****

**Computer Science and Technologies**

**NoSQL Databases**

**(Sem. 6, 2023/2024)**

**Project:**

**Migrating from MariaDB to MongoDB**

**Students: Mentor:**

1. Marija Dejanoska MSc Benjamin Besimi
2. Romana Stojceska

June 2024, Tetovo

# Introduction

In the context of this project, the primary objective is to explore and demonstrate the process of transitioning from a traditional relational database model to a modern NoSQL database. This migration aims to leverage the strengths of MongoDB, such as its ability to handle unstructured data, scalability, and flexibility in data modeling.

By undertaking this project, we aim to provide a practical example of how data can be effectively migrated and transformed to meet the demands of applications that require high performance and flexibility in data storage and retrieval. Additionally, this project will serve as a comprehensive guide for understanding the challenges and solutions involved in data migration, including schema design, data integrity, and the use of appropriate tools and technologies for seamless transition.

Contents

[Relational Database Design and Data Modeling 3](#_Toc169168947)

[Database Schema Overview 3](#_Toc169168948)

[Table Descriptions and Relationships 3](#_Toc169168949)

[Choice of Data Types and Constraints 4](#_Toc169168950)

[Data Population 4](#_Toc169168951)

[Screenshots of Populated Data 4](#_Toc169168952)

[Relationships 7](#_Toc169168953)

[Choice of NoSQL Database 7](#_Toc169168954)

[Justification for MongoDB 7](#_Toc169168955)

[Comparison with Other NoSQL Databases 8](#_Toc169168956)

[NoSQL Database Modeling 8](#_Toc169168957)

[Schema and Structure 8](#_Toc169168958)

[Mapping Relational Concepts to NoSQL 10](#_Toc169168959)

[Data Migration Process 10](#_Toc169168960)

[Migration Code 10](#_Toc169168961)

[Challenges and Resolutions 11](#_Toc169168962)

[Conclusion 12](#_Toc169168963)

# Relational Database Design and Data Modeling

## Database Schema Overview

The relational database in MariaDB is designed with the following four tables:

### Table Descriptions and Relationships

1. Users

* **Columns**: user\_id (PK), user\_name, user\_email.
* **Description**: Stores user information.
* **Relationships**: One-to-Many relationship with Playlists (user\_id in Playlists references user\_id in Users).

1. Songs

* **Columns**: song\_id (PK), song\_title, artist\_id (FK), album\_id (FK), song\_genre, song\_length, song\_release\_date.
* **Description**: Stores song information.
* **Relationships**: Many-to-Many relationship with Playlists through Contains (song\_id in Contains references song\_id in Songs).

1. Playlists

* **Columns**: playlist\_id (PK), user\_id (FK), playlist\_title.
* **Description**: Stores playlist information.
* **Relationships**: Many-to-One relationship with Users (user\_id references user\_id in Users), Many-to-Many relationship with Songs through Contains (playlist\_id in Contains references playlist\_id in Playlists).

1. **Contains (Playlist Songs)**

* **Columns**: playlist\_id (FK), song\_id (FK).
* **Description**: Junction table to handle many-to-many relationship between Playlists and Songs.

1. Artist

* **Columns**: artist\_id (PK), artist\_name, artist\_genre.
* **Description**: Stores information about music artists.
* **Relationships**: One-to-Many relationship with Songs (artist in Songs references artist\_id in Artists).

1. Albums

* **Columns**: album\_id (PK), album\_title, artist\_id (FK), album\_genre, album\_release\_date.
* **Description**: Stores details about music albums.
* **Relationships**: Many-to-One relationship with Artists (artist\_id references artist\_id in Artists). One-to-Many relationship with Songs (album\_id in Songs references album\_id in Albums).

### Choice of Data Types and Constraints

1. Primary **Keys**: Used for unique identification of records.
2. Foreign **Keys**: Ensure referential integrity between tables.

