Department of Mathematics and Computer Science
Faculty of Data Analysis
UNIME

Database mod NoSQL

Final Report

Comparative Analysis of Database Performance in a Library Management System across Various NoSQL solutions

Students: Alina Atayeva (533021)
Alikhan Alashybay (536353)
Professors: Armando Ruggeri, Antonio Celesti

Academic Year 2022/2023

Table of Contents

Introduction

Description of the project

The management and organization of data play a crucial role in the efficiency and effectiveness of various systems and applications. With the advent of NoSQL databases, the database landscape has expanded beyond traditional SQL databases, offering diverse options for storing and retrieving data. In this project, we focused on implementing a library management system and examining the performance of five different database types: SQL, Cassandra, MongoDB, RedisDB, and Neo4j. The objective of this project was to evaluate the capabilities and performance of these databases by executing queries of varying complexity on different dataset sizes. By comparing the results, we aimed to identify the strengths and weaknesses of each database type in the context of a library management system. This analysis will help us understand the suitability of each database for specific use cases and guide future database selection decisions.

To conduct a comprehensive assessment, we employed a dataset of varying sizes, ranging from 250,000 to 1 million records. By scaling the dataset, we aimed to observe the behavior and performance of the databases under increasing data loads. Our focus was not only on simple queries but also on more complex queries that simulate real-world scenarios in a library management system.

Throughout the project, we closely monitored and measured various performance metrics such as query execution time, resource utilization, and scalability. These metrics allowed us to assess the databases' efficiency in handling large datasets and complex query patterns. Additionally, we considered factors like data integrity, ease of use, and flexibility in the design and implementation of the library management system.

This report presents the findings and analysis of our comparative study, highlighting the strengths and weaknesses of each database type based on their performance and suitability for a library management system. The insights gained from this research will assist developers, system architects, and database administrators in making informed decisions when selecting a NoSQL database for similar applications.

Description of the database management system

For our library management system, we opted to develop a NoSQL database using MongoDB, a document-oriented database. The structure of the database revolves around three main collections: Books, Borrowers, and Borrowing History.

The Books collection stores information about each book in the library. Each document represents a book and contains fields such as _id (a unique identifier for the book), title, author, public`tion date. Additionally, we included a borrower_id field to keep track of which borrower currently has the book.
The Borrowers collection stores information about the borrowers (students) in the library. Each document represents a borrower and includes fields like borrower_id (a unique identifier for the borrower), name, and email. To keep track of the books borrowed by a borrower, we included a table named borrowing history.
The Borrowing History collection maintains a record of the borrowing events that have occurred in the library. Each document represents a borrowing event and includes fields such as _id (a unique identifier for the borrowing event), book_id, borrower_id, borrow_date, return_date.
By structuring our NoSQL database in this manner, we can effectively store and manage information about books, borrowers, and their borrowing history. The flexibility of MongoDB allows us to easily add or modify fields based on specific requirements, ensuring the library management system can adapt to evolving needs.

Problem of database management

When managing a library system using different database types, one of the challenges is maintaining data consistency and integrity across multiple databases. Each database type has its own data model and query language, which can lead to inconsistencies and discrepancies in data representation and storage.
For example, if we have a library management system that stores information about books, borrowers, and borrowing history, we may face challenges when synchronizing data between different databases. SQL databases follow a structured schema with predefined relationships, while NoSQL databases like MongoDB, Cassandra, or Neo4j have their own data models and query languages.
If we store data in SQL databases for one part of the system and use NoSQL databases for another, ensuring data consistency becomes complex. Updates or modifications made to data in one database may not be reflected immediately in the other databases, leading to inconsistencies and incorrect information. For example, if a borrower returns a book and the update is only made in the SQL database, the NoSQL databases may still reflect that the book is borrowed.
Furthermore, different database types may have varying performance characteristics and query capabilities. A query that executes efficiently in one database type may perform poorly or require

complex transformations in another. This discrepancy in query performance can impact the overall system's responsiveness and user experience.

Another challenge is the need for expertise in multiple database technologies. Developers and administrators must be well-versed in the intricacies of each database type, including their data models, query languages, and optimization techniques. Maintaining expertise across multiple technologies can be time-consuming and may introduce complexities during development, troubleshooting, and maintenance.

Overall, managing a library system using different database types requires careful consideration of data consistency, query performance, and the expertise needed to maintain and synchronize data across multiple databases. Proper data synchronization mechanisms, data migration strategies, and standardized data access layers can help mitigate these challenges.

Implementation

Big data generation

During the implementation of our library management system and the subsequent performance testing, we encountered a significant challenge in obtaining large datasets of fake data for testing purposes. We initially explored various websites and online resources that offer pre-generated data. However, finding specific datasets with the exact number of records we needed (250,000, 500,000, 750,000, and 1 million) proved to be quite difficult. The available datasets were often insufficient in size or didn't align with the structure required for our books table.

To overcome this hurdle, we decided to take matters into our own hands and developed a Python code to generate random data specifically tailored to our library management system. This approach allowed us to control the generation process and create datasets of the desired size. By implementing a custom data generation script, we were able to ensure the accuracy and consistency of the generated data, specifically referencing the books table within our database.

By creating our own data generation solution, we were able to meet the testing requirements for the different dataset sizes. This allowed us to accurately evaluate the performance of our database systems under realistic and scalable conditions. Generating our own random data using Python not only provided the necessary flexibility but also ensured that the generated datasets were representative of the library management system's data characteristics.

