Web application built with NodeJS based on a Game database normalized and turned into MYSQL by a python script

Data base: https://www.kaggle.com/datasets/fronkongames/steam-games-dataset

I'm using just one cvs file, this file contains the hole Steam digital store videogames catalog, this data has not been normalized.

The original file contains 74690 rows and 39 columns.

The Columns will be the following

1. appID = unique identifier for each app (string).
2. name = Game name (string).
3. releaseDate = Release date (string).
4. estimatedOwners = # Estimated owners (string, e.g.: "0 - 20000").
5. peakCCU = Number of concurrent users, yesterday (int).
6. required_age = Age required to play, 0 if it is for all audiences (int).
7. price = Price in USD, 0.0 if its free (float).
8. dlcCount = Number of DLCs, 0 if you have none (int).
9. About the game = Detailed description of the game (string).
10. languages = Comma-separated enumeration of supporting languages.
11. fullAudioLanguages = Comma-separated enumeration of languages with audio support.
12. reviews = game reviews
13. headerImage = Header image URL in the store (string).
14. website = Game website (string).
15. supportWeb = Game support URL (string).
16. supportEmail = Game support email (string).
17. supportWindows = Does it support Windows? (bool).
18. supportMac = Does it support Mac? (bool).
19. supportLinux = Does it support Linux? (bool).
20. metacriticScore = Metacritic score, 0 if it has none (int).
21. metacriticURL = Metacritic review URL (string).
22. userScore = Users score, 0 if it has none (int).
23. positive = Positive votes (int).
24. negative = Negative votes (int).
25. scoreRank = Score rank of the game based on user reviews (string).
26. achievements = Number of achievements, 0 if it has none (int).
27. recommens = User recommendations, 0 if it has none (int).
28. notes = Extra information about the game content (string).
29. averagePlaytime = Average playtime since March 2009, in minutes (int).
30. averageplaytime2W = Average playtime in the last two weeks, in minutes (int).
31. medianPlaytime = Median playtime since March 2009, in minutes (int).

32. medianPlaytime2W = # Median playtime in the last two weeks, in minutes (int).
33. developers = Developer name (string).
34. publishers = Publisher name (string).
35. categories = Game categories (string)
36. genres = Gender name (string).
37. screenshots = Game screenshot URL (string).
38. movies = Game movie URL (string).
39. tags = Tag key (string, int).

Analysis of the data

Quality: The data appears to be well-structured, with clear column headers and consistent data types. Most columns have relevant and meaningful data, such as game titles, release dates, genre information, supported languages, and review scores. However, some columns contain redundant or unnecessary information.

Level of Detail: The data provides a substantial level of detail about each game, including key attributes like release date, estimated owners, peak concurrent players (CCU), required age, price, supported languages, reviews, achievements, playtime, developers, publishers, categories, genres, and tags. This level of detail allows for comprehensive analysis and comparisons between games.

Documentation: The data lacks documentation inside the database, make it challenging to understand the purpose of certain attributes. For example, it is unclear what "DLC count" represents without additional context. Lucky for us, in the website when I get the data. There is good documentation about what every in-database means. So maybe make it more clear inside the database will be an improvement.

Interrelation: The data does not explicitly provide information on how different tables or entities are related. Without an Entity-Relationship diagram or foreign key references, it is difficult to determine the relationships between entities such as games, developers, and publishers. This is the reason for me to choose this database.

Use: The dataset can be used for various analyses, such as identifying popular games based on owners and peak CCU, examining pricing strategies and their impact on user reviews, exploring language preferences, and understanding the distribution of game genres and categories.

Discoverability: The dataset contains useful information for discovering and exploring trends in the gaming industry. Researchers, analysts, and game developers can utilize this dataset to gain insights into the performance of different games and understand user preferences.

Modifying the data for the project proposes.

The database is really complex with a lot of rows (74690) and columns (39). Since normalization itself does not directly handle the presence of null data in a database and since the proposes of this project is to create a complex, yet not oversize database, I will use a random piece of the database.

Also, I will eliminate directly some columns since normalization does not handle the presence of null data in a database column starting with score_rank because 99% of that column was blank, also I will eliminate User Score because it has 99.94% as 0 scores. The columns median playtime two weeks and average playtime two weeks are also eliminated since they have 97.42 % of 0 values and therefore, all these 4 rows are irrelevant.

So, at the end, the size of the database I will use for the project will be 4000 rows and 35 columns, making the database more meanable without taking out the complexity.

Why is an interesting database and questions to ask a database app?

