ‘year’ to make sure there isn’t anything strange.

```
SELECT *
FROM movies
ORDER BY year
LIMIT 5;
```

```
moviesdb=# SELECT *
moviesdb-# FROM movies
moviesdb-# ORDER BY year
moviesdb-# LIMIT 5;
 movieid |            title             | year
---------+------------------------------+------
    7065 | Birth of a Nation, The       | 1915
    7243 | Intolerance                  | 1916
   62383 | 20,000 Leagues Under the Sea | 1916
   48374 | Father Sergius (Otets Sergiy) | 1917
    8511 | Immigrant, The               | 1917
(5 rows)
```

We can also check for **NULL** values for the attribute 'year' in the table 'movies'. There should be zero **NULL** values.

```
SELECT COUNT(year)
FROM movies;
```

```
moviesdb=# SELECT COUNT(year)
moviesdb-# FROM movies;
 count
-------
 10681
(1 row)
```

We can also check for any rows where 'year = 0' to ensure there are no **NULL** values.

```
SELECT COUNT(year)
FROM movies;
WHERE year = 0;
```

```
moviesdb=# SELECT COUNT(year)
moviesdb-# FROM movies
moviesdb-# WHERE year = 0;
 count
-------
     0
(1 row)
```

Another way to check for **NULL** values is to check for rows where 'year > 1500'.

```
SELECT COUNT(year)
FROM movies
WHERE year > 1500;
```

```
moviesdb=# SELECT COUNT(year)
moviesdb-# FROM movies
moviesdb-# WHERE year > 1500;
 count
-------
 10681
(1 row)
```

We can also test the cases where there is no genre associated with a movie (no genres listed case).

```
SELECT *
FROM has_genre
WHERE gen_title = '(no genres listed)';
```

```
moviesdb=# SELECT *
moviesdb-# FROM has_genre
moviesdb-# WHERE gen_title = '(no genres listed)';
 movieid |     gen_title
---------+-------------------
    8606 | (no genres listed)
(1 row)
```

E) Similarly, we will continue to try to find any unknown or invalid data in any of the other attributes for the other tables.

```
SELECT COUNT(rating)
FROM ratings
WHERE rating = 0 OR rating < 0;
```

```
moviesdb=# SELECT COUNT(rating)
moviesdb-# FROM ratings
moviesdb-# WHERE rating = 0 OR rating < 0;
 count
-------
     0
(1 row)
```

```
SELECT *
FROM ratings
WHERE rating < 1
LIMIT 5;
```

```
moviesdb=# SELECT *
moviesdb-# FROM ratings
moviesdb-# WHERE rating < 1
moviesdb-# LIMIT 5;
 userid | movieid | rating |    time
--------+---------+--------+------------
      8 |     590 |    0.5 | 1116547591
      8 |    1035 |    0.5 | 1115860115
      8 |    2378 |    0.5 | 1115858710
      8 |    2380 |    0.5 | 1115859173
      8 |    2383 |    0.5 | 1115859158
(5 rows)
```

```
SELECT COUNT(userid)
FROM ratings
WHERE userid < 0;
```

```
moviesdb=# SELECT COUNT(userid)
moviesdb-# FROM ratings
moviesdb-# WHERE userid < 0;
 count
-------
     0
(1 row)
```

```
SELECT COUNT(movieid)
FROM ratings
WHERE movieid < 0;
```

```
moviesdb=# SELECT COUNT(movieid)
moviesdb-# FROM ratings
moviesdb-# WHERE movieid < 0;
 count
-------
     0
(1 row)
```

```
SELECT COUNT(time)
FROM ratings
WHERE time < 0;
```

```
moviesdb=# SELECT COUNT(time)
moviesdb-# FROM ratings
moviesdb-# WHERE time < 0;
 count
-------
     0
(1 row)
```

```
SELECT *
FROM tags
LIMIT 5;
```

```
moviesdb=# SELECT *
moviesdb-# FROM tags
moviesdb-# LIMIT 5;
 userid | movieid |       tag       |    time
--------+---------+-----------------+------------
     15 |    4973 | excellent!      | 1215184630
     20 |    1747 | politics        | 1188263867
     20 |    1747 | satire          | 1188263867
     20 |    2424 | chick flick 212 | 1188263835
     20 |    2424 | hanks           | 1188263835
(5 rows)
```

```
SELECT COUNT(userid)
FROM tags
WHERE userid <= 0;
```

```
moviesdb=# SELECT COUNT(userid)
moviesdb-# FROM tags
moviesdb-# WHERE userid <= 0;
 count
-------
     0
(1 row)
```

```
SELECT COUNT(movieid)
FROM tags
WHERE movieid <= 0;
```

```
moviesdb=# SELECT COUNT(movieid)
moviesdb-# FROM tags
moviesdb-# WHERE movieid <= 0;
 count
-------
     0
(1 row)
```

```
SELECT COUNT(tag)
FROM tags
```

```
WHERE tag IS NULL;
```

```
moviesdb=# SELECT COUNT(tag)
moviesdb-# FROM tags
moviesdb-# WHERE tag IS NULL;
 count
-------
     0
(1 row)
```

```
SELECT COUNT(time)
FROM tags
WHERE time <= 0;
```

```
moviesdb=# SELECT COUNT(time)
moviesdb-# FROM tags
moviesdb-# WHERE time <= 0;
 count
-------
     0
(1 row)
```

F) Now we will find the distribution of the values for the attribute 'year of table 'movies'.

```
SELECT year, COUNT(*)
FROM movies
GROUP BY year
ORDER BY year;
```

```
moviesdb=# SELECT year, COUNT(*)
moviesdb-# FROM movies
moviesdb-# GROUP BY year
moviesdb-# ORDER BY year;
 year | count
------+-------
 1915 |     1
 1916 |     2
 1917 |     2
 1918 |     2
 1919 |     4
 1920 |     5
 1921 |     3
 1922 |     7
 1923 |     6
 1924 |     6
 1925 |    10
 1926 |    10
 1927 |    19
 1928 |    10
 1929 |     7
 1930 |    15
 1931 |    16
 1932 |    22
 1933 |    23
 1934 |    18
 1935 |    18
 1936 |    32
 1937 |    30
 1938 |    19
 1939 |    37
 1940 |    40
 1941 |    28
 1942 |    38
 1943 |    40
 1944 |    37
 1945 |    36
 1946 |    38
 1947 |    39
 1948 |    46
 1949 |    37
 1950 |    44
 1951 |    44
 1952 |    40
 1953 |    55
 1954 |    43
 1955 |    57
 1956 |    53
 1957 |    62
 1958 |    62
 1959 |    61
 1960 |    66
 1961 |    57
 1962 |    69
 1963 |    63
 1964 |    72
 1965 |    72
 1966 |    87
 1967 |    68
 1968 |    72
 1969 |    64
 1970 |    71
 1971 |    73
 1972 |    83
 1973 |    81
 1974 |    75
 1975 |    74
 1976 |    75
 1977 |    83
 1978 |    82
 1979 |    87
 1980 |   161
 1981 |   178
 1982 |   170
 1983 |   111
 1984 |   137
 1985 |   158
 1986 |   166
 1987 |   205
 1988 |   214
 1989 |   212
 1990 |   200
 1991 |   188
 1992 |   212
 1993 |   258
 1994 |   307
 1995 |   362
 1996 |   384
 1997 |   370
 1998 |   384
 1999 |   357
 2000 |   405
 2001 |   403
 2002 |   441
 2003 |   366
 2004 |   342
 2005 |   332
 2006 |   345
 2007 |   364
 2008 |   251
(94 rows)
```

