Department of Mathematics
and Computer Science

Faculty of Data Analysis

Database Mod NoSQL
Report
Analysis of database performance in music library

Students:
Ishen Imerov
Aidana Babir

Professors:
Armando Ruggeri, Antonio Celesti

Academic Year 2022/2023

Table of Contents

INTRODUCTION……3
Project overview…….3

IMPLEMENTATION……..3
Custom Data Generation for Music Library Management System……3
MySQL……...6
MongoDB……..10
RedisDB……14
Neo4jDB……..…17
CassandraDB……..18

CONCLUSION…….22

INTRODUCTION

Project Overview

Efficient management and organization of data are integral to the smooth functioning of various systems and applications. The emergence of NoSQL databases has broadened the database landscape beyond traditional SQL databases, offering a plethora of options for data storage and retrieval. This project focuses on the development of a music library management system and evaluates the performance of five distinct database types: SQL, Cassandra, MongoDB, RedisDB, and Neo4j. The primary goal is to assess the capabilities and efficiency of these databases by executing queries of varying complexities on datasets of different sizes. Through this comparative analysis, we aim to discern the strengths and weaknesses of each database type within the context of a music library management system. This evaluation will provide valuable insights into the suitability of each database for specific use cases, facilitating informed decisions in future database selection processes.

To ensure a comprehensive evaluation, datasets ranging from 250,000 to 1 million records are utilized, enabling observation of database behavior and performance under varying data loads. The assessment encompasses not only simple queries but also more intricate ones, replicating real-world scenarios encountered in music library management systems.

Throughout the project, diligent monitoring and measurement of performance metrics such as query execution time, resource utilization, and scalability are conducted. These metrics offer a clear understanding of each database's efficiency in managing large datasets and handling complex queries. Additionally, factors like data integrity, usability, and adaptability are considered during the design and implementation phases of the music library management system.

This report presents the outcomes and analysis derived from our comparative study, shedding light on the strengths and limitations of each database type concerning their performance and applicability to a music library management system. The insights garnered from this research endeavor will serve as valuable guidance for developers, system architects, and database administrators in making informed decisions when selecting a suitable NoSQL database for similar applications.

Description of the database management system

The database management system (DBMS) for our music library relies on Python and MySQL to efficiently organize, retrieve, and manage music-related data. It comprises four main tables: Songs, Albums, Artists, and Genres.

The Songs table stores details about individual songs, including title, duration, release date, genre, artist, album, and lyrics. The Albums table contains information about albums, such as album name and release date. The Artists table stores data regarding artists, including artist name, biography, and origin. The Genres table maintains details about music genres, consisting of genre names. Primary keys (song_id, album_id, artist_id, genre_id) ensure uniqueness and facilitate efficient record retrieval.

Data import and insertion are handled by a Python script connecting to the MySQL database and inserting data from CSV files into the respective tables. Each CSV file contains

information about songs, albums, artists, and genres, which are iteratively processed and inserted into the corresponding database tables. Data insertion is performed using SQL INSERT queries, ensuring accurate representation of each record. Error handling is implemented to manage exceptions during data insertion.
The MySQL-based DBMS offers flexibility and scalability to adapt to evolving requirements. New tables or attributes can be added, and existing ones modified, without significant disruption to the system. This DBMS provides a robust solution for efficiently managing a music library. By structuring data into tables and leveraging MySQL's capabilities, the system supports comprehensive storage, retrieval, and management of music-related information.
Overall, the implemented database management system facilitates effective organization and maintenance of music library data, ensuring optimal performance and scalability.

Problem of database management
In overseeing a music library system that encompasses songs, albums, artists, and genres, grappling with the intricacies of database management presents a formidable task.

Data Consistency and Integrity:
Ensuring consistency and integrity across varied database types poses a significant hurdle. Each database, whether it's SQL or NoSQL, operates with its distinct data model and query language. The utilization of different databases for various components of the music library system may result in disparities in data representation and storage.

For instance, when synchronizing data between SQL databases, known for their structured schema and predefined relationships, and NoSQL databases like MongoDB or Cassandra, each with its unique data model and query language, discrepancies may emerge. Updates applied to data in one database might not promptly propagate to others, potentially introducing inconsistencies in the information related to songs, albums, artists, or genres.

Query Performance:
The performance characteristics and query capabilities of diverse database types exhibit significant variance. A query that executes efficiently in one database type may exhibit subpar performance or necessitate intricate transformations in another. Such discrepancies in query performance can profoundly impact the responsiveness and user experience of the music library system.

Expertise in Multiple Database Technologies:
Proficiency in the intricacies of each database type, encompassing their data models, query languages, and optimization techniques, is essential for developers and administrators. Managing multiple database technologies demands substantial dedication to maintain competence across different platforms. This expertise is vital for ensuring smooth development, troubleshooting, and maintenance of the music library system.

Addressing these challenges entails the implementation of robust data synchronization mechanisms, effective data migration strategies, and standardized data access layers. By meticulously considering factors such as data consistency, query performance, and the expertise required for managing diverse databases, organizations can adeptly navigate the complexities inherent in database management for music library systems.

Implementation

In our pursuit of developing and thoroughly testing our music library management system, we encountered a notable challenge: the scarcity of datasets that matched our system's structure and size requirements. Existing resources failed to provide datasets of the exact sizes we needed—ranging from 250,000 to 1 million records—and often didn't align with the intricacies of our system.

Instead of expanding efforts in searching for appropriate datasets, we made a strategic decision to craft our own dataset. This approach allowed us to sidestep the constraints posed by existing resources and empowered us to tailor the data generation process to our system's specifications. By creating our dataset, we ensured precise control over the data generation, guaranteeing accuracy and consistency in the datasets.

Furthermore, we recognized that the conventional databases for music libraries often include numerous relationships that weren't all relevant to our testing requirements. To streamline our testing process and simplify our data model, we opted to design our database with fewer relations, ensuring that it accurately reflected the essential aspects of our music library management system.

