Bitter Melons:

A Database Architecture for Review Aggregation

Sanad Masannat 24217734

sanad.masannat@ucdconnect.ie

Contents

List of Tables and Figures ii
  Figures ii
  Tables ii
Introduction 1
Database Plan: A Schematic View 2
Database Structure: A Normalized View 7
Database Views 12
Procedural Elements 16
Example Queries: Database In Action 21
Conclusions 24
Acknowledgements 25
References 25

List of Tables and Figures

Figures

Figure 1: ER Diagram of New Tables with Features Table 3
Figure 2: ER Diagram of Entire Schema 7
Figure 3: Sample Review Data 11
Figure 4: Sample User Data 11
Figure 5: Sample Critics Data 11
Figure 6: Sample Search Data 11
Figure 7: Sample Synopsis Data 12
Figure 8: Sample Sweetness Score Data 12
Figure 9: Top 10 View 13
Figure 10: Top Critics View 14
Figure 11: Franchise Sweetness View 15
Figure 12: Favourable Reviews View 16
Figure 13: Login Functionality 17
Figure 14: Password Reset Function 18
Figure 15: Reset Search History 19
Figure 16: Sweetness Score Trigger 20
Figure 17: Search Bar Functionality with Example 21
Figure 18: IGN Review Query 22
Figure 19: User's Seach Query 22
Figure 20: Negative Reviews by filmfanatic Query 23
Figure 21: Average Review Score by Critic 24

Tables

Table 1: List of Entities and their Attibutes 4
Table 2: List of Tables and Roles 7

Introduction

The objective of this project was to create a database that would support a review aggregate website akin to RottenTomatoes called BitterMelons. A database, as described by Kroenke and Auer, is a self-describing collection of integrated tables. An integrated table is a table which stores the data and relationships they have with each other (Kroenke 2013). The main motivation behind this project was to combine complex SQL queries and procedures to help support the creation of this application.

To accomplish this, the 7 design steps outlined by Hatlin and Morgan were followed which led to various new tables and functionalities being added to the *movies* database. These 7 steps were as follows (Halpin 2010):

1. Transform familiar examples into elementary facts.
2. Draw the fact types, and apply a population check.
3. Check for entity types to be combined, and note any arithmetic derivations.
4. Add uniqueness constraints, and check the arity of fact types.
5. Add mandatory role constraints, and check for logical derivations.
6. Add value, set-comparison, and subtyping constraints.
7. Add other constraints and perform final checks.

The main purpose of using the *movies* database was to support backwards compatibility, especially as the foundations are already in place. The data initially contained in the database consisted of text variables to support fields such as the name of the feature, franchise name, actors and directors name, gender, ethnicity and various other different fields. The tables in the previous database contained numerical fields which were mainly used for primary and foreign keys, year values, budget values and gross values. Adding on to the previous tables, 6 new tables were created:

- One to store critic information
- One for user information
- One to store a user’s search history
- One to store all the reviews made by a user
- One to store sweetness scores of all features

- One to store the feature's synopses

This would allow the efficient querying of data while helping to avoid redundancy in records. The creation of these tables also brings up the question of normalisation and if these newly created tables adhere to normalisation principles, which will be brought up in a later section.

The database, on top of basic querying and inserting data, should allow users to perform various functionalities in the application. The most important functionality this application would need to perform is to search for features in the following ways:

1. Search for a movie/franchise by a name/single actor
2. Search for all media with reviews above/below a certain score
3. Search through all media in the database for a key word or phrase
4. Search for reviews from specific a domain

As with most applications, handling user data is very important. As such, the application would need to store a user's email, username and password to allow them to log into the application. Following this, the application must also let the use have control over what data is stored about them such as their search history. The application should also provide a way for users to recover their password and update it. Finally, there must provide a way to update the sweetness score of a feature in real time as opposed to having to update the score manually as that would take up extra resources.

This report aims to answer most of the questions presented above and talk about the design of the schema and why the schema presented is an optimal way of managing the review aggregate website. It will cover the schema of the database, if the database is normalised properly and if it adheres to BCNF and in turn, 3NF, 2NF and 1NF, some examples of the queries, views and procedural functions made specifically to help the review aggregate site.

Database Plan: A Schematic View

The database works off the previous movies database received in class while adding in extra tables to help support the media review aggregation site. There were 11 tables in the previous version of the *movies* database. For this project, 7 new tables were added to aid the aggregation review site, specifically: Reviews, Users, Critics, Feature Synopsis, Sweetness Index and User Search History. Figure 1 below shows an ER diagram that depicts how each of these newly created tables relate to the features table of the database.

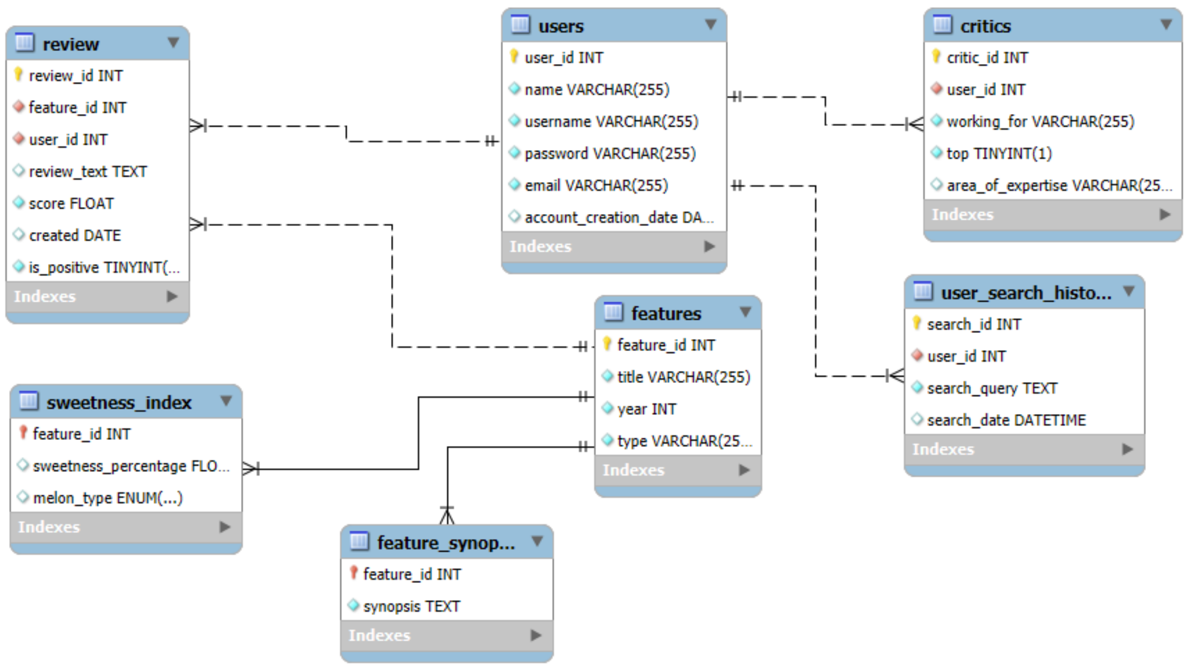


Figure 1: ER Diagram of New Tables with Features Table

One of the most important tables is the "features" tables as it contains all the features and their relevant information in it. The "features" table's attribute consists of the feature id, year it was released, name of the feature and the type of feature it is (i.e. is it a movie, tv show etc.). As seen in Figure 2, the features table is connected to 10 other entities here from its feature_id alone, namely budget, domestic gross, international gross, reviews, franchise features, feature role, feature work, feature synopsis and sweetness index.

Another important entity is the "user" table. While it does not interact with as many tables compared to features, storing user information is important for any application, not just a movie aggregation site. This database currently stores the user's name, username, password, email and when they created the account. This would allow users to log-in or create an account as all the relevant information is stored in the table. Thus, when a new user is created or logged in, the table would just need to be altered or queried. For security purposes, the application would need to handle all encryption and secure storage of the sensitive information of this table.

Finally, the last important table would be the review's table. This table would store all the information that makes up a review which include the following: review id (which acts as a primary key), who wrote the reviews (user id), what was written in the review (review text), what feature they are reviewing (feature id), what score they gave the feature (score) and if the review is positive or negative(is_positive). This is important for the review aggregate site as it should store the reviews in case users want to search for specific reviews based upon criteria such as who wrote the review or words included in the review. It will also be relevant in updated the sweetness score as each time a new review is added, the score would need to be adjusted.

These 3 tables are the most connected to other tables (or entities) in the schema, meaning they play a key role in how the application would work (Users and Reviews). In Table 1, all the tables, their corresponding attributes and how they are connected to other tables are shown while Figure 2 displays this in the form of an ER-Diagram in a more detailed way.

Table 1: List of Entities and their Attributes

Table	**Attributes**	**Entities Connected to and how**
Features	• feature_id • title • year • type	• Feature Role - feature_id • Feature Work- feature_id • Franchise Features- feature_id • Feature Synopsis - feature_id • Review - feature_id • Budget - feature_id • Domestic Gross - feature_id • International Gross - feature_id • Sweetness Index - feature_id
Feature Role	• role_id • feature_id • person • role • gender • ethnicity	• Features - feature_id • Role Type - role_id
Franchises	• franchise_id • name • studio	• Franchise Features - franchise_id
Franchise Features	• feature_id • franchise_id • franchise_number	• Franchise Features - franchise_id • Features - feature_id

Budget	• feature_id • currency_id • amount	• Currency - currency_id • Features - feature_id
Domestic Gross	• feature_id • currency_id • amount	• Currency - currency_id • Features- feature_id
International Gross	• feature_id • currency_id • amount	• Currency - currency_id • Features- feature_id
Currency	• currency_id • currency_name • currency_symbol	• Budget - currency_id • Domestic Gross - currency_id • International gross - currency_id
Genres	• feature_id • comedy • romance • action • fantasy • horror • mystery • science_fiction • historical • espionage • crime • drama • youth • biography • political	• Features - feature_id
Role Type	• role_id • role_type	• Feature Role – role_id

Users	• user_id • name • username • password • email • active_since	• Reviews – user_id • Critics – user_id • User Search History – user_id
Critics	• critic_id • user_id • working_for • top • area_of_expertise	• Users – user_id
Reviews	• review_id • feature_id, • user_id, • review_text, • score • created • is_positive	• Users • Features - feature_id
User Search history	• search_id • user_id • seach_query • search_date	• Users – user_id
Feature Work	• job_id, • feature_id • person • job • gender • ethnicity	• Features - feature_id
Sweetness Index	• feature_id • sweetness_percentage • melon_type	• Features - feature_id

