CIS 550 Final Project – No Game No Life

Junyong Zhao, Weicong Dai, Ziyang Luo and Letian Chu

May 8, 2020

1. Introduction

Do you know what is the most valuable thing in the tough time of corona virus? According the data from Amazon, the most needed thing in the pandemic time is the Switch, rather than the toilet paper or the sanitizer [1]. Our website provides the information about the game people play every day to help users have a better understand to the game they play and recommend the potential games they may be interested. The database seems very important for the gamers to search, to discover, to create and to compare.

In order to resolve this problem, we came up with the idea of developing a user interface that uses a game database to provide detailed game information for popular game on the market. For this project, we used a set of data provided by the Kaggle to create a game database and developed a web application that can retrieve the useful game data information. This web application provides four main functionalities. First of all, users can login to the page by creating the account in the database or login with google. Only by login can the user be redirected to Dashboard. Secondly, after login the page, 12 random games are generated on the page. Thirdly, discover page allow user to see different game data based on the user input filter. Last but not least, similar game can be recommended and compared by the user input game. We wish, by providing the similar game to which the user is playing and compare them side by side, users can be better aware of what they want to play next.

2. Data Processing & Table Population

2.1 Data Description

The game data used in this project are downloaded from Kaggle. Kaggle is a famous data science community with lots of datasets. It contains various latest game datasets with information of sales, ratings, and so on. And we are able to acquire appropriate dataset from it for this specific project. Our original data set can be found in the following websites [2]:
Games: `https://www.kaggle.com/jummyegg/rawg-game-dataset`
Game reviews from vandal: `https://www.kaggle.com/floval/12-000-video-game-reviews-from-vandal`
Game sales with image: `https://www.kaggle.com/ashaheedq/video-games-sales-2019`
Games include all the games used in our application. This is the primary table which links other tables together. In addition, the Game table has some necessary information of each game, such as release date, name, and developer. Game reviews from vandal and Game sales with images contain other required information, in which the image URL and introduction URL are most important.

2.2 Data Cleaning and Preprocessing

1. The downloaded data are in several CSV files. Each file is a unique table with several attributes. For each table, a subset of unnecessary attributes is filtered out.
2. For missing data, we dropped some NaN values in the filtered files (e.g. release year) and preserved necessary ones for further computations (e.g. rating).
3. We calculate the average score of each game genre. Then for some of the games that missed user rating and website rating, we filled the average score of the genre that the games belong to.
4. We removed some rows that contained useless information (e.g. a portion of game names with messy code).
5. The genres and platforms of each game are splitted into different rows to allow for a more effective search.

2.3 Summary Statistics

The tables after preprocessing are showed in the following:

Table Name	Size	Number of Rows	Attributes
Game.csv	16.8MB	310,514	4
Game_genre.csv	7.3MB	449,043	2
Game_plat.csv	5.2MB	429,601	2
Game_class.csv	4.2MB	310,518	2
Website.csv	2.1MB	13,864	3
Platforms.csv	4KB	50	1
Genre.csv	4KB	19	1
Classification	4KB	6	1

Table 1: All Important Datasets Used

2.4 Populating Tables and Connecting to AWS Server

Using our sqlDDL.sql under /data directory, we populated our tables based on the cleaned data. To meet the foreign key constraint, the table should be populated based on the following order: 1. Game; 2. Website, Rating, Platforms, Genre, Classification; 3. Game_genre; Game_plat; Game_class; The code related could be found on the Appendix A.

2.5 Relational Schema and ER diagram

2.5.1 Relational Schema:

```
Game: (id, name, release_date, developer)
PRIMARY KEY (id)

Platform: (name)
PRIMARY KEY (name)
```

```
Genre: (name)
PRIMARY KEY (name)

Classification: (name, description)
PRIMARY KEY (name)

Rating: (id, user_rating, website_rating)
PRIMARY KEY (id)
CONSTRAINT FOREIGN KEY id REFERENCES Game(id)

Game_genre: (game_id, genre_name)
PRIMARY KEY (id, genre_name),
CONSTRAINT FOREIGN KEY game_id REFERENCES Game(id),
CONSTRAINT FOREIGN KEY genre_name REFERENCES Genre(name)

Game_plat: (game_id, plat_name)
CONSTRAINT FOREIGN KEY game_id REFERENCES Game(id),
CONSTRAINT FOREIGN KEY plat_name REFERENCES Platform(name)

Game_class: (game_id, class_name)
CONSTRAINT FOREIGN KEY game_id REFERENCES Game(id),
CONSTRAINT FOREIGN KEY class_name REFERENCES Classification(name)

Website: (id, vg_url, photo_url)
PRIMARY KEY (id)
CONSTRAINT FOREIGN KEY id REFERENCES Game(id)
```

