Università degli studi di messina
MIFT Department

Database Project

Curriculum Management System

Supervisor: Armando Ruggeri

Student: Irmuunbilig Burenzevseg

Academic year 2022/2023

1. INTRODUCTION

Objective: The primary objective of this study is to thoroughly analyze and compare the performance of five prominent database management systems—MySQL, MongoDB, Cassandra, Redis, and Neo4j—in the context of a university curriculum management system. This system is responsible for handling course catalogues, course scheduling, and curriculum mapping. The aim is to identify the most suitable database solution for managing these complex data sets efficiently, under varying workloads and query complexities.

Motivation: Selecting the appropriate database management system is critical to ensuring the scalability, reliability, and efficiency of the curriculum management system. With the growing complexity of educational data—ranging from structured relational data to graph-based relationships—this study provides valuable insights into how each DBMS performs in handling this diverse data. The findings will help determine the optimal choice for such applications, balancing data integrity, performance, and scalability.

2. DBMS Solutions Considered

MySQL: Relational DBMS

Overview:

MySQL is one of the most widely adopted relational database management systems (RDBMS). Known for its robustness, MySQL manages structured data by organising it into tables, rows, and columns. It supports SQL (Structured Query Language), which is the standard language for relational databases. MySQL's ACID compliance ensures reliability, especially for transactional data, and makes it a strong candidate for applications requiring complex queries and data integrity.

Key Features:

- **ACID Compliance:** Ensures robust transaction management and data

consistency across multiple operations.

- **Atomicity**: This guarantees that each transaction is treated as a single unit, which either completely succeeds or completely fails. If any part of the transaction fails, the entire transaction is rolled back, leaving the database in its original state.
- **Consistency**: Ensures that a transaction brings the database from one valid state to another, maintaining database integrity. After a transaction, all rules, such as data types, triggers, and constraints, must be respected. For example, if a transaction breaks a foreign key constraint, it will not be committed, ensuring the database remains consistent.
- **Isolation**: This property ensures that the execution of transactions is isolated from one another. In other words, the intermediate steps of a transaction are invisible to other transactions until the transaction is committed. This prevents issues like dirty reads (when one transaction reads data that is part of an incomplete transaction). Various levels of isolation exist, such as Read Uncommitted, Read Committed, Repeatable Read, and Serializable.
- **Durability**: This guarantees that once a transaction is committed, it will remain committed, even in the event of a system crash. The changes made by the transaction are permanently recorded in the database's storage, so they will persist even after a power loss or system failure.

- **Scalability:** Supports both vertical scaling (upgrading hardware) and horizontal scaling (through replication and sharding).
- **Security:** Built-in security mechanisms, including SSL encryption and fine-grained user access control.
- **Use Case Fit:** Well-suited for transactional operations, ensuring reliable performance for tasks like course scheduling and curriculum mapping where relationships between entities are structured and clearly defined.

MongoDB: Document-Oriented NoSQL DBMS

Overview:
MongoDB is a NoSQL database that stores data in BSON (Binary JSON) format, allowing for flexible and dynamic schema designs. MongoDB excels in managing unstructured or semi-structured data, making it a powerful choice for applications that require rapid development and iterative data model adjustments.

- **Key Features:**
    - **Schema Flexibility:** Allows for dynamic schema evolution, making it ideal for handling varying or nested data structures such as course descriptions and syllabus updates.
    - **Document-Oriented Storage:** Enables the storage of complex data objects like course catalogues and professor information in a nested, JSON-like format.
    - **Horizontal Scalability:** Provides built-in sharding capabilities, enabling seamless scaling across distributed environments.
    - **Use Case Fit:** Particularly effective for applications that require storing diverse and evolving data, such as dynamic course catalogues and student progression tracking.