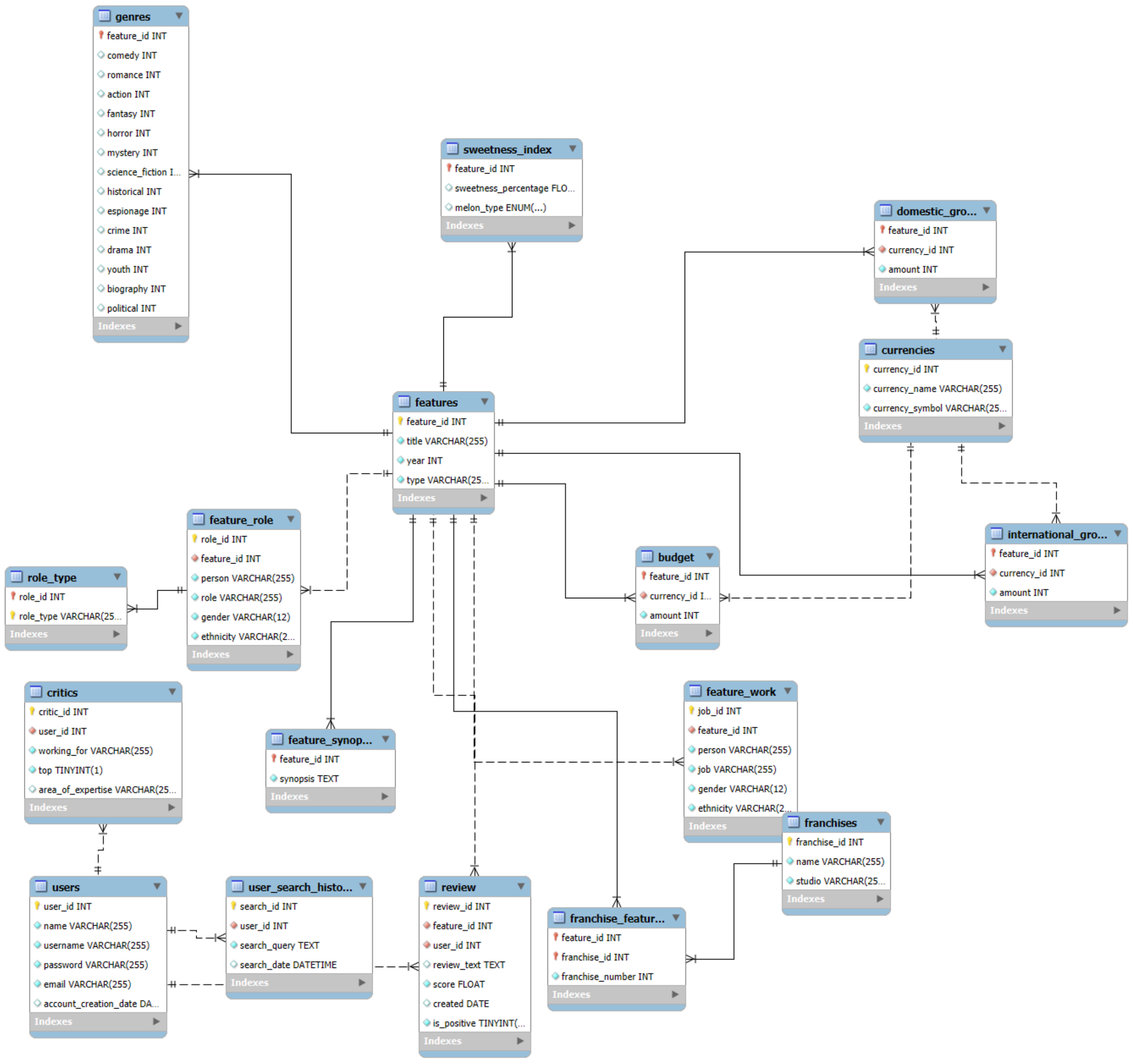


Figure 2: ER Diagram of Entire Schema

Database Structure: A Normalized View

There are 17 tables in this database. Table 2 below lists all the current tables and what each of their roles are:

Table 2: List of Tables and Roles

Table	Purpose	Primary Key

Budget	Stores how much money was needed to make the movie and in which currency	Feature_id
Critics	Stores a list of the all the critic accounts that have been made, their relevant information and which users from the users are critics.	Critic_id
Currency	Stores the different types of currency and what symbol represents it	Currency_id
Domestic Gross	How much money it made in its home country	Feature_id
Feature Role	Stores what character the actor played in the move and the details of the actor	Role_id
Feature Work	Stores the information about the type of person who has worked on the feature such as actor, director, producer etc.	Job_id
Features	Store all the features in the database and the information related to them.	Feature_id
Franchise Features	Stores a list of which movies belong to which franchise and which number they are in the current franchise	Feature_id & Franchise_id
Franchises	Stores all current franchises that are in this database	Franchise_id
Genres	Stores a score for each possible genres for a given movie with a higher score meaning that its leans more closely to that genre	Feature_id
International Gross	How much money it made globally	Feature_Id

Reviews	This table will store all the reviews for a certain feature. It will also store when the review was made, who made it, if the review was positive and what score the reviewer gave the feature	Review_id
Role Type	Stores that type of role the character was for a given role in Feature Role	Role_type & Role_id
Sweetness Index	Would store all current features sweetness score and if they are HoneyDew or HoneyDon't	Feature_id
User Search history	Would be used to store a user's search history so that they can access previous searches	User_id & search_id
Users	Stores a user's information that would be required by the application such name, username, email, password and when they created the account. Would be used for login purposes and to map critics, reviews and search history to each user	User_id

To determine if the tables are in 1NF, 2NF, 3NF and BCNF, it best to approach normalisation from First Normal form and move towards Third Normal Form.

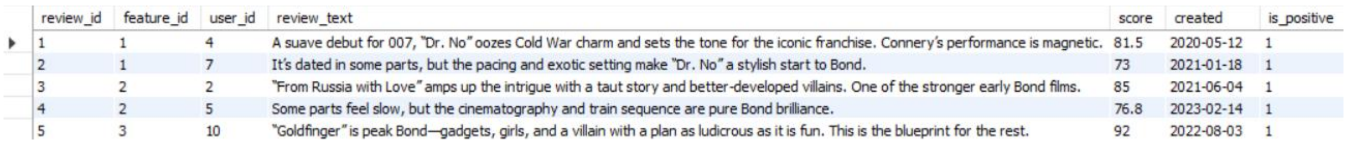
The condition for a table to adhere to First Normal Form is if the domain of all attributes is atomic (Silberschatz et. al 2020). This means that elements of a domain are indivisible, meaning that every table has a unique primary key to ensure there is no repeating and/or redundant groups of data. Looking at Figure 3, each table has a primary key that would ensure all records in the table is uniquely identifiable. As for redundancy, all attributes to each table contains the relevant information, with redundant information being removed. It is important to remove all redundant information because, as mentioned by Jan L. Harrington, redundant information such as unnecessarily duplicated information can take up a fair amount of disk space and can become harder to maintain if there is many redundant information in the schema (Harrington 2016). As all conditions of 1NF are met, the database is in 1NF.

Moving onto 2NF, the conditions for 2NF are that the database is in 1NF and that there are no partial dependencies. As mentioned above, the database is in 1NF so that condition is met. The second condition means that, according to *Database System Concepts*, for attributes α, β and γ, for a functional dependency (α→β) there is not a proper subset γ →β (Silberschatz et. al 2020). In the case of the current schema, all tables are reliant on their primary key or super key. A super key is a type of key in which when multiple columns uniquely reference a record in the table. For example, the email attribute for users could be also in the critics table but that would create a partial dependency. If we move it to the user table's however, the email becomes dependant on the user_id and the critic_id can use that as a foreign key to access it, thus preventing a partial dependency.

Finally, for a database to be in 3NF, the conditions that need to be met are that the database is in 2NF and there are no transient dependencies. The 2NF condition is met as mentioned in the earlier paragraph. The final condition of 3NF is that every attribute is only dependant on the primary key and nothing else. All tables currently satisfy this through methods such as the use of super keys. For example, the Franchise Features table has a super key which is comprised of franchise_id and feature_id, meaning that if separate, there would be a dependency on each other but as we used the two as the key, we avoid this. For other tables such as Reviews, all other attributes rely solely on the primary key, in Reviews case all attributes rely solely on the review_id.

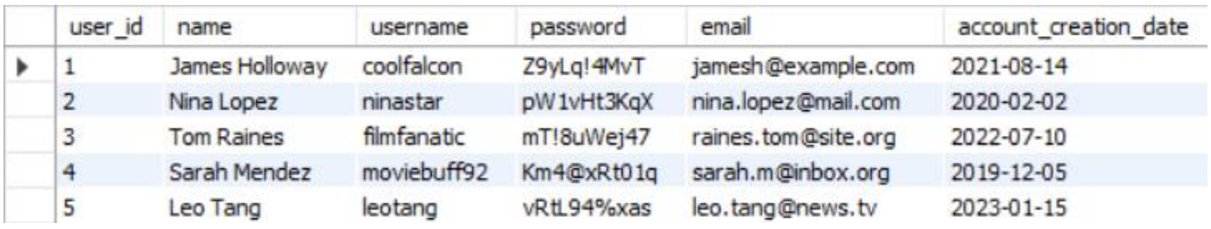
BCNF, as mentioned in *Database System Concepts,* is like 3NF but 3NF has slightly relaxed constraints (Silberschatz et. al 2020). The conditions for BCND to be held is that for a relation schema R with respect to n BCNF with respect to a set F of functional dependencies if for all functional dependencies in F+ of the form α →β where α ⊆R and β ⊆ R either α →β is trivial (this means that β⊆α) or α is a super key for R. This condition holds across the schema. For example, in the critics table, critic_id serves as the primary key. Despite user_id acting as a foreign key to reference the users table, no attribute other than the key is functionally dependent on any other non-candidate key. Another example is for the User Search History table in that while one column doesn't uniquely represent each table, it has a super key (search_id and user_id) which does, which meets the second condition. The use of foreign keys here help keep all tables normalised by preventing any violations.

Figures 3 – 8 show example records that are stored in the newly created tables to showcase the data the database will be working with. The data has been limited to 5 entries for ease of reading.