2.5.2 ER Diagram

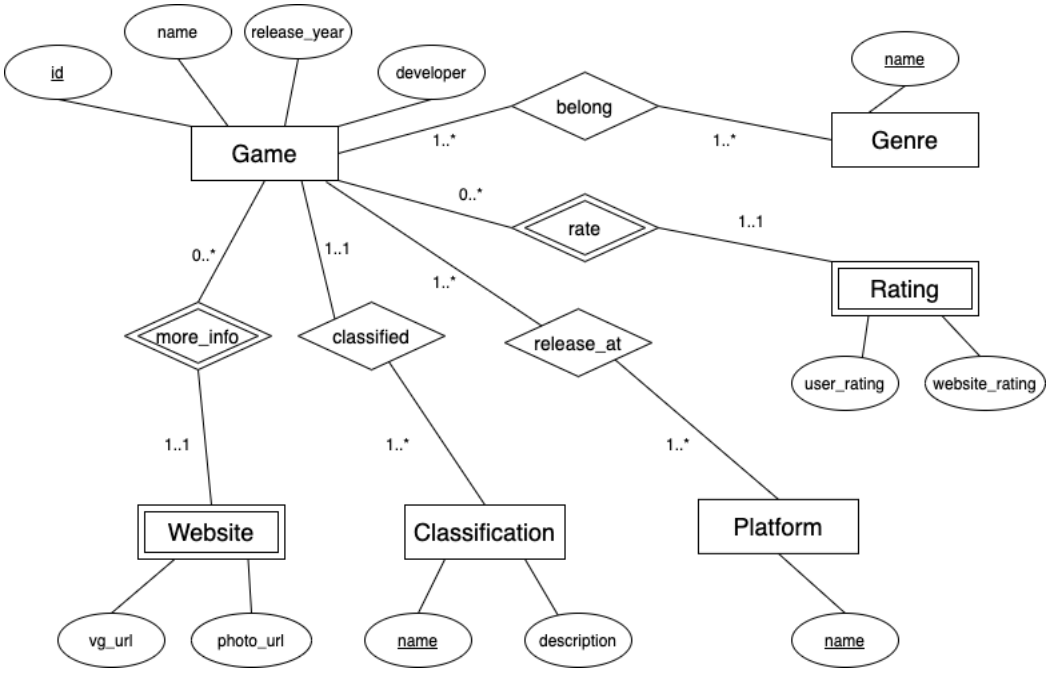

Figure 1: ER Diagram

2.6 Normal Form

Our relations are all in BCNF, as, in each table, all functional dependencies are implied by the keys:

Game: id → name, release year, developer
Game_genre: game_id → genre_name; **Game_plat:** game_id → plat_name
Game_class: game_id → class_name; **Website:** game_id → vg_url, photo_url
Rating: game_id → user_rating, website_rating;
Platforms: platforms → platforms
Genre: name → name; Classification: classification → classification

3. System Architecture

• Login Page

The login page provides two ways for the user to login. User can register in the database and store the username and password. User can also login with google with the google API. Only when you log in with existed account in our database can you redirected to the Dashboard Page. After redirecting, the user can also choose to logout and redirect to the login page.

• Dashboard

The dashboard is the page that appears after login. When user upload the page, it randomly generates 12 games with their image, name, and link for their website. User can click the game image or the name to enter the homepage of the game. Also, there is a search bar. User can search for the game you want by their name and get the image of the game. User also can click the image of the game to enter into the homepage and get more information user want about user's favorite game.

• Discover page

In this page, every user will see many games ordered by their rating based on the game filter where exists multiple attributes. The application provides the potential games which the user may favor.