This database is interesting because it provides a good compressive information about various video games in the most use digital store in the world. This data can be use in the gaming industry for understand the gamer preferences and the success and fail of videogames.

Here are some questions I could be answer with a database application.

- Which games have the most players, and how does this correlate with their peak concurrent users and review scores?
- What is the average price of games in different genres, and how does price affect user recommendations and ratings?
- Which developers and publishers have the most games in the dataset, and what are the average review scores and playtime for their games?
- What are the most popular supported languages for games, and how do language preferences differ based on genre and region?
- How does the average playtime for games vary across different genres and age ratings?
- Which genres have the most positive or negative reviews, and are there any specific themes or mechanics associated with these trends?
- What is the distribution of achievements across games, and how do achievements impact player engagement and playtime?
- Do games with full audio support in multiple languages receive more positive reviews than those with limited audio options?
- How does the average playtime vary between single-player and multiplayer games?
- Which games have the highest Metacritic scores, and what attributes do they share that contribute to their success?

Normalizing the data

1NF

1NF consist basically in have atomic values in every column, based on that I change and organize the following columns:

Estimate owners: I create an entity for estimate owners with its own id, and I separate the min and max values.

estimate Owners
estimate_owners_id
min_estimated_owners
max_estimated_owners

Audio	Subtitles
Audio_ID	Audio_ID
Audio	Subtitles

Supported audio and full audio languages: I create two entities with its own ID's, both entities only contain the ID and the respective languages.

Developers and publishers: create two entities with its own ID's, both entities only contain the ID and the respective developers and publishers.

Developers	Publishers
Devs_id	publishers_id
Devs	publishers

Categories	Genres	Tags
categories_id	genres_id	tags_id
Categories	genres	tags

Categories, Genres, Tags: create three entities with its own ID's, all the entities only contain the ID and the respective values.

Screeshots and Trailers (Movies): I create two entities with its own ID's, both entities only contain the ID and the respective URLS.

Trailer	Screenshots
Trailer_id	screenshots_id
trailer_URL	screenshots_URL

2NF

2NF basically consist on eliminate all the partial dependencies on a database.

So basically, I create some entities and make some incomplete 3NF (separation of topics) trying not to have any partial dependency in the process. The first row of any table is the title of the table and all the cells that have different font color, are foreign keys.

Games
App_Id
Title
Realise_date
price
total_DLC_packs
Age_rating
sinopsis
support_web
support_email
Language_ID
system_id
users_data_id
Cotegorization_id
Devs_Publushers_id
media_id
review_id

Language	Audio	Subtitles
Language_ID	Audio_ID	subtitles_ID
Audio_ID	Audio	Subtitles
subtitles_ID	**Language_ID**	**Language_ID**

Categorization	Categories	Genres	Tags
Cotegorization_id	categories_id	genres_id	tags_id
categories_id	Categories	genres	tags
genders_id	**Cotegorization_id**	**Cotegorization_id**	**Cotegorization_id**
tags_id			
notes			

Studios	Developers	Publishers
studios_id	Devs_id	publishers_id
publishers_id	Devs	publishers
Devs_id	studios_id	studios_id

media	Trailer	Screenshots
media_id	Trailer_id	screenshots_id
cover_img	trailer_URL	screenshots_URL
game_website	**media_id**	**media_id**
Trailer_id		
screenshots_id		

