Edward Rees

Computer Science

CS 333 Introduction to Database Systems

Spring 2022

Design, Load, and Explore a Movies database

Table of Contents

Chapter 1: Project Description

The Goal of this Project

The goal of this project is to better understand the process of creating and working with a database. This will be done by understanding various components of the database design, such as loading data from a file, designing and building a database based on information provided, testing the database with sample queries, exploring the database, querying the database created and optimizing queries for the database. In doing these, a stronger foundation and understanding of database design will be gained.

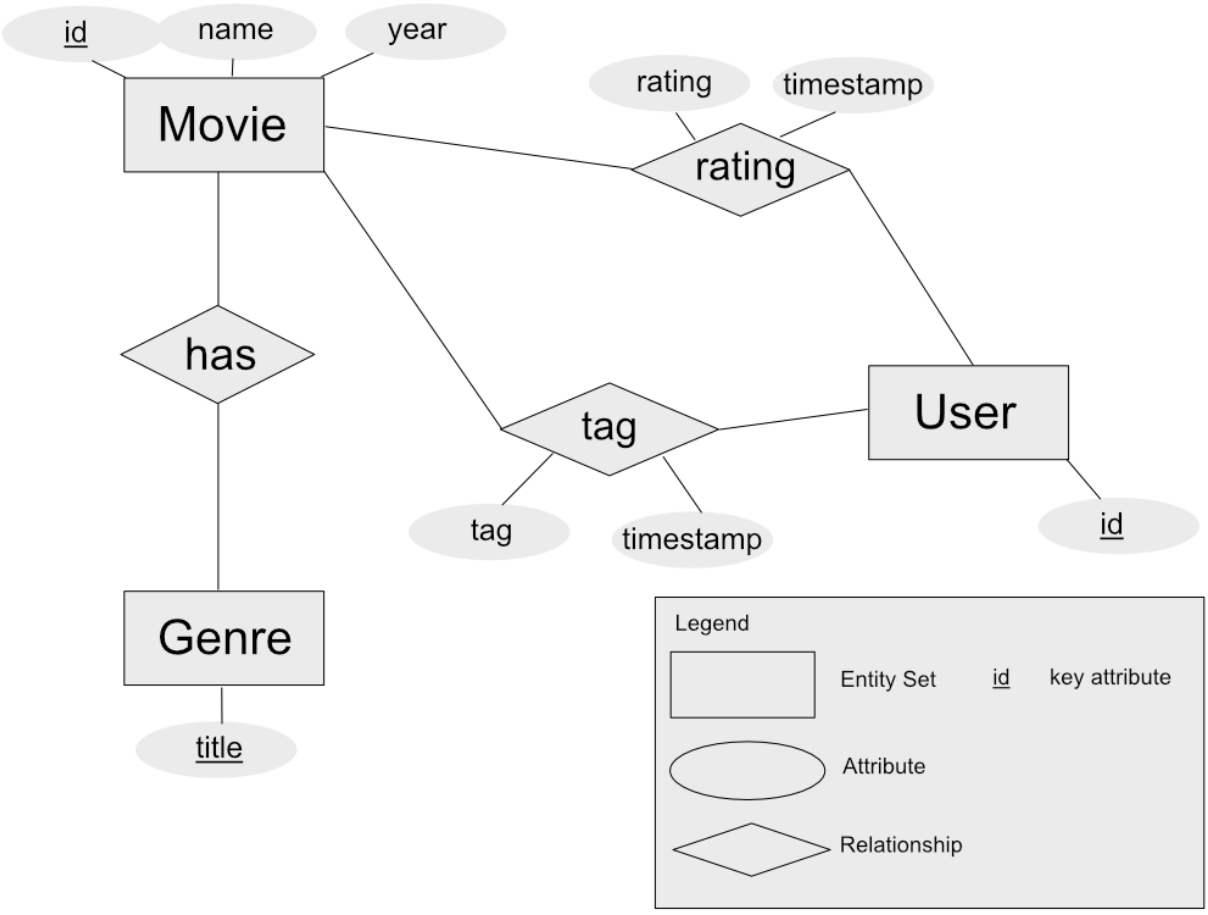
Data Exploration

The dataset contains three text (“txt”) files. The first is the movies.txt file, which contains a list of movies with each row containing the movie id, the movie name with the year released included, and a list of genres. The movie id is an integer, while the movie name and the genres are all strings. The second is the ratings.txt file, which contains a list of ratings from a given user id, movie id, rating, and the timestamp for the rating. All of the attributes from the ratings.txt are integers, as they are all represented as numbers. The third file is the tags.txt file, which contains the user id, the movie id, the tag, and the timestamp. The user id, movie id, and timestamp are all integers, while the tag itself is a string.

Chapter 2: Database Design

E/R Diagram

When looking at the data, I came up with the E/R diagram below. I saw that Movie would have to be its own entity set, with the attributes of id and name. I initially thought of Rating and Tag as another entity set, until noticing that userId is common in both, which made me think that User can be its own entity set, with an ID attribute. This leads to Rating and Tag both being relationships between Movie and User, with Rating having a rating and timestamp as additional attributes, and Tag having tag and timestamp as additional attributes. I then considered Genre as its own entity set connecting with Movie, with the genre being a primary key, with Movie and Genre having a many-to-many relationship.



Logical Schema

Movie (<u>id</u>: int, name: text, year: integer)
User (<u>id</u>: int)

Rating (userId: int, movieId: int, rating: text, timestamp: timestamp)
Tag (userId: int, movieId:int, tag: text, timestamp: timestamp)
Genre (gen_title: text)

