July, 2025

Mario Graph Database

Authors:
Vincenzo Siano 934168
Francesco Renna 925031
Selin Dökmen 925034

Contents

1 Introduction

This project aimed to create a visual network of relationships in Mario Universe and aimed to achieve that by creating a graph database that contains relationships between the characters and the games of the Mario franchise, as well as the relationships between the characters in the game series. A network of characters which are mainly plot's characters, game bosses and general enemies have been created, which are linked to the games of the Mario franchise among different gaming consoles. The graph also displays personal relationships between other characters. Web and API scraping has been done to obtain the data and also a Kaggle dataset has been used to enrich the data. After the Data Acquisition step, the data has been cleaned and integrated with the other data files from the other sources. After making sure of the quality of the data, it has been stored on a NoSQL graph database that has been constructed on Neo4j.

2 Data Sources

For the acquisition of data, three distinct approaches were employed: Web API [2], Web Scraping [3], and the utilization of a videogames dataset sourced from Kaggle [1]. Web APIs (Application Programming Interfaces) are used as structured communication protocols through which interoperability between different software components is enabled. Predefined functionalities and data endpoints are exposed, with responses typically returned in JSON (JavaScript Object Notation) format—a lightweight data-interchange format based on key-value pairs (objects) and ordered collections (arrays), often featuring nested structures. Web scraping, by contrast, is conducted through the automated extraction of information directly from web page content, with formalized APIs being bypassed. Additionally, a curated dataset was integrated into the workflow from Kaggle. Through the combination of these three data acquisition techniques, comprehensive coverage was achieved and data robustness was enhanced for subsequent analysis and integration.

Our Data Acquisition step is divided as follows in order to get:

- Characters and Games;
- Enemies;
- Bosses;
- API's general relationships.

2.1 Web API

Giant Bomb API has been used in order to retrieve data about the characters and their relationships with the other characters, whether they are a friend or enemy. The steps of the code that handles the retrieval of API data are as follows:

1. **API Configuration and Authentication**
   The API key was securely loaded from a `.env` file using the `dotenv` package. HTTP requests were handled using the `requests` library, and a custom user-agent header was set to comply with the API's usage policies. A delay between consecutive requests was enforced to adhere to the defined rate limits.
2. **Franchise Search and Character Extraction**
   The *Mario* franchise was queried through the API; this step is needed since the API contains not only Mario API. Once the franchise was located, the corresponding `api_detail_url` was extracted. Subsequently, a follow-up request was issued to retrieve the list of characters associated with the franchise, each identified by name and a reference URL.
3. **Retrieval of the Character's Relationships**
   For each character, detailed data was retrieved from the provided `api_detail_url`. The response was parsed to extract relational fields, specifically *friends* and *enemies*. Each identified relationship was stored in the form:

   (Character Name, Related Character Name, Relationship Type)
4. **Batch Processing and Control**
   To manage the volume of data and respect API usage constraints, the character list was partitioned into batches. For each run, only a specified subset of characters was processed, with the size and index of the batch configurable through predefined parameters.
5. **Data Aggregation and Persistence**
   All extracted relationships were aggregated into a `pandas DataFrame`. The resulting dataset was either written to a new CSV file or appended to an existing one. Headers were included only when a new file was created. This strategy allowed for the incremental collection of data over multiple runs without duplication.

2.2 Web Scraping

To further enrich the information related to the Mario video game universe, web scraping was performed on Mario Wiki, focusing on videogames, main characters, common enemies, and boss characters. All scraping was performed using Python in Jupyter notebooks, leveraging libraries such as requests, BeautifulSoup, and pandas. The approach ensured scalable and structured extraction, respecting the page layouts and preparing data for integration and quality assessment phases. The scraping process was divided into three separate notebooks, each targeting a specific category, discussed in the following subsections.

2.2.1 Characters and Games

The process begins by scraping the index page listing all Mario characters, from which links to individual character pages were extracted. For each character, the following data was collected:

- character's name;
- role;
- list of games the character appears in;

To add more detail to our character data, it has performed another parallel scraping task which is about the character's species (if available) that is located on a specific section of the character's page. After, a comprehensive list of Mario games was extracted, including:

- game title;
- release year;
- platform or console;

The collected data was saved to CSV files for easier processing and integration. Additionally, a sales dataset (sourced from Kaggle) was included to provide additional information related to the games and it has been handled in this part.

