# PROJECT REPORT OF DATABASE MODULE B

***UNIVERSITA DEGLI STUDI DI MESSINA***

**MIFT DEPARTMENT**

Prof. ARMANDO RUGGERI

PULI NUTHAN

540214

VIJAYARAJAN SANGEETHA VIDHYA HARINI

533994

ACADEMIC YEAR – 2022/2023

## INDEX

# Introduction

The Aim of this project is to implement a Course Management System and analyze the performance of different database systems: MySQL, MongoDB, Neo4j, Redis, and Cassandra. We evaluate and measure the performance and query execution times across datasets of varying sizes and different query complexities.

Here's a brief explanation about the Databases we used:

## MySQL

MySQL is a relational database management system (RDBMS) that stores data in structured tables with predefined schemas. It enforces strict relationships between tables using foreign keys and supports SQL queries for data manipulation. MySQL is best for applications that require strong consistency, transactional integrity, and structured data storage, such as banking systems, e-commerce, and enterprise applications.

## MongoDB

MongoDB is a NoSQL document-oriented database that stores data in JSON-like BSON documents instead of tables. This allows for flexible schemas, making it ideal for semi-structured and rapidly changing data. It scales horizontally and performs well for read-heavy and large-scale applications like content management, real-time analytics, and social media platforms.

## Cassandra

Apache Cassandra is a distributed, wide-column NoSQL database designed for high availability and scalability. It excels at handling large-scale write operations and is optimized for fault tolerance, meaning it can operate even if some nodes fail. Cassandra is best suited for big data applications, IoT, and real-time analytics, where fast writes and scalability are crucial.

## Redis

Redis is a high-performance, in-memory key-value store known for its blazing-fast data retrieval. Since it keeps data in RAM, it is often used for caching, session storage, real-time leaderboards, and message queues. While incredibly fast, Redis is best for temporary or frequently accessed data rather than long-term storage.

## Neo4j

Neo4j is a graph database designed for handling complex relationships between data. It stores information as nodes and edges, making it ideal for network analysis, recommendation systems, and fraud detection. Unlike traditional databases, Neo4j is optimized for traversing connections quickly, making it perfect for social networks, supply chain management, and knowledge graphs.

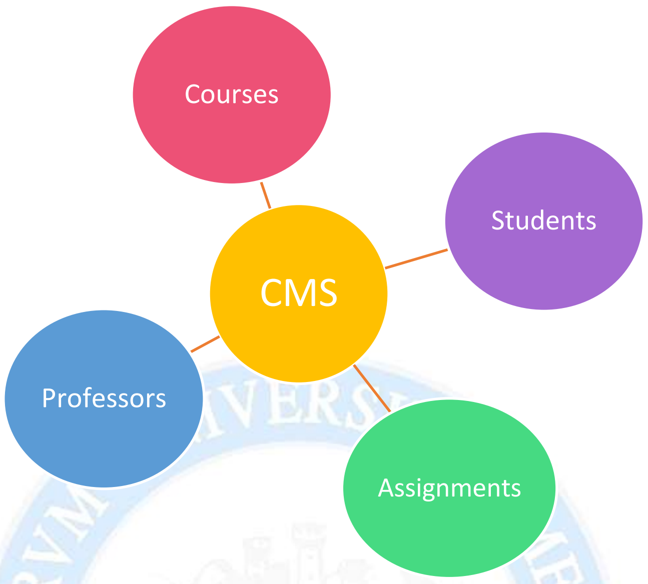
## Overview of the Project

1. A Synthetic Dataset of 1 million records was generated using the Faker Library in Python. The contents of the Dataset are as follows:
    - *course_id*
    - *course_name*
    - *course_content*
    - *student_id*
    - *student_name*
    - *student_email_address*
    - *professor_id*
    - *professor_name*
    - *professor_email_address*
    - *assignment_id*
    - *assignment_title*
    - *submission_status*
    - *score*
2. The Course Management System is implemented in the Docker Desktop with the following databases:
    - **MySQL**
    - **MongoDB**
    - **Cassandra**
    - **Redis**
    - **Neo4j**

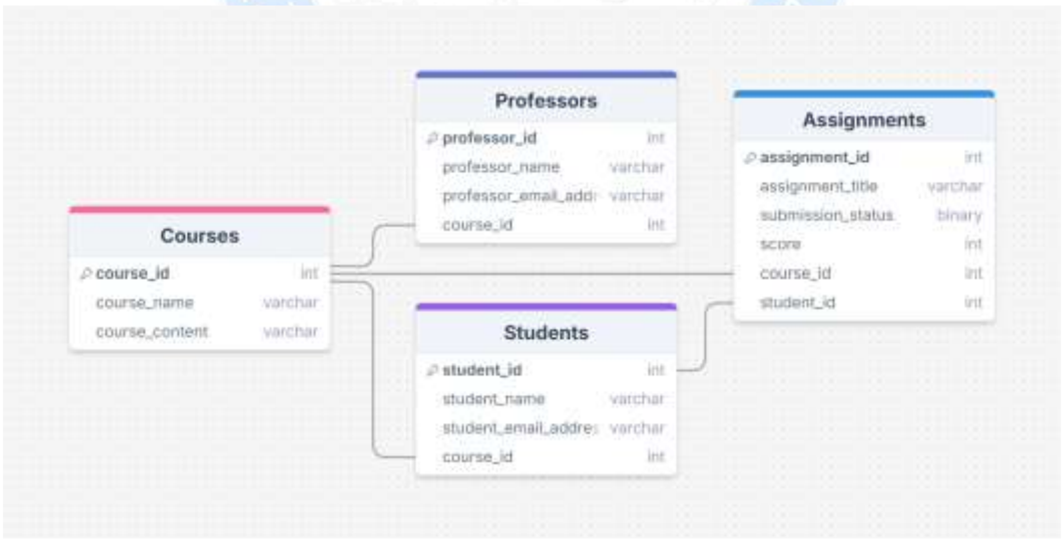
    Each Database is created in its respective container within a network.
3. The contents of the dataset were inserted into each database into the following tables:
    - **Courses:** Contains course details such as course ID, name, and content.
    - **Students:** Contains student IDs, names, email addresses, and enrolled courses.
    - **Professors:** Contains professor IDs, names, email addresses, and assigned courses.
    - **Assignments:** Contains assignment IDs, titles, submission statuses, scores, and student-courses relationships.
4. 4 Queries of increasing complexities were executed in total of 31 times by taking 25%, 50%, 75%, 100% (250,000, 500,000, 750,000, 1,000,000 records respectively) of the 1 million records' Dataset.
5. The execution time of the first experiment and the average execution time of the following 30 experiments were recorded and analyzed to observe the performance of each database under different conditions.

## Data Generation

We have used Faker Library in Python to generate the fake data of 1 million records. The Dataset consists of 4 tables they are: Courses, Students, Professors and Assignments.



The data Model was visualized below to understand the contents of the generated data.



We have only generated one dataset of 1 million records, instead of 4 datasets (250,000, 500,000, 750,000, 1,000,000) as we intend to use this one dataset and implement 25%, 50%, 75% and 100% of this one dataset each time to run the experiments.

# Data Configuration

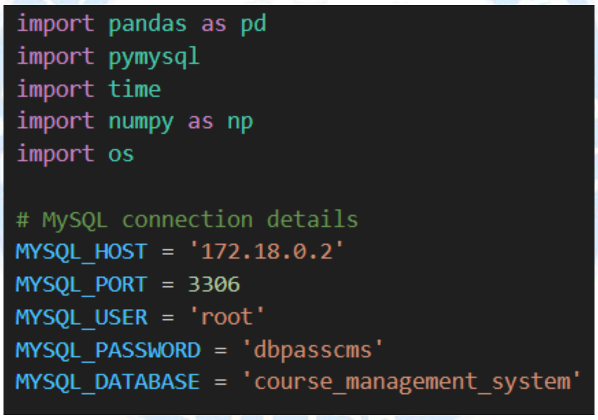
We have created a docker compose file, which first pulls the images of **mysql, mongodb, cassandra, redis** and **neo4j** if they don't exist. Then it creates a *Course Management System (CMS)* database environment using five different databases: *MySQL, MongoDB, Neo4j, Redis, and Cassandra,* each running in its own container.