Below is the python code for creation of dummy data for different sizes of datasets.

```
import csv
import random
import os
from faker import Faker
fake = Faker()

# Set the path to the desktop directory
desktop_path = os.path.expanduser("~/Desktop")
print('Desktop path: ', desktop_path)

# Generate unique primary keys
book_id_counter = 1
borrower_id_counter = 1
borrowing_id_counter = 1

# Generate data and save to CSV files
with open(os.path.join(desktop_path, 'books.csv'), 'w', newline='') as books_file, \
      open(os.path.join(desktop_path, 'borrowers.csv'), 'w', newline='') as borrowers_file, \
      open(os.path.join(desktop_path, 'borrowing_history.csv'), 'w', newline='') as borrowing_history_file:

   books_writer = csv.writer(books_file)
   borrowers_writer = csv.writer(borrowers_file)
   borrowing_history_writer = csv.writer(borrowing_history_file)

   # Write headers to CSV files
   books_writer.writerow(['book_id', 'title', 'author', 'publication_date'])
   borrowers_writer.writerow(['borrower_id', 'name', 'email'])
   borrowing_history_writer.writerow(['borrowing_id', 'book_id', 'borrower_id', 'borrow_date', 'return_date'])
   print('Headers written to CSV files')

   # Generate and save book data
   for _ in range(1000000):
      title = fake.catch_phrase()
      author = fake.name()
      publication_date = fake.date_between(start_date='-10y', end_date='today')
      books_writer.writerow([book_id_counter, title, author, publication_date])
      book_id_counter += 1
   print('Book data generated and saved to CSV file')

   # Generate and save borrower data
   for _ in range(950000):
      name = fake.name()
      email = fake.email()
      borrowers_writer.writerow([borrower_id_counter, name, email])
      borrower_id_counter += 1
   print('Borrower data generated and saved to CSV file')

   # Generate and save borrowing history data
   for _ in range(900000):
      book_id = random.randint(1, 900000)
      borrower_id = random.randint(1, 900000)
      borrow_date = fake.date_between(start_date='-1y', end_date='today')
      return_date = fake.date_between(start_date=borrow_date, end_date='today')
      borrowing_history_writer.writerow([borrowing_id_counter, book_id, borrower_id, borrow_date,
return_date])
      borrowing_id_counter += 1
   print('Borrowing history data generated and saved to CSV file')
```

SQL MariaDB

For the SQL database solution, we opted to use MariaDB as our database management system. To interact with the database, we established a connection via the shell terminal, leveraging the command-line interface provided by MariaDB. This allowed us to execute SQL queries and perform various operations directly from the terminal.

To automate the process of including data into the MariaDB database, we developed a Python code using the MariaDB connector library. This code facilitated the insertion of data into the database tables efficiently and accurately. By leveraging the capabilities of Python and the MariaDB connector, we were able to automate the data inclusion process, saving time and effort in manually entering large datasets. This automation ensured consistency and integrity in the data population process, enabling us to focus on the performance testing and analysis of the SQL database solution.

```
import csv
import pymysql
# Connect to MySQL database
conn = pymysql.connect(host='localhost', user='root', password='', db='library1m')
cursor = conn.cursor()
print("Connected to MySQL database!")

# Create temporary database
cursor.execute("CREATE DATABASE IF NOT EXISTS library1m")
cursor.execute("USE library1m")

# Create books table
cursor.execute('''CREATE TABLE IF NOT EXISTS books (
                book_id INT PRIMARY KEY,
                title VARCHAR(255),
                author VARCHAR(255),
                publication_date DATE
            )''')

# Create borrowers table
cursor.execute('''CREATE TABLE IF NOT EXISTS borrowers (
                borrower_id INT PRIMARY KEY,
                name VARCHAR(255),
                email VARCHAR(255)
            )''')

# Create borrowing_history table
cursor.execute('''CREATE TABLE IF NOT EXISTS borrowing_history (
                borrowing_id INT PRIMARY KEY,
                book_id INT,
                borrower_id INT,
                borrow_date DATE,
                return_date DATE,
                FOREIGN KEY (book_id) REFERENCES books(book_id),
                FOREIGN KEY (borrower_id) REFERENCES borrowers(borrower_id)
            )''')

# Insert data from books.csv
with open('/Users/kila/Desktop/books.csv', 'r') as file:
    csv_data = csv.reader(file)
    next(csv_data)  # Skip header row
    for row in csv_data:
        book_id, title, author, publication_date = row
        query = f"INSERT INTO books (book_id, title, author, publication_date) VALUES ({book_id}, '{title}', '{author}',
'{publication_date}')"
        cursor.execute(query)
```

Hence four queries of increasing complexity are constructed as follows:

```
Simple:
   SELECT * FROM books;

Intermediate:
   SELECT title, author, publication_date
   FROM books
   WHERE publication_date LIKE '2021%';

Complex:
   SELECT b.title, b.author, bo.name
   FROM books b
   JOIN borrowing_history bh ON b.book_id = bh.book_id
   JOIN borrowers bo ON bo.borrower_id = bh.borrower_id
   WHERE DATE_FORMAT(bh.borrow_date, '%Y') = '2023';

Advanced:
  SELECT b.title, b.author, bh.borrow_date
   FROM books b
   JOIN borrowing_history bh ON b.book_id = bh.book_id
   WHERE YEAR(bh.borrow_date) = 2022
   GROUP BY b.title, b.author, bh.borrow_date
   HAVING COUNT(*) > (
      SELECT AVG(borrows_per_book)
      FROM (
         SELECT COUNT(*) AS borrows_per_book
         FROM borrowing_history
         WHERE YEAR(borrow_date) = 2022
         GROUP BY book_id
      ) AS subquery
   )
   AND b.author = 'Rebecca Patterson'
   ORDER BY bh.borrow_date DESC;
```

Function of each query:

Simple:

The simple query, "SELECT * FROM books;", retrieves all the data from the "books" table. It returns a complete list of all books in the library, including all their attributes and fields.

Intermediate:

The intermediate query, "SELECT title, author, publication_date FROM books WHERE publication_date LIKE '2021%';", retrieves the title, author, and publication date of books from the "books" table. It filters the results based on books that have a publication date in the year 2021.

Complex:

The complex query, "SELECT b.title, b.author, bo.name FROM books b JOIN borrowing_history bh ON b.book_id = bh.book_id JOIN borrowers bo ON bo.borrower_id = bh.borrower_id WHERE DATE_FORMAT(bh.borrow_date, '%Y') = '2023';", retrieves the title and author of books from the "books" table along with the name of the borrowers who borrowed those books. It joins the

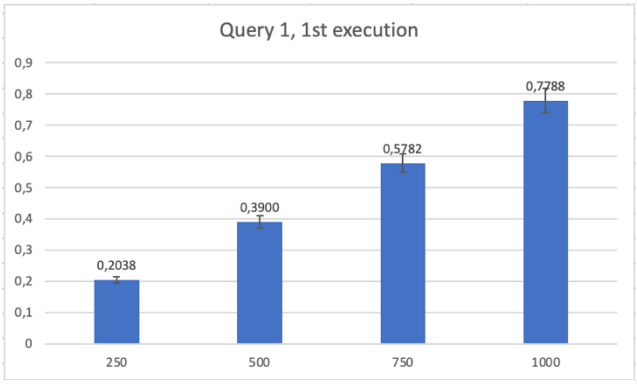
"books," "borrowing_history," and "borrowers" tables based on their respective IDs and filters the results to include only books that were borrowed in the year 2023.