2.2.2 Enemies

Similarly, data was scraped from a page listing common and general enemies appearing across Mario games. For each enemy, the following attributes were retrieved:

- enemy's name;
- role;
- games in which the enemy appears;

These records are also cleaned and stored in CSV format. For the enemies it has been implemented a more detailed scraping process which consists of several methods to find the page title of a specific game series, which includes the list of the games the enemy appeared in.

2.2.3 Bosses

The data collection process followed similar steps to the character and enemy pipelines and was stored in structured CSV files. More specifically, the overall scraping logic for bosses and enemies is very similar, since for both it has been used a similar parallel scraping method, although the strategy applied for bosses is more sophisticated:

- It performs a more tailored link-finding strategy;
- There are slight differences in the processing of an individual entity's page. Boss pages have more consistent title structures compared to enemies.
- The biggest difference, though, is that the bosses' part includes an extra enrichment step to acquire species data, since on the Mario wiki the enemies do not have this type of feature.

2.3 Kaggle Dataset

We have downloaded the `Video_Games.csv` dataset from Kaggle in order to enrich the data by adding the sales data for many games. This raw dataset contains data for many games, thus we performed a filtering for those of the Mario franchise. This has been achieved by using both exact name matching and a keyword search (e.g., "Mario", "Luigi") to deal with edge cases.

2.4 Software Architecture

The software architecture of our project follows a modular and reproducible pipeline approach, using Python as the main programming language and Jupyter Notebooks (.ipynb) for interactive data processing. The environment is organized into a sequence of notebooks that perform data acquisition and cleaning, data integration, data quality assessment, and data storage. External data sources are accessed via web scraping (using requests, BeautifulSoup) and the GiantBomb API, with authentication handled securely through a `.env` configuration file.

For data storage and querying, we employed Neo4j Aura, a cloud-hosted graph database platform, which is accessed via the official Neo4j Python driver. The database schema is graph-based, with nodes representing entities such as characters and games, and edges modeling their relationships (e.g., `CHARACTER_IN`, `FRIEND_WITH`, `ENEMY_WITH`). To ensure flexible and efficient modeling, dynamic relationship types have been created.

This architecture facilitates scalability, separation of concerns, and reusability of components. It also supports rapid querying and visualization of relationships across the Mario franchise's universe of characters and games.

3 Data Exploration

In this phase, we explored the datasets collected from multiple sources to understand their structure, identify inconsistencies and prepare the data for integration and analysis. This involved aligning naming conventions and schemas across sources.

3.1 Data Cleaning

The data cleaning process was essential for ensuring the reliability of the scraped and external data. We adopted several key strategies:

- **Robust scraping logic:** For character, boss and enemy pages, we used flexible scraping logic that could adapt to structural inconsistencies across wiki pages. This included fallbacks when specific HTML elements were missing.
- **Filtering irrelevant data:** Non-informative or administrative wiki links were filtered before visiting individual entity pages, ensuring that only relevant characters, enemies and bosses were processed.
- **Deduplication:** URLs and entries were tracked to avoid re-scraping or including the same character/entity multiple times.
- **Rate limiting:** To maintain ethical scraping practices and prevent IP blocks, we introduced delays between web requests.
- **Restructuring tables:** Tables were standardized with consistent column formats. For example, headers such as `System`, `Console` and `Format` were harmonized under a unified label.
- **Text normalization:** We applied lowercasing, removed special characters (such as quotes,

parentheses, and underscores) and standardized console names using a predefined mapping dictionary.

- **Handling missing values:** Rows missing critical fields (e.g., game title, character name, console, or year) were dropped. Missing values in numeric fields like sales were filled with 0 when appropriate.
- **Data enrichment:** Species information was scraped separately and added to character datasets. Sales data from Kaggle was merged using a combination of exact name matching and keyword-based fuzzy matching.

