UNIVERSITÀ STUDI DI MESSINA

MIFT DEPARTMENT

DATA ANALYSIS PROGRAM

Database Mod B

Comparison of Databases

Professor:
Armando Ruggeri

Student:
ASSAD ULLAH
TALHA MANSOOR QURESHI

ID: ***533325***

536138

ACADEMIC YEAR – 2022/2023

INTRODUCTION

There are several difficulties that frequently arise in the field of library management systems. The use of antiquated cataloguing methods by libraries is a widespread problem that makes effective book management and searches difficult. Adopting a contemporary library management system that has sophisticated cataloguing and search capabilities and gives users a simpler method to find resources is one potential answer. Accessibility is another issue, especially if resources are limited to in-person visits. Libraries should think about investing in web portals that enable remote access to digital resources like e-books and journals in order to solve this.

Another ongoing issue that affects inventories and operations is late or missing returns. Automated due date reminders and the implementation of digital lending platforms for e-books might be the solution in this situation. In the meanwhile, slow checkout procedures may make customers angry. The introduction of smartphone apps and self-checkout kiosks simplifies the borrowing procedure.

Furthermore, given the size of library holdings, efficient resource management might be tricky. Comprehensive resource tracking is made possible by using library management software.

The purpose of this report is to present a comprehensive comparison of various database systems for a Library Management System. The database systems under consideration are NoSQL databases including Redis, Cassandra, Neo4j, MongoDB, and the traditional relational database MySQL. This comparison aims to evaluate the performance, scalability, flexibility, and query capabilities of each database system in the context of a library management system.

For this comparison, We utilized a dataset representative of a library management system. The dataset includes information about books, authors, borrowers, transactions, We will produce numerous datasets of varying sizes, spanning from 250,000 to 1,000,000 records, to ensure accurate and thorough analysis. These datasets will be added to the selected databases and will have similar informational content. We may evaluate each solution's efficiency and efficacy by contrasting the results of queries run on these databases.

In addition, we'll create and run a series of increasingly sophisticated queries to assess how well the databases work. These queries will use various entities and selection filters, giving us a better understanding of how each database manages more complex processes. In order to obtain reliable statistics, we will automate the experiments and measure query execution times. Each test will be run at least 30 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.

Database Comparison

Redis

- Redis is an in-memory data store known for its lightning-fast read and write operations.
- It excels in caching and high-speed data retrieval.
- Well-suited for scenarios requiring rapid access to frequently used data.
- Redis stores data in key-value pairs and supports various data structures, such as strings, lists, sets, and sorted sets.

Cassandra

- Cassandra is a distributed NoSQL database designed for high availability and scalability.
- Suitable for handling large amounts of data across multiple nodes.
- Well-suited for write-intensive applications but may involve more complexity in query design.
- Offers tunable consistency levels for balancing performance and consistency.

Neo4j

- Neo4j is a graph database that excels in managing highly interconnected data.
- Ideal for scenarios involving complex relationships, such as social networks or recommendation systems.
- Graph-based queries are expressive and efficient for traversing relationships.
- May not be as performant for purely tabular data compared to other NoSQL databases.

MongoDB

- MongoDB is a popular document-oriented database.
- Flexible schema allows easy adaptation to evolving data structures.
- Suitable for scenarios requiring dynamic and unstructured data.
- Performs well for simple to moderately complex queries but might face challenges with extremely complex joins.

MySQL

- MySQL is a traditional relational database with a structured schema.
- Well-suited for scenarios involving structured data with well-defined relationships.
- Supports complex queries, aggregations, and joins.
- May require more effort for horizontal scalability compared to some NoSQL databases.

SCHEMAS AND INSERTING DATA

Redis:

```
# Insert CSV data into Redis
filename = 'library_dataset4.csv'

with open(filename, 'r') as file:
    csv_data = csv.reader(file)

    # Skip header row if present
    next(csv_data)

    # Iterate over the rows and insert the data into Redis
    for row in csv_data:

        if len(row) != 9:
            continue
        # Skip rows with incorrect number of values

        title, author, isbn, available, borrower_name, borrower_email, borrower_phone, borrow_date, return_date = row
        r.hset(f'book:{isbn}', 'Book Title', title)
        r.hset(f'book:{isbn}', 'Author', author)
        r.hset(f'book:{isbn}', 'ISBN', isbn)
        r.hset(f'book:{isbn}', 'Available', available)
        r.hset(f'book:{isbn}', 'Borrower Name', borrower_name)
        r.hset(f'book:{isbn}', 'Email', borrower_email)
        r.hset(f'book:{isbn}', 'Phone', borrower_phone)
        r.hset(f'book:{isbn}', 'Borrow Date', borrow_date)
        r.hset(f'book:{isbn}', 'Return Date', return_date)
```

In this code, each book is stored as a Redis Hash with the ISBN as the key. The book's attributes are stored as fields within the Hash. This allows you to efficiently retrieve and manage individual book records using Redis's fast key-value store.

Cassandra

```
# Create a Cassandra cluster
cluster = Cluster(['localhost'], auth_provider=PlainTextAuthProvider(
    username='username', password='password'))

# Connect to the Cassandra session
session = cluster.connect()


session.execute(
    "CREATE KEYSPACE IF NOT EXISTS library WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 }")

session.set_keyspace('library')

# Create a table for books
session.execute(
    """
    CREATE TABLE IF NOT EXISTS books (
        title text,
        author text,
        isbn text,
        available boolean,
        borrower_name text,
        email text,
        phone text,
        borrow_date timestamp,
        return_date timestamp,
        PRIMARY KEY (isbn)
    )
    """
```

- The script executes a CQL query to create a table named "books" within the "library" keyspace.
- The table has columns for book information such as title, author, ISBN, availability, borrower name, email, phone, borrow date, and return date.
- The primary key is set to the ISBN, allowing efficient data retrieval based on ISBN values.

