MET CS 669 Database Design and Implementation for Business
Term Project Iteration 6

Name:

Table of Contents

Project Direction Overview

I would like to create a web application designed to track a user's media consumption across several different mediums. The application, which I will call "TunedIn," will be a place where users can log the different media they're currently consuming, have consumed, or plan to consume. The media in question can be movies, TV shows, books, music (in album form), and video games. Additionally, the user will be able to track information about the vendor of the media and use this information to organize and revisit their media buys as well as give recommendations to others

I believe an application like this is important because an overall media tracking system does not appear to exist, leaving people to log their consumption on multiple different websites dedicated to only a specific medium (think Letterboxd or MyAnimeList), or create a Twitter thread or blog dedicated to tracking, or alternatively not track their watches/reads/listens at all. As someone who considers herself to be scatterbrained, I also often jump from media to media without finishing a series or completing a playthrough, and I forget what episode I left on or what chapter of a book I was on, and sometimes I completely forget that I was in the middle of watching or reading something. When people ask me what I've been watching or reading or playing, or what my favorite movie or show is, I often blank and fail to answer the question. I can safely presume that I am not the only one with this issue, and therefore an application dedicated to tracking the status of someone's varied media consumption is a convenient solution. I also believe that tracking media consumption encourages users to think more critically about what they are consuming and how they feel about it, rather than finishing a series or book or game and simply moving on to the next one.

TunedIn will offer a place for users to input information about a piece of media, the main focus being on a name, a rating (if completed/desired), and a status on the consumption. Additional information can be added based on relevance, such as author information for a book, or the URL for something consumed online so it can easily be provided to a friend when recommending the media to them, or a link to another existing database-esque website which may contain a blurb/critic ratings of the media (again, think Letterboxd or MyAnimeList or IMDB, etc), and more. Users will be able to log what chapter of a book they are on, or what episode of a show they are on, so that if they drop it and choose to resume it later, they can easily pick up from where they left off. Additionally, users can track when they started and finished a piece of media, to log past watches, and/or determine if it might be time to revisit something they previously consumed and enjoyed. Each kind of media has information that is specific to it, as well as information that can be applicable to multiple kinds of media, such as a genre and a review. Through my multiple project iterations, I will be able to refine my scope and determine what information is most necessary and relevant for the user to have logged, and create the most useful version of my database.

Use Cases and Fields

Use Case #1: Account Setup

Because TunedIn will be a web application, it will be necessary to register to keep track of each individual user's media.

Field	What it stores	Why it's needed
username	This stores a username	Users can share usernames to

	associated with each account	see what another user is consuming, and users can also have multiple accounts if they would prefer
first_name	This stores the user's first name.	This can be displayed on screen and shared with other users.
last_name	This stores the user's first name.	This can be displayed on screen and shared with other users.
account_created	This stores the date on which the user created the account.	Users may want to know how long they have been using the application to track their media consumption.

Use Case #2: Logging a Piece of Media
This is the supertype that users will log that contains common information across all media types before logging the specific subtype.

Field	What it stores	Why it's needed
media_name	This stores the name of the media.	This is necessary for the user to know what media they consumed.
date_released	This stores the date that the media was released.	This helps the user determine their viewing trends by date and also distinguish between potential media remakes or remasters with different release dates.

Use Case #3: Logging a Movie
A movie will be one of the media types that the application will encourage users to log, and has fields that are specific to it. The other media types will track similar information and will be able to be differentiated in the database using a flag.

Field	What it stores	Why it's needed
director	This stores the name of the director of the movie.	This is useful for the user if they want to track how much they like movies by a certain director

		or want to sort by a certain director.
production_company	This stores the name of the production company of the movie, such as A24 or Studio Ghibli.	This is useful for the user to track how much they like the works of a certain company or sort by them.

Use Case #4: Logging a Genre
Since genres are applicable to any media type, users will be able to log them for any media consumed.

Field	What it stores	Why it's needed
category	This is a catch-all for the different media types where the category/large genres can be stored, such as things like fiction or nonfiction for books, documentary or short film for movies, RPG or shooter for video games, etc.	This allows for users to sort and search for certain pieces of media in a broad way.
main_genre	This stores the primary genre of a piece of media, which can differ in definition based on the user, but is generally a broader category such as horror, action, etc.	This allows for users to sort media by genre when organizing and also looking for previously logged media.
subgenre	This stores the subgenre of a piece of a media, which can be a specifier, like historical fantasy vs urban fantasy, etc.	This is optional but allows for more specific searching and sorting for the user.

Use Case #5: Logging a Review
Since reviews are applicable to any media type, users will be able to log them for any media consumed.

Field	What it stores	Why it's needed
out_of_ten	This stores a number rating out of 10.	This is a typical method of rating not just media, but anything, and will let users reflect on what they enjoyed or disliked.

review	This stores a text review.	This lets the user give clarification on the number rating and offers more information about the media.
review_link	This stores an optional link that leads to an external blog review or a website like Letterbox or Goodreads that the user may have posted their review on.	This lets the user organize relevant info in one place by allowing them to keep track of different websites they may have used to review a piece of media.

