SOEN 363 Team Project Phase 1

Malcolm Arcand Laliberté 26334792

Thaneekan Thankarajah 40192306

Hao Yi Liu 40174210

Jonah Ball 40178421

In the first phase of our project for SOEN 363, our team worked on merging gaming data from two different sources: IGDB and Rawg. We ran into a few hurdles, especially with figuring out how to stitch together data from both APIs. We used tools such as MySQL Workbench, Python to automate data handling, and Git to keep all our work in sync. Our main goal was to make sure we could pull the specific data we needed from both IGDB and Rawg and then find a way to fit it all together into one database.

Link to repository: https://github.com/Shredsauce/SOEN_363_PROJECT/

APIs

The APIs we chose were IGDB [1] and Rawg [2]. There are differences in the way the requests are made as well as the formatting of the response. For this reason, separate functions had to be created in order to populate the database.

IGDB allows for a more specific request.The parameters resemble that of an SQL query. All the necessary information can thus be obtained from a single request to the /games endpoint.For example, to retrieve the platform names and platform families we simply include 'platforms, platforms.name, platforms.platform_family' to the request's fields section. So to include the game's summary, we simply include 'summary' to the request.

Rawg's /games endpoint, on the other hand, returns each game's platform information automatically. This makes for a larger response which includes unnecessary information. Furthermore, it is impossible to retrieve a game's summary from the /games endpoint without making another request to their server. Rawg does have its advantages in filtering, which is a feature we used when creating our JSON mapping files.

Tools Used

MySQL + Workbench

We chose MySQL over PostgreSQL due to our team's existing familiarity. MySQL has the added benefit of being more popular which means more help resources are available. To run queries and DDL/DML files, we used MySQL Workbench[3] in conjunction with MySQL Server/XAMPP.

dbForge Studio

We used dbForge Studio[4] to generate the DML statements. We used existing database records since we have to handle large volumes of data.

Postman

Upon selecting MySQL as our database management system (DBMS), we next focused on managing API requests for our selected APIs. We utilized Postman[5], a tool offering a straightforward user interface for sending POST requests and previewing their JSON responses. This approach helped us understand the data structure we would be dealing with and guided us in formulating effective API requests, essential for organizing and storing data in our database efficiently.

Python + JSON

To streamline data collection, we leveraged Python scripting for automation, enabling quick and efficient acquisition of large datasets. We used the requests library for sending POST requests, and specifically for IGDB, we employed a Python IGDB API Wrapper[6] to enhance code readability. This approach ensured all received data was converted into JSON format and stored in files, simplifying preliminary data inspection. Before populating our MySQL database using Python DML scripts, we conducted a preliminary mapping of datasets from different APIs (e.g., matching RAWG and IGDB game IDs based on game names). This method facilitated a seamless database population process, as it allowed us to execute mapping logic in advance, streamlining the insertion of mapped data into the MySQL database.

Github Repository

During our first meeting as a team we created a Github[7] repository. This ensured that every member was able to push and pull their files, making collaborative work easier. This was an important step seeing as the number of both scripts and generated JSON files quickly grew. Integrating source control enhanced our workflow by enabling us to modify files confidently. If a change ended up breaking our project, we could easily revert back.

Challenges

The main challenge that our team faced was in figuring out how to map the data sets together. We initially hadn't decided on how to efficiently merge the two data sets together and it took a fair amount of thinking and experimenting to come up with the method of processing the data as JSON before inserting it into the database. We also struggled to decide on what information to retain from the APIs since there was a large amount of complexity to the data being received. By leveraging JSON, however, we were able to easily identify common attributes and relationships between the datasets, such as matching entity names or IDs, which facilitated a more seamless merging process. We were also able to use it to identify which data would be relevant to our project. This method not only optimized our data integration workflow and helped our data structuring, but also ensured that the data insertion into our MySQL database was both accurate and efficient.

An unexpected challenge that came up while working on our project was after a project cleanup. The number of files in the project was growing. Scripts and generated json files were moved into their separate folder. Half of our team is using PyCharm for Python development, while the other half uses VS Code. Both editors handle the environment root differently which causes issues when referencing other files in the project. This issue was fixed by forcing the files to use the project's root when executing.

One of the initial approaches to linking the datasets was to populate the database with one dataset and then find the equivalent entries in the other. We started off inserting entries using IGDB's API. For each IGDB game, we sent an individual request to Rawg's API for the equivalent game. This was not ideal since we had to make a call to Rawg for every IGDB game. Not only was the script slow to run, but we were making an unnecessary amount of calls to Rawg.We were only including games that were present via both APIs. We came up with the alternative approach of populating the database separately using IGDB and Rawg. It should be noted that each API has a limit per request, so multiple requests to each API was necessary. IGDB's games are inserted first. Afterwards, games obtained from Rawg are compared against those already present in the database.

