Polybius: Video Game Database

The Ocelots
lr4631: Lily Ready
brl7498: Brady Lax
jak9527: Jacob Karvelis
lek7518: Lydia Klecan
cid3081: Camille Decker

April 20, 2023

1 Introduction

This project takes place in the Video Game domain. The application itself highlights the collection of favorite games and online friends while providing the user with the opportunity to search for games under various criteria using our relational database. This is a command line application that requires a user login with username and password from the database.

2 Design

2.1 Conceptual Model

One of the most important decisions made in the creation of this EER diagram was asserting that any attribute with a "name" should be made an entity. This cleaned up our relationships and will make the database more intuitive to use in the future. The *Collection* entity is weak; when a user is removed from the system, their collections should be removed as well. The *Creator* entity will either take on the role of "publishes" or "develops" in relation to a video game. Any given video game must have at least one publisher and one developer to enter the system, but any given publisher

or developer need not have a video game attached. This is because a 1..N cardinality would erroneously require every publisher to have developed a game (and every developer to have published a game).

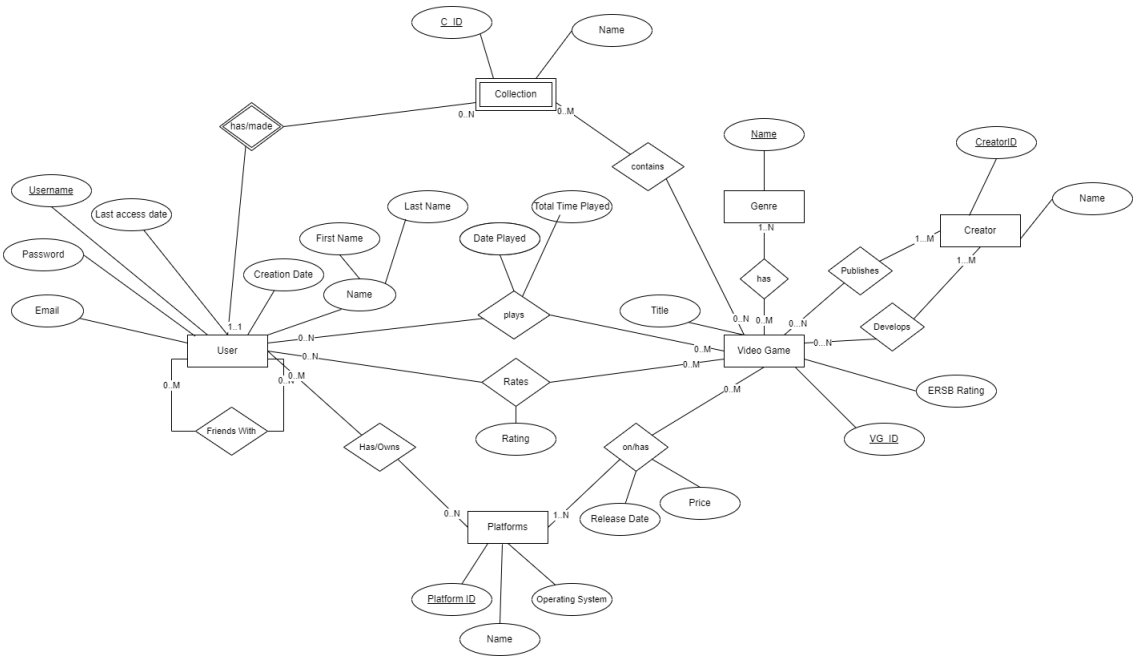

Figure 1: EER Diagram

2.2 Reduction to tables

This section includes the reduction to of our ER diagram to tables. We did this by creating a table for each entity with a column for each attribute. The N:M relationships were reduced to tables as well. Other relationships were represented with foreign keys. Each relation has required values and any foreign keys listed below its reduction. The PLAYS table works as follows: each time a user plays a game, a new row in the PLAYS table is made. The total time played in each row reflects the amount of time the user spent playing that game ONLY during that specific session.