Use Case #6: Logging a Status
Each piece of media will have a consumption status that can be logged by the user.

Field	What it stores	Why it's needed
overall_status	This stores a general status like Planned, In Progress, Completed, Dropped, or On Hold.	This helps users keep track of all the media they want to watch, are watching, and have watched, as well as media they chose to not complete and media they plan to return to, which is handy for not forgetting their progress.
section	This stores a larger indicator of progress, such as a season of a TV show, a chapter for a book, a section/act of a video game, etc	This is helpful for giving context to the part of the media the user is on, or is an indicator that they finished it.
subsection	This stores a smaller indicator of progress, such as an episode of a TV show, a page of a book, a level of a video game, etc.	This is helpful for tracking the exact part of a piece of media that the user left off on so they can jump back in.
date_started	This stores the date when the user started consuming the media.	This is helpful for the user to know when they first began a piece of media and for tracking how long it is taking them to complete it.
date_finished	This stores the last date the user interacted with/finished the media.	This is helpful for tracking the amount of time the user has spent with the media and for

		tracking how long ago they interacted with the media to see if it is worth a revisit.

Use Case #7: Logging a Vendor
Media is offered by multiple vendors and these vendors can be frequented by users, and therefore are logged in the database.

Field	What it stores	Why it's needed
vendor_name	This stores the name of the vendor.	This is needed for the user to know what vendors they buyt from.
vendor_type	This stores the type of vendor, such as website, streaming service, or physical storefront.	This is useful for the user to know where they acquired a piece of media.
vendor_link	This stores a link to the vendor website, if available.	This can help the user track past visits to an online vendor and allow them to revisit it or recommend it.

Use Case #8: Vendor-Media Relationship
This connects the vendor to the piece of media and provides useful information about the purchase.

Field	What it stores	Why it's needed
vendor_name	This stores the name of the vendor and will reference the vendor table.	This lets users know where they bought or can buy a piece of media.
date_purchased	This stores the date of the purchase of the media from the vendor.	This can assist with tracking the consumption of media by time.
media_link	This stores a direct link to the media on the vendor site, if available/applicable	This can help the user revisit their purchase easier or recommend the purchase to a friend.
price	This stores the price of the media purchase, if applicable	This can help users budget and track their money spent on media and also play into their

		review of the media by determining if it was worth the price they paid for it.

Use Case #9: Series, Anthology, Collection
Some pieces of media are part of a series or anthology or collection, such as a book series or short film anthology, and users can specify which series a piece of media is part of.

Field	What it stores	Why it's needed
SAC_name	This stores the name of the series, anthology, collection or other extended body of work that a piece of media is part of.	This helps users see how their media is connected as well as track their consumption of a series or the like.
sub_SAC_name	This is optional and stores the name of a subseries/sub-extended body of work within a larger series.	This further helps users organize and search for media.
SAC_part	This is optional and stores a number to indicate which part of a chronological series the piece of media is (such as the second book in a trilogy, etc)	This helps users track completion of a connected/chronological body of media.

Use Case #10: Rating
Most pieces of media have a rating, and users will be able to log this.