Each of these steps ensured that the final datasets were clean, consistent and suitable for integration, quality assessment and load into a graph database.

4 Data Integration and Enrichment

4.1 Dataset Exploration

We began the integration process by exploring all individual datasets obtained from web scraping (plus Kaggle's dataset) and API calls. These included characters, enemies, bosses, sales data, game information, and inter-character general relationships. Each dataset was inspected for consistency in naming conventions, presence of duplicates, completeness of attributes, and formatting anomalies that could interfere with merging. We also verified that the data structures aligned well with our final graph schema and documented potential challenges, such as varying naming formats for consoles and characters. Our goal was to create two final, clean datasets: one containing all game appearances and another containing all inter-character relationships.

4.2 Correspondences Investigation

To integrate datasets from different sources, we conducted a correspondences investigation, focusing on matching entities based on their names. This involved standardizing the names of characters, games, and consoles by converting to lowercase, removing special characters (such as parentheses, quotes, and underscores), and applying manual mappings where necessary. This process allowed us to identify which entities from the API matched those found in the scraped data and to ensure consistency before merging.

4.3 Data Preparation

The preparation phase involved consolidating and transforming data into a uniform structure suitable for integration.

Specifically:

- Character, enemy, and boss datasets were concatenated into a single dataframe by renaming their respective identifier columns to a common name (i.e., `Figure`). This allows to treat all entities uniformly.
- We merged this unified dataset with the games dataset to add attributes such as `Year`, `Console`, and `Sales` to each game appearance.
- Console names were normalized using a predefined dictionary to account for naming variations across sources.
- Rows with missing or invalid data in essential fields were removed, and platforms appearing less than 10 times were filtered out to reduce noise.
- Game titles were matched with external sales data using exact and fuzzy keyword matching and rounded for consistency.

This resulted in a single, clean dataset saved as `merged_data.csv`, which combines all character and game appearance data.

4.4 Schemas Integration

Schemas integration was the final step before database insertion. The cleaned and merged datasets were aligned with the Neo4j graph schema. Nodes for `Game` and `Character` were generated, and relationships (`CHARACTER_IN`, `ENEMY_IN`, `BOSS_IN`) were formed based on the role of each figure in a game. In parallel, inter-character relationships from the API were filtered and enriched with species information before being saved in `merged_data_API.csv`. Specifically, it created a comprehensive mapping of every character to their species, resulting in a lookup table where each unique character name points to its species. This has been done to avoid the insertion of character nodes without species property, since the API does not provide it.

This dual-schema approach allowed us to cleanly separate appearance-based relationships from character-to-character links, while maintaining referential integrity across both.

5 Data Storage

In this phase, we transformed the integrated, enriched tabular data into a graph-based structure and stored it in a Neo4j AuraDB instance. We have chosen this type of model for the following reasons:

1. This representation allows efficient querying and

visualization of the relationships between entities in the Mario Universe.

2. Data is not in real-time, because there are few write operations for the Mario franchise, and several reads, which are known to be faster in this type of database. In fact, graph-based databases are fast for reads and slow on write.

5.1 Database's Structure

The Neo4j graph model consists of two main node types and several types of directed relationships.

5.1.1 Nodes

We defined two main node types:

- **Character**: Includes all characters (plot's characters, bosses, enemies), with properties the `name` and `species`.
- **Game**: Represents each game in the Mario franchise, with properties of `name`, `console`, `year`, and `sales`.

Each node was created with unique constraints to avoid duplication and significantly speed up data import. For example, game nodes use a composite ID combining name, console, and year to ensure uniqueness. Then, there are indexes on other properties (i.e., year, console and species) to speed up future queries that will filter on these ones. Data was imported from the final, cleaned datasets using Cypher queries and the APOC library to dynamically generate relationships; its key feature is that the type of the relationship is taken directly from the `Relation` column in our data (e.g., `CHARACTER_IN`, `BOSS_IN`, `ENEMY_IN`), which is a flexible way to model different roles. For the Game nodes, we replaced `NaN` in sales filling with `zero` before importing the data; it is a common approach which cleans data at the source to ensure that the DB can handle these values without issues. After, the method for importing character-to-character relationships:

- efficiently populates the graph with them;
- enriches the graph by creating new character nodes for figures who may not have appeared in a game but are part of the franchise;
- carefully fills missing species information without overwriting existing data.