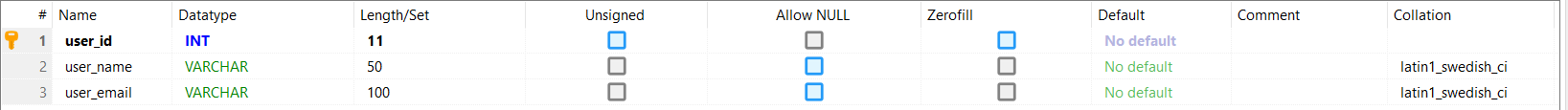
Data **Types**: Chosen based on the nature of the data:

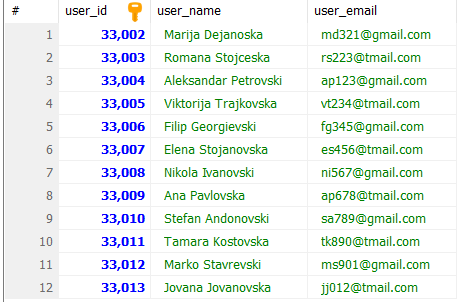
* VARCHAR for strings
* INT for integers
* DATE for date values

# Data Population

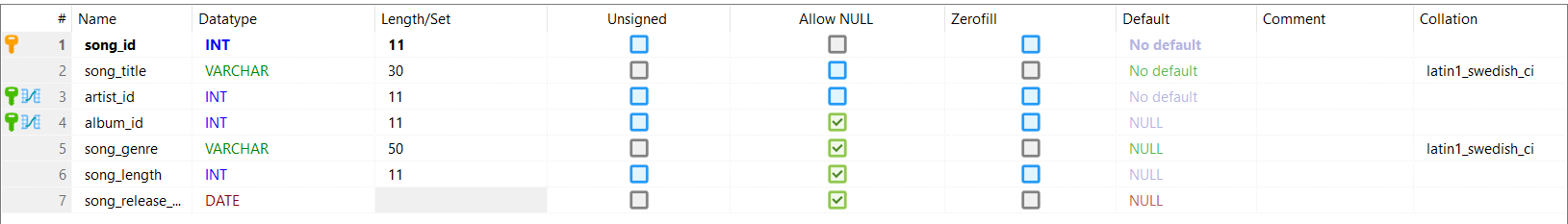
## Screenshots of Populated Data

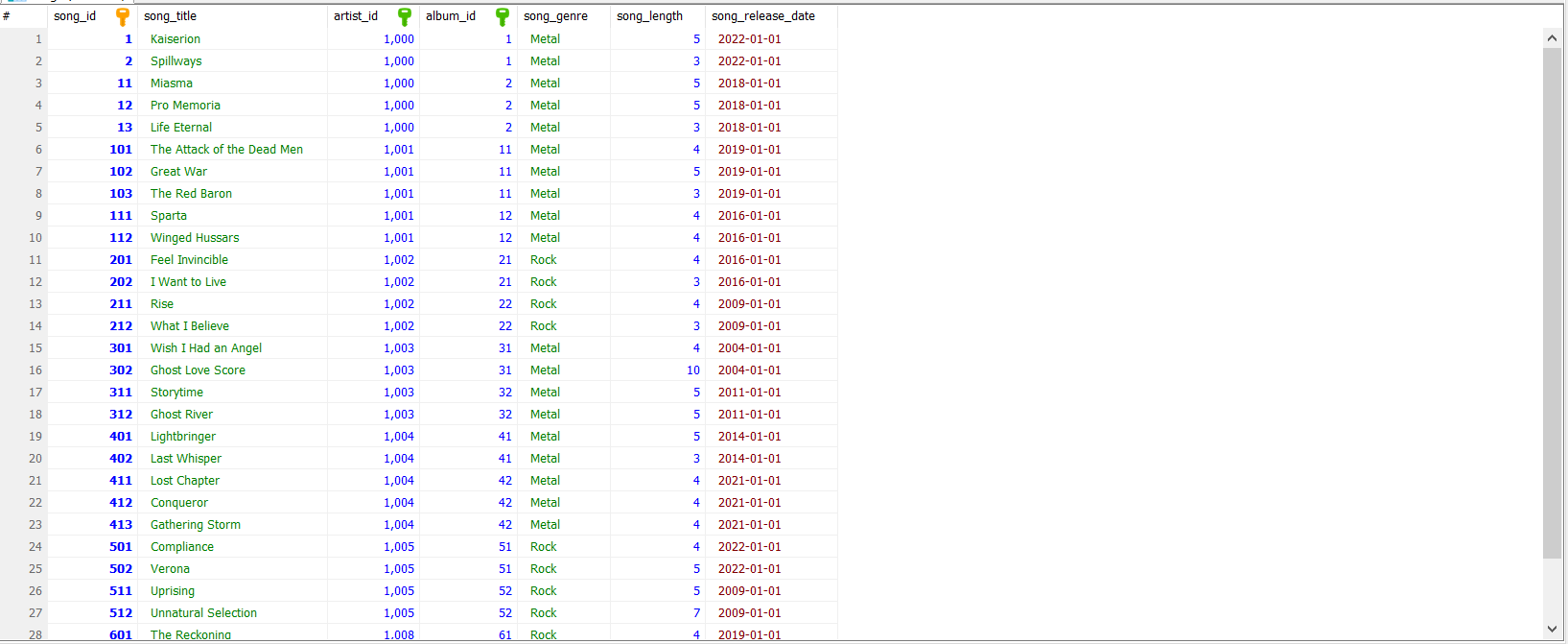
1. **Users**





1. **Songs**





1. **Playlists**



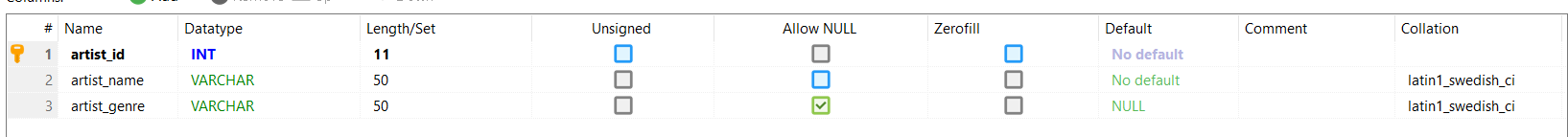


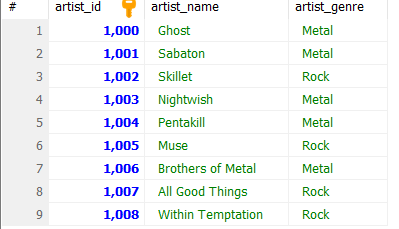
1. **Contains**



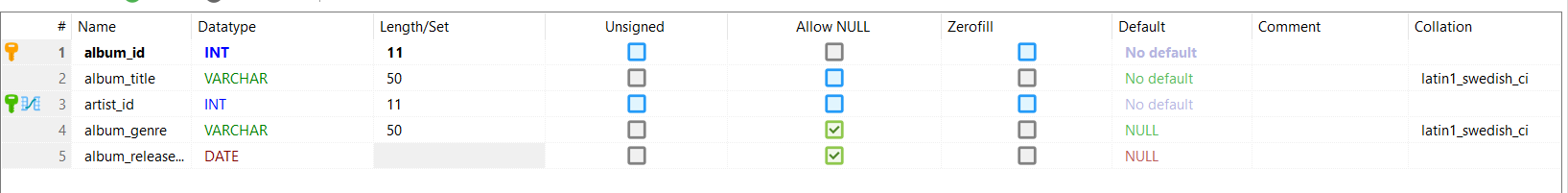


1. **Artists**



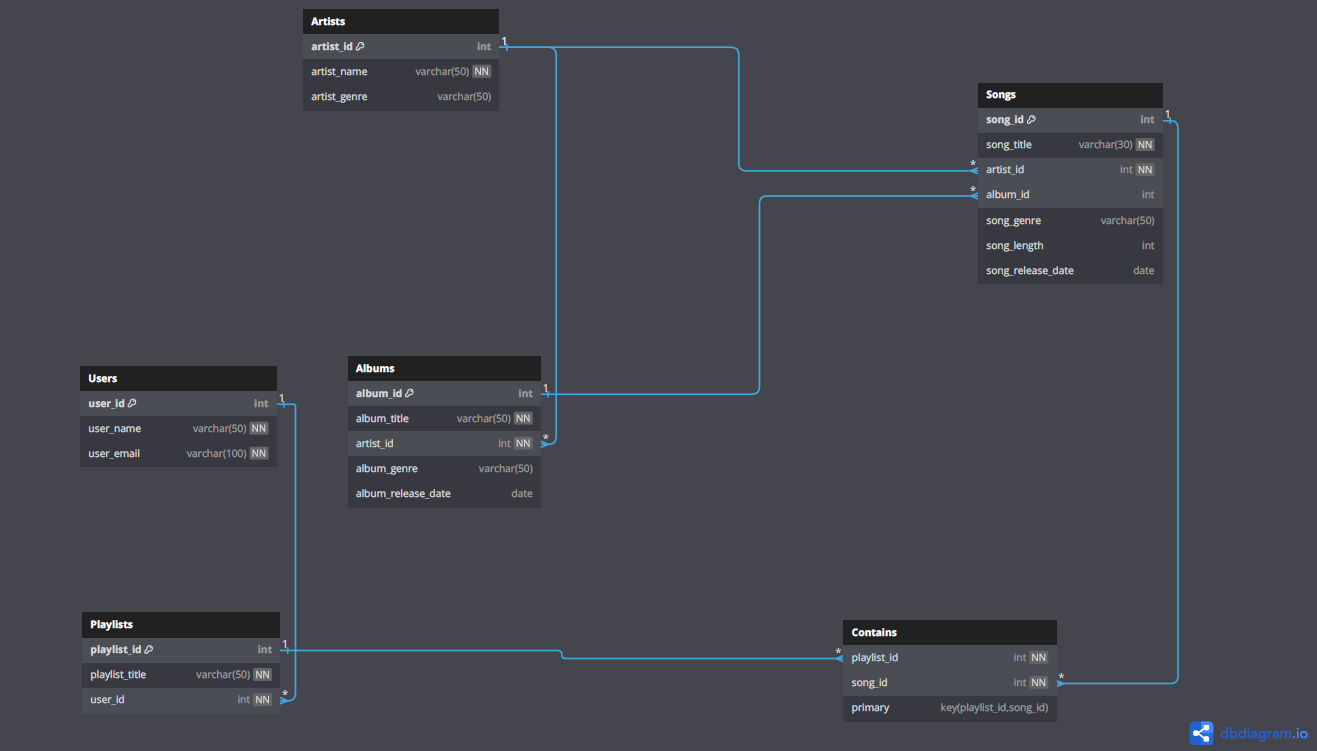


1. **Albums**





## Relationships



# Choice of NoSQL Database

## Justification for MongoDB

After evaluating several NoSQL databases, we chose MongoDB due to its flexible document-oriented data model, scalability, and performance benefits. Here are the key reasons for our choice:

1. **Flexible Data Model**: MongoDB's document-oriented model allows us to store complex nested data structures in a single document, which is suitable for our use case.
