The Movies Dataset

1 Problem Description

The selected dataset includes information for movies released before or on July 2017. The data was collected from TMDB and GroupLens and is a compilation of several csv files which can be found at https://www.kaggle.com/rounakbanik/the-movies-dataset. There are five main csv files containing data points for the credits (cast and crew member information), plot keywords, links, metadata, and ratings. The links file includes TMDB and IMDB IDs and the metadata file contains several attributes such as movie title, budget, revenue, release dates, and languages.

The objective will be to construct a database in PostgreSQL 11 with sensible relations that will facilitate data retrieval and searching. The database could be used for several purposes such as searching for movie recommendations, retrieving information about a movie, or doing analysis to answer questions such as what production companies yield the top rated or highest grossing films.

There have been several attempts at creating movie recommenders using the dataset in Python, which have been shared with the kaggle website. Various types of recommendation systems have been tested, such as making suggestions based on popularity, ratings, or content descriptions. We attempted to recreate 2 types of movie recommenders to determine whether PostgreSQL is as well equipped as Python or other tools to handle a movie recommender.

1.1 Data

As previously mentioned, the complete dataset is an ensemble of several csv files. Each movie has a unique ID, which is referenced throughout the different csv documents. The following csv fields were heavily used throughout our project:

- Movie ID: unique identifier
- Original Title
- Keywords: plot keywords available as a stringified JSON Object
- Cast: list of main cast members available as a stringified JSON Object
- Crew: list of main crew members (such as directors and writers) available as a stringified JSON Object
- Revenue
- Genres
- Collection: refers to a series of related movies such as a trilogy or a franchise
- Rating: ratings are made on a 5-star scale

1.2 Database System

The data will be loaded into Postgres database, version 11 due to the data having relations; we are also dealing with several M:M relationships. Additionally, our data includes

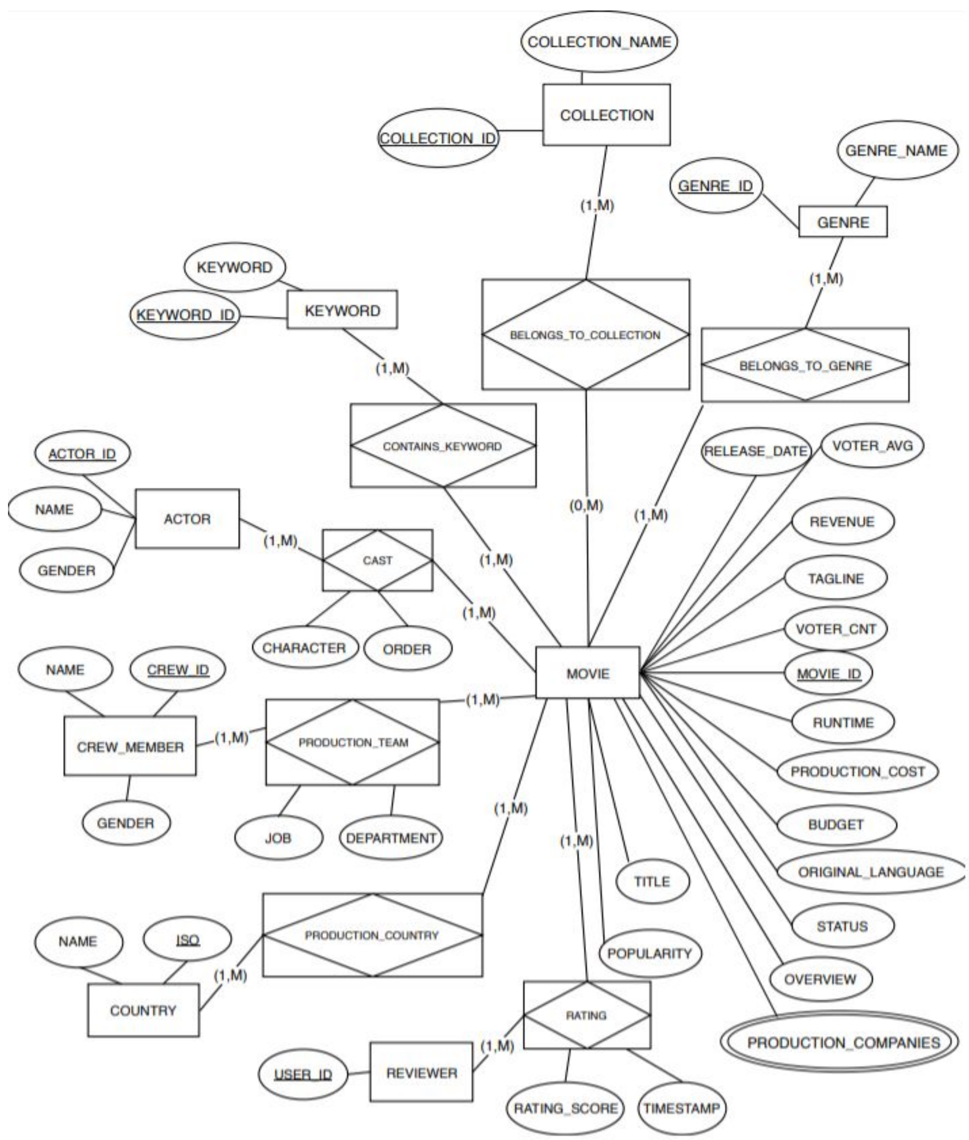
JSON objects. Since Postgres supports JSON data, we can apply general functions and queries to easily transform JSON objects into more manageable elements.