5.1.2 Relationships

The graph supports both appearance-based and interpersonal relationships:

- **CHARACTER_IN**, **BOSS_IN**, **ENEMY_IN**: Link characters to the games they appear in, depending on their role.
- **FRIEND_WITH**, **ENEMY_WITH**: Define general relationships between characters in the Mario franchise, sourced from the GiantBomb API.

All relationships are directed and semantically typed to allow precise traversal and pattern matching.

5.2 Database's Statistics

The final Neo4j graph database contains a total of **1055 nodes** and **9482 relationships**. The node types and relationship types are distributed as follows:

- **Nodes**:
  - `Character`: 664;
  - `Game`: 391;
- **Relationships**:
  - `CHARACTER_IN`: 3324;
  - `ENEMY_IN`: 2438;
  - `BOSS_IN`: 1196;
  - `FRIEND_WITH`: 1456;
  - `ENEMY_WITH`: 1068;

These figures confirm the completeness and richness of the data model, which captures both character-to-game and character-to-character relationships across the Mario franchise. The prevalence of `CHARACTER_IN` and `ENEMY_IN` relationships reflects the wide distribution of characters across games, while the `FRIEND_WITH` and `ENEMY_WITH` relations enrich the graph with inter-character connections.

5.3 Sample Queries

Cypher is Neo4j's declarative and GQL (i.e., International Standard query language for graph databases) conformant query language. Cypher is similar to SQL, but optimized for graphs, as it is intuitive and close to natural language, providing a visual way of matching patterns and relationships by having its own design based on ASCII-art type of syntax.

Several Cypher queries were executed to validate and explore the graph. Below, we report a couple of them.

Top 10 Characters by Game Appearances

First, let's analyze the game appearances of the characters in the Mario franchise by performing a query that returns the top 10 most occurring characters. The results are sorterd in descending order and shown in Figure 1.

```
MATCH (c:Character)-[]->(g:Game)
RETURN c.name AS Character, COUNT(DISTINCT
    g) AS Appearances
```

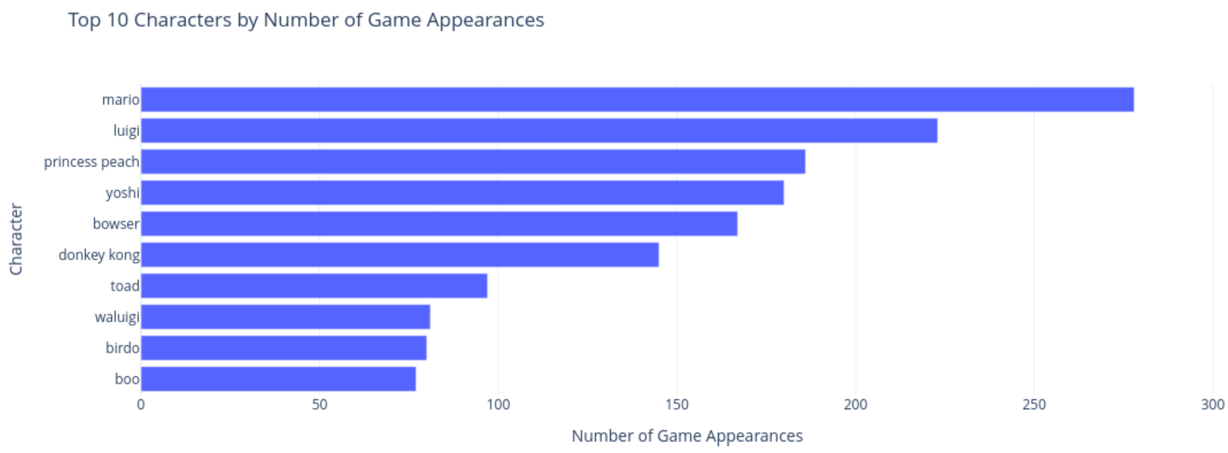

Figure 1: Table output of the first query.

```
ORDER BY Appearances DESC
LIMIT 10
```

Of course, the characters with the most game appearances are 'mario' and 'luigi'.