G) Now let's find the distribution of the movies across different decades using the query below.

```
SELECT decade, COUNT(*)
FROM
(
SELECT case when year between 1910 and 1919 then
'1910'
when year between 1920 and 1929 then '1920'
when year between 1930 and 1939 then '1930'
when year between 1940 and 1949 then '1940'
when year between 1950 and 1959 then '1950'
when year between 1960 and 1969 then '1960'
when year between 1970 and 1979 then '1970'
when year between 1980 and 1989 then '1980'
when year between 1990 and 1999 then '1990'
when year between 2000 and 2009 then '2000'
end AS decade
FROM movies
t
GROUP BY decade
ORDER BY decade ASC;
```

```
moviesdb=# SELECT decade, COUNT(*)
moviesdb-# FROM
moviesdb-# (
moviesdb(# SELECT case when year between 1910 and 1919 then '1910'
moviesdb(# when year between 1920 and 1929 then '1920'
moviesdb(# when year between 1930 and 1939 then '1930'
moviesdb(# when year between 1940 and 1949 then '1940'
moviesdb(# when year between 1950 and 1959 then '1950'
moviesdb(# when year between 1960 and 1969 then '1960'
moviesdb(# when year between 1970 and 1979 then '1970'
moviesdb(# when year between 1980 and 1989 then '1980'
moviesdb(# when year between 1990 and 1999 then '1990'
moviesdb(# when year between 2000 and 2009 then '2000'
moviesdb(# end AS decade
moviesdb(# FROM movies
moviesdb(# ) t
moviesdb-# GROUP BY decade
moviesdb-# ORDER BY decade ASC;
 decade | count
--------+-------
 1910   |    11
 1920   |    83
 1930   |   230
 1940   |   379
 1950   |   521
 1960   |   690
 1970   |   784
 1980   |  1712
 1990   |  3022
 2000   |  3249
(10 rows)
```

H) Now we'll find the distribution of the genres across the movies.

```
SELECT gen_title, COUNT(*)
FROM has_genre
GROUP BY gen_title;
```

```
moviesdb=# SELECT gen_title, COUNT(*)
moviesdb-# FROM has_genre
moviesdb-# GROUP BY gen_title;
     gen_title      | count
--------------------+-------
 IMAX               |    29
 Crime              |  1118
 Animation          |   286
 Documentary        |   482
 Romance            |  1685
 Mystery            |   509
 Children           |   528
 Musical            |   436
 Film-Noir          |   148
 Fantasy            |   543
 Horror             |  1013
 Drama              |  5339
 Action             |  1473
 (no genres listed) |     1
 Thriller           |  1706
 Western            |   275
 Sci-Fi             |   754
 Comedy             |  3703
 Adventure          |  1025
 War                |   511
(20 rows)
```

I) We will also find the distribution of the ratings values.

```
SELECT rating, COUNT(*)
FROM ratings
```

```
GROUP BY rating;
```

```
moviesdb=# SELECT rating, COUNT(*)
moviesdb-# FROM ratings
moviesdb-# GROUP BY rating;
 rating |  count
--------+---------
    0.5 |   94988
      1 |  384180
    1.5 |  118278
      2 |  790306
    2.5 |  370178
      3 | 2356676
    3.5 |  879764
      4 | 2875850
    4.5 |  585022
      5 | 1544812
(10 rows)
```

J)

i) Now let's find how many movies have no tags, but have ratings.

```
SELECT CAST (count(movieid) AS INTEGER) FROM movies
WHERE movieid
NOT IN (SELECT DISTINCT movieid FROM tags)
AND movieid IN (SELECT DISTINCT movieid FROM
ratings);
```

```
moviesdb=# SELECT CAST(count(movieid) AS INTEGER) FROM movies
moviesdb-# WHERE movieid
moviesdb-# NOT IN (SELECT DISTINCT movieid FROM tags)
moviesdb-# AND movieid IN (SELECT DISTINCT movieid FROM ratings);
 count
-------
  3080
(1 row)
```

ii) Let's also find how many movies have no ratings, but have tags.

```
SELECT CAST(count(movieid) AS INTEGER) FROM movies
WHERE movieid
IN (SELECT DISTINCT movieid FROM tags WHERE movieid
NOT IN (SELECT DISTINCT movieid FROM ratings));
```

```
moviesdb=# SELECT CAST(count(movieid) AS INTEGER) FROM movies
moviesdb-# WHERE movieid
moviesdb-# IN (SELECT DISTINCT movieid FROM tags WHERE movieid NOT IN (SELECT DISTINCT movieid FROM ratings));
 count
-------
     4
(1 row)
```

iii) We can also find how many movies have neither tags nor ratings.

```
SELECT CAST(count(movieid) AS INTEGER) FROM movies
WHERE movieid
NOT IN (SELECT DISTINCT movieid FROM ratings) AND
movieid NOT IN (SELECT DISTINCT movieid FROM tags);
```

```
moviesdb=# SELECT CAST(count(movieid) AS INTEGER) FROM movies
moviesdb-# WHERE movieid
moviesdb-# NOT IN (SELECT DISTINCT movieid FROM ratings) AND movieid NOT IN (SELECT DISTINCT movieid FROM tags);
 count
-------
     0
(1 row)
```

iv) And lastly, we shall find how many movies have both tags and ratings.

```
SELECT CAST(count(movieid) AS INTEGER) FROM movies
WHERE movieid IN ((SELECT DISTINCT movieid FROM
ratings) INTERSECT (SELECT DISTINCT movieid FROM
tags));
```

```
moviesdb=# SELECT CAST(count(movieid) AS INTEGER) FROM movies
moviesdb-# WHERE movieid IN ((SELECT DISTINCT movieid FROM ratings) INTERSECT (SELECT DISTINCT movieid FROM tags));
 count
-------
  7597
(1 row)
```

We expect that the results from i, ii, iii, and iv will add up to the result of **`COUNT(movieid)`** from the 'movies' table. 3080 + 4 + 0 + 7597 = 10,681 which is correct.