Chapter 3: Load Data and Database Testing

Section A: Load Data

Repository: https://github.com/EdwardRees/Movie-Database

SQL Code:
db.sql

```
CREATE DATABASE moviesdb;
DROP TABLE IF EXISTS movies CASCADE;
DROP TABLE IF EXISTS users CASCADE;
DROP TABLE IF EXISTS ratings;
DROP TABLE IF EXISTS tags;
DROP TABLE IF EXISTS genres;
DROP TABLE IF EXISTS has_genres;
CREATE TABLE movies(
 id SERIAL PRIMARY KEY,
 title TEXT NOT NULL,
 year INTEGER NOT NULL
);
CREATE TABLE users(id SERIAL PRIMARY KEY);
CREATE TABLE ratings(
 movieId INTEGER NOT NULL,
 userId INTEGER NOT NULL,
 rating FLOAT NOT NULL,
 ratingTime INTEGER,
 FOREIGN KEY (movieId) REFERENCES movies(id),
 FOREIGN KEY (userId) REFERENCES users(id)
);
CREATE TABLE tags(
 movieId INTEGER NOT NULL,
 userId INTEGER NOT NULL,
 tag TEXT NOT NULL,
 tagTime INTEGER,
 FOREIGN KEY(movieId) REFERENCES movies(id),
 FOREIGN KEY(userId) REFERENCES users(id)
);
CREATE TABLE genres(
 genreTitle TEXT NOT NULL
);
CREATE TABLE has_genre (
 movieId INTEGER NOT NULL,
```

```
 genreTitle TEXT NOT NULL
);

\copy movies(id, title, year) FROM './out/movies.txt' DELIMITER ';';
\copy genres(genreTitle) FROM './out/genres.txt' DELIMITER ';';
\copy has_genre(movieId, genreTitle) FROM './out/has_genre.txt' DELIMITER ';';
\copy users(id) FROM './out/users.txt' DELIMITER ',';
\copy ratings(userId, movieId, rating, ratingTime) FROM './out/ratings.txt' DELIMITER
',';
\copy tags(userId, movieId, tag, tagTime) FROM './out/tags.txt' DELIMITER ',';
```

File Editing Code / Supporting Code:

Constants.py

```
DELIM1 = ';'
DELIM2 = ','
```

Download.py

```
from shutil import move
from os import mkdir, path
import wget
import zipfile

def prep():
 if path.exists("./movies"):
   print("movies directory already exists")
   print("Skipping setup process")
   return
 url = "https://www.dropbox.com/s/2rn7qc51yvmb766/movies.zip?dl=1"
 movieszip = wget.download(url, 'movies.zip')
 print(movieszip)

 path.isdir('./out') or mkdir('./out')
 path.isdir("./movies") or mkdir("./movies")

 move(movieszip, "./movies/movies.zip")

 print("Extracting movies.zip...")
 with zipfile.ZipFile("./movies/movies.zip", 'r') as zip_ref:
   zip_ref.extractall("./movies")
```

```
 print("Done!")

if __name__ == '__main__':
 prep()
```

Files.py

```
from sys import argv
from movies import parseMovies, validateMovies, outputMovies
from genres import parseGenres, outputGenres
from tags import parseTags, validateTags, outputTags
from ratings import parseRatings, outputRatings
from download import prep
from users import Users


def main():
   prep()
   if(len(argv) > 1):
     if(argv[1] == '-h'):
       print("Usage: python3 files.py [-h,-d]\n-h: help\n-d: debug. Will print out the values of the files parsed as the original list and dictionary values.\nIf no flag is passed, the program will continue with the default behavior of outputting the files to the output directory.")
       return
     elif(argv[1] == '-d'):
         movies = parseMovies()
         invalidMovies = validateMovies(movies)
         if(len(invalidMovies) > 0):
             print("Invalid movies found. Update parser to handle the following cases:")
             print("Invalid movies:")
             for movie in invalidMovies:
                 print(f"{movie['id']}:{movie['name']}")
         else:
             print(movies)
         genres = parseGenres()
         outputGenres(genres)
         tags = parseTags()
         invalidTags = validateTags(tags)
         if(len(invalidTags) > 0):
             print("Invalid tags found. Update parser to handle the following cases:")
             print("Invalid tags:")
```

```
            for tag in invalidTags:
                print(tag)
        else:
            print(tags)
        ratings = parseRatings()
        print(ratings)
    else:
        movies = parseMovies()
        invalidMovies = validateMovies(movies)
        if(len(invalidMovies) > 0):
            print("Invalid movies found. Update parser to handle the following cases:")
            print("Invalid movies:")
            for movie in invalidMovies:
                print(f"{movie['id']}:{movie['name']}")
        else:
            outputMovies(movies)
        genres = parseGenres()
        outputGenres(genres)
        tags = parseTags()
        invalidTags = validateTags(tags)
        if(len(invalidTags) > 0):
            print("Invalid tags found. Update parser to handle the following cases:")
            print("Invalid tags:")
            for tag in invalidTags:
                print(tag)
        else:
            outputTags(tags)
        ratings = parseRatings()
        outputRatings(ratings)
        Users.outputUsers()
    print("Done!")

if __name__ == '__main__':
    main()
```

Genres.py

```
from util import readFile
from constants import DELIM1, DELIM2

def parseGenres():
 print("Parsing genres...")
```

```
 genreList = []
 moviesFile = readFile('./movies/movies.txt')
 count = 1
 # Format: movieId:movieName (year):genre1|genre2|genre3
 for line in moviesFile.split("\n"):
   if line == '':
     continue
   line = line.split(":")
   movieId = line[0]
   genres = line[-1].split("|")
   for genre in genres:
     genreList.append({
       'movieId': movieId,
       'genre': genre
     })
 return genreList

def getGenreList(genres):
 genreList = []
 for genre in genres:
   if genre['genre'] not in genreList:
     genreList.append(genre['genre'])
 return genreList


def outputGenres(genres):
 print("Writing genres...")
 with open('./out/genres.txt', 'w') as f:
   genreList = getGenreList(genres)
   for genre in genreList:
     f.write(f"{genre}\n")
 with open('./out/has_genre.txt', 'w') as f:
   for genre in genres:
     f.write(f"{genre['movieId']}{DELIM1}{genre['genre']}\n")
```

Movies.py

```
from util import readFile
```

```
from constants import DELIM1, DELIM2

def parseMovies():
 print("Parsing movies...")
 movies = []
 genres = []
 moviesFile = readFile('./movies/movies.txt')
 for line in moviesFile.split('\n'):
   if line == '':
     continue
   line = line.split(":")
   movieId = line[0]
   genres = line[-1].split("|")
   movieName = ':'.join(line[1:-1])
   movieYear = movieName.split("(")[-1].split(")")[0]
   movieName = movieName.replace(f"({movieYear})", "")
   movieName = movieName.strip()
   movies.append({
     'id': movieId,
     'name': movieName,
     'year': movieYear,
   })
 return movies

def validateMovies(movies):
 print("Validating movies...")
 invalidMovies = []
 for movie in movies:
   if movie['year'] == '':
     print(f"{movie['id']}:{movie['name']}")
     invalidMovies.append(movie)
   if f"({movie['year']})" in movie['name']:
     print(f"{movie['id']}:{movie['name']}")
     invalidMovies.append(movie)
   try:
     int(movie['year'])
   except ValueError:
     invalidMovies.append(movie)
     print(movie['name'])
 return invalidMovies
```

```
def outputMovies(movies):
 print("Writing movies...")
 with open('./out/movies.txt', 'w') as f:
   for movie in movies:
     f.write(f"{movie['id']}{DELIM1}{movie['name']}{DELIM1}{movie['year']}\n")
```