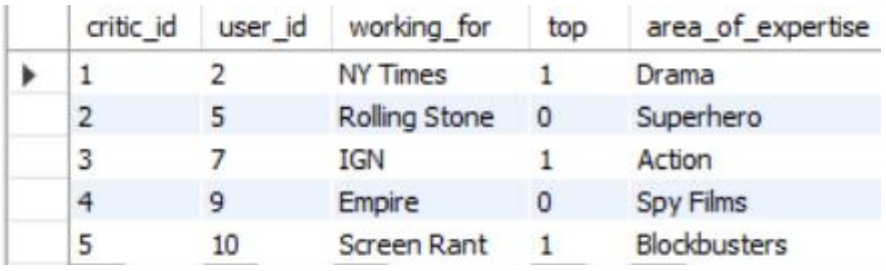
review_id	feature_id	user_id	review_text	score	created	is_positive
1	1	4	A suave debut for 007, "Dr. No" oozes Cold War charm and sets the tone for the iconic franchise. Connery's performance is magnetic.	81.5	2020-05-12	1
2	1	7	It's dated in some parts, but the pacing and exotic setting make "Dr. No" a stylish start to Bond.	73	2021-01-18	1
3	2	2	"From Russia with Love" amps up the intrigue with a taut story and better-developed villains. One of the stronger early Bond films.	85	2021-06-04	1
4	2	5	Some parts feel slow, but the cinematography and train sequence are pure Bond brilliance.	76.8	2023-02-14	1
5	3	10	"Goldfinger" is peak Bond—gadgets, girls, and a villain with a plan as ludicrous as it is fun. This is the blueprint for the rest.	92	2022-08-03	1

Figure 3: Sample Review Data



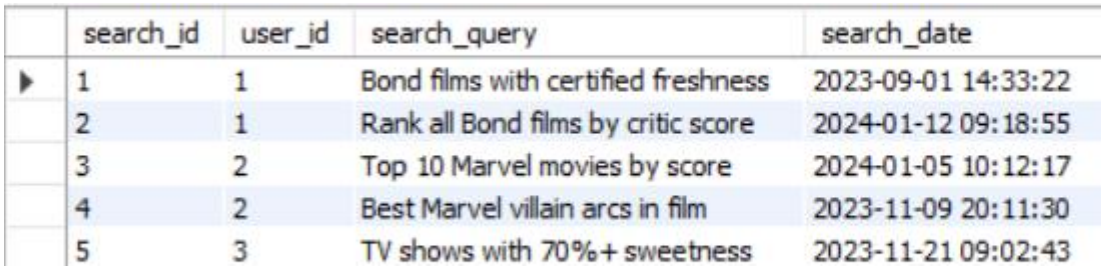
user_id	name	username	password	email	account_creation_date
1	James Holloway	coolfalcon	Z9yLq!4MvT	jamesh@example.com	2021-08-14
2	Nina Lopez	ninastar	pW1vHt3KqX	nina.lopez@mail.com	2020-02-02
3	Tom Raines	filmfanatic	mT!8uWej47	raines.tom@site.org	2022-07-10
4	Sarah Mendez	moviebuff92	Km4@xRt01q	sarah.m@inbox.org	2019-12-05
5	Leo Tang	leotang	vRtL94%xas	leo.tang@news.tv	2023-01-15

Figure 4: Sample User Data



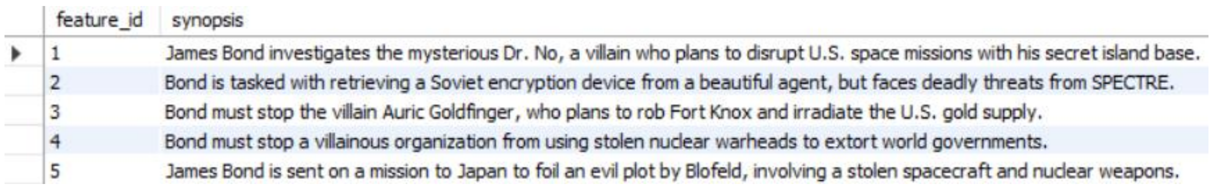
critic_id	user_id	working_for	top	area_of_expertise
1	2	NY Times	1	Drama
2	5	Rolling Stone	0	Superhero
3	7	IGN	1	Action
4	9	Empire	0	Spy Films
5	10	Screen Rant	1	Blockbusters

Figure 5: Sample Critics Data



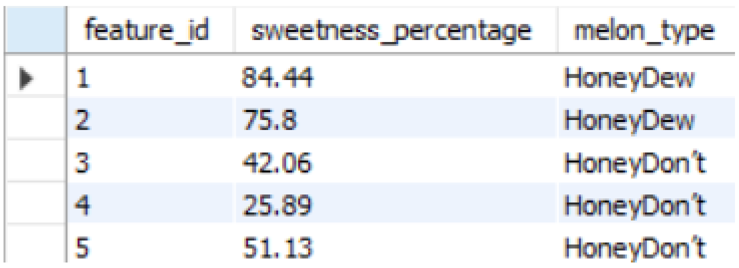
search_id	user_id	search_query	search_date
1	1	Bond films with certified freshness	2023-09-01 14:33:22
2	1	Rank all Bond films by critic score	2024-01-12 09:18:55
3	2	Top 10 Marvel movies by score	2024-01-05 10:12:17
4	2	Best Marvel villain arcs in film	2023-11-09 20:11:30
5	3	TV shows with 70%+ sweetness	2023-11-21 09:02:43

Figure 6: Sample Search Data



feature_id	synopsis
1	James Bond investigates the mysterious Dr. No, a villain who plans to disrupt U.S. space missions with his secret island base.
2	Bond is tasked with retrieving a Soviet encryption device from a beautiful agent, but faces deadly threats from SPECTRE.
3	Bond must stop the villain Auric Goldfinger, who plans to rob Fort Knox and irradiate the U.S. gold supply.
4	Bond must stop a villainous organization from using stolen nuclear warheads to extort world governments.
5	James Bond is sent on a mission to Japan to foil an evil plot by Blofeld, involving a stolen spacecraft and nuclear weapons.

Figure 7: Sample Synopsis Data



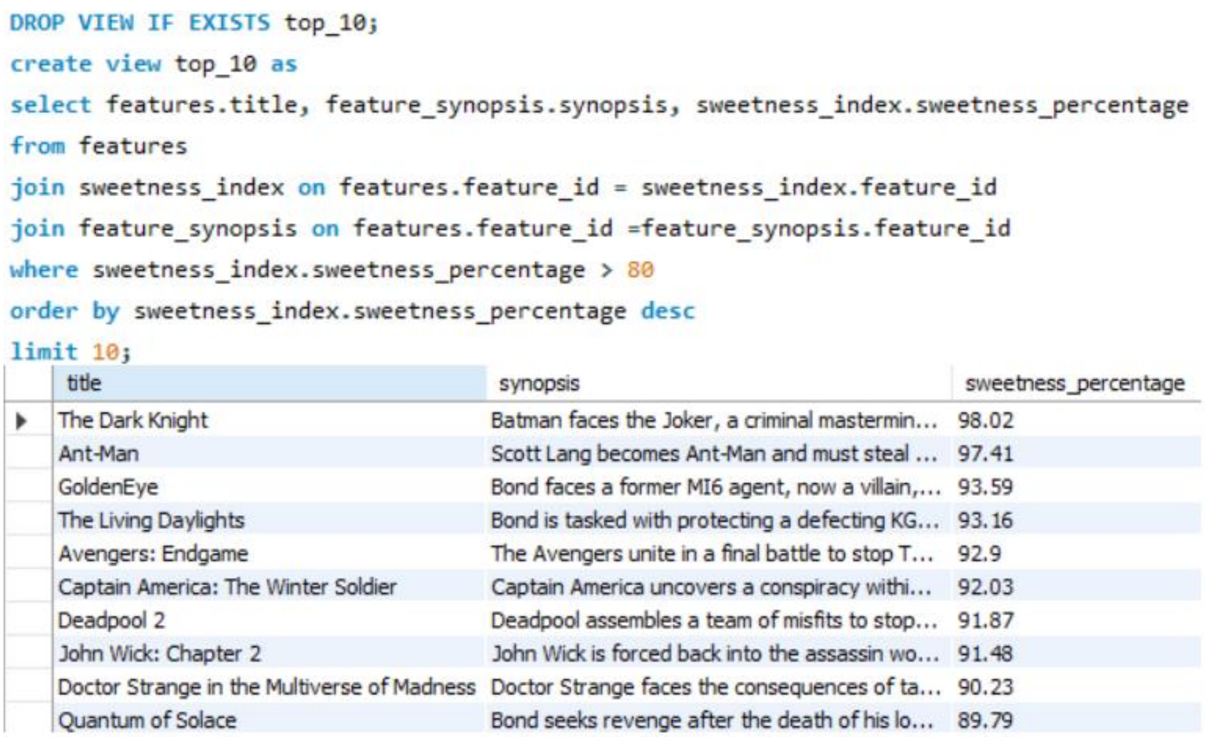
feature_id	sweetness_percentage	melon_type
1	84.44	HoneyDew
2	75.8	HoneyDew
3	42.06	HoneyDon't
4	25.89	HoneyDon't
5	51.13	HoneyDon't

Figure 8: Sample Sweetness Score Data

Database Views

Views are virtual tables which are derived from a SELECT query on another table or tables (Connoly 2016). These tend to capture more complex queries or query results that one would use often in other queries to simplify them. These also provide security in that, as mentioned by Silberschatz et al., not all users need to have access to all information in the schema. It also allows one to handle any changes as views are not precomputed, they are computed at run time (Silberschatz et. al 2020). For the project, 4 views have been created.