CHAPTER 4: QUERY THE DATABASE AND OPTIMIZE THE QUERIES

Now that we are done with testing out the database, we can move forward with performing analytical queries and optimization.

a) General Queries

1) Find the most reviewed movie, (that is, the movie with the highest number of reviews). Show the movie id, movie title, and the number of reviews.

In order to perform this request, we will need to **`JOIN`** two tables and use multiple aggregate functions. Before executing the SQL query, I created an index on the attribute 'movieid' from the 'ratings' table because of the fact that there is an extensive amount of rows in the 'ratings' table. It also cut the running time by nearly 15,000 milliseconds.

```
CREATE INDEX idx_movieid ON ratings(movieid);
SELECT m.movieid, title, count_ratings
FROM movies AS m
JOIN (
SELECT COUNT(*) AS count_ratings, movieid
FROM ratings
```

```
GROUP BY movieid
ORDER BY count_ratings
DESC LIMIT 1) AS t
ON m.movieid = t.movieid;
```

```
moviesdb=# CREATE INDEX idx_movieid ON ratings(movieid);
CREATE INDEX
moviesdb=# SELECT m.movieid, title, count_ratings
moviesdb-# FROM movies AS m
moviesdb-# JOIN (
moviesdb(# SELECT COUNT(*) AS count_ratings, movieid
moviesdb(# FROM ratings
moviesdb(# GROUP BY movieid
moviesdb(# ORDER BY count_ratings
moviesdb(# DESC LIMIT 1) AS t
moviesdb-# ON m.movieid = t.movieid;
 movieid |    title     | count_ratings
---------+--------------+---------------
     296 | Pulp Fiction |         34864
(1 row)
```

2) Find the highest reviewed movie (the movie with the most 5-star reviews). Show the movie id, movie title, and the number of reviews.

Similar to the previous query, we will need to **JOIN** two tables and use aggregate functions again, but this time, using the **WHERE** condition. No index is required for this query as it has no effect on the execution of the query.

```
SELECT m.movieid, title, count_reviews
FROM movies AS m
JOIN (
SELECT movieid, COUNT(*) AS count_reviews
FROM ratings
WHERE rating = 5
GROUP BY movieid
ORDER BY count_reviews
DESC LIMIT 1) AS t
ON m. movieid = t.movieid;
```

```
moviesdb=# SELECT m.movieid, title, count_reviews
moviesdb-# FROM movies AS m
moviesdb-# JOIN (
moviesdb(# SELECT movieid, COUNT(*) AS count_reviews
moviesdb(# FROM ratings
moviesdb(# WHERE rating = 5
moviesdb(# GROUP BY movieid
moviesdb(# ORDER BY count_reviews
moviesdb(# DESC LIMIT 1) AS t
moviesdb-# ON m.movieid = t.movieid;
 movieid |           title            | count_reviews
---------+----------------------------+---------------
     318 | Shawshank Redemption, The |         16460
(1 row)
```

3) Find the number of movies that are associated with at least 4 different genres.

To perform this query, one single subquery is needed with the **COUNT()** function. No index is required as this query was executed rather quickly.

```
SELECT CAST(COUNT(*) AS INTEGER)
FROM (
SELECT COUNT(gen_title) AS genre_count
FROM has_genre
GROUP BY movieid) AS t
WHERE genre_count >= 4;
```

```
moviesdb=# SELECT CAST(COUNT(*) AS INTEGER)
moviesdb-# FROM (
moviesdb(# SELECT COUNT(gen_title) AS genre_count
moviesdb(# FROM has_genre
moviesdb(# GROUP BY movieid) AS t
moviesdb-# WHERE genre_count >= 4;
 count
-------
   968
(1 row)
```

4) Find the most popular genre across all movies (genre associated with the highest number of movies).

To execute this query, we can simply use the **COUNT()** function with its accompanying **GROUP BY** and **ORDER BY** functions on just the 'has_genre' table. No index is required as this is simple counting.

```
SELECT gen_title, COUNT(gen_title)
FROM has_genre
GROUP BY gen_title
ORDER BY COUNT
DESC LIMIT 1;
```

```
moviesdb=# SELECT gen_title, COUNT(gen_title)
moviesdb-# FROM has_genre
moviesdb-# GROUP BY gen_title
moviesdb-# ORDER BY COUNT
moviesdb-# DESC LIMIT 1;
 gen_title | count
-----------+-------
 Drama     |  5339
(1 row)
```

5) Find the genres that are associated with the best reviews (genres of movies that have more high ratings than low ratings). Display the genre, the number of high ratings (>=4.0), and the number of low ratings (< 4.0).

In order to perform this query, it'll require multiple **NATURAL JOIN**s within complex subqueries and between two subqueries. The use of the **COUNT()** function will also be required.

```
SELECT gen_title, high_ratings, low_ratings
FROM (
(SELECT gen_title, COUNT(*) AS high_ratings FROM
has_genre NATURAL JOIN ratings WHERE rating >= 4
GROUP BY gen_title) AS high
NATURAL JOIN
(SELECT gen_title, COUNT(*) AS low_ratings FROM
has_genre NATURAL JOIN ratings WHERE rating < 4 GROUP
BY gen_title) AS low)
WHERE high_ratings > low_ratigs;
```

```
moviesdb=# SELECT gen_title, high_ratings, low_ratings
moviesdb-# FROM (
moviesdb(# (SELECT gen_title, COUNT(*) AS high_ratings FROM has_genre NATURAL JOIN ratings WHERE rating >= 4 GROUP BY gen_title) AS high
moviesdb(# NATURAL JOIN
moviesdb(# (SELECT gen_title, COUNT(*) AS low_ratings FROM has_genre NATURAL JOIN ratings WHERE rating < 4 GROUP BY gen_title) AS low)
moviesdb-# WHERE high_ratings > low_ratings;
  gen_title  | high_ratings | low_ratings
-------------+--------------+-------------
 Animation   |       275590 |      243522
 Crime       |       826375 |      648582
 Documentary |        64986 |       38468
 Drama       |      2455297 |     1888901
 Film-Noir   |        94675 |       36917
 IMAX        |         5501 |        3579
 Musical     |       250613 |      230561
 Mystery     |       356799 |      274145
 Romance     |       977944 |      923939
 War         |       348331 |      219732
 Western     |       108365 |      102094
(11 rows)
```

An attempt at making an index on the attribute ‘rating’ from the table ‘ratings’ was used. But as you will see, the index had no effect on the execution of this particular query due to the index not even being used. There was not much of a difference in running times as it remained relatively consistent. We can know this by using the **`EXPLAIN ANALYZE`** function.

