UNIVERSITÀ DEGLI STUDI DI MESSINA

MIFT DEPARTMENT

DATA ANALYSIS PROGRAM

Database Mod B

Discussion and comparison the performance of 5 famous databases

Professor:
Armando Ruggeri

Student:
Yahia Ahmed

ID: ***532118***

ACADEMIC YEAR – 2022/2023

—

Table of contents

Introduction

The rapid growth of social networking platforms has transformed the way people connect, interact, and share information. In this project, we aim to develop a NoSQL database solution for a social networking platform that specifically caters to the needs of students and faculty members. The platform will provide a seamless environment for users to create profiles, share posts, exchange messages, and engage with each other.

To accomplish this, we will explore and compare various NoSQL databases, including MySQL, MongoDB, Cassandra, Redis, and Neo4j. Each of these databases offers unique features and characteristics that can impact the performance and scalability of our social networking platform. By evaluating these databases and analyzing their capabilities, we can determine which one best suits the requirements of our application.

To ensure accurate and comprehensive analysis, we will create multiple datasets of increasing sizes, ranging from 250,000 to 1,000,000 records. These datasets will contain similar information content and will be inserted into the chosen databases. By comparing the results of queries performed on these databases, we can assess the efficiency and effectiveness of each solution.

Furthermore, we will design and execute a set of queries with increasing degrees of complexity to evaluate the performance of the databases. These queries will involve different entities and selection filters, allowing us to understand how each database handles more intricate operations. We will automate the experiments and measure query execution times, running each test at least 31 times to obtain reliable statistics, considering two histograms will be created in which one of them will be to visualize the first execution (without caching) and the average of the 30 other execution times.

The results of these experiments will be saved in an electronic spreadsheet and presented through histograms, showcasing the response times in milliseconds. By analyzing the data, we will determine the database that exhibits superior query execution times under equal hardware and software conditions.

In conclusion, this project aims to develop and evaluate a NoSQL database solution for a social networking platform targeting students and faculty members. By comparing multiple databases, creating datasets, and conducting performance tests, we will identify the most efficient database to support the platform's requirements. The findings of this study will provide valuable insights into the selection and optimization of NoSQL databases for social networking applications.

DBMSs chosen and main characteristics

In this project, we have chosen to evaluate and compare five different database management systems (DBMSs) for the development of a social networking platform: MySQL, MongoDB, Cassandra, Redis, and Neo4j. Each of these DBMSs offers distinct features and characteristics that can impact their suitability for our application. Below, we provide an overview of each DBMS and highlight their main characteristics:

1. **MySQL**: MySQL is a widely-used open-source relational database management system (RDBMS) that has been a popular choice for many years. Although it is a relational database, it is included in this comparison as a representative of traditional SQL databases. MySQL offers ACID (Atomicity, Consistency, Isolation, Durability) compliance, robust data integrity, and transaction support. It provides a structured and schema-based approach to data storage and retrieval.

2. **MongoDB**: MongoDB is a document-oriented NoSQL database that stores data in flexible, JSON-like documents. It is designed for scalability and high-performance applications, making it suitable for handling large volumes of data. MongoDB's flexible document model allows for dynamic schema changes and supports complex queries, indexing, and ad-hoc queries. It also provides horizontal scalability through sharding, enabling distribution of data across multiple servers.

3. **Cassandra**: Apache Cassandra is a highly scalable and distributed NoSQL database known for its ability to handle massive amounts of data across multiple commodity servers while providing continuous availability. Cassandra is designed to handle high read and write throughput with low latency. It offers a decentralized architecture, tunable consistency levels, and a flexible data model. Cassandra is particularly well-suited for applications that require high availability and fault tolerance.

4. **Redis**: Redis is an open-source in-memory data structure store that can serve as a database, cache, and message broker. It is known for its exceptional performance, high speed, and low latency. Redis stores data in key-value pairs and supports various data structures, such as strings, lists, sets, and sorted sets. It provides advanced features like data replication, persistence, and pub/sub messaging, making it a versatile choice for real-time applications and caching purposes.

5. **Neo4j**: Neo4j is a graph database that focuses on storing and querying highly connected data. It represents data as nodes, relationships, and properties, allowing for the efficient traversal of complex relationships. Neo4j provides a native graph processing engine, enabling powerful graph queries and graph analytics. It is well-suited for applications that require deep

relationship exploration, such as social networks, recommendation engines, and fraud detection systems.

By evaluating these five DBMSs, we aim to assess their suitability for our social networking platform based on criteria such as performance, data model flexibility, query capabilities, and ease of use. The following sections of this report will delve into the experimentation process, including dataset creation, query performance evaluation, and analysis of the results, in order to determine the most appropriate DBMS for our application.

Data Models and schemas

MySql Schema:

schema.txt
You, 1 second ago | 2 authors (Yahia and others)

```
CREATE DATABASE IF NOT EXISTS social_media;

USE social_media;

CREATE TABLE IF NOT EXISTS users (
  user_id VARCHAR(50) PRIMARY KEY,
  username VARCHAR(255),
  full_name VARCHAR(255),
  email VARCHAR(255),
  password VARCHAR(255),
  profile_picture VARCHAR(255),
  bio TEXT
);      Yahia, 2 days ago • 2nd commit with schema

CREATE TABLE IF NOT EXISTS posts (
  post_id VARCHAR(50),
  user_id VARCHAR(50),
  content TEXT,
  timestamp TIMESTAMP,

  PRIMARY KEY (post_id, user_id)
);

CREATE TABLE IF NOT EXISTS messages (
  message_id VARCHAR(50) PRIMARY KEY,
  sender_user_id VARCHAR(50),
  receiver_user_id VARCHAR(50),
  content TEXT,
  timestamp TIMESTAMP
);

CREATE TABLE IF NOT EXISTS connections (
  user_id_1 VARCHAR(50),
  user_id_2 VARCHAR(50),
  timestamp TIMESTAMP,
  PRIMARY KEY (user_id_1, user_id_2)
);
```

redis:

```
Yahia, 2 days ago | 1 author (Yahia)

for the users:

HSET user:{user_id} username {username} full_name {full_name} email {email} password {password} profile_picture {profile_picture} bio {bio}


for the posts:

HSET post:{post_id} user_id {user_id} content {content} timestamp {timestamp} post_count {post_count} message_count {message_count} connection_count {connection_count}

for the messages:      Yahia, 2 days ago • 2nd commit with schema

HSET message:{message_id} sender_user_id {sender_user_id} receiver_user_id {receiver_user_id} content {content} timestamp {timestamp}

Connections --we used ZADD for sorting:

ZADD connections:{user_id_1} {timestamp} {user_id_2}
```