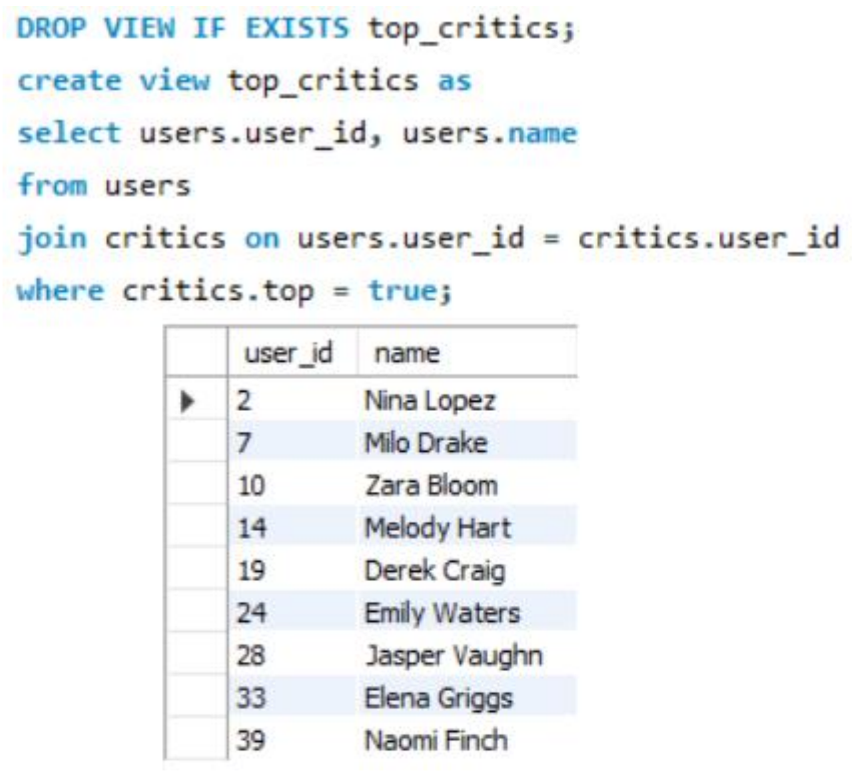
The first view created is top_10. The main purpose of this view is capturing the movies with the highest sweetness score and showing users the movie, score and its synopsis. This would be used in the applications front page so users can see what the current highest rated movies of all time are. Figure 9 shows the SQL code and its output.


```
DROP VIEW IF EXISTS top_10;
create view top_10 as
select features.title, feature_synopsis.synopsis, sweetness_index.sweetness_percentage
from features
join sweetness_index on features.feature_id = sweetness_index.feature_id
join feature_synopsis on features.feature_id =feature_synopsis.feature_id
where sweetness_index.sweetness_percentage > 80
order by sweetness_index.sweetness_percentage desc
limit 10;
```

title	synopsis	sweetness_percentage
The Dark Knight	Batman faces the Joker, a criminal mastermin...	98.02
Ant-Man	Scott Lang becomes Ant-Man and must steal ...	97.41
GoldenEye	Bond faces a former MI6 agent, now a villain,...	93.59
The Living Daylights	Bond is tasked with protecting a defecting KG...	93.16
Avengers: Endgame	The Avengers unite in a final battle to stop T...	92.9
Captain America: The Winter Soldier	Captain America uncovers a conspiracy withi...	92.03
Deadpool 2	Deadpool assembles a team of misfits to stop...	91.87
John Wick: Chapter 2	John Wick is forced back into the assassin wo...	91.48
Doctor Strange in the Multiverse of Madness	Doctor Strange faces the consequences of ta...	90.23
Quantum of Solace	Bond seeks revenge after the death of his lo...	89.79

Figure 9: Top 10 View

The next view created is top_critics. The purpose of this view is to allow users to look for critics whose reviews are highly regarded. Users can more easily filter out or search for reviews written by top critics, making a more informed decision on what features they would want to watch. While this may seem simple, this query was made into a view because of the changing nature of rankings. As a view calculates the output at real-time rather than pre-compute them, the ranking can be output without having to worry about updating the query every time the ranking changes.

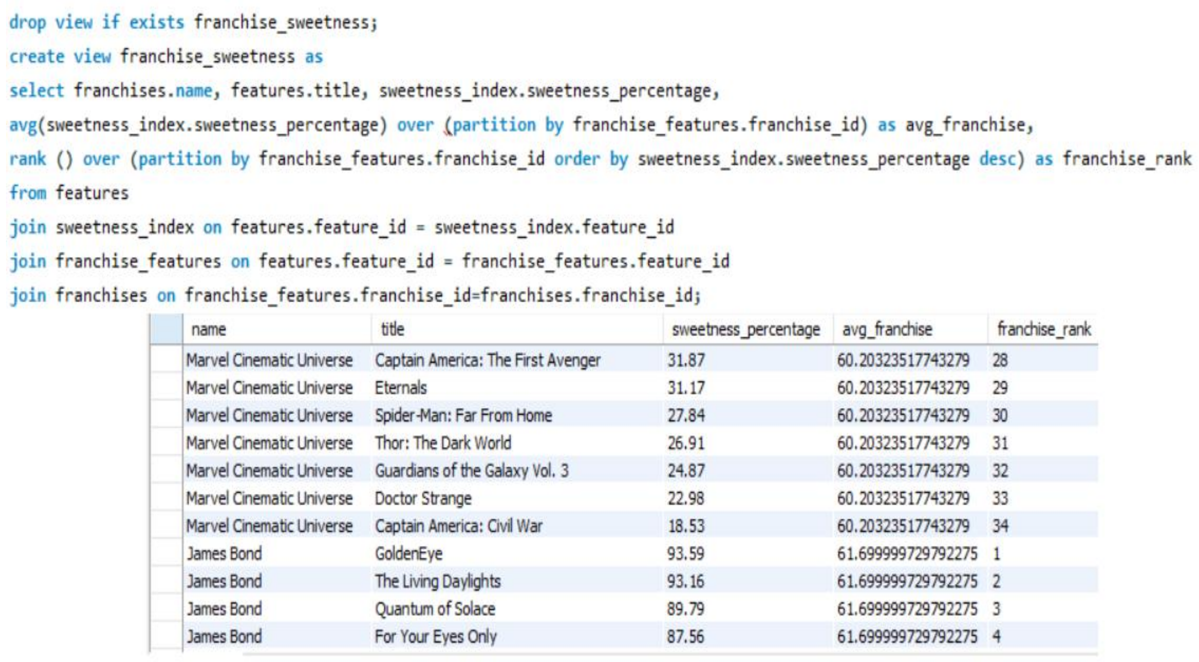


```sql
DROP VIEW IF EXISTS top_critics;
create view top_critics as
select users.user_id, users.name
from users
join critics on users.user_id = critics.user_id
where critics.top = true;
```

user_id	name
2	Nina Lopez
7	Milo Drake
10	Zara Bloom
14	Melody Hart
19	Derek Craig
24	Emily Waters
28	Jasper Vaughn
33	Elena Griggs
39	Naomi Finch

Figure 10: Top Critics View

Franchise_sweetness was created for the purpose of letting users have a high-level view of all franchises and the franchise's average sweetness. The features within the franchise are also ranked according to their own sweetness score. This allows producers to look at the trends of their movies to make decisions regarding the franchise's success. It also lets users see how their favourite franchise is compared to other franchises (for example, Marvel vs DC). The reason this was created as a view is because the top critic pool can change over time. Newer critics can be added, and older ones could be no longer considered a top critic hence the reason it is a view.

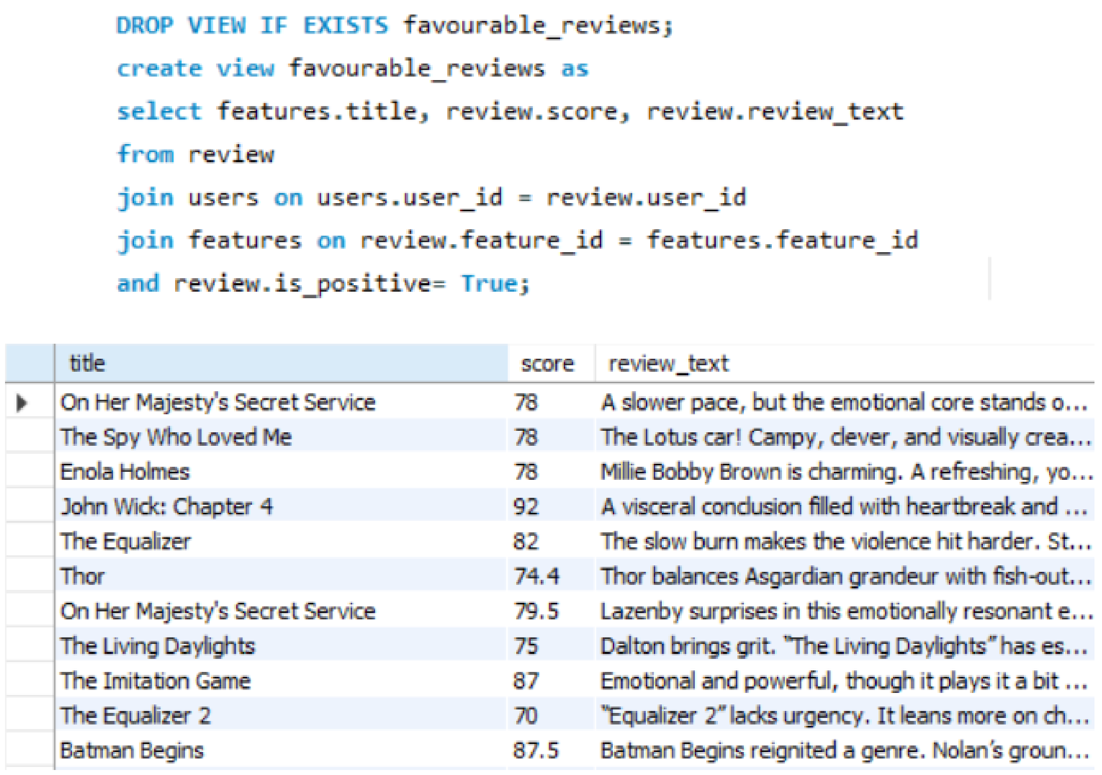


```sql
drop view if exists franchise_sweetness;
create view franchise_sweetness as
select franchises.name, features.title, sweetness_index.sweetness_percentage,
avg(sweetness_index.sweetness_percentage) over (partition by franchise_features.franchise_id) as avg_franchise,
rank () over (partition by franchise_features.franchise_id order by sweetness_index.sweetness_percentage desc) as franchise_rank
from features
join sweetness_index on features.feature_id = sweetness_index.feature_id
join franchise_features on features.feature_id = franchise_features.feature_id
join franchises on franchise_features.franchise_id=franchises.franchise_id;
```

name	title	sweetness_percentage	avg_franchise	franchise_rank
Marvel Cinematic Universe	Captain America: The First Avenger	31.87	60.20323517743279	28
Marvel Cinematic Universe	Eternals	31.17	60.20323517743279	29
Marvel Cinematic Universe	Spider-Man: Far From Home	27.84	60.20323517743279	30
Marvel Cinematic Universe	Thor: The Dark World	26.91	60.20323517743279	31
Marvel Cinematic Universe	Guardians of the Galaxy Vol. 3	24.87	60.20323517743279	32
Marvel Cinematic Universe	Doctor Strange	22.98	60.20323517743279	33
Marvel Cinematic Universe	Captain America: Civil War	18.53	60.20323517743279	34
James Bond	GoldenEye	93.59	61.699999729792275	1
James Bond	The Living Daylights	93.16	61.699999729792275	2
James Bond	Quantum of Solace	89.79	61.699999729792275	3
James Bond	For Your Eyes Only	87.56	61.699999729792275	4