Before creating the index:

```
moviesdb=# EXPLAIN ANALYZE
moviesdb-# SELECT gen_title, high_ratings, low_ratings
moviesdb-# FROM (
moviesdb(# (SELECT gen_title, COUNT(*) AS high_ratings FROM has_genre NATURAL JOIN ratings WHERE rating >= 4 GROUP BY gen_title) AS high
moviesdb(# NATURAL JOIN
moviesdb(# (SELECT gen_title, COUNT(*) AS low_ratings FROM has_genre NATURAL JOIN ratings WHERE rating < 4 GROUP BY gen_title) AS low)
moviesdb-# WHERE high_ratings > low_ratings;
```

```
                                                     QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
----------
 Hash Join  (cost=620851.49..620856.81 rows=7 width=23) (actual time=40912.746..40944.617 rows=11 loops=1)
   Hash Cond: ((has_genre.gen_title)::text = (low.gen_title)::text)
   Join Filter: ((count(*)) > low.low_ratings)
   Rows Removed by Join Filter: 9
   ->  Finalize GroupAggregate  (cost=310822.88..310827.95 rows=20 width=15) (actual time=20408.241..20439.055 rows=20 loops=1)
         Group Key: has_genre.gen_title
         ->  Gather Merge  (cost=310822.88..310827.55 rows=40 width=15) (actual time=20408.202..20438.966 rows=60 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Sort  (cost=309822.86..309822.91 rows=20 width=15) (actual time=20365.961..20365.970 rows=20 loops=3)
                     Sort Key: has_genre.gen_title
                     Sort Method: quicksort  Memory: 26kB
                     Worker 0:  Sort Method: quicksort  Memory: 26kB
                     Worker 1:  Sort Method: quicksort  Memory: 26kB
                     ->  Partial HashAggregate  (cost=309822.22..309822.42 rows=20 width=15) (actual time=20365.751..20365.772 rows=20 loops=3)
                           Group Key: has_genre.gen_title
                           Batches: 1  Memory Usage: 24kB
                           Worker 0:  Batches: 1  Memory Usage: 24kB
                           Worker 1:  Batches: 1  Memory Usage: 24kB
                           ->  Parallel Hash Join  (cost=1053.52..288108.43 rows=4342759 width=7) (actual time=7408.829..17322.722 rows=4365613 loops=3)
                                 Hash Cond: (ratings.movieid = has_genre.movieid)
                                 ->  Parallel Seq Scan on ratings  (cost=0.00..243167.61 rows=2096122 width=4) (actual time=7404.659..14096.030 rows=1668561 loops=3)
                                       Filter: (rating >= '4'::double precision)
                                       Rows Removed by Filter: 1664790
                                 ->  Parallel Hash  (cost=894.95..894.95 rows=12685 width=11) (actual time=4.030..4.031 rows=7188 loops=3)
                                       Buckets: 32768  Batches: 1  Memory Usage: 1280kB
                                       ->  Parallel Index Only Scan using has_genre_pkey on has_genre  (cost=0.29..894.95 rows=12685 width=11) (actual time=0.043..4.623 rows=21564 loops=1)
                                             Heap Fetches: 0
   ->  Hash  (cost=310028.36..310028.36 rows=20 width=15) (actual time=20504.453..20505.498 rows=20 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Subquery Scan on low  (cost=310023.09..310028.36 rows=20 width=15) (actual time=20504.314..20505.462 rows=20 loops=1)
               ->  Finalize GroupAggregate  (cost=310023.09..310028.16 rows=20 width=15) (actual time=20504.312..20505.455 rows=20 loops=1)
                     Group Key: has_genre_1.gen_title
                     ->  Gather Merge  (cost=310023.09..310027.76 rows=40 width=15) (actual time=20504.300..20505.407 rows=59 loops=1)
                           Workers Planned: 2
                           Workers Launched: 2
                           ->  Sort  (cost=309023.07..309023.12 rows=20 width=15) (actual time=20408.963..20408.972 rows=20 loops=3)
                                 Sort Key: has_genre_1.gen_title
                                 Sort Method: quicksort  Memory: 26kB
                                 Worker 0:  Sort Method: quicksort  Memory: 26kB
                                 Worker 1:  Sort Method: quicksort  Memory: 26kB
                                 ->  Partial HashAggregate  (cost=309022.43..309022.63 rows=20 width=15) (actual time=20408.763..20408.782 rows=20 loops=3)
                                       Group Key: has_genre_1.gen_title
                                       Batches: 1  Memory Usage: 24kB
                                       Worker 0:  Batches: 1  Memory Usage: 24kB
                                       Worker 1:  Batches: 1  Memory Usage: 24kB
                                       ->  Parallel Hash Join  (cost=1053.52..287573.37 rows=4289812 width=7) (actual time=7488.107..17422.982 rows=4290119 loops=3)
                                             Hash Cond: (ratings_1.movieid = has_genre_1.movieid)
                                             ->  Parallel Seq Scan on ratings ratings_1  (cost=0.00..243167.61 rows=2070567 width=4) (actual time=7481.538..14170.425 rows=1664790 loops=3)
                                                   Filter: (rating < '4'::double precision)
                                                   Rows Removed by Filter: 1668561
                                             ->  Parallel Hash  (cost=894.95..894.95 rows=12685 width=11) (actual time=6.398..6.399 rows=7188 loops=3)
                                                   Buckets: 32768  Batches: 1  Memory Usage: 1280kB
                                                   ->  Parallel Index Only Scan using has_genre_pkey on has_genre has_genre_1  (cost=0.29..894.95 rows=12685 width=11) (actual time=0.030..7.538 rows=2156
4 loops=1)
                                                         Heap Fetches: 0
 Planning Time: 1.037 ms
 Execution Time: 40944.978 ms
(57 rows)
```