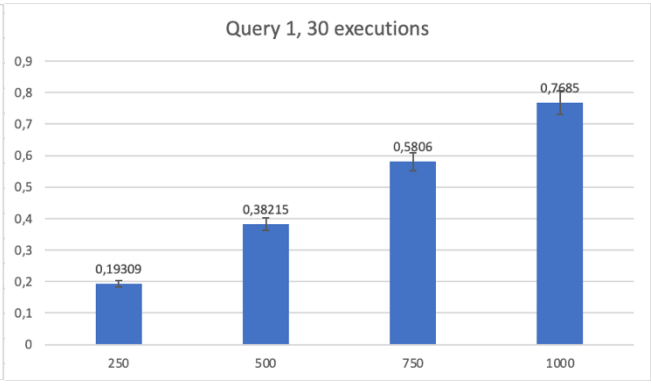
Advanced:

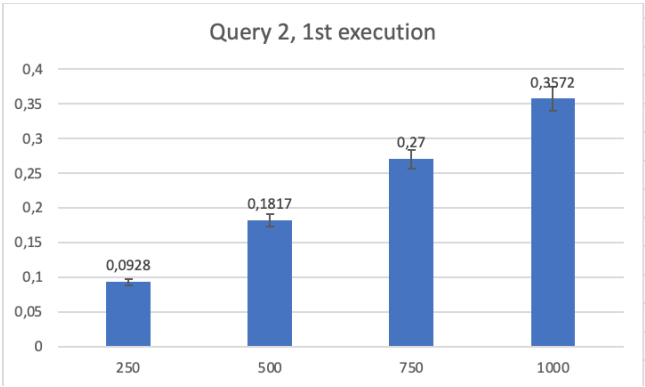
The advanced query is a complex and intricate one. It retrieves the title, author, and borrow date of books from the "books" table, joined with the "borrowing_history" table. It includes additional conditions and calculations. The query filters the results to include only books borrowed in the year 2022 and groups them by title, author, and borrow date. It then applies a HAVING clause to include only books that were borrowed more times than the average number of borrows per book in 2022. Finally, it further narrows down the results to books by a specific author (Rebecca Patterson) and sorts them in descending order based on the borrow date.

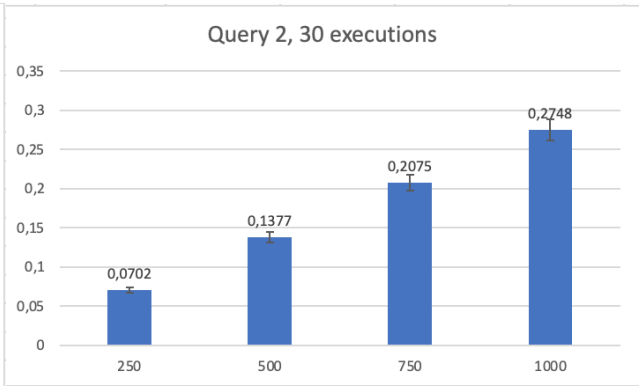
A runtime checker.py code is created to record the time it takes for 31 executions per each query and per each dataset size of the SQL database. The results are presented in the form of histograms.

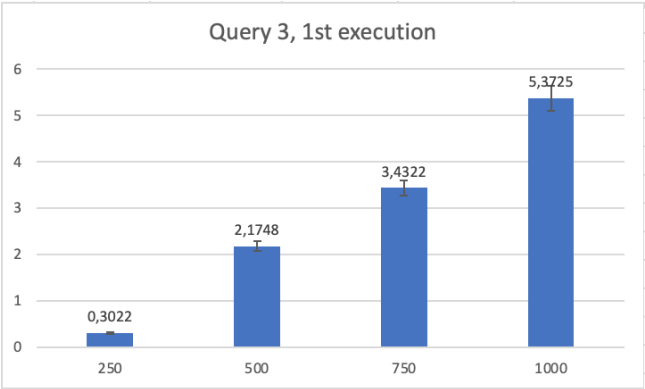
The results:

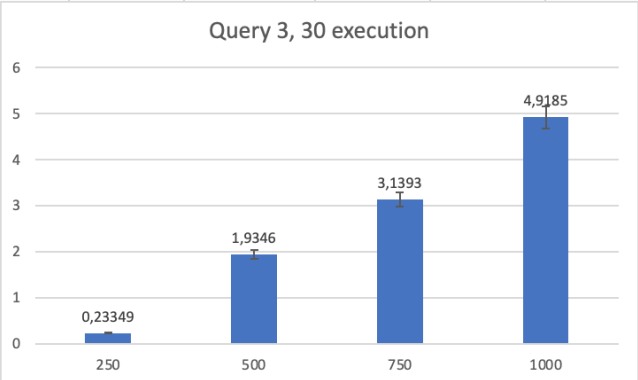


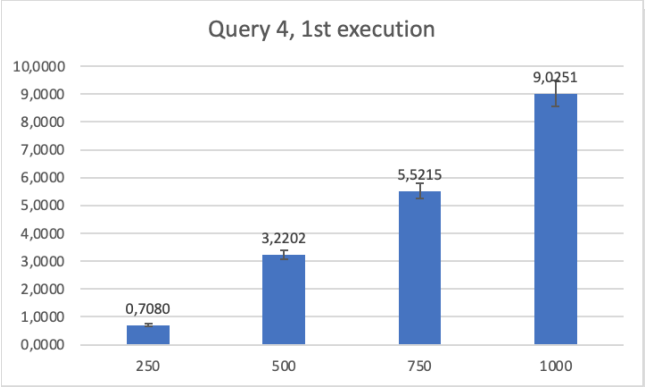


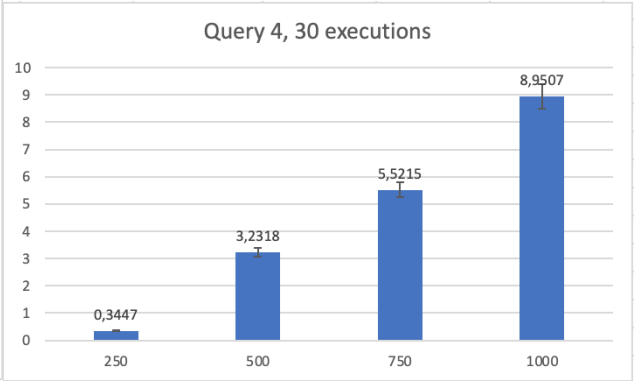












From the histograms above, it is clear that the increase in size of the dataset results in the increase of the execution time. Moreover, the complexity of the queries also results in the increase of the execution time.