Figure 11: Franchise Sweetness View

The final view created, favourable_reviews, has a variety of uses. The first use is to allow users to see if their favourite movies have any positive reviews. The next use is to allow developers to of the app to highlight some movies, such as recent releases, on the front page. The reason this became a view as opposed to a query is primarily due to speed. It is faster to query through this view than to query through the full review table if searching for favourable reviews. It also allows for further filtration of positive reviews.



```sql
DROP VIEW IF EXISTS favourable_reviews;
create view favourable_reviews as
select features.title, review.score, review.review_text
from review
join users on users.user_id = review.user_id
join features on review.feature_id = features.feature_id
and review.is_positive= True;
```

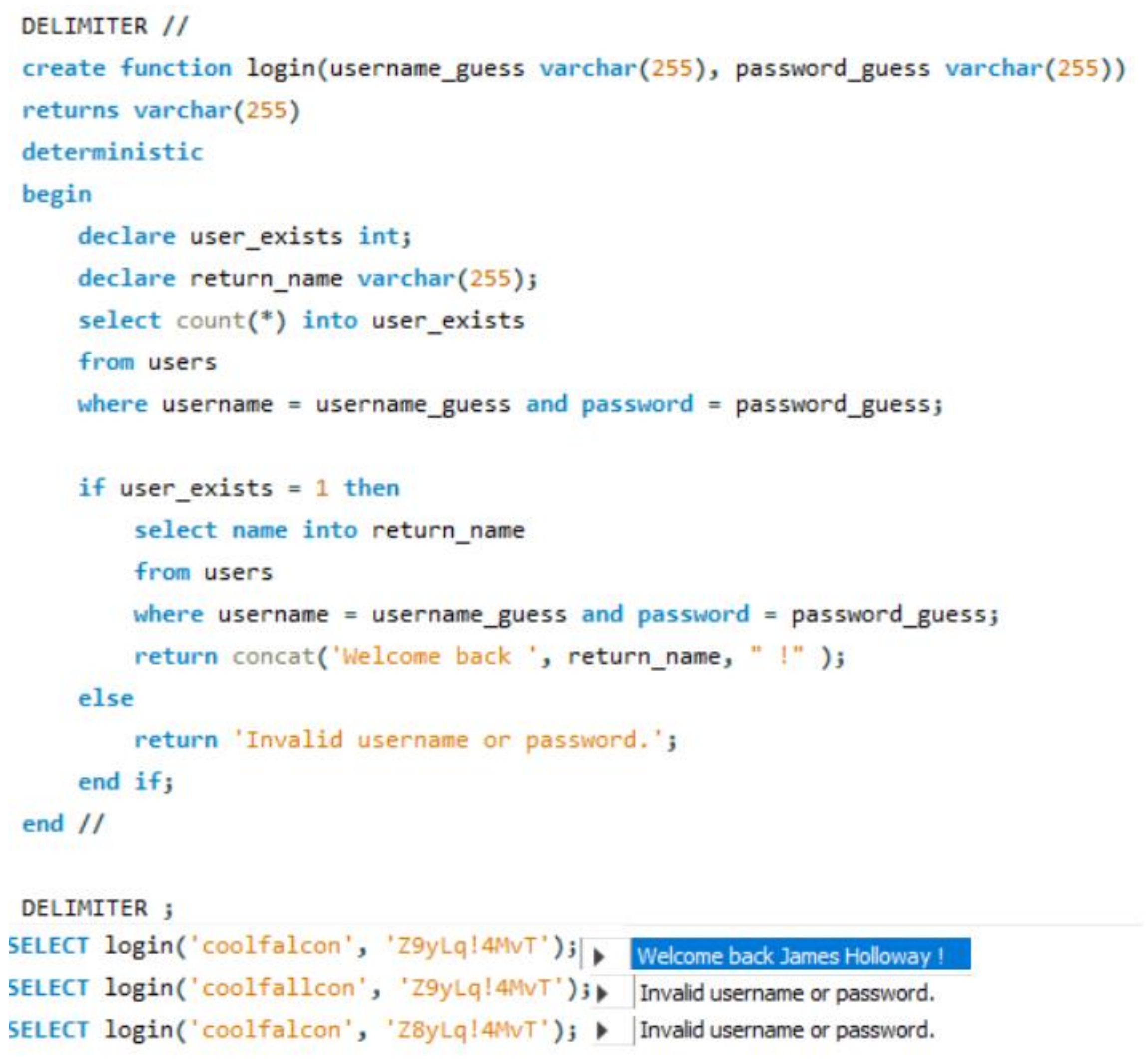
title	score	review_text
On Her Majesty's Secret Service	78	A slower pace, but the emotional core stands o...
The Spy Who Loved Me	78	The Lotus car! Campy, clever, and visually crea...
Enola Holmes	78	Millie Bobby Brown is charming. A refreshing, yo...
John Wick: Chapter 4	92	A visceral conclusion filled with heartbreak and ...
The Equalizer	82	The slow burn makes the violence hit harder. St...
Thor	74.4	Thor balances Asgardian grandeur with fish-out...
On Her Majesty's Secret Service	79.5	Lazenby surprises in this emotionally resonant e...
The Living Daylights	75	Dalton brings grit. "The Living Daylights" has es...
The Imitation Game	87	Emotional and powerful, though it plays it a bit ...
The Equalizer 2	70	"Equalizer 2" lacks urgency. It leans more on ch...
Batman Begins	87.5	Batman Begins reignited a genre. Nolan's groun...

Figure 12: Favourable Reviews View

Procedural Elements

MySQL not only makes use of views to support database applications, but they also make use of stored programs called procedural elements. Procedural elements allow developers to extend the capabilities of an SQL application by introducing functions, triggers and procedures to a database. They essentially act as a programming language within the SQL application by introducing techniques such as case statements and loops. These help provide a database many benefits such as allowing them to handle errors, encapsulation of complex calculations and better security as one can enable controlled access to sensitive data by using appropriate privilege selection (DuBois 2013). This schema employs 5 procedural elements, each with their own important roles.

As with most applications, a user login page is needed. The first procedural element acts as a basic login function. The way this procedural element works is it takes in the username and password in, queries the users table to see if there is a user that matches these credentials. If so, it would output "Welcome Back [name]", otherwise it would just output "Invalid Username or Password". The vague message adds a layer of security as any malicious user would not know if their password or username was the issue. However, the login itself is insecure as it does allow unlimited attempts, which can be changed in later versions. Figure 13 below shows a sample login test with either the username or password being incorrect and one working test case.



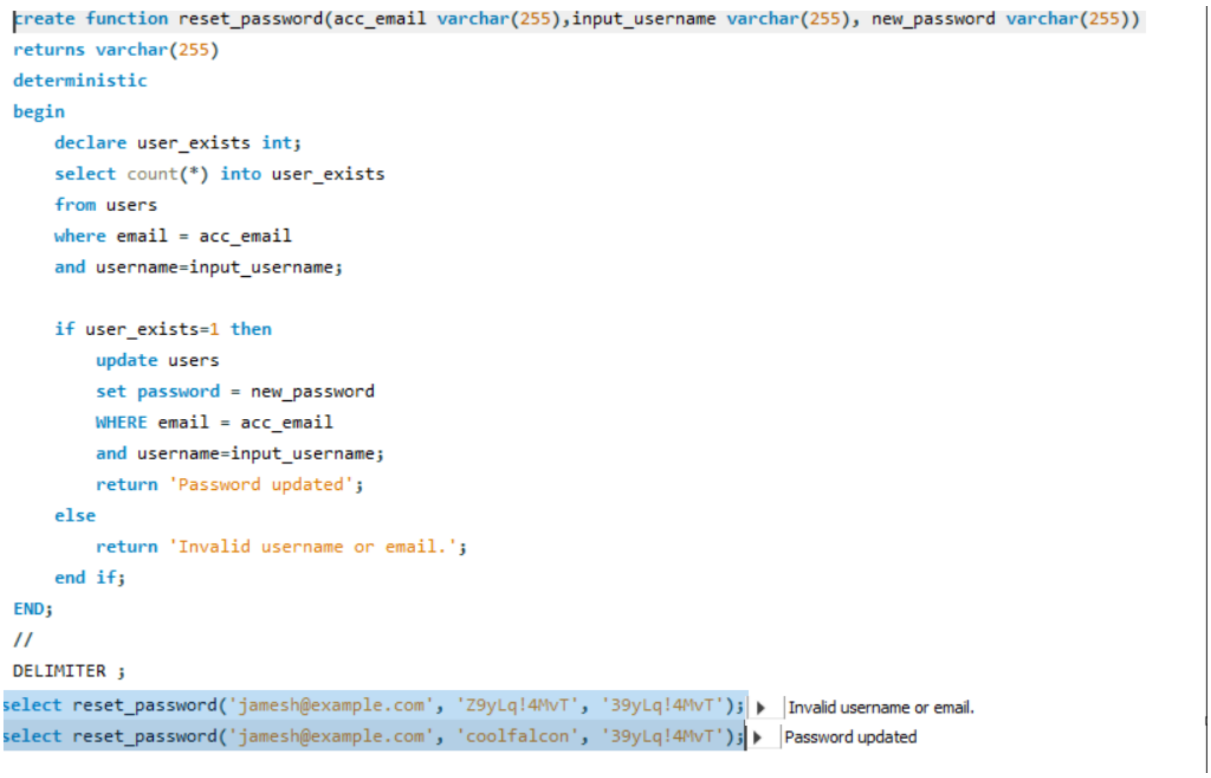
```
DELIMITER //
create function login(username_guess varchar(255), password_guess varchar(255))
returns varchar(255)
deterministic
begin
    declare user_exists int;
    declare return_name varchar(255);
    select count(*) into user_exists
    from users
    where username = username_guess and password = password_guess;

    if user_exists = 1 then
        select name into return_name
        from users
        where username = username_guess and password = password_guess;
        return concat('Welcome back ', return_name, " !" );
    else
        return 'Invalid username or password.';
    end if;
end //

DELIMITER ;
SELECT login('coolfalcon', 'Z9yLq!4MvT');    Welcome back James Holloway !
SELECT login('coolfallcon', 'Z9yLq!4MvT');   Invalid username or password.
SELECT login('coolfalcon', 'Z8yLq!4MvT');    Invalid username or password.
```

Figure 13: Login Functionality

Following this, another function was created to allow users to reset their passwords, given that they remember their username and email address used to create the account. The logic for this function works similar to what is stated above in that it takes in their username and email address. However, it will also take in their newly desired password. If we can find a user that matches the username and email address combination, the password will be changed. If either is incorrect or no such user exists, the password is not updated the password, and a custom error message is displayed. The reason this and the login function are procedural functions is because if it is not done within MySQL, it would make the database prone to attacks, which would allow malicious actors to steal another person's identity. The procedural functions here add another security layer, albeit a weak one. Figure 14 tests this function by passing in incorrect credentials and a correct on.