Ratings.py

```
from util import readFile
from constants import DELIM1, DELIM2
from users import Users

def parseRatings():
 print("Parsing ratings...")
 ratings = []
 ratingsFile = readFile('./movies/ratings.txt')
 for line in ratingsFile.split('\n'):
   if line == '':
     continue
   line = line.split(":")
   Users.addToUsers(line[0])
   userId = line[0]
   movieId = line[1]
   rating = line[2]
   timestamp = line[3]
   ratings.append({
     'userId': userId,
     'movieId': movieId,
     'rating': rating,
     'timestamp': timestamp,
   })
 return ratings

def outputRatings(ratings):
 print("Writing ratings...")
 with open('./out/ratings.txt', 'w') as f:
   for rating in ratings:

f.write(f"{rating['userId']}{DELIM2}{rating['movieId']}{DELIM2}{rating['rating']}{DELI
M2}{rating['timestamp']}\n")
```

Tags.py

```
from util import readFile
from constants import DELIM1, DELIM2
from users import Users

def parseTags():
 print("Parsing tags...")
 tags = []
 tagsFile = readFile("./movies/tags.txt")
 for line in tagsFile.split("\n"):
   if line == '':
     continue
   line = line.split(":")
   userId = line[0]
   Users.addToUsers(userId)
   movieId = line[1]
   timestamp = line[-1]
   tag = ':'.join(line[2:-1])
   tags.append({
     'userId': userId,
     'movieId': movieId,
     'timestamp': timestamp,
     'tag': tag
   })
 return tags

def validateTags(tags):
 print("Validating tags...")
 invalidTags = []
 for line in tags:
   if len(line) != 4:
     print("Invalid tag:", line)
     invalidTags.append(line)
 return invalidTags

def outputTags(tags):
 print("Writing tags...")
 with open("./out/tags.txt", "w") as f:
   for line in tags:

f.write(f"{line['userId']}{DELIM2}{line['movieId']}{DELIM2}{line['tag']}{DELIM2}{line[
'timestamp']}\n")
```

Util.py

```
def readFile(file):
 with open(file, 'r') as f:
   return f.read()
```

Users.py

```
from util import readFile
from constants import DELIM1, DELIM2



class Users:
   _users = list()

   @classmethod
   def addToUsers(cls, userId):
       cls._users.append(userId)

   @classmethod
   def __sortUsers(cls):
       print("Sorting users...")
       cls._users = list(set(cls._users))
       cls._users = [int(userId) for userId in cls._users]
       cls._users.sort()

   @classmethod
   def outputUsers(cls):
       cls.__sortUsers()
       print("Writing users...")
       with open("./out/users.txt", "w") as f:
           for user in cls._users:
               f.write(f"{user}\n")
```

Run.sh

```
#!/bin/bash

# This script is used to run the application.

# Check for dependencies
```

```
echo "Checking dependencies..."

# Check if wget is installed
which wget >/dev/null
if [ $? -ne 0 ]; then
 echo "wget is not installed. Please install it and try again."
 exit 1
fi

# Check if python3 is installed
which python3 >/dev/null
if [ $? -ne 0 ]; then
 echo "python3 is not installed. Please install it and try again."
 exit 1
fi

# Check if pip3 is installed
which pip3 >/dev/null
if [ $? -ne 0 ]; then
 echo "pip3 is not installed. Please install it and try again."
 exit 1
fi

# Check if wget is installed in pip3
pip3 show wget >/dev/null
if [ $? -ne 0 ]; then
 pip3 install wget
 echo "wget is not installed in pip3. Please install it and try again."
 exit 1
fi

# Check if postgres is installed
which psql >/dev/null
if [ $? -ne 0 ]; then
 echo "postgres is not installed. Please install it and try again."
 exit 1
fi

echo "No dependencies missing, continuing..."
echo ""
echo "Running files program..."
echo ""
```

```
python3 files.py

echo "Finished running files program..."
echo ""

echo "Checking for database ..."
echo ""

psql -U postgres -c "SELECT 1 FROM pg_database WHERE datname = 'moviesdb'" | grep -q 1
if [ $? -ne 0 ]; then
 echo "Database does not exist. Creating database..."
 psql -U postgres -c "CREATE DATABASE moviesdb;"
 echo "Database created."
 echo ""
else
 echo "Database exists."
 echo ""
fi

echo "Running database script..."
echo ""
psql moviesdb < db.sql

echo "Finished running database script..."
echo ""

echo "Done..."
```

Section B: Database Testing

A. List your tables:

```
moviesdb=# \d
              List of relations
 Schema |     Name      |   Type   |   Owner
--------+---------------+----------+------------
 public | genres        | table    | edwardrees
 public | has_genre     | table    | edwardrees
 public | movies        | table    | edwardrees
 public | movies_id_seq | sequence | edwardrees
```

```
 public | ratings       | table    | edwardrees
 public | tags          | table    | edwardrees
 public | users         | table    | edwardrees
 public | users_id_seq  | sequence | edwardrees
(8 rows)
```

B. Data types of your tables

```
moviesdb=# \d genres
              Table "public.genres"
   Column   | Type | Collation | Nullable | Default
------------+------+-----------+----------+---------
 genretitle | text |           | not null |
```

```
moviesdb=# \d movies
                        Table "public.movies"
 Column |  Type   | Collation | Nullable |              Default
--------+---------+-----------+----------+-----------------------------------
 id     | integer |           | not null | nextval('movies_id_seq'::regclass)
 title  | text    |           | not null |
 year   | integer |           | not null |
Indexes:
    "movies_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "genres" CONSTRAINT "genres_movieid_fkey" FOREIGN KEY (movieid)
REFERENCES movies(id)
    TABLE "ratings" CONSTRAINT "ratings_movieid_fkey" FOREIGN KEY (movieid)
REFERENCES movies(id)
    TABLE "tags" CONSTRAINT "tags_movieid_fkey" FOREIGN KEY (movieid) REFERENCES
movies(id)
```

```
moviesdb=# \d ratings
                   Table "public.ratings"
   Column   |       Type       | Collation | Nullable | Default
------------+------------------+-----------+----------+---------
 movieid    | integer          |           | not null |
 userid     | integer          |           | not null |
 rating     | double precision |           | not null |
 ratingtime | integer          |           |          |
Foreign-key constraints:
    "ratings_movieid_fkey" FOREIGN KEY (movieid) REFERENCES movies(id)
```

```
    "ratings_userid_fkey" FOREIGN KEY (userid) REFERENCES users(id)

moviesdb=# \d tags
                Table "public.tags"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 movieid | integer |           | not null |
 userid  | integer |           | not null |
 tag     | text    |           | not null |
 tagtime | integer |           |          |
Foreign-key constraints:
    "tags_movieid_fkey" FOREIGN KEY (movieid) REFERENCES movies(id)
    "tags_userid_fkey" FOREIGN KEY (userid) REFERENCES users(id)

moviesdb=# \d has_genre
              Table "public.has_genre"
   Column   |  Type   | Collation | Nullable | Default
------------+---------+-----------+----------+---------
 movieid    | integer |           | not null |
 genretitle | text    |           | not null |


C. Size of your tables
moviesdb=# select count(*) from genres;
 count
-------
    20
(1 row)
moviesdb=# select count(*) from movies;
 count
-------
 10681
(1 row)

moviesdb=# select count(*) from ratings;
  count
----------
 10000054
(1 row)

moviesdb=# select count(*) from tags;
 count
-------
 95580
```

```
(1 row)

moviesdb=# select count(*) from users;
 count
-------
 71567
(1 row)

moviesdb=# select count(*) from has_genre;
 count
-------
 21564
(1 row)
```