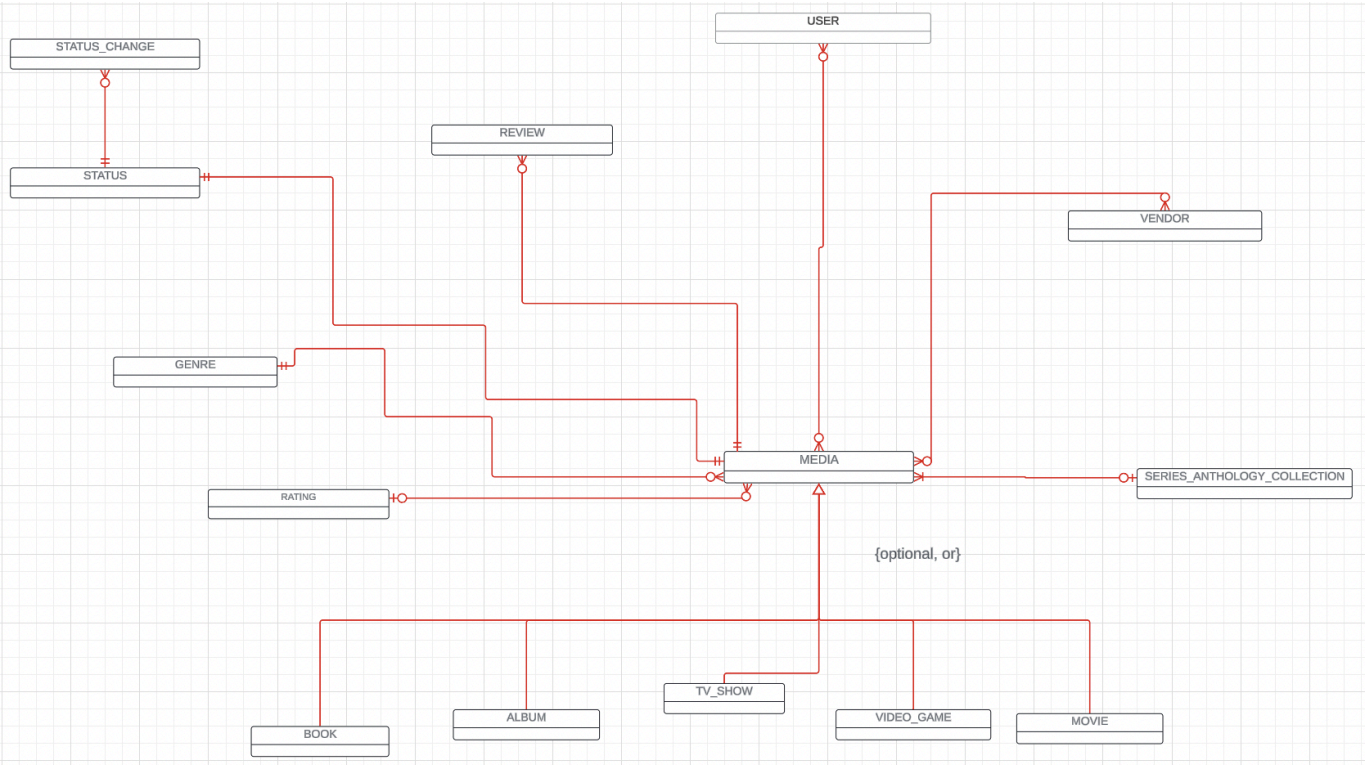
Field	What it stores	Why it's needed
rating	This stores the rating of a piece of media, such as G or PG-13 or R in the case of movies, E or T in the case of video games, etc.	This allows users to sort by the rating of the media they are looking for and keep track of their consumption trends.

Structural Database Rules

1) Each user may log zero to many pieces of media, and each piece of media may be logged by zero to many users.
2) A piece of media is a book, album, TV show, movie, video game, or none of these.

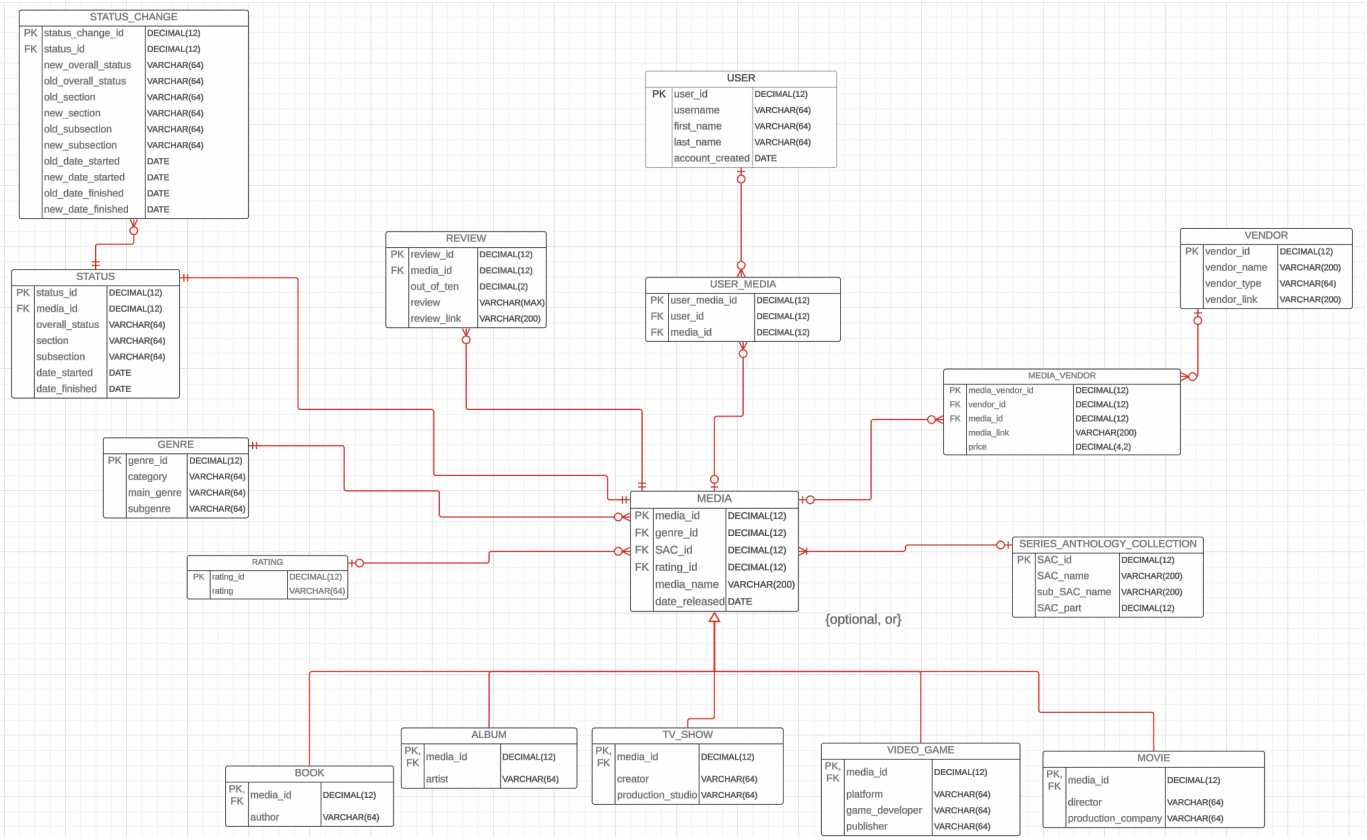
3) Each piece of media may have zero to many reviews, and each review will apply to one piece of media.
4) Each piece of media will have one status, and each status will apply to one to many pieces of media.
5) Each piece of media will have one genre, and each genre can apply to zero to many pieces of media.
6) Each piece of media can be part of one series/anthologies/collections, and each series/anthology/collection can contain many pieces of media.
7) Each piece of media can be purchased/rented from zero to many vendors, and each vendor can offer zero to many pieces of media.
8) Each piece of media may have one rating, and each rating can apply to zero to many pieces of media.
9) Each status may have one to many status changes, and each status change will apply to one status.

