2023/24

Relatório – Bases de Dados Avançadas

POPULAR MOVIES DATASETS

GROUP 20:

LARA ÂNGELO, 56945

MARIA JERÓNIMO, 56887

TIAGO SILVA, 59446

Selected Kaggle Dataset:

https://www.kaggle.com/datasets/whenamancodes/popular-movies-datasets-58000-movies?select=ratings.csv

Description of the dataset:

This dataset, named "ml-latest," captures user interactions with the MovieLens movie recommendation service. It comprises 27,753,444 ratings and 1,108,997 tags given by 283,228 randomly selected users for 58,098 movies. The data spans from January 9, 1995, to September 26, 2018, and the dataset itself was created on the latter date.
Each user is identified by a unique ID, and no additional information about the users is provided.

Goal 1: Scheme for both databases

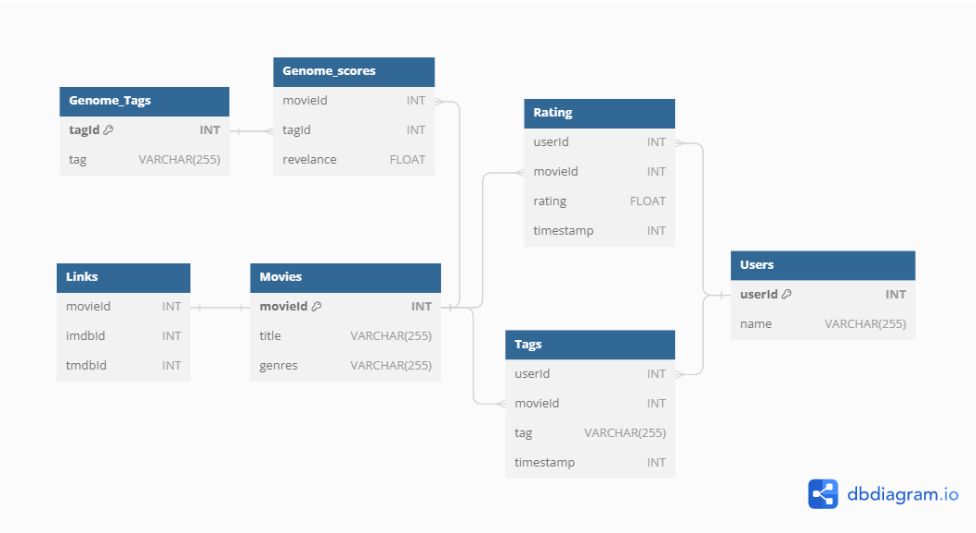

Figure 1: MySQL Schema

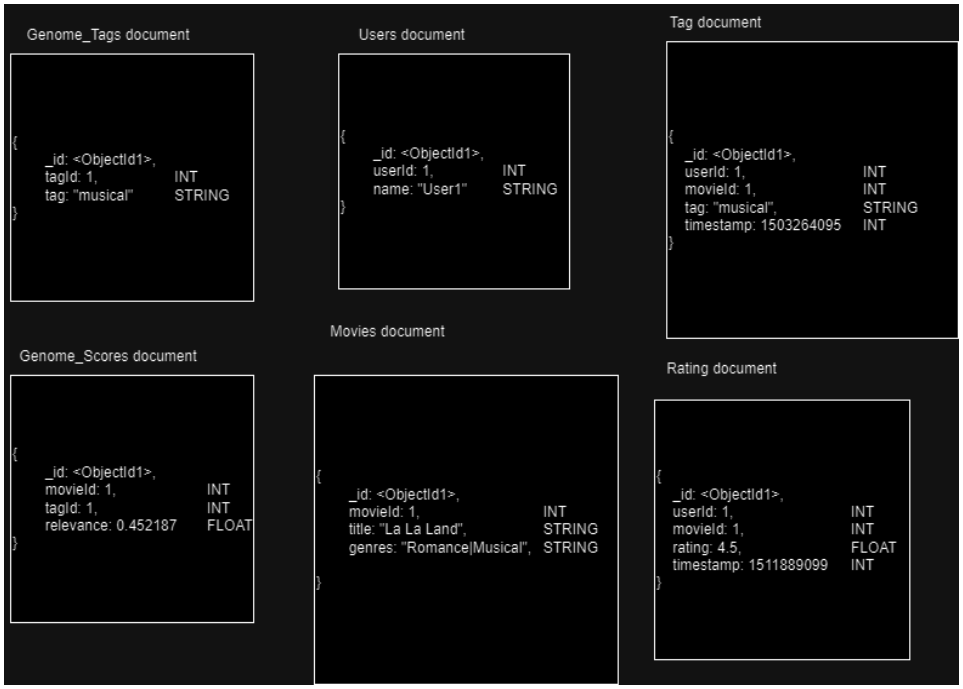

Figure 2: MongoDB Schema

Goal 2: Create the databases

To achieve the second goal we created two databases, one using MySQL and another using MongoDB following the structure defined previously.
For MySQL we created four (4) tables: “Rating”; “Movies”; "Tags”; “Users” with the following schemas.

