SWAPI report
Data Acquisition, extraction and storage - project

Ángel Luque Lázaro, Gabriel Ravelomanana Nampoina

January 7, 2024

Github repository:https://github.com/OrangelLuke/DataAcquisitionSwapi

1 Introduction

Our initial source of data for the project was Letterboxd, extracting information from its API. However, this API is in beta phase still and, even though we sent a request for access well in advance, we didn't receive any reply. Therefore, we had to switch to a different (open, public, free) API: SWAPI. This API contains information about the Star Wars franchises, divided into people, planets, vehicles, starships, films and species, each with their attributes and relations among them. We believe that this dataset is perfect to learn to perform extraction through API and validate, correct and store data from a complex enough dataset.

Another major setback has been that one of our team members stopped replying before starting the actual work, so finally, our project that was envisioned for three people, has been done only by two.

2 Data extraction

As aforementioned, the data extraction has been performed by API calls. For this, we understand how the data is structured in the API (https://swapi.dev/) and design a python code that, using the rqquests library, extracts all the information, going automatically through all entities and all their respective pages. We save this information into a python dictionary, and then export the data into json files.

3 Data validation

The first phase of validation is done during the extraction. In the first request for a resource (e.g., characters), the swapi API includes in the response the number of items that we should expect (e.g., 82 people). These items are spread throughout pages containing 10 items each (therefore, for people, throughout 9 pages), each page containing a link to the next one (null in the last page). Our validation here is to check that the number of items extracted is the same as the one indicated in the first request. If this doesn't happen, we indicate that some information was lost.

Later, our main task of validation is finding out which fields are incomplete. We consider and incomplete or invalid field one that is either empty or "unknown". We use a function that tells us for each item in each entity (e.g. vehicles), which fields are invalid (e.g. 'vehicles7': ['consumables', 'pilots']). This information is stored in a dictionary.

A final and straight-forward form of validation is showing that the relations between entities and objects hold, as we were able to use a relational database for our data.

4 Data cleaning

The first most basic thing we do to clean our data is assigning an unique id to each object that can identify it, and remove the links that the API used. This id is created by using the name of the entity and the number it had in the API (e.g. 'vehicles7').

Our biggest attempt at data cleaning is to complete the missing information in our data. For this, our idea has been to ask chatGPT to complete the missing information. To achieve this, we identify the missing values, and generate chatGPT prompts that asks the AI service for this information. In this prompt we also specify that if the information is not known, just return 'unknown'.

This process is rather tricky due to some characteristics of our data. Firs of all, we identify our data by the aforementioned (arbitrary) ids. However, chatGPT does not know about these, so we need to translate these ids to the actual names in the prompt. Some transformation is also done to the name of the fields we are requesting to ease the understanding (like replacing underscore for space). Additionally, when we request information about a field that is related to other object in our data (e.g., the homeworld of a character is a planet from the entity planets), we need to provide in the prompt a list of all possible values, as we need to keep consistency in the relations between objects (e.g. for homeworld, we can't just accept any planet, but only those that are in the 'planets' entity already). Finally, to show chatGPT the format in which we want the info to be outputted, most of the times we need to provide a real correct example.

The process as to how all of this has been done can be seen in the code (accompanied with comments).

Finally, we need a function that reads the information given by chatGPT and updates the values in our data. It is noteworthy that this function only updated information that was invalid/incomplete, and not the fields that were correct already (even if chatGPT includes them in its response).

This technique helped us to fill some missing information. However, most of missing information still remained missing, even after using our tool. This feels rational, in the sense that if the information is not available in a StarWars database (probably because this information has not been mentioned in the movies, series, books, etc), it is more likely that that information is not available anywhere, and that ChatGPT won't find more.

There also exists a ChatGPT API that we could have used, and that is easier to use and call, using openai library in Python. But, since that API is not free, we implemented this toy semi-manual version to show that it works. A bigger dataset would have been more problematic and would have cost much time using our semi-manual version.

5 Data storage

To store our data, we choose to use a relational database management system called SQLite, to better represent complex relationships between our data. The use of SQLite also eases information retrieval and allows query optimization that ensures that queries are processed quickly.
We create a table for each of our entities:

- people, with a primary key string named "id"
- planets, with a primary key string named "id"
- films, with a primary key string named "id"
- vehicles, with a primary key string named "id"
- starships, with a primary key string named "id"
- species, with a primary key string named "id"

To handle many-to-many interactions (for example, someone might have interaction (piloted, or transported by) different starships, while a starship might have interacted (piloted by or transported) different persons), we choose to create intermediate table, that just represents the interaction.

To do that, we worked on the json documents obtained when we extracted data using the API. An entry in the json document for a certain table would look like this:

$$[attribute1 : value1, attribute2 : value2, attribute3 : [value3, value4, value5]]$$

where attribute3 represents here the one-to-many interactions between that table (for example people) and another table (for example starships).

We used Python and the Pandas library to process the data and transform that entry into three entries where their attribute3 would each have following values: value3, value4, value5. Handling that process is really hard in SQLite; thus, we decided to keep using Python and Pandas.

Then, we create the intermediate table, with two attribute (each attribute referring to each of the two tables of the interaction).

After processing the data, the final step is to store original tables and intermediate tables using SQLite.