2 Team

The project will be performed by the following team members:

Isis Ramirez, Statistics Department, Professional Masters in Computational Science & Engineering
Siri Manjunath, Computer Science Department, Professional Masters in Computer Science
Xiaoye Zhang, Computer Science Department, Master of Computer Science

3 Design



ERD ENTITY DESCRIPTIONS:

- **MOVIE**(MOVIE_ID, TITLE, POPULARITY, BUDGET, REVENUE, RUNTIME, ...)
  The MOVIE entity is the central entity of our ERD, and holds various basic information about movies to help with analysis.
  - MOVIE_ID - A unique number to identify different movies
  - TITLE - The original title of a movie, could be in different languages
  - POPULARITY - A numeric popularity score
  - BUDGET - The budget of a movie with US dollar as unit
  - REVENUE - The revenue of a movie with US dollar as unit
  - RUNTIME - The runtime of a movie with minute as unit
  - STATUS - The current status of a movie, could be released, planned or cancelled
  - TAGLINE - The tagline of a movie
  - OVERVIEW - The overview of a movie
  - VOTER_AVERAGE - TMDB vote averages
  - VOTER_COUNT - TMDB vote counts
  - PRODUCTION_COMPANY - The production company of a movie
  - RELEASE_DATE - The data the movie was released
  - ORIGINAL_LANGUAGE - The original language of a movie

- **ACTOR**(ACTOR_ID, GENDER, NAME),
  The ACTOR entity holds information such as gender and name about actors and actresses that may appear in a movie.
- **CAST_IN_MOVIE**(ACTOR_ID, MOVIE_ID, CHARACTER, ORDER)
  The CAST associative entity relates the ACTOR table to the MOVIE table by mapping the actresses and actors cast in a particular movie. It lists the character played by the actor and the opening credit order of the actor.

- **CREW_MEMBER**(CREW_ID, GENDER, NAME)
  The CREW_MEMBER entity holds the gender and name about crew workers such as possible directors or writers a movie may have.
- **PRODUCTION_TEAM**(CREW_ID, MOVIE_ID, JOB, DEPARTMENT)
  The CREW_MEMBER table is associated to the MOVIE table using the PRODUCTION_TEAM associative entity. It maps crew members to a particular movie and lists the job and department of the crew member.

- **COUNTRY**(ISO, NAME)
  The COUNTRY table holds information about countries where production for a movie might have taken place in. ISO is a country code as identified by the International Organization for Standardization.
- **PRODUCTION_COUNTRY**(ISO, MOVIE_ID)
  The COUNTRY table is associated to the MOVIE table using the PRODUCTION_COUNTRY relationship. It maps movies to their production country or countries.

- **REVIEWER**(USER_ID)
  The REVIEWER table has records pertaining to IMDB users who have reviewed movies.
- **RATING**(USER_ID, MOVIE_ID, RATING_SCORE, TIMESTAMP)
  The REVIEWER table is associated to the MOVIE table using the RATING associative entity. It maps a reviewer, their rating, and a timestamp of when the rating was submitted to a particular movie.