• Recommendation page

In this page, users can enter the name of their favorite games and get recommendation. The recommendation is based on the genres, ratings and developers of the game. There are two sections of the recommendation. The first section of recommendation is based on the genre and rating of the user input game. The page will recommend some games with similar genres and rating with the user entered game. In terms of the second section, the page will list the games released by the developers of input game. User is likely to play those games since the style of them usually resemble each other.

• Comparison Page

In this page, users can enter two games that they would dive deeper to. Since many games could be similar in terms of genre and platform and this comparison page will list all the attribute of the game side by side to help user to decide. Third-party APIs of IGDB game database [3] is also used in this page to grab storylines that describe the input games.

4. Queries

In total, we implemented 12 queries in this project to achieve the above functionalities. To better illustrate these queries, four of them are selected and discussed below.

- Query 1 Randomly choose 12 games which has website URL and image.
- Query 2 Search the game by its name and return game's name, photo of the game and the website URL of the game.
- Query 3 Filter the game by specific release_year, genre, classification and platform, then return only top 10 satisfied game name with their user_rating, website_rating ordered by its user_rating, website_rating.
- Query 4 Recommend games based on genre, platform and rating. The result would have equal/more genres (platforms) of the input, and rating distance variant no more than ±1.
- Query 5 Recommend other good games from the same developer

The code of the queries could be found in the Appendix B.

5. Performance Evaluation

We performed query optimization on the most two complicated queries on our application by choosing a better join order and putting index on the tables.

The first one is the recommending query, which will suggest the games that the user might like based on the user's input. The algorithm is to choose the games in our database with the same (or more) genre as well as the same (or more) platforms. The potential recommendation is then filtered by the rating distance from the input, and ordered by user rating. This query will give games that are close to the user input in terms of genre and platform, as well as guarantee that the recommendation is commonly recognized by others as close to the input. The query will involve joining and aggregating the Game, Game_class, Game_plat, and the Rating table. These tables represented the many-many relationship that a game could have many genres and platforms. Genres and Platforms could also contain more than one game. Before optimization, we used temporary results to store the genre number of all the games and then pick the game that has an equal or larger number. This intermediate query will involve almost 309,766 Games and 400,000 of genre and platform records, which is obviously not efficient. And it will take roughly 2,000*ms* to execute. To optimize this query, we change the join order to firstly select all the games that contain the same genre (platform) with the input and then count the number of the genre (platform). This will reduce the number of the games to aggregate or join while also keeping the same result (only the targeted result will be kept since they should have equal or more genre (platform) than the input). After this join order optimization, we manage to reduce the execution time down to around 1,000*ms*.

The second query is also used in the recommendation page, where we will recommend to the user other popular games developed by the same developer. This query is particularly slow and it takes initially 5,700*ms* to give the result. Since it will scan the entire Game table, which contains 309,766 rows and take advantage of some helper functions in SQL to parse the developer string and compare. So, we first joined the table with the Website table, since the

Game table contains a huge amount of games but not all of them are recognized as popular enough to have a specific website URL. And the company that developed that game is very unlikely to have the funding for another one. Moreover, we also created an index based on developers and that helped to improve the performance as well and we managed to drop the execution time to 1,600*ms*.

6. Technical Challenges

The first challenge that we faced was cleaning the datasets. A significant portion of the data has meaningless numbers, which makes the function not work properly. For example, there is some wrong release year in the Gama dataset. We not only cleaned the relevant data in the Game table, but we also did the same processing through the join operation in the table which relates to the Game table.

The second challenge that we faced was that how to interface with different functions and combine multiple parameters in one function. Passing the result from different functions and make use of POST (GET) to pass those parameters to one function. As a result, we can fulfill the interface with different functions to achieve some feature.

The third challenge is that how to recommend games more in line with user experience. Generally, the game website only recommends game that has the same genre or same platform of input game. From the user's perspective, that is not enough. So, we come up with a plan which can give a more specific recommendation. For our recommendation function, not only we recommend games with the same genre or input of input game, but also the number of genres of those games is greater than the number of genres of the input game. Furthermore, the difference in rating between recommended games with the input game is within ±1. The recommendation plan more in line with user experience. It encourages us to make a complex query to achieve this function.