It defines a custom bridge network (*cms-network*) to allow seamless communication. Each database is exposed on its respective port (e.g., MySQL on **3306**, MongoDB on **27017**, Neo4j on **7474/7687**, Redis on **6379**, and Cassandra on **9042**).

We have used python to interact with the databases. We have copied respective python scripts and csv files containing the generated data to each respective container.

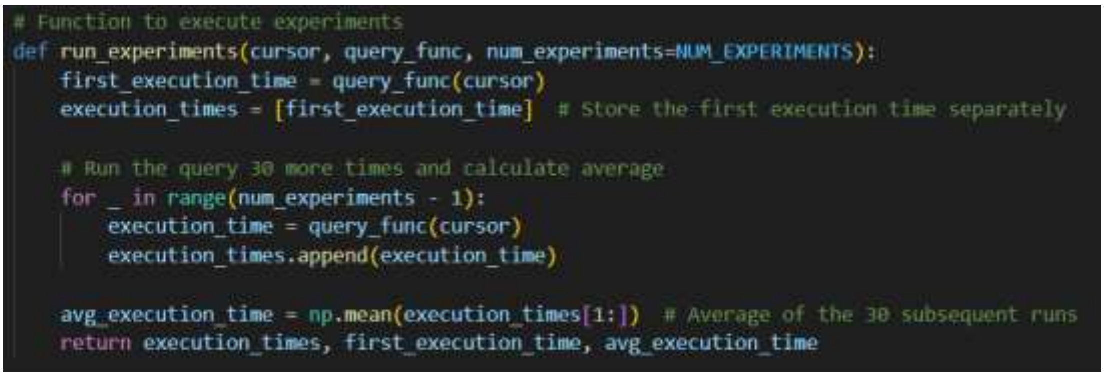
***For MySQL:***

It starts by establishing a connection to a MySQL database using **pymysql**, where it creates a database (*course_management_system*) and creates four tables: *Courses, Students, Professors, and Assignments*. The script reads data from a CSV file containing **1 million records** and inserts subsets of **250K, 500K, 750K, and 1M** records into the database.



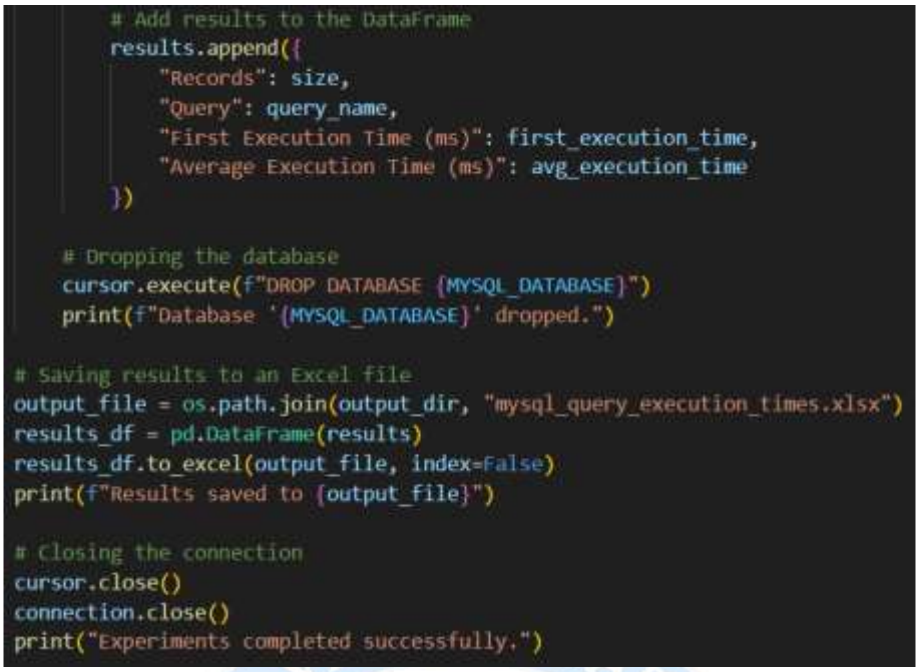
```python
import pandas as pd
import pymysql
import time
import numpy as np
import os

# MySQL connection details
MYSQL_HOST = '172.18.0.2'
MYSQL_PORT = 3306
MYSQL_USER = 'root'
MYSQL_PASSWORD = 'dbpasscms'
MYSQL_DATABASE = 'course_management_system'
```

Once the data is loaded, the script executes **four predefined queries**. Each query runs **31 times**, measuring execution times for the **first execution** (cold start) and calculating the **average execution time** across subsequent runs. After each test, the database is dropped and recreated to ensure consistency in benchmarking.



```python
# Function to execute experiments
def run_experiments(cursor, query_func, num_experiments=NUM_EXPERIMENTS):
    first_execution_time = query_func(cursor)
    execution_times = [first_execution_time]  # Store the first execution time separately

    # Run the query 30 more times and calculate average
    for _ in range(num_experiments - 1):
        execution_time = query_func(cursor)
        execution_times.append(execution_time)

    avg_execution_time = np.mean(execution_times[1:])  # Average of the 30 subsequent runs
    return execution_times, first_execution_time, avg_execution_time
```

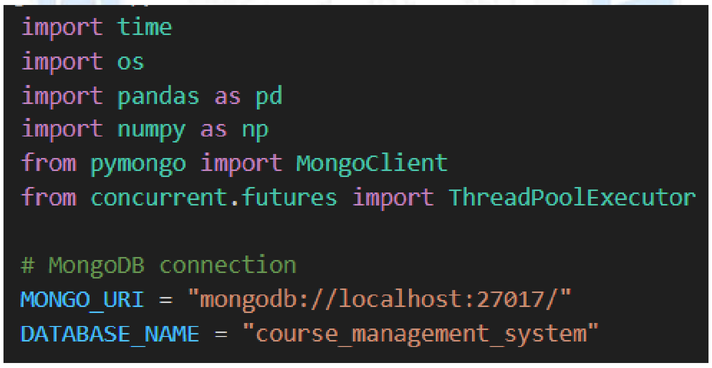
The query execution results are then stored in an Excel file for further analysis.



```python
        # Add results to the DataFrame
        results.append({
            "Records": size,
            "Query": query_name,
            "First Execution Time (ms)": first_execution_time,
            "Average Execution Time (ms)": avg_execution_time
        })

    # Dropping the database
    cursor.execute(f"DROP DATABASE {MYSQL_DATABASE}")
    print(f"Database '{MYSQL_DATABASE}' dropped.")

# Saving results to an Excel file
output_file = os.path.join(output_dir, "mysql_query_execution_times.xlsx")
results_df = pd.DataFrame(results)
results_df.to_excel(output_file, index=False)
print(f"Results saved to {output_file}")

# Closing the connection
cursor.close()
connection.close()
print("Experiments completed successfully.")
```

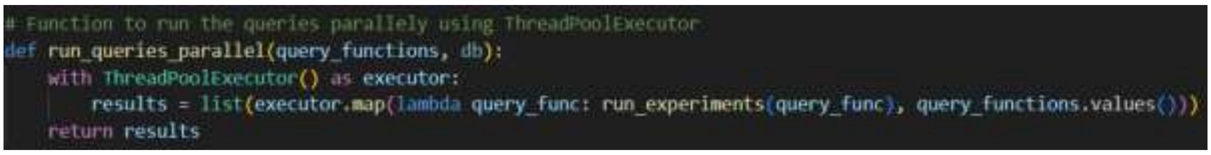
***For MongoDB:***

Similar to MySQL, we connect to a MongoDB database using **pymongo**, load data from the CSV file, and insert it into four collections: *Students, Courses, Professors, and Assignments*.