- **KEYWORD**(KEYWORD_ID, KEYWORD)
  The Keyword entity is an associative entity and is related to the Movies entity. It represents keywords used to identify or search for a certain movie(s).
- **CONTAINS_KEYWORD**(MOVIE_ID, KEYWORD_ID)
  This entity maps out any keyword associated to a movie. It relates the KEYWORD and MOVIES tables.

- **GENRE**(GENRE_ID, GENRE_NAME)
  The GENRE associates with the MOVIE entity and represents which genre the movie is.
- **BELONGS_TO_GENRE**(MOVIE_ID, GENRE_ID)
  This relates the GENRE entity to the MOVIE entity. It maps movies to their associated genre or genres.

- **COLLECTION**(COLLECTION_ID, COLLECTION_NAME)
  This entity refers to sets of movies such as trilogies and sequels and is related to MOVIES.
- **BELONGS_TO_COLLECTION**(MOVIE_ID, COLLECTION_ID)
  This relates the COLLECTION entity to the MOVIE entity. It maps movies to collections they are part of.

Team Member Siri Manjunath took ownership of this task.

After loading the data into the appropriate tables, we have the following record count:

Table	Rows
Actor	43772
Keyword	8562
Movie	3784
Rating	413338
Cast_In_Movie	76072
Crew_Member	42404
Production_Team	90517

Country	84
Production_Country	5361
Reviewer	10608
Contains_Keyword	33834
Genre	20
Belongs_to_Genre	9515
Collection	458
Belongs_to_Collection	806

USE CASE:

As previously mentioned, the database could be used for several purposes such as searching for movie recommendations, retrieving information about a movie, or doing analysis to determine what genres or production companies produce the most popular or highly rated movies. This project focuses on creating 2 types of movie recommenders. We created a recommender based on ratings and a recommender based on content such as genre, cast, keywords, and directors.

4 Methods

Cleaning:

JSON objects in the data were first parsed and validated in Python using regular expressions before being loaded into the appropriate tables. Numeric and string data did not require any cleaning.

Loading

After the data was cleaned and prepared, the data was then loaded into the database using the COPY command in Postgres. The csv file contained the attributes for the ACTOR/CAST_IN_MOVIE and CREW_MEMBER/PRODUCTION_TEAM data in JSON objects and arrays; the CAST_IN_MOVIE and PRODUCTION_TEAM tables are associative entities mapping actors and crew members to the corresponding movies. All the actors and crew members for a movie were stored in an array as JSON objects.

Below is a simplification of the actor/cast data for 'Toy Story' (1995). Not all attributes and actors are show. The same format is seen in the crew member data.

[{"cast_id": 14, "character": "Woody (voice)" , "gender": 0, "name": "Tom Hanks",...}, {"cast_id": 16, "character": "Mr. Potato Head (voice)",

"gender": 0, "name": "Don Rickles", ...}, {"cast_id": 17, "character": "Slinky Dog (voice)" , "gender": 0, "id": 12899, "name": "Jim Varney",...}, ...]

The data for ACTOR/CAST_IN_MOVIE and CREW_MEMBER/PRODUCTION_TEAM was loaded to the dummy tables ACTOR_JSON and CREW_JSON as is. The desired data for individual actors and crew members was then extracted using JSON functions and operators and loaded into the target tables; the dummy tables were then discarded. The following JSON functions and operators were used:

- **<data>::json**
  - This operator transforms the inputted data into a valid JSON format.
- **json_array_elements(<array>)**
  - This function separates the JSON objects from the inputted array.
- **<json object> ->> <element>**
  - This operator retrieves the desired element as text from the inputted JSON object.