The fourth challenge is tuning the back-end of the website. Our application is very robust, and it can deal with different situations when a user enters something. if the user enters the game which does not in the database, we will warn that game not exists. Besides, in our comparison page, if the game user enters does not have an image in our database, the back-end will automatically match the results with a picture to ensure that all results can be returned instead of nothing returning. The meticulous logic of the back-end improves the overall robustness of the application.

The fifth challenge is the user administration system. In order to switch between customized login and google login, we set some variables and two classes in the application's back-end to decide where the user's information redirects to. Besides, to record the user's information, we create a table in our dataset to manage the information of users.

The last challenge is that how to gain information from third-party source. When fetching data from IGDB API [3], an issue called Cross-Origin Request Blocked is encountered. That is

because the user browser will decline the calls to any other site other than our own domain. Therefore, to access data from a third party, we have to apply Cross-origin resource sharing (CORS) mechanism. This mechanism allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

7. Extra Credit

- Our project implemented user login/register function; user info will be stored on the back-end database.
- Our project allows user to login with social media account, like Google account.
- Our project used third-party APIs of the IGDB database to enrich the comparison application.
- Our project hosted on AWS cloud server.

8. References

[1] Kain, E. (2020). The Nintendo Switch is Sold Out Everywhere Thanks to The Coronavirus. Retrieved 8 May 2020, from https://www.forbes.com/sites/erikkain/2020/03/27/the-nintendo-switch-is-sold-out-everywhere-thanks-to-the-coronavirus/#2bcdffb25350/

[2] Video Game Charts, Game Sales, Top Sellers, Game Data - VGChartz. (2020). Retrieved 8 May 2020, from https://www.vgchartz.com/

[3] IGDB.com - Credits, Top Critics, Reviews, Videos and Screenshots. (2020). Retrieved 8 May 2020, from https://www.igdb.com/discover/

Appendix A: SQL DDLs

DDL1: Game

```
CREATE TABLE Game (
    id INT(10),
    name VARCHAR(256) NOT NULL,
    release_year INT(4) NOT NULL,
    developer VARCHAR(256) NOT NULL,
    PRIMARY KEY (id)
)
```

DDL2: Genre

```
CREATE TABLE Genre (
    name VARCHAR(256),
    PRIMARY KEY (name)
)
```

DDL3: Game_genre

```
CREATE TABLE Game_genre (
    game_id INT(10),
    genre_name VARCHAR(256),
    PRIMARY KEY (id, genre_name),
    CONSTRAINT FOREIGN KEY game_id REFERENCES Game(id),
    CONSTRAINT FOREIGN KEY genre_name REFERENCES Genre(name)
)
```

DDL4: Rating

```
CREATE TABLE Rating (
    id INT(10),
    user_rating DECIMAL(3, 1),
    website_rating DECIMAL(3, 1),
    PRIMARY KEY (id)
)
```

DDL5: Platform

```
CREATE TABLE Platform (
    name VARCHAR(256),
    PRIMARY KEY (name)
)
```

DDL6: Game_plat

```
CREATE TABLE Game_plat (
    game_id INT(10),
    plat_name VARCHAR(256),
```

```
    PRIMARY KEY (id, plat_name),
    CONSTRAINT FOREIGN KEY game_id REFERENCES Game(id),
    CONSTRAINT FOREIGN KEY plat_name REFERENCES Platform(name)
)
```

DDL7: Classification

```
CREATE TABLE Classification (
    name VARCHAR(256),
    description VARCHAR(256),
    PRIMARY KEY (name)
)
```

DDL8: Game_class

```
CREATE TABLE Game_class (
    game_id INT(10),
    class_name VARCHAR(256),
    PRIMARY KEY (id, class_name),
    CONSTRAINT FOREIGN KEY game_id REFERENCES Game(id),
    CONSTRAINT FOREIGN KEY class_name REFERENCES Classification(name)
)
```

DDL9: Website

```
CREATE TABLE Website (
    id INT(10),
    vg_url VARCHAR(256),
    photo_url VARCHAR(256)
    PRIMARY KEY (id)
)
```

DDL10: user_info[1]

```
CREATE TABLE user_info (
    username VARCHAR(256),
    password VARCHAR(256),
    PRIMARY KEY (username) NOT NULL
)
```

[1] This table dose not have a connection with the game relations, therefore not discussed in previous sections