Inserting data:

```
# Read the dataset CSV file
filename = 'library_dataset.csv'

with open(filename, 'r') as file:
    csv_data = csv.reader(file)
    next(csv_data)  # Skip header row if present

    # Iterate over the rows and insert the data into Cassandra
    for row in csv_data:
        title, author, isbn, available, borrower_name, email, phone, borrow_date, return_date = row

        available = True if available.lower() == 'true' else False

        session.execute(
            """
            INSERT INTO books (title, author, isbn, available, borrower_name, email, phone, borrow_date, return_date)
            VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)
            """,
            (title, author, isbn, available, borrower_name,
             email, phone, borrow_date, return_date)
        )
```

- The script reads data from a CSV file named "library_dataset.csv".
- It iterates through the rows of the CSV file and extracts values for each column.
- The **INSERT INTO** CQL query is used to insert the extracted values into the "books" table

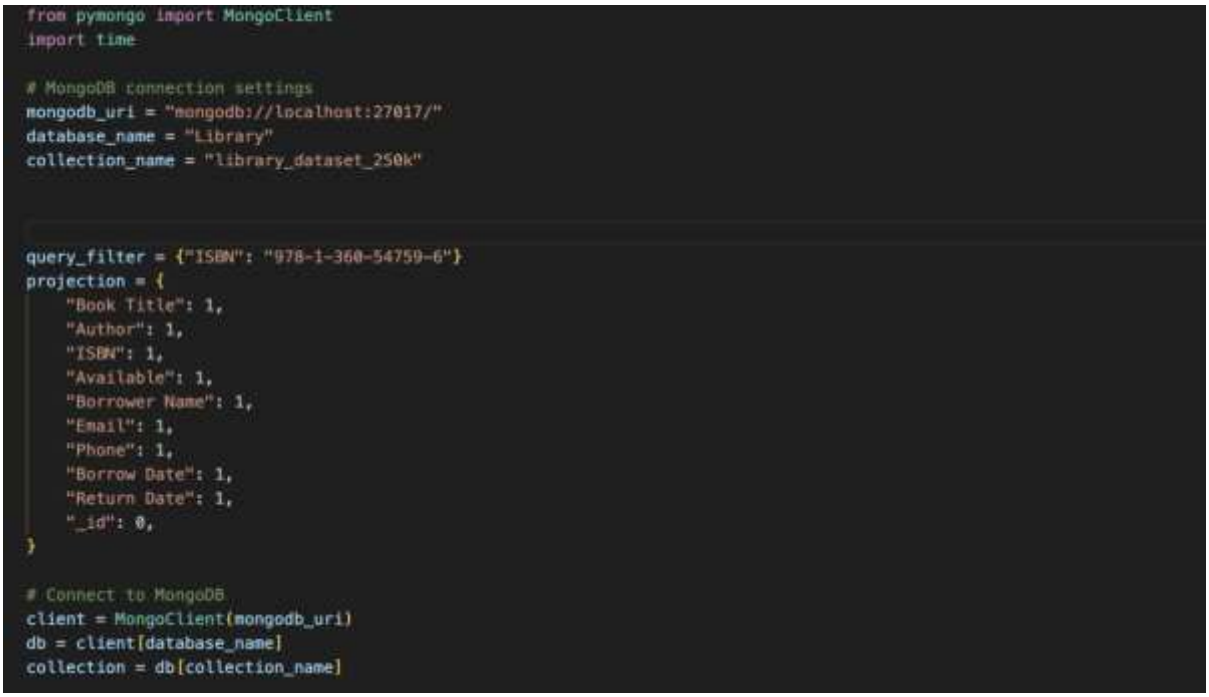
NEO4J

```
# Connect to your Neo4j instance
uri = "bolt://localhost:7687"
username = "neo4j"
password = "password"

graph = Graph(uri, auth=(username, password))

# Load CSV data into Neo4j
filename = 'library_dataset4.csv'


def create_or_update_book_node(row):
    book_node = Node("Book", ISBN=row['ISBN'])
    book_node['Book Title'] = row['Book Title']
    book_node['Author'] = row['Author']
    book_node['Available'] = row['Available']
    book_node['Borrower Name'] = row['Borrower Name']
    book_node['Email'] = row['Email']
    book_node['Phone'] = row['Phone']
    book_node['Borrow Date'] = row['Borrow Date']
    book_node['Return Date'] = row['Return Date']

    existing_book = graph.nodes.match("Book", ISBN=row['ISBN']).first()
    if existing_book:
        existing_book.update(book_node)
        return existing_book
    else:
        graph.create(book_node)
        return book_node
```

The function create_or_update_book_node(row) takes a row of data (represented as a dictionary) and creates or updates a corresponding Neo4j Node representing a book.
It sets properties of the book node based on the values in the CSV row.
The function first checks if a book node with the same ISBN already exists in the graph database. If it does, it updates the existing node's properties. If not, it creates a new book node.