D. Data Values

```
moviesdb=# select * from movies limit 5;
 id |            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)

moviesdb=# select count(title) from movies;
 count
-------
 10681
(1 row)

moviesdb=# select * from movies order by year desc limit 5;
  id   |                    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)

moviesdb=# select * from movies order by year limit 5;
  id   |            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)

moviesdb=# select count(year) from movies;
 count
-------
 10681
(1 row)

moviesdb=# select count(year) from movies where year > 1500;
 count
-------
 10681
(1 row)


moviesdb=# select count(movieid) from has_genre where genretitle='(no genre listed)';
 count
-------
     0
(1 row)
```

E. Extra queries

```
1. moviesdb=# select * from movies where not (movies is not null);
 id | title | year
----+-------+------
(0 rows)

moviesdb=# select * from ratings where not (ratings is not null);
 movieid | userid | rating | ratingtime
---------+--------+--------+------------
(0 rows)

moviesdb=# select * from tags where not (tags is not null);
 movieid | userid | tag | tagtime
---------+--------+-----+---------
(0 rows)

moviesdb=# select * from users where not (users is not null);
```

```
 id
----
(0 rows)

moviesdb=# select * from genres where not (genres is not null);
 genretitle
------------
(0 rows)
```

2. Select year, count(title) from movies group by year order by year asc;

```
 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)
```

3. moviesdb=# WITH series AS (SELECT generate_series(1910, 2000, 10) AS r_from), range AS (SELECT r_from, (r_from + 9) AS r_to FROM series) SELECT r_from as decade, (SELECT count(title) FROM movies WHERE year BETWEEN r_from AND r_to) as count FROM range;

```
 decade | count
--------+-------
   1910 |    11
   1920 |    83
   1930 |   230
   1940 |   379
   1950 |   521
   1960 |   690
   1970 |   784
   1980 |  1712
   1990 |  3022
   2000 |  3249
(10 rows)
```

4. moviesdb=# select genretitle as genre, count(movieid) from has_genre group by genre;

```
      genre        | 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)
```

5. moviesdb=# select rating, count(movieid) from ratings group by rating order by rating asc;

```
 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)
```

6. Find movies that have:

i. No tags, but have ratings

SELECT count(*) FROM movies WHERE id IN (SELECT DISTINCT movieid FROM ratings) AND id NOT IN (SELECT DISTINCT movieid FROM tags);

```
 count
-------
  3080
(1 row)
```

ii. no ratings, but have tags

```
moviesdb=# select count(*) from movies where id in (select distinct movieid from tags) and id
not in (select distinct movieid from ratings);
 count
-------
     4
(1 row)
```

iii. No tags and no ratings

```
SELECT count(*) FROM movies WHERE id NOT IN (SELECT DISTINCT movieid FROM tags)
AND id NOT IN (SELECT DISTINCT movieid FROM ratings);
 count
-------
     0
(1 row)
```

iv. both tags and ratings

```
SELECT count(*) FROM movies WHERE id IN (SELECT DISTINCT movieid FROM tags) AND
id IN (SELECT DISTINCT movieid FROM ratings); count
-------
  7597
(1 row)
```

Chapter 4: Query the Database and Optimize the Queries

Code: https://github.com/EdwardRees/Movie-Database/blob/main/erees-code-phase3.sql

Section A: General Queries

```
1. SELECT m.movieid,
  m.title,
  COUNT(r.userid)
FROM movies m,
  ratings r
WHERE m.movieid = r.movieid
GROUP BY m.movieid,
  m.title
ORDER BY COUNT(r.userid) DESC
LIMIT 1;
 movieid |     title     | count
---------+---------------+-------
     296 | Pulp Fiction  | 34864
(1 row)


2. SELECT m.movieid,
  m.title,
  COUNT(r.rating)
FROM movies m,
  ratings r
WHERE m.movieid = r.movieid
  AND r.rating = 5
GROUP BY m.movieid,
  m.title
ORDER BY COUNT(r.rating) DESC
LIMIT 1;

 movieid |          title           | count
---------+--------------------------+-------
     318 | Shawshank Redemption  The  | 16460
(1 row)

3. SELECT COUNT(h1.movieid)
FROM has_genre h1,
  (
```

```
    SELECT movieid,
      COUNT(genre)
    FROM has_genre
    GROUP BY movieid
    HAVING COUNT(genre) > 4
  ) h2
WHERE h1.movieid = h2.movieid;

 count
-------
  1163
(1 row)

4. SELECT genre,
  COUNT(movieid)
FROM has_genre
GROUP BY genre
ORDER BY COUNT(movieid) DESC
LIMIT 1;

 genre | count
-------+-------
 Drama |  5339
(1 row)

5.
A. SELECT g.genre,
  g.high,
  g.low
FROM (
    (
      SELECT has_genre.genre,
        COUNT(ratings.rating) AS high
      FROM has_genre
        NATURAL JOIN ratings
      WHERE ratings.rating >= 4.0
      GROUP BY has_genre.genre
    ) g1
    NATURAL JOIN (
      SELECT has_genre.genre,
        COUNT(ratings.rating) AS low
      FROM has_genre
        NATURAL JOIN ratings
      WHERE ratings.rating < 4.0
```

```
      GROUP BY has_genre.genre
    ) g2
  ) g
ORDER BY g.high DESC;

    genre    |  high   |   low
-------------+---------+---------
 Drama       | 2455297 | 1888901
 Comedy      | 1847429 | 2086639
 Action      | 1300608 | 1544741
 Thriller    | 1273266 | 1311169
 Adventure   | 1031661 | 1089413
 Romance     |  977944 |  923939
 Crime       |  826375 |  648582
 Sci-Fi      |  673728 |  816761
 Fantasy     |  505891 |  522591
 Children    |  380262 |  439887
 Mystery     |  356799 |  274145
 War         |  348331 |  219732
 Horror      |  319514 |  448711
 Animation   |  275590 |  243522
 Musical     |  250613 |  230561
 Western     |  108365 |  102094
 Film-Noir   |   94675 |   36917
 Documentary |   64986 |   38468
 IMAX        |    5501 |    3579
(19 rows)



B.
SELECT g.genre,
  g.high,
  g.low
FROM (
    (
      SELECT has_genre.genre,
        COUNT(movies.year) AS high
      FROM has_genre
        NATURAL JOIN movies
      WHERE movies.year >= 2000
      GROUP BY has_genre.genre
    ) g1
    NATURAL JOIN (
```

```
      SELECT has_genre.genre,
        COUNT(movies.year) AS low
      FROM has_genre
        NATURAL JOIN movies
      WHERE movies.year < 2000
      GROUP BY has_genre.genre
    ) g2
  ) g
ORDER BY high DESC;

    genre    | high | low
-------------+------+------
 Drama       | 1758 | 3581
 Comedy      | 1220 | 2483
 Thriller    |  660 | 1046
 Romance     |  585 | 1100
 Action      |  498 |  975
 Crime       |  393 |  725
 Adventure   |  325 |  700
 Horror      |  263 |  750
 Documentary |  252 |  230
 Fantasy     |  207 |  336
 Sci-Fi      |  195 |  559
 Mystery     |  171 |  338
 Children    |  170 |  358
 Animation   |  129 |  157
 War         |  123 |  388
 Musical     |   99 |  337
 Western     |   31 |  244
 Film-Noir   |   15 |  133
 IMAX        |   12 |   17
(19 rows)
```

