Database Project-Phase II

SOEN 363 Section S
Data Systems for Software Engineers

Presented to
Ali Jannatpour

Prepared by
Christa Abou-Arraje (40226631)
Despina Koulisakis (40190212)
Sarah Abellard (40184667)
Nour Hassoun (40233077)

Department of Computer Science & Software Engineering
Concordia University

Montreal, Canada

December 1st, 2024

Phase II

Overview

The migration from MySQL to MongoDB marked a significant shift in how data is structured, accessed, and scaled in our system. The MySQL database previously served as the backbone for storing relational data. However, this phase required us to transition to MongoDB, which is a NoSQL database. This phase focuses on implementing and fine-tuning this migration while preserving the system's integrity and introducing new data layers.

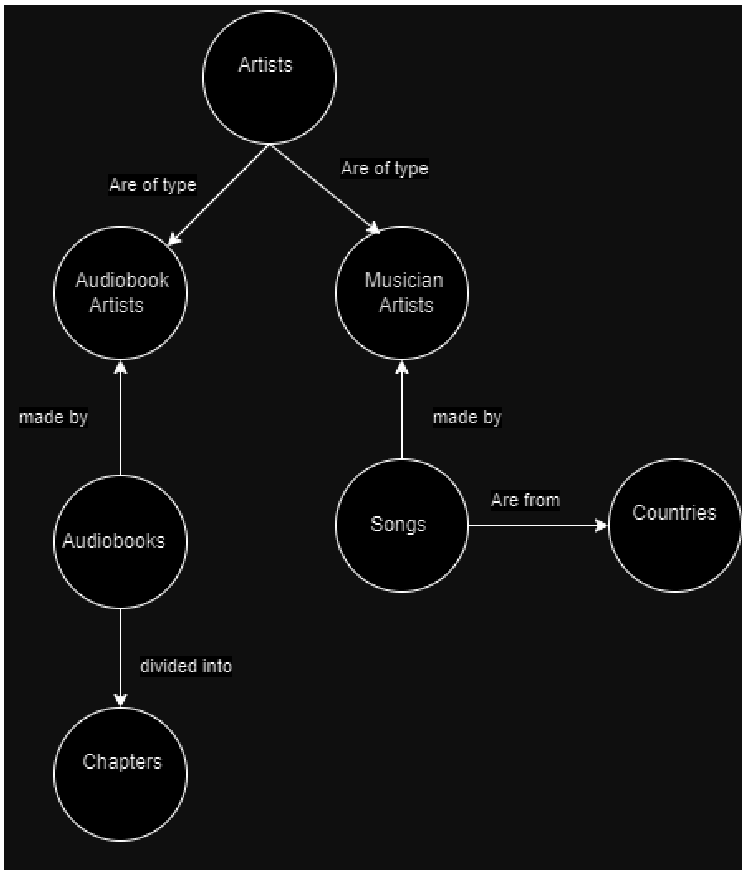
Data Integration

The integration began by using Python scripts to extract data from MySQL tables and populate corresponding collections in MongoDB. Each MySQL table such as Artists, Songs, Audiobooks was transformed into a MongoDB collection. Collections in MongoDB inherently allow embedding and referencing, which was utilized for entities like Chapters, which remain dependent on Audiobooks. Additionally, as suggested during a POD, audio blobs for the songs were added to simulate realistic database sizes.

In MongoDB, the Chapters collection was migrated while ensuring proper references to Audiobooks were maintained. To address binary data stored in the audio_blob field of the Songs table, transformations were applied to store the data as hexadecimal strings compatible with BSON.

Additionally, text indexing was introduced in the Audiobooks collection on the description field to enable efficient text searches. This indexing facilitates querying based on keywords within audiobook descriptions.