MONGO DB

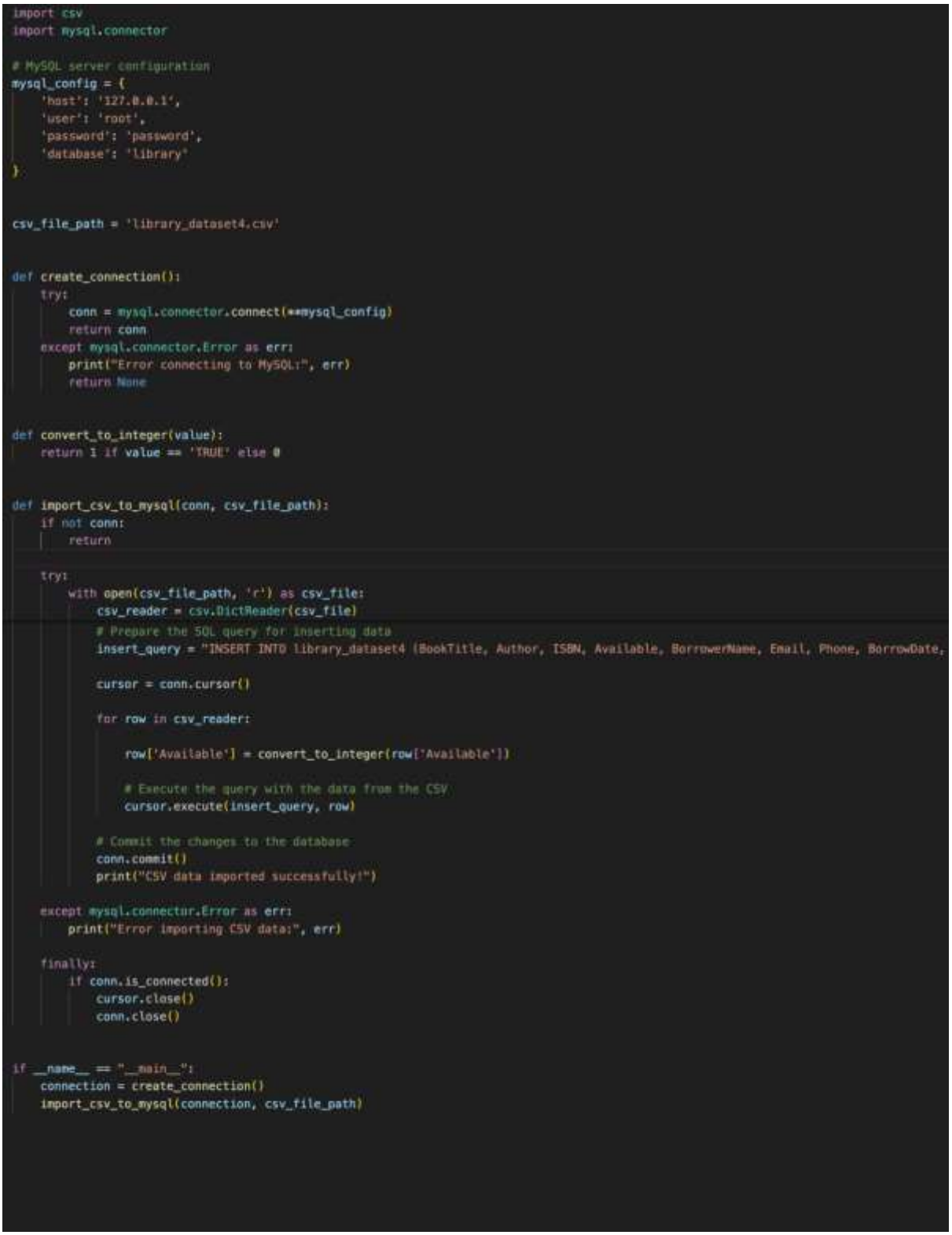
To utilize MongoDB, you'll need to acquire the MongoDB Community Server and MongoDB Compass. Once the MongoDB server is configured, you can launch MongoDB Compass to establish both the database and collection. After that you can import the csv file and perform query directly on the dataset after establishing the connection. Here is a snapshot



```
from pymongo import MongoClient
import time

# MongoDB connection settings
mongodb_uri = "mongodb://localhost:27017/"
database_name = "Library"
collection_name = "library_dataset_250k"


query_filter = {"ISBN": "978-1-360-54759-6"}
projection = {
    "Book Title": 1,
    "Author": 1,
    "ISBN": 1,
    "Available": 1,
    "Borrower Name": 1,
    "Email": 1,
    "Phone": 1,
    "Borrow Date": 1,
    "Return Date": 1,
    "_id": 0,
}

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]
```

MYSQL

To work with MySQL, you'll need to obtain MySQL Workbench. The choice of using MySQL is subjective; I personally opted for it due to its user-friendly nature. Once the connection is set up, you can import datasets directly. However, I chose to utilize VSCode and here's a visual representation of that.



```
import csv
import mysql.connector

# MySQL server configuration
mysql_config = {
    'host': '127.0.0.1',
    'user': 'root',
    'password': 'password',
    'database': 'library'
}


csv_file_path = 'library_dataset4.csv'


def create_connection():
    try:
        conn = mysql.connector.connect(**mysql_config)
        return conn
    except mysql.connector.Error as err:
        print("Error connecting to MySQL:", err)
        return None


def convert_to_integer(value):
    return 1 if value == 'TRUE' else 0


def import_csv_to_mysql(conn, csv_file_path):
    if not conn:
        return

    try:
        with open(csv_file_path, 'r') as csv_file:
            csv_reader = csv.DictReader(csv_file)
            # Prepare the SQL query for inserting data
            insert_query = "INSERT INTO library_dataset4 (BookTitle, Author, ISBN, Available, BorrowerName, Email, Phone, BorrowDate,

            cursor = conn.cursor()

            for row in csv_reader:

                row['Available'] = convert_to_integer(row['Available'])

                # Execute the query with the data from the CSV
                cursor.execute(insert_query, row)

            # Commit the changes to the database
            conn.commit()
            print("CSV data imported successfully!")

    except mysql.connector.Error as err:
        print("Error importing CSV data:", err)

    finally:
        if conn.is_connected():
            cursor.close()
            conn.close()


if __name__ == "__main__":
    connection = create_connection()
    import_csv_to_mysql(connection, csv_file_path)
```

