

# **"Driving Efficiency: Performance Criteria in Student Information System Database Selection"**

**Database Mod B**

**Professor:**

**Armando Ruggeri**

**Student:**

**Mohammad Ali Mumtaz**

**ID: 539454**

# Introduction

- This project aims to develop an innovative Student Information System (SIS) using a NoSQL database to adapt to the changing landscape of educational data management. The main goal is to create a dynamic platform that enables faculty and administrators to efficiently manage student data in a holistic manner. The system will not only store data but also incorporate advanced features to handle various types of student information like grades, attendance records, and demographic details. By utilizing the flexibility and scalability of NoSQL technology, the project seeks to build a responsive solution capable of adjusting to the dynamic nature of educational data.
- This report serves as an extensive guide, outlining the methodology employed to implement a specialized NoSQL database for the Student Information System (SIS). The primary objective is to streamline data management while catalyzing a significant shift in administrative capabilities within the educational domain. By dissecting the intricacies of this database project, we aim to explore its numerous benefits, including improved accessibility, scalability, and a deeper insight into student performance. Through this initiative, we aim to contribute meaningfully to the convergence of technology and education, fostering an environment conducive to academic achievement and administrative efficiency.

# Background

For a long time, relational databases have been dominating the market when it comes to data storage and management. However, the growth of the amount of data that is needed to be stored and managed in these databases increased the need for scalability and performance.

- Therefore, NoSQL or "Not Only SQL" databases have been introduced and emerged as a new way to deal with data storage. It provided more efficient solutions for numerous use cases. Big Data is the term used for a significantly large amount of structured or unstructured data that cannot, in some cases, be handled with traditional databases .
- The term NoSQL was used for the first time in 1988 by Carlo Strozzi when he presented his relational database that does not implement the SQL interface. Then, the term started referring to databases that do not use "RDBMS" meaning they do not use tables or fixed schema for storing and managing the data. That was the most significant shift from the relational databases that used to define the schema at the beginning of the process which made it harder to maintain when data needed to evolve.
- Relational databases and their proprieties caused some limitations in handling vast amounts of data. These limitations arose after some big companies like Facebook and Google demanded solutions for their services that support a massive number of users. These limitations have resulted in faster moving toward non-relational databases.

# Problem formulation

NoSQL is classified into four main technologies, and each of them is designed to meet some use cases scenarios and offer solutions for them . There are hundreds of databases that belong to these categories depending on their data model storage.

Henceforth, several pieces of research have been conducted to compare these technologies with relational databases, bearing in mind the performance in different situations.

Consequently, to deliver high-quality service, the communications between the databases and the application are always a crucial matter to consider. These communications are mainly done by four operations which are **INSERT**, **UPDATE**, **SELECT**, and **DELETE**.

Applications generally unevenly use these operations, whereas an application may need to ensure the high performance of one or two of them over the other. Therefore, this experiment evaluates the performance of the chosen databases by calculating the time each of these operations takes to complete the task

# Objectives

- The main goal of this work is to compare the performance of different technologies of NoSQL in order to observe their behavior in different situations and conditions. Additionally, the developed applications in the experiment must be reusable to facilitate future work on top of this one.
- To determine how the data type influences the performance of the selected databases.
- To find how the amount of data influences the performance of the selected databases.
- To find how the various NoSQL databases perform compared to each other against various operations.

# Database Comparison

## MySQL

- MySQL is a classic relational database known for its structured schema.
- Ideal for situations involving organized data with clearly defined relationships.
- Capable of handling complex queries, aggregations, and joins.
- Horizontal scalability may require more resources compared to certain 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

## 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



# • DATA GENERATION Overview:

- This code effectively generated synthetic student data, totaling 1,000,000 records.
- By leveraging the Faker library for realistic names and random values for attributes such as age, grades, and attendance, the dataset includes essential student information.
- The generated data is intelligently segmented into four CSV files, each representing different portions of the dataset with 250,000, 500,000, 750,000, and 1,000,000 records respectively.
- These CSV files, now conveniently stored in the specified directory, serve as representative datasets for various testing scenarios or database population. The completion message confirms the successful generation and storage of this synthetic student data.

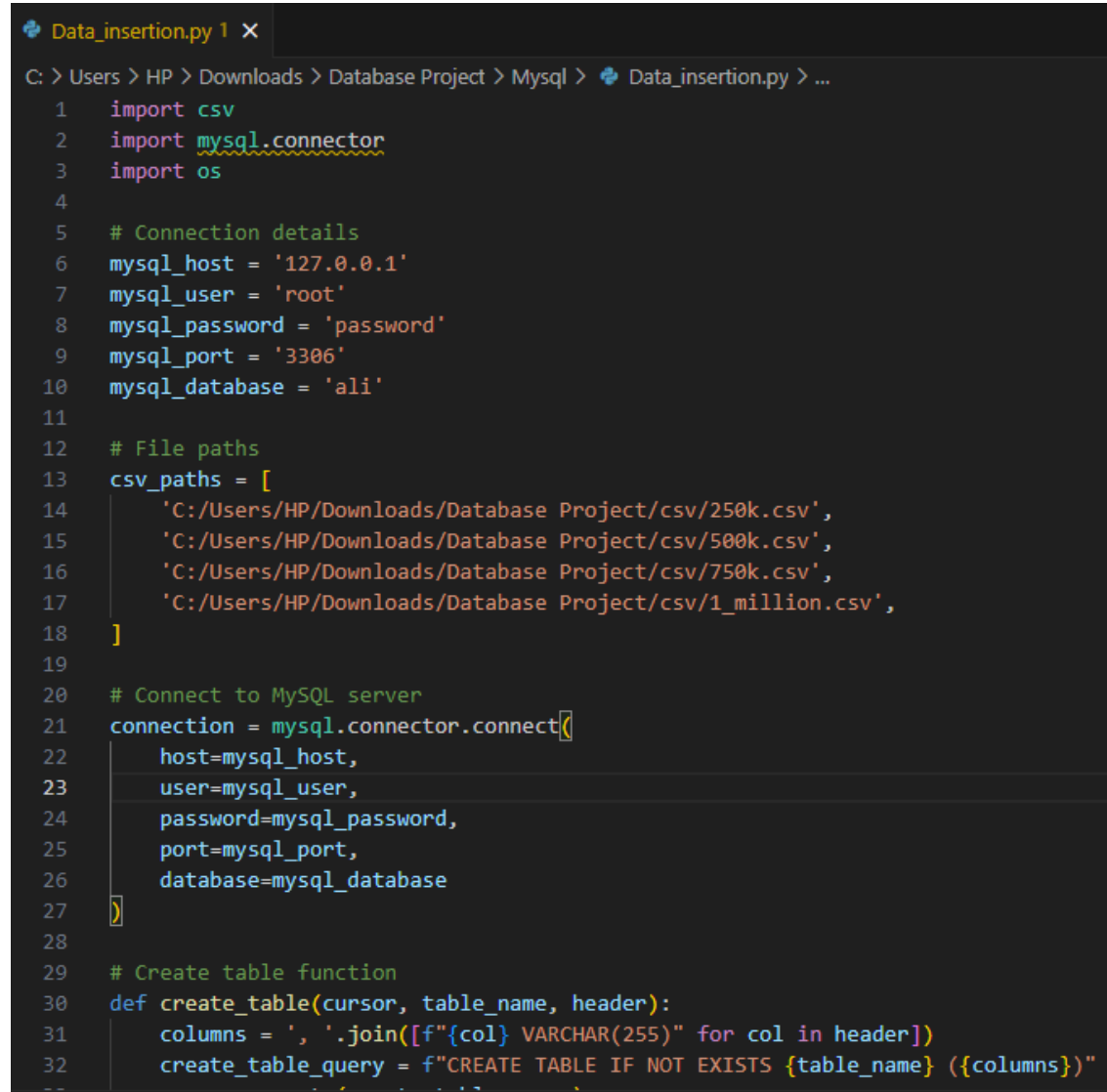
```
data_generator
C: > Users > HP > Downloads > Database Project > data_generator > generate_student_data

1  import pandas as pd
2  from faker import Faker
3  import random
4
5  fake = Faker()
6
7  def generate_student_data(num_records):
8      student_ids = set()
9      data = {}
10         'StudentID': [],
11         'FirstName': [],
12         'LastName': [],
13         'Age': [],
14         'Grade': [],
15         'Attendance': [],
16     }
17
18     while len(data['StudentID']) < num_records:
19         student_id = f'ST{str(len(data["StudentID"]) + 1).zfill(6)}'
20         if student_id not in student_ids:
21             student_ids.add(student_id)
22             data['StudentID'].append(student_id)
23             data['FirstName'].append(fake.first_name())
24             data['LastName'].append(fake.last_name())
25             data['Age'].append(random.randint(18, 25))
26             data['Grade'].append(random.randint(60, 100))
27             data['Attendance'].append(random.randint(0, 100))
28
29     return pd.DataFrame(data)
30
31 csv_file_path = 'C:/Users/HP/Downloads/Database Project/csv_mongo/'
32
```