Below we see the code for loading the ACTOR and CAST_IN_MOVIE data into the target tables. The code for the CREW_MEMBER and PRODUCTION_TEAM tables is similar.

```
-- Insert into ACTOR (target table)
INSERT INTO ACTOR(ACTOR_ID, NAME, GENDER)
SELECT DISTINCT CAST(sub2.ACTOR_ID as INTEGER), sub2.NAME, sub2.GENDER
FROM
(SELECT sub.INDIVIDUALS::json ->> 'id' as ACTOR_ID,
sub.INDIVIDUALS::json ->> 'gender' as GENDER,
sub.INDIVIDUALS::json ->> 'name' as NAME
FROM
(SELECT json_array_elements(ALLDATA::json) as INDIVIDUALS
FROM ACTOR_JSON) sub) sub2;
```

	actor_id integer	gender character varying (10)	name character varying
1	54830	1	Sophia Bush
2	164938	1	Tara Karsian
3	60541	0	Abraham Boyd
4	49740	1	Daniela Holtz
5	1336845	0	Jeff Bailey
6	1405	2	Zbigniew Zapasiewicz

```sql
-- Insert into CAST_IN_MOVIE (target table)
INSERT INTO CAST_IN_MOVIE(ACTOR_ID, MOVIE_ID, CHARACTER, CREDIT_ORDER)
SELECT DISTINCT CAST(sub2.ACTOR_ID as INTEGER), CAST(sub2.MOVIE_ID as
INTEGER), sub2.CHARACTER, CAST(sub2.order as INTEGER)  FROM
(SELECT sub.INDIVIDUALS::json ->> 'id' as ACTOR_ID,
sub.INDIVIDUALS::json ->> 'character' as CHARACTER,
sub.INDIVIDUALS::json ->> 'order' as ORDER,
MOVIE_ID as MOVIE_ID
FROM
(SELECT MOVIE_ID, json_array_elements(ALLDATA::json) as INDIVIDUALS
FROM  ACTOR_JSON) sub) sub2;
```

	actor_id integer	movie_id integer	character character varying	credit_order integer
1	45213	5731	Caroline	5
2	146487	2334	Doctor	33
3	188452	197	Drinker #2	47
4	984489	1813	Fight Fan	33
5	17390	1540	Albert	4
6	1781334	559	Dog Walker (uncredited)	97
7	7085	1496	Roscoe	6

5 Deliverables

VIEW:

The field ‘production_companies’ in the movie table is a multi-valued attribute; several production companies may help fund a film. This field was presented as an array of JSON objects; each JSON object consists of a company name, and a company id. Below we see a subset of movie ID’s and the associated production companies from the MOVIE table.

movie_id integer	production_companies json
2	[{"name":"Villealfa Filmproduction Oy","id":2303},{"name":"Finnish Film Foundation","id":2396}]
3	[{"name":"Villealfa Filmproduction Oy","id":2303}]
5	[{"name":"Miramax Films","id":14},{"name":"A Band Apart","id":59}]
6	[{"name":"Universal Pictures","id":33},{"name":"Largo Entertainment","id":1644},{"name":"JVC Entertainment Networks","id":4248}]
11	[{"name":"Lucasfilm","id":1},{"name":"Twentieth Century Fox Film Corporation","id":306}]
12	[{"name":"Pixar Animation Studios","id":3}]
13	[{"name":"Paramount Pictures","id":4}]
14	[{"name":"DreamWorks SKG","id":27},{"name":"Jinks/Cohen Company","id":2721}]
15	[{"name":"RKO Radio Pictures","id":6},{"name":"Mercury Productions","id":11447}]
16	[{"name":"Fine Line Features","id":8},{"name":"Zentropa Entertainments","id":76},{"name":"Danmarks Radio (DR)","id":119},{"name":"SVT ...
17	[{"name":"Constantin Film","id":47},{"name":"Impact Pictures","id":248},{"name":"Isle of Man Film","id":2268},{"name":"UK Film Council","i...
18	[{"name":"Columbia Pictures","id":5},{"name":"Gaumont","id":9}]
19	[{"name":"Paramount Pictures","id":4},{"name":"Universum Film (UFA)","id":12372}]
20	[{"name":"El Deseo","id":49},{"name":"Milestone Productions","id":77}]
21	[{"name":"Bruce Brown Films","id":13723}]
22	[{"name":"Walt Disney Pictures","id":2},{"name":"Jerry Bruckheimer Films","id":130}]