Execution

DATA GENERATION

Certainly, I previously discussed the process of data insertion and executing queries. However, it's essential to verify whether the obtained data is authentic or generated synthetically. Given that our objective is to compare databases, we require diverse sets of simulated data, ranging from 250,000 to 1 million data entries. To achieve this, I employed the Python module named FAKER. Below is the snapshot of that code with comments for better understanding

```python
import csv
from faker import Faker
import random
from datetime import datetime, timedelta

# Set the number of records for the dataset
dataset_size = 250000

# Initialize Faker
fake = Faker()

# Create a list to store the generated data
data = []

# Generate the data
for _ in range(dataset_size):
    # Generate fake book data
    book_title = fake.catch_phrase()
    book_author = fake.name()
    book_isbn = fake.isbn13()
    book_available = random.choice([True, False])

    # Generate fake borrower data
    borrower_name = fake.name()
    borrower_email = fake.email()
    borrower_phone = fake.phone_number()

    # Generate fake borrowing data
    borrow_date = fake.date_time_this_year()
    return_date = borrow_date + timedelta(days=random.randint(7, 30))

    # Add the data to the dataset list
    data.append([book_title, book_author, book_isbn, book_available,
                 borrower_name, borrower_email, borrower_phone,
                 borrow_date, return_date])
```

```python
# Define the filename
filename = "library_dataset4.csv"

# Save the dataset as a CSV file
with open(filename, 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Book Title', 'Author', 'ISBN', 'Available',
                     'Borrower Name', 'Email', 'Phone',
                     'Borrow Date', 'Return Date'])  # Write header
    writer.writerows(data)

print(f"Dataset of size {dataset_size} saved as {filename}.")
```

Queries

REDIS

QUERY 1

This code benchmarks the time it takes to retrieve data associated with a specific Redis key using the HGETALL command and records the execution times in a list. The key is 'book:isbn123'

```
query_times = []

for _ in range(30):
    key = 'book:isbn123'
    start_time = time.time()
    value = r.hgetall(key)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 2

By utilizing the zrange command to retrieve a range of objects with scores from a Redis sorted set, this code performs benchmarking. The execution times for each of the 30 iterations are recorded.

```
for _ in range(30):
    key = 'books:availability'
    start_rank = 0
    end_rank = 9
    start_time = time.time()
    results = r.zrange(key, start_rank, end_rank, withscores=True)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 3

Redis's keys command is used in this code to benchmark how long it takes to obtain keys that match a given pattern. The execution times are recorded in a list for a total of 30 iterations.

- The pattern "book: *" matches all keys in the Redis database that have the prefix "book:" as their first character.
- The asterisk * is a wildcard character that represents any sequence of characters.

```
for _ in range(30):
    pattern = 'book:*'
    start_time = time.time()
    keys = r.keys(pattern)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 4

This code is using a specified key pattern for each of the 30 iterations, decodes the data, modifies, and augments the information, and then compiles the results into a list of dictionaries. The key pattern is the same as in previous query.

```
keys = r.keys(pattern)

results = []

# Iterate over keys and retrieve book information
for key in keys:
    key = key.decode()
    book_info = r.hgetall(key)

    if book_info is not None:

        title = book_info.get(b'Title')
        author = book_info.get(b'Author')
        isbn = book_info.get(b'ISBN')
        available = book_info.get(b'Available')

        # Check if any of the fields are None
        if title is not None:
            title = title.decode()
        if author is not None:
            author = author.decode()
        if isbn is not None:
            isbn = isbn.decode()
        if available is not None:
            available = available.decode()

        result = {
            'Title': title,
            'Author': author,
            'ISBN': isbn,
            'Available': available,
            'Modified Field': 'Some modification',
            'Additional Field': 'Some additional data',

        }

        results.append(result)
```

CASSANDRA

QUERY 1

This code calculates the time needed to run a SELECT query in a Cassandra, retrieving information from the "books" table where the ISBN matches "isbn123," for each of the 30 repetitions, and records the execution times in a list.

```
for _ in range(30):
    key = 'isbn123'
    start_time = time.time()

    # Execute the SELECT query
    query = f"SELECT * FROM books WHERE isbn = '{key}'"
    result = session.execute(query)

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 2

This code calculates the amount of time needed to run a SELECT query with a limit of 10 rows on the "books" table in system Cassandra for each of the 30 repetitions. The execution times are then recorded in a list.

```
for _ in range(30):
    start_rank = 0
    end_rank = 9
    start_time = time.time()

    query = f"SELECT * FROM books LIMIT {end_rank}"
    result = session.execute(query)

    rows = []
    for row in result:
        rows.append(row)

        if len(rows) >= end_rank:
            break

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 3

This code retrieves data from the "books_by_isbn" table where the pattern matches "book:," calculates the time it takes to perform a SELECT query in a Cassandra, for each of the 30 repetitions, and saves the execution times in a list.

```
for _ in range(30):
    pattern = 'book:'
    start_time = time.time()

    # Execute the SELECT query with the composite primary key
    query = f"SELECT * FROM books_by_isbn WHERE pattern = '{pattern}'"
    result = session.execute(query)

    for _ in result:
        pass

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 4

A benchmarking process is carried out by this code in a Cassandra database. It calculates the time it takes to run a SELECT query for each of the 30 iterations, retrieving rows with a given title from the "books_by_title" table, processing those rows, and producing a modified result set. A list of the execution times is kept for benchmarking purposes.

```
for _ in range(30):
    title = 'Reverse-engineered cohesive Internet solution'
    start_time = time.time()

    # Execute the SELECT query to retrieve all rows with the specified title
    query = f"SELECT * FROM books_by_title WHERE title = '{title}'"
    rows = session.execute(query)

    results = []


    for row in rows:

        result = {
            'ISBN': row.isbn,
            'Title': row.title,
            'Author': row.author,
            'Available': row.available,
            'Borrower Name': row.borrower_name,
            'Email': row.borrower_email,
            'Phone': row.borrower_phone,
            'Borrow Date': row.borrow_date,
            'Return Date': row.return_date,
            'Modified Field': 'Some modification',
            'Additional Field': 'Some additional data',

        }
        results.append(result)

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

NEO4J

QUERY 1

This code creates a Cypher query to get attributes of a book with a specified ISBN, performs the query, calculates the time required, logs the execution times in a list, and outputs the query result for each of the 30 iterations.