Conceptual Entity-Relationship Diagram



This is the conceptual ERD updated to include the Status_change entity connected to the Status entity.

Full DBMS Physical ERD



This is the physical ERD updated to include the attributes of the Status_change history table.

Stored Procedure Execution and Explanations

```
-- adding users
CREATE OR REPLACE PROCEDURE add_user(
    username_arg IN VARCHAR,
    first_name_arg IN VARCHAR,
    last_name_arg IN VARCHAR)
    LANGUAGE plpgsql
AS
$$
BEGIN
    INSERT INTO Users(user_id, username, first_name, last_name, account_created)
    VALUES(nextval('user_seq'), username_arg, first_name_arg, last_name_arg, CURRENT_DATE);
END;
$$;

START TRANSACTION;
DO
$$ BEGIN
    CALL add_user('msamala', 'Megha', 'Samala');
    CALL add_user('nbardhan', 'Neha', 'Bardhan');
    CALL add_user('aismail', 'Afra', 'Ismail');
    CALL add_user('sanand', 'Soummitra', 'Anand');
    CALL add_user('mkoleti', 'Manish', 'Koleti');
    CALL add_user('tnguyen', 'Tim', 'Nguyen');
END $$;
COMMIT TRANSACTION;
```

```sql
-- adding rating
CREATE OR REPLACE PROCEDURE add_rating(
    rating_arg IN VARCHAR)
    LANGUAGE plpgsql
AS
$$
BEGIN
    INSERT INTO Rating(rating_id, rating)
    VALUES(nextval('rating_seq'), rating_arg);
END;
$$;

START TRANSACTION;
DO
$$ BEGIN
    CALL add_rating('TV-Y');
    CALL add_rating('TV-Y7');
    CALL add_rating('TV-G');
    CALL add_rating('TV-PG');
    CALL add_rating('TV-14');
    CALL add_rating('TV-MA');
    CALL add_rating('G');
    CALL add_rating('PG');
    CALL add_rating('PG-13');
    CALL add_rating('R');
    CALL add_rating('NC-17');
    CALL add_rating('E');
    CALL add_rating('E10+');
    CALL add_rating('T');
    CALL add_rating('M');
    CALL add_rating('NR');
    CALL add_rating('Explicit');
END $$;
COMMIT TRANSACTION;
```

```sql
-- adding a media as a book
CREATE OR REPLACE PROCEDURE add_book(
    media_name_arg IN VARCHAR,
    date_released_arg IN DATE,
    author_arg IN VARCHAR) LANGUAGE plpgsql
AS
$$
DECLARE
    v_media_id DECIMAL(12);
BEGIN
    SELECT media_id
    INTO v_media_id
    FROM Media
    WHERE media_name = media_name_arg AND date_released = date_released_arg;

    INSERT INTO Book(media_id, author)
    VALUES(v_media_id, author_arg);
END;
$$;

START TRANSACTION;
DO
$$ BEGIN
    CALL add_book('The Honjin Murders', CAST('04-AUG-2020' AS DATE), 'Seishi Yokomizo');
    CALL add_book('Death on Gokumon Island', CAST('05-JUL-2022' AS DATE), 'Seishi Yokomizo');
    CALL add_book('Mistborn: The Final Empire', CAST('17-JUL-2006' AS DATE), 'Brandon Sanderson');
    CALL add_book('Jujutsu Kaisen', CAST('05-MAR-2018' AS DATE), 'Gege Akutami');
    CALL add_book('Bleach', CAST('07-AUG-2001' AS DATE), 'Tite Kubo');
END $$;
COMMIT TRANSACTION;
```