The final result would look like this:

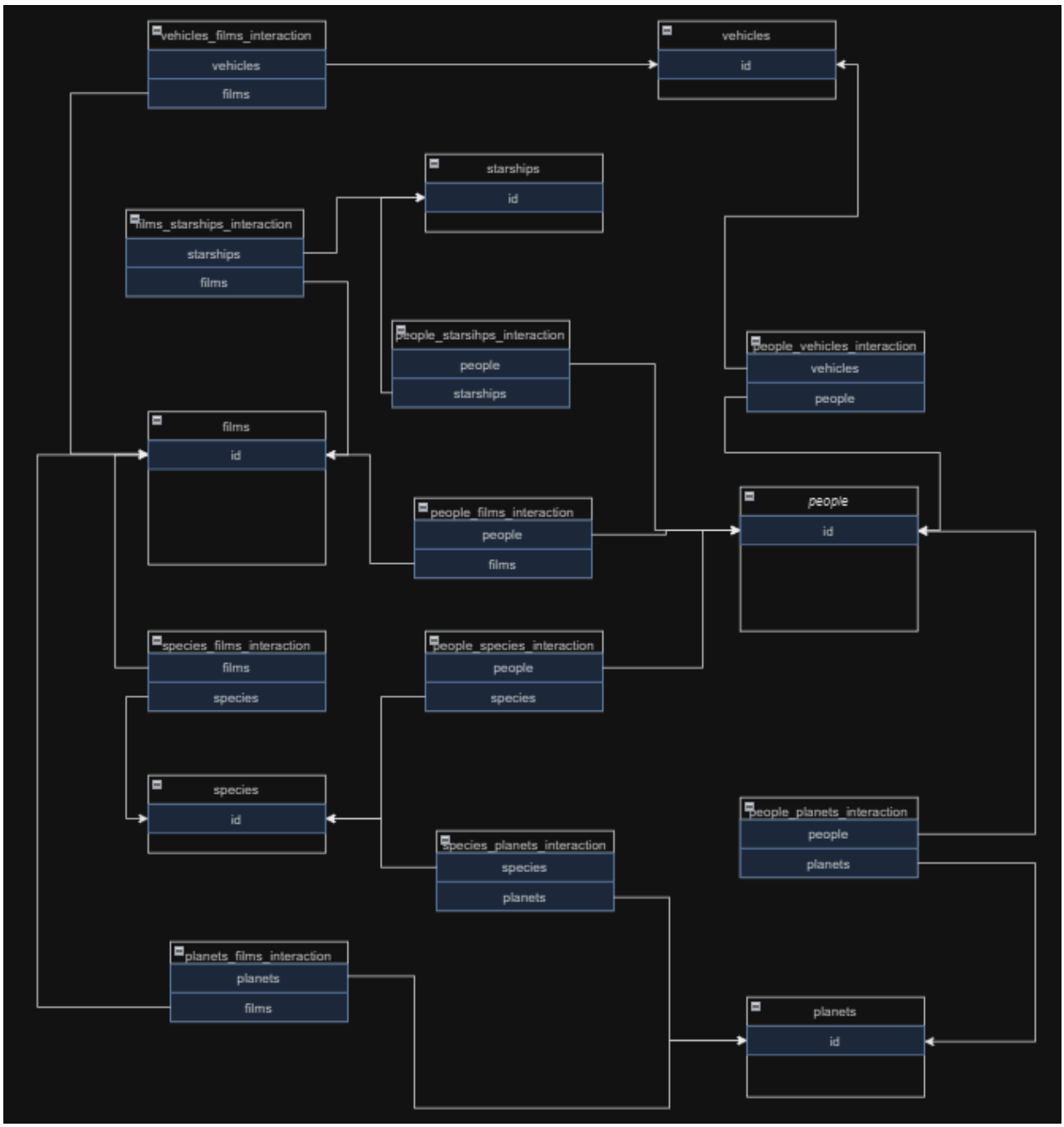

Table 1: Diagram of the final result

We can see here, that relationships are more or less complex between our entities. Using anything than a relational database management system would have been arduous.

6 Remarks about code structure

In order to show the process of our work, we have decided to keep intermediary code in the repository. Hence, here's a little explanation as to what each of the files contains:

- **ExtractData.py**: first approach to extracting data through API. Contains the data extraction, its validation/analysis (number of items, and invalid/incomplete values) and its export to json files.
- **swapi.py**: includes most of what the previous file had, but also adds all the process of data correction with chatGPT (as well as reading the data from its response). Only contains some examples of this process.
- **correct_data.py**: same as previous file, but instead of only some examples, contains all possible prompts and responses from the data correction process (entire process).

- **create_intermediate_table.py**: handles the many-to-many interactions using pandas library, and create intermediate tables.
- **create_table.sql**: sqlite queries that store all original tables and intermediate tables into a db file.

Being strict, the entire project can be followed only by the last 3 files (first 2 are intermediary code).

7 Conclusion

Through a meticulous process, we accomplished several key objectives that significantly improved the usability and quality of the data that we extracted from SWAPI. Throughout this project, we encountered challenges typical of data extraction and cleaning processes. Our initial focus centered on leveraging Python's versatility and the API interaction capabilities it offers. The use of the Pandas library has been a great success in processing and structuring data into usable format. SQLite provided a storage solution compatible with Python, and that is easy to setup, providing a platform for storing our cleaned data. In conclusion, this project helped us to gain invaluable hands-on experience in data extraction, cleaning, and storage processes.