```
cypher_query = (
    "MATCH (b:Book {ISBN: '978-1-5452-4534-7'}) "
    "RETURN b.`Book Title` AS title, b.Author AS author, b.ISBN AS isbn, "
    "b.Available AS available, b.`Borrower Name` AS borrower_name, "
    "b.Email AS email, b.Phone AS phone, b.`Borrow Date` AS borrow_date, "
    "b.`Return Date` AS return_date"
)

query_times = []


for _ in range(30):
    start_time = time.time()
    result = graph.run(cypher_query).evaluate()
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
    print(result)
```

QUERY 2

This code creates a Cypher query to extract particular book properties for each of the 30 Neo4j iterations, sorts the results by availability in descending order, skips a predetermined start rank, and caps the number of results. It then runs the query, iterates through the outcomes, prints each record, calculates the execution times, and stores them in a list.

```
cypher_query = (
    "MATCH (b:Book) "
    "RETURN b.`Book Title` AS title, b.Author AS author, b.ISBN AS isbn, "
    "b.Available AS available, b.`Borrower Name` AS borrower_name, "
    "b.Email AS email, b.Phone AS phone, b.`Borrow Date` AS borrow_date, "
    "b.`Return Date` AS return_date "
    "ORDER BY b.Available DESC "
    "SKIP $start_rank LIMIT $limit"
)
start_rank = 0
limit = 10
with driver.session() as session:

    for _ in range(30):
        start_time = time.time()
        result = session.run(cypher_query, start_rank=start_rank, limit=limit)
        for record in result:
            print(record)
        end_time = time.time()
        execution_time = end_time - start_time
        query_times.append(execution_time)
```

QUERY 3

This Neo4j code runs a Cypher query to obtain all Book nodes from the graph over the course of a 30-iteration loop, extracts a single record from each query result, visits the Book node, and records the execution timings while storing them in a list.

```
cypher_query = (
    "MATCH (b:Book) "
    "RETURN b"

)


with driver.session() as session:

    start_time = time.time()
    for _ in range(30):
        result = session.run(cypher_query)
        record = result.single()
        if record:

            print(record['b'])
        end_time = time.time()
        execution_time = end_time - start_time
        query_times.append(execution_time)
```

QUERY 4

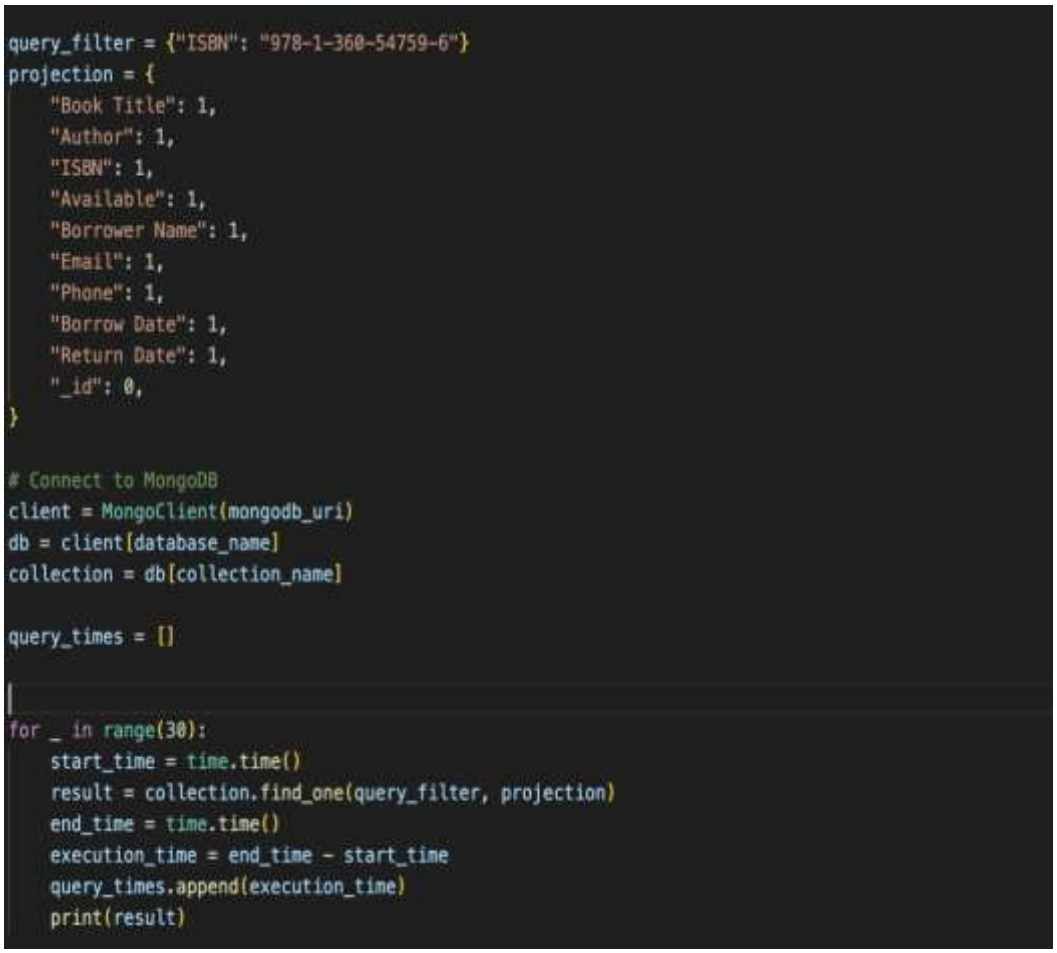
To get book properties (Title, Author, ISBN, and Available) from the graph's Book nodes, this Neo4j code runs a benchmarking operation. Following that, it calculates execution times for 30 iterations and adds those results to a list. Each of the received results is handled separately to extract attributes like title, author, ISBN, and availability.