The way we compare if two games are the same is by exact game name match as well as release year. This avoids a possible scenario where two games may have the same name. However, this is not a perfect way to compare games. Games may be entered into both Rawg and IGDB with different names. There is also the unlikely scenario where two different games released in the same year may have the same name.

We also encountered several challenges during the process of writing SQL queries. Our main issue was the keywords compatibility in MySQL. MySQL Workbench doesn't recognize certain keywords like 'inner join', 'full join', 'intersect', and 'except'. In order to navigate around this restriction, we decided to use alternative approaches. Instead of using 'inner join', we simply opted for the 'join' keyword. However, since there wasn't a direct keyword equivalence for 'full join', we combined results from the left join and right join with a union operator which simulates the desired outcomes.

```
(Select G.game_id, G.name, Ge.genre_id, Ge.name
From game G
Left join game_genre GG ON G.game_id=GG.game_id
Left join genre Ge on GG.genre_id=Ge.genre_id)
Union
(Select G.game_id, G.name, Ge.genre_id, Ge.name
From game G
Right join game_genre GG ON G.game_id=GG.game_id
Right join genre Ge on GG.genre_id=Ge.genre_id);
```

We used 'Union' to demonstrate the 'full outer join' query:

```
Select G.game_id, G.name, Ge.genre_id, Ge.name
From game G
Full outer join game_genre GG ON G.game_id=GG.game_id
Full outer join genre Ge on GG.genre_id=Ge.genre_id
```

'Intersect' and 'except' only appear as visual bugs as we are able to execute them even though they are shown as errors.

Trigger

Our database contains the following trigger which prevents the deletion of a platform that has games associated with it.

```
DELIMITER $$
CREATE TRIGGER Check_Games_Associated
BEFORE DELETE ON platform
FOR EACH ROW
BEGIN
   DECLARE game_count INT;
     SELECT  COUNT(*)  INTO  game_count  FROM  game_platform  WHERE  platform_id  =
OLD.platform_id;
   IF game_count > 0 THEN
       SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot delete platform as there are games associated
to it.';
   END IF;
END$$
DELIMITER ;
```

The following is the method used to ensure it functions correctly.

First we get a list of platforms and how many games are on it.

```
Select p.platform_id, p.name, count(gp.game_id)
From game_platform gp
```

```
JOIN platform p ON gp.platform_id = p.platform_id
Group by platform_id;
```

We see that there are a few platforms that only have one game associated with it. We'll use one of these for our test. For example:

Platform 'Family Computer Disk System' with platform_id 9.

Next we run a command to attempt deleting the platform with platform_id 9:

```
DELETE FROM platform WHERE platform_id = 9;
```

Our trigger successfully prevented the deletion of the platform and we get the following error message:

'Error Code: 1644. Cannot delete platform as there are games associated to it.'

Next we run a command to find the single game associated with the platform we're trying to delete.

```
SELECT game.* FROM game
JOIN game_platform ON game.game_id = game_platform.game_id
WHERE game_platform.platform_id = 9;
```

We get a single row with a game named '19: neunzehn' with game_id 8.

