Final Project

Non-relational databases. UPC - MERIT

Jordi Masip i Riera

10/10/2025

1. Introduction

This project focuses on developing the backend of a movies website (similar to TheMovieDB.org) using three NoSQL databases: MongoDB, Redis and Neo4j. The project aims to demonstrate the use of these NoSQL databases in a single backend system, solving real-world problems. This project must be completed in groups of two students.

2. Technical details

Building a dynamic website requires knowledge of several technologies that are out of the scope of this course. For that reason, we have already implemented some parts of the website for you. We have used Python, Flask (Web Framework) and Jinja2 (template engine) to build a three-page website:

- `/` - main page
- `/movie/<tmdb_id>` - movie details page
- `/search?query=<movie-title>` - search results page

To deliver a functional working website, some functions currently return hard-coded data. Your task in this project will be to modify those functions to work with a real database.

3. Database design

In this project, we will work with two entities: **movies** and **users**. MongoDB will be responsible of storing all the information related to movies. Neo4j will be used to build a recommender system. It will only store the necessary information related to movies and users in order to provide recommendations. Redis will be used to cache the results of expensive queries.

4. Setup environment

4.1 Environment requirements

- OS: Linux (alternatives: Docker or WSL)
- Python 3.11+ (including `pip`)
- MongoDB 7.0+ Community Edition
- Redis v8.0+
- Neo4j 5+

4.2 Download the website's source code

Download `movies_website_src.zip` from Atenea.

4.3 Install Python dependencies

First of all, we need to install some Python libraries. To do so, we will first create a virtual environment to isolate these libraries from the rest of the system (to avoid affecting other applications or services written in Python):

```
$ cd website/
[website]$ python3 -m venv venv
```

Now that we have created a virtual environment named `venv`, we need to activate it to use it:

```
[website]$ source venv/bin/activate
```

> **Attention!** the virtual environment needs to be activated every time a new terminal is opened.

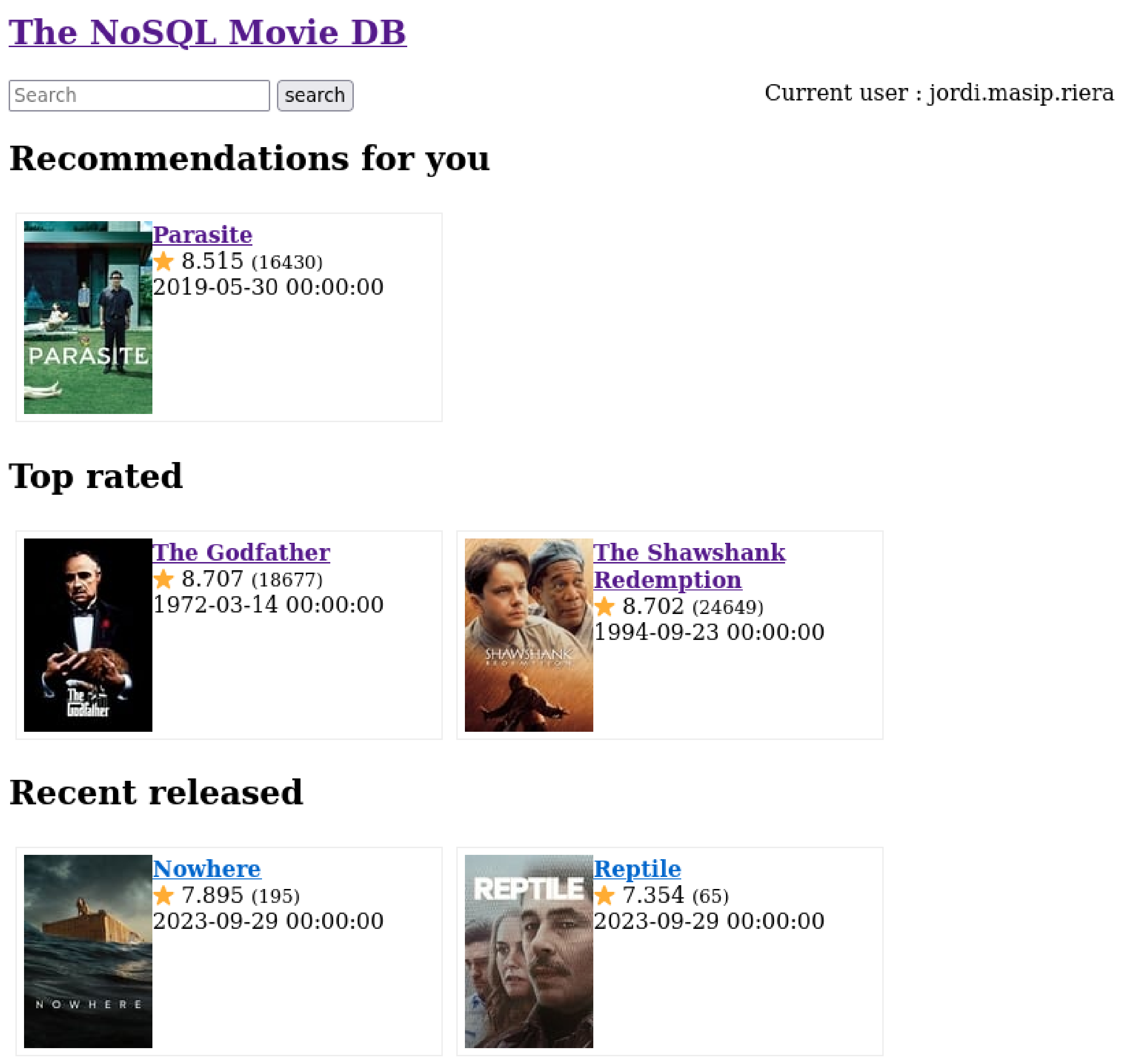
You will see that your shell prompt now looks like this `(venv)[website]$`. Now you can proceed installing the Python dependencies that are defined in `requirements.txt`:

```
(venv)[website]$ pip install -r requirements.txt
```