Cassandra: Distributed NoSQL DBMS

Overview:
Apache Cassandra is a highly scalable NoSQL database designed to manage large volumes of data across multiple servers without any single point of failure. With its peer-to-peer distributed architecture, Cassandra is particularly suited for environments that demand high availability and fault tolerance, such as applications with massive datasets and the need for constant uptime.

- **Key Features:**
    - **Peer-to-Peer Architecture:** Ensures that data is replicated across nodes, providing both high availability and redundancy.
    - **Linear Scalability:** Allows seamless scaling by simply adding more nodes to the cluster, with no downtime or interruption.

- **Tunable Consistency:** Offers configurable read and write consistency levels, making it adaptable to different workload demands.
- **Use Case Fit:** Suitable for large-scale operations that require high write throughput, such as real-time course enrollment systems where latency and availability are critical.

Redis: In-Memory Key-Value Store

Overview:
Redis is a high-performance, in-memory key-value store known for its exceptionally fast data access speeds. Its ability to handle a variety of data structures such as strings, hashes, lists, and sets, combined with its in-memory architecture, makes it a popular choice for real-time analytics, caching, and ephemeral data processing.

- **Key Features:**
  - **In-Memory Storage:** Provides extremely low-latency access, which is particularly beneficial for caching or real-time applications, such as instantly fetching course data for a dashboard.
  - **Data Structures:** Supports a variety of complex data types, allowing for versatile data manipulation, such as storing and retrieving course schedules and professor availability.
  - **Persistence Options:** Offers both snapshotting and append-only file (AOF) persistence mechanisms for data durability despite its primary in-memory nature.
  - **Use Case Fit:** Best suited for applications that demand rapid access to frequently accessed data, such as caching course schedules for real-time display, though less effective for complex, persistent queries.

Neo4j: Graph DBMS

Overview:
Neo4j is a native graph database that excels in managing highly connected data. It stores data in nodes (entities) and edges (relationships), which makes it ideal for applications requiring sophisticated traversal of relationships, such as curriculum

mapping, where courses, prerequisites, and dependencies form a complex web of connections.

- **Key Features:**
    - **Graph Data Model:** Naturally represents entities and their relationships, making it a powerful tool for modelling the interconnected data in a university curriculum, such as the relationships between courses, prerequisites, and programs.
    - **Cypher Query Language:** A specialized query language designed for graph data, allowing for efficient querying and traversal of connected nodes.
    - **ACID Transactions:** Ensures data integrity during complex graph updates, a crucial feature when managing interrelated curriculum data.
    - **Use Case Fit:** Ideally suited for curriculum mapping and any application requiring complex relationship analysis, where understanding the dependencies and interactions between courses is essential.

3. Design

Data Model

The data model for the curriculum management system includes the following core entities:

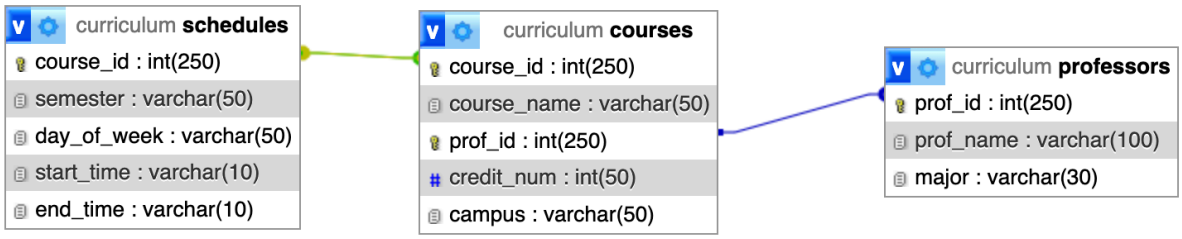
- **Courses:** stores information about each course, including its unique identifier (`course_id`), name, credit hours, assigned professor (`prof_id`), and the campus where it is offered.
- **Schedules:** Links to the Courses table via `course_id` and contains details about course timetables, such as the semester, days of the week, and start and end times. This ensures that each course is associated with a specific schedule.
- **Professors:** The table stores information about faculty members, including their unique identifier (`prof_id`), name, and area of expertise. It connects to the Courses table to assign professors to specific courses, facilitating faculty management.