After creating the index:

```
moviesdb=# CREATE INDEX idx_rating on ratings(rating);
CREATE INDEX
moviesdb=# EXPLAIN ANALYZE
moviesdb-# SELECT gen_title, high_ratings, low_ratings
moviesdb-# FROM (
moviesdb(# (SELECT gen_title, COUNT(*) AS high_ratings FROM has_genre NATURAL JOIN ratings WHERE rating >= 4 GROUP BY gen_title) AS high
moviesdb(# NATURAL JOIN
moviesdb(# (SELECT gen_title, COUNT(*) AS low_ratings FROM has_genre NATURAL JOIN ratings WHERE rating < 4 GROUP BY gen_title) AS low)
moviesdb-# WHERE high_ratings > low_ratings;
```

```
                                                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------
 Hash Join  (cost=620851.49..620856.81 rows=7 width=23) (actual time=42176.486..42202.798 rows=11 loops=1)
   Hash Cond: ((has_genre.gen_title)::text = (low.gen_title)::text)
   Join Filter: ((count(*)) > low.low_ratings)
   Rows Removed by Join Filter: 9
   ->  Finalize GroupAggregate  (cost=310822.88..310827.95 rows=20 width=15) (actual time=20988.866..21014.358 rows=20 loops=1)
         Group Key: has_genre.gen_title
         ->  Gather Merge  (cost=310822.88..310827.55 rows=40 width=15) (actual time=20988.851..21014.287 rows=58 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Sort  (cost=309822.86..309822.91 rows=20 width=15) (actual time=20926.619..20926.627 rows=19 loops=3)
                     Sort Key: has_genre.gen_title
                     Sort Method: quicksort  Memory: 26kB
                     Worker 0:  Sort Method: quicksort  Memory: 26kB
                     Worker 1:  Sort Method: quicksort  Memory: 26kB
                     ->  Partial HashAggregate  (cost=309822.22..309822.42 rows=20 width=15) (actual time=20926.433..20926.447 rows=19 loops=3)
                           Group Key: has_genre.gen_title
                           Batches: 1  Memory Usage: 24kB
                           Worker 0:  Batches: 1  Memory Usage: 24kB
                           Worker 1:  Batches: 1  Memory Usage: 24kB
                           ->  Parallel Hash Join  (cost=1053.52..288108.43 rows=4342759 width=7) (actual time=7998.413..17871.086 rows=4365613 loops=3)
                                 Hash Cond: (ratings.movieid = has_genre.movieid)
                                 ->  Parallel Seq Scan on ratings  (cost=0.00..243167.61 rows=2096122 width=4) (actual time=7993.276..14659.155 rows=1668561 loops=3)
                                       Filter: (rating >= '4'::double precision)
                                       Rows Removed by Filter: 1664790
                                 ->  Parallel Hash  (cost=894.95..894.95 rows=12685 width=11) (actual time=4.906..4.907 rows=7188 loops=3)
                                       Buckets: 32768  Batches: 1  Memory Usage: 1280kB
                                       ->  Parallel Index Only Scan using has_genre_pkey on has_genre  (cost=0.29..894.95 rows=12685 width=11) (actual time=0.041..5.577 rows=21564 loops=1)
                                             Heap Fetches: 0
   ->  Hash  (cost=310028.36..310028.36 rows=20 width=15) (actual time=21187.540..21188.354 rows=20 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Subquery Scan on low  (cost=310023.09..310028.36 rows=20 width=15) (actual time=21187.313..21188.300 rows=20 loops=1)
               ->  Finalize GroupAggregate  (cost=310023.09..310028.16 rows=20 width=15) (actual time=21187.312..21188.289 rows=20 loops=1)
                     Group Key: has_genre_1.gen_title
                     ->  Gather Merge  (cost=310023.09..310027.76 rows=40 width=15) (actual time=21187.283..21188.219 rows=60 loops=1)
                           Workers Planned: 2
                           Workers Launched: 2
                           ->  Sort  (cost=309023.07..309023.12 rows=20 width=15) (actual time=21114.037..21114.046 rows=20 loops=3)
                                 Sort Key: has_genre_1.gen_title
                                 Sort Method: quicksort  Memory: 26kB
                                 Worker 0:  Sort Method: quicksort  Memory: 26kB
                                 Worker 1:  Sort Method: quicksort  Memory: 26kB
                                 ->  Partial HashAggregate  (cost=309022.43..309022.63 rows=20 width=15) (actual time=21113.867..21113.880 rows=20 loops=3)
                                       Group Key: has_genre_1.gen_title
                                       Batches: 1  Memory Usage: 24kB
                                       Worker 0:  Batches: 1  Memory Usage: 24kB
                                       Worker 1:  Batches: 1  Memory Usage: 24kB
                                       ->  Parallel Hash Join  (cost=1053.52..287573.37 rows=4289812 width=7) (actual time=8134.236..18115.025 rows=4290119 loops=3)
                                             Hash Cond: (ratings_1.movieid = has_genre_1.movieid)
                                             ->  Parallel Seq Scan on ratings ratings_1  (cost=0.00..243167.61 rows=2070567 width=4) (actual time=8126.842..14879.170 rows=1664790 loops=3)
                                                   Filter: (rating < '4'::double precision)
                                                   Rows Removed by Filter: 1668561
                                             ->  Parallel Hash  (cost=894.95..894.95 rows=12685 width=11) (actual time=7.174..7.175 rows=7188 loops=3)
                                                   Buckets: 32768  Batches: 1  Memory Usage: 1280kB
                                                   ->  Parallel Index Only Scan using has_genre_pkey on has_genre has_genre_1  (cost=0.29..894.95 rows=12685 width=11) (actual time=0.035..8.512 rows=2156
4 loops=1)
                                                         Heap Fetches: 0
 Planning Time: 7.447 ms
 Execution Time: 42203.122 ms
(57 rows)
```

6) Find the genres that are associated with the most recent movies (genres that have more recent movies than old movies). Display the genre, the number of recent movies (>=2000), and the number of old movies (< 2000).

Similar to the previous query, this one will also require multiple **NATURAL JOIN**s within complex subqueries and between two subqueries. The use of the **COUNT()** function will also be required.

```
SELECT gen_title, recent, old
FROM (
(SELECT gen_title, COUNT(*) AS recent FROM has_genre
NATURAL JOIN movies WHERE year >= 2000 GROUP BY
gen_title) AS recent
NATURAL JOIN
(SELECT gen_title, COUNT(*) AS old FROM has_genre
NATURAL JOIN movies WHERE year < 2000 GROUP BY
gen_title) AS old)
WHERE recent > old;
```

```
moviesdb=# SELECT gen_title, recent, old
moviesdb-# FROM (
moviesdb(# (SELECT gen_title, COUNT(*) AS recent FROM has_genre NATURAL JOIN movies WHERE year >= 2000 GROUP BY gen_title) AS recent
moviesdb(# NATURAL JOIN
moviesdb(# (SELECT gen_title, COUNT(*) AS old FROM has_genre NATURAL JOIN movies WHERE year < 2000 GROUP BY gen_title) AS old)
moviesdb-# WHERE recent > old;
  gen_title  | recent | old
-------------+--------+-----
 Documentary |    252 | 230
(1 row)
```

Another attempt at making an index was made on the attribute 'year' from the table 'movies'. But as you will see, the index had no effect on the execution of this particular query due to the index not even being used. There was not much of a difference in running times as it remained relatively consistent. We can know this by using the **`EXPLAIN ANALYZE`** function.