Finally you are ready to start the web server:

```
(venv)[website]$ flask --app app run --debug
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 946-042-880
```

Open the following URL in your browser http://localhost:5000/ and verify that you see a main page that looks like the screenshot:



The NoSQL Movie DB

Search search Current user : jordi.masip.riera

Recommendations for you

Parasite
★ 8.515 (16430)
2019-05-30 00:00:00

Top rated

The Godfather
★ 8.707 (18677)
1972-03-14 00:00:00

The Shawshank Redemption
★ 8.702 (24649)
1994-09-23 00:00:00

Recent released

Nowhere
★ 7.895 (195)
2023-09-29 00:00:00

Reptile
★ 7.354 (65)
2023-09-29 00:00:00

5. Tasks

Before you begin the development of the website take your time to learn about `pymongo` and `neo4j` Python libraries:

- PyMongo tutorial: https://pymongo.readthedocs.io/en/stable/tutorial.html
- Neo4j python manual: https://neo4j.com/docs/python-manual/current/

According to the Product team, our website needs to provide the following content in each page:

1. Main page:
    - Carrousel of top rated movies
    - Carrousel of recent released movies
    - Carrousel of recommended movies
2. Movie details page:
    - Information about the movie
    - Carrousel of similar movies (based on other users with similar tastes)
    - List of usernames of other users who like the movie
3. Search results:
    - List of movies matching search query

In `services.py` there is already a function implemented to fulfill this requirements, but it currently returns hard-coded data. In the following tasks, you will need to modify all of these functions to use a real database. **Please note that you must follow the same data structure as the returned value**, otherwise other parts of the web application will break.

Task 1 (0p): Import the movies dataset into MongoDB using `mongoimport`. Then update all documents to ensure the value type of a `release_date` is `ISODate()` instead of `string`. If some value cannot be converted to `ISODate`, use `null` as a fallback.

> During the import you will see several warnings like this: `continuing through error: E11000 duplicate key error collection: test.movies index: _id_ dup key`. It is safe to ignore.

Task 2 (2p): Implement the following function:

- `get_movie_details(movie_id)`: returns detailed information for the specified movie_id
- `get_top_rated_movies()`: function which returns top rated 25 movies with more than 5k votes.
- `get_recent_released_movies()`: function which returns recently released movies that have been reviewed by at least 50 users.
- `get_same_genres_movies(movie_id, genres)`: return movies that match at least one genre.