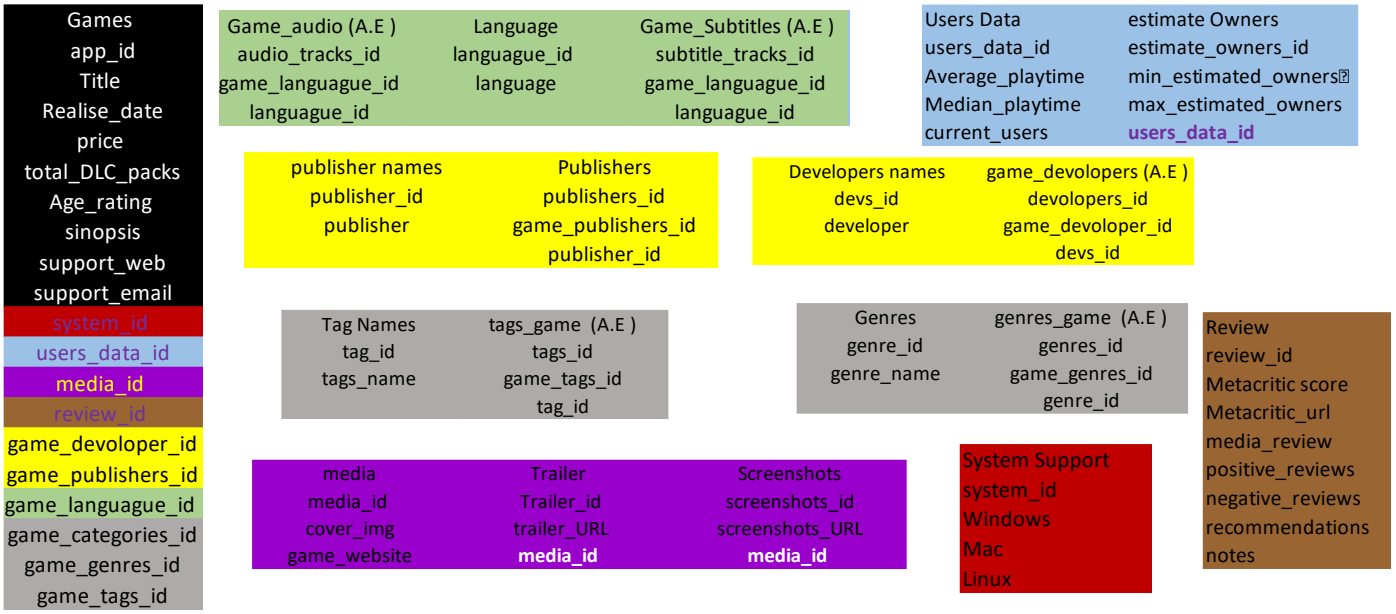
Users Data	estimate Owners
users_data_id	estimate_owners_id
Average_playtime	min_estimated_owners
Median_playtime	max_estimated_owners
current_users	**users_data_id**
estimate_owners_id	

System Support
system_id
Windows
Mac