Pictured above are some of the stored procedures I have implemented in my project to assist with populating the database tables. I have a variety of stored procedures to represent different use cases that I have outlined for my project, such as a procedure for adding a new user account, a procedure for adding content ratings to select from, and more, and the comments above each procedure talk about the purpose of the procedure. My database is populated using the stored procedures as well as manual inserts. All of the stored procedures can be viewed in my SQL script file. Additionally, I added error checking functionality, as shown below:

```
169 -- throw error if username is already taken when creating new user account
170 CREATE OR REPLACE FUNCTION user_username_func()
171     RETURNS TRIGGER LANGUAGE plpgsql
172     AS $$
173     BEGIN
174         IF EXISTS(
175             SELECT username FROM Users WHERE username = NEW.username
176         ) THEN
177             RAISE EXCEPTION USING MESSAGE = 'Username is already taken',
178             ERRCODE = 23000;
179         END IF;
180         RETURN NEW;
181     END;
182     $$;
183 CREATE TRIGGER user_username_trg
184 BEFORE UPDATE OR INSERT ON Users
185 FOR EACH ROW
186 EXECUTE PROCEDURE user_username_func();
187
188 INSERT INTO Users(user_id, username, first_name, last_name, account_created)
189 VALUES(nextval('user_seq'), 'msamala', 'Sdnfsdf', 'Hjsdfjb', CURRENT_DATE);
190

Data Output   Messages   Notifications

ERROR:  Username is already taken
CONTEXT:  PL/pgSQL function user_username_func() line 6 at RAISE

SQL state: 23000
```

This function makes it such that usernames cannot be reused for different accounts.

Question Identification and Explanations

First Query: Which subscription service is the most used for watching TV shows? This question is useful because it allows users to determine which subscription service appears to have the most media available, and is therefore the most worth paying for compared to the others. Additionally, if a subscription service is being used less, users can be made aware and can choose to cancel their subscription.

Second Query: What is the best reviewed video game that falls under the “fantasy” genre? This question is useful because it is a good example of the kinds of questions users will ask while using the database to determine what piece of media they might choose to interact with next. Creating a query for this question allows for users to see what high quality media, specifically video games, is out there being recommended to them in a genre of their choice.

Third Query: What is the average rating given to a piece of media by each user? This question is useful because users can see which users are harsher critics than others, and use the average rating given by each user to determine which users are consuming more quality media, and then use that information to determine what media they might interact with next - they may take recommendations from users that give a higher rating to media on average.

Query Executions and Explanations

First Query:

```
908 -- first query: Which subscription service is the most used for watching TV shows?
909 SELECT vendor_name, COUNT(*) AS use_count
910 FROM Vendor
911 JOIN Media_vendor ON Media_vendor.vendor_id = Vendor.vendor_id
912 JOIN Media ON Media.media_id = Media_vendor.media_id
913 JOIN TV_show ON TV_show.media_id = Media.media_id
914 GROUP BY vendor_name;
915
```

Data Output | Messages | Notifications

	vendor_name character varying (200)	use_count bigint
1	Netflix	3
2	HBO Max	1

The query that answers the first question has 4 tables joined: Vendor, Media_vendor, Media, and TV_show, and uses the COUNT function to count how many TV shows were watched on each streaming subscription service. From this query, it is clear that Netflix is the most used service for watching TV shows.

Second Query:

```
941 -- second query: What is the best reviewed video game that falls under the “fantasy” genre?
942 SELECT media_name, out_of_ten
943 FROM Review
944 JOIN Media ON Media.media_id = Review.media_id
945 JOIN Genre ON Genre.genre_id = Media.genre_id
946 JOIN Video_game on Video_game.media_id = Media.media_id
947 WHERE main_genre = 'Fantasy' OR subgenre = 'Fantasy'
948 ORDER BY out_of_ten DESC;
949
```

Data Output | Messages | Notifications

	media_name character varying (200)	out_of_ten numeric (2)
1	Final Fantasy VII Remake	9
2	Final Fantasy XVI	9
3	Fire Emblem: Three Houses	8

This query joins the Video_game subtype with the Media supertype and the Review and Genre tables to answer the question. The joins combined with the WHERE clause limits the results to video games that only have “Fantasy” listed as one of the genres, and the ORDER BY clause puts the highest rated video game at the top of the result set. From this query, it is shown that Final Fantasy VII Remake and Final Fantasy XVI are tied for the highest rated fantasy video games in the database.