Task 3 (1.5p): Implement the `search_movie(title)` function to search movies by title and sort the results using a custom score calculated as `textScore * popularity`. It should also return facets for these fields: `genre`, `releaseYear` and `votes`.

Task 4 (1.5p): Performance is crucial in our website and our goal is to ensure the Main Page and Details Page take less than 75ms to load. Steps:

1. All `service.py` functions are decorated with `@measure`, which measures the time it takes to run the decorated function and passes the measure to `store_metric(metric_name, measure)`. Implement `store_metric()` and

get_metrics() so that get_metrics() returns the accumulated 90th and 95th percentiles for the given metrics.

2. Now that you have metrics for each service, make the necessary adjustments or improvements to reduce the response time.

Task 6 (2p): Implement the function get_similar_movies(movie_id) to return a list of movies with a similar plot. To achieve this, we will use a pre-trained sentence embedding model to convert each movie plot into a vector representation that captures its semantic meaning. Then, by measuring the similarity of the embedding of the target movie with those of all other movies, the ones with the highest similarity scores will be considered the most similar and returned as the result.

Technical details:

Use the Python library sentence_transformers and the embedding model avsolatorio/GIST-small-Embedding-v0 to create the embeddings. See the following example:

```python
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("avsolatorio/GIST-small-Embedding-v0")

plot = "Young hobbit Frodo Baggins, after inheriting a mysterious ring" \
      "from his uncle Bilbo, ..."
embedding = model.encode(plot, normalize_embeddings=True)
# array([-6.06717207e-02,  2.83089466e-02,  2.55503319e-02, -3.32188420e-02,
#          1.87798720e-02,  6.17786348e-02,  1.21961543e-02,  4.46404554e-02,
#          1.75194535e-02, -2.19592974e-02,  5.17288502e-03, -1.23826422e-01,
#          ...
#          6.09852336e-02,  8.57207086e-03,  1.08801927e-02, -1.12664541e-02,
#         -4.10457216e-02, -2.72842664e-02, -8.69187806e-03, -1.71744209e-02,
#          5.31951059e-03,  7.07699265e-03, -2.19221134e-02,  3.18536460e-02],
#        dtype=float32)
```

Use Redis to store and search similar embeddings. To do so, store each movie embedding in a Hash. This allows us to store the embedding and the movie_id together. Then, we can use the Full-Text Search functionality to create vector index. You will need to specify the number of dimensions (vector length), which for this model is 384, and the similarity function cosine. Here you can find some examples on the Redis-py documentation.

Create a Python script named populate_redis.py which retrieves all movies with a vote_count greater than 500, generates an embedding of the plot (or overview, if plot is missing) and stores it in Redis.

Finally, implement the function get_similar_movies(movie_id) by retrieving the `movie_id` with similar embeddings.

Task 7 (0p): Implement a Python script named `populate_neo4j.py` to import the movies, usernames and likes into Neo4j.