Linux

Review
review_id
Metacritic score
Metacritic_url
media_review
positive_reviews
negative_reviews
recommendations

3NF

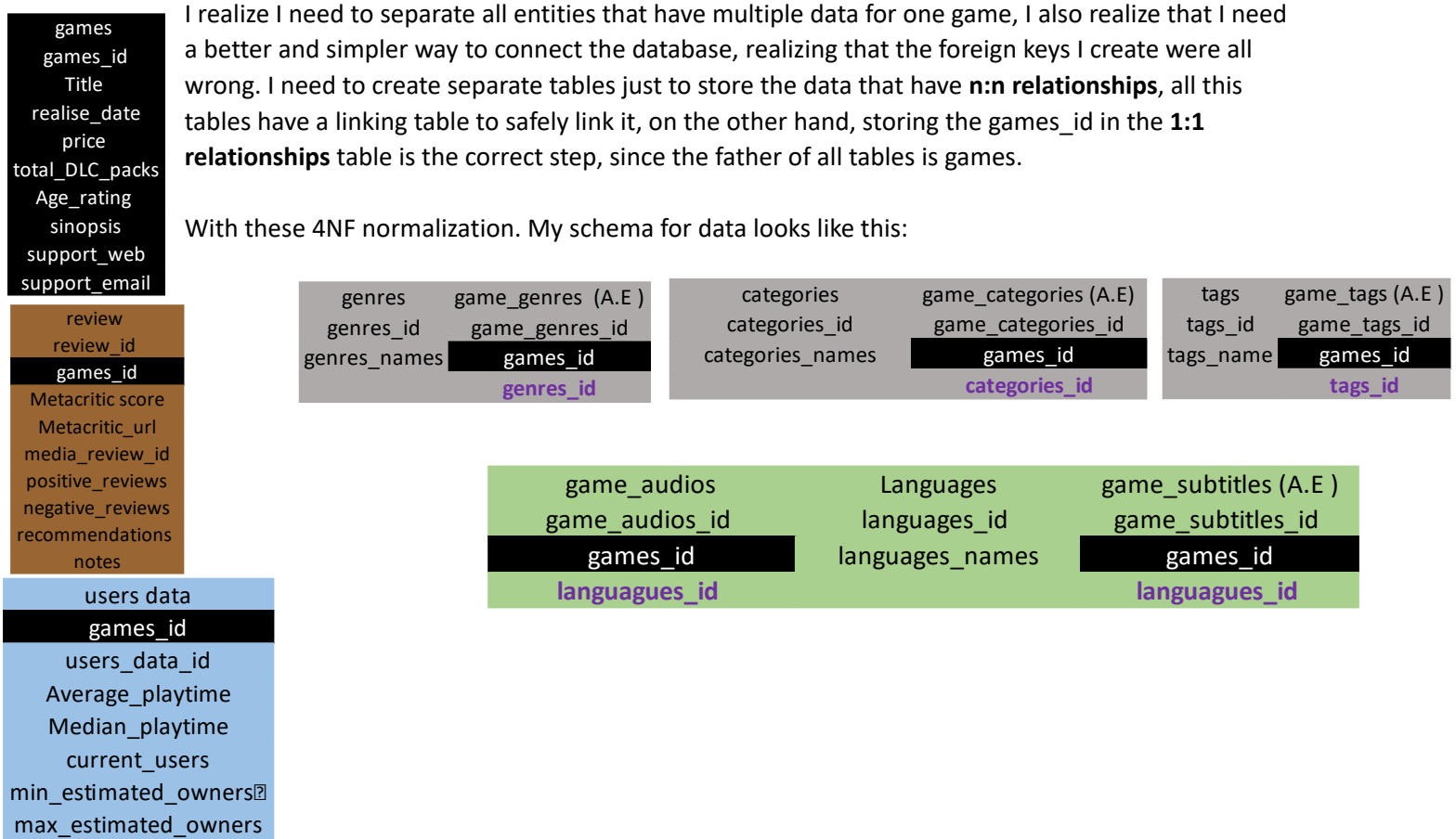
The 3NF is to check the transitive dependencies, that leave to changes somethings in the schema