USER(<u>username</u>, password, email, last_access_date, creation_date, first_name, last_name)

//none of the above values can be null

VIDEO_GAME(vg_id, title, ersb_rating)
//ersb_rating cannot be null, there is a set domain, title can not be null

COLLECTION(collection_id, *username*, name)
//name cannot be null

GENRE(name)

PLATFORMS(platform_id, name, os)
//name cannot be null
//username foreign key refers to username in USER, not null

CREATOR(creator_id,name)
//name cannot be null

FRIENDS_WITH(*uid*, *fid*)
//uid foreign key refers to username in USER, not null
//fid foreign key refers to username in USER, not null

PLAYS(date_played, *username*, *vg_id*, total_playtime)
//username foreign key refers to username in USER, not null
//vg_id foreign key refers to vg_id in VIDEO_GAME, not null

RATES(*username*, *vg_id*, rating)
//rating cannot be NULL
//username foreign key refers to username in USER, not null
//vg_id foreign key refers to vg_id in VIDEO_GAME, not null

USER_HAS/OWNS_PLATFORM(*username*, *platform_id*)
//username foreign key refers to username in USER, not null
//platform_id foreign key refers to platform_id in PLATFORMS, not null

VIDEO_GAME_ON/HAS_PLATFORM(*vg_id*, *platform_id*, release_date, price)
//vg_id foreign key refers to vg_id in VIDEO_GAME, not null

//platform_id foreign key refers to platform_id in PLATFORMS, not null

PUBLISHES(*creator_id*, *vg_id*)
//creator_id foreign key refers to creator_id in CREATOR, not null
//vg_id foreign key refers to vg_id in VIDEO_GAME, not null

DEVELOPS(*creator_id*, *vg_id*)
//creator_id foreign key refers to creator_id in CREATOR, not null
//vg_id foreign key refers to vg_id in VIDEO_GAME, not null

HAS_GENRE(*vg_id*, *genre_name*)
//vg_id foreign key refers to vg_id in VIDEO_GAME, not null
//genre_name foreign key refers to name in GENRE, not null

COLLECTION_CONTAINS(*collection_id*, *vg_id*)
//collection_id foreign key refers to collection_id in COLLECTION, not null
//vg_id foreign key refers to vg_id in VIDEO_GAME, not null

2.3 Data Requirements/Constraints

USER: email - Must be unique to each user.

VIDEO_GAME: ersb_rating - Has a set domain of seven values: E, E10+, T, M, A, RP, RP17+, and is required, as there are ratings for Rating Pending, so even a game in progress should have a rating.
title should be unique.

FRIENDS_WITH: Users cannot be friends with themselves, so the uid and fid are both required and unique from each other.

RATES: rating- Has a set domain of 5 values: 1, 2, 3, 4, 5 and is required.

2.4 Sample instance data

This section includes sample entities for every entity type in the EER diagram. There is also a sample table for every relationship.

USER:

Username	Password	Email	Last Access Date	Creation Date	First Name	Last Name
polobronson	Hbhbhb$b$hbhbhg$g$	lr4631@rit.edu	22/Feb/2023	20/Feb/2023	Lily	Ready
fuzzball32	72939For	lek7518@rit.edu	22/Feb/2023	16/April/2021	Lydia	Klecan
PureFyre	insecure33password!	jak9527@rid.edu	22/Feb/2023	4/September/2021	Jacob	Karvelis
oinkoink	RollIntheMud287	brl7498@rit.edu	22/Feb/2023	1/January/2023	Brady	Lax
chamomeleon	Xky78*d5	cid3081@rit.edu	22/Feb/2023	17/November/2019	Camille	Decker

VIDEO_GAMES:

vg_id	Title	ESRB Rating
00000001	Hollow Knight	E10+
00000002	Just Dance 2018	E
00000003	Minecraft: Java Edition	E10+
00000004	Pokemon Scarlet	E
00000005	Overcooked! All You Can Eat	E