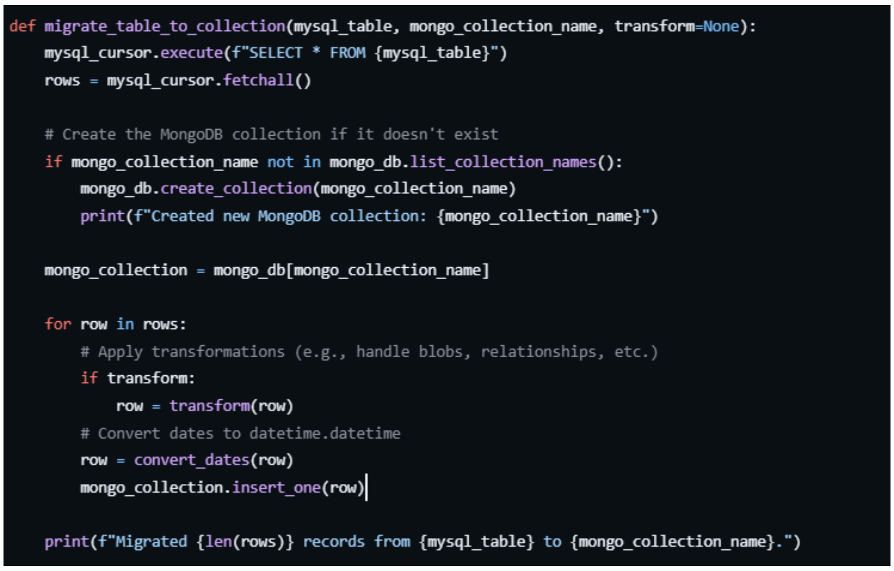
Data Model



Approach

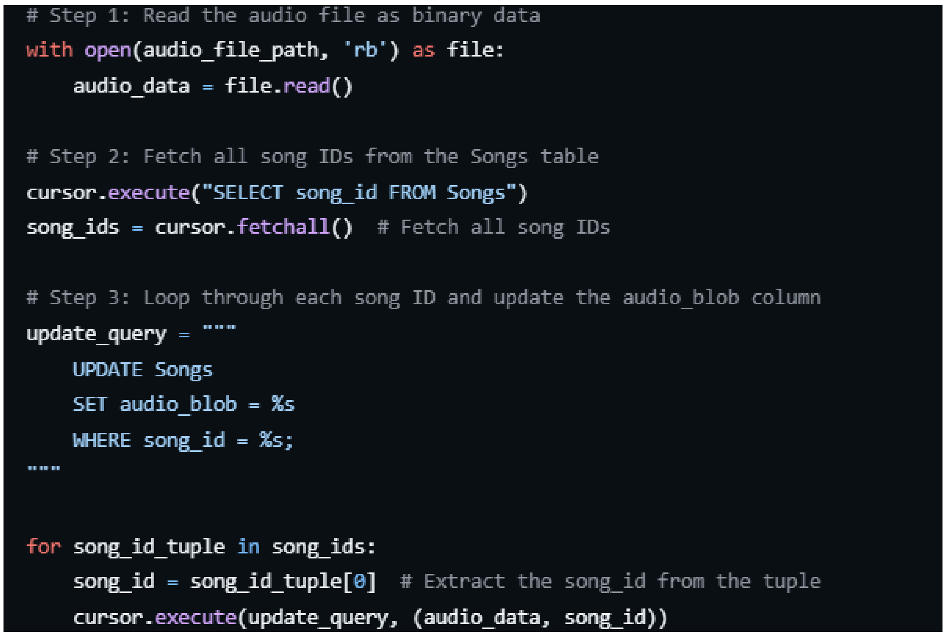
The migration was performed in a series of steps:

1. **Data Extraction**: MySQL data was queried using Python's pymysql library.
2. **Transformation**: Data was processed to match MongoDB's document structure. Binary blobs were converted to hexadecimal, and datetime.date objects were converted to datetime.datetime for BSON compatibility.



```python
def migrate_table_to_collection(mysql_table, mongo_collection_name, transform=None):
    mysql_cursor.execute(f"SELECT * FROM {mysql_table}")
    rows = mysql_cursor.fetchall()

    # Create the MongoDB collection if it doesn't exist
    if mongo_collection_name not in mongo_db.list_collection_names():
        mongo_db.create_collection(mongo_collection_name)
        print(f"Created new MongoDB collection: {mongo_collection_name}")

    mongo_collection = mongo_db[mongo_collection_name]

    for row in rows:
        # Apply transformations (e.g., handle blobs, relationships, etc.)
        if transform:
            row = transform(row)
        # Convert dates to datetime.datetime
        row = convert_dates(row)
        mongo_collection.insert_one(row)

    print(f"Migrated {len(rows)} records from {mysql_table} to {mongo_collection_name}.")
```