A view was created to match movie id's to individual production companies; having a view of separated production companies will make analysis and searches by companies easier. Postgres provides support for json object and json arrays, which made the extraction of individual companies from the arrays painless. The json_array_elements() function was used to extract the individual json objects from the arrays (red subquery). Once the json objects were separated, the ->> operator was used to select individual fields from the json elements. In our case, we used the ->> operator to select either the company names or the company id's (green subquery).

```
DROP VIEW IF EXISTS MOVIE_PRODUCTION_COMPANY;
create view MOVIE_PRODUCTION_COMPANY as
select DISTINCT movie_id, allcomp.company_name, allcomp.comp_id from
-- get unique names, id's
(select movie_id,
       e ->> 'name' as company_name, -- get name, id from json
elements
       e ->> 'id' as comp_id from
(select movie_id, json_array_elements(production_companies) as e from
movie) as elements -- get json elements from arrays
) allcomp;
```

Below we show a subset of the view results:

	movie_id integer	company_name text	comp_id text
1	2	Finnish Film Foundation	2396
2	2	Villealfa Filmproduction Oy	2303
3	3	Villealfa Filmproduction Oy	2303
4	5	A Band Apart	59
5	5	Miramax Films	14
6	6	JVC Entertainment Networks	4248
7	6	Largo Entertainment	1644
8	6	Universal Pictures	33
9	11	Lucasfilm	1
10	11	Twentieth Century Fox Film Corporation	306
11	12	Pixar Animation Studios	3

Team member Isis Ramirez took ownership of this task.

QUERIES:

After the transformation of the 'production_companies' attribute and creation of the previous discussed view, analysis was performed.

Query 1:

The following query returns the 5 production companies that produced movies in the 'US' with the highest average revenue. The aggregate function 'AVG' in combination with 'GROUP BY' was used to determine the average revenue for all companies. In order to only return the top 5 companies, the results were first sorted and then limited. In order to filter by the production country, a join was performed with the the production_country table.

```
-- Top 5 Production Companies with HIGHEST AVERAGE REVENUE IN US
select UPPER(mpc.company_name) as COMPANY_NAME, CAST(avg(m.revenue) AS
INT) avgRevenue from MOVIE_PRODUCTION_COMPANY mpc
natural join movie m
natural join production_country pc
where pc.iso = 'US'
GROUP BY UPPER(mpc.company_name)
ORDER BY avgRevenue desc
LIMIT 5;
```

	company_name text	avgrevenue integer
1	SECOND MATE PR...	1013329906
2	THE SAUL ZAENTZ ...	898827882
3	PATALEX IV PRODU...	895921036
4	HEYDAY FILMS	856630081
5	LAURA ZISKIN PRO...	808951275

Query 2:

The following query returns the 5 production movies with the highest average revenue along with their highest grossing film and the film's revenue. Subqueries were used to return only the highest grossing production companies (in red) and to match the highest film revenue (in green).

```
-- Top 5 Companies with HIGHEST AVERAGE REVENUE w/ highest grossing
movie
select UPPER(mpc.company_name) as production_company, m.original_title
as highest_grossing_movie, m.revenue as highest_rev from
MOVIE_PRODUCTION_COMPANY mpc
natural join movie m
where UPPER(mpc.company_name) in
( -- Match Top 5 High Revenue Production Companies
select UPPER(mpc_sub.company_name) from MOVIE_PRODUCTION_COMPANY
mpc_sub
natural join movie m_sub
GROUP BY UPPER(mpc_sub.company_name)
ORDER BY CAST(avg(m_sub.revenue) AS INT) desc
LIMIT 5) and
m.revenue =
( -- Match highest revenue
select max(m_sub2.revenue) from MOVIE_PRODUCTION_COMPANY mpc_sub2
natural join movie m_sub2
where UPPER(mpc_sub2.company_name)=UPPER(mpc.company_name))
ORDER BY m.revenue desc;
```

	production_company text	highest_grossing_movie text	highest_rev integer
1	SECOND MATE PRODUC...	Pirates of the Caribbean: Dead Man's Chest	1065659812
2	HEYDAY FILMS	Harry Potter and the Philosopher's Stone	976475550
3	THE SAUL ZAENTZ COMP...	The Lord of the Rings: The Two Towers	926287400
4	PATALEX IV PRODUCTIO...	Harry Potter and the Goblet of Fire	895921036
5	LAURA ZISKIN PRODUCTI...	Spider-Man 3	890871626

Team member Isis Ramirez took ownership of this task.