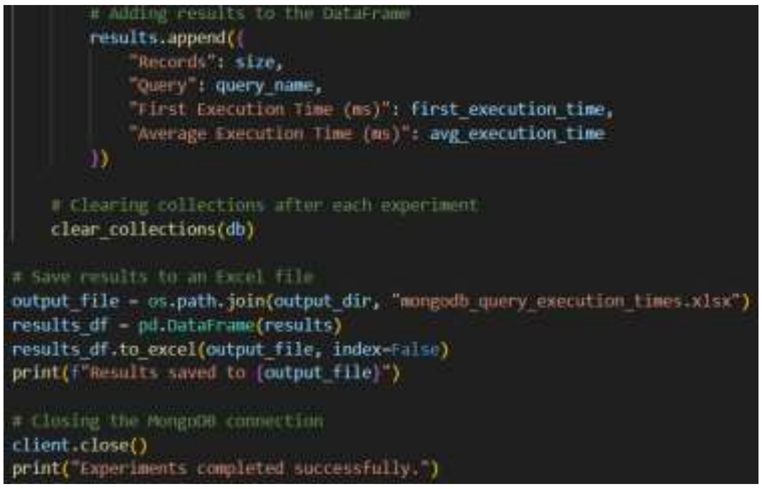
```python
import time
import os
import pandas as pd
import numpy as np
from pymongo import MongoClient
from concurrent.futures import ThreadPoolExecutor

# MongoDB connection
MONGO_URI = "mongodb://localhost:27017/"
DATABASE_NAME = "course_management_system"
```

Indexes are created to optimize query performance. Then we defined four queries just like we did for MySQL. Each query is executed **31 times** and we recorded both the **first execution time** and the **average execution time** over subsequent runs. The queries are executed in parallel using **ThreadPoolExecutor** to simulate real-world workloads efficiently.



```python
# Function to run the queries parallely using ThreadPoolExecutor
def run_queries_parallel(query_functions, db):
    with ThreadPoolExecutor() as executor:
        results = list(executor.map(lambda query_func: run_experiments(query_func), query_functions.values()))
    return results
```

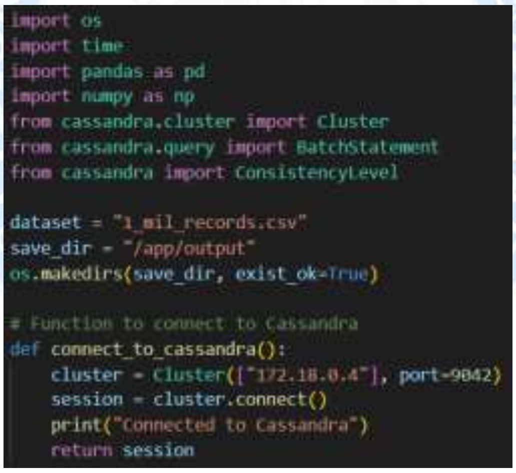
Experiments are conducted for different dataset sizes (**250K, 500K, 750K, and 1M** records), and results are saved in an Excel file for performance analysis. After each experiment, the collections are cleared to ensure consistency in subsequent runs.



```
        # Adding results to the DataFrame
        results.append({
            "Records": size,
            "Query": query_name,
            "First Execution Time (ms)": first_execution_time,
            "Average Execution Time (ms)": avg_execution_time
        })

    # Clearing collections after each experiment
    clear_collections(db)

# Save results to an Excel file
output_file = os.path.join(output_dir, "mongodb_query_execution_times.xlsx")
results_df = pd.DataFrame(results)
results_df.to_excel(output_file, index=False)
print(f"Results saved to {output_file}")

# Closing the MongoDB connection
client.close()
print("Experiments completed successfully.")
```

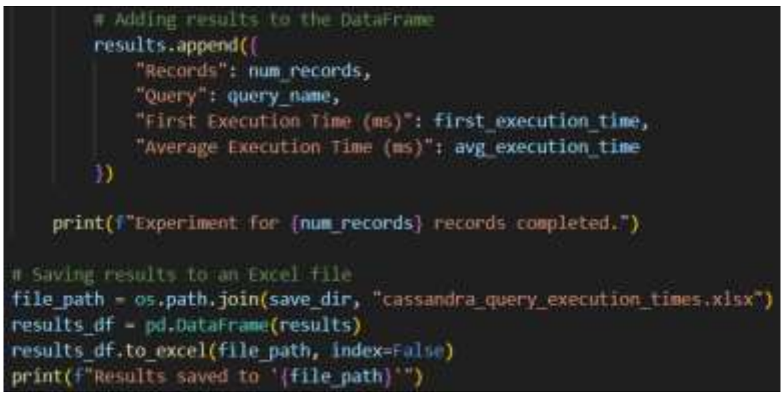
***For Cassandra:***

And similarly, we connect to a Cassandra cluster, create a **keyspace** and four tables (*Courses, Students, Assignments, Professors*), and insert data from the CSV file in batches.



```
import os
import time
import pandas as pd
import numpy as np
from cassandra.cluster import Cluster
from cassandra.query import BatchStatement
from cassandra import ConsistencyLevel

dataset = "1_mil_records.csv"
save_dir = "/app/output"
os.makedirs(save_dir, exist_ok=True)

# Function to connect to Cassandra
def connect_to_cassandra():
    cluster = Cluster(["172.18.0.4"], port=9042)
    session = cluster.connect()
    print("Connected to Cassandra")
    return session
```

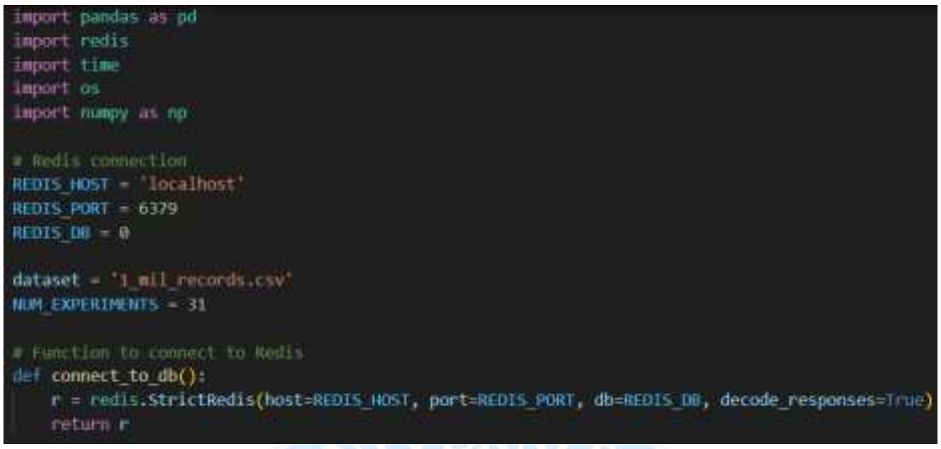
Then we execute four queries 31 times for different dataset sizes (**250K, 500K, 750K, 1M records**) and measure **first and average execution times**. The database is reset before each experiment. The results are then stored in an **Excel file**.



```
        # Adding results to the DataFrame
        results.append({
            "Records": num_records,
            "Query": query_name,
            "First Execution Time (ms)": first_execution_time,
            "Average Execution Time (ms)": avg_execution_time
        })

    print(f"Experiment for {num_records} records completed.")

# Saving results to an Excel file
file_path = os.path.join(save_dir, "cassandra_query_execution_times.xlsx")
results_df = pd.DataFrame(results)
results_df.to_excel(file_path, index=False)
print(f"Results saved to '{file_path}'")
```