Before creating the index:

```
moviesdb=# EXPLAIN ANALYZE
moviesdb-# SELECT gen_title, recent, old
moviesdb-# FROM (
moviesdb(# (SELECT gen_title, COUNT(*) AS recent FROM has_genre NATURAL JOIN movies WHERE year >= 2000 GROUP BY gen_title) AS recent
moviesdb(# NATURAL JOIN
moviesdb(# (SELECT gen_title, COUNT(*) AS old FROM has_genre NATURAL JOIN movies WHERE year < 2000 GROUP BY gen_title) AS old)
moviesdb-# WHERE recent > old;
                                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=1443.54..1444.00 rows=7 width=23) (actual time=29.274..29.286 rows=1 loops=1)
   Hash Cond: ((has_genre.gen_title)::text = (old.gen_title)::text)
   Join Filter: ((count(*)) > old.old)
   Rows Removed by Join Filter: 18
   ->  HashAggregate  (cost=674.19..674.39 rows=20 width=15) (actual time=13.736..13.743 rows=19 loops=1)
         Group Key: has_genre.gen_title
         Batches: 1  Memory Usage: 24kB
         ->  Hash Join  (cost=253.13..641.39 rows=6559 width=7) (actual time=4.295..10.766 rows=7106 loops=1)
               Hash Cond: (has_genre.movieid = movies.movieid)
               ->  Seq Scan on has_genre  (cost=0.00..331.64 rows=21564 width=11) (actual time=0.022..2.566 rows=21564 loops=1)
               ->  Hash  (cost=212.51..212.51 rows=3249 width=4) (actual time=2.699..2.700 rows=3249 loops=1)
                     Buckets: 4096  Batches: 1  Memory Usage: 147kB
                     ->  Seq Scan on movies  (cost=0.00..212.51 rows=3249 width=4) (actual time=0.395..1.888 rows=3249 loops=1)
                           Filter: (year >= 2000)
                           Rows Removed by Filter: 7432
   ->  Hash  (cost=769.11..769.11 rows=20 width=15) (actual time=15.491..15.493 rows=20 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Subquery Scan on old  (cost=768.71..769.11 rows=20 width=15) (actual time=15.461..15.469 rows=20 loops=1)
               ->  HashAggregate  (cost=768.71..768.91 rows=20 width=15) (actual time=15.459..15.464 rows=20 loops=1)
                     Group Key: has_genre_1.gen_title
                     Batches: 1  Memory Usage: 24kB
                     ->  Hash Join  (cost=305.41..693.68 rows=15005 width=7) (actual time=4.081..11.173 rows=14458 loops=1)
                           Hash Cond: (has_genre_1.movieid = movies_1.movieid)
                           ->  Seq Scan on has_genre has_genre_1  (cost=0.00..331.64 rows=21564 width=11) (actual time=0.032..1.900 rows=21564 loops=1)
                           ->  Hash  (cost=212.51..212.51 rows=7432 width=4) (actual time=3.998..3.999 rows=7432 loops=1)
                                 Buckets: 8192  Batches: 1  Memory Usage: 326kB
                                 ->  Seq Scan on movies movies_1  (cost=0.00..212.51 rows=7432 width=4) (actual time=0.015..2.167 rows=7432 loops=1)
                                       Filter: (year < 2000)
                                       Rows Removed by Filter: 3249
 Planning Time: 0.975 ms
 Execution Time: 29.484 ms
(31 rows)
```

After creating the index:

```
moviesdb=# EXPLAIN ANALYZE
moviesdb-# SELECT gen_title, recent, old
moviesdb-# FROM (
moviesdb(# (SELECT gen_title, COUNT(*) AS recent FROM has_genre NATURAL JOIN movies WHERE year >= 2000 GROUP BY gen_title) AS recent
moviesdb(# NATURAL JOIN
moviesdb(# (SELECT gen_title, COUNT(*) AS old FROM has_genre NATURAL JOIN movies WHERE year < 2000 GROUP BY gen_title) AS old)
moviesdb-# WHERE recent > old;
                                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=1392.11..1392.56 rows=7 width=23) (actual time=28.080..28.089 rows=1 loops=1)
   Hash Cond: ((has_genre.gen_title)::text = (old.gen_title)::text)
   Join Filter: ((count(*)) > old.old)
   Rows Removed by Join Filter: 18
   ->  HashAggregate  (cost=622.75..622.95 rows=20 width=15) (actual time=13.154..13.159 rows=19 loops=1)
         Group Key: has_genre.gen_title
         Batches: 1  Memory Usage: 24kB
         ->  Hash Join  (cost=201.69..589.96 rows=6559 width=7) (actual time=3.710..10.284 rows=7106 loops=1)
               Hash Cond: (has_genre.movieid = movies.movieid)
               ->  Seq Scan on has_genre  (cost=0.00..331.64 rows=21564 width=11) (actual time=0.015..2.591 rows=21564 loops=1)
               ->  Hash  (cost=161.08..161.08 rows=3249 width=4) (actual time=1.959..1.960 rows=3249 loops=1)
                     Buckets: 4096  Batches: 1  Memory Usage: 147kB
                     ->  Bitmap Heap Scan on movies  (cost=41.46..161.08 rows=3249 width=4) (actual time=0.194..1.030 rows=3249 loops=1)
                           Recheck Cond: (year >= 2000)
                           Heap Blocks: exact=56
                           ->  Bitmap Index Scan on idx_year  (cost=0.00..40.65 rows=3249 width=0) (actual time=0.175..0.175 rows=3249 loops=1)
                                 Index Cond: (year >= 2000)
   ->  Hash  (cost=769.11..769.11 rows=20 width=15) (actual time=14.889..14.890 rows=20 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Subquery Scan on old  (cost=768.71..769.11 rows=20 width=15) (actual time=14.874..14.879 rows=20 loops=1)
               ->  HashAggregate  (cost=768.71..768.91 rows=20 width=15) (actual time=14.873..14.876 rows=20 loops=1)
                     Group Key: has_genre_1.gen_title
                     Batches: 1  Memory Usage: 24kB
                     ->  Hash Join  (cost=305.41..693.68 rows=15005 width=7) (actual time=4.180..10.802 rows=14458 loops=1)
                           Hash Cond: (has_genre_1.movieid = movies_1.movieid)
                           ->  Seq Scan on has_genre has_genre_1  (cost=0.00..331.64 rows=21564 width=11) (actual time=0.027..1.721 rows=21564 loops=1)
                           ->  Hash  (cost=212.51..212.51 rows=7432 width=4) (actual time=4.099..4.100 rows=7432 loops=1)
                                 Buckets: 8192  Batches: 1  Memory Usage: 326kB
                                 ->  Seq Scan on movies movies_1  (cost=0.00..212.51 rows=7432 width=4) (actual time=0.014..2.265 rows=7432 loops=1)
                                       Filter: (year < 2000)
                                       Rows Removed by Filter: 3249
 Planning Time: 0.891 ms
 Execution Time: 28.309 ms
(33 rows)
```

b) Debiasing the Ratings of Users

Now we will move on to more complicated queries. We will attempt to find the top 10 movies (with attributes ‘movieid’ and ‘title’) that have received the most biased ratings.

What does having biased ratings mean? Suppose a user is always positively biased and rates all movies with 5 stars, or always negatively biased and rates movies with 1 or 2 stars. We want to find these users and de-bias their ratings, that is, find the average rating for a movie after neutralizing the rating of these users. There are several approaches to neutralize a biased rating. For example, we can remove these ratings from the database. Or, we can replace the biased rating with 3.5, or with the average rating of the movie at that moment. We can also reduce the weight of the rating of biased users when we want to compute the average rating of a movie.