Third Query:

```sql
-- third query: What is the average rating given to a piece of media by each user?
CREATE OR REPLACE VIEW User_ratings AS
SELECT username, media_name, out_of_ten
    FROM Review
    JOIN Media ON Media.media_id = Review.media_id
    JOIN User_media ON User_media.media_id = Media.media_id
    RIGHT JOIN Users ON Users.user_id = User_media.user_id
    ORDER BY username, out_of_ten DESC;

SELECT user_ratings.username, ROUND(AVG(out_of_ten), 2) AS avg_rating
FROM User_ratings
GROUP BY user_ratings.username;
```

Data Output | Messages | Notifications

	username character varying (64)	avg_rating numeric
1	aismail	8.25
2	mkoleti	[null]
3	msamala	8.40
4	nbardhan	9.00
5	sanand	8.00
6	tnguyen	7.00

The view I created is called User_ratings and shows all of the ratings out of 10 each user has given to pieces of media, and includes all users regardless of if they have reviewed any media or not by using the RIGHT JOIN on the Users table. The ORDER BY clause displays the results in alphabetical order of usernames, and descending order of ratings out of 10 within each username. Using the view in my query, I used the AVG aggregate function to retrieve the average rating given by each username to a piece of media. From this we see that some users give lower ratings on average than others, and one user has not rated any media yet.

Index Identification and Creations

The primary keys of my database are already indexed, and the foreign keys of my database need to be indexed. I have concluded that most of the foreign keys do not need unique indexes because most of them are non-unique - for example, foreign keys such as genre_id and SAC_id, among others, do not need unique indexes because many pieces of media can fall under the same genre or series/anthology/collection. The only foreign key that needs to have a unique index is Status.media_id because of the 1:1 relationship between Status and Media. Additionally, in one of my queries I used main_genre and subgenre in the WHERE clause, and therefore will be giving non-unique indexes to these two columns in the Genre table. Below is the creation of the indexes:

```sql
-- index creation
CREATE INDEX media_genre_idx
ON Media(genre_id);
CREATE INDEX media_SAC_idx
ON Media(SAC_id);
CREATE INDEX media_rating_idx
ON Media(rating_id);
CREATE INDEX user_media_user_idx
ON User_media(user_id);
CREATE INDEX user_media_idx
ON User_media(media_id);
CREATE INDEX media_vendor_media_idx
ON Media_vendor(media_id);
CREATE INDEX media_vendor_vendor_idx
ON Media_vendor(vendor_id);
CREATE INDEX review_media_idx
ON Review(media_id);
CREATE UNIQUE INDEX status_media_idx
ON Status(media_id);
CREATE INDEX book_media_idx
ON Book(media_id);
CREATE INDEX album_media_idx
ON Album(media_id);
CREATE INDEX tv_show_media_idx
ON TV_show(media_id);
CREATE INDEX video_game_media_idx
ON Video_game(media_id);
CREATE INDEX movie_media_idx
ON Movie(media_id);
```

Data Output | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 175 msec.
```

```sql
-- query-driven indexes
CREATE INDEX main_genre_idx
ON Genre(main_genre);
CREATE INDEX main_genre_idx
ON Genre(subgenre);
```

Data Output | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 175 msec.
```

History Table Demonstration

Pictured below is my history table for the Status table, called Status_history. This table is important because users will update their status based on how far they've progressed through a piece of media and if they've just started or finished it.

```
1043  -- history table
1044  CREATE TABLE Status_history (
1045      status_change_id DECIMAL(12) PRIMARY KEY NOT NULL,
1046      status_id DECIMAL(12) NOT NULL,
1047      new_overall_status VARCHAR(64),
1048      old_overall_status VARCHAR(64),
1049      new_media_section VARCHAR(64),
1050      old_media_section VARCHAR(64),
1051      new_media_subsection VARCHAR(64),
1052      old_media_subsection VARCHAR(64),
1053      new_date_started DATE,
1054      old_date_started DATE,
1055      new_date_finished DATE,
1056      old_date_finished DATE,
1057      FOREIGN KEY (status_id) REFERENCES Status (status_id)
1058      );
1059
1060  CREATE SEQUENCE status_history_seq START WITH 1;
```

Below is my trigger creation featuring the old and new versions of all of the possible fields a user could update in the Status table, and this trigger goes off on the update of any of those fields in the status table. As soon as any part of a status is updated, the update history is logged in the Status_history table.

```
1062  CREATE OR REPLACE FUNCTION Status_history_func()
1063  RETURNS TRIGGER LANGUAGE plpgsql
1064  AS $trigfunc$
1065      BEGIN
1066          INSERT INTO Status_history(
1067          status_change_id,
1068          status_id,
1069          new_overall_status,
1070          old_overall_status,
1071          new_media_section,
1072          old_media_section,
1073          new_media_subsection,
1074          old_media_subsection,
1075          new_date_started,
1076          old_date_started,
1077          new_date_finished,
1078          old_date_finished)
1079          VALUES(
1080          nextval('status_history_seq'),
1081          NEW.status_id,
1082          NEW.overall_status,
1083          OLD.overall_status,
1084          NEW.media_section,
1085          OLD.media_section,
1086          NEW.media_subsection,
1087          OLD.media_subsection,
1088          NEW.date_started,
1089          OLD.date_started,
1090          NEW.date_finished,
1091          OLD.date_finished);
1092      RETURN NEW;
1093      END;
1094  $trigfunc$;
1095
1096  CREATE TRIGGER Status_history_trg
1097  BEFORE UPDATE OF overall_status, media_section, media_subsection, date_started, date_finished ON Status
1098  FOR EACH ROW
1099  EXECUTE PROCEDURE Status_history_func();
```