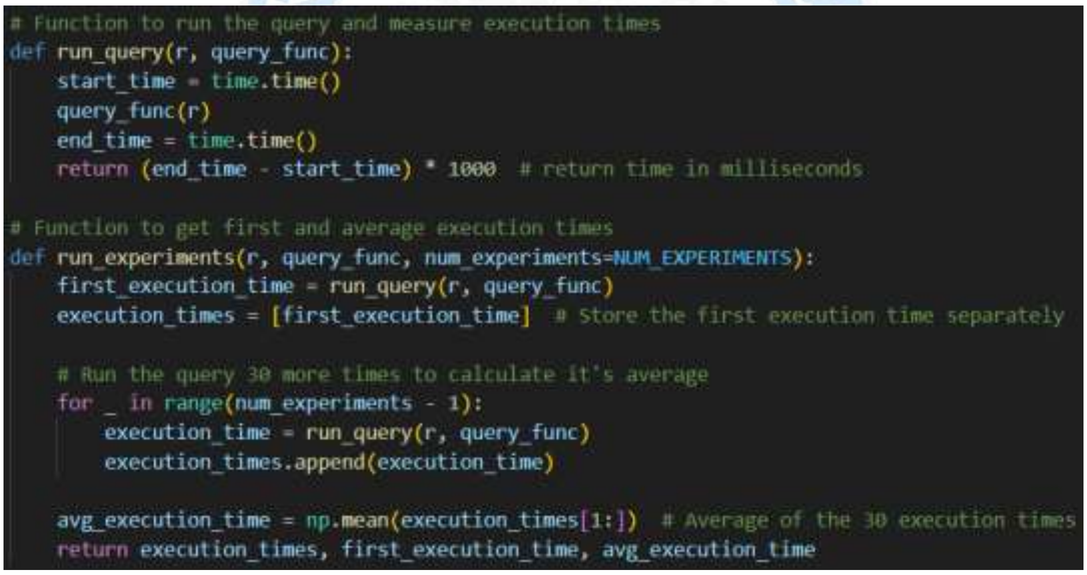
***For Redis:***

We first connect to a Redis database and load *student, course, professor, and assignment* records from the CSV file using Batch Processing. The data is stored using **Redis Hashes** for individual records and **Sets/Zsets** for indexing relationships and scores. **Redis pipelines** and **batch processing** are used to optimize insertion performance.



```python
import pandas as pd
import redis
import time
import os
import numpy as np

# Redis connection
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 0

dataset = '1_mil_records.csv'
NUM_EXPERIMENTS = 31

# Function to connect to Redis
def connect_to_db():
    r = redis.StrictRedis(host=REDIS_HOST, port=REDIS_PORT, db=REDIS_DB, decode_responses=True)
    return r
```

We run four predefined queries 31 times and measure execution time to analyse its performance.



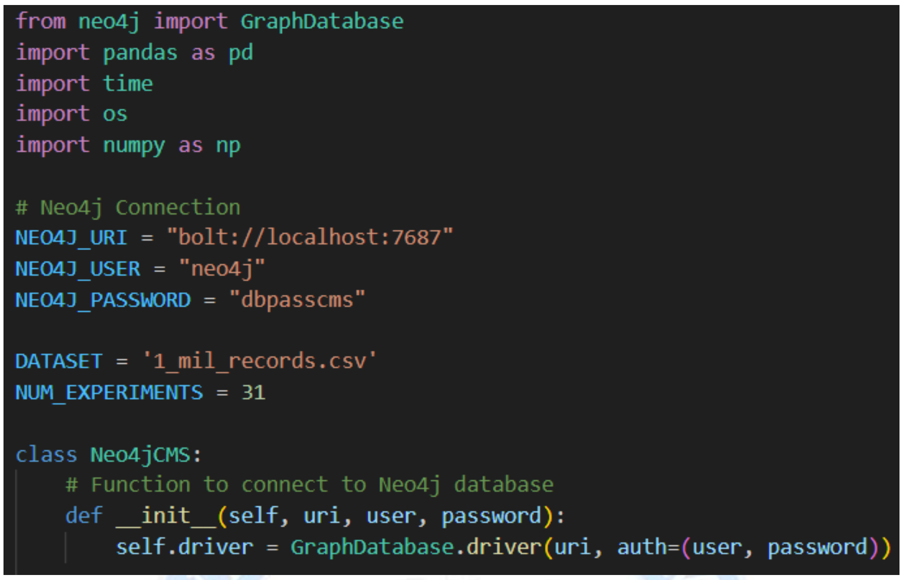
```python
# Function to run the query and measure execution times
def run_query(r, query_func):
    start_time = time.time()
    query_func(r)
    end_time = time.time()
    return (end_time - start_time) * 1000  # return time in milliseconds

# Function to get first and average execution times
def run_experiments(r, query_func, num_experiments=NUM_EXPERIMENTS):
    first_execution_time = run_query(r, query_func)
    execution_times = [first_execution_time]  # Store the first execution time separately

    # Run the query 30 more times to calculate it's average
    for _ in range(num_experiments - 1):
        execution_time = run_query(r, query_func)
        execution_times.append(execution_time)

    avg_execution_time = np.mean(execution_times[1:])  # Average of the 30 execution times
    return execution_times, first_execution_time, avg_execution_time
```

We run the experiments for different dataset sizes (250K, 500K, 750K, and 1M records). The results, including **first execution time** and **average execution time** across 30 subsequent runs, are saved in an Excel file for further analysis. Before inserting a new dataset, the Redis database is flushed to ensure clean and consistent testing.

```python
# Clear Redis database before inserting new records
r.flushdb()
print(f"Redis database cleared.")
```

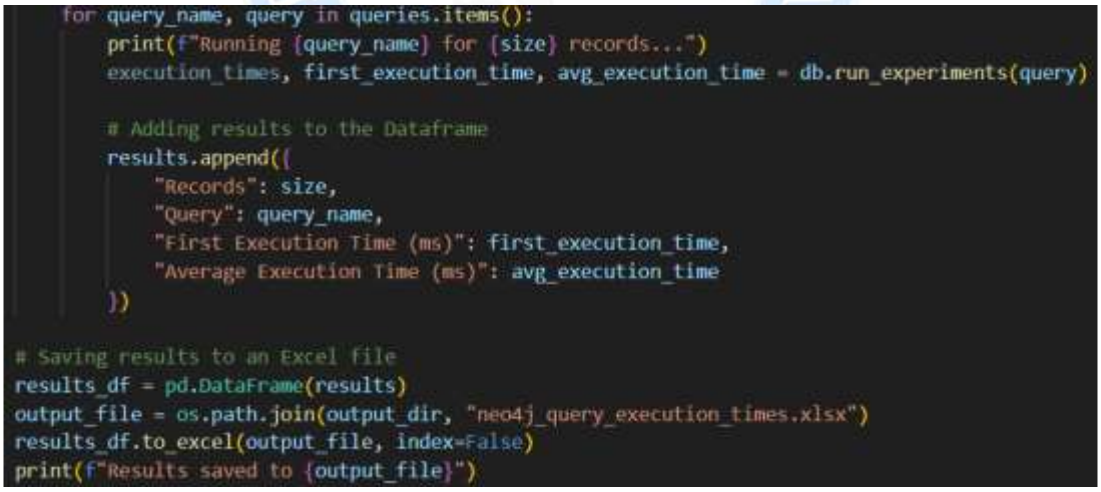
***For neo4j:***

We begin by connecting to a Neo4j instance and then proceed to clear the existing database using a Cypher query that deletes all nodes and relationships then, we load the dataset from the CSV file.



```python
from neo4j import GraphDatabase
import pandas as pd
import time
import os
import numpy as np

# Neo4j Connection
NEO4J_URI = "bolt://localhost:7687"
NEO4J_USER = "neo4j"
NEO4J_PASSWORD = "dbpasscms"

DATASET = '1_mil_records.csv'
NUM_EXPERIMENTS = 31

class Neo4jCMS:
    # Function to connect to Neo4j database
    def __init__(self, uri, user, password):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))
```