Our custom data generation solution not only fulfilled our need for datasets of varying sizes but also facilitated comprehensive performance testing of our database systems. Leveraging Python for data generation provided us with the flexibility needed to simulate real-world data and usage patterns inherent in music libraries. The provided Python code efficiently generates synthetic data for different dataset sizes, populating CSV files with records for songs, albums, artists, and genres. This approach supports scalable and flexible performance testing of our music library management system, enabling us to evaluate its performance under different loads and scenarios accurately.

```
 import os
import csv
from faker import Faker
import random
from datetime import datetime, timedelta
import uuid

fake = Faker()
predefined_genres = [
    "Pop",
    "Rock",
    "Hip hop",
    "Jazz",
```

```
    "Blues",
    "Country",
    "Electronic",
    "R&B (Rhythm and Blues)",
    "Reggae",
    "Classical",
    "Folk",
    "Indie",
    "Metal",
    "Punk",
    "Soul",
    "Funk",
    "Alternative",
    "Dance",
    "Gospel",
    "Latin"
]

# Function to generate a unique ID
def generate_uuid():
    return str(uuid.uuid4())

# Function to generate fake data for Song entity
def generate_song(song_id, artist_id, album_id, genre_id):
    song_title = fake.catch_phrase()
    duration = random.randint(120, 600)  # Duration in seconds, between 2 to 10 minutes
    release_date = fake.date_between(start_date='-10y', end_date='today')
    lyrics = fake.paragraph(nb_sentences=10)
    return [song_id, song_title, duration, genre_id, release_date, artist_id, album_id, lyrics]

# Function to generate fake data for Album entity
def generate_album(album_id):
    album_name = fake.catch_phrase()
    release_date = fake.date_between(start_date='-10y', end_date='today')
    return [album_id, album_name, release_date]

# Function to generate fake data for Artist entity
def generate_artist(artist_id):
    artist_name = fake.name()
    biography = fake.paragraph(nb_sentences=5)
    origin = fake.country()
    return [artist_id, artist_name, biography, origin]

# Function to generate fake data for Genre entity
def generate_genre(genre_id, index):
    genre_name = predefined_genres[index]
    return [genre_id, genre_name]

# Define the number of records
num_records = 1000000

# Generate unique IDs for albums, artists, and genres
album_ids = [generate_uuid() for _ in range(num_records)]
artist_ids = [generate_uuid() for _ in range(num_records)]
genre_ids = [generate_uuid() for _ in range(len(predefined_genres))]

# Generate fake data for albums, artists, and genres
```

```
albums = [generate_album(album_id) for album_id in album_ids]
artists = [generate_artist(artist_id) for artist_id in artist_ids]
genres = [generate_genre(genre_id, index) for index, genre_id in enumerate(genre_ids)]

# Generate fake data for songs, associating them with specific artists and albums
songs = []
for song_id in range(num_records):
    # Randomly select an album ID and artist ID from the available lists
    album_id = random.choice(album_ids)
    artist_id = random.choice(artist_ids)
    genre_id = random.choice(genre_ids)
    songs.append(generate_song(song_id, artist_id, album_id, genre_id))

# Create a folder named "csv" if it doesn't exist
folder_name = "csv"
if not os.path.exists(folder_name):
    os.makedirs(folder_name)

# Write data to CSV files in the "csv" folder
with open(os.path.join(folder_name, 'songs.csv'), 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerow(['song_id', 'song_title', 'duration', 'genre_id', 'release_date', 'artist_id', 'album_id', 'lyrics'])
    writer.writerows(songs)

with open(os.path.join(folder_name, 'albums.csv'), 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerow(['album_id', 'album_name', 'release_date'])
    writer.writerows(albums)

with open(os.path.join(folder_name, 'artists.csv'), 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerow(['artist_id', 'artist_name', 'biography', 'origin'])
    writer.writerows(artists)

with open(os.path.join(folder_name, 'genres.csv'), 'w', newline='', encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerow(['genre_id', 'genre_name'])
    writer.writerows(genres)
```

MySQL

```
import csv
import pymysql
import uuid

# Connect to MySQL database
conn = pymysql.connect(host='localhost', user='root', password='', db='music_library')
cursor = conn.cursor()
print("Connected to MySQL database!")

# Function to drop tables if they exist
def drop_tables():
    cursor.execute("DROP TABLE IF EXISTS songs")
    cursor.execute("DROP TABLE IF EXISTS albums")
    cursor.execute("DROP TABLE IF EXISTS artists")
```

```
    cursor.execute("DROP TABLE IF EXISTS genres")

# Drop tables if they exist
drop_tables()
# Create temporary database
cursor.execute("CREATE DATABASE IF NOT EXISTS music_library")
cursor.execute("USE music_library")

# Create songs table
cursor.execute('''CREATE TABLE IF NOT EXISTS songs (
                    song_id VARCHAR(36) PRIMARY KEY,
                    song_title VARCHAR(255),
                    duration INT,
                    release_date DATE,
                    artist_id VARCHAR(36),
                    album_id VARCHAR(36),
                    genre_id VARCHAR(36),
                    lyrics TEXT
                )''')

# Create albums table
cursor.execute('''CREATE TABLE IF NOT EXISTS albums (
                    album_id VARCHAR(36) PRIMARY KEY,
                    album_name VARCHAR(255),
                    release_date DATE
                )''')

# Create artists table
cursor.execute('''CREATE TABLE IF NOT EXISTS artists (
                    artist_id VARCHAR(36) PRIMARY KEY,
                    artist_name VARCHAR(255),
                    biography TEXT,
                    origin VARCHAR(255)
                )''')

# Create genres table
cursor.execute('''CREATE TABLE IF NOT EXISTS genres (
                    genre_id VARCHAR(36) PRIMARY KEY,
                    genre_name VARCHAR(255)
                )''')

# Insert data from songs.csv
with open('csv/songs.csv', 'r') as file:
    csv_data = csv.reader(file)
    next(csv_data)  # Skip header row
    for row in csv_data:
        song_id, song_title, duration, release_date, genre_id, artist_id, album_id, lyrics = row
        query = f"INSERT INTO songs (song_id, song_title, duration, genre_id, release_date, artist_id, album_id,
lyrics) VALUES (%s, %s, %s, %s, %s, %s, %s, %s)"
        cursor.execute(query, (song_id, song_title, int(duration), release_date, genre_id, artist_id, album_id,
lyrics))
print('Data inserted into songs table!')

# Insert data from albums.csv
with open('csv/albums.csv', 'r') as file:
    csv_data = csv.reader(file)
    next(csv_data)  # Skip header row
    for row in csv_data:
        album_id, album_name, release_date = row
        query = f"INSERT INTO albums (album_id, album_name, release_date) VALUES (%s, %s, %s)"
        cursor.execute(query, (album_id, album_name, release_date))
print('Data inserted into albums table!')

# Insert data from artists.csv
with open('csv/artists.csv', 'r') as file:
    csv_data = csv.reader(file)
```

```
    next(csv_data)  # Skip header row
    for row in csv_data:
        artist_id, artist_name, biography, origin = row
        query = f"INSERT INTO artists (artist_id, artist_name, biography, origin) VALUES (%s, %s, %s, %s)"
        cursor.execute(query, (artist_id, artist_name, biography, origin))
print('Data inserted into artists table!')

# Insert data from genres.csv
with open('csv/genres.csv', 'r') as file:
    csv_data = csv.reader(file)
    next(csv_data)  # Skip header row
    for row in csv_data:
        genre_id, genre_name = row
        query = f"INSERT INTO genres (genre_id, genre_name) VALUES (%s, %s)"
        cursor.execute(query, (genre_id, genre_name))
print('Data inserted into genres table!')

# Commit the changes and close the connection
conn.commit()
print('Changes committed!')
conn.close()
print('Connection closed!')
```