Each database system models entities like **courses, schedules, and professors** in a

way that best suits its underlying architecture, providing flexibility for different use cases:

- **MySQL:** Normalized tables for courses, schedules, and professors, with foreign keys maintaining relationships between entities.
- **MongoDB:** Collections for courses, schedules, and professors, with embedded documents for representing relationships.
- **Cassandra:** Column families designed for high write throughput, optimized for denormalized storage of course and schedule data.
- **Redis:** Key-value pairs where each key corresponds to an entity like a course or schedule, and values represent structured data using Redis' native data types.
- **Neo4j:** Nodes representing courses, schedules, and professors, with relationships illustrating connections, such as prerequisites and teaching assignments.

Schema:



4. Tools and Technologies Used

- **MySQL:** Managed using **phpMyAdmin**, a web-based database management tool that provides an intuitive interface for handling MySQL databases, facilitating tasks such as database creation, query execution, and overall database administration.
- **MongoDB: MongoDB Compass** is employed for visual database management and query execution, allowing for the easy exploration and manipulation of data within MongoDB databases. It provides features like schema visualization,

aggregation pipeline building, and performance monitoring.

- **Cassandra:** It is interacted with via **CQL (Cassandra Query Language)** through the Cassandra driver. This setup allows for efficient querying and management of large-scale, distributed NoSQL databases designed for high availability and fault tolerance.
- **Redis:** Managed and monitored through **RedisInsight**, a powerful GUI for visualizing, analyzing, and optimizing Redis databases. RedisInsight aids in real-time monitoring, query analysis, and performance optimization.
- **Neo4j:** Administered using **Neo4j Desktop** and **Neo4j Browser**, tools designed for graph database management, query execution, and data visualization. Neo4j enables intuitive exploration of complex relationships between data points and facilitates efficient querying using Cypher.

5. Implementation

Data Generation

To simulate realistic data for our curriculum management system, the Python **Faker** library was utilized to generate datasets of increasing sizes: **250k, 500k, and 750k records**. These datasets were created to mimic the real-world complexity of academic institutions, including professor profiles, course details, and scheduling data. Once generated, the data was formatted and exported as CSV files for consistent import across each database system.

1. **Initialization**:
    - Imported essential libraries: `Faker`, `csv`
    - Initialized the `Faker` object to simulate realistic data for professors, courses, and schedules.
2. **Generate Data**:
    - Looped through the specified dataset sizes (250k, 500k, 750k records) to generate relevant data for the **Professors**, **Courses**, and **Schedules** tables.
    - The generated data adhered to the schema requirements of the curriculum management system. Each record was created with realistic

attributes, such as professor names, course titles, schedules with specific times, and campus information.