TRIGGER:

We create a trigger to update the corresponding vote_count and vote_average value, which are averages and counts acquired from TMDB, in the movie table once new IMDB rating records are inserted. Since users' IMDB rating scores range from 0 to 5 and the vote_average values taken from TMDB range from 0 to 10, we use 2 to multiply the IMDB rating score to compute and update the average score in movie table.

```
create or replace function updateMovie()
returns trigger as
$$
declare
	newcount integer;
	newavg numeric;
begin
	select vote_count+1, (vote_count*vote_average*1.0 +
new.rating_score)*2/(vote_count+1)*1.0 into newcount, newavg
	from movie
	where movie_id = new.movie_id;

	update movie
	set vote_average = round(newavg,1), vote_count = newcount
	where movie_id = new.movie_id;
```

```
	raise notice 'new.movie_id:%, newcnt:%, newavg:%',
new.movie_id, newcount, newavg;
	return new;
end;
$$
language plpgsql;
```

Team member Xiaoye Zhang and Siri Manjunath took ownership of this task.

FUNCTIONS:

Two different movie recommenders were tested. Inspiration was taken from movie recommenders coded in Python from kaggle users. The first recommender takes into account ratings, while the second recommender considers movie metadata to make more personally attuned suggestions.

Weighted Rating Recommender:

The weighted rating[1] recommender takes a specific movie as input and will recommend several related movies with the highest weighted rating score based on similar genre or other options. The weighted rating is defined as follows:

$$\text{Weighted Rating} = \left(\frac{v}{v+m} * \mathrm{R}\right) + \left(\frac{m}{v+m} * C\right)$$

- v - the number of votes for the movie
- m - the minimum votes required to be considered
- R - the average rating score of the movie
- C - the mean vote rating across whole dataset

The recommender function requires 3 parameters: the title of chosen movie, the minimum votes, and user preferences.For the second parameter, we will use 90% as our cutoff, filter 10% movies having insufficient vote number and keep the remaining 90% movies as recommendation pool. According to the dataset, we will set m to 1800.

The function will first calculate the C value, calculate and store the weighted rating score among the remaining movies by creating a new table.

```
/* calc C among whole dataset */
select avg(vote_average) into C from movie;
/* calc the weighted rating score for each valid record */
create table wrating as
```

```
select movie_id, vote_count as v, vote_average as R,
round((vote_count*1.0/(vote_count+min_vote)*vote_average) +
(min_vote*1.0/(vote_count+min_vote)*C), 4) as WR
from movie
where vote_count > min_vote;
```

Then return different queries depending on users' choices. For example, if user find similar movies having the same actors as the specific movie. We will consider the main actors according to credit_order, instead of whole cast:

```
select distinct original_title, wr
from wrating natural join movie natural join cast_in_movie natural
join actor
where name in (select name from movie natural join cast_in_movie
natural join actor where original_title = title and credit_order <
10) and original_title != title
order by wr desc;
```

The WR recommender provide 5 options for user preference:

1. 'genre':

```
select * from WR_Recommender('Léon', 1800, 'genre') limit 10;
```

	movie_title text	weightedrating numeric
1	The Shawshank Redemption	8.1301
2	The Dark Knight	8.0585
3	The Godfather	8.0198
4	Fight Club	8.0040
5	Pulp Fiction	7.9755
6	Forrest Gump	7.8765
7	Schindler's List	7.7552
8	Se7en	7.7063
9	La vita è bella	7.6758
10	The Green Mile	7.6607

2. ‘collection’

```
select * from WR_Recommender('Harry Potter and the Half-Blood
Prince', 1800, 'collection');
```

	movie_title text	weightedrating numeric
1	Harry Potter and the Prisoner of Azkaban	7.4043
2	Harry Potter and the Philosopher's Stone	7.2822
3	Harry Potter and the Goblet of Fire	7.2410
4	Harry Potter and the Chamber of Secrets	7.1711
5	Harry Potter and the Order of the Phoenix	7.1609

3. ‘keyword’

```
select * from WR_Recommender('Batman', 1800, 'keyword') limit 10;
```

	movie_title text	weightedrating numeric
1	The Dark Knight	8.0585
2	Fight Club	8.0040
3	Full Metal Jacket	7.2908
4	Batman Begins	7.2898
5	Iron Man	7.2347
6	The Incredibles	7.1493
7	Jaws	7.0579
8	Taken	6.9730
9	Gangs of New York	6.7713
10	Spider-Man	6.7031