Cassandra DB

For the Cassandra database, we established a connection by utilizing a Docker container and pulling the Cassandra database image. This allowed us to create a containerized environment for running Cassandra. With the container up and running, we proceeded to upload the necessary data for our library management system.

To upload the data, we executed the required commands using the Cassandra Query Language (CQL). We utilized CQL statements to define and create the appropriate tables, ensuring they aligned with the data structure required for our library management system. Once the tables were in place, we used a Python code to insert the data into the Cassandra database, leveraging the Cassandra driver for Python.

Executing the queries involved utilizing CQL to formulate the required statements. The queries were designed to retrieve specific information from the Cassandra database based on the given requirements. These queries encompassed different levels of complexity, including simple, intermediate, complex, and advanced queries, allowing us to assess the performance and capabilities of Cassandra in handling various query scenarios.

By connecting to Cassandra through the Docker container, uploading the data, and executing the queries, we were able to evaluate the functionality and performance of Cassandra as a database solution for our library management system.

The queries are as follows:

```
Easy:
  SELECT * FROM books;

Intermediate:
   SELECT title, author, publication_date
   FROM books
   WHERE publication_date >= '2021-01-01' AND publication_date < '2022-01-01'
   ALLOW FILTERING;

Complex:
# Query: Retrieve borrowing_history for a specific year
      borrowing_history_query = """
         SELECT book_id, borrower_id, borrow_date
         FROM borrowing_history
         WHERE borrow_date >= '2023-01-01' AND borrow_date < '2024-01-01'
         ALLOW FILTERING
      """
# Query: Retrieve books for the retrieved book IDs
      books_query = """
         SELECT book_id, title, author
         FROM books
         WHERE book_id IN {}
         ALLOW FILTERING
      """
# Query: Retrieve borrowers for the retrieved borrower IDs
      borrowers_query = """
         SELECT borrower_id, name
         FROM borrowers
         WHERE borrower_id IN {}
         ALLOW FILTERING
      """

Advanced:
# Query 1: Retrieve book IDs and author for a specific author
      author_books_query = """
         SELECT book_id, author
         FROM books
         WHERE author = 'Rebecca Patterson'
         ALLOW FILTERING
      """
# Query 2: Retrieve borrowing history for the retrieved book IDs in the specified year
      borrowing_history_query = """
         SELECT book_id, borrow_date
         FROM borrowing_history
         WHERE book_id IN {}
         AND borrow_date >= '2022-01-01' AND borrow_date < '2023-01-01'
         ALLOW FILTERING
      """
# Query 3: Retrieve book titles for the retrieved book IDs
      books_query = """
         SELECT book_id, title
         FROM books
         WHERE book_id IN {}
         ALLOW FILTERING
      """
```

The results:

Once again, we see that the time results increase as the dataset size increases. Moreover, the more complex is the query, the more time it takes to be executed.

MongoDB

For 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:

```
db.books.find({})
```

Intermediate:

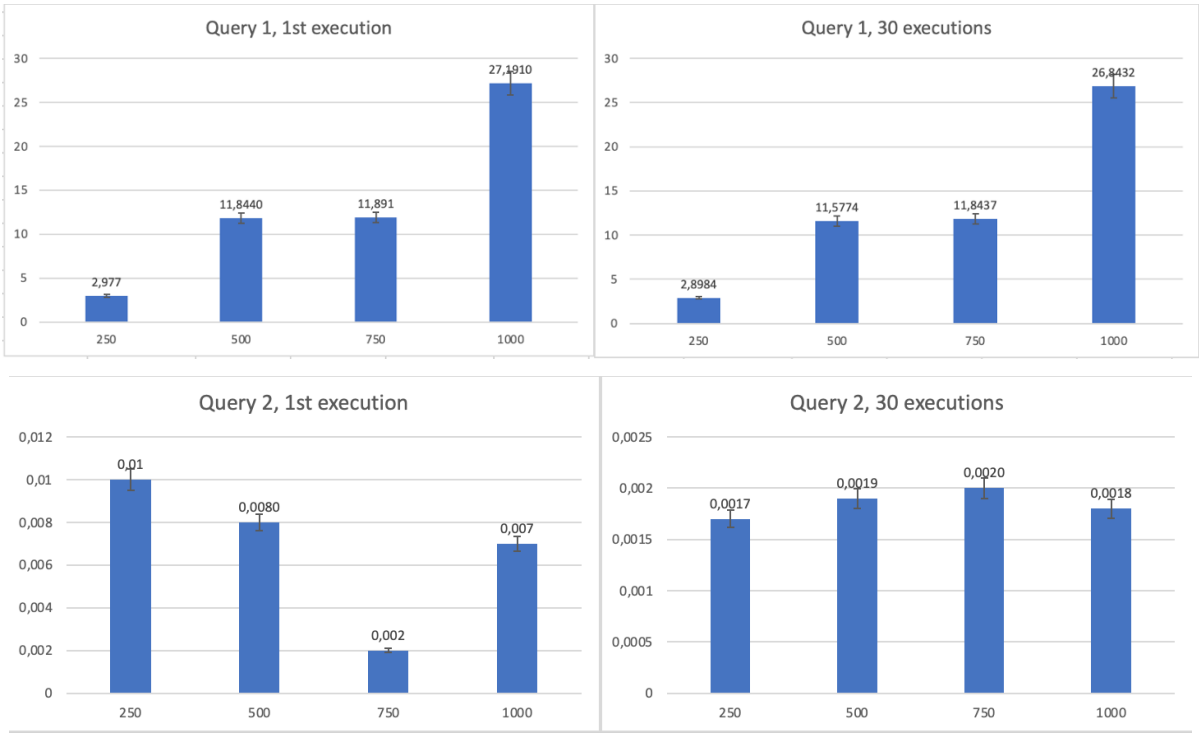
```
pipeline = [
  {
    "$match": {
      "publication_date": {
        "$regex": "^2021"
      }
    }
  },
  {
    "$project": {
      "_id": 0,
      "title": 1,
      "author": 1,
      "publication_date": 1
    }
  }
]
```