COLLECTIONS:

C_id	Username	name
00000001	polobronson	Completed Games
00000382	fuzzball32	Lydia's Games
0000786	PureFyre	Favorite games :)
00000002	oinkoink	Piggy Playlist
00000053	chamomeleon	yikes-playlist

GENRE:

Name
Metroidvania
Action
Survival Crafting
Adventure
Indie Game

PLATFORM:

P_id	Name	OS
00000001	PC	Windows
00000443	Xbox	Xbox system software
00000234	PlayStation 4	Orbis OS
00000002	Nintendo Switch	Horizon
00000094	PlayStation 5	Orbis OS

CREATOR:

C_id	Name
00000001	Team Cherry
00000222	Ubisoft Milan
00001312	Mojang
00000002	Game Freak
00077838	Ghost Town Games

FRIENDS_WITH:

UID	FID
polobronson	XDGamingDX
fuzzball32	oinkoink
PureFyre	Sof
oinkoink	neighneigh
chamomeleon	crispycrispy

PLAYS:

Date Played	Username	vg_id	Total time
13/Aug/2022	fuzzball32	00000002	3045
22/Feb/2022	polobronson	00000001	430
20/Feb/2022	PureFyre	00000003	710
22/Feb/2023	oinkoink	00000004	30000
12/Feb/2023	chamomeleon	00000005	4513

RATES:

Username	vg_id	rating
fuzzball32	00000002	5 stars
polobronson	00000001	5 stars
PureFyre	00000003	5 stars
oinkoink	00000004	4 stars
chamomeleon	00000005	5 stars

USER_HAS/OWNS_PLATFORM:

Username	P_id
polobronson	00000001
fuzzball32	00000443
PureFyre	00000001
oinkoink	00000002
chamomeleon	00000002

VIDEO_GAME_ON/HAS_PLATFORM:

vg_id	P_id	Release Date	Price
00000001	00000001	24/Feb/2017	14.99
00000002	00000443	24/Feb/2017	19.96
00000003	00000001	18/Nov/2011	29.99
00000004	00000002	18/Nov/2022	59.99
00000005	00000002	10/Nov/2020	39.99

PUBLISHES:

C_id	vg_id
00000001	00000001
00000222	00000002
00001312	00000003
00000002	00000004
00077838	00000005

DEVELOPS:

C_id	vg_id
00000001	00000001
00000223	00000002
00001312	00000003
00000002	00000004
00077838	00000005

HAS_GENRE:

vg_id	Genre Name
00000001	Metroidvania
00000002	Dance
00000003	Survival Crafting
00000004	Adventure
00000005	Indie Game

COLLECTION_CONTAINS:

C_id	vg_id
00000001	00000001
00000382	00000002
00000786	00000003
00000002	00000004
00000053	00000005

3 Implementation

Below includes samples of the SQL statements used to create the tables and the SQL insert statements used in the application program to insert new data in the database.

3.1 Examples of SQL to Create Tables

CREATE TABLE video_game(
vg_id INTEGER PRIMARY KEY,
title VARCHAR,
esrb_rating VARCHAR(4)
)
CREATE TABLE genre(
name VARCHAR PRIMARY KEY
)

3.2 Examples of SQL to Populate Data

USER:
INSERT INTO user VALUES(Icarus, fa11in9, waxwings@gmail.com, 2023-01-22 20:22:59.000000, 2022-06-21 02:09:00.000000, Icarus, Daedaluson)
INSERT INTO user VALUES(hungryMan, r3@ch1ng, baddad@gmail.com, 2023-01-22 20:22:59.000000, 2022-06-21 02:09:00.000000, Tantalus, Zeuson)

VIDEO_GAME:
INSERT INTO video_game VALUES(1, Epic Gamer, M)
INSERT INTO video_game VALUES(2, Fake It!, E)

COLLECTION:
INSERT INTO collection VALUES(1, Icarus, Favorite Games)
INSERT INTO collection VALUES(30, hungryMan, Sad Games)