Download the file `movies_likes.csv` from Atenea

Task 8 (0.5p): Implement the `get_movie_likes(username)` function

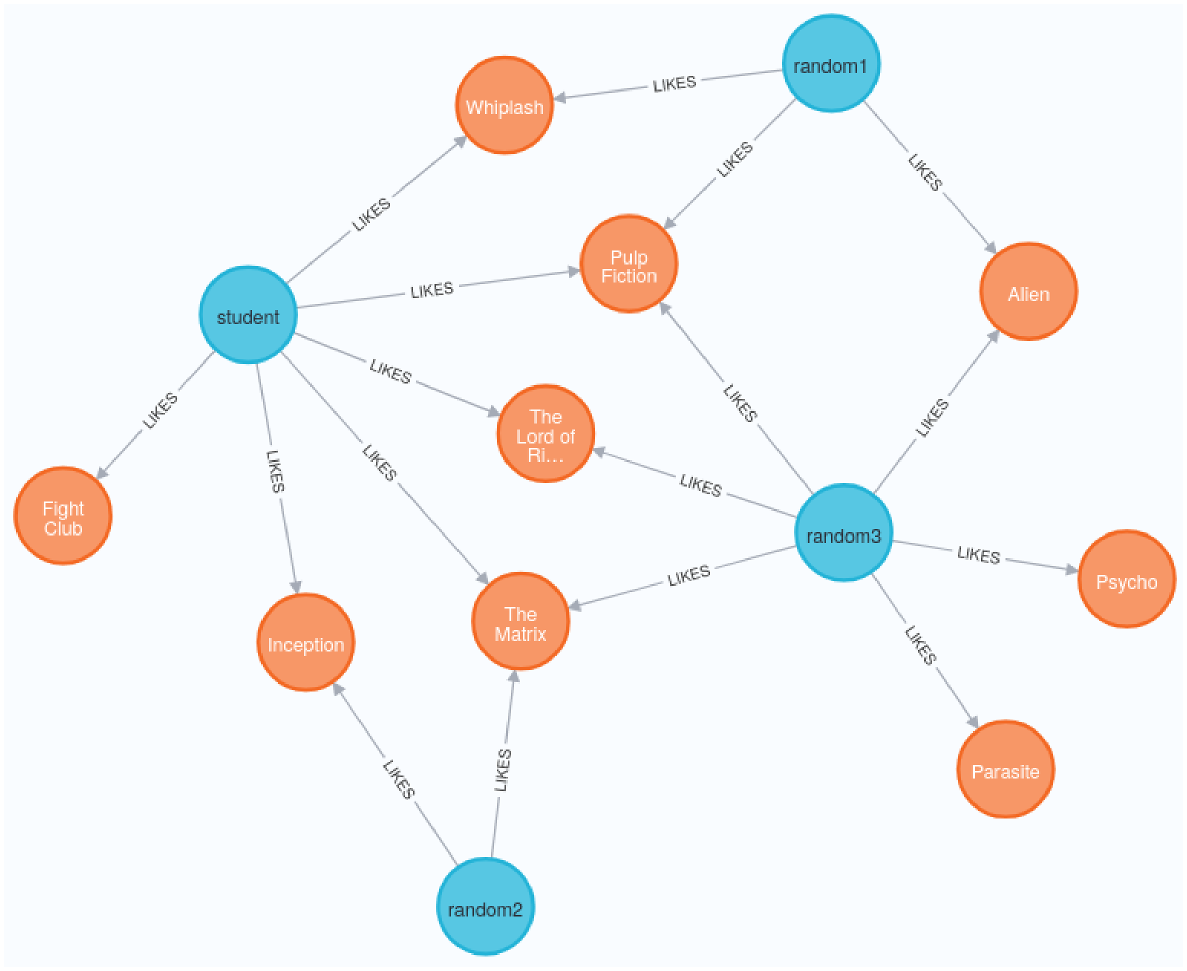
Task 9 (2.5p): Implement a recommender system in `get_recommendations_for_user(username)` using Collaborative Filtering technique. In our recommender system, we will perform the following steps:

1. Look for other users who like the same movies as the active user (`username`)
2. Compute the similarity between the active user and each other user who has at least one movie in common
3. Select the top k nearest neighbors based on a similarity metric
4. Find the movies liked by the top k neighbors that the user has not already liked
5. Rank these movies by the number of likes

We will use **Jaccard Index**, also known as intersection over union, as a similarity metric:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Example: given the following graph we will simulate step by step how the recommender system should work in order to recommend movies to username `student`.



1. We find a list of users who like the same movies as `student` and we obtain `random1`, `random2` and `random3`.
2. Now we need to compute the similarity index for each pair of users. First we will calculate the intersection (movies both users like) and then the union of movies (combination of movies both users like). Finally we will divide the intersection by union to obtain the jaccard index.

Result:

active user	neighbor user	intersection	union	**jaccard index**
student	random3	3	9	0.333
student	random1	2	7	0.286
student	random2	2	6	0.333

3. Now we sort the neighbor users by the jaccard index in descending order and we keep the top k users.

4. Now we find which movies the neighbor users like that active user has not already liked (we assume that the active user has not seen the movie).
5. Finally we rank these movies by the number of likes.

title	likes
Alien	2
Parasite	1
Psycho	1

6. Submission instructions

Create a zip file that contains the source code of your website and a README clearly explaining which team member was responsible for implementing each task.

Please include the name of all team members in the zip file. For example: `student1_student2_finalproject.zip`.