3. **Data Insertion**: Using pymongo, the data was inserted into MongoDB collections, creating new collections dynamically as needed.



```python
# Step 1: Read the audio file as binary data
with open(audio_file_path, 'rb') as file:
    audio_data = file.read()

# Step 2: Fetch all song IDs from the Songs table
cursor.execute("SELECT song_id FROM Songs")
song_ids = cursor.fetchall()  # Fetch all song IDs

# Step 3: Loop through each song ID and update the audio_blob column
update_query = """
    UPDATE Songs
    SET audio_blob = %s
    WHERE song_id = %s;
"""

for song_id_tuple in song_ids:
    song_id = song_id_tuple[0]  # Extract the song_id from the tuple
    cursor.execute(update_query, (audio_data, song_id))
```

Challenges

1. Binary Data Migration

The audio_blob field from MySQL required transformation into hexadecimal format to be compatible with BSON. This added a preprocessing step to ensure successful insertion without errors.

2. Datetime Compatibility

MongoDB expects datetime.datetime objects, while MySQL stores dates as datetime.date. A recursive transformation function was implemented to resolve this incompatibility during migration.

3. Schema Flexibility

MongoDB's flexible schema posed challenges in ensuring relational consistency. Relationships such as Audiobooks and their dependent Chapters required careful referencing and validation to prevent orphaned documents.

4. SSL Configuration for MongoDB Atlas

Initial connectivity issues arose due to SSL certificate verification errors. These were resolved by explicitly configuring the *certifi* library to handle certificates securely.

5. Dynamic Network Access

Since MongoDB Atlas enforces IP whitelisting, ensuring the correct IP configuration for development environments was a crucial step.

github link: https://github.com/christa-ux/Soen363-Project

(please check project-2-queries branch)

Phase I

Overview

The link between the two data sources is established through the Artists table. All songs fetched from the last.fm API (which include details such as song titles, song IDs, artist IDs and chart_rank) are placed in the Songs table. Then those song records' artist IDs were then linked and added to the Artists table which contains artists and their details (artist IDs name and more). This established a connection between the last.fm API which fetched songs (including their artists) and the spotify get artists API used to populate the Artists table.

The Artists table demonstrates an IS-A relationship with two sub-entities: MusicianArtists and AudiobookArtists. The Artists table serves as a parent entity and it contains general attributes for all artists such as their name. The MusicianArtists and AudiobookArtists tables are child entities that inherit from Artists and have specific attributes. The MusicianArtists table adds attributes like popularity and followers. The AudiobookArtists table contains attributes like biography and total books

In our schema, the Chapters table is a weak entity dependent on Audiobooks. This is because Chapters can't exist independently and they must belong to an audiobook. To do so, it uses a composite primary key: chapter_id + audiobook_id. This ensures that each chapter is uniquely identified within its parent audiobook. If an audiobook is deleted, all related chapters are also removed due to the ON DELETE CASCADE rule.

We implemented a complex referential integrity mechanism using a trigger in MySQL to make sure that no artist exceeds 10 songs in the database. We created a before-insert trigger on the Songs table. The trigger is fired each time a new song is about to be inserted. For example, if there is an attempt to insert 11 songs for an artist, it will fail due to the trigger, as only 10 songs per artist are allowed. If an artist already has 10 songs, any further attempts to insert songs for the artist will be blocked and the error message will be returned. By creating this trigger, we prevent any data integrity issues related to oversaturation of songs for a particular artist.

We have a view for users with full access to the Songs table, allowing them to see all key columns: song_id, title, atrist_id, and chart_rank. This view is designed for users with high-level privileges such as administrators or analysts who would need comprehensive information about all songs. We also have a view that limits data visibility for users with restricted access, so we can ensure they only see essential information. It focuses on showing only the title and chart_rank of songs that are highly ranked. It ensures that only songs with a chart_rank of 50 or better are accessible, filtering out less popular or less relevant songs. This view is suitable for general users that don’t need to access full song information.