Neo4j:

here attached the cypher statement used to do the schema and also to import the csv files using the Bulk import:

```
LOAD CSV FROM 'file:///users_500k.csv' AS row
CREATE (:User {
  user_id: row[0],
  username: row[1],
  full_name: row[2],
  email: row[3],
  password: row[4],
  profile_picture: row[5],
  bio: row[6]
});

LOAD CSV FROM 'file:///posts_500k.csv' AS row
CREATE (:Post {
  post_id: row[0],
  user_id: row[1],
  content: row[2],
  timestamp: datetime(row[3])
});

LOAD CSV FROM 'file:///messages_500k.csv' AS row
CREATE (:Message {
  message_id: row[0],
  sender_user_id: row[1],
  receiver_user_id: row[2],
  content: row[3],
  timestamp: datetime(row[4])
});

LOAD CSV FROM 'file:///friends_500k.csv' AS row
CREATE (:Connection {
  user_id_1: row[0],
  user_id_2: row[1],
  timestamp: datetime(row[2])
});
```

Cassandra Model:

dra > schema.txt

```
CREATE KEYSPACE IF NOT EXISTS social_media
  WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};

USE social_media;

CREATE TABLE IF NOT EXISTS users (
  user_id UUID PRIMARY KEY,
  username text,
  full_name text,
  email text,
  password text,
  profile_picture text,      Yahia, 2 days ago • 2nd commit with schema
  bio text
);

CREATE TABLE IF NOT EXISTS posts (
  post_id UUID PRIMARY KEY,
  user_id UUID,
  content text,
  timestamp timestamp
);

CREATE TABLE IF NOT EXISTS messages (
  message_id UUID PRIMARY KEY,
  sender_user_id UUID,
  receiver_user_id UUID,
  content text,
  timestamp timestamp
);

CREATE TABLE IF NOT EXISTS connections (
  user_id_1 UUID,
  user_id_2 UUID,
  timestamp timestamp,
  PRIMARY KEY (user_id_1, user_id_2)
);
```

MongoDB:

```
db.createCollection("users")
db.createCollection("posts")
db.createCollection("messages")
db.createCollection("friends")      You, 13 seconds ago • Uncommitted c


User:

User ID (unique identifier)
Username
Full name
Email
Password (hashed or encrypted)
Profile picture
Bio or About Me section


Posts:

Post ID (unique identifier)
User ID of the author
Content (text)
Timestamp for post creation


Messages:

Message ID (unique identifier)
Sender User ID
Receiver User ID
Content (text)
Timestamp for message sent



Friends/Connections:

User ID of the first user
User ID of the second user
Timestamp for connection establish
```

Implementation Phase

1- **Fake Data Generation**:

before starting the database feeding data phase we have to generate the data that will be pushed to the databases to be able to do our queries , In the data generations I used a famous and stable python library called **Faker,** after the data generation of different dataset sizes 250k,500k,750k and 1m , I saved the data rows in csv files, since it is a supported extension files which has many supported modules in python, and also I put it in a csv files instead of json, because some databases like mySQL and neo4j, it has the feature of the fast bulk import (especialy when you have a large datasets).

(since the script used is little bit long, I will not be able to put it here, but it is available on the guthub repo)

Database filling with data:

MySQL:

firstly, I loaded the data from the dedicated CSV file of a specific dataset size:

```python
def load_data_from_csv(file_name):
    data = []
    with open(file_name, 'r', encoding='utf-8') as file:
        reader = csv.reader(file)
        for row in reader:      Yahia, 2 days ago • 1st commit
            data.append(row)
    return data
```

Then using the data returned from the csv file, I created a map data structure, then I used it to feed the data to the database:

```python
# Users Table
for row in users_data_1m:
    user_id, username, full_name, email, password, profile_picture, bio = row
    user = {
        'user_id': uuid.UUID(user_id).bytes,
        'username': username,
        'full_name': full_name,
        'email': email,
        'password': password,
        'profile_picture': profile_picture,
        'bio': bio
    }
    insert_data('users', user)

# Posts Table
for row in posts_data_1m:
    post_id, user_id, content (variable) post_id: Any
    post = {
        'post_id': uuid.UUID(post_id).bytes,
        'user_id': uuid.UUID(user_id).bytes,
        'content': content,
        'timestamp': datetime.strptime(timestamp.split('.')[0], '%Y-%m-%d %H:%M:%S')
    }
    insert_data('posts', post)

# Messages Table
for row in messages_data_1m:
    message_id, sender_id, receiver_id, content, timestamp = row
    message = {
        'message_id': uuid.UUID(message_id).bytes,
        'sender_user_id': uuid.UUID(sender_id).bytes,
        'receiver_user_id': uuid.UUID(receiver_id).bytes,
        'content': content,
        'timestamp': datetime.strptime(timestamp.split('.')[0], '%Y-%m-%d %H:%M:%S')
    }
    insert_data('messages', message)

# Connections Table
for row in friends_data_1m:
    user1_id, user2_id, timestamp = row
    friend = {
        'user_id_1': uuid.UUID(user1_id).bytes,
        'user_id_2': uuid.UUID(user2_id).bytes,
        'timestamp': datetime.strptime(timestamp.split('.')[0], '%Y-%m-%d %H:%M:%S')
    }
```

then we insert the data using :

```python
def insert_data(table_name, data):
    columns = ', '.join(data.keys())
    placeholders = ', '.join(['%s'] * len(data))
    query = f"INSERT INTO {table_name} ({columns}) VALUES ({placeholders})"
    cursor.execute(query, tuple(data.values()))
```

Redis:

same approach as mySQL, same load function from csv and same Map data structure used.