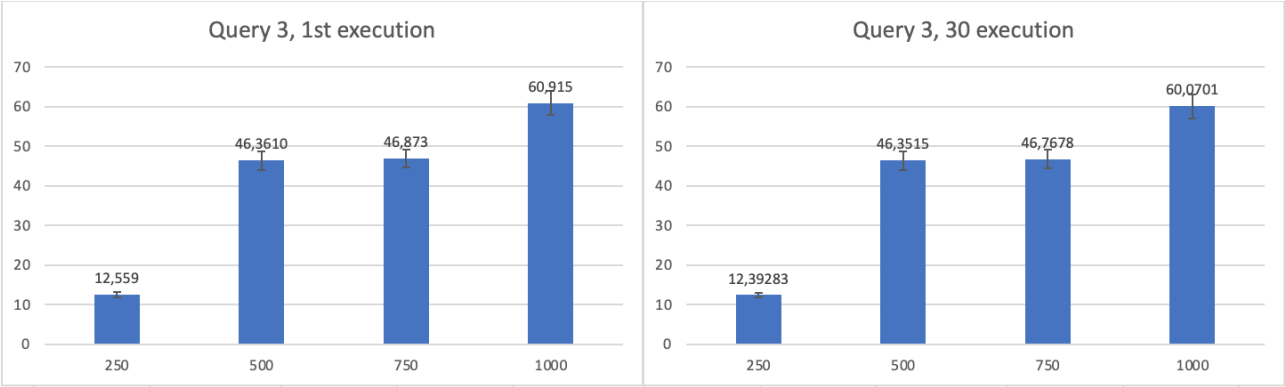
Complex:

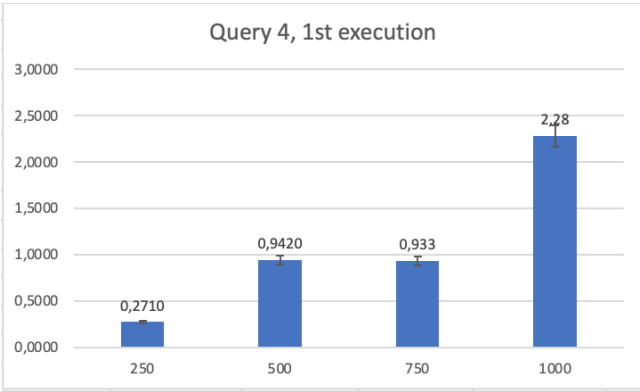
```
pipeline = [
  {
    "$lookup": {
      "from": "borrowing_history",
      "localField": "book_id",
      "foreignField": "book_id",
      "as": "borrowing"
    }
  },
  {
    "$unwind": "$borrowing"
  },
  {
    "$lookup": {
      "from": "borrowers",
      "localField": "borrowing.borrower_id",
      "foreignField": "borrower_id",
      "as": "borrower"
    }
  },
  {
    "$unwind": "$borrower"
  },
  {
    "$match": {
      "$expr": {
        "$eq": [
          { "$substr": ["$borrowing.borrow_date", 0, 4] },
          "2023"
        ]
      }
    }
  },
```

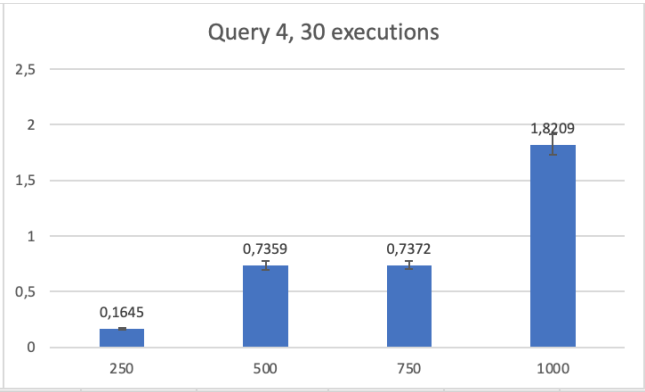
```
{
    "$project": {
      "_id": 0,
      "title": 1,
      "author": 1,
      "name": "$borrower.name"
    }
  }
]

Advanced:
 pipeline = [
    {
      "$lookup": {
        "from": "borrowing_history",
        "localField": "book_id",
        "foreignField": "book_id",
        "as": "borrowing_history"
      }
    },
    {
      "$match": {
        "borrowing_history.borrow_year": 2022,
        "author": "Rebecca Patterson"
      }
    },
    {
      "$group": {
        "_id": "$_id",
        "title": {"$first": "$title"},
        "author": {"$first": "$author"},
        "borrow_date": {"$first": "$borrowing_history.borrow_date"}
      }
    },
    {
      "$sort": {"borrow_date": -1}
    }
]
```

The results are as follows:









The results are similar in terms of pattern to the previous databases. However, we see that MongoDB takes much more time executing the queries than other databases.

RedisDB

For Redis database we used the same approach of using the Docker container and pulling the image and then connecting through the port.