Wiggler's Game Relationships

Secondly, we have performed another query to find the games in which the character named 'wiggler' appears in. This is accomplished with the following query:

```
MATCH p = (c:Character {name: 'wiggler'})-[
    r]->(g:Game)
RETURN p
```

The ouput graph is displayed in Figure 2.

Friends or Enemies of Mario that appear in a specific game and console

Finally, we wanted to visualized the friends or enemies of 'mario' that have a relationship with the game called 'super mario bros.' which has been published on 'NES' console. This query highlights both inter-character relationships and game appearances ones. The ouput graph is displayed in Figure 3.

```
MATCH (c1:Character)-[r:FRIEND_WITH|
    ENEMY_WITH]->(c2:Character)-[a1]->(g:
    Game)<-[a2]-(c1)
WHERE c1.name = "mario" AND g.name = "super
     mario bros." AND g.console = "NES"
RETURN c1,r,c2,a1,g,a2
```

6 Data Quality

To ensure the reliability and usability of the integrated data, we conducted a data quality assessment performing the following tasks:

1. Compare the characters obtained with the two data acquisition methods (i.e., Web Scraping and API).

2. Evaluation the quality of the data collected for the Mario Graph Database, focusing on these three primary dimensions: completeness, redundancy, consistency.

For the first, we found that there are 26 more characters in scraped data compared to API data, with 88 characters common to both; we observed that the sources are highly complementary. Then, for each data quality dimension, we have checked the data quality of both the raw data and the integrated, enriched data. Furthermore, for the game's release year, we checked also for accuracy and currency.

Completeness

We evaluated the completeness of the datasets by calculating the proportion of missing values. The table-level scores are pretty high (the lowest is for bosses with 82%). Then, the completeness of the columns of each dataset (i.e., species, year, console) are also calculated seperately. In fact, we have high score of completeness of the Species column for the characters (i.e., about 94%), while there is a low score for the bosses (i.e., 27%) because the Mario wiki website often does not provide this information in the bosses pages. Finally, for year and console we don't have any missing values. After the integration, the complete-

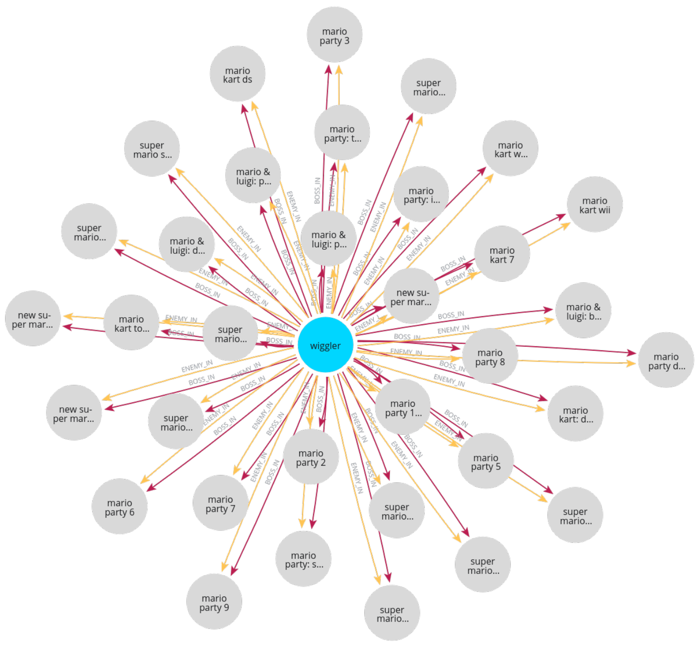

Figure 2: Graph output of the second query (i.e., Wiggler's Game Relationships).