Figure 3: MySQL tables

For MongoDB we created 5 collections: “Rating”; “Movies”; "Tags”; “Users”; “Counters”. The fields present in these collections are the same as the columns for mySQL, with only the addition of the “_id” field that is automatically created by MongoDB.
We also created the collection “Counters” to auxiliary in the process of inserting new data for Goal 3.

Goal 3: Queries

We created six queries for each database:

- Two simple queries:
  - Retrieve all movies that start with the letter 'A' and have the genre “Sci-Fi”. (262 columns)
  - Fetch the most popular genre. (Drama)
- Two complex queries:
  - Query to find out which movies a user has rated.
  - Query to determine the tags that a user has used for movies.
- One insert
  - We inserted a new movie with 3 ratings and 3 tags associated.
- One update
  - We updated one of the ratings previously inserted.

	movieId	title	genres
0	680	Alphaville (Alphaville, une étrange aventure d...	Drama|Mystery|Romance|Sci-Fi|Thriller
1	748	Arrival, The (1996)	Action|Sci-Fi|Thriller
2	1127	Abyss, The (1989)	Action|Adventure|Sci-Fi|Thriller
3	1200	Aliens (1986)	Action|Adventure|Horror|Sci-Fi
4	1214	Alien (1979)	Horror|Sci-Fi
..	...	...	...
257	193455	Aniara (2018)	Sci-Fi
258	193575	Another Time (2018)	Comedy|Drama|Romance|Sci-Fi
259	193683	Alien Code (2017)	Mystery|Sci-Fi|Thriller
260	193763	Assassination Classroom: Graduation (2016)	Action|Comedy|Sci-Fi
261	193777	Almost Home (2014)	Adventure|Animation|Sci-Fi

[262 rows x 3 columns]

Figure 4: Example of DataFrame output for Query 1

Since we used the movieId field instead of the _id field created by MongoDB, we created the collection Counters (Figure 5). This collection allows us to save the last inserted movieId, so that when in case of a new insertion the movieId for the movie is correctly incremented.

Figure 5: Document from Counters collection

Goal 4: Indexing and Optimization

In this section we will dive into the comparative analysis of the performance from each query previously referred. In which in the initial version no indexes were utilized. We used the Explain statement/ method to understand how the database query will be executed. In the relational database, MySQL, no changes to the relational schema were applied due to the already positive performance improvements demonstrated using indexes.

Query 1: Retrieve all movies that start with the letter 'A' and have the genre “Sci-Fi”.

Infrastructure	Without Optimization (s)	With Optimization (s)
MySQL	0.1233578	0.0200107
MongoDB	0.2348795	0.2134218

Table 1: Query 1 performance

Initial Queries:

MySQL:
- The initial MySQL query used a LIKE clause on the 'genres' column and 'title' column for filtering movies.
- The EXPLAIN statement indicated a full table scan (58191: The estimated number of rows to be examined).

```
Query Execution Plan:
(1, 'SIMPLE', 'm', None, 'ALL', None, None, None, None, 58191, 1.23, 'Using where')
```

Figure 6: EXPALIN statement without indexes

MongoDB:
- The initial MongoDB query used regular expressions for case-insensitive matching on 'title' and 'genres' fields.
- The explain method provided details about keys and documents examined: 'totalDocsExamined': 58101.

Optimized Queries:

MySQL:
- An index (idx_title_genres) was created on the 'title' and 'genres' columns to optimize the query.
- The EXPLAIN statement indicated the use of the index and shows a lower estimated number of rows to be examined (6608).

```
Query Execution Plan:
(1, 'SIMPLE', 'm', None, 'range', 'idx_title_genres', 'idx_title_genres', '1023', None, 6608, 11.11, 'Using where; Using index')
```

Figure 7: EXPLAIN statement with index

MongoDB:
- An index on 'title' and 'genres' was created for MongoDB as well. A projection was also used to fetch only the necessary fields, resulting in further optimization.
- The MongoDB query that used the index performed worse that the initial one. (0.378597 s)
- The Projection created bettered the performance in an almost insignificant way. (Table 1)

Query 2: Fetch the most popular genre.

Infrastructure	Without Optimization (s)	With Optimization (s)
MySQL	0.2074178	0.0975887
MongoDB	0.1620786	0.1545634

Table 2: Query 2 performance

Initial Queries:

MySQL:
- The initial MySQL query used a GROUP BY clause on the 'genres' column to count movies in each genre and sorted the results to find the most popular genre.
- The EXPLAIN statement indicated a full table scan (58191: The estimated number of rows to be examined).

MongoDB:
- The initial MongoDB query used an aggregation pipeline to group by 'genres' and calculate the count, sorted in descending order, and limited to one result.
- The explain method was used to analyze the aggregation pipeline and showed that no indexes were being used.

Optimized Queries:

MySQL:
- An index (idx_genres) was created on the 'genres' column to optimize the query.
- The EXPLAIN statement indicated the use of the index although the number of rows to be examined remained the same.

MongoDB:
- An index on 'genres' was created for MongoDB as well.
- The explain method indicated that the indexes were not being utilized. Indicating that this query wasn't correctly optimized.

Query 3: Query to find out which movies a user has rated.