GENRE:
INSERT INTO genre VALUES(Comedy)
INSERT INTO genre VALUES(Horror)

PLATFORM:
INSERT INTO platform VALUES(1, Xbox, Linux)
INSERT INTO platform VALUES(9, Commodore 64, Windows)

CREATOR:
INSERT INTO creator VALUES(1, John Doe)
INSERT INTO creator VALUES(2, Jane Doe)

FRIENDS_WITH:
INSERT INTO friends_with VALUES(Icarus, hungryMan)
INSERT INTO friends_with VALUES(xXGamerXx, Fubbins)

PLAYS:
INSERT INTO plays VALUES(2023-01-28 09:52:08.000000, Icarus, 30, 10)
INSERT INTO plays VALUES(2023-01-15 20:00:22.000000, xXGamerXx, 88,
9)

RATES:
INSERT INTO rates VALUES(Icarus, 30, 3)
INSERT INTO rates VALUES(Gamer2, 2, 5)

USER_HAS/OWNS_PLATFORM:
INSERT INTO user_has/owns_platform VALUES(Icarus, 3)
INSERT INTO user_has/owns_platform VALUES(hungryMan, 6)

VIDEO_GAME_ON/HAS_PLATFORM:
INSERT INTO video_game/has_platform VALUES(60, 3, 2011-04-27 03:04:23.000000,
40)
INSERT INTO video_game/has_platform VALUES(40, 2, 2002-08-20 00:38:19.000000,
10)

PUBLISHES:
INSERT INTO publishes VALUES(45, 76)
INSERT INTO publishes VALUES(1, 10)

DEVELOPS:
INSERT INTO develops VALUES(45, 76)
INSERT INTO develops VALUES(1, 10)

HAS_GENRE:
INSERT INTO has_genre VALUES(5, Comedy)
INSERT INTO has_genre VALUES(41, Horror)

COLLECTION_CONTAINS:
INSERT INTO collection_contains VALUES(89, 6)
INSERT INTO collection_contains VALUES(19, 10)

3.3 Data Loading Strategies

We generated our tables using Mockaroo, and then edited values as needed in Google Drive (EX: ersb_rating was generated as a number between 1 and 7, which we then used Google Sheet's Find And Replace to swap it for one of the actual ESRB Ratings). We then created the base tables and their columns, and used Datagrip to load in our CSV's, which filled out the database's tables with the data we generated from Mockaroo. Below are some examples of the commands that would have been needed to fill out the genre table in the initial loading of the data had we not used the tools we did.

INSERT INTO genre VALUES(Comedy)
INSERT INTO genre VALUES(ComedyRomance)
INSERT INTO genre VALUES(FantasyRomance)
INSERT INTO genre VALUES(ActionCrimeDrama),(Action),(Horror),(Thriller)
INSERT INTO genre VALUES(DramaWar)

4 Data Analysis

4.1 Hypothesis

The number of games a user has rated is the best predictor of how many followers they have.

4.2 Data Preprocessing

We utilized Google Sheets to find any missing values in categories like the total playtime, which is just null if a player hasn't played any games before. As the rest of the column is measured in minutes we substituted this null value with a zero, as we didn't want to discount the users on our platform who haven't played any of their games, as our hypothesis hinges on the effect of user interaction with their games. We didn't need to do any trimming of outliers, as our data was generated randomly with Mockaroo, using normal and Gaussian distributions to do so. With the quantity of data we were generating, there were no significant outliers.

For extracting the data, we utilized complex SQL queries to join our various tables to retrieve a table that was formatted the way we wanted it to be so that it could be quickly analyzed. For example, counting up how many times a user reviewed a game, rather than looking at individual reviews, which is how they're currently stored in the table, and then using that select joined with several others so we could get various measures of each user's interaction with the platform.