The data is inserted into the database in **batches**. Once the data is inserted, we run four predefined queries 31 times and record the **first execution time** and the **average execution time** for each query. The results are then stored in an Excel file



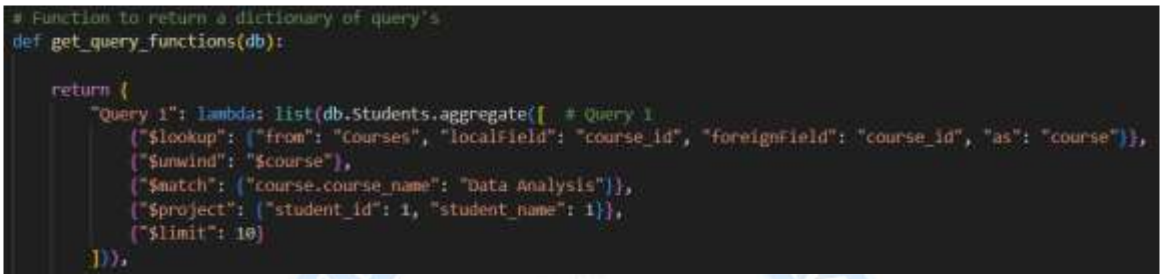
```python
    for query_name, query in queries.items():
        print(f"Running {query_name} for {size} records...")
        execution_times, first_execution_time, avg_execution_time = db.run_experiments(query)

        # Adding results to the Dataframe
        results.append({
            "Records": size,
            "Query": query_name,
            "First Execution Time (ms)": first_execution_time,
            "Average Execution Time (ms)": avg_execution_time
        })

# Saving results to an Excel file
results_df = pd.DataFrame(results)
output_file = os.path.join(output_dir, "neo4j_query_execution_times.xlsx")
results_df.to_excel(output_file, index=False)
print(f"Results saved to {output_file}")
```

## Queries:

We analyse the execution times of queries across multiple database systems, including MySQL, MongoDB, Neo4j, Redis, and Cassandra. We perform experiments by running predefined queries on datasets of increasing sizes (250K, 500K, 750K, and 1M records) and measure both the first execution time and the average execution time over multiple runs. The results help evaluate the performance of each database in handling different types of queries.

**Query 1: Retrieve 10 Students Enrolled in "Data Analysis".**

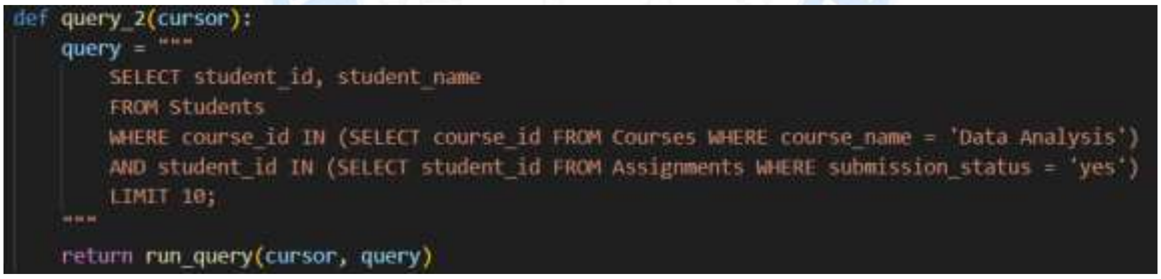
We executed a query to fetch 10 student IDs and names from the Students table who are enrolled in the "Data Analysis" course using 'course_id'.



```
# Function to return a dictionary of query's
def get_query_functions(db):

    return {
        "Query 1": lambda: list(db.Students.aggregate([  # Query 1
            {"$lookup": {"from": "Courses", "localField": "course_id", "foreignField": "course_id", "as": "course"}},
            {"$unwind": "$course"},
            {"$match": {"course.course_name": "Data Analysis"}},
            {"$project": {"student_id": 1, "student_name": 1}},
            {"$limit": 10}
        ])),
```

**Query 2: Retrieve 10 Students who submitted Assignments and are Enrolled in "Data Analysis".**

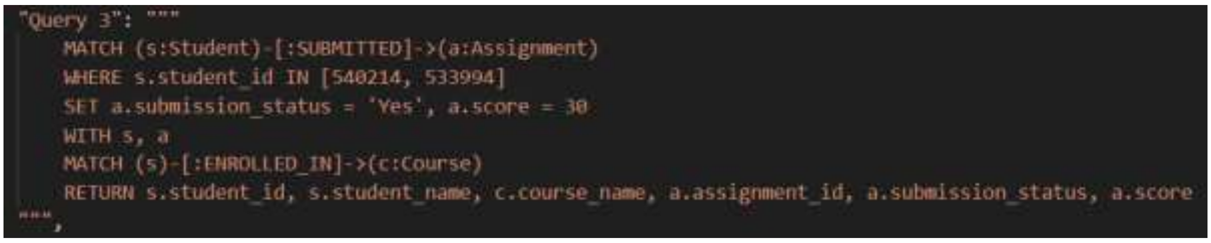
We executed a query to fetch 10 student IDs and names from the Students table who submitted assignments who has 'submission_status' in the Assignments table as "Yes" and who is also enrolled in the "Data Analysis" course using 'course_id'.



```
def query_2(cursor):
    query = """
        SELECT student_id, student_name
        FROM Students
        WHERE course_id IN (SELECT course_id FROM Courses WHERE course_name = 'Data Analysis')
        AND student_id IN (SELECT student_id FROM Assignments WHERE submission_status = 'yes')
        LIMIT 10;
    """
    return run_query(cursor, query)
```

**Query 3: Update Assignment Status and Retrieve the Updated Data for two students.**

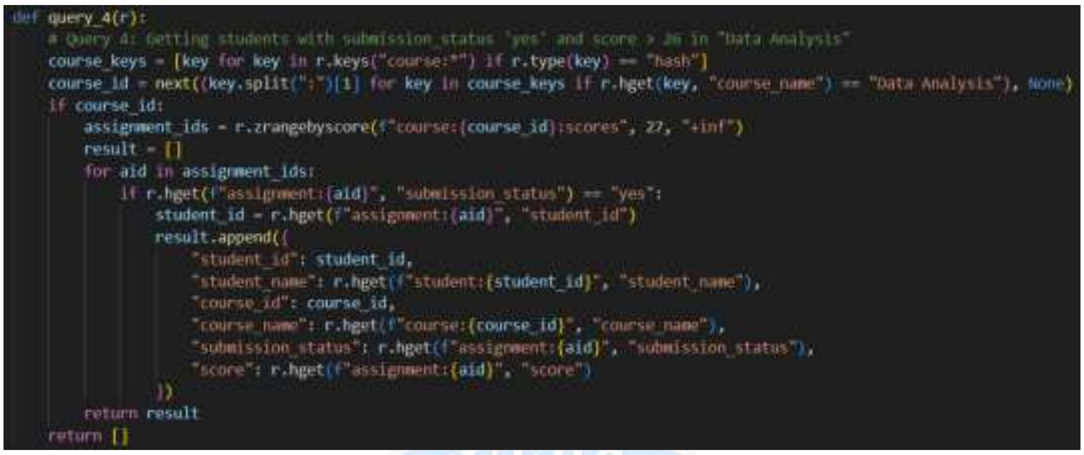
We executed a query to update the assignment status i.e., 'submission_status' to yes and score to 30 for two specific students having 'student_id' - "533994" and "540214". After updating, we executed the query to retrieve their 'student_id', 'student_name' from Students table; 'course_id', 'course_name' from Courses table; and 'assignment_id', 'submission_status', 'score' from Assignments table.



```
"Query 3": """
    MATCH (s:Student)-[:SUBMITTED]->(a:Assignment)
    WHERE s.student_id IN [540214, 533994]
    SET a.submission_status = 'Yes', a.score = 30
    WITH s, a
    MATCH (s)-[:ENROLLED_IN]->(c:Course)
    RETURN s.student_id, s.student_name, c.course_name, a.assignment_id, a.submission_status, a.score
""",
```

**Query 4: Retrieve Students with Submission Status “Yes” and score “>26”.**

We executed a query to retrieve the students’ ‘student_id’, ‘student_name’ from Students table; ‘course_id’, ‘course_name’ from Courses table; and ‘submission_status’, ‘score’ from Assignments table.