Then the Insert function looked like:

```python
def insert_data(key, data):
    redis_client.set(key, json.dumps(data))
```

neo4j:

since neo4j takes a massive amount of time, for the data to be loaded, I used the Bulk import command in the neo4j web server:

Cassandra:

Same approach as mySql and reddis, same load data functiona and same data structure, then I used this function to insert the records:

```
def insert_data(table_name, data):
    insert_query = f"INSERT INTO {table_name} ({', '.join(data.keys())}) VALUES ({', '.join(['%s'] * len(data))})"
    session.execute(insert_query, tuple(data.values()))      Yahia, 2 days ago • 1st commit
```

mongoDB:

Same approach as mySql, cassnadra and reddis, same load data function and same data structure, then I used this function to insert the records:

```
def insert_data(collection_name, data):
    collection = db[collection_name]
    collection.insert_many(data)
```

*** The code is completely uploaded in the github remote repo ***

Queries Implementation

Every query is executed 31 times

MySQL:

query1: “Query 1: retrieve only users “bJames”

```
post_query = """
SELECT username FROM users WHERE username = 'bjames'
"""
```

query2:"Query 2: retrieve users who posted at least 3 things"

```
post_query = """
SELECT user_id FROM posts GROUP BY user_id HAVING COUNT(*) >= 3
"""      Yahia, 2 days ago • 1st commit
```

query3:"Query 3: retrieve users who posted at least 3 things and sent at least 5 private messages to anybody"

```
# Query users who posted at least 3 things
post_query = """
SELECT user_id FROM posts GROUP BY user_id HAVING COUNT(*) >= 3
"""
cursor.execute(post_query)
post_result = cursor.fetchall()
post_user_ids = [str(row[0]) for row in post_result]

# Query users who sent at least 5 private messages      Yahia, 2 days ago • 1st co
message_query = """
SELECT sender_user_id FROM messages GROUP BY sender_user_id HAVING COUNT(*) >= 5
"""
cursor.execute(message_query)
message_result = cursor.fetchall()
message_user_ids = [str(row[0]) for row in message_result]

# Find common user_ids between the two sets
user_ids = set(post_user_ids) & set(message_user_ids)
```

query4:"Query 4: retrieve users who posted at least 3 things and sent at least 5 private messages to someone they are connected in a relationship"

```
post_query = """
SELECT CAST(user_id AS CHAR) FROM posts GROUP BY user_id HAVING COUNT(*) >= 3
"""
cursor.execute(post_query)
post_result = cursor.fetchall()
post_user_ids = [str(row[0]) for row in post_result]

# Query users who sent at least 5 private messages
message_query = """
SELECT CAST(sender_user_id AS CHAR) FROM messages GROUP BY sender_user_id HAVING COUNT(*) >= 5
"""
cursor.execute(message_query)
message_result = cursor.fetchall()
message_user_ids = [str(row[0]) for row in message_result]

# Query connections
connection_query = "SELECT CAST(user_id_1 AS CHAR), CAST(user_id_2 AS CHAR) FROM connections"
cursor.execute(connection_query)
connection_result = cursor.fetchall()
connection_user_ids = [(str(row[0]), str(row[1])) for row in connection_result]      Yahia, 2 days ago

# Find common user_ids between the three sets
user_ids = set(post_user_ids) & set(message_user_ids) & set(connection_user_ids)
```

Cassandra:

query1: "Query 1: retrieve only users "bJames""

```
query = "SELECT * FROM users WHERE username = 'bjames' ALLOW FILTERING"
```

query2:"Query 2: retrieve users who posted at least 3 things"

```
query = "SELECT user_id, COUNT(*) as count FROM posts"
```

query3:"Query 3: retrieve users who posted at least 3 things and sent at least 5 private messages to anybody"

```
post_query = "SELECT user_id FROM posts"      Yahia, 2 days ago • 1st commi
post_result = session.execute(post_query)
post_counts = {}
for row in post_result:
    user_id = row.user_id
    if user_id in post_counts:
```

```
# Query users who posted at least 3 things
post_query = "SELECT user_id, COUNT(*) AS post_count FROM posts"
post_result = session.execute(post_query)
post_user_ids = [row.user_id for row in post_result if row.post_count >= 3]

# Query users who sent at least 5 private messages
message_query = "SELECT sender_user_id, COUNT(*) AS message_count FROM messages"
message_result = session.execute(message_query)
message_user_ids = [row.sender_user_id for row in message_result if row.message_count >= 5]

# Query connections
connection_query = "SELECT user_id_1, user_id_2 FROM connections"
connection_result = session.execute(connection_query)
connection_user_ids = [(row.user_id_1, row.user_id_2) for row in connection_result]

# Find common user_ids between the three sets
user_ids = set(post_user_ids) & set(message_user_ids) & set(connection_user_ids)
```