```
cypher_query = (
    "MATCH (b:Book) "
    "RETURN b.`Title` AS title, b.`Author` AS author, b.`ISBN` AS isbn, b.`Available` AS available"
)


with driver.session() as session:

    for _ in range(30):
        start_time = time.time()
        result = session.run(cypher_query)
        results = [record for record in result]
        end_time = time.time()
        execution_time = end_time - start_time
        query_times.append(execution_time)

        for record in results:
            title = record['title']
            author = record['author']
            isbn = record['isbn']
            available = record['available']
```

MONGO DB

QUERY 1

This code measures how long it takes to retrieve book-related data from a collection in MongoDB using a particular query filter. The code is fetching a document from the collection when the "ISBN" field matches the value "978-1-360-54759-6," which is how the query filter is defined as "ISBN": "978-1-360-54759-6". The programme prints the received findings after measuring the execution times for 30 repetitions.



```
query_filter = {"ISBN": "978-1-360-54759-6"}
projection = {
    "Book Title": 1,
    "Author": 1,
    "ISBN": 1,
    "Available": 1,
    "Borrower Name": 1,
    "Email": 1,
    "Phone": 1,
    "Borrow Date": 1,
    "Return Date": 1,
    "_id": 0,
}

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]

query_times = []

for _ in range(30):
    start_time = time.time()
    result = collection.find_one(query_filter, projection)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
    print(result)
```

QUERY 2

This code measures how long it takes to extract book details from a collection in MongoDB, sorting by the "Available" column in descending order and using a projection to select which fields to include. The code also includes pagination with "skip" and "limit" settings, counts the number of executions (30) and prints the results.

```
query_filter = {}
projection = {
    "Book Title": 1,
    "Author": 1,
    "ISBN": 1,
    "Available": 1,
    "Borrower Name": 1,
    "Email": 1,
    "Phone": 1,
    "Borrow Date": 1,
    "Return Date": 1,
    "_id": 0,
}

# Skip and limit values for pagination
start_rank = 0
limit = 10

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]

query_times = []

for _ in range(30):
    start_time = time.time()
    result = collection.find(query_filter, projection).sort(
        "Available", -1).skip(start_rank).limit(limit)
    for record in result:
        print(record)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 3

This code measures execution times for 30 iterations while retrieving a single document (book information) from a MongoDB collection, excluding the "_id" field from the projection, and reporting the result when it is available.

```
query_filter = {}
projection = {
    "_id": 0,
}

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]

query_times = []


for _ in range(30):
    start_time = time.time()
    result = collection.find_one(query_filter, projection)
    if result:
        print(result)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 4

The code examines the book's "Book Title," "Author," "ISBN," and "Available" details, and after each try, it keeps track of how long it takes to locate and get this data from the collection.

```
query_filter = {}
projection = {
    "_id": 0,
    "Book Title": 1,
    "Author": 1,
    "ISBN": 1,
    "Available": 1
}

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]

query_times = []


for _ in range(30):
    start_time = time.time()
    result = collection.find(query_filter, projection)
    results = list(result)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

MYSQL

QUERY 1

MySQL query to extract certain book information from a dataset whose ISBN matches "978-1-360-54759-6." Then, after measuring the time required to run the query 30 times, publishing the results of the query and keeping track of the execution times in a list.

```
query = """
SELECT `BookTitle`, Author, ISBN, Available, `BorrowerName`, Email, Phone, `BorrowDate`, `ReturnDate`
FROM library_dataset4
WHERE ISBN = '978-1-360-54759-6'
"""

# Execute the query and measure the execution time
query_times = []
for _ in range(30):
    start_time = time.time()
    cursor.execute(query)
    result = cursor.fetchall()
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
    print(result)
```

QUERY 2

MySQL query to retrieve specific book details from a dataset, ordering the results by availability in descending order and limiting the output to 10 records

```
query = """
SELECT `BookTitle`, Author, ISBN, Available, `BorrowerName`, Email, Phone, `BorrowDate`, `ReturnDate`
FROM library_dataset4
ORDER BY Available DESC
LIMIT 10 OFFSET 0
"""

# Execute the query and measure the execution time
query_times = []
for _ in range(30):
    start_time = time.time()
    cursor.execute(query)
    result = cursor.fetchall()
    for record in result:
        print(record)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 3

MySQL query to retrieve a single set of specific book details from a dataset

```
query = """
SELECT `BookTitle`, Author, ISBN, Available, `BorrowerName`, Email, Phone, `BorrowDate`, `ReturnDate`
FROM library_dataset4
LIMIT 1
"""

# Execute the query and measure the execution time
query_times = []
for _ in range(30):
    start_time = time.time()
    cursor.execute(query)
    result = cursor.fetchall()
    if result:
        print(result[0])
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

QUERY 4

MySQL query to retrieve specific book details (Title, Author, ISBN, and Availability) from a dataset