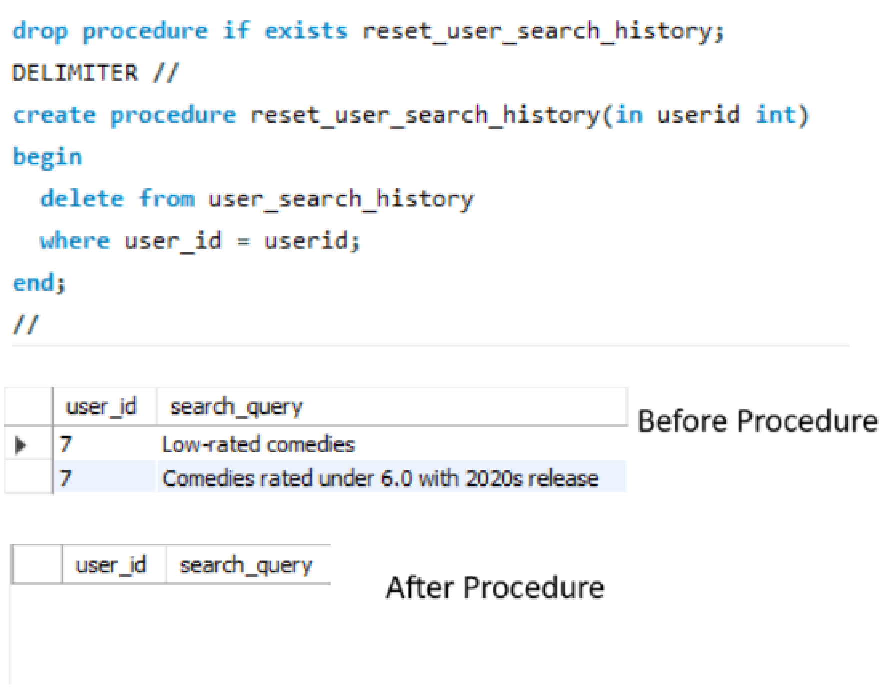


```
create function reset_password(acc_email varchar(255),input_username varchar(255), new_password varchar(255))
returns varchar(255)
deterministic
begin
    declare user_exists int;
    select count(*) into user_exists
    from users
    where email = acc_email
    and username=input_username;

    if user_exists=1 then
        update users
        set password = new_password
        WHERE email = acc_email
        and username=input_username;
        return 'Password updated';
    else
        return 'Invalid username or email.';
    end if;
END;
//
DELIMITER ;
select reset_password('jamesh@example.com', 'Z9yLq!4MvT', '39yLq!4MvT');    Invalid username or email.
select reset_password('jamesh@example.com', 'coolfalcon', '39yLq!4MvT');    Password updated
```

Figure 14: Password Reset Function

The next procedure, reset_user_search, resets any searches the user has done within the application. This is important as the user should have control over the data the application should store. This allows them to reset the data the application stores of them which would lead to a reset in user preferences as well. It would also allow them to erase any particular mistakes or accidental searches made within the application. Figure 15 shows this in action.



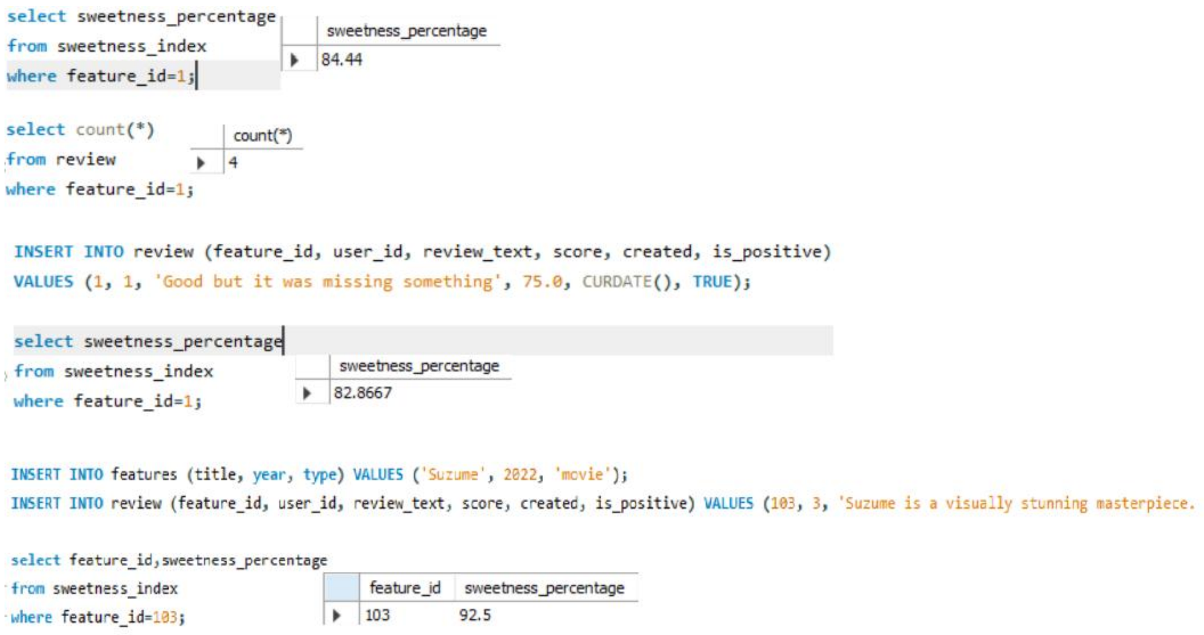
```
drop procedure if exists reset_user_search_history;
DELIMITER //
create procedure reset_user_search_history(in userid int)
begin
  delete from user_search_history
  where user_id = userid;
end;
//
```

user_id	search_query
7	Low-rated comedies
7	Comedies rated under 6.0 with 2020s release

user_id	search_query

Figure 15: Reset Search History

Within the database, a trigger was added to the sweetness_score table such that whenever a review is added for a specific feature, it would update the table. How the table is updated is dependant on how many reviews are available in the reviews table. If there is only one review (the newly added review), a new record is added to the sweetness_index table which would use the review score as the sweetness score (which then determines if the feature is *HoneyDew* or *HoneyDon't*). If there is more than one review in the table, the record is updated with the newly calculated average using the formula New_Score = (Old_Score * (number_of_reviews -1) + (review_score)) / number_of_reviews. This was chosen as a trigger because it would become difficult to manually update the sweetness_index table each time a review is added. Figure 16 shows how the trigger works in the different ways a review is added such as when a new feature is added.



```
select sweetness_percentage
from sweetness_index
where feature_id=1;
```

```
sweetness_percentage
84.44
```

```
select count(*)
from review
where feature_id=1;
```

```
count(*)
4
```

```
INSERT INTO review (feature_id, user_id, review_text, score, created, is_positive)
VALUES (1, 1, 'Good but it was missing something', 75.0, CURDATE(), TRUE);
```

```
select sweetness_percentage
from sweetness_index
where feature_id=1;
```

```
sweetness_percentage
82.8667
```

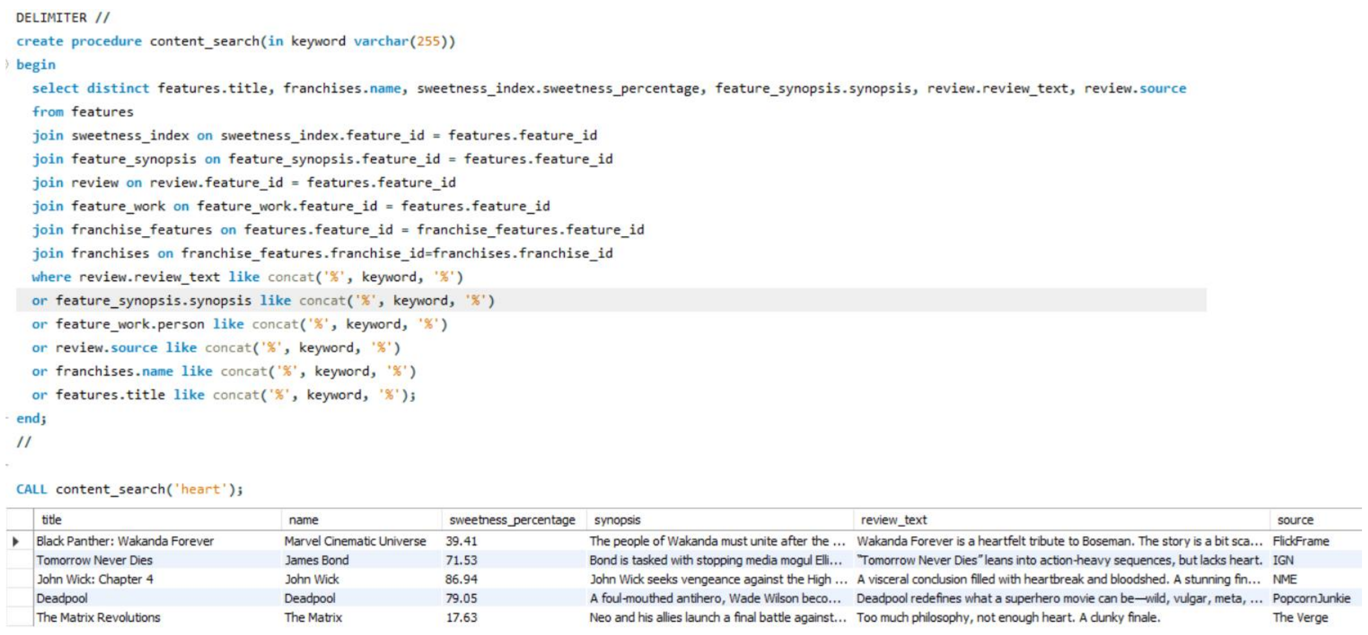
```
INSERT INTO features (title, year, type) VALUES ('Suzume', 2022, 'movie');
INSERT INTO review (feature_id, user_id, review_text, score, created, is_positive) VALUES (103, 3, 'Suzume is a visually stunning masterpiece.
```

```
select feature_id,sweetness_percentage
from sweetness_index
where feature_id=103;
```

```
feature_id   sweetness_percentage
103          92.5
```