4.3 Data Analytics & Visualization

Table 1: Cross-Correlation Matrix

	Followers	Games Rated	Hours Played	Collections
Followers	1.000000	0.448885	0.271449	0.134511
Games Rated	0.448885	1.000000	0.727284	0.124942
Hours Played	0.271449	0.727284	1.000000	-0.056313
Collections	0.134511	0.124942	-0.056313	1.000000

Table 1 shows a matrix of cross-correlation coefficients created in Python using the pandas package. Each cell measures the similarity of two se-

ries/vectors in range [-1, 1]. The closer to 0, the less often its series are seen moving up and down together. It is important to note that a negative value does not mean less correlation; rather, it is a vector pointing in the opposite direction. A perfect correlation only exists between two of the same attribute vector, like Follower count perfectly predicting Follower count, which isn't useful for any real analysis.

From this matrix, we can see that "number of followers" is most highly correlated with "number of games rated." In other words, follower count tends to go up and down with number of games rated more often than with hours played or number of collections. Although, in some way, this proves our hypothesis, the correlation of 0.45 is not particularly high. Therefore, while "games rated" is the best predictor, it isn't a good predictor.

Another interesting point here is the correlation between hours played and games rated. This relationship has the highest correlation with a value of 0.73 and so indicates a much better predictor of user interaction on our platform.

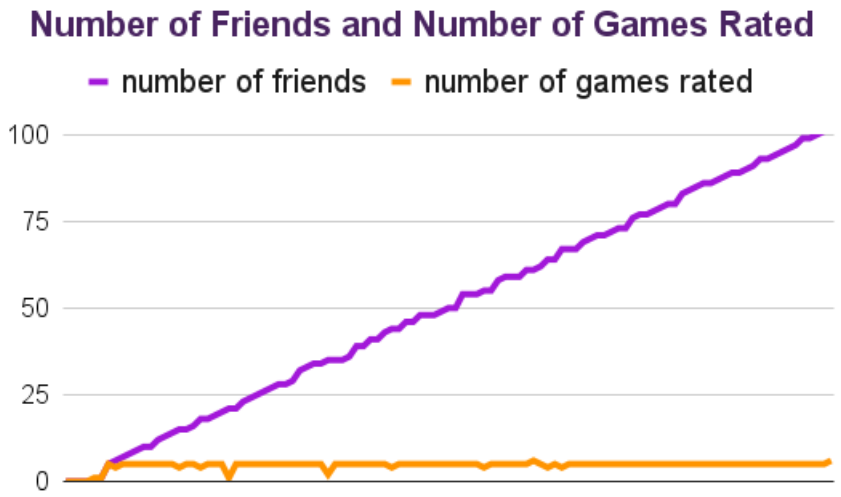

Figure 2: Number of Friends and Number of Games Rated

Each of Figures 2, 3, and 4 were made using Google Sheets. Each one uses the number of followers a user has as a comparison for another attribute. To display the correlation between the two, the first step was to sort the number of followers in ascending order so that it created a trend line. Then,