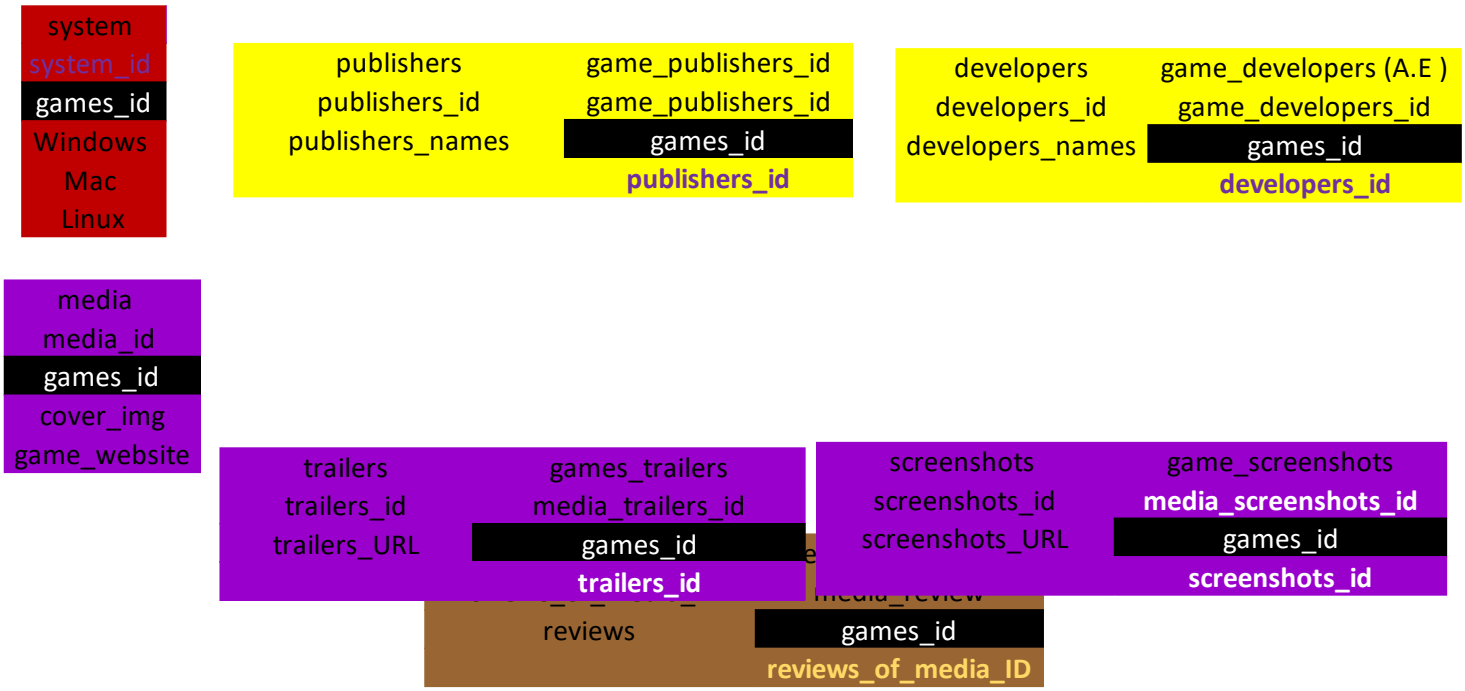


4NF

I realize I need to separate all entities that have multiple data for one game, I also realize that I need a better and simpler way to connect the database, realizing that the foreign keys I create were all wrong. I need to create separate tables just to store the data that have **n:n relationships**, all this tables have a linking table to safely link it, on the other hand, storing the games_id in the **1:1 relationships** table is the correct step, since the father of all tables is games.

With these 4NF normalization. My schema for data looks like this:






```
elif support_web is not None and len(support_web) > MAX_URL_LENGTH:
```

RELATIONSHIPS

- **Games (1) ---- (1) Review**

  Each game has only one review_id, and each attribute in the 'reviews' table is associated with one game.

- **Games (1) ---- (1) Media**

  Each game has only one media_id, and each attribute in the 'media' table is associated with one game.

- **Games (1) ---- (1) System Support**

  Each game has only one system_support_id, and each attribute in the 'system_support' table is associated with one game.

- **Games (1) ---- (1) Users Data**

  Each game has only one users_data_id, and each attribute in the 'users_data' table is associated with one game.

- **Games (N) ---- (N) Tags**

  Each game can have multiple tags, and each tag can be associated with multiple games through the 'game_tags' table.

- **Games (N) ---- (N) Categories**

  Each game can have multiple categories, and each category can be associated with multiple games through the 'game_categories' table.

- **Games (N) ---- (N) Genres**

  Each game can have multiple genres, and each genre can be associated with multiple games through the 'game_genres' table.

- **Games (N) ---- (N) Publishers**

  Each game can have multiple publishers, and each publisher can be associated with multiple games through the 'game_publishers' table.

- **Games (N) ---- (N) Developers**

  Each game can have multiple developers, and each developer can be associated with multiple games through the 'game_developers' table.

- **Games (N) ---- (N) Trailers**

  Each game can have multiple trailers, and each trailer can be associated with multiple games through the 'game_trailers' table.

- **Games (N) ---- (N) Screenshots**

  Each game can have multiple screenshots, and each screenshot can be associated with multiple games through the 'game_screenshots' table.

- **Games (N) ---- (N) Audios**

  Each game can have multiple audio languages, and each audio language is stored in languages tables and is associated with multiple games through the 'game_audios' table.

- **Games (N) ---- (N) Subtitles**

  Each game can have multiple subtitle languages, and each subtitle language is stored in languages tables and is associated with multiple games through the 'game_subtitles' table.