C.

A1. Without index:

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------
 Sort  (cost=457290.76..457290.80 rows=19 width=23) (actual time=4662.108..4662.893
rows=19 loops=1)
   Sort Key: (count(ratings.rating)) DESC
   Sort Method: quicksort  Memory: 26kB
   ->  Merge Join  (cost=457280.06..457290.35 rows=19 width=23) (actual
time=4662.061..4662.870 rows=19 loops=1)
```

```
      Merge Cond: (has_genre.genre = has_genre_1.genre)
      ->  Finalize GroupAggregate  (cost=230091.01..230095.82 rows=19 width=15) (actual
time=3477.896..3477.945 rows=19 loops=1)
            Group Key: has_genre.genre
            ->  Gather Merge  (cost=230091.01..230095.44 rows=38 width=15) (actual
time=3477.890..3477.935 rows=57 loops=1)
                  Workers Planned: 2
                  Workers Launched: 2
                  ->  Sort  (cost=229090.99..229091.03 rows=19 width=15) (actual
time=3469.654..3469.656 rows=19 loops=3)
                        Sort Key: has_genre.genre
                        Sort Method: quicksort  Memory: 26kB
                        Worker 0:  Sort Method: quicksort  Memory: 26kB
                        Worker 1:  Sort Method: quicksort  Memory: 26kB
                        ->  Partial HashAggregate  (cost=229090.39..229090.58 rows=19 width=15)
(actual time=3469.620..3469.622 rows=19 loops=3)
                              Group Key: has_genre.genre
                              Batches: 1  Memory Usage: 24kB
                              Worker 0:  Batches: 1  Memory Usage: 24kB
                              Worker 1:  Batches: 1  Memory Usage: 24kB
                              ->  Hash Join  (cost=603.17..207106.32 rows=4396815 width=15) (actual
time=10.621..3030.053 rows=4384484 loops=3)
                                    Hash Cond: (ratings.movieid = has_genre.movieid)
                                    ->  Parallel Seq Scan on ratings  (cost=0.00..135970.25 rows=2125180
width=13) (actual time=0.945..2310.731 rows=1675267 loops=3)
                                          Filter: (rating >= '4'::double precision)
                                          Rows Removed by Filter: 1658084
                                    ->  Hash  (cost=333.63..333.63 rows=21563 width=12) (actual
time=9.540..9.540 rows=21563 loops=3)
                                          Buckets: 32768  Batches: 1  Memory Usage: 1200kB
                                          ->  Seq Scan on has_genre  (cost=0.00..333.63 rows=21563
width=12) (actual time=0.104..3.418 rows=21563 loops=3)
      ->  Finalize GroupAggregate  (cost=227189.05..227193.86 rows=19 width=15) (actual
time=1184.162..1184.919 rows=19 loops=1)
            Group Key: has_genre_1.genre
            ->  Gather Merge  (cost=227189.05..227193.48 rows=38 width=15) (actual
time=1184.158..1184.912 rows=57 loops=1)
                  Workers Planned: 2
                  Workers Launched: 2
                  ->  Sort  (cost=226189.03..226189.07 rows=19 width=15) (actual
time=1182.575..1182.576 rows=19 loops=3)
                        Sort Key: has_genre_1.genre
                        Sort Method: quicksort  Memory: 26kB
                        Worker 0:  Sort Method: quicksort  Memory: 26kB
```