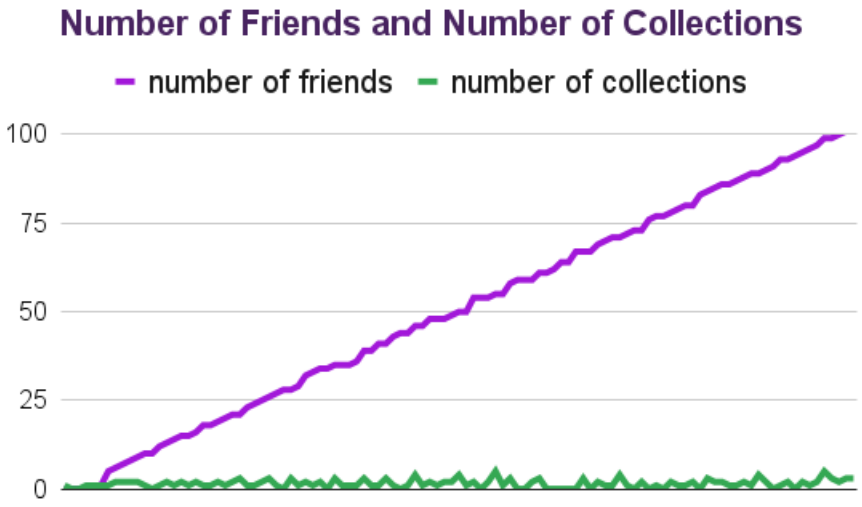

Figure 3: Number of Friends and Number of Collections

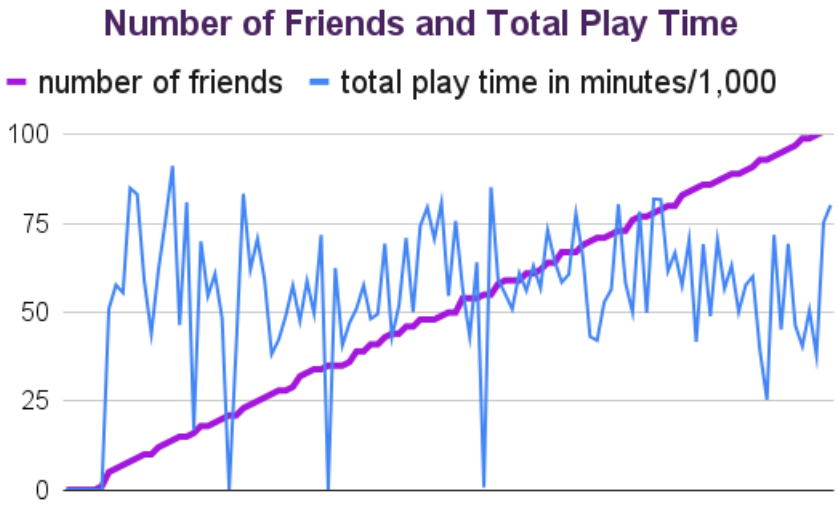

Figure 4: Number of Friends and Total Playtime

it could be seen if the other attributes (number of collections, games rated, or hours played) followed the shape of that line. For the Number of Friends and Number of Games Rated graph, which was the most correlated, the

correlation can be seen at the beginning of the line, and then it diverges pretty dramatically. The graph for the Number of Friends and Number of Collections is seemingly not correlated at all. The graph for the Number of Friends and Total Play Time required some additional data manipulation. All of the values of Total Play Time were divided by 1,000 in order to keep the values in the same range. While the values for this are more varied, there is still no clear correlation.

4.4 Conclusions

Due to the fact that we used randomized data, the results of our analysis don't hold much value. However, our hypothesis was proven correct regarding the possible predictors of how many followers a user has, because out of the three attributes analyzed, the number of games rated had the most correlation with the follower count. Our exploratory analysis seen in the graphs did not produce any correlation (which was to be expected due to our random data generation). As you can see in Figures 2, 3 and 4, there doesn't appear to be any correlation between each attribute and the follower count.

However, we were able to see more clear results through our cross-correlation matrix. The correlation between follower count and the number of games a user has rated was the value with the highest correlation. This would indicate that the number of games rated actually is the best predictor for the number of followers, proving our hypothesis (technically) correct. This information is only slightly useful, as ultimately, a correlation coefficient of 0.45 is not considered highly correlated. Also, in the cross-correlated matrix, there are much greater predictors present, like the hours played and the number of games rated, with a score of 0.73. So while our hypothesis is technically correct, it wouldn't be practically useful as a predictor.

5 Lessons Learned

We frequently bumped up against issues with our SQL not returning what we thought it would, or joins not working the way we intended in the programming and analytic phases. To fix this, team members would write out the SQL in a DataGrip console, and continue to finesse and finagle the queries

until they returned the appropriate information. These were then copied over and formatted to fit the application. This drastically increased the speed we were working at, and we all gained a lot more familiarity with SQL through our repeated trial and error.