- **Games (N) ---- (N) Reviews of Media**

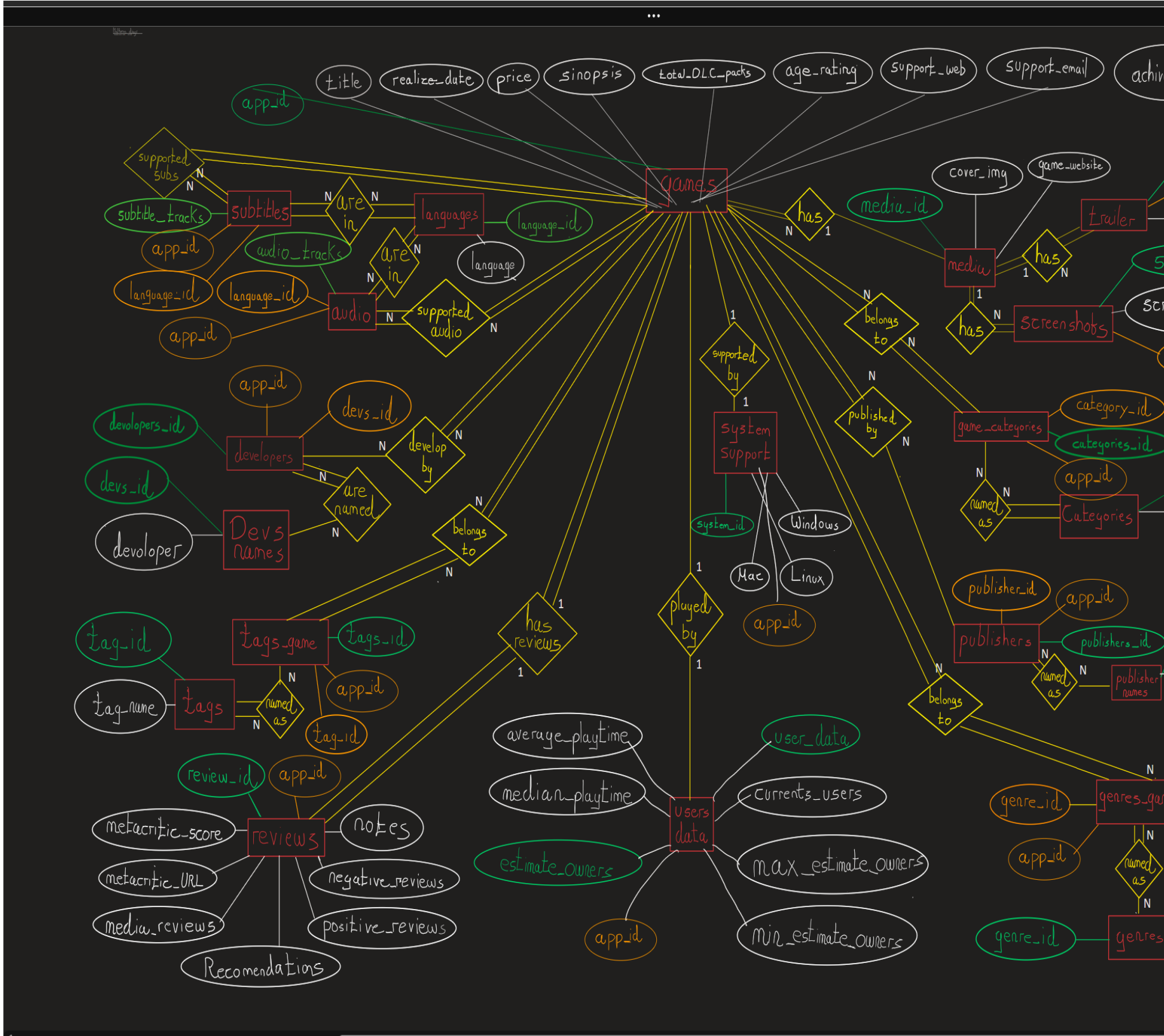
  Each game can have multiple media reviews, (the reviews the database have of magazines,websites,etc) and each media review can be associated with multiple games through the 'game_reviews_of_media' table.

ER diagram

The ER Diagram consists of several entities represented by red rectangles, each containing white circles representing attributes. The primary keys, denoted by green circles, uniquely identify each record in their respective entities. Foreign keys, indicated by orange circles, establish relationships between different entities. The relationships, depicted in yellow, signify the connections between the entities.

The notation of "N" and "1" next to the relationships indicates the type of relationship between two entities. "N" represents a one-to-many or many-to-many relationship, implying that multiple records in one entity can be associated with multiple records in the other. On the other hand, "1" represents a one-to-one or one-to-many relationship, signifying that one record in an entity is associated with either one or multiple records in another entity.

To get a visual representation of the schema, please refer to the attached PNG image and the OneNote document. The image provides a clear overview of how the entities are related through their respective primary and foreign keys, forming an organized and efficient relational database structure.


games
title
realize_date
price
sinopsis
total_DLC_packs
age_rating
support_web
support_email
app_id
supported subs
subtitles
subtitle_tracks
app_id
language_id
app_id
are in
languages
language_id
language
audio_tracks
are in
audio
language_id
app_id
supported audio
has
media_id
cover_img
game_website
trailer
media
has
screenshots
belongs to
supported by
system support
system_id
Windows
Mac
Linux
app_id
published by
game_categories
category_id
categories_id
app_id
named as
Categories
developers_id
devs_id
developers
app_id
devs_id
develop by
are named
Devs names
devoloper
belongs to
tag_id
tag_name
tags
named as
tags_game
tags_id
app_id
tag_id
has reviews
played by
publisher_id
app_id
publishers
publishers_id
named as
publisher names
belongs to
review_id
app_id
metacritic_score
metacritic_URL
media_reviews
reviews
notes
negative_reviews
positive_reviews
Recomendations
average_playtime
median_playtime
estimate_owners
app_id
users data
user_data
currents_users
max_estimate_owners
min_estimate_owners
genre_id
app_id
genre_id
named as
genres

Creating the data in MYSQL

Table Relationships:

The tables that start with 'game_' are used to establish a many-to-many relationship between the 'games' table and other entities (e.g., publishers, developers, tags, genres, etc.). This allows each game to have multiple publishers, developers, tags, genres, etc., and each publisher, developer, tag, trailers, etc. Can be associated with multiple games.

```
CREATE TABLE tags (
    tags_id INT AUTO_INCREMENT NOT NULL,
    tags_names VARCHAR(50),
    PRIMARY KEY (tags_id)
);
```

```
CREATE TABLE game_tags (
    game_tags_id INT AUTO_INCREMENT NOT NULL,
    tags_id INT,
    games_id INT,
    FOREIGN KEY (games_id) REFERENCES games(games_id),
    FOREIGN KEY (tags_id) REFERENCES tags(tags_id),
    PRIMARY KEY (game_tags_id)
);
```

The tables 'reviews', 'media', 'user_data', and 'system_support' establish a one-to-one relationship with the 'games' table. Each game can have only one entry in these tables, and each entry in these tables is linked to a specific game.

Data Elimination:

```
games_id INT UNIQUE,
```

The unique values in the 'media', 'system_support', 'users_data', and 'reviews' refering to the games_id eliminate duplicates, this is because the python script this duplicated games are being eliminated from the games table, but here I have to do this way to assuring uniqueness in the 1:1 tables.

Primary Keys:

```
games_id INT AUTO_INCREMENT NOT NULL,
PRIMARY KEY (games_id)
```

Each table has a primary key defined and with a NOT NULL property, which ensures a unique identifier for each record in the table.

Foreign Keys:

```
FOREIGN KEY (games_id) REFERENCES games(games_id),
FOREIGN KEY (categories_id) REFERENCES categories(categories_id),
```

Foreign keys establish relationships between different tables, ensuring data integrity and referential integrity. They link records in one table to the corresponding records in another table.

Data Types:

```
tags_id INT AUTO_INCREME
tags_names VARCHAR(50),
```

The data types for each column have been appropriately chosen to ensure efficient storage and handling of data.

Data Size Constraints:

Maximum lengths are set for certain VARCHAR fields to prevent data overflow and ensure data consistency.

Script to export the CVS file a normalize MYSQL structure

Here are some key points and explanations about the script:

Handling Empty Cells:

The script uses the pd.isna() function from Pandas to check for empty cells in the CSV file. For example, in the case of the 'Developers' column, it checks if the cell is empty and continues to the next row since the developers names are just stores once in the developers column but for the support_web, the function pd.isna() is changing the value to NULL, because this value need to be stored in MYSQL structure.

```
if pd.isna(developers):
    continue
```

```
if pd.isna(support_web):
    support_web = None
```

Avoiding Data Repetition:

```
if name in unique_titles:
    continue
```

To avoid inserting duplicate data into some tables that don't to store duplicate values like 'games', 'publishers', 'languages', etc. the script maintains a set called unique_ NAME. Example: Before inserting a new game's data into the database, it checks if the game title is already in the set. If it is, it skips that row and moves on to the next one.

Length Restrictions:

```
elif support_web is not None and len(support_web) > MAX_URL_LENGTH:
```

Some fields in the CSV file, such as 'Support url' and 'Support email', may have data that exceeds the maximum length allowed in the database. To handle this, the script truncates the data to fit within the allowed length before inserting it into the respective tables.

Splitting Multiple Data:

```
genres_list = genres.split(',')
```

Some columns in the CSV file, like 'Tags', 'Genres', 'Categories', 'Screenshots', and 'Movies' (trailers), may contain multiple values separated by commas. The script uses the split() function to split these values into individual elements and processes them accordingly. For example, it splits the tags, genres, and categories into lists and adds each tag, genre, or category separately to their respective tables.

Using Lists and Appends:

```
if trailers_tuples:
    sql_query = "INSERT INTO trailers (trailers_URL) VALUES (%s)"
```

```
trailers_tuples.extend([(m.strip(),) for m in trailers_list])
```

The script uses lists to store data temporarily before inserting it into the database.

For example, it creates lists such as games_tuples, trailers_tuples, screenshots_tuples, etc., to store data that will be inserted into their respective tables later using the executemany() function.