3. **Export Data**:
    - The generated data was stored in Python lists, which were subsequently exported to CSV files using `csv.DictWriter`.

```python
from faker import Faker
import csv

fake = Faker()

# Define the dataset sizes
dataset_sizes = [250000, 500000, 750000]    # 250000, 500000, 750000

# Generate and export data for each dataset size
for size in dataset_sizes:
    # 
    professors = []
    courses = []
    schedules = []

    # professors table
    for prof_id in range(1, size + 1):
        prof = {
            'prof_id': prof_id,       #Primary key
            'prof_name': fake.name(),
            'major': fake.random_element(elements=('Physics', 'Mathematics', 'English', 'Chemistry', 'Biology')),
        }
        professors.append(prof)

    # courses table
    prof_ids = list(range(1, size + 1))   # making a list to connect
    for course_id in range(1, size + 1):
        prof_id = fake.random_element(elements=prof_ids)    #connecting course with the corresponding prof
        prof_ids.remove(prof_id)
        course = {
            'course_id': course_id,    #primary key
            'course_name': fake.random_element(elements=('Subject 1', 'Subject 2', 'Subject 3', 'Subject 4')),  #course catalog
            'prof_id': prof_id,        #foreign key
            'campus': fake.random_element(elements=('Annunziata', 'Papardo', 'Policlinic', 'Central')),
            'credit_num': fake.random_element(elements=(3,6,9,12))
        }
        courses.append(course)


    # schedule table
    course_ids = list(range(1, size + 1))    # making a list to connect
    for schedule_id in range(1, size+1):
        course_id = fake.random_element(elements=course_ids)   #creating a schedule with the corresponding course
        course_ids.remove(course_id)
        schedule = {
            'course_id': course_id,  # foreign key & the primary key
            'semester': fake.random_element(elements=('Spring', 'Autumn', 'Winter')),
            'day': fake.random_element(elements=("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")),
            "start_time": fake.time(pattern="%I:%M %p"),
            "end_time": fake.time(pattern="%I:%M %p")
        }
        schedules.append(schedule)


    # Export the generated data to CSV files
    professors_filename = f'professors_{size}.csv'
    courses_filename = f'courses_{size}.csv'
    schedules_filename = f'schedules_{size}.csv'

    with open(professors_filename, 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=professors[0].keys())
        writer.writeheader()
        writer.writerows(professors)

    with open(courses_filename, 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=courses[0].keys())
        writer.writeheader()
        writer.writerows(courses)

    with open(schedules_filename, 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=schedules[0].keys())
        writer.writeheader()
        writer.writerows(schedules)

    print(f'Datasets with {size} records generated and exported to {professors_filename}, {courses_filename}, {schedules_filename}')
```

Datasets Generated

1. **250,000 records**
    - Professors: `professors_250000.csv`
    - Courses: `courses_250000.csv`
    - Schedules: `schedules_250000.csv`
2. **500,000 records**
    - Professors: `professors_500000.csv`
    - Courses: `courses_500000.csv`
    - Schedules: `schedules_500000.csv`
3. **750,000 records**
    - Professors: `professors_750000.csv`
    - Courses: `courses_750000.csv`
    - Schedules: `schedules_750000.csv`

Database Creation and Data Import

The datasets were imported into MySQL, MongoDB, Cassandra, Redis, and Neo4j. Each database is prepared to host the generated datasets. Configuration and setup details are provided for each database in the `dataset_insertion` folder.

Query Execution

To evaluate the performance of each database, three queries of varying complexity were defined for each dataset. These queries were designed to challenge different aspects of database performance, including data retrieval speed, index efficiency, and query optimisation.

Each dataset was queried 31 times to obtain precise and reliable performance measurements while accounting for potential caching mechanisms. The first execution was considered the "initial run," capturing the cold start performance before any caching occurred. The subsequent **30 executions** were then used to calculate the **average execution time**, which provided a more accurate assessment of the database's performance under normal operating conditions, where caching effects were in play.

This methodology thoroughly evaluated each database's raw performance and caching

efficiency. By averaging the results from 30 runs after the initial execution, we gained insight into how well each database optimised repeated queries and handled both complex and straightforward query operations.

This comprehensive approach to query performance analysis enabled a detailed comparison of the databases, highlighting their strengths and weaknesses in different scenarios.

MySQL

- **Language & Libraries**: Python, mysql.connector
- **Process**:
    - Established a connection to the MySQL server using connection parameters (host, username, password, and database).
    - Temporarily disabled foreign key checks to expedite data insertion, reducing overhead.
    - Used the `executemany()` function to batch-insert rows from CSV files with a batch size of 1000, significantly improving insertion speed by minimizing transaction overhead.
    - Re-enabled foreign key checks once the data import was complete to ensure referential integrity.
- **Performance**: Batching operations reduced the time spent on individual inserts, improving performance under large dataset imports. Execution times were printed for each table for analysis.