By far one of the greatest issues we had in the course of this project was getting the sort by genre functionality when doing any kind of searching within the application. Every other kind of sorting was implemented with relative ease, with any issues being resolved within a day or so. However, with the way we have the data set up and the way we generated genres, there's no real way to sort by the genres in a meaningful way. After trying to reach the professor over email and during office hours, we finally learned the lesson that Professor Johnson explained on one of the first days: If a client asks for something that isn't feasible, just don't show them that feature. We implemented the genre search in the way that made the most sense (though it had no effect on the display) and just didn't include it in the video.

6 Resources

We used Mockaroo to create most of our data. To create the video game titles, we used a spreadsheet of games that one of our group members already had, and we copied over information from there to adjust some of that data.

For data analytics, we used Google Sheets over Excel, as our group was more familiar with it, and it had the same functionalities. We also took advantage of Python and the Pandas package to perform analysis on the extracted data. This is how we got our cross-correlation matrix. We used Canva to create the graphics for the poster.

7 Appendix

7.1 Indexes

Indexes were used in order to speed up the recommendation system so that they wouldn't need to search through the whole table. The indexes were made on the plays table for the attributes vg_id, date_played, and total_playtime.

7.1.1 Index A

The first index was plays_pk and it held unique values for vg_id, username, date_played.

CREATE index plays_vg_id_username_date_played_index ON plays (vg_id, username, date_played);

7.1.2 Index B

The second index was plays_idx_date, which held the date_played in descending order.

CREATE index plays_idx_date ON plays (date_played desc);

7.1.3 Index C

The third index was plays_idx_total_playtime, which had total_playtime in descending order.

CREATE index plays_idx_total_playtime ON plays (total_playtime desc);

7.2 SQL Queries for Analysis

7.2.1 Query A

Retrieves the username of a player, how many people that user is following, how many games that user has rated, the total playtime of all games that

user has, and how many collections that user has.

```
SELECT DISTINCT fol.username, fol.Following, rat.Ratings, time.Playtime,
cols.ColCount
    FROM "user" LEFT OUTER JOIN
      (SELECT "user".username, COUNT(fw.fid) AS Following
        FROM "user" LEFT OUTER JOIN
        "friends_with" AS fw ON "user".username = fw.uid
        GROUP BY "user".username)
    AS fol ON "user".username = fol.username INNER JOIN
      (SELECT "user".username, COUNT("rates".rating) AS Ratings
        FROM "user" LEFT OUTER JOIN
        "rates" ON "user".username = "rates".username
        GROUP BY "user".username)
    AS rat ON "user".username = rat.username INNER JOIN
      (SELECT "user".username, SUM("plays".total_playtime) AS Playtime
        FROM "user" LEFT OUTER JOIN
        "plays" ON "user".username = "plays".username
        GROUP BY "user".username)
    AS time ON "user".username = time.username INNER JOIN
      (SELECT "user".username,
      COUNT("collection".collection_id) AS ColCount
        FROM "user" LEFT OUTER JOIN
        "collection" ON "user".username = "collection".username
        GROUP BY "user".username)
    AS cols ON "user".username = cols.username;
```

7.2.2 Query B

Retrieves the number of collections a user has.

```
SELECT COUNT(name) FROM collection WHERE username LIKE ?
```

7.2.3 Query C

Retrieves the number of followers a given follower has.

```
SELECT COUNT(fid) FROM friends_with WHERE fid like ?
```

7.2.4 Query D

Retrieves the number of followers a given user has.

SELECT COUNT(uid) FROM friends_with WHERE uid like ?