```
                  Worker 1:  Sort Method: quicksort  Memory: 26kB
                  ->  Partial HashAggregate  (cost=226188.43..226188.62 rows=19 width=15)
(actual time=1182.552..1182.554 rows=19 loops=3)
                        Group Key: has_genre_1.genre
                        Batches: 1  Memory Usage: 24kB
                        Worker 0:  Batches: 1  Memory Usage: 24kB
                        Worker 1:  Batches: 1  Memory Usage: 24kB
                        ->  Hash Join  (cost=603.17..204893.93 rows=4258901 width=15) (actual
time=3.336..793.186 rows=4271245 loops=3)
                              Hash Cond: (ratings_1.movieid = has_genre_1.movieid)
                              ->  Parallel Seq Scan on ratings ratings_1  (cost=0.00..135970.25
rows=2058520 width=13) (actual time=0.024..253.448 rows=1658084 loops=3)
                                    Filter: (rating < '4'::double precision)
                                    Rows Removed by Filter: 1675267
                              ->  Hash  (cost=333.63..333.63 rows=21563 width=12) (actual
time=3.266..3.266 rows=21563 loops=3)
                                    Buckets: 32768  Batches: 1  Memory Usage: 1200kB
                                    ->  Seq Scan on has_genre has_genre_1  (cost=0.00..333.63
rows=21563 width=12) (actual time=0.013..1.089 rows=21563 loops=3)
 Planning Time: 5.142 ms
 Execution Time: 4663.690 ms
```

A2. With index:

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------
 Sort  (cost=456124.93..456124.97 rows=19 width=23) (actual time=2388.458..2389.117
rows=19 loops=1)
   Sort Key: (count(ratings.rating)) DESC
   Sort Method: quicksort  Memory: 26kB
   ->  Merge Join  (cost=456114.23..456124.52 rows=19 width=23) (actual
time=2388.420..2389.104 rows=19 loops=1)
         Merge Cond: (has_genre.genre = has_genre_1.genre)
         ->  Finalize GroupAggregate  (cost=229502.20..229507.01 rows=19 width=15) (actual
time=1224.294..1224.336 rows=19 loops=1)
               Group Key: has_genre.genre
               ->  Gather Merge  (cost=229502.20..229506.63 rows=38 width=15) (actual
time=1224.289..1224.326 rows=57 loops=1)
                     Workers Planned: 2
                     Workers Launched: 2
                     ->  Sort  (cost=228502.17..228502.22 rows=19 width=15) (actual
time=1215.270..1215.271 rows=19 loops=3)
                           Sort Key: has_genre.genre
```

```
                    Sort Method: quicksort  Memory: 26kB
                    Worker 0:  Sort Method: quicksort  Memory: 26kB
                    Worker 1:  Sort Method: quicksort  Memory: 26kB
                    ->  Partial HashAggregate  (cost=228501.58..228501.77 rows=19 width=15)
(actual time=1215.232..1215.234 rows=19 loops=3)
                          Group Key: has_genre.genre
                          Batches: 1  Memory Usage: 24kB
                          Worker 0:  Batches: 1  Memory Usage: 24kB
                          Worker 1:  Batches: 1  Memory Usage: 24kB
                          ->  Hash Join  (cost=603.17..206606.89 rows=4378938 width=15) (actual
time=12.570..810.140 rows=4384484 loops=3)
                                Hash Cond: (ratings.movieid = has_genre.movieid)
                                ->  Parallel Seq Scan on ratings  (cost=0.00..135757.61 rows=2116539
width=13) (actual time=0.061..253.167 rows=1675267 loops=3)
                                      Filter: (rating >= '4'::double precision)
                                      Rows Removed by Filter: 1658084
                                ->  Hash  (cost=333.63..333.63 rows=21563 width=12) (actual
time=12.348..12.348 rows=21563 loops=3)
                                      Buckets: 32768  Batches: 1  Memory Usage: 1200kB
                                      ->  Seq Scan on has_genre  (cost=0.00..333.63 rows=21563
width=12) (actual time=0.034..3.421 rows=21563 loops=3)
        ->  Finalize GroupAggregate  (cost=226612.03..226616.85 rows=19 width=15) (actual
time=1164.122..1164.762 rows=19 loops=1)
              Group Key: has_genre_1.genre
              ->  Gather Merge  (cost=226612.03..226616.47 rows=38 width=15) (actual
time=1164.119..1164.755 rows=57 loops=1)
                    Workers Planned: 2
                    Workers Launched: 2
                    ->  Sort  (cost=225612.01..225612.06 rows=19 width=15) (actual
time=1162.297..1162.298 rows=19 loops=3)
                          Sort Key: has_genre_1.genre
                          Sort Method: quicksort  Memory: 26kB
                          Worker 0:  Sort Method: quicksort  Memory: 26kB
                          Worker 1:  Sort Method: quicksort  Memory: 26kB
                          ->  Partial HashAggregate  (cost=225611.42..225611.61 rows=19 width=15)
(actual time=1162.271..1162.272 rows=19 loops=3)
                                Group Key: has_genre_1.genre
                                Batches: 1  Memory Usage: 24kB
                                Worker 0:  Batches: 1  Memory Usage: 24kB
                                Worker 1:  Batches: 1  Memory Usage: 24kB
                                ->  Hash Join  (cost=603.17..204403.50 rows=4241584 width=15) (actual
time=3.920..779.332 rows=4271245 loops=3)
                                      Hash Cond: (ratings_1.movieid = has_genre_1.movieid)
```

```
                        ->  Parallel Seq Scan on ratings ratings_1  (cost=0.00..135757.61
rows=2050150 width=13) (actual time=0.035..248.326 rows=1658084 loops=3)
                              Filter: (rating < '4'::double precision)
                              Rows Removed by Filter: 1675267
                        ->  Hash  (cost=333.63..333.63 rows=21563 width=12) (actual
time=3.825..3.825 rows=21563 loops=3)
                              Buckets: 32768  Batches: 1  Memory Usage: 1200kB
                              ->  Seq Scan on has_genre has_genre_1  (cost=0.00..333.63
rows=21563 width=12) (actual time=0.015..1.184 rows=21563 loops=3)
 Planning Time: 1.250 ms
 Execution Time: 2389.711 ms
```

Analysis:
I noticed that with the index, the planning time and execution time were about halved. While I don't see it explicitlyshown in the Explain Analyze portion, seeing as to how the planning time went down that much and execution time too, I expect that the index was used.

B1. Without index:

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
-----------------------------
 Sort  (cost=1720.43..1720.47 rows=19 width=23) (actual time=33.158..33.192 rows=19
loops=1)
   Sort Key: (count(movies.year)) DESC
   Sort Method: quicksort  Memory: 26kB
   ->  Hash Join  (cost=1719.58..1720.02 rows=19 width=23) (actual time=33.135..33.174
rows=19 loops=1)
         Hash Cond: (has_genre.genre = g2.genre)
         ->  HashAggregate  (cost=770.00..770.19 rows=19 width=15) (actual time=17.965..17.998
rows=19 loops=1)
               Group Key: has_genre.genre
               Batches: 1  Memory Usage: 24kB
               ->  Hash Join  (cost=257.12..737.21 rows=6559 width=12) (actual time=6.158..15.430
rows=7106 loops=1)
                     Hash Cond: (has_genre.movieid = movies.movieid)
                     ->  Seq Scan on has_genre  (cost=0.00..333.63 rows=21563 width=12) (actual
time=0.013..3.575 rows=21563 loops=1)
                     ->  Hash  (cost=216.51..216.51 rows=3249 width=10) (actual time=3.877..3.877
rows=3249 loops=1)
                           Buckets: 4096  Batches: 1  Memory Usage: 169kB
```