MongoDB

- **Language & Libraries**: Python, pymongo
- **Process**:
    - Established a connection to MongoDB using the `pymongo` library, connecting to the local MongoDB instance via host and port.
    - Parsed the CSV files and converted them into BSON (Binary JSON) documents.
    - Employed `bulk_write()` to insert the documents in bulk into their

respective collections (`Professors`, `Courses`, `Schedules`).
    - Measured and printed execution times for each dataset size.
- **Performance**: Bulk write operations reduced the number of round-trips between the client and the server, leading to faster data import, especially with larger datasets. MongoDB's schema flexibility allowed for dynamic data import without predefined schemas, reducing the need for upfront schema design.

Neo4j

- **Language & Libraries**: Python, neo4j
- **Process**:
    - Connected to the Neo4j database using the Bolt protocol via the `neo4j` Python driver.
    - Created indexes on key properties like `Professor ID`, `Course ID`, and `Schedule ID` to optimize subsequent queries.
    - Batched the import of nodes and relationships using the `UNWIND` clause, with a batch size of 1000, ensuring efficient processing of graph data.
    - Execution times were printed for both node and relationship insertions.
- **Performance**: Index creation before data import significantly improved the performance of subsequent queries. Neo4j's native graph processing and optimization for connected data allowed for efficient handling of the relational aspects of curriculum data.

Cassandra

- **Language & Libraries**: Python, cassandra-driver
- **Process**:
    - Connected to the Cassandra cluster and created the necessary keyspaces and tables.
    - Utilized `BatchStatement` to batch insert rows with a batch size of 50, ensuring compliance with Cassandra's batch size limitations.
    - Employed a `ThreadPoolExecutor` to enable parallel data import,

leveraging Cassandra's ability to handle large amounts of writes efficiently.
    - Execution times for importing each dataset were measured and printed.
- **Performance**: Cassandra's distributed architecture allowed for horizontal scaling and high throughput during the data import. Parallel processing and batch operations ensured that data import was executed quickly across the cluster.

Redis

- **Language & Libraries**: Python, redis
- **Process**:
    - Established a connection to the Redis instance using `redis.Redis`, specifying host and port.
    - Leveraged Redis's `pipeline()` feature to batch commands, minimizing network round-trips during data insertion.
    - Inserted each row as a hash (using `hset`), with key prefixes indicating the dataset (e.g., `professors:12345`).
    - Printed execution times after inserting each dataset.
- **Performance**: Redis's in-memory architecture led to extremely fast data writes. However, the performance was slightly diminished when dealing with larger datasets and more complex structures, as Redis is primarily designed for lightweight key-value storage rather than complex querying.

Query Execution

To evaluate the performance of each database, three queries of varying complexity were defined for each dataset. These queries were designed to challenge different aspects of database performance, including data retrieval speed, index efficiency, and query optimisation.

Each dataset was queried 31 times to obtain precise and reliable performance measurements while accounting for potential caching mechanisms. The first execution was considered the "initial run," capturing the cold start performance before any caching occurred. The subsequent **30 executions** were then used to calculate the **average execution time**, which provided a more accurate assessment of the database's performance under normal operating conditions, where caching effects were in play.

This methodology thoroughly evaluated each database's raw performance and caching efficiency. By averaging the results from 30 runs after the initial execution, we gained insight into how well each database optimised repeated queries and handled both complex and straightforward query operations.

This comprehensive approach to query performance analysis enabled a detailed comparison of the databases, highlighting their strengths and weaknesses in different scenarios.

6. Experiments

The performance of each query was measured in terms of execution time. The average execution time for each query across different databases and dataset sizes was recorded. Suitable SQL or NoSQL statements were created for each query based on the database type. These queries were run using Python, leveraging the appropriate database connectors and execution methods available in the database libraries. The execution time for each query was measured and recorded for further analysis. This method facilitated a detailed comparison of the databases' performance in terms of query processing speed and efficiency. By analysing execution times across different datasets and databases, it was possible to identify performance differences and determine which database performed more efficiently under varying workload conditions.