4. ‘cast’

```
select * from WR_Recommender('Star Wars', 1800, 'cast') limit 10;
```

	movie_title text	weightedrating numeric
1	The Empire Strikes Back	7.7874
2	The Lion King	7.6096
3	Return of the Jedi	7.4920
4	A Clockwork Orange	7.4539
5	Blade Runner	7.4247
6	Raiders of the Lost Ark	7.2969
7	Apocalypse Now	7.2696
8	Indiana Jones and the L...	7.1743
9	Star Wars: Episode III - R...	6.8938
10	Indiana Jones and the T...	6.8334

5. 'crew'

```
select * from WR_Recommender('Toy Story', 1800, 'crew');
```

	movie_title text	weightedrating numeric
1	ハウルの動く城	7.3641
2	Finding Nemo	7.3359
3	Ratatouille	7.1898
4	The Incredibles	7.1493
5	Toy Story 2	7.0204
6	Corpse Bride	6.8227
7	A Bug's Life	6.6331
8	Cars	6.5417

Team member Xiaoye Zhang took ownership of this task.

Jaccard Recommender:

The Jaccard movie recommender takes a specified movie and returns the 10 movies with the highest jaccard similarity score. The jaccard similarity score is defined as the intersection of 2 sets divided by the union of the 2 sets. In this instance, the sets are defined as the keywords (K), genres (G), directors (D), and actors (A) of the 2 movies that are being compared.

$$Jaccard\ Score = \frac{|(K, G, D, A)_{movie\ 1} \cap (K, G, D, A)_{movie\ 2}|}{|(K, G, D, A)_{movie\ 1} \cup (K, G, D, A)_{movie\ 2}|}$$

$$= \frac{count\ of\ similar\ K, G, D, A}{count\ of\ all\ distinct\ K, G, D, A}$$

The jaccard movie recommender uses three functions as follows:

1. JaccardScore(movieID INT, movieID2 INT): The Jaccard Score functions takes 2 movie id's, calculates and returns the jaccard similarity score between the 2 specified movies. The score is calculated through several queries that retrieve the count of distinct and similar keywords, actors, genres, and directors between the two movies. The counts of distinct elements are summed as are the counts of similar elements; afterwards the counts are divided.

Below is an example, comparing the movies 'Toy Story' (1995) and 'The Lion King' (1994).

```
select * from JaccardScore(862, 8587);
```

	jaccardscore numeric
1	0.03

2. JaccardTable(movieID INT): The JaccardTable function takes in a movie ID and calculates the jaccard score between the specified movie and all the other movies in the database. The function returns a table containing the id's, titles, and jaccard scores of all the movies that were compared to the specified movie. This function uses a FOR loop and calls the JaccardScore function to retrieve all the scores.

Below is an example, using the movie 'Toy Story' (1995). Not all the results are shown.
select * from JaccardTable(862);

	movieid2 integer	movietitle text	jindex numeric
1	2887	And the Band Played On	0.00
2	6187	Nettoyage à sec	0.00
3	2692	Der rote Elvis	0.00
4	1850	Man on the Moon	0.02
5	2428	The Greatest Story Ever Told	0.00
6	4993	5 Card Stud	0.00
7	430	One, Two, Three	0.02
8	2771	American Splendor	0.02
9	5227	Hercules in New York	0.03
10	9901	Freedom Downtime	0.00
11	7096	Merlin	0.00

...

3. JacRecommend(movieID INT, title VARCHAR): The JacRecommend function requires a movie ID, and title. The parameters are validated to ensure that the inputted ID and movie title match the data in the database. If invalid parameters are entered, the values '-1' is returned. If the parameters are valid, the function returns the 10 movies with the highest jaccard similarity scores when compared to the specified movie. This function sorts and limits the results of JaccardTable in order to select the top 10 recommendations.