In order to check that my trigger is working, I updated one of the existing statuses in the table 3 times with different information:

```
UPDATE Status
SET overall_status = 'In Progress', media_section = 'Chapter 6', date_started = CURRENT_DATE
WHERE status_id =
    (SELECT status_id
     FROM Status
     WHERE media_id =
     (SELECT media_id
     FROM Media
     WHERE media_name = 'Resident Evil 4 Remake'
      AND date_released = CAST('24-MAR-2023' AS DATE)));

UPDATE Status
SET media_section = 'Chapter 14'
WHERE status_id =
    (SELECT status_id
     FROM Status
     WHERE media_id =
     (SELECT media_id
     FROM Media
     WHERE media_name = 'Resident Evil 4 Remake'
      AND date_released = CAST('24-MAR-2023' AS DATE)));

UPDATE Status
SET overall_status = 'Completed', media_section = 'Chapter 16', date_finished = CAST('30-JUN-2024' AS DATE)
WHERE status_id =
    (SELECT status_id
     FROM Status
     WHERE media_id =
     (SELECT media_id
     FROM Media
     WHERE media_name = 'Resident Evil 4 Remake'
      AND date_released = CAST('24-MAR-2023' AS DATE)));
```

Data Output | Messages | Notifications

```
UPDATE 1

Query returned successfully in 90 msec.
```

After selecting all from the Status_history table, it is shown that the updates are all logged in the table, proving that the trigger is working correctly and all changes to statuses are being logged without any missing. Users can easily use the database to track their real-time progress on a piece of media while their past statuses are remembered.

```
SELECT * FROM Status_history;
```

Data Output | Messages | Notifications

	status_change_id [PK] numeric (12)	status_id numeric (12)	new_overall_status character varying (64)	old_overall_status character varying (64)	new_media_section character varying (64)	old_media_section character varying (64)	new_media_subsection character varying (64)	old_media_subsection character varying (64)	new_date_sta date
1	1	19	In Progress	Planned	Chapter 6	[null]	[null]	[null]	2024-06-18
2	2	19	In Progress	In Progress	Chapter 14	Chapter 6	[null]	[null]	2024-06-18
3	3	19	Completed	In Progress	Chapter 16	Chapter 14	[null]	[null]	2024-06-18

Data Visualizations

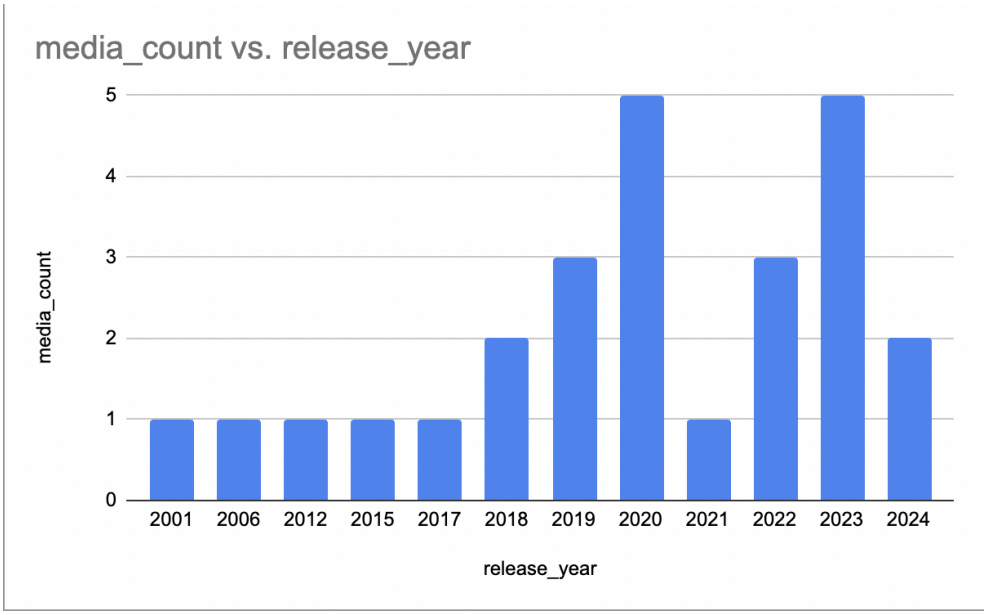
One useful question to ask regarding my database is: How recent is the media that users are interacting with? This can offer useful insights to users who want to see what kind of media is trending amongst other users, as well as see their own viewing habits and determine what era of media they might prefer. The query I used was the following:

```
1136 -- visualization queries
1137 SELECT date_part('year', date_released)::DECIMAL AS release_year, COUNT(*) as media_count
1138 FROM Media
1139 GROUP BY date_part('year', date_released)
1140 ORDER BY release_year;
1141
```

Data Output | Messages | Notifications

	release_year numeric	media_count bigint
1	2001	1
2	2006	1
3	2012	1
4	2015	1
5	2017	1
6	2018	2
7	2019	3
8	2020	5
9	2021	1
10	2022	3
11	2023	5
12	2024	2

As there are only 2 columns of results in this query, I opted to use a bar chart to create the following visualization after importing the results csv into Google Sheets:



From this bar chart, we can see that users have logged media across 12 years of media release years, and it is clear that more recent media is popular compared to media from earlier in the 2000s and before, with the most media being logged from the years of 2020 and 2023.