Query 1: Fetch All Courses Offered in the Spring

- This query retrieves a list of all courses offered during the Spring semester. It focuses on the database's ability to efficiently retrieve a large set of records based on a seasonal filter. This query involves a simple condition based on the semester attribute.

Query 2: Fetch Names of Professors Who Teach at 'Papardo'

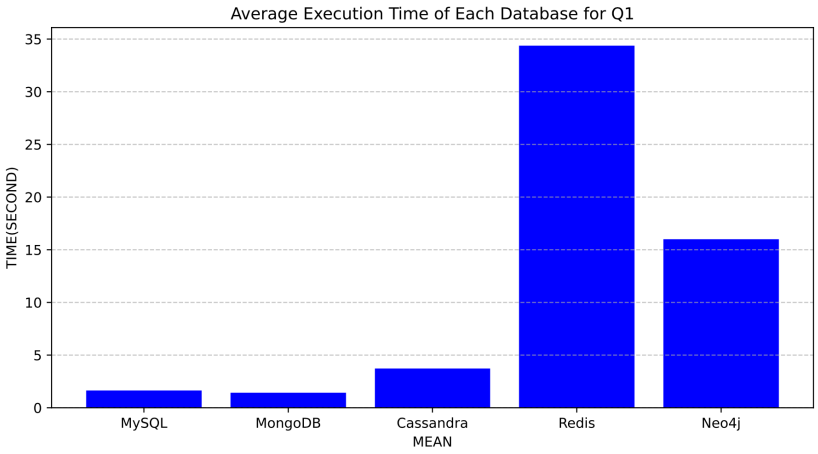
- This query identifies all professors who teach at the 'Papardo' campus. It challenges the database to filter through the data and return only the relevant names associated with a specific location. The query involves a `JOIN` operation

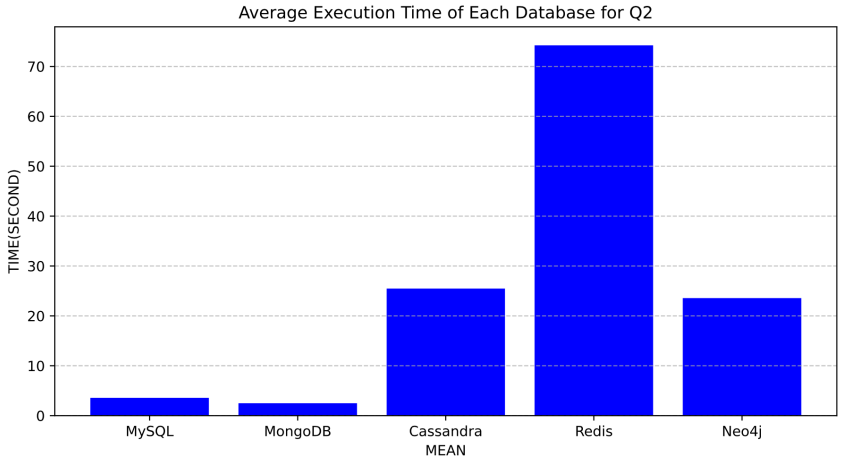
or filter condition based on campus location.