Therefore, four queries are built, each increasing in complexity, in the following manner:

```
SQL:

Simple:
    SELECT * FROM songs;

Intermediate:
    SELECT song_title, artist_id, release_date
    FROM songs
    WHERE release_date LIKE '2021%';

Complex:
    SELECT s.song_title, a.artist_name, b.album_name
    FROM songs s
    JOIN artists a ON s.artist_id = a.artist_id
    JOIN albums b ON s.album_id = b.album_id
    WHERE YEAR(s.release_date) = 2023;


Advanced:
    SELECT s.song_title AS song_title, a.artist_name AS artist_name, s.release_date
    FROM songs s
    JOIN artists a ON s.artist_id = a.artist_id
    WHERE YEAR(s.release_date) = 2022
    GROUP BY s.song_title, a.artist_name, s.release_date
    HAVING COUNT(*) > (
        SELECT AVG(songs_per_artist)
```

```
    FROM (
        SELECT COUNT(*) AS songs_per_artist
        FROM songs
        WHERE YEAR(release_date) = 2022
        GROUP BY artist_id
    ) AS subquery
)
ORDER BY s.release_date DESC;
```

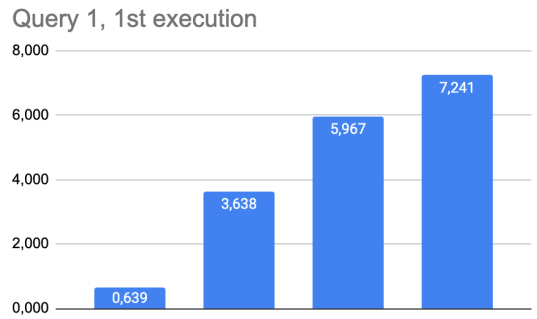
Simple query: This query, "SELECT * FROM songs;", retrieves all the data from the "songs" table. It provides a complete list of all songs in the database, including all their attributes and fields. This is useful when a comprehensive overview of the entire song database is required.

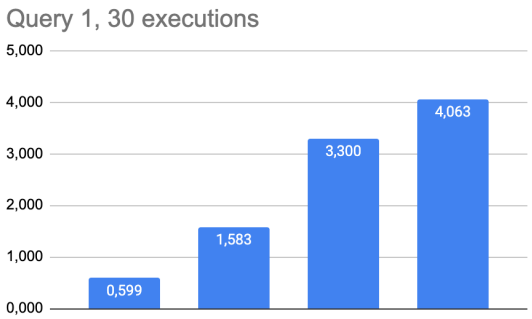
Intermediate: The intermediate query, "SELECT song_title, artist_id, release_date FROM songs WHERE release_date LIKE '2021%';", focuses on retrieving specific attributes from the "songs" table. It selects the song title, artist ID, and release date. The query filters the results to include only songs that have a release date in the year 2021. This query is helpful when looking for songs released in a particular year.

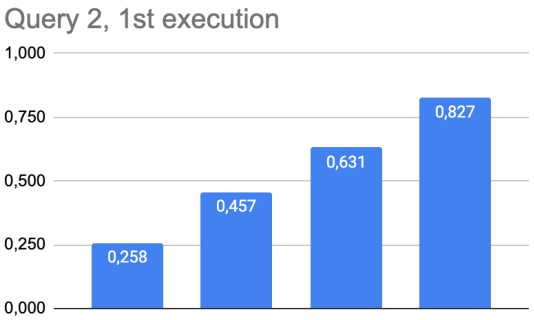
Complex: The complex query, "SELECT s.song_title, a.artist_name, b.album_name FROM songs s JOIN artists a ON s.artist_id = a.artist_id JOIN albums b ON s.album_id = b.album_id WHERE YEAR(s.release_date) = 2023;", retrieves information about songs along with their associated artists and albums. It joins the "songs," "artists," and "albums" tables based on their respective IDs and filters the results to include only songs released in the year 2023. This query provides insights into the songs released in a specific year along with details about the artists and albums.

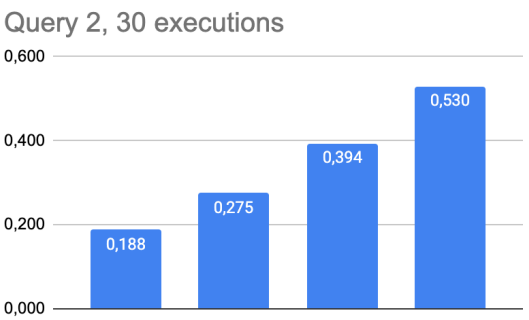
Advanced: The advanced query is complex and involves additional conditions and calculations. It selects the song title, artist name, and release date from the "songs" table, joined with the "artists" table. The query filters the results to include only songs released in the year 2022. It then groups the results by song title, artist name, and release date, calculates the count of songs per artist, and compares it with the average number of songs per artist in 2022. The query further narrows down the results to include only songs by artists who released more songs than the average in 2022, sorting them in descending order based on the release date. This query provides insights into the productivity of artists during a specific year.

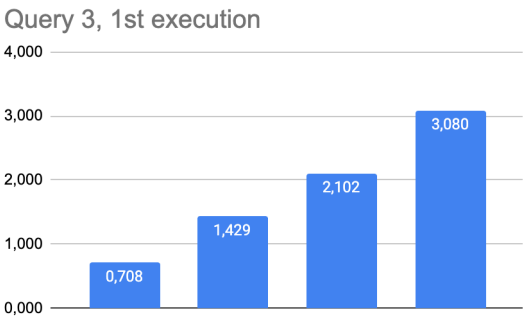
The runtime_checker.py code is designed to record the execution time for each query and each dataset size of the SQL database. It conducts 31 executions per query and per dataset size. The results are then presented in the form of histograms, which provide a visual representation of the execution times. This tool helps analyze the performance of each query under different dataset sizes, aiding in optimizing query performance and database efficiency. The results are presented in the form of histograms:

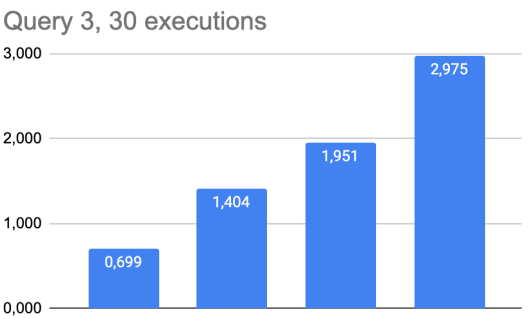


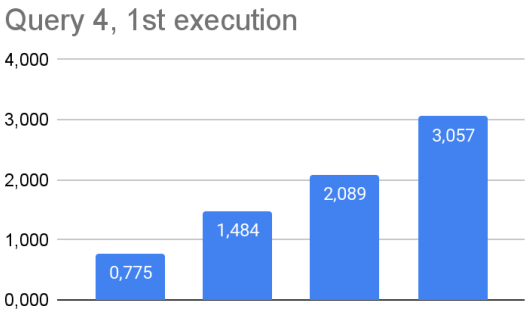


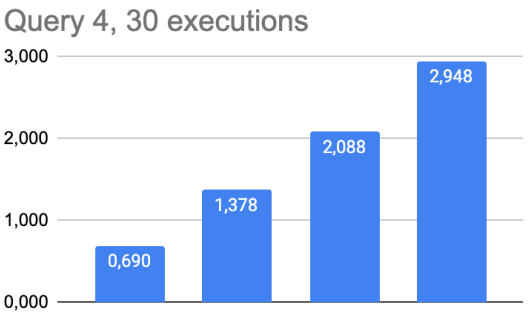












The histograms indicate that as the dataset grows larger, the execution time also increases. Additionally, the complexity of the queries contributes to this increase in execution time.