2. **Scalability**: MongoDB provides horizontal scaling, which is essential for handling large datasets and high-traffic applications.
3. **Performance**: MongoDB's indexing and querying capabilities are optimized for performance, making it efficient for retrieving and manipulating data.
4. **Ease of Use**: MongoDB offers a simple setup process and has extensive community support, making it easier to troubleshoot and find resources.

## Comparison with Other NoSQL Databases

|  |  |  |  |
| --- | --- | --- | --- |
| NoSQL Database | Redis | Cassandra | Neo4j |
| Pros | High performance for in-memory data storage, simple key-value store, suitable for caching and real-time analytics. | Highly scalable and available, suitable for large datasets, provides a distributed and fault-tolerant system. | Excellent for graph-based data models, efficient querying of relationships, suitable for use cases like social networks and recommendation engines. |
| Cons | Limited querying capabilities, not ideal for complex data structures. | Complex setup and maintenance, eventual consistency model can be challenging to manage. | Not suitable for document-oriented data, less mature community support compared to MongoDB, can be complex for non-graph use cases. |

# NoSQL Database Modeling

## Schema and Structure

In MongoDB, the data is modeled using collections and embedded documents. Here is the schema for our collections:

* 1. **User Collection**



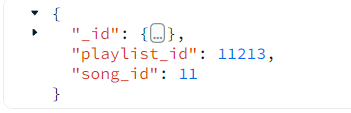
* 1. **Songs Collection**



* 1. **Playlists Collection**



* 1. **Contains Collection**



* 1. **Artists Collection**



* 1. **Albums Collection**



## Mapping Relational Concepts to NoSQL

1. **One-to-Many Relationships:** In MongoDB, these relationships are represented by embedding references (ObjectIDs) in the documents.
2. **Many-to-Many Relationships:** Managed through arrays of ObjectIDs referencing related documents. For example, the Playlists collection contains an array of song IDs.