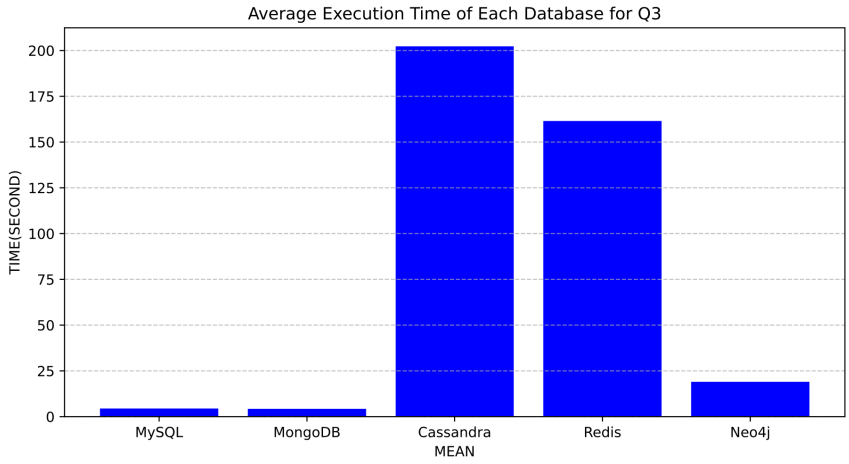
Query 3: Fetch Names of Professors Who Teach at 'Papardo Campus' in Any Spring Semester

- This query is a bit more complex as it not only filters professors by their campus but also restricts the results to those teaching during the Spring semester. It requires handling multiple conditions (campus and semester) and joining tables to ensure both criteria are met.

Average Execution Times:







7. Conclusions

From the experiments conducted, the following conclusions can be drawn:

MongoDB:

- **A top performer** across all dataset sizes, consistently outpacing other databases in terms of speed and efficiency.
- **Highly optimized** for large datasets and complex queries, excelling in scenarios that require rapid data retrieval and manipulation.
- Ideal for **low-latency applications** where performance consistency across diverse workloads is paramount, making it the most dependable option for scalable, high-demand environments.

MySQL:

- Demonstrated **robust performance**, only slightly slower than MongoDB but still highly efficient across different query complexities and dataset sizes.
- A solid choice for applications that demand **steady, reliable performance**, especially in transactional and structured data environments.
- Well-suited for **low-latency, high-reliability requirements**, offering consistent query execution times with strong optimization for relational data.

Neo4j:

- Showed **moderate performance** overall, with noticeable improvements as the dataset size increased, highlighting its efficiency in processing graph-based queries as data complexity scales.
- Best suited for **graph-based applications**, where relationships between data points are central to the workload.
- An excellent option for systems that can **tolerate moderate execution times**, especially when working with interconnected data, as Neo4j leverages its native graph processing capabilities for effective data exploration.

Cassandra:

- Performed **efficiently for simpler queries** like Q1 and Q2, benefiting from its distributed, fault-tolerant architecture designed for high availability.
- However, faced **significant performance degradation** with more complex queries (Q3), resulting in it being the slowest database for intricate workloads.
- Best suited for **scenarios involving simpler queries** and distributed data, but **less effective** when dealing with more complex, multi-condition queries that require extensive data filtering and joining.

Redis:

- Despite its **in-memory architecture** designed for speed, Redis struggled with the **volume and complexity** of the queries, which dampened its performance in this context.
- Excellent for **real-time data caching and lightweight queries**, but **not ideal** for large-scale, complex datasets that require significant processing.
- Should be **avoided for use cases** where deep query performance and handling of high data volume are critical, as its architecture is not tailored for such workloads.

Summary Recommendations:

- **MongoDB and MySQL**: These are the most **reliable and consistent choices** for applications requiring **low-latency, high-performance query execution** across a wide variety of data sizes and query complexities. MongoDB excels in

handling unstructured data, while MySQL offers robust relational database management.

- **Neo4j**: A strong contender for applications dealing with **graph-based data** or highly interconnected datasets. It provides **balanced performance** and is particularly well-suited for scenarios where **moderate query times** are acceptable in exchange for powerful graph traversal capabilities.
- **Cassandra**: Works well for **distributed, fault-tolerant systems** handling simpler queries, but should be approached cautiously for **complex, multi-faceted queries**, where its performance declines sharply.
- **Redis**: Best used for **fast, in-memory operations** but is **not suitable for complex query processing** or large datasets, making it less viable for applications where detailed query performance is essential.