The queries are as follows:

```
Simple:
    # Retrieve all books
    book_keys = r.keys('book:*')
    for key in book_keys:
        book = r.hgetall(key)
        print(book)
```

Intermediate:

```
    # Query: Retrieve books based on publication_date pattern
    book_keys = r.keys('book:*')
    for key in book_keys:
        book_data = r.hgetall(key)
        if book_data:
            publication_date = book_data.get(b'publication_date', b'').decode()
            if publication_date.startswith('2021'):
                title = book_data.get(b'title', b'').decode()
                author = book_data.get(b'author', b'').decode()
```

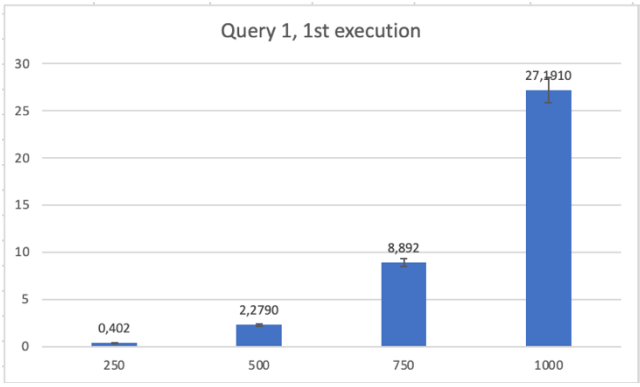
Complex:

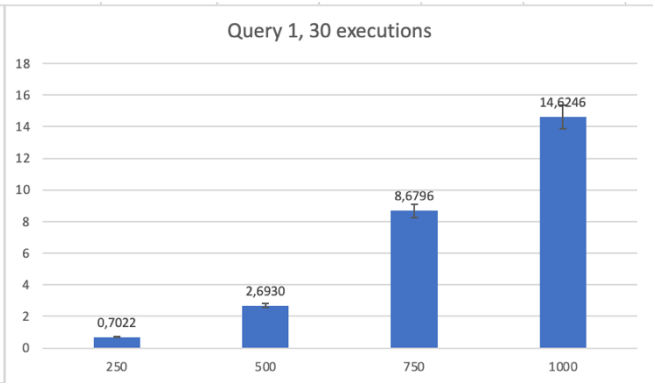
```
    # Retrieve borrowing history IDs with filtering
    borrowing_keys = r.keys("borrowing_history:*")
    #print(borrowing_keys)

    for key in borrowing_keys:
        borrowing = r.hgetall(key)
        borrow_date = borrowing[b'borrow_date'].decode('utf-8')

        # Filter by borrow_date
        if borrow_date.startswith('2023'):
            borrowing_id = key.decode('utf-8').split(':')[1]
            book_key = "book:" + borrowing_id
            borrower_key = "borrower:" + borrowing_id

            # Retrieve book and borrower data
            book = r.hgetall(book_key)
            borrower = r.hgetall(borrower_key)
```

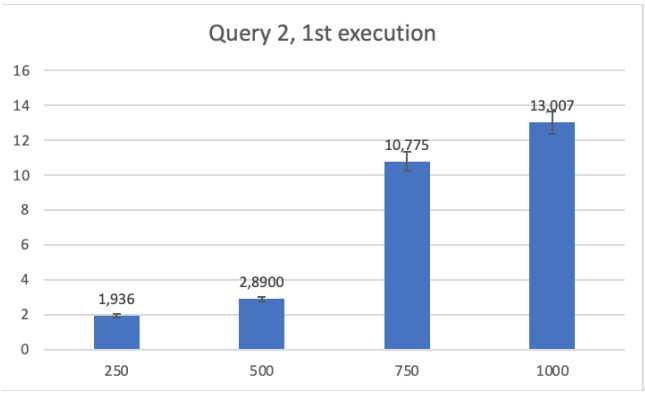
Advanced:

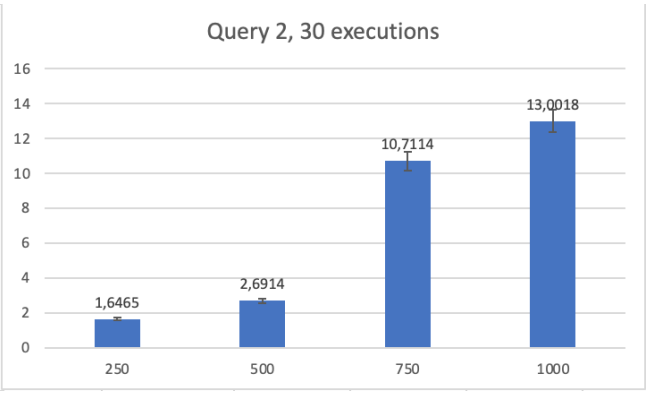
```
# Step 1: Fetch all book IDs
book_ids = []
books = r.keys("book:*")
for book_key in books:
    book = r.hgetall(book_key)
    if b'book_id' in book:
        book_id = book[b'book_id'].decode('utf-8')
        book_ids.append(book_id)

# Step 2: Filter books by author and borrowing year
for book_id in book_ids:
    book_key = f"book:{book_id}"
    book = r.hgetall(book_key)
    if book.get(b'author', b'').decode('utf-8') == 'Rebecca Patterson':
        borrowing_key = f"borrowing_history:{book_id}"
        borrowings = r.smembers(borrowing_key)
        for borrowing in borrowings:
            borrow_data = r.hgetall(borrowing)
            borrow_date = borrow_data.get(b'borrow_date', b'').decode('utf-8')
            if borrow_date.startswith('2022'):
                result = {
                    'title': book.get(b'title', b'').decode('utf-8'),
                    'author': book.get(b'author', b'').decode('utf-8'),
                    'borrow_date': borrow_date
                }
                print(result)

# Step 3: Compute average borrows per book
borrowing_key = f"borrowing_history:{book_id}"
borrowings = r.smembers(borrowing_key)
borrow_count = len(borrowings)
avg_borrows_per_book = sum(len(r.smembers(f"borrowing_history:{other_book_id}")) for other_book_id in book_ids) /
len(book_ids)

# Step 4: Filter books with borrow count greater than average
if borrow_count > avg_borrows_per_book:
    result = {
        'title': book.get(b'title', b'').decode('utf-8'),
        'author': book.get(b'author', b'').decode('utf-8'),
        'borrow_date': borrow_date
    }
    print(result)
```