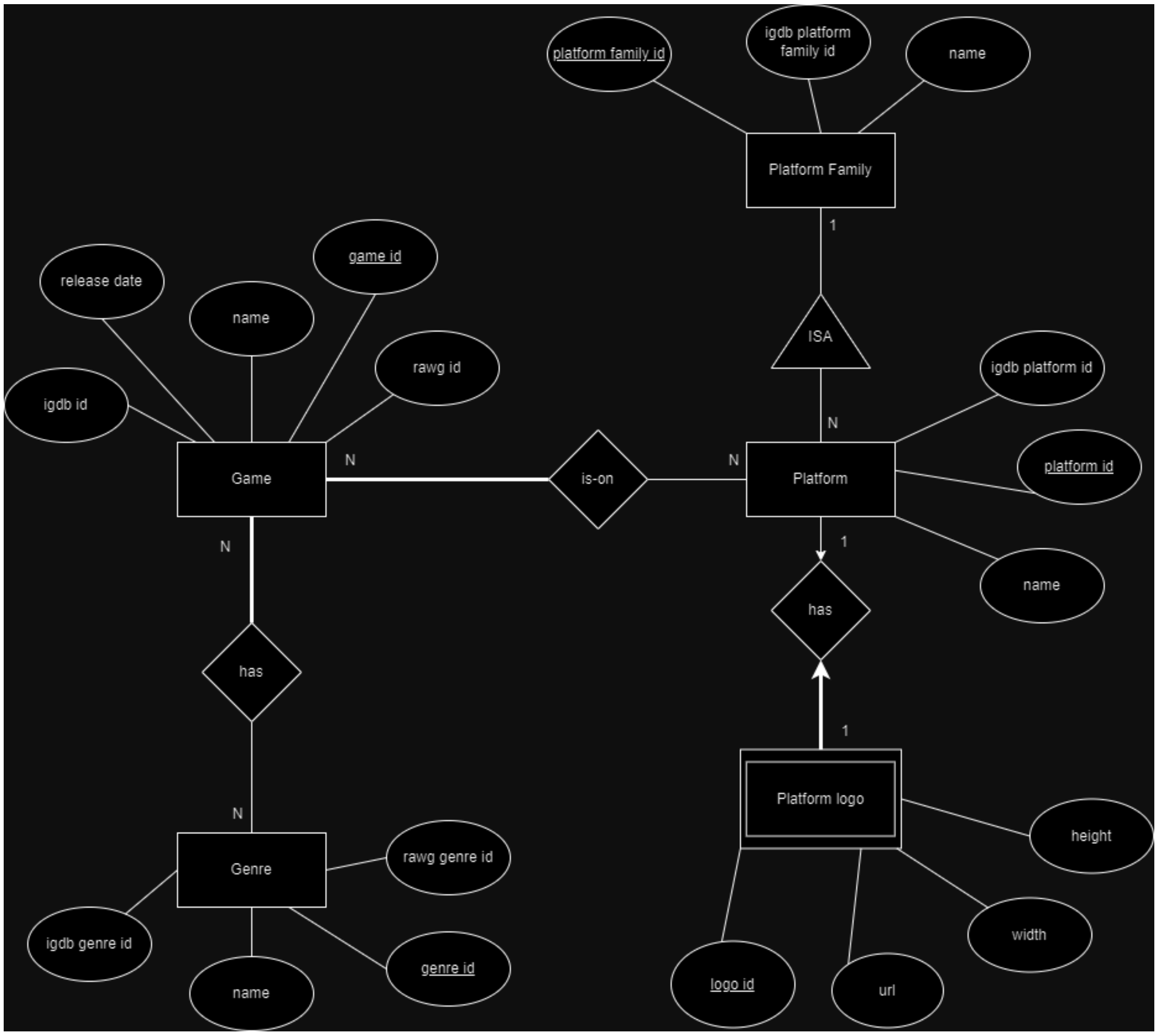
We run a command to delete the game. We make sure to first delete the dependent rows in game_genre and game_platform

```
DELETE FROM game_genre WHERE game_id = 8;
DELETE FROM game_platform WHERE game_id = 8;
DELETE FROM game WHERE game_id = 8;
```

We can now run the delete platform command again. This successfully deletes the platform at platform_id 9 since there are no longer any games associated with it.

```
DELETE FROM platform WHERE platform_id = 9;
```

Entity Relationship Diagram



The Entity-Relationship Diagram (ERD) shows the Game entity having total participation in its relationship with the Platform entity, both entities having a many-to-many relationship with each other. The Genre entity has partial participation while Game has total relationship, as every game needs a genre; both have a many-to-many relationship with each other as a genre can have many games and a game can have more than 1 genre.

The Platform Family entity has an ISA relationship with the Platform entity. The Platform Family entity refers to a company, example: Playstation, so the Platforms would be PS3, PS4, etc.

The Platform logo entity contains the information for each platform's logo. This is a weak entity since without the Platform entity, you would not be able to identify it by its attributes. Platform logo also has total participation with Platform, while Platform has partial participation, since all platforms do not have logos, but all platform logos must have a platform. Both of these entities have a one-to-one relationship with each other.

Translating the ERD to MySQL leads to a total of 7 tables being created:

- game
- genre
- platform
- platform_family
- platform_logo
- game_platform
- game_genre

For all of these tables, we had decided to create unique primary keys that were independent of the IGDB and RAWG ids, an example of this is game_id, platform_id, etc.

game_platform and game_genre are essentially a translation of the relationship both pairs of tables have with each other; Game is-on Platform and Game has Genre, respectively. In both of these tables, the primary key consists of the primary keys from the original table. For example, the game_genre table combines the game table's game_id and genre table's genre_id, their respective primary keys, to create the primary key.

Here is the DDL snippet to demonstrate this:

```sql
CREATE TABLE game_genre (
   game_id INT,
   genre_id INT,
   FOREIGN KEY (game_id) REFERENCES game(game_id),
   FOREIGN KEY (genre_id) REFERENCES genre(genre_id)
   PRIMARY KEY (game_id, genre_id)
);
```

References

[1] IGDB API: https://api-docs.igdb.com/
[2] Rawg API: https://rawg.io/apidocs
[3] MySQL Workbench: https://dev.mysql.com/downloads/workbench/
[4] DBForge Studio: https://www.devart.com/dbforge/mysql/studio/main-overview.html
[5] Postman: https://www.postman.com/
[6] Python IGDB API Wrapper: https://github.com/twitchtv/igdb-api-python
[7] Github: https://github.com/