Appendix B: Important SQL Queries

Query 1: Randomly Choose Games

```
SELECT Website.id FROM Game, Website WHERE Game.id = Website.id ORDER BY
rand() Limit + numsGame
select DISTINCT Game.name AS name, Website.vg_url AS vg_url,
Website.photo_url AS photo_url From Game, Website Where '${gvid}' =
Website.id AND Game.id ='${gvid}'
```

Query 2 Search the game by name

```
select Game.name AS name2, Website.vg_url AS vg2_url, Website.photo_url AS
photo2_url From Game, Website Where Game.id = Website.id AND Game.name
LIKE'${gname}'
```

Query 3 Filter the game by specific release year, genre, classification and platform

```
select name from Game where lower(name) = lower('${inputTitle}');
select release_year from Game where lower(name) = lower('${inputTitle}');
select user_rating, website_rating from Rating where id = (select id from
Game where lower(name) = lower('${inputTitle}'));
select plat_name from Game_plat where game_id = (select id from Game where
lower(name) = lower('${inputTitle}'));
select class_name from Game_class where game_id = (select id from Game
where lower(name) = lower('${inputTitle}'));
select genre_name from Game_genre where game_id = (select id from Game
where lower(name) = lower('${inputTitle}'));
select vg_url, photo_url from Website where id = (select id from Game where
lower(name) = lower('${inputTitle}'))
```

Query 4 Game recommendation based on similar genres, platforms and rating distance

```
(select * from (
(select t6.id, name, release_year, user_rating, website_rating, vg_url,
photo_url from (
select Game.id, name, release_year from Game, (
select t4.id from (
select id, count(genre_name) as gen_count from (
select t2.id, t2.genre_name from
(select genre_name from Game_genre
where game_id = (select id from Game
where lower(name) = lower("${inputTitle}"))) t1,
(select * from Game join Game_genre on id = game_id) t2
where t1.genre_name = t2.genre_name and t2.name != lower("${inputTitle}")
) t3 group by id
) t4 where t4.gen_count >=
```

```
(select count(genre_name) as gen_count from (
select genre_name from Game_genre
where game_id = (select id from Game
where lower(name) = lower("${inputTitle}"))
) t1)
) t5 where Game.id = t5.id
) t6 join Rating on t6.id = Rating.id
join Website on t6.id = Website.id
order by user_rating DESC, website_rating DESC, release_year DESC
LIMIT 10)) t7 where sqrt(power((user_rating - (select user_rating from
Rating
where id = (select id from Game where lower(name) =
lower('${inputTitle}'))))), 2)) <= 1)
union
(select * from (
(select t6.id, name, release_year, user_rating, website_rating, vg_url,
photo_url from (
select Game.id, name, release_year from Game, (
select t4.id from (
select id, count(plat_name) as plat_count from (
select t2.id, t2.plat_name from
(select plat_name from Game_plat
where game_id = (select id from Game where lower(name) =
lower('${inputTitle}'))) t1,
(select * from Game join Game_plat on id = game_id) t2
where t1.plat_name = t2.plat_name and t2.name != lower('${inputTitle}')) t3
group by id) t4
where t4.plat_count >=
(select count(plat_name) as plat_count from (
select plat_name from Game_plat
where game_id = (select id from Game
where lower(name) = lower('${inputTitle}'))
) t1)) t5 where Game.id = t5.id) t6
join Rating on t6.id = Rating.id
join Website on t6.id = Website.id
order by user_rating desc, website_rating desc, release_year desc
limit 10)) t7 where sqrt(power((user_rating - (select user_rating from
Rating
where id = (select id from Game where lower(name) =
lower('${inputTitle}'))))), 2)) <= 1)
order by user_rating DESC, website_rating DESC, release_year DESC;
```

Query 5 Game recommendation based on developers

```
select t2.id, t2.name, t2.release_year, vg_url, photo_url from Website, (
```

```
select Game.id, Game.name, Game.release_year from Game, (
select * from Game where lower(Game.name) = lower('${inputTitle}')) t1
where find_in_set(substring_index(t1.developer, ',', 1), Game.developer)
or find_in_set(substring_index(t1.developer, ',', -1), Game.developer)) t2
where t2.name != lower('${inputTitle}') and t2.id = Website.id
order by release_year desc limit 20;
```