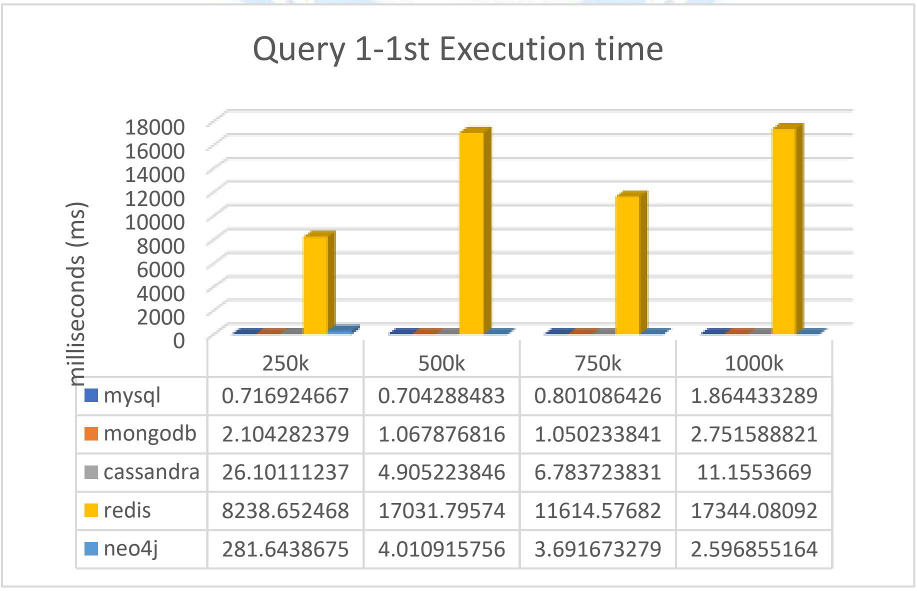
```python
def query_4(r):
    # Query 4: Getting students with submission_status 'yes' and score > 26 in "Data Analysis"
    course_keys = [key for key in r.keys("course:*") if r.type(key) == "hash"]
    course_id = next((key.split(":")[1] for key in course_keys if r.hget(key, "course_name") == "Data Analysis"), None)
    if course_id:
        assignment_ids = r.zrangebyscore(f"course:{course_id}:scores", 27, "+inf")
        result = []
        for aid in assignment_ids:
            if r.hget(f"assignment:{aid}", "submission_status") == "yes":
                student_id = r.hget(f"assignment:{aid}", "student_id")
                result.append({
                    "student_id": student_id,
                    "student_name": r.hget(f"student:{student_id}", "student_name"),
                    "course_id": course_id,
                    "course_name": r.hget(f"course:{course_id}", "course_name"),
                    "submission_status": r.hget(f"assignment:{aid}", "submission_status"),
                    "score": r.hget(f"assignment:{aid}", "score")
                })
        return result
    return []
```

Each query was executed 31 times, and the first execution time along with the average execution time for the next 30 experiments were recorded accordingly. The results were saved in an excel sheet for each database for performance analysis.

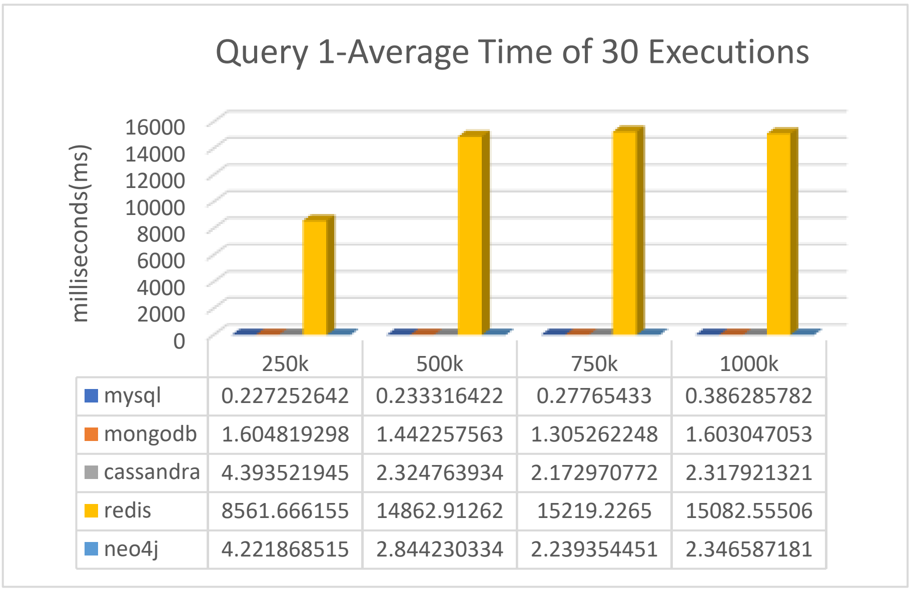
## Statistics

We plotted histograms for the first execution time and average execution time of subsequent 30 executions of the 4 queries.

**Query 1:**

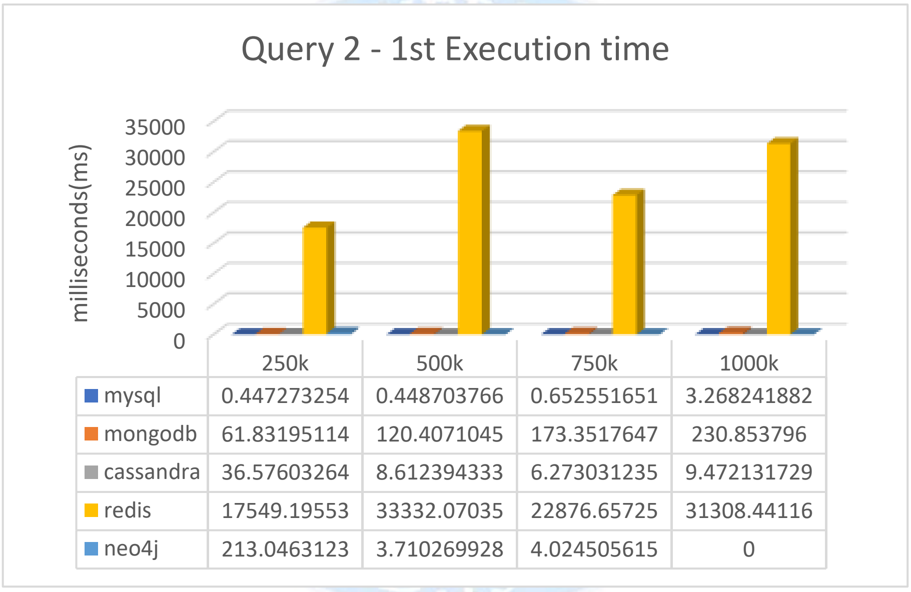


| | 250k | 500k | 750k | 1000k |
|---|---|---|---|---|
| mysql | 0.716924667 | 0.704288483 | 0.801086426 | 1.864433289 |
| mongodb | 2.104282379 | 1.067876816 | 1.050233841 | 2.751588821 |
| cassandra | 26.10111237 | 4.905223846 | 6.783723831 | 11.1553669 |
| redis | 8238.652468 | 17031.79574 | 11614.57682 | 17344.08092 |
| neo4j | 281.6438675 | 4.010915756 | 3.691673279 | 2.596855164 |