7.2.5 Query E

Retrieves a user's top 10 most played games to be displayed.

```
SELECT title, SUM(total_playtime) AS total playtime
    FROM plays NATURAL JOIN video_game
    WHERE plays.username = ? GROUP BY title
    ORDER BY SUM(plays.total_playtime) DESC LIMIT 10
```

7.2.6 Query F

Retrieves a user's top 10 best rated games to be displayed.

```
SELECT title, rating FROM rates
    NATURAL JOIN video_game WHERE rates.username = ?
    ORDER BY rates.rating DESC LIMIT 10
```

7.2.7 Query G

Retrieves a user's top 10 most played game, to be displayed in order of rating and then by playtime.

```
SELECT title, rating, SUM(total_playtime) AS total playtime
    FROM rates INNER JOIN plays ON rates.username = plays.username
    AND rates.vg_id = plays.vg_id INNER JOIN video_game
    ON plays.vg_id = video_game.vg_id WHERE rates.username = ?
    AND plays.username = ? GROUP BY rating, title
    ORDER BY rates.rating DESC,
    SUM(plays.total_playtime) DESC LIMIT 10
```

7.2.8 Query H

Retrieves the top 20 played games of the last 90 days for recommendation.

SELECT title, SUM(total_playtime) AS total_playtime
FROM plays NATURAL JOIN video_game vg WHERE *plays.date_played* $\geq$?
GROUP BY title ORDER BY SUM(total_playtime) DESC LIMIT 20

7.2.9 Query I

Selects for display the top five games released this month based on total playtime.

SELECT title, SUM(total_playtime) AS total_playtime
FROM plays NATURAL JOIN video_game vg NATURAL JOIN
video_game_on/has_platform AS vgplatt
WHERE EXTRACT(Month FROM vgplatt.release_date) = ?
AND extract(year FROM vgplatt.release_date) = ?
GROUP BY title
ORDER BY SUM(total_playtime) DESC LIMIT 5

7.2.10 Query J

Retrieves the top 20 games among a user's friends/people they're following for recommendation.

SELECT title, SUM(total_playtime) AS total_playtime
FROM plays NATURAL JOIN video_game vg
WHERE plays.username IN
(SELECT fw.fid FROM friends_with AS fw WHERE uid = ?)
GROUP BY title
ORDER BY SUM(total_playtime) DESC LIMIT 20

7.2.11 Query K

Retrieves the most played genre among the games that a user has played, organized by playtime.

SELECT g.genre_name FROM plays p

```
JOIN has_genre g ON p.vg_id = g.vg_id WHERE p.username = ?
GROUP BY g.genre_name
ORDER BY SUM(p.total_playtime) DESC LIMIT 1
```

7.2.12 Query L

Based on the above query, retrieves a list of other users with the same top genre

```
SELECT username FROM ( SELECT t.username, t.genre_name, t.playtime
FROM
    ( SELECT username, genre_name, SUM(total_playtime)
      AS playtime FROM plays INNER JOIN has_genre hg
      ON plays.vg_id = hg.vg_id GROUP BY genre_name, username)
      t INNER JOIN
          (SELECT username, MAX(playtime) AS max_playtime FROM
              (SELECT username, genre_name, SUM(total_playtime)
              AS playtime FROM plays INNER JOIN has_genre hg ON
              plays.vg_id = hg.vg_id GROUP BY genre_name, username)
          x GROUP BY username )
      y ON t.username = y.username AND t.playtime = y.max_playtime
    ORDER BY t.playtime DESC) AS topGenres
    WHERE genre_name = ?
```

7.2.13 Query M

Utilizing the list of similar users retrieved above, gets a user's name, and the title, playtime and genre of the game they have played the most in their top genre. This is then used to build a list of the top 10 most played games in a genre and recommend them to another user.

```
SELECT p.username, p.vg_title, g.genre_name, SUM(p.total_playtime)
    AS playtime FROM plays p NATURAL JOIN has_genre g
    NATURAL JOIN video_game vg
    WHERE p.username = ? AND g.genre_name = ?
    GROUP BY p.username, p.vg_title, g.genre_name
    ORDER BY playtime DESC LIMIT 1;
```

7.2.14 Query N

Selects all usernames and passwords from the users so that the passwords can be hashed.

SELECT username, password FROM "user"

7.2.15 Query O

Updates all user's passwords with a generated hash version of their password.

UPDATE "user" SET password = ? WHERE username = ?