# SCHEMAS AND INSERTING DATA

## MYSQL:

- To utilize MySQL, acquiring MySQL Workbench is recommended.
- The decision to use MySQL is subjective; I personally selected it for its intuitive interface.
- Once the connection is established, datasets can be imported directly.
- However, I opted to utilize VSCode , and here's a visual representation of that choice.



```
Data_insertion.py 1 X
C: > Users > HP > Downloads > Database Project > Mysql > Data_insertion.py > ...

1 import csv
2 import mysql.connector
3 import os
4
5 # Connection details
6 mysql_host = '127.0.0.1'
7 mysql_user = 'root'
8 mysql_password = 'password'
9 mysql_port = '3306'
10 mysql_database = 'ali'
11
12 # File paths
13 csv_paths = [
14     'C:/Users/HP/Downloads/Database Project/csv/250k.csv',
15     'C:/Users/HP/Downloads/Database Project/csv/500k.csv',
16     'C:/Users/HP/Downloads/Database Project/csv/750k.csv',
17     'C:/Users/HP/Downloads/Database Project/csv/1_million.csv',
18 ]
19
20 # Connect to MySQL server
21 connection = mysql.connector.connect(
22     host=mysql_host,
23     user=mysql_user,
24     password=mysql_password,
25     port=mysql_port,
26     database=mysql_database
27 )
28
29 # Create table function
30 def create_table(cursor, table_name, header):
31     columns = ', '.join([f"{col} VARCHAR(255)" for col in header])
32     create_table_query = f"CREATE TABLE IF NOT EXISTS {table_name} ({columns})"
```

# Cassandra

- This Python script connects to a local Cassandra database, creates a keyspace and table if they don't exist, and inserts student data from four CSV files.
- The CSV data is structured into a predefined Cassandra table with columns (id, StudentID, FirstName, LastName, Age, Grade, Attendance).
- The script prints messages for each successful data insertion and any encountered errors, providing an overview of the process. Finally, it closes the Cassandra session and cluster connections.

```
inserting2.py 3 X
C: > Users > HP > Downloads > Database Project > Cassandra > inserting2.py > ...

1  from cassandra.cluster import Cluster
2  from cassandra.auth import PlainTextAuthProvider
3  from cassandra.query import BatchStatement
4  import pandas as pd
5  import uuid
6
7  # Define Cassandra connection parameters
8  cassandra_host = '172.18.0.3'
9  cassandra_port = 9042
10 cassandra_username = 'MyDatabase' # Username from the credentials
11 cassandra_password = 'Password' # Password from the credentials
12
13 # Create PlainTextAuthProvider for authentication
14 auth_provider = PlainTextAuthProvider(username=cassandra_username, password=cassandra_password)
15
16 # Create a Cluster instance
17 cluster = Cluster([cassandra_host], port=cassandra_port, auth_provider=auth_provider)
18
19 # Connect to the Cassandra cluster
20 session = cluster.connect()
21
22 # Define keyspace name
23 keyspace_name = 'my_keyspace'
24
25 # Define table names
26 table_names = ['table_250k', 'table_500k', 'table_750k', 'table_1million']
27
28 # Create keyspace if not exists
29 session.execute(f"CREATE KEYSPACE IF NOT EXISTS {keyspace_name} WITH replication = {{'class': 'Si
30
31 # Use the keyspace
32 session.execute(f"USE {keyspace_name}")
33
```

# Redis

- This Python script connects to a local Redis database and inserts student data from four CSV files (named '250k.csv', '500k.csv', '750k.csv', and '1\_million.csv').
- Each CSV file is read into a Pandas Data Frame, converted into JSON format, and stored in Redis with a unique key ('data\_set\_1', 'data\_set\_2', etc.).
- The script prints a confirmation message for each dataset insertion and closes the Redis connection.
- The final output indicates the completion of data insertion into the Redis database.

inserting\_data.py 1 X

C:\> Users > HP > Downloads > Database Project > Redis > inserting\_data.py > ...

```
1  import os
2  import redis
3  import pandas as pd
4  import json
5
6  # Define Redis connection parameters
7  redis_host = '127.0.0.1'
8  redis_port = 6379
9  redis_password = None
10
11 # Connect to Redis
12 r = redis.Redis(host=redis_host, port=redis_port, password=redis_password)
13
14 # Directory containing CSV files
15 csv_directory = r'C:\Users\HP\Downloads\Database Project\csv'
16
17 # List of CSV files
18 csv_files = [
19     '250k.csv',
20     '500k.csv',
21     '750k.csv',
22     '1_million.csv',
23 ]
24
25 # Iterate through CSV files
26 for idx, csv_file in enumerate(csv_files, start=1):
27     key = f'data_set_{idx}' # Redis key for storing data
28     csv_path = os.path.join(csv_directory, csv_file)
29
30     # Check if the file exists
31     if os.path.exists(csv_path):
32         # Read CSV file into a DataFrame
```

# Neo4j

- The code efficiently inserts data from CSV files into a Neo4j graph database using the Neo4j Python driver.
- It establishes a connection to the Neo4j database, reads data from CSV files into Pandas DataFrames, and iteratively creates nodes with specified labels and properties in the database using Cypher queries.
- The code handles multiple data sets, accommodating various sizes by assigning corresponding labels.
- Upon completion, it confirms the data insertion, demonstrating a systematic approach for managing large-scale graph data sets effectively.

```
inserting_data.py X
C:\> Users > HP > Downloads > Database Project > neo4j > inserting_data.py > ...
1 from neo4j import GraphDatabase
2 import pandas as pd
3 import os
4
5 URI = "bolt://localhost:7687"
6 USERNAME = "neo4j"
7 PASSWORD = "password"
8
9 def create_label_and_insert_data(driver, label, csv_file_path):
10     df = pd.read_csv(csv_file_path)
11     with driver.session() as session:
12         # Create label for the nodes
13         session.run(f"CREATE CONSTRAINT FOR (p:{label}) REQUIRE p.StudentID IS UNIQUE")
14
15         # Insert data into nodes with the specified label
16         for index, row in df.iterrows():
17             session.run(f"""
18                 CREATE (:{label} {{
19                     StudentID: $student_id,
20                     FirstName: $first_name,
21                     LastName: $last_name,
22                     Age: $age,
23                     Grade: $grade,
24                     Attendance: $attendance
25                 }})
26             """,
27                 student_id=row['StudentID'],
28                 first_name=row['FirstName'],
29                 last_name=row['LastName'],
30                 age=row['Age'],
31                 grade=row['Grade'],
32                 attendance=row['Attendance'])
33
```