Figure 16: Sweetness Score Trigger

The final procedure, which is arguably the most important, is the content_search procedure. This procedure takes in a keyword or phrase, searches through reviews, features, franchises, feature_synopsis and feature_work, which are joined together, and will output any features which contains the keyword or phrase. It will also output the name of the relevant franchise if applicable, the sweetness score of the feature, and the review that could have caused the movie to be outputted. As there could be many records in the joined table, only a single version of the feature is output. This procedure would act as the backbone of the application in that it allows for the user to search for any feature of their choosing by inputting a word or phrase. Figure 17 shows how the procedure works using the search term “heart”.



```sql
DELIMITER //
create procedure content_search(in keyword varchar(255))
begin
  select distinct features.title, franchises.name, sweetness_index.sweetness_percentage, feature_synopsis.synopsis, review.review_text, review.source
  from features
  join sweetness_index on sweetness_index.feature_id = features.feature_id
  join feature_synopsis on feature_synopsis.feature_id = features.feature_id
  join review on review.feature_id = features.feature_id
  join feature_work on feature_work.feature_id = features.feature_id
  join franchise_features on features.feature_id = franchise_features.feature_id
  join franchises on franchise_features.franchise_id=franchises.franchise_id
  where review.review_text like concat('%', keyword, '%')
  or feature_synopsis.synopsis like concat('%', keyword, '%')
  or feature_work.person like concat('%', keyword, '%')
  or review.source like concat('%', keyword, '%')
  or franchises.name like concat('%', keyword, '%')
  or features.title like concat('%', keyword, '%');
end;
//

CALL content_search('heart');
```

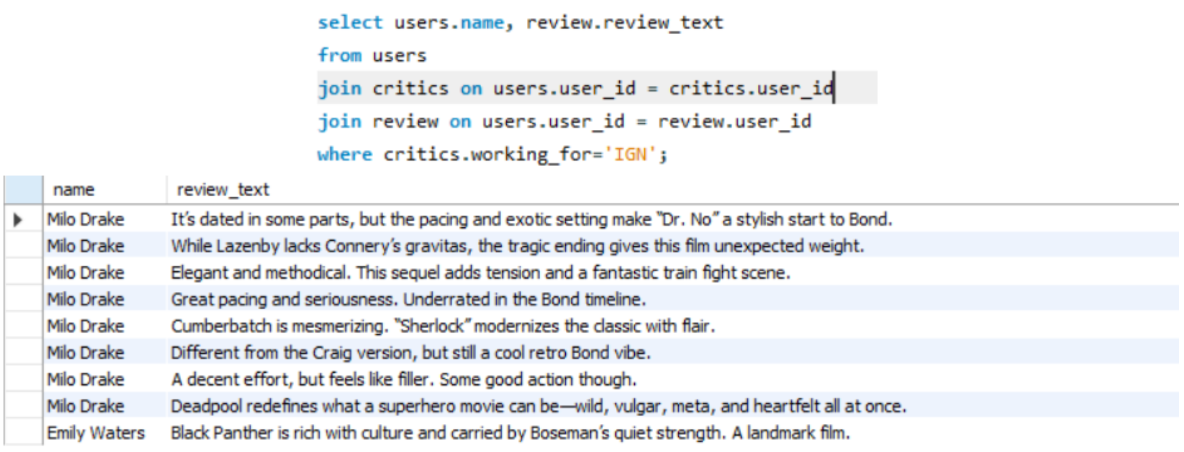
title	name	sweetness_percentage	synopsis	review_text	source
Black Panther: Wakanda Forever	Marvel Cinematic Universe	39.41	The people of Wakanda must unite after the ...	Wakanda Forever is a heartfelt tribute to Boseman. The story is a bit sca...	FlickFrame
Tomorrow Never Dies	James Bond	71.53	Bond is tasked with stopping media mogul Elli...	"Tomorrow Never Dies" leans into action-heavy sequences, but lacks heart.	IGN
John Wick: Chapter 4	John Wick	86.94	John Wick seeks vengeance against the High ...	A visceral conclusion filled with heartbreak and bloodshed. A stunning fin...	NME
Deadpool	Deadpool	79.05	A foul-mouthed antihero, Wade Wilson beco...	Deadpool redefines what a superhero movie can be—wild, vulgar, meta, ...	PopcornJunkie
The Matrix Revolutions	The Matrix	17.63	Neo and his allies launch a final battle against...	Too much philosophy, not enough heart. A clunky finale.	The Verge

Figure 17: Search Bar Functionality with Example

Example Queries: Database In Action

Procedural elements added more functionality to the database schema by introducing functions and procedures to perform tasks such as login and searching for content while views took query outputs that could be used often to query for logic when combined with other tables or frequently change. While this is important, the database needs to be able to query and output data from simple and complex queries. Most queries will remain queries due to the static nature of what the user will query.

The first query is to output the reviews made by critics who work in IGN. Figure 18 contains a screenshot of the code and output of this query. This is a relatively simple query which can filter out reviews to look for reviews made by a company the user trusts. This query can even be use with the earlier top_critic view to further filter out the results. The major reason the top_critic view has not been made into a query is because it not only accomplishes what this query aims to do, the query itself is static and is likely to not change outside of extreme circumstances.

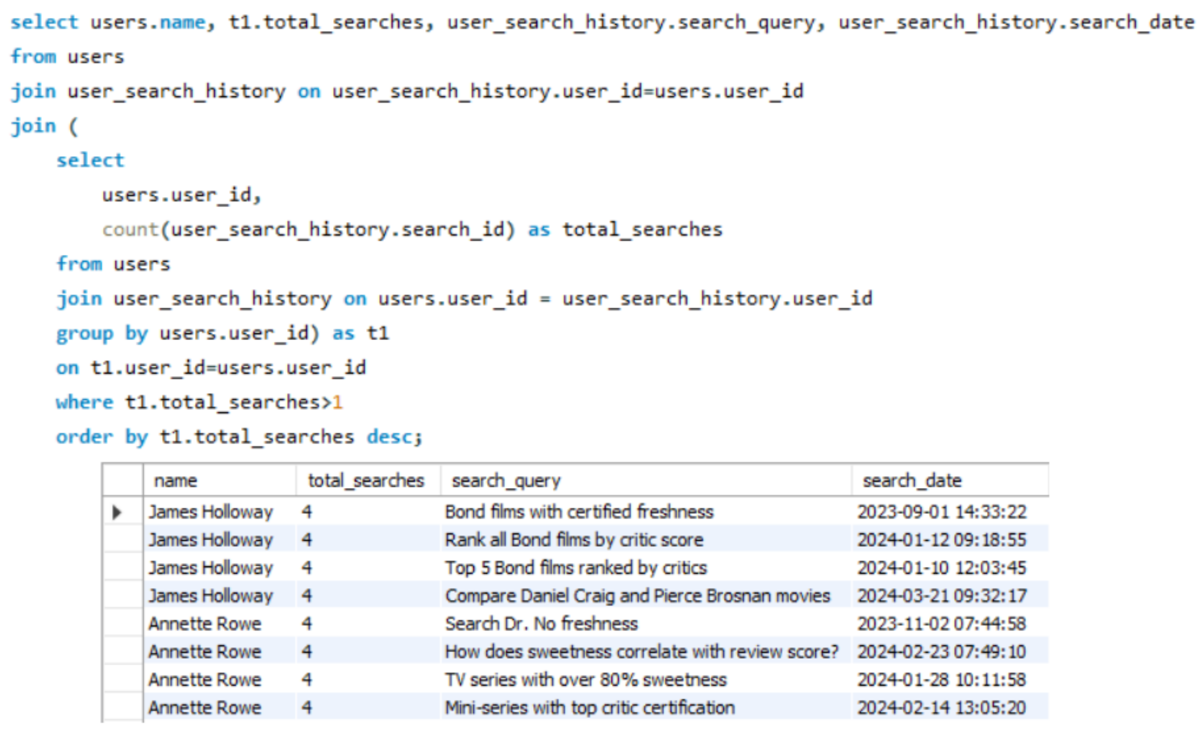


```sql
select users.name, review.review_text
from users
join critics on users.user_id = critics.user_id
join review on users.user_id = review.user_id
where critics.working_for='IGN';
```

name	review_text
Milo Drake	It's dated in some parts, but the pacing and exotic setting make "Dr. No" a stylish start to Bond.
Milo Drake	While Lazenby lacks Connery's gravitas, the tragic ending gives this film unexpected weight.
Milo Drake	Elegant and methodical. This sequel adds tension and a fantastic train fight scene.
Milo Drake	Great pacing and seriousness. Underrated in the Bond timeline.
Milo Drake	Cumberbatch is mesmerizing. "Sherlock" modernizes the classic with flair.
Milo Drake	Different from the Craig version, but still a cool retro Bond vibe.
Milo Drake	A decent effort, but feels like filler. Some good action though.
Milo Drake	Deadpool redefines what a superhero movie can be—wild, vulgar, meta, and heartfelt all at once.
Emily Waters	Black Panther is rich with culture and carried by Boseman's quiet strength. A landmark film.

Figure 18: IGN Review Query

In a similar vein to the above query, the next query outputs all users search history, how many searches they have done on the application and when they performed the search. However, the query will filter out users who have not performed more than one search. This can be useful from an analytical standpoint as the application programmers can track user engagement from searches and see what can be added feature (both media and application wise). This query also has the potential to made into a view because as users engage more with the database, there can be more data from an analytical perspective. However, this is more for developers rather than users to gather information about what s frequently searched, hence it remains as a query.