Redis:

query1: "Query 1: retrieve only users "bJames"**"**

query2:"Query 2: retrieve users who posted at least 3 things**"**

```
# Find users with at least 3 posts
user_ids_posted_at_least_3 = []
keys = r.scan_iter(match=POST_PREFIX + '*')
for key in keys:
    key_type = r.type(key)
    if key_type == b'set':
        post_count = r.scard(key)
        if post_count >= 3:
            user_id = key.decode('utf-8').split(':')[1]
            user_ids_posted_at_least_3.append(user_id)

# Fetch users with at least 3 posts
users_with_at_least_3_posts = []
for user_id in user_ids_posted_at_least_3:
    user = r.hgetall(USER_PREFIX + user_id)
    users_with_at_least_3_posts.append(user)
```

query3:”Query 3: retrieve users who posted at least 3 things and sent at least 5 private messages to anybody”

```
user_ids_posted_at_least_3 = []
keys = r.scan_iter(match=POST_PREFIX + '*')
for key in keys:
    key_type = r.type(key)
    if key_type == b'set':
        post_count = r.scard(key)
        if post_count >= 3:
            user_id = key.decode('utf-8').split(':')[1]
            user_ids_posted_at_least_3.append(user_id)

# Find users who sent at least 5 private messages
user_ids_sent_at_least_5_messages = []      Yahia, 2 days ago • 1st commit
keys = r.scan_iter(match=MESSAGE_PREFIX + '*')
for key in keys:
    key_type = r.type(key)
    if key_type == b'set':
        message_count = r.scard(key)
        if message_count >= 5:
            user_id = key.decode('utf-8').split(':')[1]
            user_ids_sent_at_least_5_messages.append(user_id)

# Fetch users who satisfy both conditions
users_with_at_least_3_posts_and_5_messages = list(
    set(user_ids_posted_at_least_3).intersection(user_ids_sent_at_least_5_messages))
```

query4:”Query 4: retrieve users who posted at least 3 things and sent at least 5 private messages to someone they are connected in a relationship”

```
    user_ids_posted_at_least_3 = []
    keys = r.scan_iter(match=POST_PREFIX + '*')
    for key in keys:
        if r.type(key) == b'set':
            post_count = r.scard(key)
            if post_count >= 3:
                user_id = key.decode('utf-8').split(':')[1]
                user_ids_posted_at_least_3.append(user_id)

    # Find users who sent at least 5 private messages
    users_with_at_least_3_posts_and_5_messages = []
    keys = r.scan_iter(match=MESSAGE_PREFIX + '*')
    for key in keys:
        if r.type(key) == b'set':
            message_count = r.scard(key)
            if message_count >= 5:
                user_id = key.decode('utf-8').split(':')[1]
                users_with_at_least_3_posts_and_5_messages.append(user_id)

    # Fetch users who satisfy both conditions
    users = []
    for user_id in users_with_at_least_3_posts_and_5_messages:
        user = r.hgetall(USER_PREFIX + user_id)
        users.append(user)
```

Mongodb:

query1: "Query 1: retrieve only users "bJames""

```
query = {'username': 'bjames'}
```

query2:"Query 2: retrieve users who posted at least 3 things"

```
    posted_at_least_3 = db.posts.aggregate([
    {"$group": {"_id": "$user_id", "count": {"$sum": 1}}},
    {"$match": {"count": {"$gte": 3}}}
], allowDiskUse=True)
        Yahia, 2 days ago • 1st commit
    user_ids_posted_at_least_3 = [user["_id"] for user in posted_at_least_3]
```

query3:”Query 3: retrieve users who posted at least 3 things and sent at least 5 private messages to anybody”

```
    You, 1 second ago • Uncommitted changes
users_with_3_posts = db.posts.aggregate([
    {"$group": {"_id": "$user_id", "post_count": {"$sum": 1}}},
    {"$match": {"post_count": {"$gte": 3}}}
], allowDiskUse=True)

user_ids_with_3_posts = [user["_id"] for user in users_with_3_posts]

users_with_5_private_messages = db.messages.aggregate([
    {"$group": {"_id": "$Sender User ID", "message_count": {"$sum": 1}}},
    {"$match": {"message_count": {"$gte": 5}}}
], allowDiskUse=True)

user_ids_with_5_private_messages = [user["_id"] for user in users_with_5_private_messages]

users_with_3_posts_and_5_private_messages = db.users.find({
    "_id": {"$in": user_ids_with_3_posts},
    "_id": {"$in": user_ids_with_5_private_messages}
})
```

query4:”Query 4: retrieve users who posted at least 3 things and sent at least 5 private messages to someone they are connected in a relationship”

```
users_with_3_posts = db.posts.aggregate([
    {"$group": {"_id": "$user_id", "post_count": {"$sum": 1}}},
    {"$match": {"post_count": {"$gte": 3}}}
], allowDiskUse=True)

user_ids_with_3_posts = [user["_id"] for user in users_with_3_posts]

users_with_5_private_messages = db.messages.aggregate([
    {"$group": {"_id": "$Sender User ID", "message_count": {"$sum": 1}}},
    {"$match": {"message_count": {"$gte": 5}}}
], allowDiskUse=True)

user_ids_with_5_private_messages = [user["_id"] for user in users_with_5_private_messages]

users_with_3_posts_and_5_private_messages = db.users.aggregate([
{"$match": {"_id": {"$in": user_ids_with_3_posts}}},
{"$lookup": {
    "from": "friends",
    "let": {"user_id": "$_id"},
    "pipeline": [
        {"$match": {
            "$expr": {
                "$or": [
                    {"$eq": ["$$user_id", "$User ID of the first user"]},
                    {"$eq": ["$$user_id", "$User ID of the second user"]}
```

Neo4j

query1: "Query 1: retrieve only users "bJames""

```
query = '''
    MATCH (u:User {username: 'bjames'})
    RETURN u      Yahia, 2 days ago • neo4j is added
'''
```

query2:"Query 2: retrieve users who posted at least 3 things"

```
query = '''
    MATCH (p:Post)
    WITH p.user_id AS user_id, COUNT(*) AS postCount
    WHERE postCount >= 3
    RETURN user_id
      You, 9 seconds ago • Uncommitted changes
'''
```

query3:"Query 3: retrieve users who posted at least 3 things and sent at least 5 private messages to anybody"

```
query = '''
    MATCH (m:Message)
    WITH m.sender_user_id AS user_id, COUNT(*) AS mCount
    WHERE mCount >= 5
    RETURN user_id
'''      Yahia, 2 days ago • neo4j is added
```

query4:"Query 4: retrieve users who posted at least 3 things and sent at least 5 private messages to someone they are connected in a relationship"