A second useful question is: How many pieces of media have each of the users completed, planned to interact with, or are currently consuming? This is a useful question because it dissects the way each user consumes media as well as how they might be using the app; for example, some users only like tracking media that is completed or planned, while others might update the app with their progress status more often. Users can also see a total number of media they have logged thus far, and this way they may see how much they interact with media in general and if they want to continue at that pace or cut back.
The query I used for this question is the following (some results are not pictured for brevity):

```
1149  SELECT username, overall_status, COUNT(*) as status_count
1150  FROM Users
1151  JOIN User_media ON User_media.user_id = Users.user_id
1152  JOIN Media ON Media.media_id = User_media.media_id
1153  JOIN Status ON Status.media_id = Media.media_id
1154  GROUP BY username, overall_status
1155  ORDER BY username;
1156
```

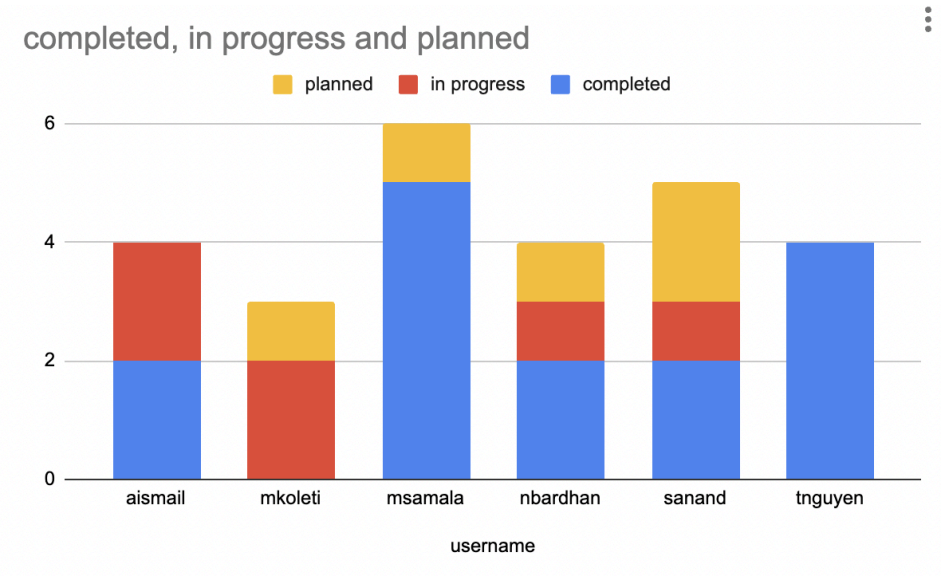
Data Output | Messages | Notifications

	username character varying (64)	overall_status character varying (64)	status_count bigint
1	aismail	Completed	2
2	aismail	In Progress	2
3	mkoleti	In Progress	2
4	mkoleti	Planned	1
5	msamala	Completed	5
6	msamala	Planned	1
7	nbardhan	Completed	2
8	nbardhan	In Progress	1
9	nbardhan	Planned	1
10	sanand	Completed	2
11	sanand	In Progress	1
12	sanand	Planned	2

In order to create the visualization for this set of results, I imported the csv into Google Sheets and pivoted the table to make it such that each “overall status” had its own column:

username	completed	in progress	planned
aismail	2	2	0
mkoleti	0	2	1
msamala	5	0	1
nbardhan	2	1	1
sanand	2	1	2
tnguyen	4	0	0

This then made it possible to create a stacked bar chart:



From this chart we can see the total number of media each user has logged, and we can see which users are logging the most and least. Additionally, it is clear that some users prefer to only log completed or planned media, while some exclusively log their media in progress. This visualization shows the variety of users that the database application can cater to.

Summary and Reflection

My database is for a web application called “TunedIn” in which users can track their media consumption across various mediums, as well as media purchases from various vendors. The application allows users to rate their media and track their progress in consuming it.

This week I added a history table and trigger function to my database to track updates to the Status table of my database, and I also came up with more relevant questions for my database to create queries based off of them, and visualizations based on these queries. All changes I’ve made can be viewed in the SQL script file.

As this is the last project iteration for the course, I am happy to see how far I have come with database design and development, and I am excited to potentially develop this database as part of an actual web application after the course finishes. While there may still be some issues with my database, I am happy with its functionality and I believe with more development, it will actually be useful to not only myself, but others as well. I am open to any suggestions and feedback on my database.