Below is an example, using the movie 'Toy Story' (1995). The best 10 recommendations are shown for a user that likes the movie 'Toy Story'.

```
select * from JacRecommend(862, 'Toy Story');
```

	movie text
1	Toy Story 2
2	Monsters, Inc.
3	A Bug's Life
4	Jungle 2 Jungle
5	Cars
6	A Grand Day Out
7	A Close Shave
8	My Favorite Martian
9	The Wrong Trousers
10	The Flintstones

Below we use the same movie, but an incorrect id is entered.

```
select * from JacRecommend(999, 'Toy Story');
```

	movie text
1	'-1'

To view the SQL code for the Jaccard Movie Recommender, please see the sql file.

Team member Isis Ramirez took ownership of this task.

COMPARISON BETWEEN 2 RECOMMENDER:

	Weighted Rating	Jaccard Score
Runtime	0.5~1s	<5min
Scope	1 at a time	4 at a time
Accuracy	Medium	High

6 Results

Using imperative and declarative SQL, we were able to create JSON data transformations, movie recommendation systems, triggers, and views which have been previously discussed. Below is a list of the deliverables.

Deliverable	**Method**
Production Company View	View
Production Company Analysis	Declarative SQL
Weighted Rating Recommender	Imperative SQL
Update Average Rating & Count Trigger	Imperative SQL/Trigger
Jaccard Movie Recommender	Imperative SQL

It is worthwhile to mention that the Jaccard recommendation systems had undesirable execution times. The execution time for the example using 'Toy Story' was approximately 6 minutes and 5 seconds.

7 Discussion

Design Decisions:

We decided to omit loading the links file, which included TMDB and IMDB ID's since the data was not of importance. Several attributes were presented in arrays of JSON objects such as production companies, production countries, genres, and keywords. The fields of utmost importance such as genres and keywords were designed as separate entities with associative entities mapping them to the movie table; these attributes were repeatedly referenced through the functions and queries. We were not as interested in the production companies and thus left the field as a multivalued attribute. Calculated attributes for the average ratings and number of ratings for each movie were added; this improves the performance of the mean weighted recommender function since it does not need to calculate these values every time it is called.

Challenges:

The biggest challenge was discovered when attempting to load the data. The data was not clean and needed much cleaning and preparation. Initially, several values that were in JSON stringified form were not actually valid. The most common instigators of errors when attempting to load JSON objects were quotation marks and single quotations used as possessive apostrophes. For time efficiency, a sample of the data was parsed in Python using regular expressions before importing it into Postgres.

Additionally, some attributes in the data were listed as arrays of JSON objects. However, this was resolved through Postgres' JSON functions. Attributes of great importance were first imported into dummy tables as arrays of JSON objects. The individual JSON objects were extracted and the elements of interested were inserted into target tables using the json_array_elements() function and the ->> operator. Attributes of lesser importance, such as 'production_companies', were kept as arrays of JSON element, but VIEWS were created using the same JSON functions and operators.

Assumptions:

The initial assumption that the data had valid JSON objects was erroneous and addressed with the JSON support functions in Postgres. We also assume that the numeric data assumed such as revenue, budget, rating scores is correct.

Changes from Original Plan:

Initially, we planned on doing an extensive analysis on what makes a film successful by investigating what type of films are more popular, highly rated, and higher grossing. However, we did not feel that the original plan provided an opportunity to showcase challenging imperative SQL functions. We decided that coding movie recommenders using imperative SQL was a more impressive and interesting feat.

Next Time / Next Steps:

The next steps would include adding more recent movies. Several movies of extreme popularity have been released such as Avengers: Infinity War and Captain Marvel. It would be interesting to do analysis on the most popular genres over time, or detailed analysis on the most popular movies. Additionally, we could determine whether the time efficiency of the movie recommenders could be improved through additional optimization and tuning.

8 Conclusion

PostgreSQL has pros and cons. The advantages that PostgreSQL provides are useful JSON support, ability to handle several many-to-many relations, thorough help documentation, and ability to support concurrency. Most of the disadvantages seen were revealed in the creation of the movie recommendation systems. When it came to the recommenders, it was challenging to implement complex mathematical operations and the execution time was poor. From a business standpoint, it would be easier and more time efficient to use a tool such as Python for movie recommendation systems.

9 References

[1] Getting Started with a Movie Recommendation System. (n.d.). Retrieved from https://www.kaggle.com/ibtesama/getting-started-with-a-movie-recommendation-system