# MongoDb

- The provided code facilitates the insertion of data from CSV files into MongoDB collections using the PyMongo library. It begins by creating a MongoClient object to connect to the local MongoDB instance.
- The `insert_data_to_mongo` function reads CSV files located at specified file paths and converts them into Pandas DataFrames.
- These DataFrames are then transformed into lists of dictionaries, with each dictionary representing a record in the CSV file.
- These records are subsequently inserted into the MongoDB collections specified by the `collection_name` parameter.
- Once the insertion process is complete, the MongoClient connection is closed to release resources.

inserting\_data.py

C:\> Users > HP > Downloads > Database Project > Mongoddb > inserting\_data.py

```
1  import pandas as pd
2  from pymongo import MongoClient
3
4  def insert_data_to_mongo(file_path, collection_name):
5      # Create a client object to connect to MongoDB
6      client = MongoClient("mongodb://localhost:27017/")
7      db = client["MyDatabase"] # Access the database
8
9      print(f"Reading CSV file from: {file_path}")
10     df = pd.read_csv(file_path)
11     records = df.to_dict(orient='records')
12     db[collection_name].insert_many(records)
13     print(f"Data inserted into {collection_name} collection")
14
15     client.close() # Close the MongoDB connection
16
17 csv_file_path = 'C:/Users/HP/Downloads/Database Project/CSV Files/'
18
19 insert_data_to_mongo(f'{csv_file_path}250k.csv', '250k')
20 insert_data_to_mongo(f'{csv_file_path}500k.csv', '500k')
21 insert_data_to_mongo(f'{csv_file_path}750k.csv', '750k')
22 insert_data_to_mongo(f'{csv_file_path}1_million.csv', '1million')
23
```



# Queries

## MySQL

- The code conducts performance analysis on MySQL databases by executing predefined SQL queries across datasets of varying sizes.
- It establishes connections to MySQL using specified parameters and executes queries targeting specific data subsets, including student grades, attendance, and first names.
- Multiple executions of each query measure execution times, with averages calculated to evaluate database performance across different scenarios.
- Results are stored in CSV files categorized by dataset size, enabling comprehensive analysis and comparison.
- This systematic approach facilitates insights into MySQL database efficiency, supporting optimization and query tuning endeavors.

```
Queries.py 1 X
C: > Users > HP > Downloads > Database Project > Mysql > Queries.py > ...

1  import time
2  import mysql.connector
3  import os
4  import csv
5
6  mysql_host = '127.0.0.1'
7  mysql_user = 'root'
8  mysql_password = 'password'
9  mysql_port = '3306'
10 mysql_database = 'ali'
11
12 table_names = ['250k', '500k', '750k', '1_million']
13
14 mysql_queries = [
15     "SELECT * FROM {table_name} WHERE Grade >= 70 AND Attendance >= 50",
16     "SELECT * FROM {table_name} WHERE Grade >= 70",
17     "SELECT * FROM {table_name} WHERE FirstName LIKE 'A%'",
18     "SELECT * FROM {table_name}"
19 ]
20
21 base_directory = 'C:/Users/HP/Downloads/Database Project/Queries_result'
22 os.makedirs(base_directory, exist_ok=True)
23
24 connection = mysql.connector.connect(
25     host=mysql_host,
26     user=mysql_user,
27     password=mysql_password,
28     port=mysql_port,
29     database=mysql_database
30 )
31
32 cursor = connection.cursor()
```

# Cassandra

- The provided code conducts performance analysis on a Cassandra database by executing predefined CQL (Cassandra Query Language) queries across tables of varying sizes. It begins by establishing a connection to the Cassandra cluster using specified connection parameters. For each table in the key space, the code creates an index on the 'Age' column to optimize query performance. It then defines four distinct CQL queries targeting different data subsets:
- Query 1: Retrieves the StudentID, FirstName, and LastName of ten students from the table.
- Query 2: Selects StudentID, FirstName, LastName, and Grade for students with grades greater than 90, limiting the result to ten entries. The ALLOW FILTERING directive enables filtering on non-indexed columns.
- Query 3: Fetches StudentID, FirstName, LastName, and Age for students younger than 20, with a limit of ten results. Again, ALLOW FILTERING facilitates filtering on the 'Age' column.
- Query 4: Retrieves StudentID, FirstName, LastName, and Attendance for ten students from the table.
- Each query is executed thirty times to measure execution times accurately. The results, including query names and execution times, are stored in CSV files organized by table and query type.

```
queries.py 1 x
C: > Users > HP > Downloads > Database Project > Cassandra > queries.py > ...

23
24 session.execute(f"""
25     CREATE INDEX IF NOT EXISTS idx_age ON {table_name} (Age);
26 """)
27
28 query_1 = f"""
29     SELECT StudentID, FirstName, LastName
30     FROM {table_name}
31     LIMIT 10;
32 """
33
34 query_2 = f"""
35     SELECT StudentID, FirstName, LastName, Grade
36     FROM {table_name}
37     WHERE Grade > 90
38     LIMIT 10 ALLOW FILTERING;
39 """
40
41 query_3 = f"""
42     SELECT StudentID, FirstName, LastName, Age
43     FROM {table_name}
44     WHERE Age < 20
45     LIMIT 10 ALLOW FILTERING;
46 """
47
48 query_4 = f"""
49     SELECT StudentID, FirstName, LastName, Attendance
50     FROM {table_name}
51     LIMIT 10;
52 """
53
```



# Redis

- It establishes a connection to Redis and verifies the connection's success. Four distinct queries are defined to retrieve specific data subsets based on criteria such as student names, grades, age, and attendance
- **Query 1 (Q1):** Retrieve all students from the dataset.
- **Query 2 (Q2):** Retrieve students whose first names start with a specific prefix (e.g., 'J').
- **Query 3 (Q3):** Retrieve students whose grades fall within a specified range (e.g., between 70 and 90).
- **Query 4 (Q4):** Retrieve students who meet specific age and attendance criteria (e.g., age  $\geq 18$  and attendance  $\geq 80\%$ ).
- Each query is executed 30 times to measure execution times accurately, and the results are stored in CSV files for further analysis. The code ensures correct folder and filename paths for

```
s.py 1 x
> HP > Downloads > Database Project > Redis > Queries.py > ...

execution_times = []

for query_index, query_description in enumerate(["Q1", "Q2", "Q3", "Q4"]):
    sanitized_query_description = sanitize_query_description(query_descri

    for i in range(30):
        start_time = time.perf_counter()

        if query_index == 0:
            all_students = redis_client.smembers(f'student_{set_name}')
        elif query_index == 1:
            name_prefix = 'J'
            students_with_name_prefix = [
                student for student in redis_client.smembers(f'student_{s
                    if redis_client.hget(f'student:{student.decode("utf-8"))_
            ]
        elif query_index == 2:
            min_grade = 70
            max_grade = 90
            students_with_grade_range = [
                student for student in redis_client.smembers(f'student_{s
                    if min_grade <= int(redis_client.hget(f'student:{student.
            ]
        elif query_index == 3:
            min_age = 18
            min_attendance_percentage = 80
            students_with_age_and_attendance = [
                student for student in redis_client.smembers(f'student_{s
                    if int(redis_client.hget(f'student:{student.decode("utf-8
                        int(redis_client.hget(f'student:{student.decode("utf-8"))
            ]
```