Mongo DB

For the MongoDB database a Docker container was created and connected to the MongoDB interface called MongoDB Compass.The queries for the MongoDB are written in aggregate function for the sake of using the Compass UI.

Simple Query:
This query retrieves all fields for each document in the "songs" collection without any filtering or transformation.

```
[
  {
    "$project": {
      "_id": 0,
      "song_id": 1,
      "song_title": 1,
      "duration": 1,
      "release_date": 1,
      "artist_id": 1,
      "album_id": 1,
      "genre_id": 1,
      "lyrics": 1
    }
  }
]
```

Intermediate Query:
This query filters songs released in the year 2021 and projects only the song title, artist ID, and release date for each matching document.

```
[
  {
      "$match": {
          "release_date": {"$regex": "^2021"}
    }
  },
  {
    "$project": {
      "_id": 0,
      "song_title": 1,
      "artist_id": 1,
      "release_date": 1
    }
  }
]
```

Complex Query:
This query performs a lookup to join the "songs" collection with the "artists" and "albums" collections based on the artist ID and album ID fields. It then filters songs released in the year 2023 and projects the song title, artist name, and album name for each matching document.

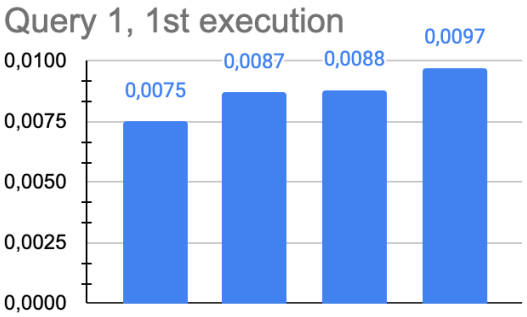
```
[
  {
    "$lookup": {
      "from": "artists",
      "localField": "artist_id",
      "foreignField": "_id",
      "as": "artist"
    }
  },
  {
    "$unwind": "$artist"
  },
  {
    "$lookup": {
      "from": "albums",
      "localField": "album_id",
      "foreignField": "_id",
```

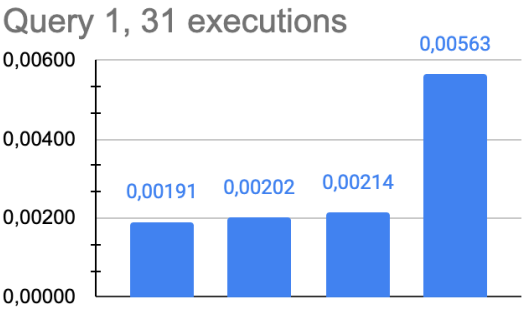
```
        "as": "album"
      }
    },
    {
      "$unwind": "$album"
    },
     {
      "$match": {
        "release_date": {"$regex": "^2023"}
      }
    },
    {
      "$project": {
        "_id": 0,
        "song_title": 1,
        "artist_name": "$artist.artist_name",
        "album_name": "$album.album_name"
      }
    }
]
```

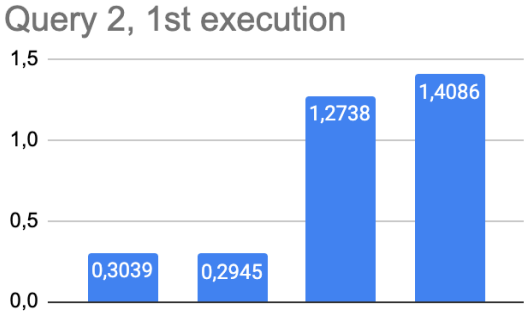
Advanced Query:
This query performs a lookup to join the "songs" collection with the "artists" collection based on the artist ID field. It then filters songs released in the year 2022 and groups them by song title, artist name, and release date. It calculates the count of songs for each group and retains only the groups where the count is greater than the average count of songs per artist. The results are sorted by release date in descending order and projected to include only the song title, artist name, and release date for each document.

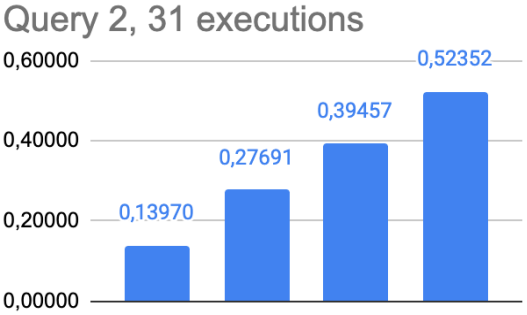
```
[
    {
      "$lookup": {
        "from": "artists",
        "localField": "artist_id",
        "foreignField": "_id",
        "as": "artist"
      }
    },
    {
      "$unwind": "$artist"
    },
    {
      "$match": {
        "release_date": {"$regex": "^2022"}
      }
    },
    {
      "$group": {
        "_id": {
          "song_title": "$song_title",
          "artist_name": "$artist.artist_name",
          "release_date": "$release_date"
        },
        "count": {"$sum": 1}
      }
    },
    {
      "$match": {
        "count": {"$gt": {"$avg": "$count"}}
      }
    },
    {
      "$sort": {"_id.release_date": -1}
    },
    {
      "$project": {
```

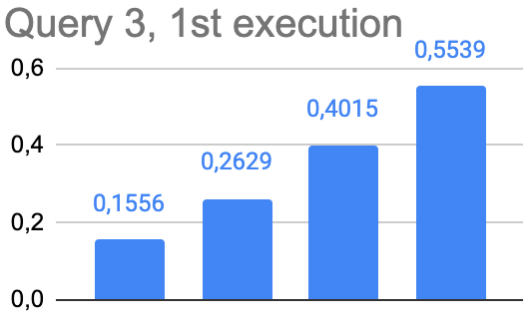
```
            "song_title": "$_id.song_title",
            "artist_name": "$_id.artist_name",
            "release_date": "$_id.release_date",
                "_id": 0
        }
    }
]
```

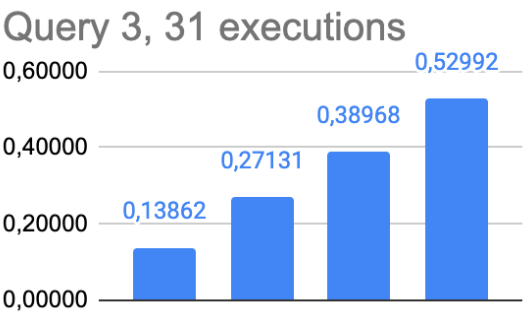


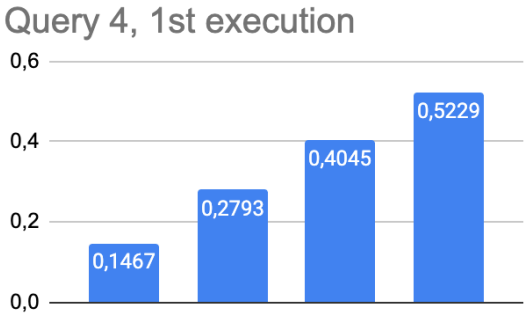


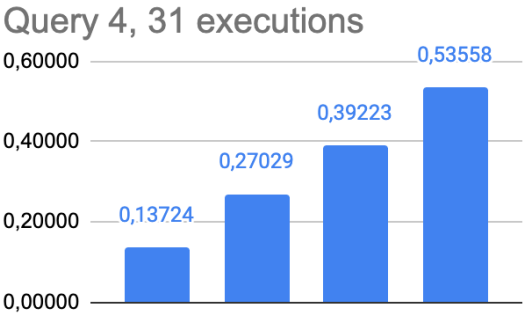












RedisDB

We adopted a similar method for the Redis database, utilizing a Docker container by pulling the image and subsequently establishing connection through the designated port.