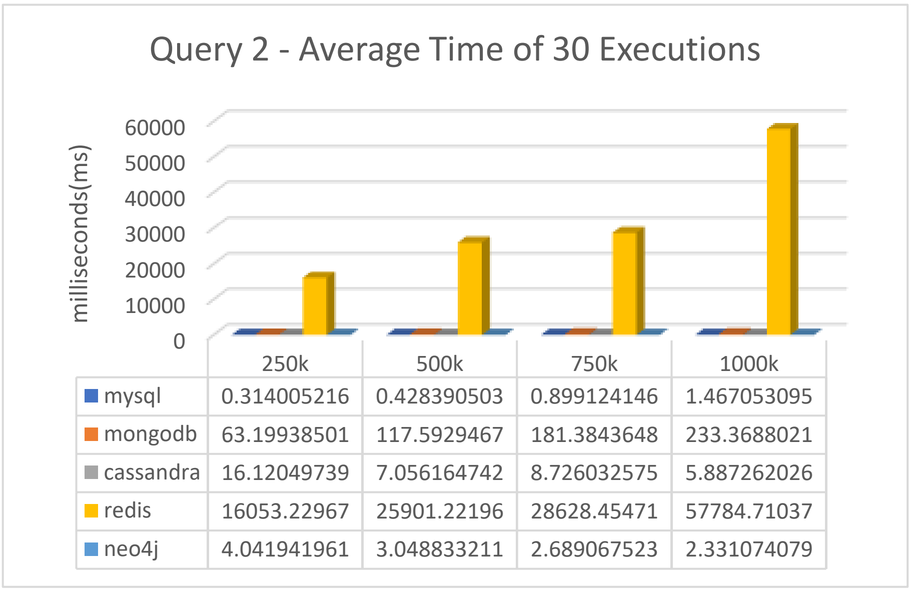


| | 250k | 500k | 750k | 1000k |
|---|---|---|---|---|
| 🟦 mysql | 0.227252642 | 0.233316422 | 0.27765433 | 0.386285782 |
| 🟧 mongodb | 1.604819298 | 1.442257563 | 1.305262248 | 1.603047053 |
| ◼ cassandra | 4.393521945 | 2.324763934 | 2.172970772 | 2.317921321 |
| 🟨 redis | 8561.666155 | 14862.91262 | 15219.2265 | 15082.55506 |
| 🟦 neo4j | 4.221868515 | 2.844230334 | 2.239354451 | 2.346587181 |

**Query 2:**

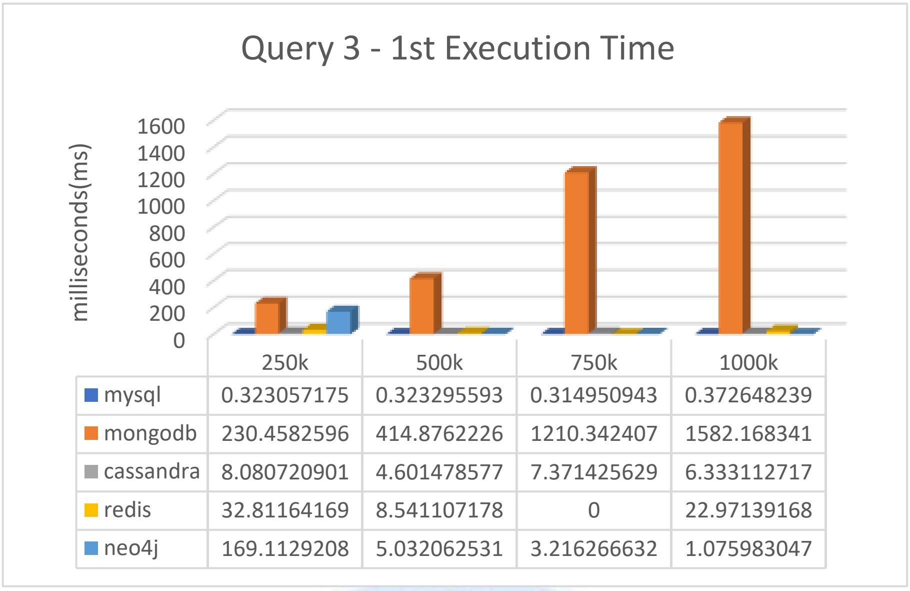


| | 250k | 500k | 750k | 1000k |
|---|---|---|---|---|
| 🟦 mysql | 0.447273254 | 0.448703766 | 0.652551651 | 3.268241882 |
| 🟧 mongodb | 61.83195114 | 120.4071045 | 173.3517647 | 230.853796 |
| ◼ cassandra | 36.57603264 | 8.612394333 | 6.273031235 | 9.472131729 |
| 🟨 redis | 17549.19553 | 33332.07035 | 22876.65725 | 31308.44116 |
| 🟦 neo4j | 213.0463123 | 3.710269928 | 4.024505615 | 0 |

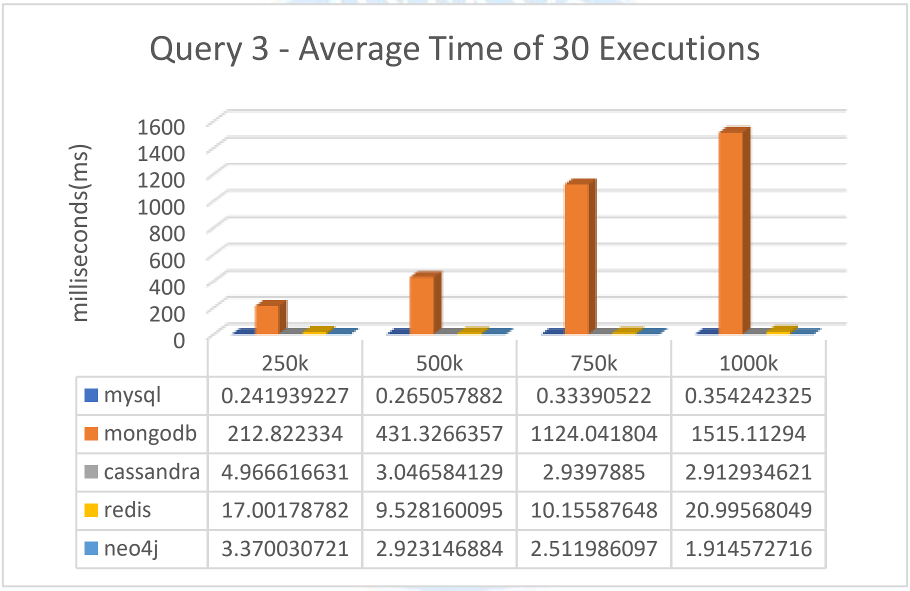


| | 250k | 500k | 750k | 1000k |
|---|---|---|---|---|
| 🟦 mysql | 0.314005216 | 0.428390503 | 0.899124146 | 1.467053095 |
| 🟧 mongodb | 63.19938501 | 117.5929467 | 181.3843648 | 233.3688021 |
| ◼ cassandra | 16.12049739 | 7.056164742 | 8.726032575 | 5.887262026 |
| 🟨 redis | 16053.22967 | 25901.22196 | 28628.45471 | 57784.71037 |
| 🟦 neo4j | 4.041941961 | 3.048833211 | 2.689067523 | 2.331074079 |