```
                    ->  Seq Scan on movies  (cost=0.00..216.51 rows=3249 width=10) (actual
time=0.629..2.747 rows=3249 loops=1)
                          Filter: (year >= '2000'::numeric)
                          Rows Removed by Filter: 7432
        ->  Hash  (cost=949.34..949.34 rows=19 width=15) (actual time=15.144..15.146 rows=19
loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              ->  Subquery Scan on g2  (cost=948.96..949.34 rows=19 width=15) (actual
time=15.126..15.132 rows=19 loops=1)
                    ->  HashAggregate  (cost=948.96..949.15 rows=19 width=15) (actual
time=15.123..15.127 rows=19 loops=1)
                          Group Key: has_genre_1.genre
                          Batches: 1  Memory Usage: 24kB
                          ->  Hash Join  (cost=309.41..873.94 rows=15004 width=12) (actual
time=4.393..12.442 rows=14457 loops=1)
                                Hash Cond: (has_genre_1.movieid = movies_1.movieid)
                                ->  Seq Scan on has_genre has_genre_1  (cost=0.00..333.63 rows=21563
width=12) (actual time=0.030..1.500 rows=21563 loops=1)
                                ->  Hash  (cost=216.51..216.51 rows=7432 width=10) (actual
time=4.343..4.343 rows=7432 loops=1)
                                      Buckets: 8192  Batches: 1  Memory Usage: 371kB
                                      ->  Seq Scan on movies movies_1  (cost=0.00..216.51 rows=7432
width=10) (actual time=0.013..2.460 rows=7432 loops=1)
                                            Filter: (year < '2000'::numeric)
                                            Rows Removed by Filter: 3249

Planning Time: 0.745ms
Execution Time: 33.304ms
B2. With Index:
    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
------------------------------
 Sort  (cost=1692.99..1693.04 rows=19 width=23) (actual time=28.192..28.196 rows=19
loops=1)
   Sort Key: (count(movies.year)) DESC
   Sort Method: quicksort  Memory: 26kB
   ->  Hash Join  (cost=1692.15..1692.59 rows=19 width=23) (actual time=28.170..28.182
rows=19 loops=1)
         Hash Cond: (has_genre.genre = g2.genre)
         ->  HashAggregate  (cost=742.57..742.76 rows=19 width=15) (actual time=13.933..13.938
rows=19 loops=1)
               Group Key: has_genre.genre
               Batches: 1  Memory Usage: 24kB
```

```
            ->  Hash Join  (cost=229.69..709.77 rows=6559 width=12) (actual time=3.673..11.565
rows=7106 loops=1)
                  Hash Cond: (has_genre.movieid = movies.movieid)
                  ->  Seq Scan on has_genre  (cost=0.00..333.63 rows=21563 width=12) (actual
time=0.008..2.522 rows=21563 loops=1)
                  ->  Hash  (cost=189.08..189.08 rows=3249 width=10) (actual time=2.250..2.251
rows=3249 loops=1)
                        Buckets: 4096  Batches: 1  Memory Usage: 169kB
                        ->  Bitmap Heap Scan on movies  (cost=65.46..189.08 rows=3249 width=10)
(actual time=0.482..1.220 rows=3249 loops=1)
                              Recheck Cond: (year >= '2000'::numeric)
                              Heap Blocks: exact=59
                              ->  Bitmap Index Scan on iyear  (cost=0.00..64.65 rows=3249 width=0)
(actual time=0.468..0.468 rows=3249 loops=1)
                                    Index Cond: (year >= '2000'::numeric)
        ->  Hash  (cost=949.34..949.34 rows=19 width=15) (actual time=14.216..14.218 rows=19
loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              ->  Subquery Scan on g2  (cost=948.96..949.34 rows=19 width=15) (actual
time=14.195..14.202 rows=19 loops=1)
                    ->  HashAggregate  (cost=948.96..949.15 rows=19 width=15) (actual
time=14.194..14.198 rows=19 loops=1)
                          Group Key: has_genre_1.genre
                          Batches: 1  Memory Usage: 24kB
                          ->  Hash Join  (cost=309.41..873.94 rows=15004 width=12) (actual
time=4.873..11.922 rows=14457 loops=1)
                                Hash Cond: (has_genre_1.movieid = movies_1.movieid)
                                ->  Seq Scan on has_genre has_genre_1  (cost=0.00..333.63 rows=21563
width=12) (actual time=0.018..1.326 rows=21563 loops=1)
                                ->  Hash  (cost=216.51..216.51 rows=7432 width=10) (actual
time=4.825..4.826 rows=7432 loops=1)
                                      Buckets: 8192  Batches: 1  Memory Usage: 371kB
                                      ->  Seq Scan on movies movies_1  (cost=0.00..216.51 rows=7432
width=10) (actual time=0.008..2.875 rows=7432 loops=1)
                                            Filter: (year < '2000'::numeric)
                                            Rows Removed by Filter: 3249
 Planning Time: 1.420 ms
 Execution Time: 28.302 ms
```

Analysis: In this case, when I added the index, the planning time nearly doubled, but the execution time with shorter. I assume that the index on year was used as the planning time increased, meaning it would've spent more time looking up the index, before using it in the execution, decreasing the execution time.

Section B: De-bias user ratings

1) INSERT INTO ratings_with_diff
SELECT r.userid,
  r.movieid,
  r.rating,
  r.time,
  a.avg,
  r.rating - a.avg AS difference
FROM ratings r,
  (
    SELECT movieid,
      AVG(rating)
    FROM ratings
    GROUP BY movieid
) a
WHERE a.movieid = r.movieid
GROUP BY r.userid,
  r.movieid,
  r.rating,
  r.time,
  a.avg,
  r.rating - a.avg;

INSERT 0 10000054

2)
UPDATE ratings r
SET rating = rd.avg_rating,
  time = extract(
    epoch
    FROM current_timestamp at time zone ‘ utc ’
)
FROM (
    SELECT userid,
      movieid,
      avg_rating,
      difference
    FROM ratings_with_diff
) rd
WHERE rd.userid = r.userid
  AND rd.movieid = r.movieid
  AND abs(rd.difference) > 3;

```
UPDATE 40498

3) CREATE TABLE ratings_with_diff2 (
  userid numeric,
  movieid numeric,
  rating double precision,
  “time” numeric,
  avg_rating double precision,
  difference double precision
);

CREATE TABLE

INSERT INTO ratings_with_diff2
SELECT r.userid,
  r.movieid,
  r.rating,
  r.time,
  a.avg,
  r.rating - a.avg AS difference
FROM ratings r,
  (
    SELECT movieid,
      AVG(rating)
    FROM ratings
    GROUP BY movieid
  ) a
WHERE a.movieid = r.movieid
GROUP BY r.userid,
  r.movieid,
  r.rating,
  r.time,
  a.avg,
  r.rating - a.avg;

INSERT 0 10000054

4) UPDATE ratings r
SET rating = rd.avg_rating,
  time = extract(
    epoch
    FROM current_timestamp at time zone ‘ utc ’
  )
FROM (
```