Simple:

```
with open(os.path.join(folder_name, "runtime_q1.txt"), "w") as file:
    for j in range(num_executions):
        start_time = time.time()
        # Iterate over all song keys and retrieve their values
        for song_key in r.smembers('songs'):
            song_data = r.hgetall(song_key)
        end_time = time.time()
        runtime = end_time - start_time

            # Write the runtime to the file
        file.write(f"{j+1}. {runtime:.4f} sec\n")


logger.info('Query 1 done!')
```

Intermediate:

```
with open(os.path.join(folder_name, "runtime_q2.txt"), "w") as file:
    for j in range(num_executions):
        start_time = time.time()
        # Retrieve songs with release date starting with '2021'
        for song_key in r.smembers('songs'):
            song_data = r.hgetall(song_key)
            release_date = song_data[b'release_date'].decode('utf-8')
            if release_date.startswith('2021'):
                song_title = song_data[b'song_title'].decode('utf-8')
                artist_id = song_data[b'artist_id'].decode('utf-8')
                release_date = song_data[b'release_date'].decode('utf-8')
        end_time = time.time()
        runtime = end_time - start_time
        # Write the runtime to the file
        file.write(f"{j+1}. {runtime:.4f} sec\n")

logger.info('Query 2 done!')
```

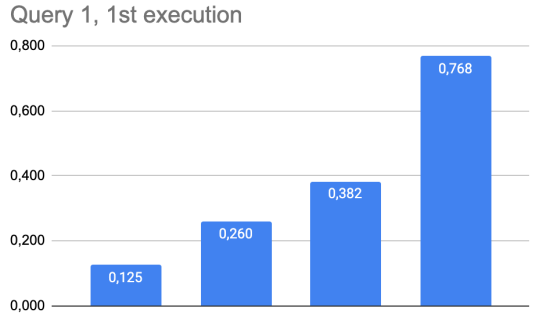
Complex:

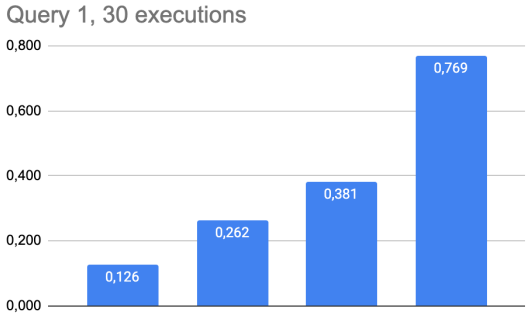
```
with open(os.path.join(folder_name, "runtime_q3.txt"), "w") as file:
    for j in range(num_executions):
        start_time = time.time()
        # Retrieve songs released in 2023 and their associated artist and album information
        for song_key in r.smembers('songs'):
            song_data = r.hgetall(song_key)
            release_date = song_data[b'release_date'].decode('utf-8')
            if release_date.startswith('2023'):
                artist_id = song_data[b'artist_id'].decode('utf-8')
                album_id = song_data[b'album_id'].decode('utf-8')
                artist_name = r.hget(f'artist:{artist_id}', 'artist_name').decode('utf-8')
                album_name = r.hget(f'album:{album_id}', 'album_name').decode('utf-8')
                song_title = song_data[b'song_title'].decode('utf-8')
        end_time = time.time()
        runtime = end_time - start_time
        # Write the runtime to the file
        file.write(f"{j+1}. {runtime:.4f} sec\n")
logger.info('Query 3 done!')
```

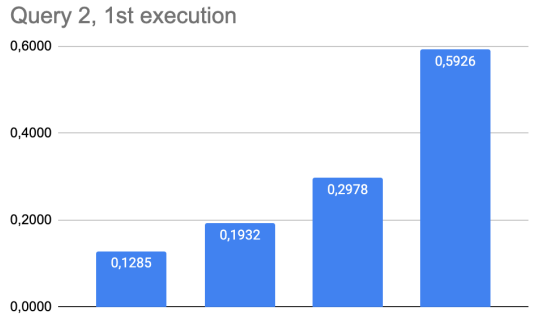
Advanced:

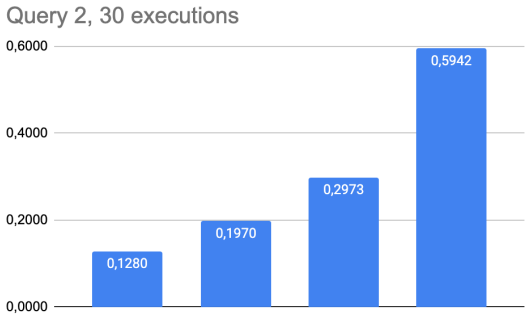
```
with open(os.path.join(folder_name, "runtime_q4.txt"), "w") as file:
    for j in range(num_executions):
        start_time = time.time()
        # Retrieve all songs released in 2021 from Redis
        songs_2021 = []
        for song_key in r.smembers('songs'):
            song_data = r.hgetall(song_key)
            release_date = song_data[b'release_date'].decode('utf-8')
            if release_date.startswith('2021'):
                songs_2021.append(song_data)
```

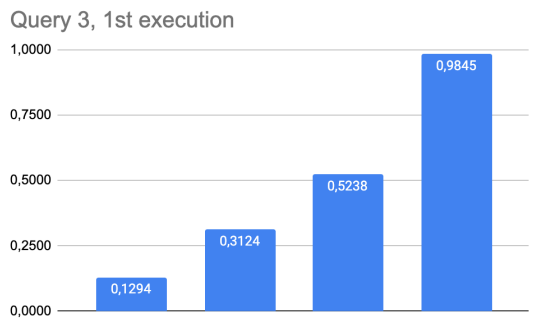
```
    # Perform manual aggregation to calculate the average number of songs per artist
    artist_song_count = {}
    for song in songs_2021:
        artist_id = song[b'artist_id'].decode('utf-8')
        artist_song_count[artist_id] = artist_song_count.get(artist_id, 0) + 1
    avg_songs_per_artist = sum(artist_song_count.values()) / len(artist_song_count)
    # Retrieve songs where the count is greater than the average
    advanced_results = []
    for song in songs_2021:
        artist_id = song[b'artist_id'].decode('utf-8')
        if artist_song_count[artist_id] > avg_songs_per_artist:
            artist_name = r.hget(f'artist:{artist_id}', 'artist_name').decode('utf-8')
            song_title = song[b'song_title'].decode('utf-8')
            release_date = song[b'release_date'].decode('utf-8')
            advanced_results.append({'song_title': song_title, 'artist_name': artist_name, 'release_date': release_date})

    # Sort the results by release date in descending order
    advanced_results.sort(key=lambda x: x['release_date'], reverse=True)
    end_time = time.time()
    runtime = end_time - start_time
    # Write the runtime to the file
    file.write(f"{j+1}. {runtime:.4f} sec\n")

logger.info('Query 4 done!')
```