Infrastructure	Without Optimization (s)	With Optimization (s)
MySQL	2.7164535	0.0040225
MongoDB	2.4192252	0.6316092

Table 3: Query 3 performance

Initial Queries:

MySQL:
- The initial MySQL query joined the 'Rating,' 'Movies,' and 'Users' tables to fetch movie details rated by a specific user.
- The EXPLAIN statement indicated this for the Rating table: 10.0: The estimated percentage of the table that will be scanned; This for the User and Movie tables: 4: The number of rows to be examined using this join.

MongoDB:
- The initial MongoDB query used a simple find operation to retrieve rated movies for a specific user.
- The explain method showed - 'totalDocsExamined': 1000017.

Optimized Queries:

MySQL:
- A composite index (idx_user_movie) on ('userId', 'movieId') was created to optimize the query.
- The EXPLAIN statement indicated the use of the index.

MongoDB:
- An index on ('userId', 'movieId') was created for MongoDB as well. A projection was also used to fetch only the necessary fields, resulting in further optimization.
- The explain method indicated that the index and projection were being utilized and that showed a reduction in the number of documents examined ('totalDocsExamined': 17).

Query 4: Query to determine the tags that a user has used for movies.

Infrastructure	Without Optimization (s)	With Optimization (s)
MySQL	6.1342172	0.0098533
MongoDB	5.3375334	3.6900792

Table 4: Query 4 performance

Initial Queries:

MySQL:
- The initial MySQL query joined the 'Users,' 'Rating,' 'Movies,' and 'Tags' tables to fetch movie details with tags for a specific user.
- The EXPLAIN statement indicated this for the Ranting table and Tag table: 10.0: The percentage of how many rows were filtered; This for the Movie table: 1: The number of rows to be examined; This for the Users table: 4: The number of rows to be examined.

MongoDB:
- The initial MongoDB query used a find operation on the 'Tags' collection to retrieve tags for movies rated by a specific user.
- The explain method showed - 'totalDocsExamined': 1000009.

Optimized Queries:

MySQL:
- Composite indexes (idx_user_movieR and idx_user_movieT) on ('userId', 'movieId') for the 'Rating' and 'Tags' tables were created to optimize the query.
- The EXPLAIN statement indicated the use of the index. For table Tag: 13: The number of rows to be examined (using an index condition). And, for table Rating: 10: The number of rows to be examined.

MongoDB:
- An index on ('userId', 'movieId') was created for MongoDB as well.
- The explain method indicated that the index was applied leading to a reduction in the number of documents examined ('totalDocsExamined': 13)

Insert

Infrastructure	Without Optimization (s)	With Optimization (s)
MySQL	0.0607988	0.0306825
MongoDB	0.2169995	0.1590101

Table 5: Insert performance

Initial Queries:

MySQL:
- The initial MySQL code inserted a new movie into the 'Movies' table, followed by ratings and tags for that movie in separate transactions.

MongoDB:
- A movie, ratings and tags for the new movie were inserted into their respective collections.

Optimized Queries:

MySQL:
- The optimized MySQL code consolidated the movie insertion, ratings insertion, and tags insertion into a single transaction.

MongoDB:
- The optimized MongoDB code combined the insertion of ratings and tags for the new movie into a single loop for better efficiency.

Update

We couldn’t find a way to improve the performance of our update queries.

Unsuccessful Test and Schema Modification Attempt

During our project development, we encountered a significant challenge when attempting to optimize the MongoDB schema. Taking the professor’s advice, we attempted to consolidate all the collections into one, anticipating potential performance improvements. To achieve that, we merged individual MongoDB collections ("Rating", "Movies", "Tags" and "Users") into a single, unified collection. We observed the following:

- The process of generating the new unified collection proved to be time-consuming, taking several hours to complete.
- As expected, the unified collection turned out to be considerably large. This resulted in a collection with a substantial amount of repeated information.
- Despite our intentions to enhance performance, the outcomes were less than favorable. The initial query test (Query 1) performed on the consolidated collection showed a significant degradation in performance (135.514 s), contrary to our expectations.

While our intentions were to optimize the MongoDB schema and improve query performance, this attempt proved unsuccessful. The trade-off between data redundancy and the complexity of managing a large, consolidated collection ultimately led to deteriorated performance in the initial query tests.

Project Replication Guide: Setting Up Databases and Executing Queries

To replicate our project, follow this step-by-step guide. Our project is encapsulated in a single file named "Project.py," meticulously organized to create tables, insert data, perform queries, and analyze their performance. The code is extensively commented for clarity, with sections corresponding to project checkpoints and goals clearly identified.

Step 1: Locate the database connection section. Uncomment and modify the connection details (e.g., host, username, password) according to your database setup.

Step 2: Uncomment the section responsible for creating tables and collections in MySQL and MongoDB.

Step 3: Uncomment the code for inserting data into both MySQL and MongoDB.

Step 4: Find the query execution section. Uncomment the specific queries you wish to run for both MySQL and MongoDB.

Step 5: Uncomment the section dedicated to performance analysis. Observe the printed explanations and the time that each query took in both MySQL and MongoDB.