In this approach, we will try to identify the biased users by computing the difference between their rating and the average rating of the movie. We will save the difference in an additional column in the ‘ratings’ table. We will then update their rating with the average rating of the movie and we will also update the timestamp. After we do this operation for all users, we will then perform the same task one more time, now using the new average rating for each movie. We may have to repeat this process more than 2 times if necessary.

At the end, we will try to extract the average rating of a movie and identify the top 10 movies that had received the most biased ratings.

Step 1) Find the difference between a user's rating and the average rating of the movie he has rated.

We will do this by creating a new table, 'ratings_with_diff', that includes all columns from table 'ratings', plus 2 new columns: the average rating and the difference (rating - average rating).

First, we create the table 'ratings_with_diff' using the same attributes of the table 'ratings'. This might take a while as the command prompt is essentially making a copy of the table 'ratings' which has 10000054 rows. We can check if the table has been made using the **\d** function in the command prompt.

```
CREATE TABLE ratings_with_diff AS TABLE ratings;
```

```
moviesdb=# \d
                List of relations
 Schema |        Name         | Type  |  Owner
--------+---------------------+-------+----------
 public | genres              | table | postgres
 public | has_genre           | table | postgres
 public | movies              | table | postgres
 public | ratings             | table | postgres
 public | ratings_with_diff   | table | postgres
 public | tags                | table | postgres
 public | users               | table | postgres
(7 rows)
```

Once the new table has been created, alter the table to add columns 'avg_rating' and 'difference'. The schema is now ratings_with_diff(userid, movieid, rating, time, avg_rating, difference). We can check this with the command \d+ ratings_with_diff.

```
ALTER TABLE ratings_with_diff ADD COLUMN avg_rating
DOUBLE PRECISION;
ALTER TABLE ratings_with_diff ADD COLUMN difference
DOUBLE PRECISION;
```

```
moviesdb=# \d+ ratings_with_diff
                                Table "public.ratings_with_diff"
   Column   |       Type       | Collation | Nullable | Default | Storage | Stats target | Description
------------+------------------+-----------+----------+---------+---------+--------------+-------------
 userid     | integer          |           |          |         | plain   |              |
 movieid    | integer          |           |          |         | plain   |              |
 rating     | real             |           |          |         | plain   |              |
 time       | numeric          |           |          |         | main    |              |
 avg_rating | double precision |           |          |         | plain   |              |
 difference | double precision |           |          |         | plain   |              |
Access method: heap
```

Now we'll create the table 'avg_ratings' with attributes 'movieid' and 'avg_rating' which will contain the average rating for each movie, hence saving time populating the table 'ratings_with_diff' later on. We'll populate this table using data from the 'ratings' table and

the use of the **AVG()** function. Like before, we can also check if the table was created and if the table contains the desired columns made. 10677 rows should be affected from these queries.

```
CREATE TABLE avg_ratings(movieid NUMERIC, avg_rating
DOUBLE PRECISION);
INSERT INTO avg_ratings SELECT movieid, AVG(rating)
AS avg_rating FROM ratings GROUP BY movieid;
```

```
moviesdb=# \d
             List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | avg_ratings       | table | postgres
 public | genres            | table | postgres
 public | has_genre         | table | postgres
 public | movies            | table | postgres
 public | ratings           | table | postgres
 public | ratings_with_diff | table | postgres
 public | tags              | table | postgres
 public | users             | table | postgres
(8 rows)


moviesdb=# \d+ avg_ratings
                                Table "public.avg_ratings"
   Column   |       Type       | Collation | Nullable | Default | Storage | Stats target | Description
------------+------------------+-----------+----------+---------+---------+--------------+-------------
 movieid    | numeric          |           |          |         | main    |              |
 avg_rating | double precision |           |          |         | plain   |              |
Access method: heap
```

We then **UPDATE** the 'avg_rating' in the 'ratings_with_diff' table with the averages from the 'avg_ratings' table. This will also take a while as 10000054 rows will be updated.

```
UPDATE ratings_with_diff SET avg_rating =
avg_ratings.avg_rating
FROM avg_ratings
WHERE ratings_with_diff.movieid =
avg_ratings.movieid;
We then UPDATE the 'difference' column. Once again,
this will take a while as 10000054 rows are being
updated.
UPDATE ratings_with_diff SET difference = rating -
avg_rating;
```

Step 2) Update the rating of users whose rating difference (absolute value) is > 3.

This will involve a subquery within the **SET** function. 40498 rows should be updated from this query.

```
UPDATE ratings_with_diff r
SET rating = (SELECT avg_rating FROM avg_ratings
WHERE r.movieid = avg_ratings.movieid)
```

```
WHERE @difference > 3;
```

Step 3) Find the new difference between a user's rating and the average rating of the movie they have rated.

To perform this step, we need to first take the new average of each movie. This information will be stored in a new table called 'avg_ratings2'. We can, again, check if this was performed successfully with the **\d** function in the command prompt. 10677 rows should be added.

```
CREATE TABLE avg_ratings2(movieid NUMERIC, avg_rating
DOUBLE PRECISION);
INSERT INTO avg_ratings2 SELECT movieid, AVG(rating)
AS avg_rating FROM ratings_with_diff GROUP BY
movieid;
```

```
moviesdb=# \d
              List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | avg_ratings       | table | postgres
 public | avg_ratings2      | table | postgres
 public | genres            | table | postgres
 public | has_genre         | table | postgres
 public | movies            | table | postgres
 public | ratings           | table | postgres
 public | ratings_with_diff | table | postgres
 public | tags              | table | postgres
 public | users             | table | postgres
(9 rows)
```

Next, **UPDATE** the table 'ratings_with_diff' with the new averages from the 'avg_ratings2' table. This will take a while as 10000054 rows will be affected.

```
UPDATE ratings_with_diff
SET avg_rating = avg_ratings2.avg_rating
FROM avg_ratings2
WHERE ratings_with_diff.movieid =
avg_ratings2.movieid;
Then, we find the new difference. This will also take
a while because of the 10000054 rows that are in this
table.
UPDATE ratings_with_diff SET difference = rating -
avg_rating;
```

Step 4) Again, update the rating of users whose rating difference (absolute value) is > 3.