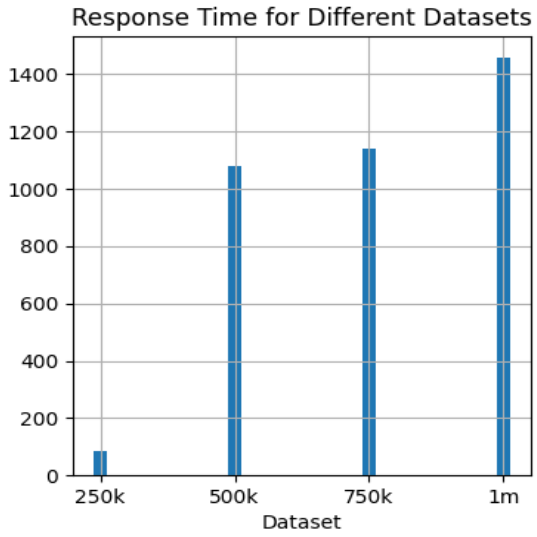
```
query1 = '''
    MATCH (m:Message)
    WITH m.sender_user_id AS user_id, COUNT(*) AS mCount
    WHERE mCount >= 5
    RETURN user_id
'''
query2 = '''
MATCH (p:Post)
    WITH p.user_id AS user_id, COUNT(*) AS postCount
    WHERE postCount >= 3
    RETURN user_id
'''

# Perform the experiments
num_experiments =31
response_times = []

with driver.session() as session:
    for i in range(num_experiments):
        # Measure the response time
        start_time = datetime.now()
        result1 = list(session.run(query1))      You, 1 second ago • Un
        result2=  list(session.run(query2))
        combined_users = set(result1).union(result2)
```

Experiments

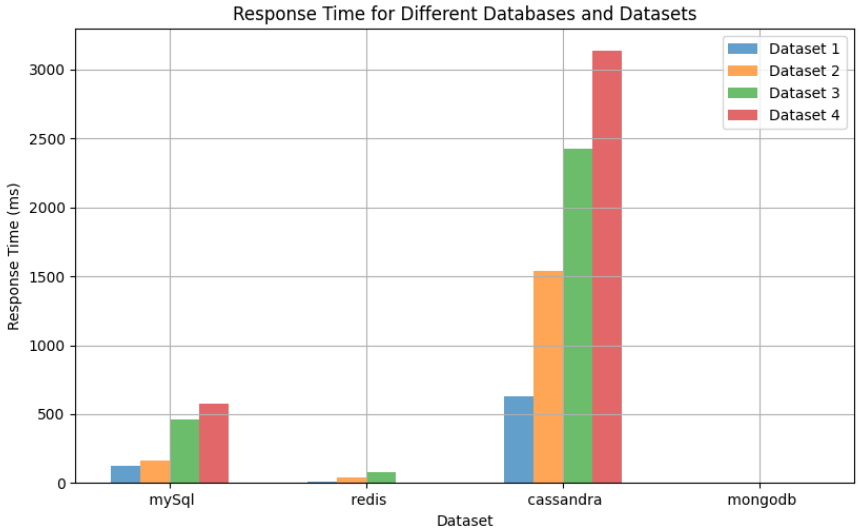
Query 1 [Execution of the first time]:

Query 1: retrieve only users “bJames”

Neo4j:

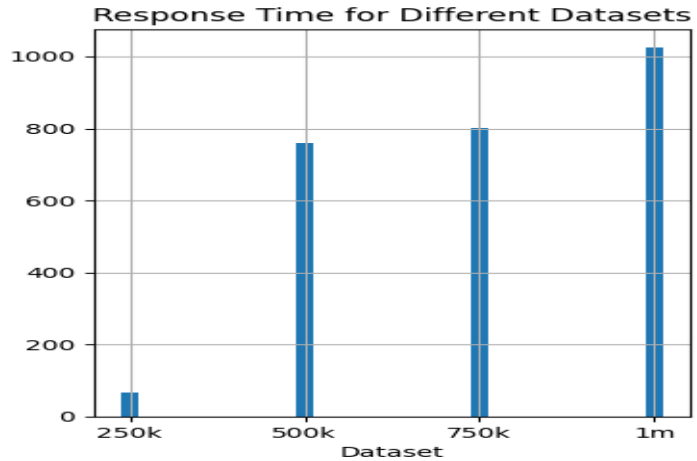


the other 4 databases:

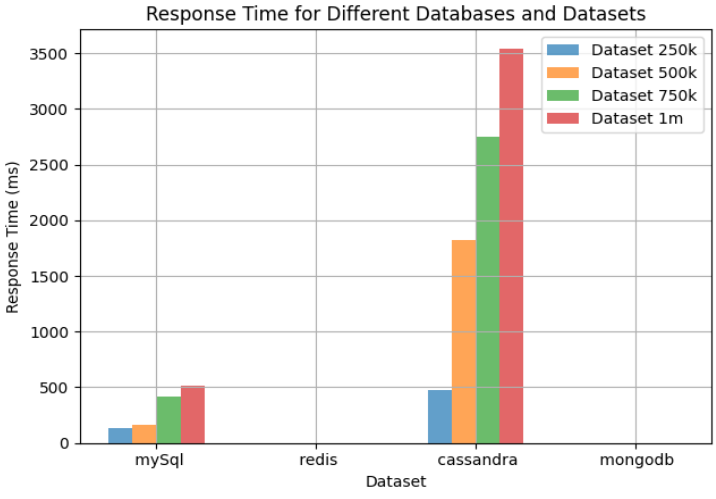


Query 1 [Execution of the Average]:

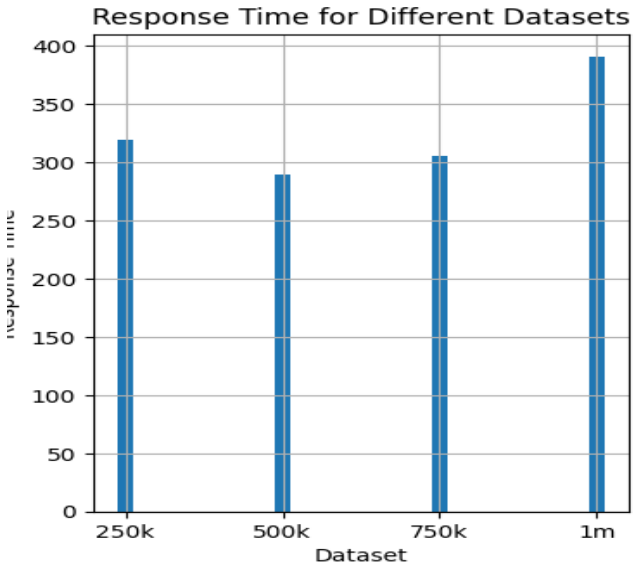
Neo4j:

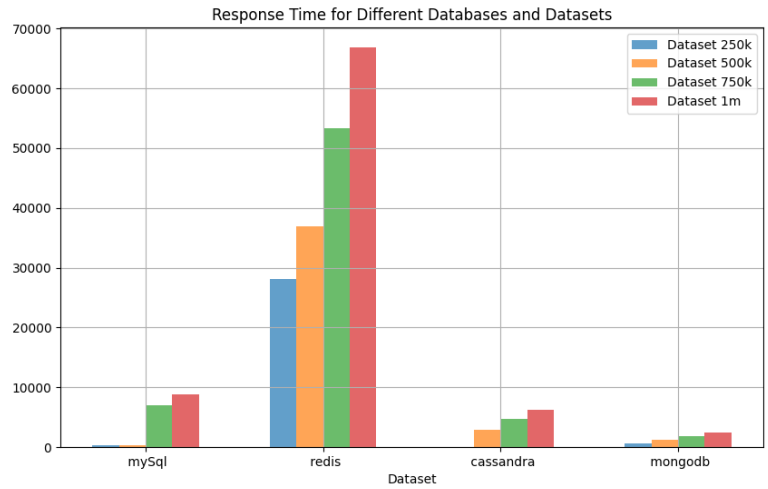


the other 4 databases:

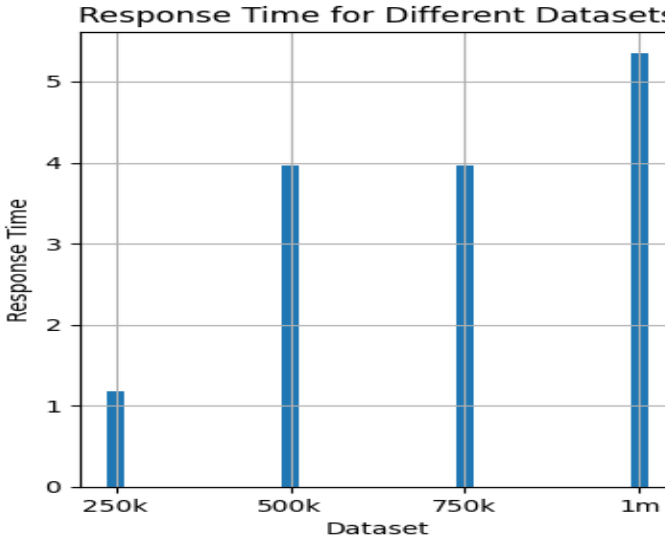


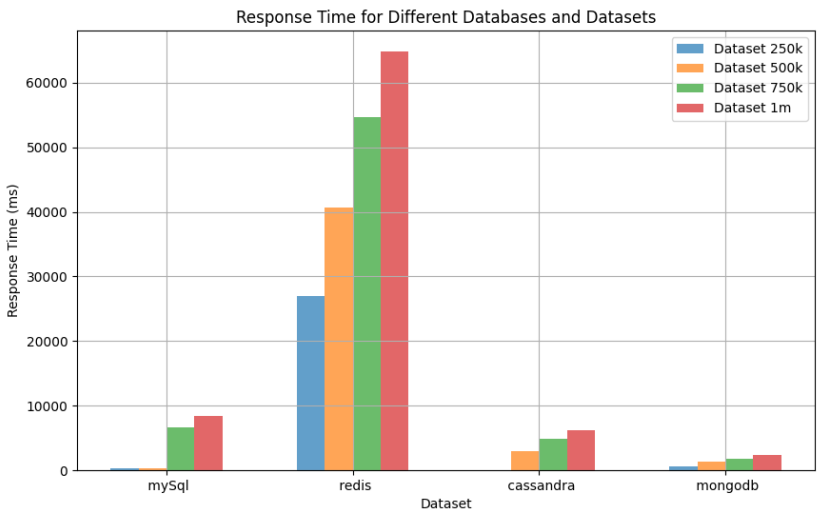
Query 2 [Execution of the first time]:





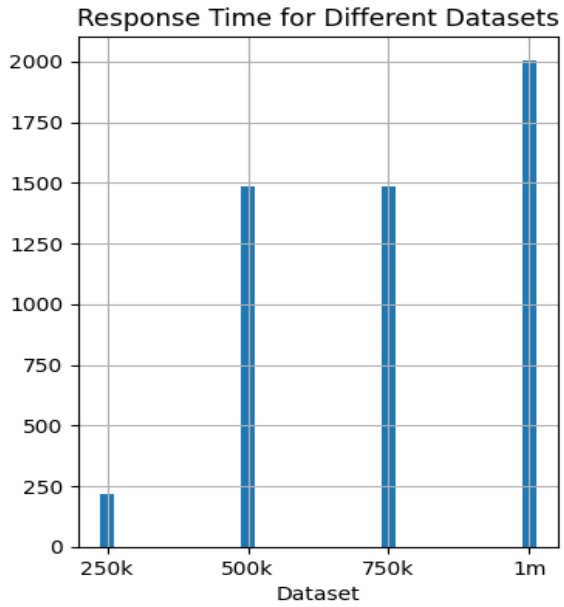
Query 2 [Execution of the Average]:

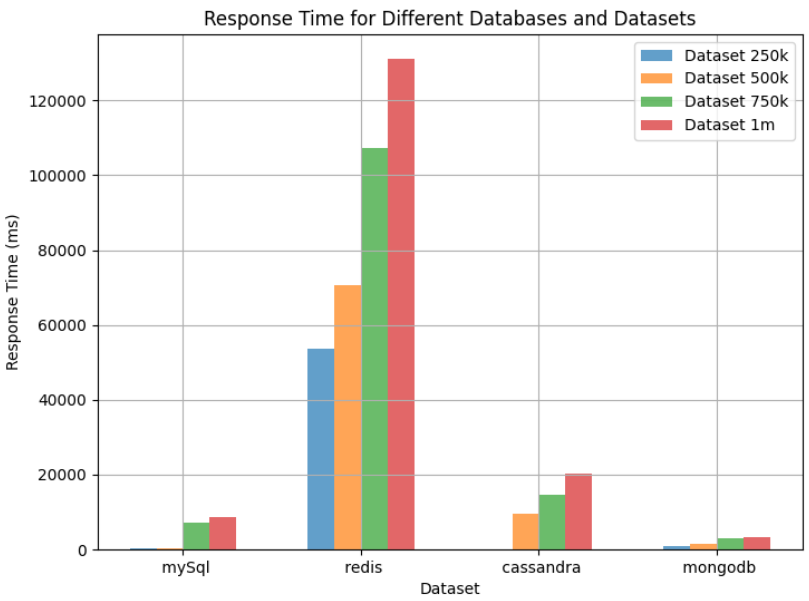




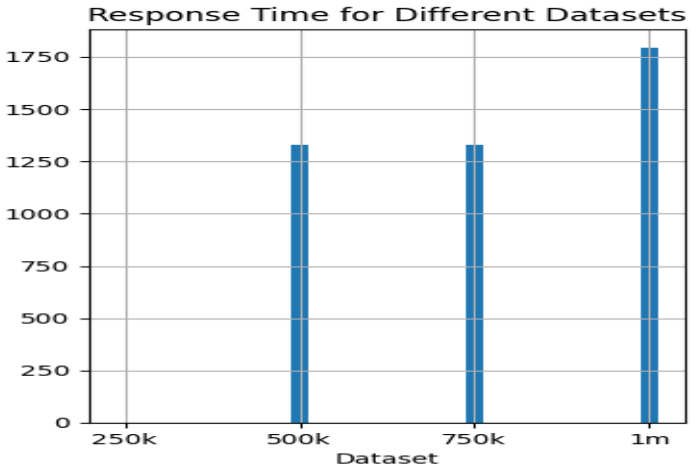
Query 3 [Execution of the first time]:

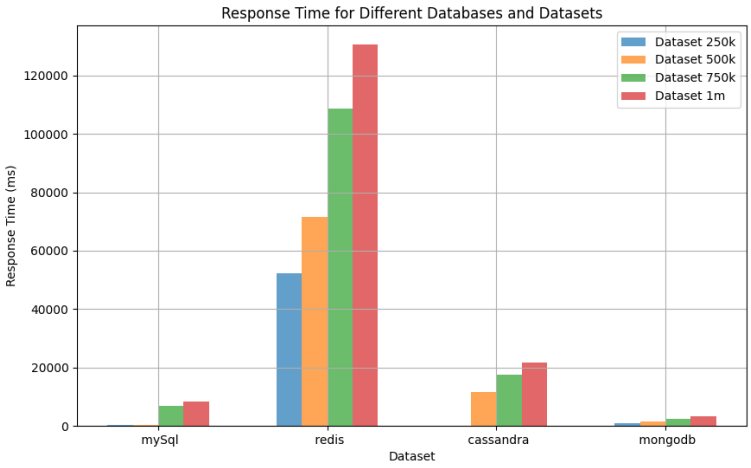
Query 3: retrieve users who posted at least 3 things and sent at least 5 private messages to anybody





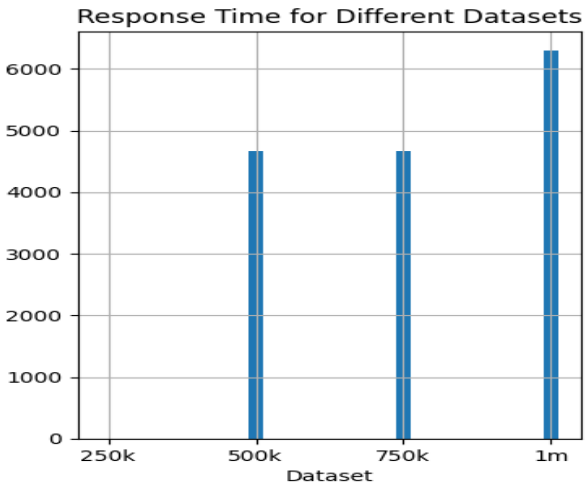
Query 3 [Execution of the Average]:

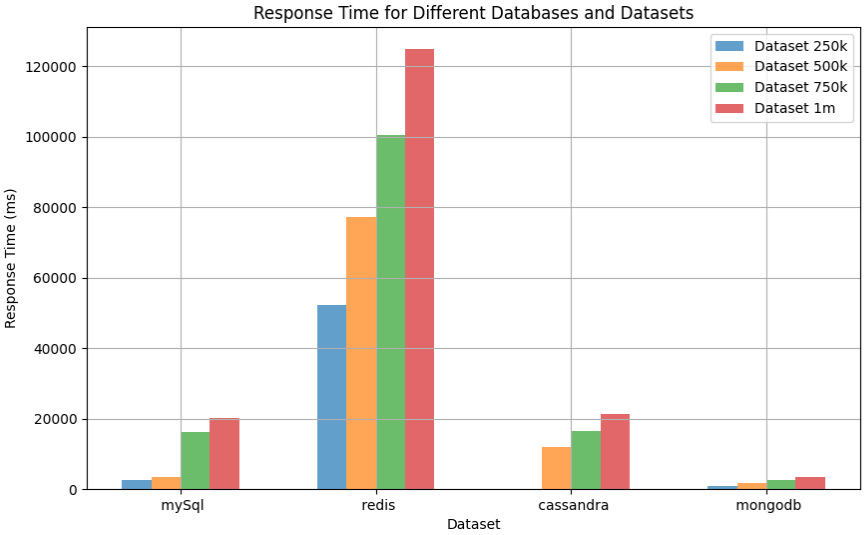




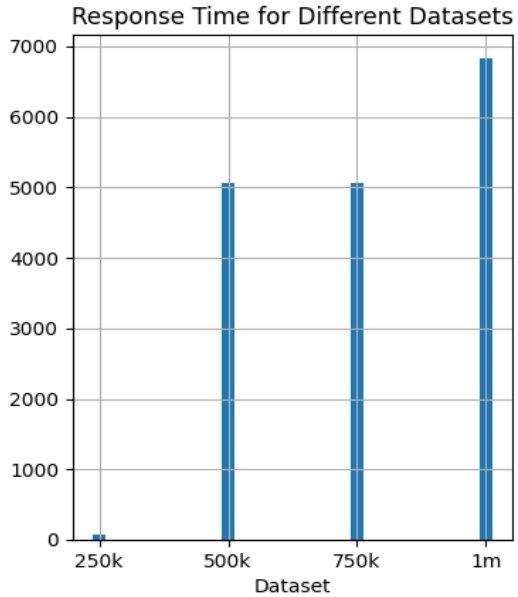
Query 4 [Execution of the first time]:

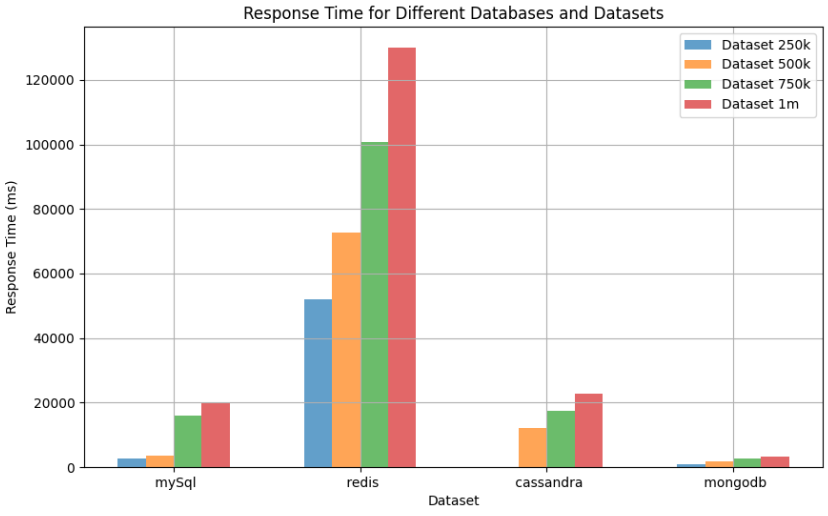
Query 4: retrieve users who posted at least 3 things and sent at least 5 private messages to someone they are connected in a relationship





Query 4 [Execution of the Average]:





* **ALL the spreadsheets with all the SI values of response times for each dataset in database are available on [github](#)******

Conclusions

In this project, we evaluated and compared five different NoSQL database management systems (DBMSs) for the development of a social networking platform. We explored MySQL, MongoDB, Cassandra, Redis, and Neo4j, each

offering distinct features and characteristics that can impact their suitability for our application.

Through our experiments, we gained valuable insights into the performance, scalability, and query capabilities of each DBMS. It is important to note that the selection of a DBMS should be based on the specific business logic and requirements of the application. Each DBMS has

its own strengths and weaknesses, making it essential to align the choice with the desired functionalities and data model.

MySQL, as a representative of traditional SQL databases, demonstrated strong transactional support and data integrity. It is well-suited for applications with structured and schema-based data requirements. MongoDB showcased its flexibility and scalability, with support for dynamic schema changes and distributed data storage. Cassandra excelled in its ability to handle massive amounts of data with high availability and fault tolerance, making it ideal for large-scale applications.

Neo4j proved its efficiency in handling highly connected data and performing complex graph queries, making it an excellent choice for applications requiring deep relationship exploration. However, it should be noted that the suitability of Neo4j depends on the nature of the data and the specific use case.

On the other hand, Redis, despite its versatility as an in-memory data structure store and caching solution, exhibited slower speeds compared to the other DBMSs in our experiments. This suggests that while Redis can be an effective tool for certain use cases, its performance may not be optimal for applications with high-speed requirements.

In conclusion, the selection of a DBMS for a social networking platform, or any application, should be based on careful consideration of the specific business requirements and the strengths of each DBMS. MySQL, MongoDB, Cassandra, and Neo4j each offer unique features and capabilities, and their suitability depends on the specific needs of the application. Redis, while versatile, may not provide the desired performance in scenarios requiring high-speed data processing.

By conducting this comparative analysis, we have laid the foundation for making an informed decision on the most suitable DBMS for our social networking platform. The insights gained from this study will help guide future development and optimization efforts, ensuring the best possible performance and user experience for our application.