# Neo4j

- The Python script provided establishes a connection to a Neo4j database and executes a sequence of predefined queries across multiple datasets.
- The first query, labeled as Q1, focuses on retrieving student data where both the grade and attendance metrics meet certain criteria, likely aiming to identify high-performing students with consistent attendance.
- In contrast, the second query, Q2, narrows its scope to students with high grades regardless of attendance, possibly for academic performance analysis.
- Query Q3 shifts the focus to explore patterns in student names, specifically targeting those whose first names begin with the letter 'A'.
- Finally, query Q4 is a general retrieval query, seeking to obtain all student records from the dataset, likely for comprehensive analysis purposes.
- To capture performance variations, each query is executed 30 times, and the execution times are recorded. These results are then stored in CSV files,

```
Queries.py X
C: > Users > HP > Downloads > Database Project > neo4j > Queries.py > ...

1  from neo4j import GraphDatabase
2  import os
3  import time
4  import csv
5
6  URI = "bolt://localhost:7687"
7  USERNAME = "neo4j"
8  PASSWORD = "password"
9
10 # Define the Neo4j queries for each dataset
11 neo4j_queries = [
12     ("Q1", "MATCH (p:d_{dataset}) WHERE p.Grade >= 70 AND p.Attendance >= 50 RETURN p"),
13     ("Q2", "MATCH (p:d_{dataset}) WHERE p.Grade >= 70 RETURN p"),
14     ("Q3", "MATCH (p:d_{dataset}) WHERE p.FirstName STARTS WITH 'A' RETURN p"),
15     ("Q4", "MATCH (p:d_{dataset}) RETURN p")
16 ]
17
18 datasets = ['250k', '500k', '750k', '1_million']
19
20 base_directory = r'C:\Users\HP\Downloads\Database Project\Queries_result\Neo4j'
21 os.makedirs(base_directory, exist_ok=True)
22
23 # Function to execute Neo4j queries
24 def execute_query(driver, query, dataset):
25     with driver.session() as session:
26         query = query.replace("{dataset}", dataset)
27         execution_times = []
28         for _ in range(30):
29             start_time = time.time()
30             result = session.run(query)
31             end_time = time.time()
32             execution_times.append(end_time - start_time)
```

# MongoDB

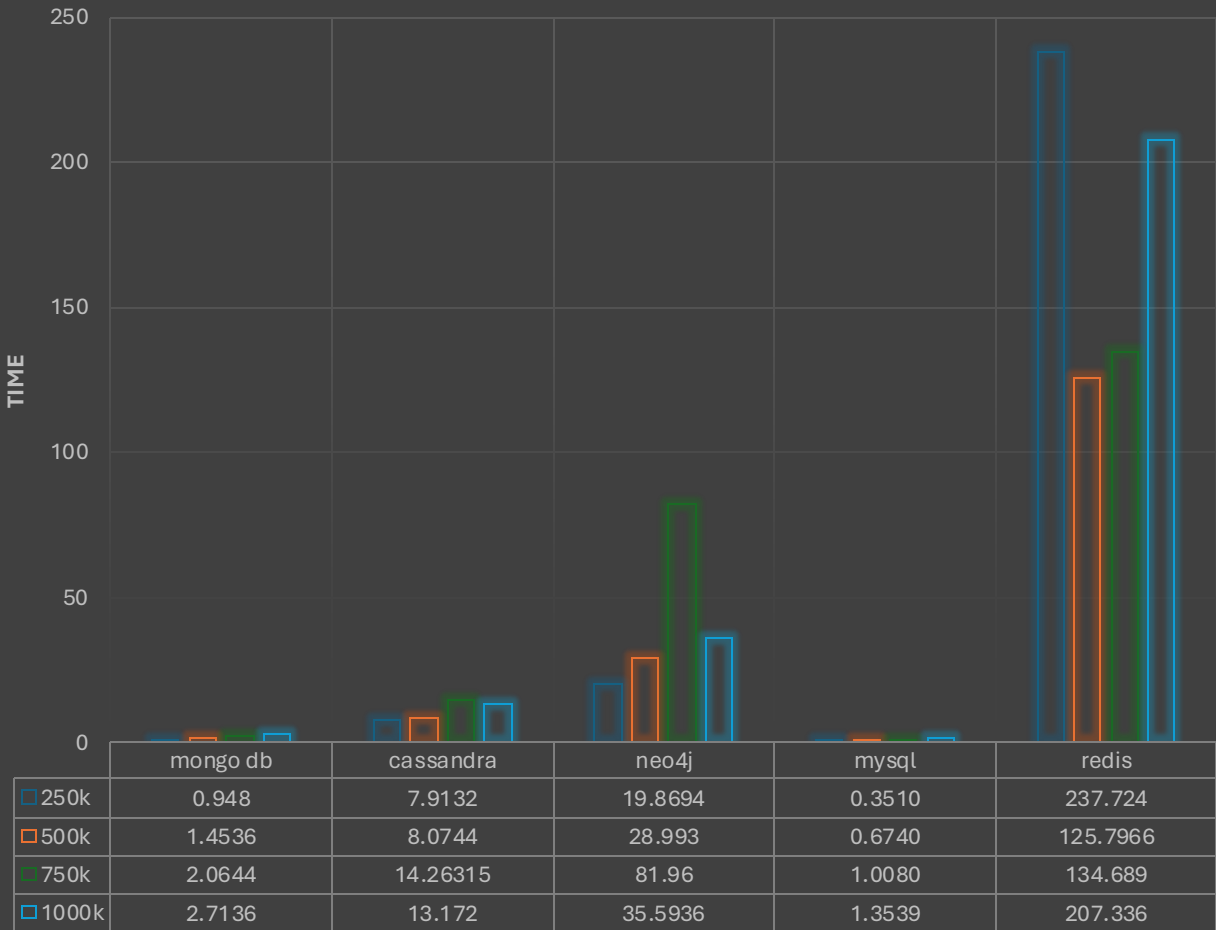
- The provided Python script interacts with a MongoDB database to execute a series of predefined queries across different datasets.
- **Overview of Queries:**
- **Q1:** Retrieves student records with grades greater than or equal to 70 and attendance greater than or equal to 50. This query likely aims to identify high-performing students with consistent attendance.
- **Q2:** Fetches student records with grades greater than or equal to 70. It focuses solely on academic performance, disregarding attendance.
- **Q3:** Searches for student records where the first name starts with the letter 'A'. This query is likely exploring specific patterns or characteristics in student names.
- **Q4:** Retrieves all student records from the dataset.

```
Queries.py  queries.py X
C: > Users > HP > Downloads > Database Project > Mongodb > queries.py > ...
1  import time
2  import os
3  import re
4  import csv
5  from pymongo import MongoClient
6
7  mongo_client = MongoClient("mongodb://localhost:27017/")
8  mongo_db = mongo_client["MyDatabase"] # Connect to the database
9
10 table_names = ['students_250k', 'students_500k', 'students_750k', 'students_1_million']
11
12 queries = [
13     ("Q1", {"Grade": {"$gte": 70}, "Attendance": {"$gte": 50}}),
14     ("Q2", {"Grade": {"$gte": 70}}),
15     ("Q3", {"FirstName": {"$regex": "^A"}}),
16     ("Q4", {})
17 ]
18
19 base_directory = r'C:\Users\HP\Downloads\Database Project\Queries_result\Mongodb'
20
21 os.makedirs(base_directory, exist_ok=True)
22
23 def sanitize_query_description(description):
24     return re.sub(r'^[a-zA-Z0-9_]', '', description)
25
26 for table_name in table_names:
27     collection = mongo_db[table_name]
28     print(f"Table: {table_name} DATASET")
29
30     total_time = 0
31     execution_times = []
32
```