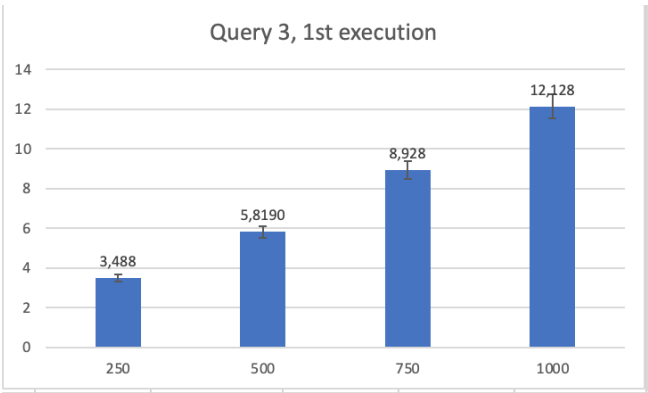
The results:

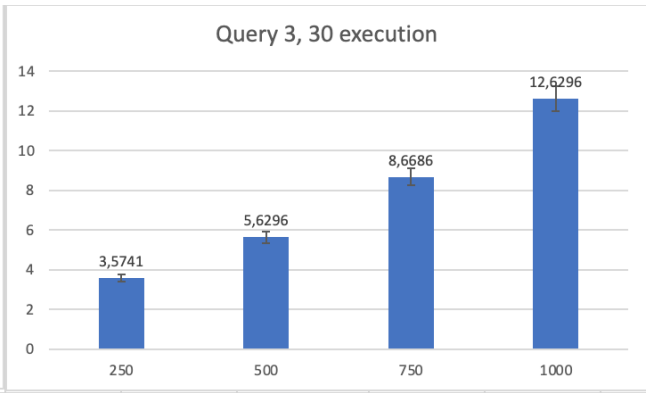


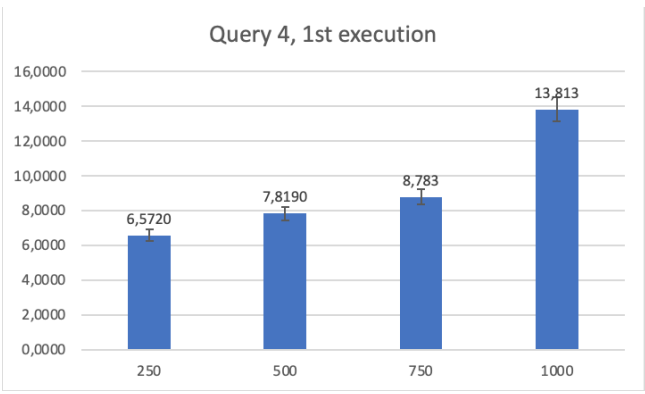


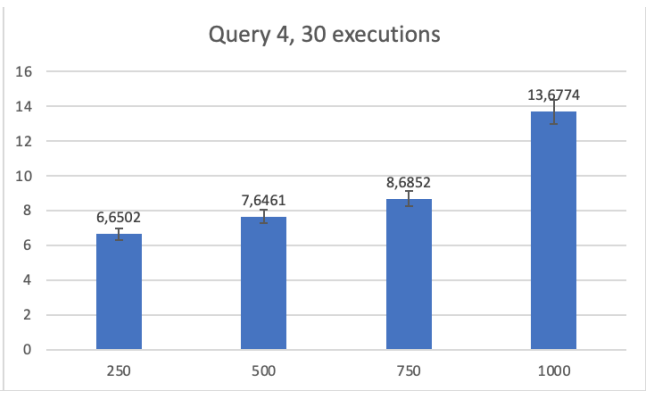












The Redis database shows the same tendency as all the databases with respect to dataset size and query complexity variations along with time. But one thing that we notice that this is one of the fastest query execution times.