**Query 3:**

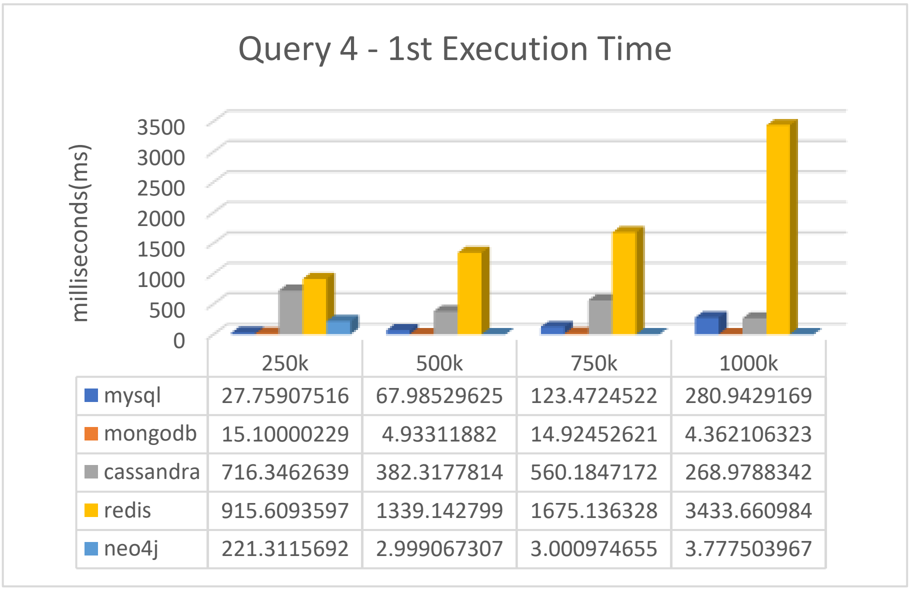


| | 250k | 500k | 750k | 1000k |
|---|---|---|---|---|
| 🟦 mysql | 0.323057175 | 0.323295593 | 0.314950943 | 0.372648239 |
| 🟧 mongodb | 230.4582596 | 414.8762226 | 1210.342407 | 1582.168341 |
| ⬜ cassandra | 8.080720901 | 4.601478577 | 7.371425629 | 6.333112717 |
| 🟨 redis | 32.81164169 | 8.541107178 | 0 | 22.97139168 |
| 🟦 neo4j | 169.1129208 | 5.032062531 | 3.216266632 | 1.075983047 |

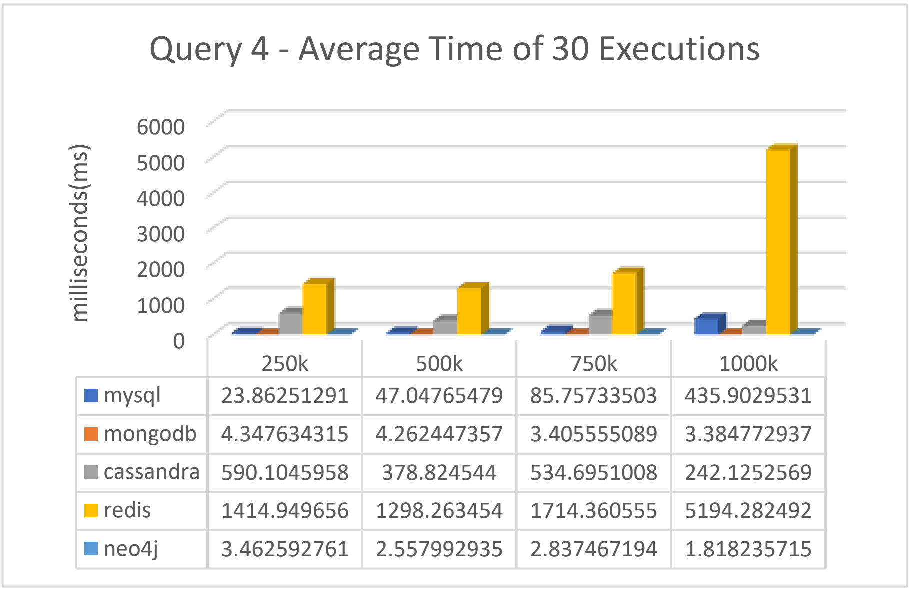


| | 250k | 500k | 750k | 1000k |
|---|---|---|---|---|
| 🟦 mysql | 0.241939227 | 0.265057882 | 0.33390522 | 0.354242325 |
| 🟧 mongodb | 212.822334 | 431.3266357 | 1124.041804 | 1515.11294 |
| ⬜ cassandra | 4.966616631 | 3.046584129 | 2.9397885 | 2.912934621 |
| 🟨 redis | 17.00178782 | 9.528160095 | 10.15587648 | 20.99568049 |
| 🟦 neo4j | 3.370030721 | 2.923146884 | 2.511986097 | 1.914572716 |

**Query 4:**



| | 250k | 500k | 750k | 1000k |
|---|---|---|---|---|
| 🟦 mysql | 27.75907516 | 67.98529625 | 123.4724522 | 280.9429169 |
| 🟧 mongodb | 15.10000229 | 4.93311882 | 14.92452621 | 4.362106323 |
| ⬜ cassandra | 716.3462639 | 382.3177814 | 560.1847172 | 268.9788342 |
| 🟨 redis | 915.6093597 | 1339.142799 | 1675.136328 | 3433.660984 |
| 🟦 neo4j | 221.3115692 | 2.999067307 | 3.000974655 | 3.777503967 |



| | 250k | 500k | 750k | 1000k |
|---|---|---|---|---|
| ■ mysql | 23.86251291 | 47.04765479 | 85.75733503 | 435.9029531 |
| ■ mongodb | 4.347634315 | 4.262447357 | 3.405555089 | 3.384772937 |
| ■ cassandra | 590.1045958 | 378.824544 | 534.6951008 | 242.1252569 |
| ■ redis | 1414.949656 | 1298.263454 | 1714.360555 | 5194.282492 |
| ■ neo4j | 3.462592761 | 2.557992935 | 2.837467194 | 1.818235715 |

## Problems Encountered and Solved

1. We first created 4 separate datasets and when we were performing experiments with all 4 four datasets it was consuming a lot of time and was not very efficient. We then created only one dataset of 1 million records and defined a function in the python script which takes subsets of 250000, 500000, 750000, 1000000 from the original dataset to perform the experiments. This increased the efficiency of the project and reduced the number of python files.
2. Initially, we faced issues running the MySQL Python script due to dependency and environment conflicts. To resolve this, we created a **Dockerfile** using the *python:3.12-slim* image, setting up a working directory and copying the script and dataset. We installed required libraries like *pymysql, pandas, cryptography*, and *openpyxl* to ensure smooth execution. The script was then executed inside the container using *CMD ["python", "/app/mysqlcms1mil.py"]*. Finally, we built and ran the container with ***docker build -t python-mysql .*** and ***docker run --rm --network cms-network python-mysql***, making execution seamless.
3. We faced issues with running the python scripts of MongoDB, Cassandra, Redis, Neo4j as they were inserting records one by one which took way too much time so, we implemented batch processing in all the scripts which inserts the data in batches. This improved the efficiency and performance of the databases.
4. We deleted the database between each experiment before inserting the next dataset to give it a fresh start which provides consistent and accurate performance results. Comparatively, it in fact took way less time to run all the experiments after this change. For MongoDB instead of deleting the entire database we only dropped the collections between each experiment, which is a much better option for the other databases too.

5. Since Cassandra and Redis didn't support Nested selects and Joins, we defined python functions for each query which is used as a workaround for their lack of Joins and Nested Selects. In these functions the queries are broken down into multiple steps and executes sequentially within the functions.

## Conclusion

This study analyzed the performance of five different database management systems MySQL, MongoDB, Neo4j, Redis, and Cassandra in handling a Course Management System (CMS). By executing predefined queries across increasing dataset sizes (250K, 500K, 750K, and 1M records) and measuring both first execution time and average execution time, we evaluated the efficiency of each database under different workloads.

To optimize performance, batch processing was implemented in all scripts, significantly reducing data insertion times. Additionally, databases were reset between experiments to ensure consistency, and for MongoDB, dropping collections instead of deleting the entire database improved efficiency. Workarounds were also applied for Cassandra and Redis, which do not support nested selects or joins, by breaking queries into multiple sequential steps.

The results revealed that **Neo4j**, **MySQL** and **Cassandra** performed significantly whereas Redis and MongoDB comparatively haven't performed well in handling insertion efficiency and struggled with complex queries.