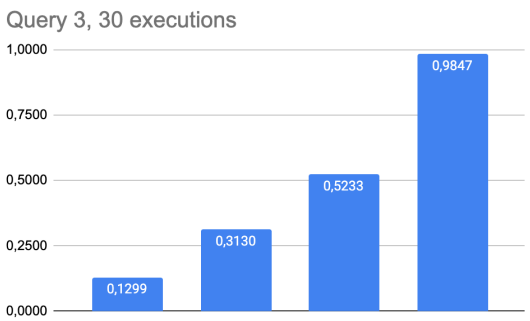


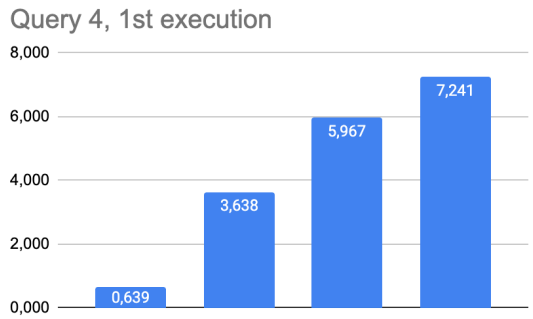


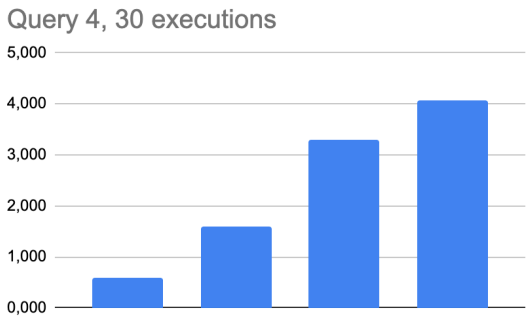








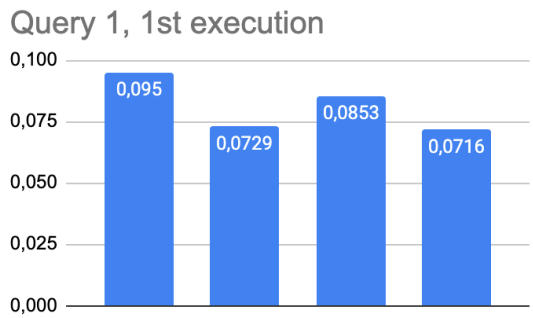


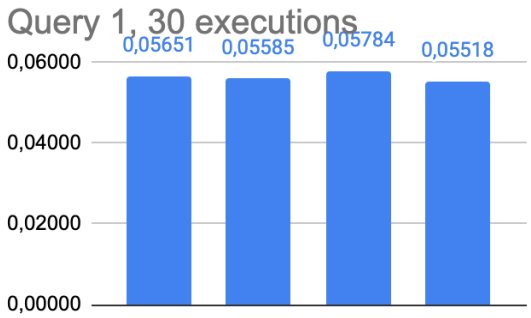


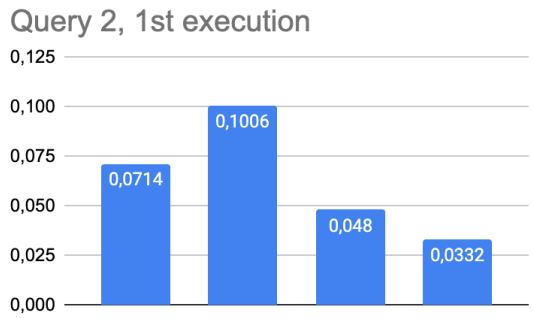
Neo4jDB

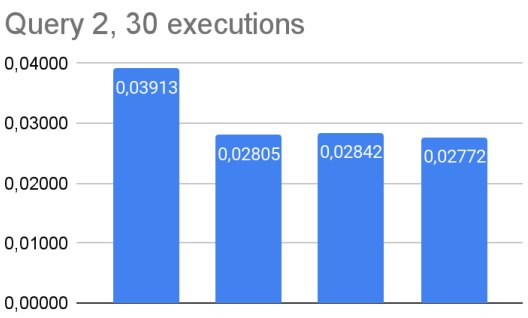
To connect to a Neo4j database using a Docker container, start by pulling the Neo4j image from Docker Hub and running a container based on that image. Map the container ports to the corresponding ports on your host machine to access the Neo4j browser interface via your web browser. Once connected, you can set a password and begin executing queries, exploring Neo4j's functionalities within the containerized environment. This approach offers a convenient and isolated way to work with Neo4j databases.

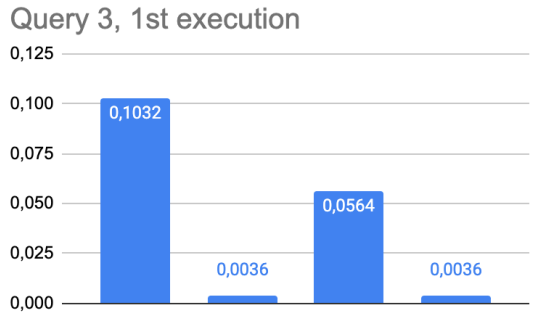
```
// Simple: Retrieves all songs in the database.
MATCH (s:Song)
RETURN s

// Intermediate: Retrieves songs released in 2021, along with their titles, artist IDs, and release dates.
MATCH (s:Song)
WHERE s.release_date STARTS WITH '2021'
RETURN s.song_title, s.artist_id, s.release_date

// Complex: Retrieves songs released in 2023, along with their titles, artist names, and album names.
MATCH (s:Song)-[:BY_ARTIST]->(a:Artist), (s)-[:IN_ALBUM]->(b:Album)
WHERE datetime(s.release_date).year = 2023
RETURN s.song_title, a.artist_name, b.album_name

// Advanced: Retrieves songs released in 2022 by artists who have released more songs than the average number of songs per artist in the database.
MATCH (s:Song)-[:BY_ARTIST]->(a:Artist)
WHERE datetime(s.release_date).year = 2022
WITH s, a
ORDER BY s.release_date DESC
WITH s.song_title AS song_title, a.artist_name AS artist_name, s.release_date AS release_date
WITH song_title, artist_name, release_date, COUNT(*) AS songs_count
WITH song_title, artist_name, release_date, songs_count, avg(songs_count) AS avg_songs_per_artist
WHERE songs_count > avg_songs_per_artist
RETURN song_title, artist_name, release_date
ORDER BY release_date DESC
```