Data Model

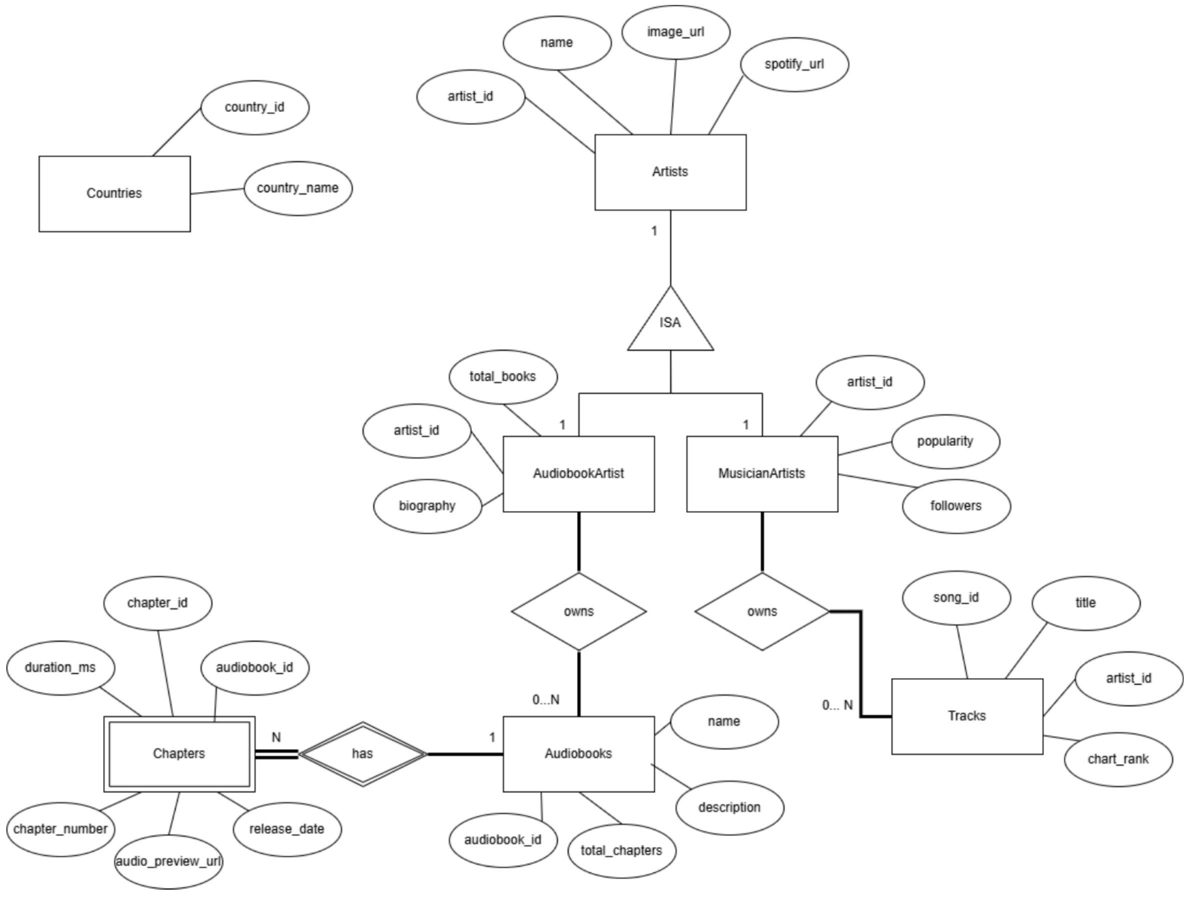


Figure 1: Entity Relationship Model for Works Database

Figure 1: Entity Relationship Model for Works Database

Approach/Challenges faced in populating the data

When collecting data for our work tables, we faced a few challenges. First, we had to find a free API that could meet all of our requirements. Many free APIs had request limits, which were insufficient for our need to fetch large amounts of data to populate our tables. Also, some APIs provided song data, but without relevant keys like country or genre, making it difficult to link the data to our Artists table or other entities in our model. We had to adjust our timeline to accommodate the delay of finding suitable APIs, and in the end, we were able to find two that met our needs. Second, since our database required data from at least two distinct sources, we encountered some challenges when it came to extracting and formatting the data from these APIs to ensure consistency. It was important to take these differences into consideration when populating the database. Third, we faced another challenge when working with the last.fm API that sorted songs by country. The issue here was that this API did not provide a predefined list of countries, and it didn't include an easy way to dynamically fetch the country data required to associate songs with specific regions. To address this, we had to hardcode a Country table in our database so we can properly fetch the data. In summary, although we faced some considerable challenges, we were able to address them properly and in due time.

Sources of API's used:

https://developer.spotify.com/documentation/web-api/ https://www.last.fm/api
https://www.last.fm/api