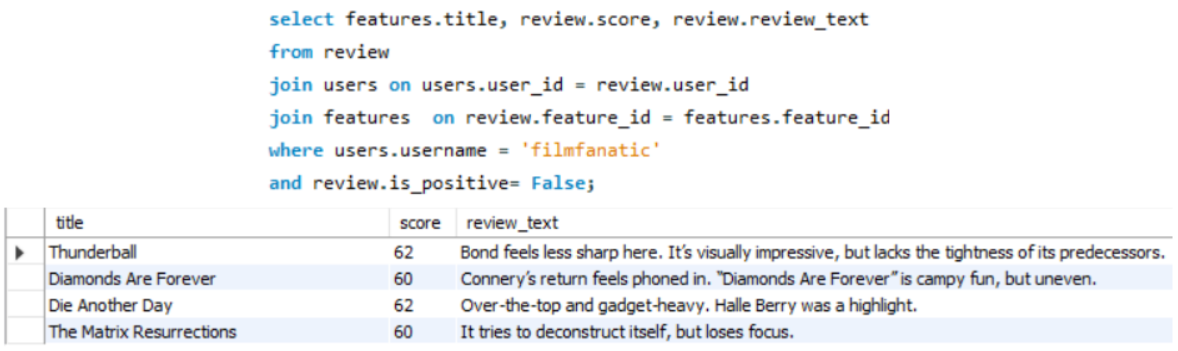


```sql
select users.name, t1.total_searches, user_search_history.search_query, user_search_history.search_date
from users
join user_search_history on user_search_history.user_id=users.user_id
join (
    select
        users.user_id,
        count(user_search_history.search_id) as total_searches
    from users
    join user_search_history on users.user_id = user_search_history.user_id
    group by users.user_id) as t1
    on t1.user_id=users.user_id
    where t1.total_searches>1
    order by t1.total_searches desc;
```

name	total_searches	search_query	search_date
James Holloway	4	Bond films with certified freshness	2023-09-01 14:33:22
James Holloway	4	Rank all Bond films by critic score	2024-01-12 09:18:55
James Holloway	4	Top 5 Bond films ranked by critics	2024-01-10 12:03:45
James Holloway	4	Compare Daniel Craig and Pierce Brosnan movies	2024-03-21 09:32:17
Annette Rowe	4	Search Dr. No freshness	2023-11-02 07:44:58
Annette Rowe	4	How does sweetness correlate with review score?	2024-02-23 07:49:10
Annette Rowe	4	TV series with over 80% sweetness	2024-01-28 10:11:58
Annette Rowe	4	Mini-series with top critic certification	2024-02-14 13:05:20

Figure 19: User's Seach Query

The third query looks through all the reviews and will find all the negative reviews made by one specific user, in this case by the user ‘filmfanatic’. When the user logs in or when another use clicks on their profile, the reviews made by filmfanatic can be displayed on their homepage, which provides quick access to features they reviewed. This query is very narrow in that sense it just focuses on a single user out of potential millions of users. Hence, it remains a query and not a view.

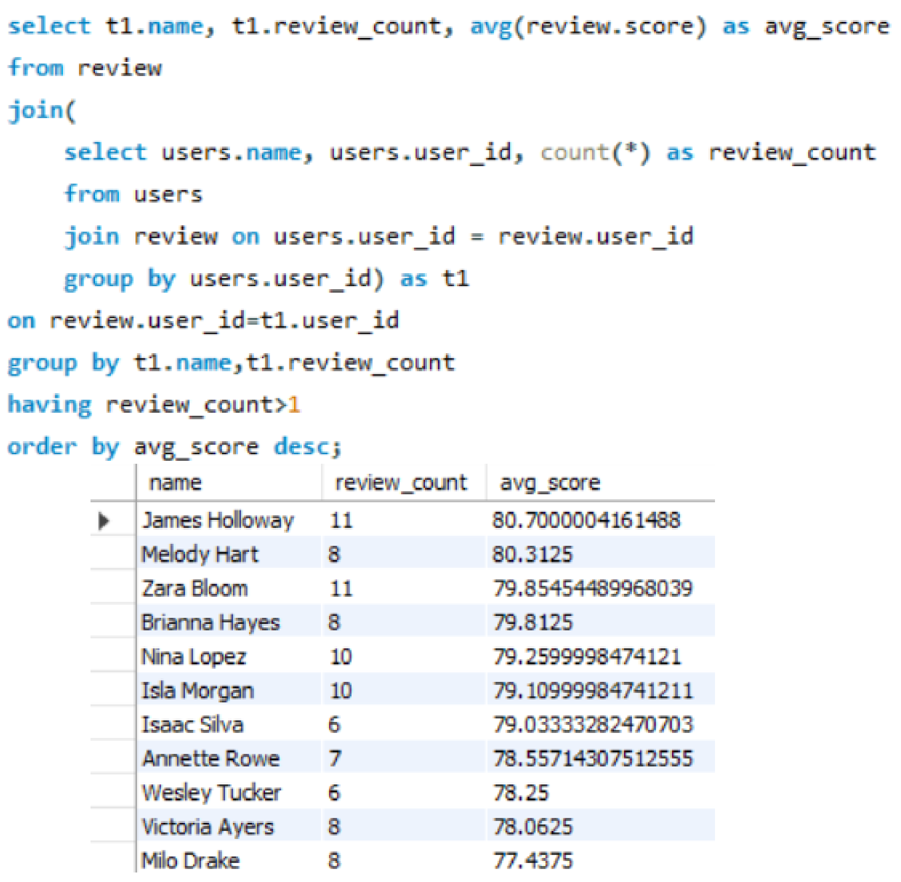


```sql
select features.title, review.score, review.review_text
from review
join users on users.user_id = review.user_id
join features  on review.feature_id = features.feature_id
where users.username = 'filmfanatic'
and review.is_positive= False;
```

title	score	review_text
Thunderball	62	Bond feels less sharp here. It’s visually impressive, but lacks the tightness of its predecessors.
Diamonds Are Forever	60	Connery’s return feels phoned in. "Diamonds Are Forever" is campy fun, but uneven.
Die Another Day	62	Over-the-top and gadget-heavy. Halle Berry was a highlight.
The Matrix Resurrections	60	It tries to deconstruct itself, but loses focus.

Figure 20: Negative Reviews by filmfanatic Query

Finally, the last query outputs the average score of a user across all reviews they have made. It will also output the number of reviews made by said user. The query once again limits this to users who have made more than one review. This is because if a user has only made a single review, they are more likely to be a new user, and their average score would just be the score of the single review they have made. The major reason this is not a view is because, while this can be dynamic as more users add reviews, there would not be much use with other tables as it primarily reports a metric as opposed to showing a relation.



```
select t1.name, t1.review_count, avg(review.score) as avg_score
from review
join(
    select users.name, users.user_id, count(*) as review_count
    from users
    join review on users.user_id = review.user_id
    group by users.user_id) as t1
on review.user_id=t1.user_id
group by t1.name,t1.review_count
having review_count>1
order by avg_score desc;
```

name	review_count	avg_score
James Holloway	11	80.7000004161488
Melody Hart	8	80.3125
Zara Bloom	11	79.85454489968039
Brianna Hayes	8	79.8125
Nina Lopez	10	79.2599998474121
Isla Morgan	10	79.10999984741211
Isaac Silva	6	79.0333282470703
Annette Rowe	7	78.55714307512555
Wesley Tucker	6	78.25
Victoria Ayers	8	78.0625
Milo Drake	8	77.4375

Figure 21: Average Review Score by Critic

Conclusions

In conclusion, the database schema employs 6 new tables in sweetness_index, review, users, critics, user_search_history and feature_synopsis, all of which provide important functionalities for the review aggregate site while working with the previously included tables. With the addition of the new tables, BCNF is still maintained due to all functional dependencies being trivial or making use of a superkey. As BCNF holds for the schema, 1NF, 2NF and 3NF are also all achieved. The procedural elements included within the project helps add important functionality such as searching for content based on keyword/phrase, updating the sweetness score every time a new review is added, logging in and password resetting. Various views and queries were created to help provide analysis for the administrators of the application or to help provide dynamic tables that, when combined with the base tables, help query information faster.

While many tables and functionalities have been added to the schema, there are a few things to consider for future of this application. From a security perspective, the login procedure itself is very insecure as there are unlimited login attempts. While displaying a custom message helps add a layer of security, if the website itself does not handle this, it would leave the schema vulnerable to stealing of data. Another thing to consider would be the addition of new tables to expand what kind of information is stored. Currently, information is stored on features and its related information, types of roles, people who worked on the movies, budget and gross, reviews and user information. We could add further information about the people who worked on the movie (such as date/place of birth) or even add location of where the feature has been shot to provide geographical information. However, as said by Coronel and Morris, compromises are to be made when designing a database (Coronel & Morris 2018) and as this adds a layer of unnecessary complications, it is not important to add. Another issue to keep in mind is that as the database grows, efficiency will need to be raised so the application is not slowed down. While not a real-time database, the database aims to retrieve and present users with data smoothly and without latency issues (Alvarez et al. 2018). Currently, the queries are fast and efficient but that is attributed to the size of the data and number of records currently stored. Time will tell how well BitterMelons will grow but in its current state, there are many hurdles to overcome and things to keep in mind but if overcome, can act as a competitor to RottenTomatoes.

Acknowledgements

I acknowledge that I have not worked with any other person during this project and any work I have used has been cited and referenced correctly in the References section. ChatGPT was only used to generate dummy data to fill the reviews, users, critics, user_search_history, sweetness_index and feature_synopsis.

References

1. Kroenke, D. M., & Auer, D. J. (2013). Database processing: Fundamentals, design, and implementation (13th ed.). Pearson Education.
2. Halpin, T. A., & Morgan, T. (2008;2010;). *Information modeling and relational databases* (2nd;2; ed.). Morgan Kaufmann Publishers.
3. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database system concepts* (7th ed.). McGraw-Hill Education.
4. Harrington, J. L. (2016). *Why good design matters in Relational database design and implementation* (4th ed). Morgan Kaufmann

5. Connolly, T. & Begg, C. (2015). *Database Systems: A Practical Approach to Design,* Implementation, and Management (6th ed.). Pearson Education
6. DuBois, P. (2013). *MySQL*. Pearson Education.
7. Coronel, C., & Morris, S. (2018). Database Systems: Design, Implementation*, & Management*, (13th ed.), Cengage
8. Mejia Alvarez, P., Zavaleta Vazquez, R. J., Ortega Cisneros, S., & Gonzalez Torres, R. E. (2024;2023;). *Real-time database systems: Fundamentals, architectures and applications* (1st ed.). Springer.