This is basically the same query as in step 2 except with the table 'avg_ratings2'. 751 rows should be updated.

```
UPDATE ratings_with_diff r
SET rating = (SELECT avg_rating FROM avg_ratings2
WHERE r.movieid = avg_ratings2.movieid)
WHERE @difference > 3;
```

We can check how the table 'ratings_with_diff' should look by viewing the first 5 rows of the table. Please note that the data itself might not match due to the lack of cohesive ordering in this table. Just make sure the attributes are all there and that the math for the 'difference' column is correct.

```
SELECT * FROM ratings_with_diff LIMIT 5;
```

```
moviesdb=# SELECT * FROM ratings_with_diff LIMIT 5;
 userid | movieid |  rating   |   time    |    avg_rating     |      difference
--------+---------+-----------+-----------+-------------------+----------------------
  26180 |     260 |  4.220209 | 965214344 | 4.278077543128199 | -0.05786842142409743
  26186 |      47 | 4.0312376 | 886301554 | 4.090626288538033 | -0.05938868630414618
  26186 |     296 |  4.157426 | 886301386 | 4.255335048668535 | -0.09790916823640572
  26186 |    1206 | 4.0051575 | 886302283 | 4.111831343868744 | -0.10667387316561872
  26188 |      50 |  4.367142 | 991775501 | 4.387107372303773 | -0.01996517183380231
(5 rows)
```

Step 5) Find the average rating for each movie before the de-biasing (from the 'ratings' table) and the average rating for each movie after the de-biasing (from the 'ratings_with_diff' table). List the top 10 movies that have the biggest difference between these two average ratings. (These are the movies that had the most biased ratings.)

This will require a very complex query containing 3 subqueries that all come together with the **NATURAL JOIN** function. Make sure the selected tables are in the right order and watch out for spelling errors. Please note that the movies may not be in order. As long as the movies match, the de-biased query is complete.

```
SELECT movieid, title, original, debiased,
@original-debiased AS bias FROM
(SELECT movieid, title FROM movies) t1
NATURAL JOIN
(SELECT movieid, avg_rating AS original FROM
avg_ratings) t2
NATURAL JOIN
(SELECT movieid, AVG(rating) AS debiased FROM
ratings_with_diff GROUP BY movieid) t3
ORDER BY bias DESC LIMIT 10;
```

The top 10 movies with the most biased ratings:

```
moviesdb=# SELECT movieid, title, original_rating, debiased_rating, @original_rating-debiased_rating AS bias_rating FROM
moviesdb-# (SELECT movieid, title FROM movies) AS t1
moviesdb-# NATURAL JOIN
moviesdb-# (SELECT movieid, avg_rating AS original_rating FROM avg_ratings) AS t2
moviesdb-# NATURAL JOIN
moviesdb-# (SELECT movieid, AVG(rating) AS debiased_rating FROM ratings_with_diff GROUP BY movieid) AS t3
moviesdb-# ORDER BY bias_rating DESC LIMIT 10;
 movieid |                                  title                                  |  original_rating   |  debiased_rating   |     bias_rating
---------+-------------------------------------------------------------------------+--------------------+--------------------+---------------------
    8484 | Human Condition I, The (Ningen no joken I)                              |            3.59375 |        4.173828125 |         0.580078125
    5793 | Time Changer                                                            | 1.9285714285714286 | 1.4897959232330322 |  0.4387755053383964
    6397 | Bizarre, Bizarre (Dr⌈le de drame ou L'0trange aventure de Docteur Molyneux) |             3.5625 |          3.9453125 |           0.3828125
    8423 | Kid Brother, The                                                        | 3.5588235294117645 |  3.918685127707089 |  0.3598615982953244
   41627 | Samurai Rebellion (J⌈i-uchi: Hairy⌈ tsuma shimatsu)                     |               4.05 |  4.405000019073486 | 0.35500001907348633
   26326 | Holy Mountain, The (Montata sagrada, La)                                |               4.05 |  4.405000019073486 | 0.35500001907348633
    5889 | Cruel Romance, A (Zhestokij Romans)                                     | 3.5555555555555554 |  3.895061731338501 |  0.3395061757829456
    6599 | Accattone                                                               |  3.642857142857143 |  3.979591829436166 |  0.3367346865790233
   25766 | Crowd, The                                                              |  3.716666666666667 |  4.038333336512248 | 0.32166666984558123
   25930 | Odd Man Out                                                             | 3.8863636363636362 |  4.194214885885065 | 0.30785124952142917
(10 rows)
```

Step 6) EXTRA: Who is the most biased user?

This is an extra step that is not required, but can be useful practice in coding in SQL and completing complicated queries. To complete this query, we can define the most biased user as having the most rows changed during de-biasing steps. Thus, a much simpler query with subqueries is needed and we would need to **COUNT()** the number of rows this user has between the 'ratings' and 'ratings_with_diff' tables.

```
SELECT userid, count(*) FROM
(SELECT userid, movieid, rating AS original FROM
ratings_with_diff) t1
NATURAL JOIN
(SELECT userid, movieid, rating AS debiased FROM
ratings) t2
WHERE original <> debiased
GROUP BY userid
ORDER BY count(*) DESC
LIMIT 1;
```

The most biased user:

```
moviesdb=# SELECT userid, COUNT(*) FROM
moviesdb-# (SELECT userid, movieid, rating AS original FROM ratings_with_diff) AS t1
moviesdb-# NATURAL JOIN
moviesdb-# (SELECT userid, movieid, rating AS debiased FROM ratings) AS t2
moviesdb-# WHERE original <> debiased
moviesdb-# GROUP BY userid
moviesdb-# ORDER BY COUNT(*) DESC
moviesdb-# LIMIT 1;
 userid | count
--------+-------
  59342 |   223
(1 row)
```

CHAPTER 5: DISCUSSION

To conclude this project, I will discuss some final observations and assumptions.

In the beginning of the project, I had made the narrow-minded assumption that the E/R design of the database only involved three tables from the three files that were initially given. After more thorough analysis and thought processes, I edited my E/R diagram to match the designated design of the database. From now on, I should refrain from assuming that the number of tables in a database equals the number of files of data given.

I didn't seem to find any constraints while designing the database, except for having to **SET** the encoding when loading data into tables that had the movie title. The reason this was a constraint for me was because of various movie titles that would have special characters in it that would make processing the data difficult and both the Python prompt and command prompt would run into encoding errors.

I ran into a significant amount of redundancy in testing my database, specifically when using the **COUNT()** function to complete various queries. There was also an extensive amount of redundancy in checking for **NULL** values throughout all the tables. Besides those two key redundancy parts, there weren't many errors that I ran into.

The percentage of unknown values in the attributes is close to zero, if we include the movie that had no genres listed. Besides that, there weren't any **NULL** nor missing values in my database.

There wasn't much benefit from using indexes in my case. They weren't even used after creating them, which suggests the database must need to be a lot more bigger in order for Postgres to be resorted to using an index.

The biggest challenge I ran into during this project was simply loading the data into the tables. I had to constantly **DROP** and re**CREATE** tables due to the primary key constraints on certain tables. Then I also ran into problems with coding in Python to split strings and appending lines properly. I spent most of the time during this portion of loading the data in coding the Python scripts to process the necessary data needed for the tables.

I also ran into challenges in testing the database as some of these queries were hard to complete. I had to resort to Google to look up various ways to code subqueries and learn new SQL functions. It was definitely helpful to learn new tricks in SQL as it made my SQL code look more concise and organized.

cccc