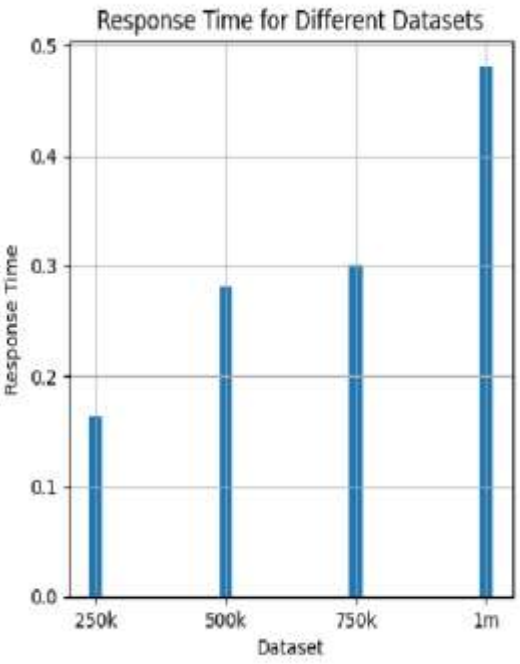
```
query = """
SELECT `BookTitle`, Author, ISBN, Available
FROM library_dataset4
"""

# Execute the query and measure the execution time
query_times = []
for _ in range(30):
    start_time = time.time()
    cursor.execute(query)
    result = cursor.fetchall()
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

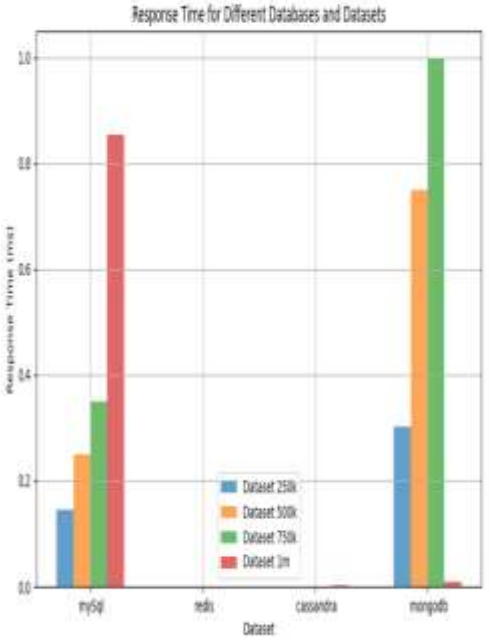
HISTOGRAMS

The Neo4j database is represented by the histogram on the left, while the performance of other databases is represented by the histogram on right side

Query 1

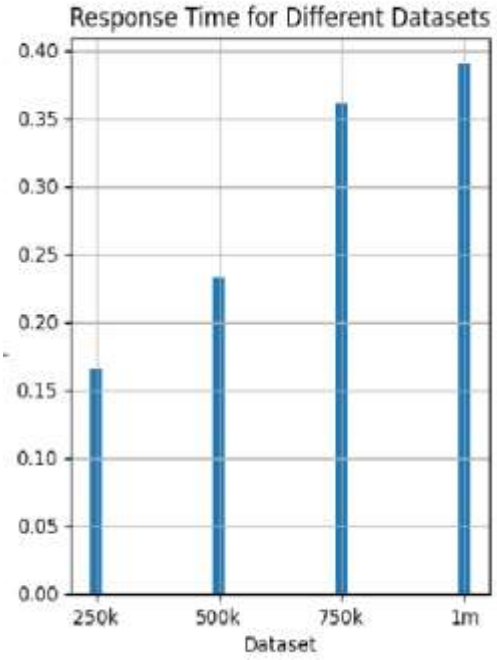
Retrieving book by ISBN

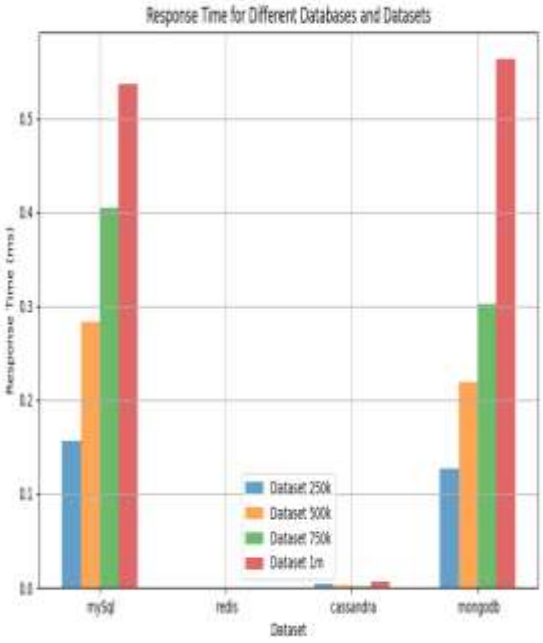




Query 2

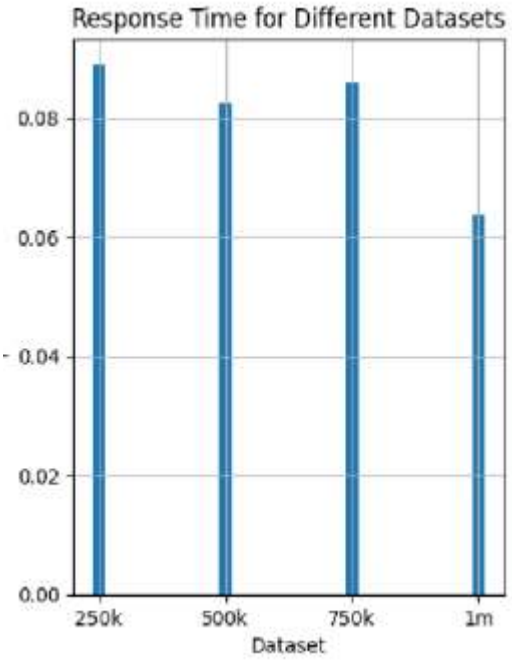
Selecting 10 Rows of book Sorting by Available Column

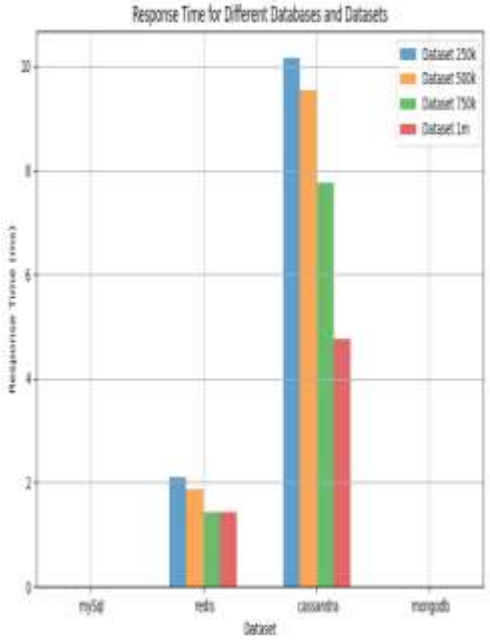




Query 3

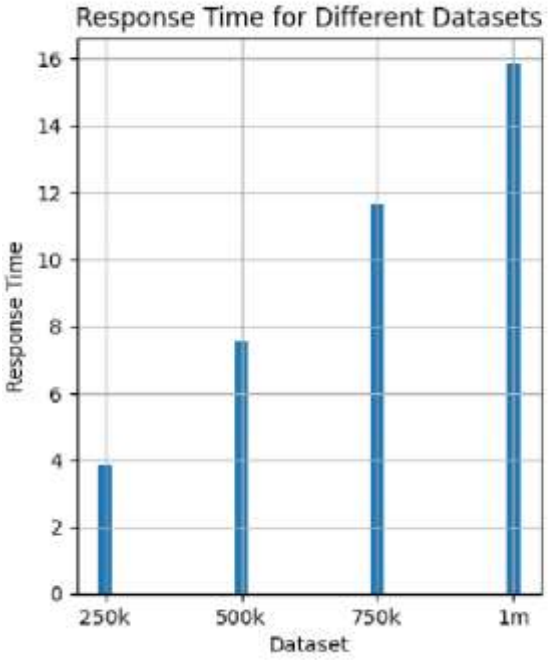
Retrieving Every detail of book, all the columns by applying limit.

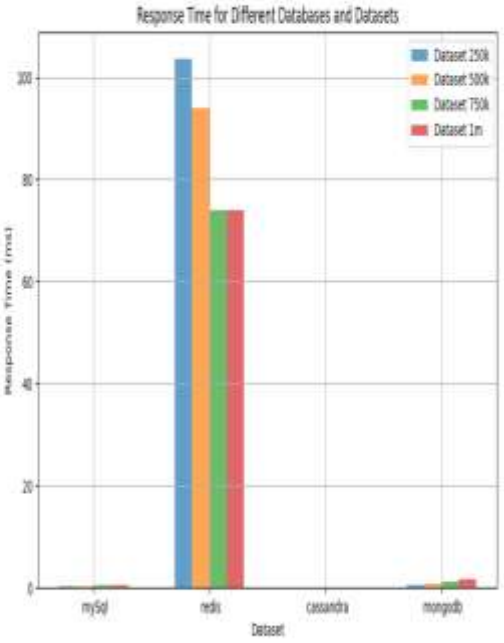




Query 4

Retrieving only few columns of books





CONCLUSIONS

In this project we have compared five different NoSQL databases and evaluated the performance on the development of library management system. As you can see from the histograms the use of database is not certain that which database should be used. The performance of Redis and Cassandra in first two queries is very good meaning very fast in retrieving data as compared to other but the performance of mongo dB and MySQL is good as compared to other. We basically got the idea that some databases perform well in one situation and worst in other situation. Every database has its best- and worst-case scenarios For Example

MongoDB:

- Flexible schema for storing book data and metadata.
- Good read and write performance for general use cases.
- Suitable for library management systems with evolving data structures.

Redis:

- Exceptional performance for caching and low-latency data access.
- Ideal for scenarios where quick access to frequently used data is crucial.

Neo4j:

- Strong performance in complex relationship-based queries.
- Suitable for systems where book data relationships play a significant role.

Cassandra:

- Scalability and high availability for large-scale library management systems.
- Good performance in write-heavy distributed environments.

MySQL

- MySQL excels in managing structured data, making it ideal for scenarios where the book database has a well-defined schema with fixed attributes such as book title, author, ISBN, publication date, etc.
- MySQL can handle complex SQL queries efficiently. This is beneficial when you need to perform advanced searches, generate reports, or analyze book data based on various criteria

And also, there are some worst case scenarios for these databases For Example: Neo4j:

- Worst-case scenarios for Neo4j involve scenarios with simple, tabular data that doesn't benefit from graph-based modeling.
- It may not be the best choice for systems where relationships between books are minimal or non-existent.

Cassandra:

- Worst-case scenarios for Cassandra include small-scale deployments where its complexity and overhead outweigh the benefits.
- It might not be suitable for applications with low write throughput or when immediate consistency is required.

Redis:

- Worst-case scenarios for Redis include scenarios where durability and data persistence are crucial.
- Redis primarily stores data in-memory, making it vulnerable to data loss in cases of server failures.

MongoDB:

- Worst-case scenarios for MongoDB can arise when the data structure is highly normalized or when complex transactions are required.
- It may not be the best fit for applications that demand strict ACID compliance.

MySQL:

- MySQL is designed for structured data with predefined schemas. In scenarios where the data is highly unstructured or semi-structured, using MySQL can be inefficient and challenging.
- Managing and querying data without a well-defined schema can lead to complex and slow SQL queries.

Through this comparative research, we have laid the framework for making an informed selection regarding the best Database Management System (DBMS) for our library management system. The useful insights gained from this study will serve as a compass for future development and optimisation projects, assuring optimal performance and an enhanced user experience in our library management application.