# Data Migration Process

## Migration Code

The following Python script uses PyMongo to migrate data from MariaDB to MongoDB:

import pymysql  
from pymongo import MongoClient  
from datetime import datetime, date  
  
# Function to convert date fields to ISO format  
def convert\_dates(row):  
 for key, value in row.items():  
 if isinstance(value, date):  
 row[key] = value.isoformat()  
 return row  
  
# Establishing MySQL connection  
conn = pymysql.connect(  
 host='127.0.0.1',  
 user='root',  
 password='admin',  
 database='musicstream'  
)  
  
# Establishing MongoDB connection  
client = MongoClient('mongodb://localhost:27017/')  
db = client['musicstream']  
  
# Function to transfer data from MySQL to MongoDB  
def transfer\_table(mysql\_table, mongo\_collection):  
 cursor = conn.cursor(pymysql.cursors.DictCursor)  
 cursor.execute(f"SELECT \* FROM {mysql\_table}")  
 rows = cursor.fetchall()  
 if rows:  
 converted\_rows = [convert\_dates(row) for row in rows]  
 try:  
 # Clear existing data in MongoDB collection  
 db[mongo\_collection].delete\_many({})  
 # Insert new data  
 db[mongo\_collection].insert\_many(converted\_rows)  
 print(f"Data from MySQL table '{mysql\_table}' transferred to MongoDB collection '{mongo\_collection}'")  
 except Exception as e:  
 print(f"Error transferring data from MySQL table '{mysql\_table}' to MongoDB collection '{mongo\_collection}': {e}")  
  
# List of tables to transfer  
tables = [  
 ('users', 'users'),  
 ('artists', 'artists'),  
 ('albums', 'albums'),  
 ('songs', 'songs'),  
 ('playlists', 'playlists'),  
 ('contains', 'contains')  
]  
  
# Transfer data for each table  
for mysql\_table, mongo\_collection in tables:  
 transfer\_table(mysql\_table, mongo\_collection)  
  
print("Data transfer completed!")

## Challenges and Resolutions

**Data Transformation:**

* **Challenge:** Mapping relational data from MySQL to MongoDB's document-oriented model.
* **Resolution:** Designed a data conversion function (convert\_dates) to ensure date fields were correctly formatted to ISO format. The convert\_dates function ensures that date fields retrieved from MySQL are formatted correctly using isoformat() before insertion into MongoDB. This function was crucial in maintaining data consistency across both databases.

**Error Handling:**

* **Challenge:** Ensuring data integrity and managing potential errors during the migration process.
* **Resolution:** Implemented robust error handling within the script (transfer\_table function) to catch exceptions that may arise during data extraction, transformation, and insertion into MongoDB

**Performance:**

* **Challenge:** Efficiently handling the migration of large datasets to MongoDB.
* **Resolution:** To optimize performance, the script adopted batch insertion (insert\_many) for transferring data to MongoDB. This approach minimizes the number of network round-trips and enhances throughput when migrating large volumes of data. Additionally, MongoDB indexing was employed strategically to expedite query execution and improve overall data migration efficiency.

# Conclusion

This project successfully migrated data from a MariaDB relational database to MongoDB, showcasing MongoDB's suitability for managing semi-structured data prevalent in music streaming applications. Through meticulous database design and thoughtful data modeling, we ensured a smooth transition while preserving data integrity. The migration process was meticulously documented, emphasizing the significance of flexible schema design in NoSQL databases and the importance of robust error handling and performance optimization techniques.

This project has underscored the value of adaptability and scalability in database technologies, highlighting the benefits and challenges of transitioning from a relational to a NoSQL paradigm. Moving forward, the insights gained will guide future database migrations and reinforce best practices in data management and integration.