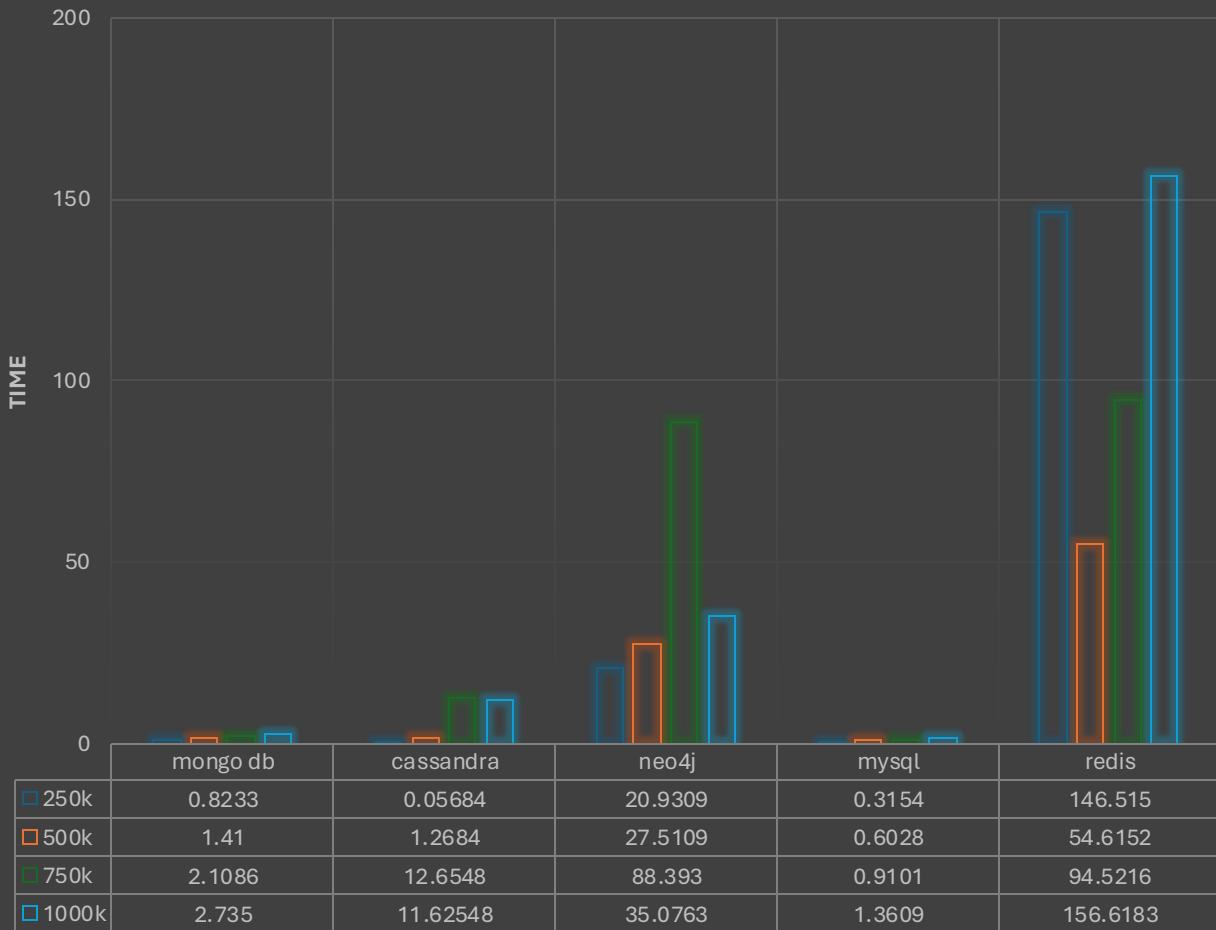
**Query1 :Retrieving Student Information by Grade and Attendance:**

- This query aims to get information about students based on both their grades and attendance.
  - Redis took the longest time to execute this query, while MySQL was the fastest.
- Redis stores data primarily in memory, while MySQL stores data on disk. Retrieving data from memory is typically faster than disk-based operations.
- Additionally, Redis may not have efficient indexing mechanisms or query optimizations for handling complex filtering conditions like grade and attendance.

Query 1: 1st run



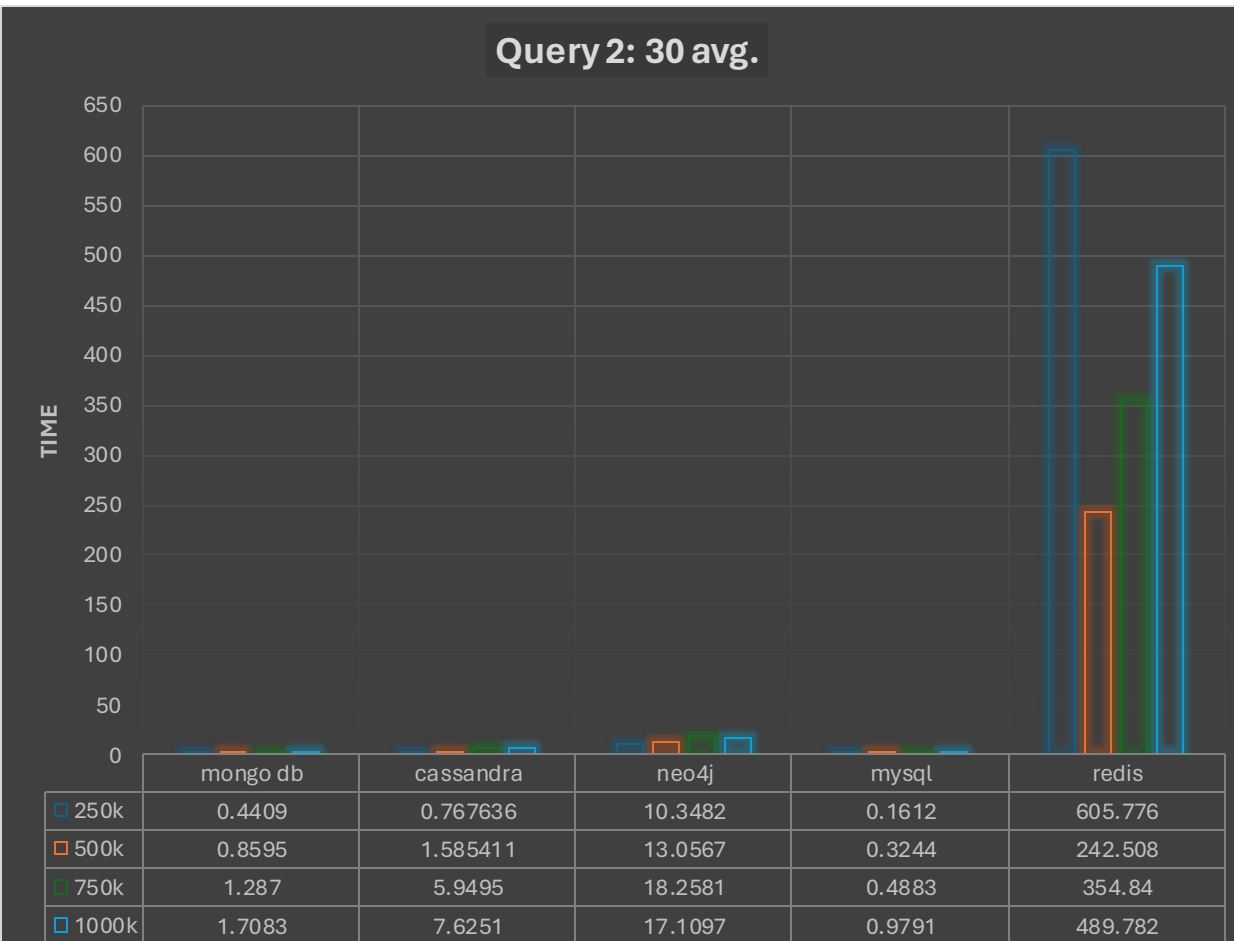
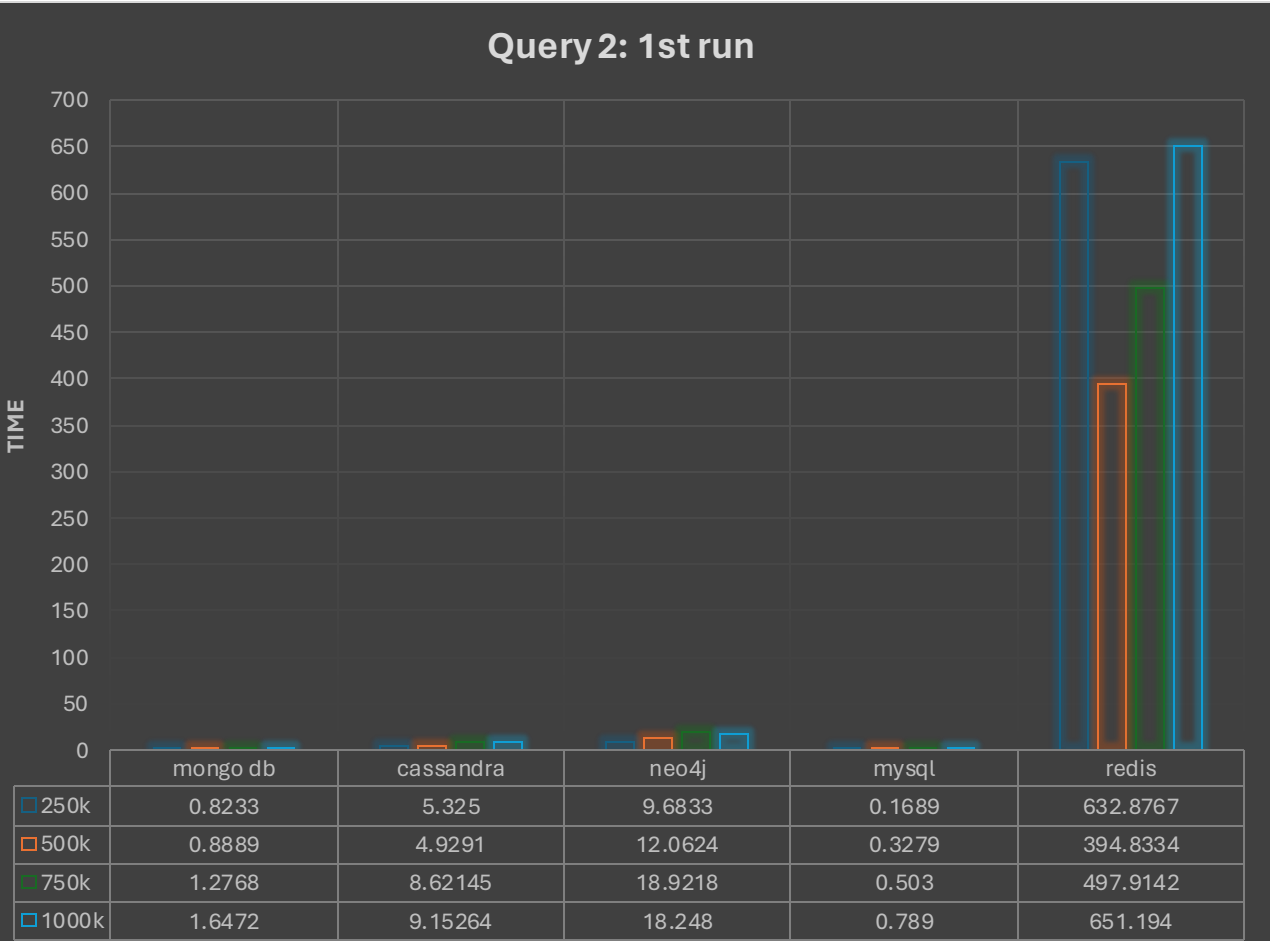
Query 1: 30 avg.