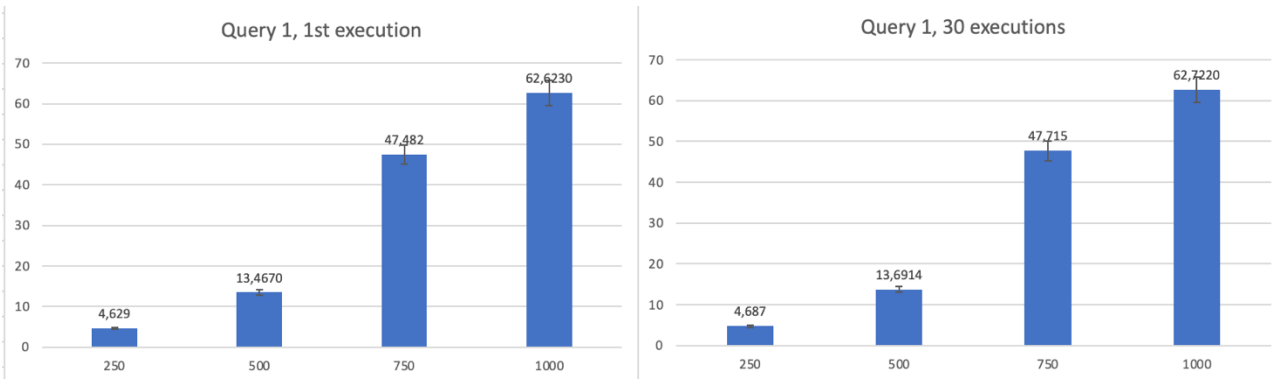
Neo4jDB

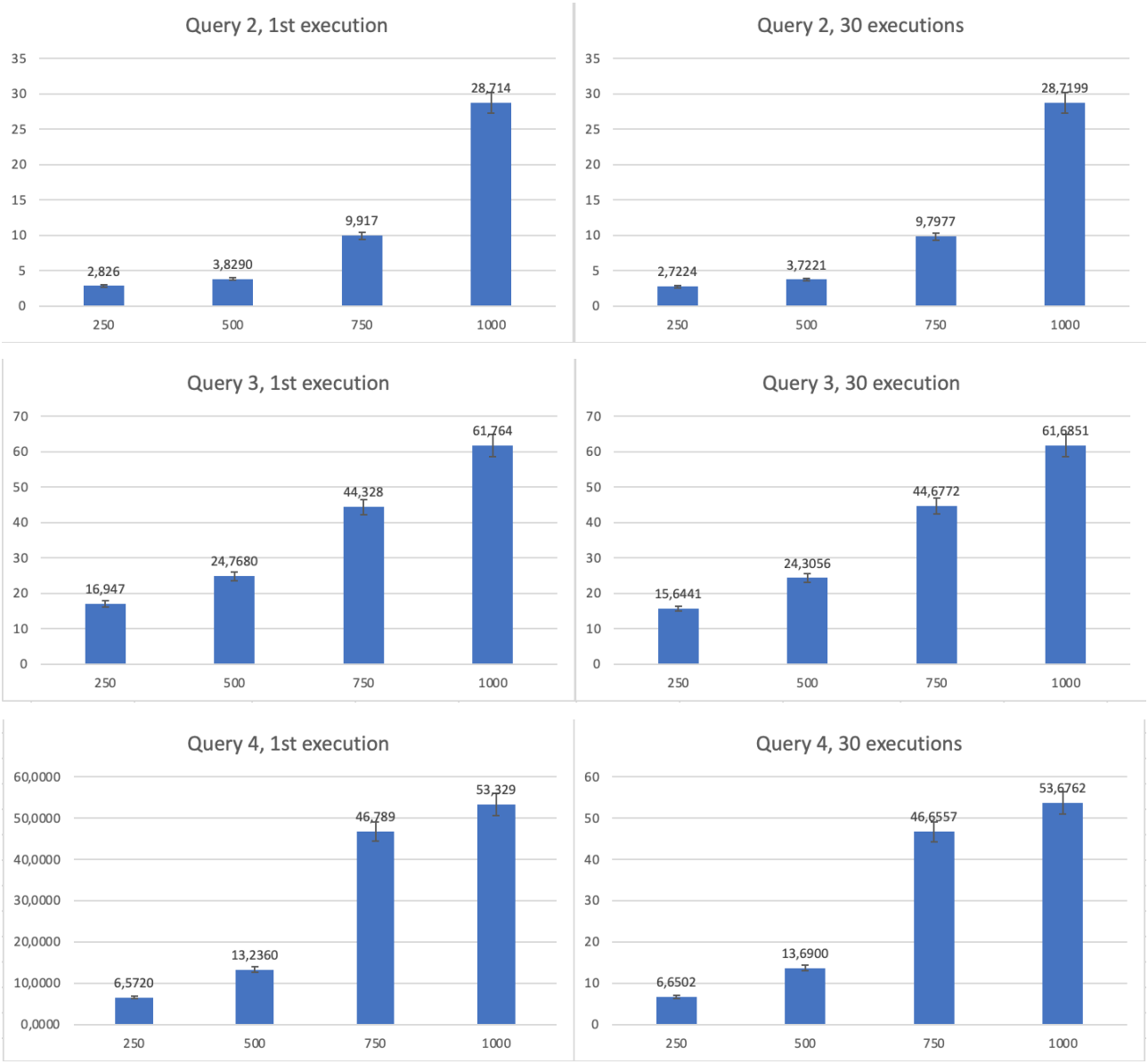
To connect to a Neo4j database using a Docker container, you need to pull the Neo4j image from Docker Hub and run a container based on that image. By mapping the container ports to the corresponding host machine ports, you can access the Neo4j browser interface through your web browser. Once connected, you can set a password and start executing queries and exploring the functionalities of Neo4j within the containerized environment. This approach provides a convenient and isolated way to work with Neo4j databases.

The queries:

```
Easy:
   MATCH (b:Book) RETURN b

Intermediate:
   MATCH (b:Book)
   WHERE b.publication_date STARTS WITH '2021'
   RETURN b.title, b.author, b.publication_date;

Complex:
   MATCH (b:Book), (bh:BorrowingHistory), (bo:Borrower)
   WHERE b.book_id = bh.book_id
   AND bo.borrower_id = bh.borrower_id
   AND substring(bh.borrow_date, 0, 4) = '2023'
   RETURN b.title, b.author, bo.name;

Advanced:
   MATCH (bh:BorrowingHistory)
   WHERE substring(bh.borrow_date, 0, 4) = '2022'
   WITH collect(DISTINCT bh.book_id) AS book_ids, collect(DISTINCT bh.borrow_date) AS borrow_dates


   MATCH (b:Book)
   WHERE b.author = 'Rebecca Patterson'
   RETURN b.book_id AS book_id, b.title AS title, b.author AS author


   WITH book_ids, borrow_dates
   MATCH (bo:Borrower)
   WHERE EXISTS((bo)-[:BORROWED]->(:BorrowingHistory {book_id: book_ids[0]}))
   RETURN title, author, borrow_dates[0] AS borrow_date, bo.name AS borrower_name;
```

The results:





The results demonstrate the same tendency of time variation with dataset size change and increase in query complexity. One thing to notice is that the Neo4j database takes more time to execute the queries for the library management system in comparison to other databases tested.

Conclusion

In conclusion, after implementing and testing the library management system across five different databases (SQL, Cassandra, MongoDB, RedisDB, and Neo4j), we can observe variations in efficiency and performance among them.

Based on our analysis, RedisDB emerges as the most efficient database for the library management system. Its in-memory storage and low-latency characteristics make it ideal for handling high-performance data operations. RedisDB excels in scenarios that require quick data retrieval and caching, providing excellent responsiveness.

Cassandra follows closely as the second most efficient database for the library management system. Its ability to handle large amounts of data across multiple servers ensures scalability and high write performance. Although its read performance may be slightly slower than RedisDB, it offers strong data durability and fault tolerance.

SQL databases, represented by MariaDB in our implementation, demonstrate reliable performance and transactional capabilities. While they may not match the raw speed of RedisDB and Cassandra for certain use cases, they provide robust performance and support a wide range of queries and operations. MariaDB proves to be a dependable choice, particularly when data consistency and integrity are essential.

MongoDB showcases good performance, but it is slightly slower compared to the aforementioned databases due to its focus on flexibility and query capabilities. Its document-oriented nature and ease of use make it suitable for various applications, including the library management system.

Neo4j, being a graph database, performs relatively slower than the other databases in our implementation. Its strength lies in managing complex relationships, but this comes at the cost of reduced raw performance compared to other types of databases. Neo4j is better suited for scenarios that heavily rely on graph traversals and analysis.

In summary, the order from fastest to slowest for the library management system, based on our analysis, is as follows: RedisDB, Cassandra, SQL (MariaDB), MongoDB, and Neo4j. It is important to note that these rankings may vary depending on the specific use case, dataset size, and optimization techniques employed. The selection of the most suitable database will depend on the specific requirements and trade-offs relevant to the library management system.