Comments for Understanding:

The script includes comments throughout the code, providing explanations for various steps and operations. These comments help readers understand what each part of the code is doing and its purpose.

```
********** INSERTING MEDIA TABLE **********
```

SQL Queries

The SQL queries in the script are used to insert data into various tables in the database. This query inserts a new row into the specified table with the provided values for each column. If there is a conflict with a unique key or primary key in the table, it performs an update instead of an insert. This way, it avoids duplicate entries and updates existing records with new values.

```
sql_query = "INSERT INTO media (games_id, cover_img, game_website) VALUES ((SELECT games_id FROM games WHERE title = %s), %s, %s) O
```

Error Handling:

```
except mysql.connector.Error as err:
    print("Error inserting data into genres table:", err)
    connection.rollback()
    cursor.close()
    exit()
```

The script includes error handling to handle potential errors during the data insertion process. If an error occurs, it rolls back the transaction and prints an error message with the name of the table before exiting the script.

Overall, the script is well-structured and handles various scenarios that may occur during the data import process. It ensures that the data is inserted into the database in a normalized manner, following the defined relationships between tables. The script's use of libraries like Pandas and MySQL Connector simplifies the data handling and database operations, making the import process efficient and reliable.

Creating a web app

1) Open MYSQL IN terminal with the command mysql
2) Connect a user in mysql environment and the make have all access. Copy this in the terminal

```
CREATE USER 'sqluser'@'127.0.0.1';
```

```
'sqluser'@'127.0.0.1' IDENTIFIED WITH mysql_native_password BY 'password';
```

```
GRANT ALL ON *.* TO 'sqluser'@'127.0.0.1' WITH GRANT OPTION ;
```

```
FLUSH PRIVILEGES;
```

3) Copy all the database.sql file in mysql terminal as it is

4) Open a new terminal and install pandas and mySQL connector with this command in terminal

```
pip install pandas mysql-connector-python
```

5) Run the python file in the play button in the corner or use in terminal cvs_injector.py
6) Start the app with the command Run web app: node index.js
7) Open the port localhost:8800

The main.js file

Routing and Database Queries:

The script defines various routes to handle different types of data: games, reviews, system support, users' data, sinopsis, categories, tags, genres, publishers, developers, audios, subtitles, reviews of media, trailers, and screenshots.

Each route retrieves data from the MySQL database using SQL queries and the db.query function.

Dynamic Rendering:

The application dynamically renders HTML views using EJS templates. Data fetched from the database is passed to the corresponding EJS templates for rendering.

Date Formatting:

The formatDate function is defined to format dates retrieved from the database into a more human-readable format.

Routes for Individual Game Data:

For each type of data (reviews, system support, users' data, sinopsis, categories, tags, genres, publishers, developers, audios, subtitles, reviews of media, trailers, screenshots), there are routes to display data for a specific game.

The routes follow a similar pattern:

Retrieve specific data (e.g., reviews) associated with a game using SQL queries.

Render the corresponding EJS template and pass the retrieved data along with the game title.

Navigation:

The script provides navigation links in the top navigation bar (topnav) to access different sections of the application, such as Home, Reviews, Media Reviews, System Support, and various data categories.

Modularity:

The script follows a modular approach by exporting a function that sets up routes and handling for different data types. This promotes maintainability and separation of concerns.

Flexibility:

The script is flexible and easily extensible, allowing you to add more data types, features, and routes as needed for your application.

Index.js file

- Import libraries necessary for the code
- Initialize Objects and Declare Constants
- Establish Database Connection
- Configure Middleware
- Require and Configure Routes
- Serve Static Files
- Templating Engines
- Starts the Server

The ejs files

There is 25 ejs files, all the multiple to multiple data has two pages, the all_ 'name' and 'name', the ones that start with all_ are the general table that shows all the data for all the games, and the 'name' are the ones that are linked in the index page, so basically you can look for any game in the index page and if you want to know anything like the reviews,publishers,genres, of the specif game, your just have to press in the view "name link there is in the page "

For the 1:1 relationships, the table do both jobs, as you can see all the reviews and one riview with the same method, but the 1:1 relationship make it possible to do it that way.

Originality

Cleaning and populating the data turned out to be the most challenging aspect of this project for me. The script itself presented numerous complexities and unpredictabilities, requiring about a week of focused effort. During this time, I not only developed the script but also designed the database, frequently updating the SQL queries and even dropping tables to accommodate changes. My approach involved a mix of ingenious solutions and some that were admittedly less elegant.

The cvs_converter function proved to be the pinnacle of complexity and difficulty in my coding journey. Drawing inspiration from an example I encountered on Stack Overflow, I initially based my code on an index.js file. As I progressed, I gradually shifted to the main.js app, devising preliminary queries and integrating them with the SQL tables. Leveraging the power of AI, I automated the creation of templates

using the app.set script. Reflecting on the process, I realize that while perfecting the Python script was essential, I might have benefited from dedicating more time to enhancing my web app.