## Query2:Retrieving Student Information by Grade:

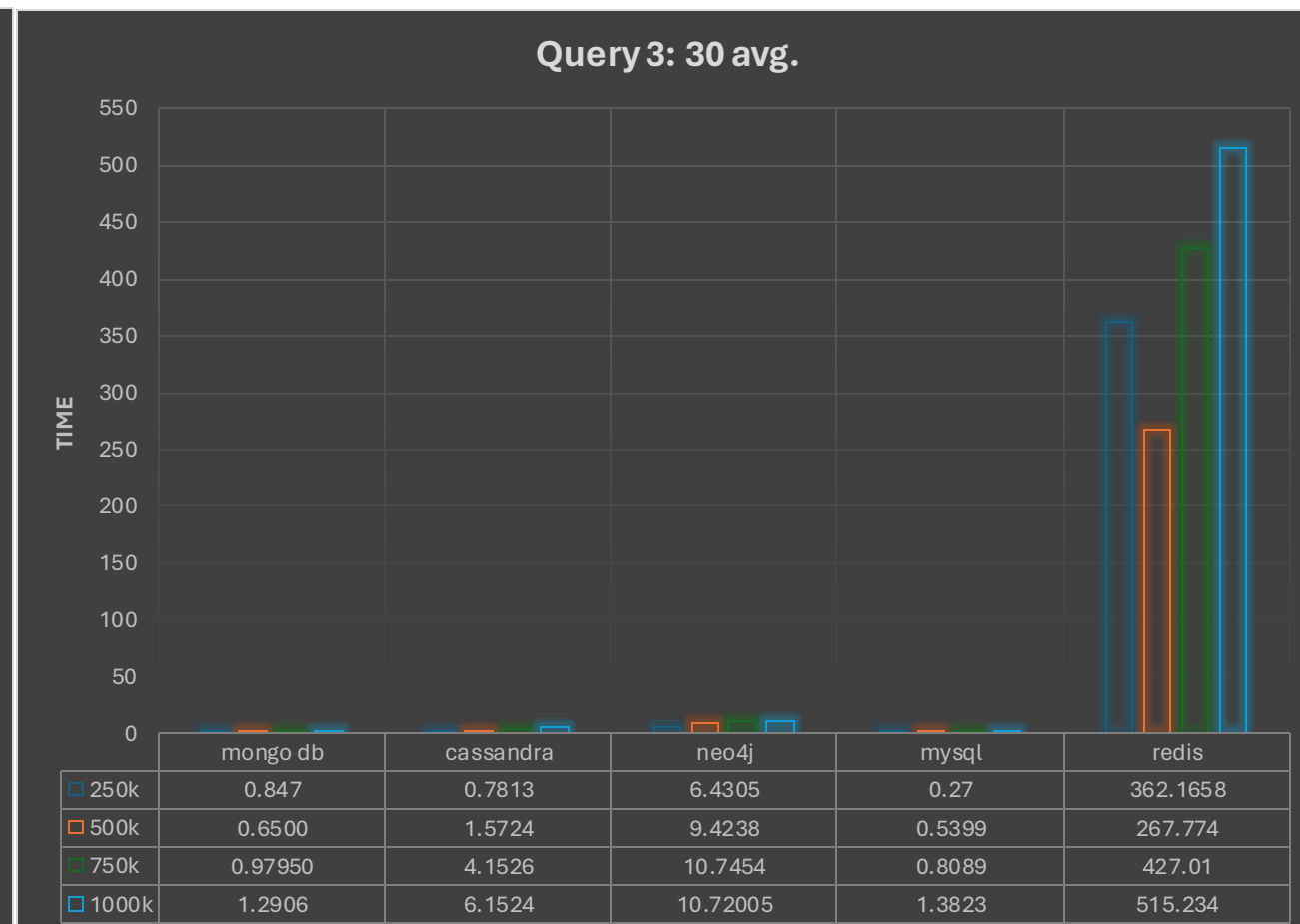
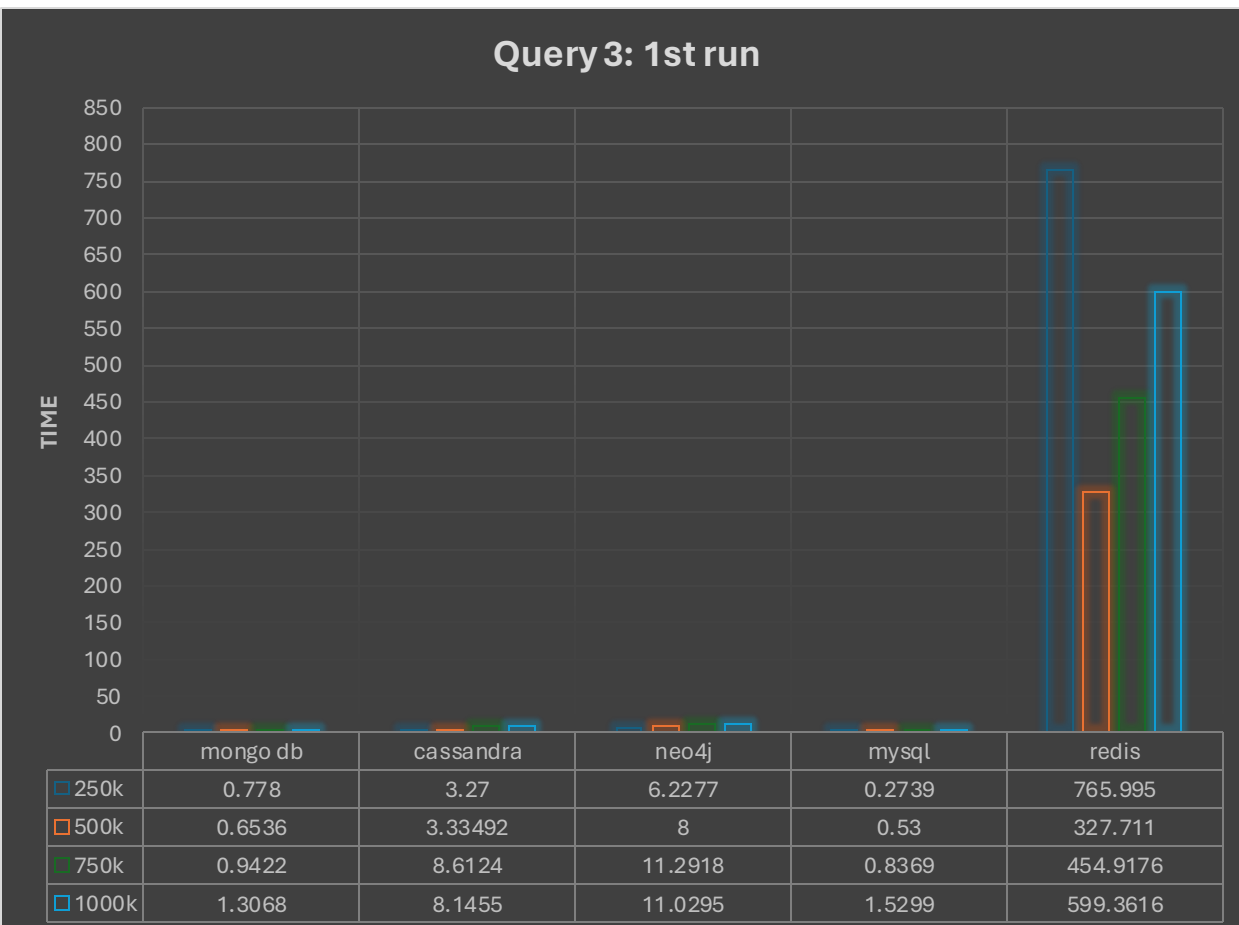
- This query focuses solely on retrieving student records based on their grades.
- Redis exhibited the longest execution time for this query as well.
- Redis's longer execution time can be attributed to its lack of indexing and querying mechanisms optimized for filtering based on grade.