```
    SELECT userid,
      movieid,
      avg_rating,
      difference
    FROM ratings_with_diff2
  ) rd
WHERE rd.userid = r.userid
  AND rd.movieid = r.movieid
  AND abs(rd.difference) > 3;

UPDATE 751

CREATE TABLE ratings_with_diff3 (
  userid numeric,
  movieid numeric,
  rating double precision,
  “ time ” numeric,
  avg_rating double precision,
  difference double precision
);

CREATE TABLE

INSERT INTO ratings_with_diff3
SELECT r.userid,
  r.movieid,
  r.rating,
  r.time,
  a.avg,
  r.rating - a.avg AS difference
FROM ratings r,
  (
    SELECT movieid,
      AVG(rating)
    FROM ratings
    GROUP BY movieid
  ) a
WHERE a.movieid = r.movieid
GROUP BY r.userid,
  r.movieid,
  r.rating,
  r.time,
  a.avg,
  r.rating - a.avg;
```

```
INSERT 0 10000054

UPDATE ratings r
SET rating = rd.avg_rating,
  time = extract(
    epoch
    FROM current_timestamp at time zone ‘ utc ’
  )
FROM (
    SELECT userid,
      movieid,
      avg_rating,
      difference
    FROM ratings_with_diff3
  ) rd
WHERE rd.userid = r.userid
  AND rd.movieid = r.movieid
  AND abs(rd.difference) > 3;

UPDATE 0

5) SELECT m.title,
  m.movieid,
  ratings.orig AS original_rating,
  ratings.debiased AS debiased_rating,
  ratings.debiased - ratings.orig bias
FROM movies m,
  (
    SELECT original.movieid,
      original.avg_rating AS orig,
      final.avg_rating AS debiased
    FROM ratings_with_diff original
      INNER JOIN ratings_with_diff3 final USING (userid, movieid)
  ) ratings
WHERE ratings.movieid = m.movieid
GROUP BY m.movieid,
  m.title,
  ratings.orig,
  ratings.debiased
ORDER BY ratings.debiased - ratings.orig DESC
LIMIT 10;
```

```
                                title                                | movieid |  original_rating   |  debiased_rating
|        bias
-------------------------------------------------------------------------------+---------+--------------------+-------------------
-----+--------------------
 Human Condition I  The (Ningen no joken I)                           |    8484 |            3.59375 |
4.173828125 |         0.580078125
 Bizarre  Bizarre (Drôle de drame ou L'étrange aventure de Docteur Molyneux)  |    6397 |
3.5625 |          3.9453125 |           0.3828125
 Kid Brother  The                                                    |    8423 | 3.5588235294117645 |
3.918685121107267 | 0.35986159169550236
 Holy Mountain  The (Montaña sagrada  La)                            |   26326 |               4.05 |
4.404999999999999 | 0.35499999999999954
 Samurai Rebellion (Jôi-uchi: Hairyô tsuma shimatsu)                 |   41627 |               4.05 |
4.404999999999999 | 0.35499999999999954
 Cruel Romance  A (Zhestokij Romans)                                 |    5889 |
3.5555555555555554 | 3.8950617283950617 | 0.33950617283950635
 Accattone                                                           |    6599 |  3.642857142857143 |
3.9795918367346936 | 0.33673469387755084
 Crowd  The                                                          |   25766 |  3.716666666666667 |
4.038333333333333 |  0.3216666666666663
 Odd Man Out                                                         |   25930 | 3.8863636363636362 |
4.194214876033058 | 0.30785123966942196
 Children Underground                                                |    4778 | 3.8636363636363638 |
4.169421487603306 | 0.30578512396694224
(10 rows)
```

Chapter 5: Discussion

When I was creating the E/R diagram in Chapter 2, I looked at the data and noticed that the genres had to be split up into its own category. I immediately noted that genres would have to be its own tables of sorts. I thought about has_genres as a relationship between genres and movies. In my original E/R diagram, this was the “has” relationship between genres and movies. After looking at the data in tags and ratings, I also noted that Users would have to be its own table too. In doing so, I started with the Movie, Genre, and Users entity sets. I then had to add the relationships of Ratings, Tags, and has_genre. For the attributes, it must have slipped my mind on including the *year* attribute on Movie in the E/R diagram, but I did have the in mind when I was creating the database later.

I don’t think I came across any constraints with the data and relationships. I think with the relationships, I realized earlier on that the Rating and Tag would use the keys from Movie and User. I also didn’t include any Weak Entity Sets as I didn’t think it would be necessary to include any. I saw that the genres could be on their own, with each movie having a relationship with the genre, which would connect the movies and genres together.

When testing my database, I did some problems with Problem 5 in Section A, especially with thinking through how to work through the problem. However, I was able to figure it out eventually after taking a walk and clearing my head. I realized I could join two relations that formed with their own joins. I do think this did have some redundancies as it required the count calculations to occur twice, but I couldn’t think of another way of approaching it that could work. I did have a couple of other approaches, but those approaches were unsuccessful as they led to the calculations being far too large and/or inefficient.

As I went through my data, I don’t believe I had any unknown values in my attributes. I certainly couldn’t find any unknowns or nulls in the second phase, and I don’t believe I came across any in the third phase.

Initially, I didn’t use any indexes, but after going to Problem 5 in Section A, I realized that using indexes could really decrease the amount of time it would take to execute a query. I didn’t use them as much as I would have liked, only because at times, I had to really think about what information I would need to pull and what information could be useful to index. I think ultimately, I didn’t use indexes as much as I wasn’t entirely comfortable with what would be an effective column to index in a given table. I do recognize the benefits of using an index though.

As aforementioned, I struggled with Problem 5 in Section A, namely calculating both the high and low values. Before using table joins, I was selecting from a has_genre and two ratings, having the count act on both ratings. This, however, ended up giving me extremely large numbers and counting values too many times. I then re-evaluated the problem and recognized that using Joins could help me solve my problem. I’m not sure about alternatives that could improve the run time asides from possibly indexing values and making better use of indexes.

In terms of challenges, I initially had a problem in Phase 2 when I parsed the files. I split the data incorrectly, without recognizing that a colon (“:”) could appear in a movie title, which caused a problem with how I was parsing and converting the file. After clocking up my memory and having an immense amount of swap space used, I cancelled the program, restarted my computer, and changed the code. After fixing this, the remainder of Phase 2 was simple for me.

For Phase 3, I think the most challenging part was thinking through some of the queries. I had to change how I thought of the problem, moving away from an imperative programming way to a more declarative programming way. However, once I was able to think through the problem and find a solution, I found the rest of Phase 3 quite straightforward.