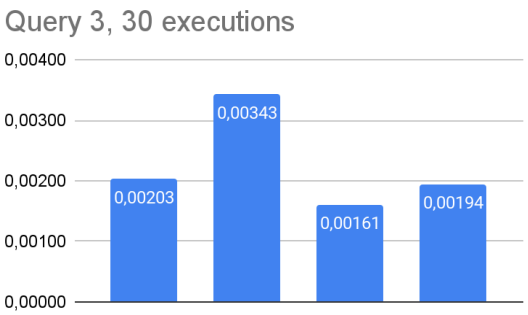


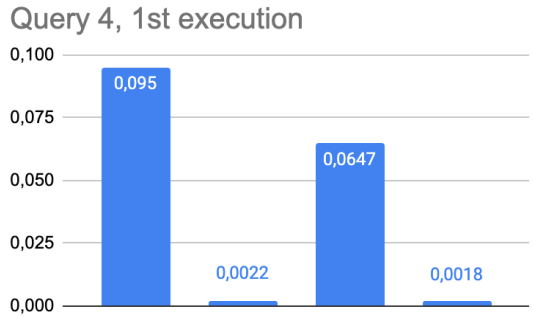


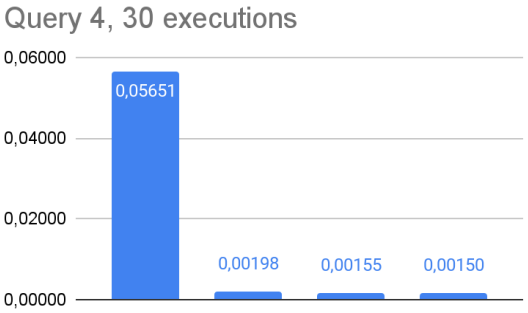












CassandraDB

Query 1:
SELECT * FROM songs;

Query 2:
SELECT song_title, duration, release_date
FROM songs
WHERE release_date >= '2023-01-01' AND release_date < '2024-01-01'
ALLOW FILTERING;

Query 3:

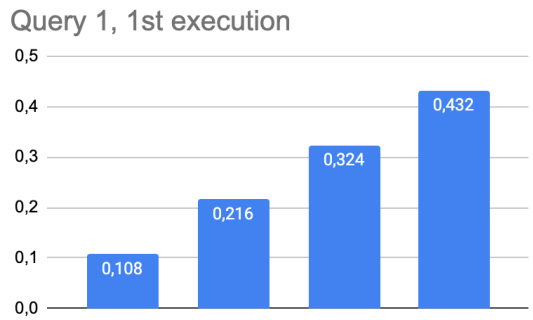
```
import os
import time
from cassandra.cluster import Cluster

# Connect to Cassandra cluster
cluster = Cluster(['localhost'], port=9042)
session = cluster.connect('music_library')

print('Successfully connected to Cassandra.')

# Define the number of executions for the complex query
```

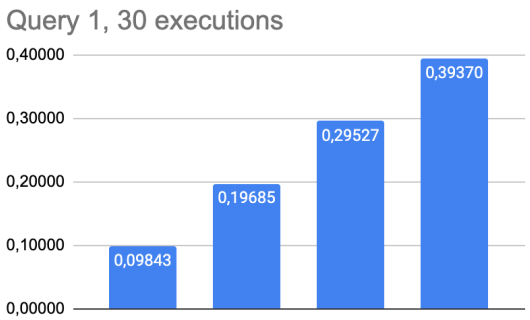
```
num_executions = 31

# Define the folder name
folder_name = os.path.join("cassandra", "run-time-results-25")

# File to store execution times
file_path = os.path.join(folder_name, "runtime_q3.txt")

# Record execution times in the file
with open(file_path, "w") as file:
    for i in range(num_executions):
        start_time = time.time()
        # Create a denormalized table to support the complex query
        create_denormalized_table_query = """
        CREATE TABLE IF NOT EXISTS denormalized_songs (
        song_id INT PRIMARY KEY,
        song_title TEXT,
        artist_name TEXT,
        album_name TEXT,
        duration INT,
        release_date DATE,
        artist_id UUID,
        album_id UUID
        );
        """
        session.execute(create_denormalized_table_query)
        print('Successfully denormalized_songs created')
        # Query songs with duration less than 300 seconds
        select_songs_query = """
        SELECT song_id, song_title, duration, release_date, artist_id, album_id
        FROM songs
        WHERE duration < 300
        ALLOW FILTERING;
        """
        # Execute query to get songs
        songs_result = session.execute(select_songs_query)
        # Iterate over selected songs
        for song in songs_result:
            # Retrieve artist name using artist_id
            artist_query = "SELECT artist_name FROM artists WHERE artist_id = %s;"
            artist_name_row = session.execute(artist_query, [song.artist_id]).one()
            artist_name = artist_name_row.artist_name if artist_name_row else None

            # Retrieve album name using album_id
            album_query = "SELECT album_name FROM albums WHERE album_id = %s;"
            album_name_row = session.execute(album_query, [song.album_id]).one()
            album_name = album_name_row.album_name if album_name_row else None

            # Insert data into denormalized table
            insert_query = """
            INSERT INTO denormalized_songs (song_id, song_title, artist_name, album_name, duration, release_date, artist_id, album_id)
            VALUES (%s, %s, %s, %s, %s, %s, %s, %s);
            """
            session.execute(insert_query, (song.song_id, song.song_title, artist_name, album_name, song.duration, song.release_date, song.artist_id, song.album_id))
        print(f"Iteration {i+1} - Data inserted into denormalized_songs table.")

        end_time = time.time()
        runtime = end_time - start_time
        file.write(f"{i+1}: {runtime:.4f} sec\n")
        # Drop the denormalized table after all iterations
        drop_table_query = "DROP TABLE IF EXISTS denormalized_songs;"
        session.execute(drop_table_query)
        print("Denormalized table dropped.")
print(f"Execution times saved in '{file_path}'.")

# Close the connection
session.shutdown()
cluster.shutdown()
print('Disconnected from the Cassandra cluster.')
```

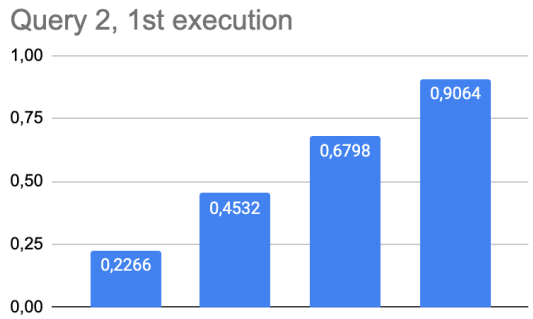
Query 4:

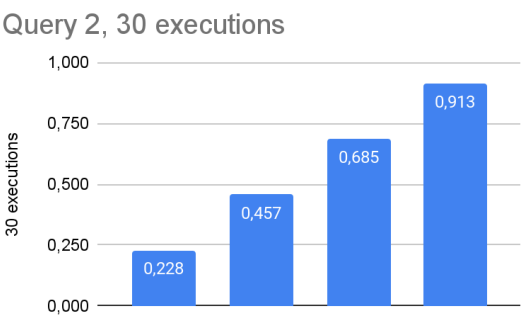
```
import os
import time
from cassandra.cluster import Cluster

# Connect to Cassandra cluster
cluster = Cluster(['localhost'], port=9042)
session = cluster.connect('music_library')

print('Successfully connected to Cassandra.')

# Record execution times in the file
num_executions = 31
folder_name = os.path.join("cassandra", "run-time-results-25")
file_path = os.path.join(folder_name, "runtime_q4.txt")

with open(file_path, "w") as file:
    for i in range(num_executions):
```

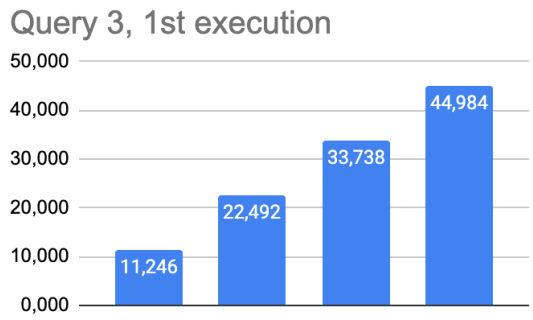
```
    start_time = time.time()

    # Drop the existing artist_song_counts table if it exists
    drop_table_query = "DROP TABLE IF EXISTS artist_song_counts;"
    session.execute(drop_table_query)
    print('Existing artist_song_counts table dropped, if it exists')

    # Create a denormalized table to store the count of songs for each artist
    create_artist_song_counts_table_query = """
        CREATE TABLE IF NOT EXISTS artist_song_counts (
            artist_id UUID PRIMARY KEY,
            song_count COUNTER
        );
    """
    session.execute(create_artist_song_counts_table_query)
    print('Successfully created artist_song_counts table')

    # Query songs with duration less than 300 seconds and update song counts for each artist
    select_songs_query = """
        SELECT artist_id
        FROM songs
        WHERE duration < 300
        ALLOW FILTERING;
    """
    songs_result = session.execute(select_songs_query)
    for song in songs_result:
        artist_id = song.artist_id

        # Increment song count for the artist
        update_count_query = f"""
            UPDATE artist_song_counts
            SET song_count = song_count + 1
            WHERE artist_id = {artist_id};
        """
        session.execute(update_count_query)

    print("Song counts updated for each artist.")

    end_time = time.time()
    runtime = end_time - start_time
    file.write(f"{i+1}: {runtime:.4f} sec\n")

print(f"Execution times saved in '{file_path}'.")

# Close the connection
session.shutdown()
cluster.shutdown()
print('Disconnected from the Cassandra cluster.')
```

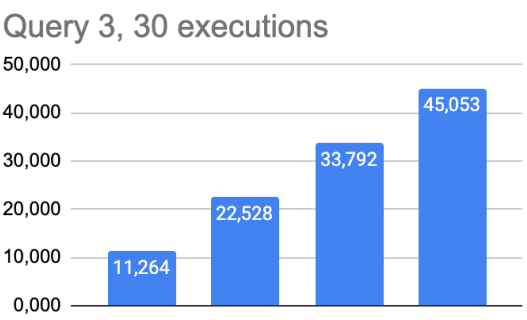


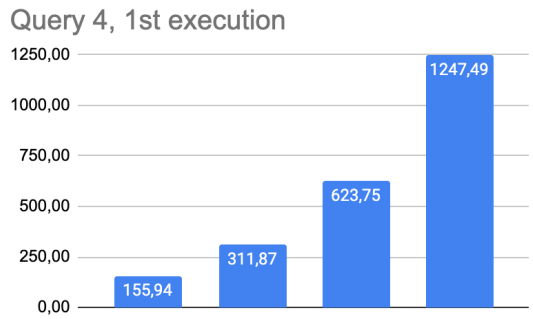


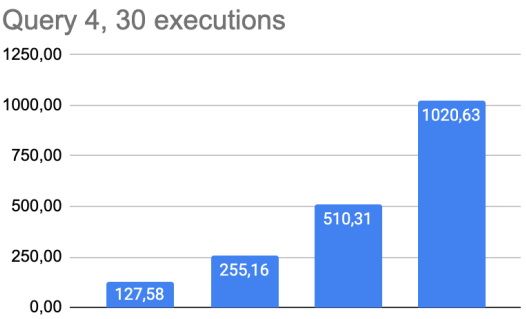












We connected to the Cassandra database by deploying a Docker container and fetching the Cassandra database image, creating a containerized environment for its operation. With the container operational, we proceeded to upload the required data for our library management system.

Data upload was performed by executing necessary commands using the Cassandra Query Language (CQL). We employed CQL statements to define and establish the appropriate tables, ensuring their compatibility with the required data structure for our library management system. Once the tables were set up, we employed Python code to insert data into the Cassandra database, utilizing the Cassandra driver for Python.

Executing queries involved crafting CQL statements to meet specific requirements, retrieving targeted information from the Cassandra database. These queries varied in complexity, ranging from simple to advanced levels, enabling us to evaluate Cassandra's performance and capabilities across various query scenarios.

By establishing connection through the Docker container, uploading data, and executing queries, we effectively assessed Cassandra's functionality and performance as a database solution for our library management system.

Conclusion

After extensive testing spanning databases ranging in size from 250 thousand to 1 million, our journey through the digital realms has led us to a resounding conclusion: MongoDB reigns supreme in our scenario. Its adeptness in handling queries of varying complexity with minimal time and resource expenditure distinguishes it as the champion of our database evaluation.

Following closely behind MongoDB is Neo4j, a standout performer with its swift performance. Despite its slight lag compared to MongoDB, Neo4j's graph-based structure adds a layer of complexity that, while potent for certain applications, presents challenges in processing relational data.

Redis, while noteworthy, falls short in comparison to MongoDB. Its architecture, while impressive, lacks the tailored optimization needed for our workload, placing it slightly behind MongoDB in our evaluation.

In our context, Cassandra's performance is hindered by its sluggish and resource-intensive operations, especially when handling complex queries such as joins across disparate tables. This impediment significantly affects the efficiency of data retrieval tasks, relegating Cassandra to a less favorable position in our evaluation.

In synthesizing our findings, MongoDB emerges as the unrivaled champion, demonstrating unparalleled efficiency and effectiveness across a spectrum of tasks. Its adaptability and scalability make it the quintessential choice for our database requirements.

It's worth noting that each database management system offers distinct strengths and weaknesses. RedisDB, with its in-memory storage and low-latency characteristics, excels in high-performance data operations. Cassandra boasts scalability and high write performance, albeit with slightly slower read performance compared to RedisDB. SQL databases, exemplified by MariaDB, offer reliability and robust transactional capabilities. MongoDB, with its focus on flexibility and query capabilities, suits various applications. Meanwhile, Neo4j's strength lies in managing complex relationships, albeit with reduced raw performance compared to other database types.

In conclusion, MongoDB stands atop the podium, followed by Neo4j, RedisDB, SQL (MariaDB), and Cassandra. The selection of the most suitable database hinges on understanding specific requirements and trade-offs relevant to the library management system.