When filtering data, Redis may need to traverse all data structures, resulting in longer execution times compared to database s like Cassandra and Neo4j, which may have specialized indexing or query mechanisms.



### Query3:Retrieving Student Information by First Name:

- Redis and Cassandra consistently demonstrated longer execution times for subsequent queries.
- Redis's longer execution time is due to its in-memory nature, which may require fetching data from disk if not available in memory. Additionally, Redis's lack of indexing or query optimizations for filtering by first name could contribute to slower execution times.
- Cassandra's longer execution times may stem from its distributed architecture and complex storage mechanisms. Querying data in Cassandra involves coordination among multiple nodes, which can introduce latency compared to databases with simpler architectures.



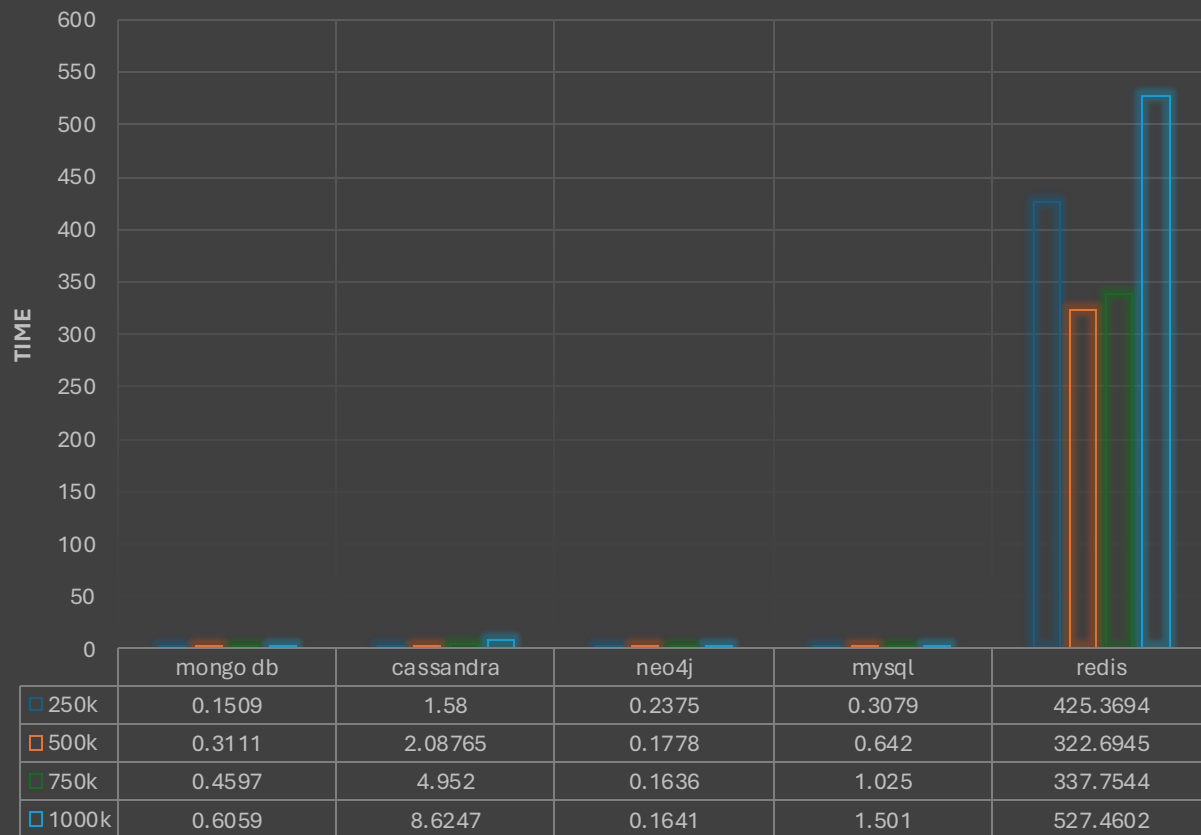
## Query4: Retrieving All Student Information:

- Redis and Cassandra continued to exhibit longer execution times compared to MySQL and Neo4j.
- Redis's longer execution time may be attributed to its in-memory storage, which may require reading data from disk if not present in memory.