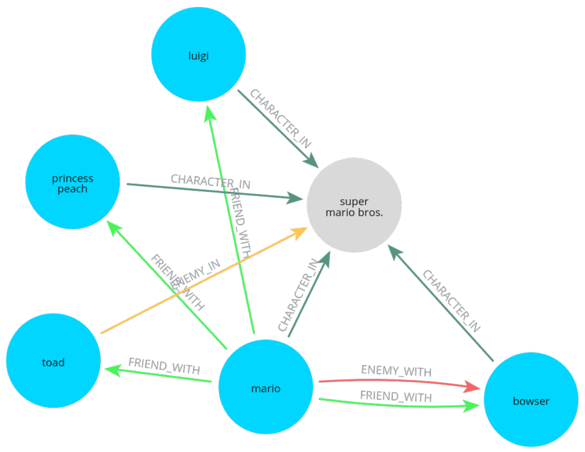

Figure 3: Graph output of the query (i.e., friends or enemies of Mario that appear in a specific game and console).

ness of scraping data is about 88%, while 80% for API data. We also evaluated completeness of sales and species in both the integrated datasets.

Redundancy

To avoid duplication, we applied strict URL tracking during scraping and enforced uniqueness on game and character identifiers during integration. We also checked for duplicate rows using unique entity fields (e.g., `name`, `title`) and removed redundant records before loading into Neo4j. In `character_df`, there are 176 redundant rows, while in `enemies_df`, this number is 42 and in `bosses_df`, this is 10. These numbers all become zero for the merged data (both for scraping and API) as the data integration step handled the duplicates.

Consistency

Text normalization (i.e., lowercasing, punctuation removal and trimming) ensured consistent naming across datasets. Console names were harmonized using a mapping dictionary and character names were standardized before merging with API data. Specifically, The `Console` column in `games_df` is a perfect example of inconsistency, therefore we measured the variety of console names before standardization, observing 73 unique values. Thanks to data integration step, we reduced this number to 27 (a reduction of 63%).

Accuracy and Currency

Game release years and sales values were verified and cleaned by cross-checking across sources. All scraping scripts were executed recently, but users are warned that both the Mario Wiki and Giant Bomb API are subject to change. For the accuracy, the temporal data available with release years allows for accurate time-based analysis. Moreover, the currency is analyzed observing the distribution of the release years that reflects the Mario franchise's history, especially its high output during the Wii and DS era, and recently with the Switch.

Overall, the data quality was assessed as high and sufficient for the analytical goals of the project.

7 Conclusions and possible future developments

We developed a complete data pipeline to construct a Neo4j graph database for the Super Mario franchise. The pipeline spans from web scraping and API integration to data cleaning, integration, enrichment and storage. Using structured CSV datasets and Python with Jupyter notebooks, we modeled relationships between characters and games, as well as character-to-character interactions.

The final graph database supports various types of queries, which allows us to explore appearances across consoles, games, year of release, personal relationships and popularity metrics such as character's frequency and sales data.

Despite the project's success, we identified areas for improvement. Firstly, not all characters could be matched across sources, which may be addressed by adding a fuzzy-matching module or expanding to other data sources to integrate even more. Secondly, some relationships from the Giant Bomb API are incomplete or could be ambiguous, which could benefit from further manual validation or a more structured data source.

Future developments of this database could be the following ones:

- Extend with new games and characters as they are released or updated on the sources used.
- Build recommendation systems.
- Create a front-end dashboard for the fanbase, including interactive exploration of the various relationships in the Mario universe.
- Neo4j is increasingly being used in generative AI contexts, especially with LLMs, Retrieval-Augmented Generation (RAG), and knowledge graph-enhanced applications. Our graph database could be expanded and empowered by generative AI. LLMs can use the graph to "reason" over knowledge paths. Graphs can define which nodes to retrieve and how they're related, improving the relevance of retrieved context for RAG. For example, this could be done using LangChain or LlamaIndex with Neo4j connectors.

The modular architecture and reproducible workflow ensure that the project is easily maintainable and extensible over time.

References

[1] Video Game Sales Dataset (Kaggle). URL: `https://www.kaggle.com/datasets/thedevastator/video-game-sales-and-ratings`.

[2] GiantBomb API. URL: `https://www.giantbomb.com/api/`.

[3] Mario Wiki. URL: `https://www.mariowiki.com/`.