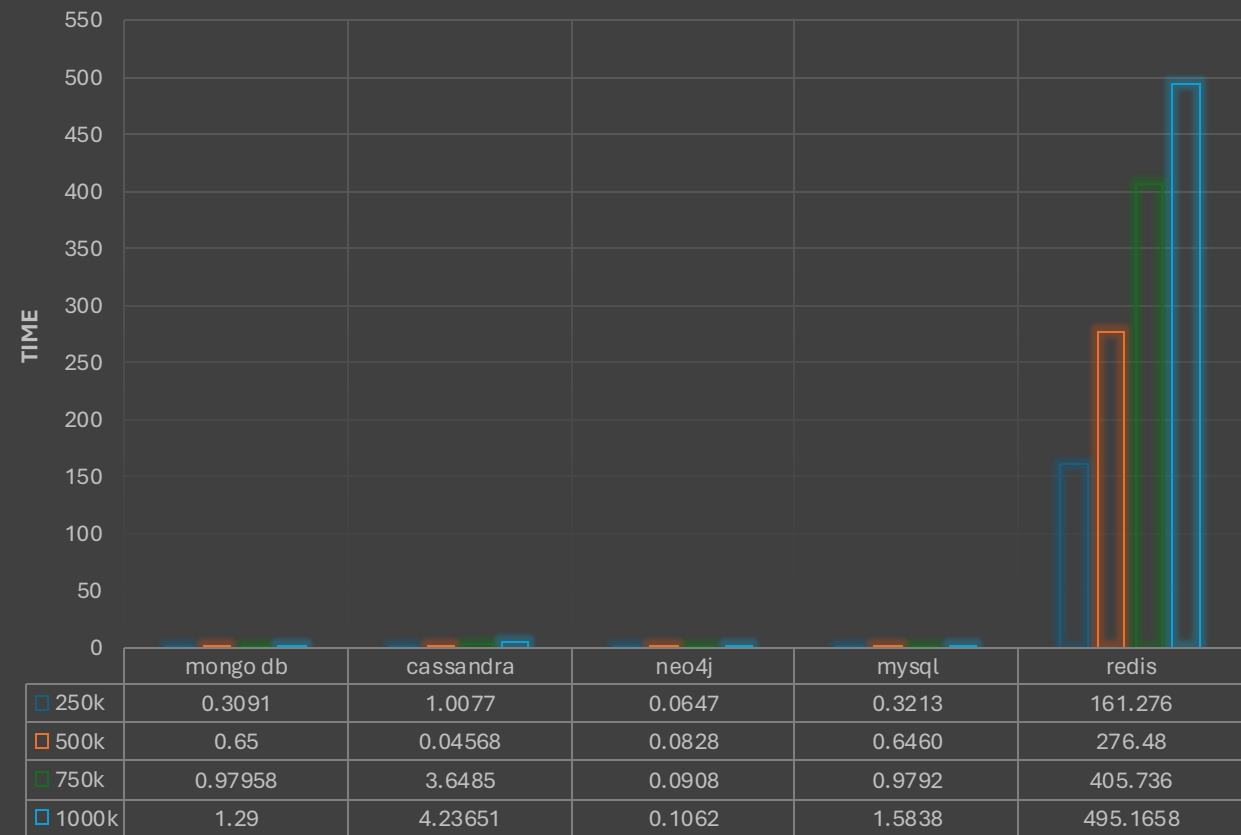
While Redis stores data primarily in memory, it may encounter performance limitations when dealing with large datasets that exceed available memory. When data cannot fit entirely in memory, Redis may need to resort to disk-based operations, which are significantly slower compared to in-memory operations.

- Cassandra's longer execution times can be attributed to its distributed architecture, where data retrieval involves coordination among multiple nodes, potentially resulting in longer response times compared to centralized databases like MySQL and Neo4j.

Query 4: 1st run



Query 4: 30 avg.





# Experiment Results:

- **First Query (Retrieving Student Information by Grade and Attendance):**
  - The experiment aimed to compare the execution times of four different MySQL queries with varying complexities. In the first query, which retrieves student information based on their grade and attendance, Redis exhibited the longest execution time, while MySQL had the shortest. This difference can be attributed to variances in data modeling and indexing approaches between the two databases.
- **Second Query (Retrieving Student Information by Grade):**
  - In the second query, which retrieves student information based solely on their grade, Redis showed the longest execution time. This can be attributed to Redis, being an in-memory data store, requiring traversal of all hashes to filter based on grade. Conversely, databases like Cassandra and Neo4j may leverage optimized indexing or query mechanisms to handle such filters more efficiently.
- **Third Query (Retrieving Student Information by First Name):**
  - Throughout subsequent MySQL queries, a consistent trend emerged where Redis and Cassandra consistently exhibited longer execution times compared to other databases. This trend is attributed to the specific characteristics and optimizations of each database system. Redis may experience longer execution times due to its in-memory nature and potential disk reads for fetching data. Additionally, Cassandra's distributed architecture and complex data storage mechanisms may result in longer query times for certain operations.
- **Fourth Query (Retrieving All Student Information):**
  - In the fourth query, which retrieves all student information, Redis and Cassandra continued to show longer execution times compared to MySQL and Neo4j. These differences are reflective of the unique architectural and indexing approaches of each database system, influencing their performance characteristics in handling such queries.



# Conclusion:

- The comparison of 30 executions affirmed the initial performance trends observed in the first execution of MySQL queries. Subsequent runs consistently demonstrated lower execution times, yet the relative performance rankings among the databases remained consistent. These findings are instrumental in evaluating the stability and relative performance of the databases across multiple query executions.
- During the implementation of the event management system project, I gained valuable insights into the functionalities of various database systems, including MongoDB, MySQL, Neo4j, Cassandra, and Redis. Throughout the development process, we found that MongoDB, MySQL, and Neo4j were relatively easier to work with compared to Cassandra and Redis.
- MongoDB proved to be a flexible and scalable NoSQL database, allowing efficient storage and retrieval of event-related data. Its document-oriented approach and robust querying capabilities made it well-suited for handling unstructured or semi-structured event data.
- MySQL, being a widely used relational database management system, provided us with the familiarity of the relational model and SQL. It offered strong data consistency, transaction support, and ACID properties, making it suitable for structured and relational event data.

- Neo4j, a graph database, excelled in handling highly connected data and complex relationships. It enabled effective modeling of event networks, social interactions, and hierarchical structures using the expressive and efficient Cypher query language.
- However, we encountered challenges when working with Cassandra and Redis. Cassandra's distributed nature and focus on scalability and fault tolerance required additional efforts to ensure smooth data modeling and querying. Redis, being an in-memory data structure store, presented complexities in managing real-time data processing and ensuring data persistence.
- Despite these challenges, exploring these different database systems provided us with a comprehensive understanding of their strengths and limitations in the context of event management. It allowed us to make informed decisions about which databases were most suitable for specific aspects of the project.

# Database Evaluation:

## **MongoDB:**

Strengths: Flexibility in handling evolving data structures, suitable for student information with dynamic attributes.

Efficient read and write performance for diverse use cases in a student information system.

Adaptable to changes in student demographics over time.

## **Redis:**

Strengths: Exceptional performance for quick access to frequently used student data.

Ideal for scenarios where low-latency data retrieval is crucial, such as checking attendance.

## **Neo4j:**

Strengths: Strong performance in handling complex relationship-based queries, beneficial for analyzing student relationships.

Suitable for systems where student interactions and connections play a significant role.

- **Cassandra:**

Strengths: Scalability and high availability, crucial for managing largescale student data.

Good performance in write-heavy distributed environments, ensuring real-time updates to student records.

- **MySQL:**

Strengths: Excellent in managing structured student data, fitting scenarios with a well-defined schema.

Efficient handling of complex SQL queries, advantageous for generating reports and analyzing student information.

- **Neo4j:**
- Worst Cases: Simple, tabular data scenarios with minimal relationships may not benefit from graph-based modeling.

- **Cassandra:**
- Worst Cases: Small-scale deployments where complexity outweighs benefits. Unsuitable for applications with low write throughput or scenarios requiring immediate consistency.

- **Redis:**
- Worst Cases: Scenarios where durability and data persistence are crucial. Vulnerable to data loss in cases of server failures due to primarily in-memory storage.

- **MongoDB:**

Worst Cases: Highly normalized data structures or scenarios requiring complex transactions.

May not be the best fit for applications demanding strict ACID compliance.

- **MySQL:**

Worst Cases: Designed for structured data with predefined schemas.

Inefficient and challenging for highly unstructured or semi-structured student data.

You are welcome to visit my GitHub account,  
accessible via

[https://github.com/0707071/Database\\_comparison](https://github.com/0707071/Database_comparison)

. There, you will find all the code files meticulously  
uploaded examination. Your insights and feedback  
are greatly appreciated.