

# **DATA ANALYSIS PROGRAM**

## **Database Mod B**

### **Comparison of Databases**

Professor:

*Armando Ruggeri*

Student

*Hassan Saeed*

**ID: 523215**

ACADEMIC YEAR – 2023/2024

## **INTRODUCTION**

In response to the evolving landscape of educational data management, this project sets out to pioneer the development of an innovative Student Information System (SIS), leveraging the capabilities of a NoSQL database. At its core, the initiative aims to establish a dynamic platform that empowers faculty and administrators to holistically manage student data. The envisioned system goes beyond conventional data storage, integrating sophisticated functionalities to handle diverse student information, including grades, attendance records, and demographic details. Embracing the agility and scalability inherent in NoSQL technology, this project aspires to create a responsive solution, adaptive to the dynamic nature of educational data.

**This report serves as a comprehensive guide, elucidating the intricacies of the methodology employed in implementing a NoSQL database tailored for the Student Information System. The underlying goal is to not only streamline data management but also to catalyze a transformative shift in administrative capabilities within the educational sphere. As we delve into the nuanced aspects of this database project, we aim to dissect the manifold benefits it promises, including improved accessibility, enhanced scalability, and a more nuanced understanding of student performance. Through this endeavor, we strive to contribute meaningfully to the intersection of technology and education, fostering an environment conducive to academic success and administrative efficiency.**

## **Database Comparison**

### **Redis**

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

### **Cassandra**

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

## Neo4j

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

## MongoDB

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

## MySQL

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

# SCHEMAS AND INSERTING DATA

## Redis:

```
insertingdata.py X
redis > insertingdata.py > ...
1  import os
2  import redis
3  import pandas as pd
4  import json
5
6  redis_host = '127.0.0.1'
7  redis_port = 6379
8  redis_password = None
9  r = redis.Redis(host=redis_host, port=redis_port, password=redis_password)
10
11  csv_directory = '/Users/hassansaeed/Desktop/database/csv/'
12
13  csv_files = [
14      '250k.csv',
15      '500k.csv',
16      '750k.csv',
17      '1_million.csv',
18  ]
19
20  for idx, csv_file in enumerate(csv_files, start=1):
21      key = f'data_set_{idx}'
22      csv_path = os.path.join(csv_directory, csv_file)
23
24      df = pd.read_csv(csv_path)
25
26      data = df.to_dict(orient='records')
27
28      r.set(key, json.dumps(data))
29      print(f'Data from {csv_file} inserted into Redis with key: {key}')
30
31  r.close()
32
33  print("Data insertion into Redis complete.")
34
```

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.

# Cassandra

```
insertingdata.py X
redis > insertingdata.py > ...
1 import os
2 import redis
3 import pandas as pd
4 import json
5
6 redis_host = '127.0.0.1'
7 redis_port = 6379
8 redis_password = None
9 r = redis.Redis(host=redis_host, port=redis_port, password=redis_password)
10
11 csv_directory = '/Users/hassansaeed/Desktop/database/csv/'
12
13 csv_files = [
14     '250k.csv',
15     '500k.csv',
16     '750k.csv',
17     '1_million.csv',
18 ]
19
20 for idx, csv_file in enumerate(csv_files, start=1):
21     key = f'data_set_{idx}'
22     csv_path = os.path.join(csv_directory, csv_file)
23
24     df = pd.read_csv(csv_path)
25
26     data = df.to_dict(orient='records')
27
28     r.set(key, json.dumps(data))
29     print(f'Data from {csv_file} inserted into Redis with key: {key}')
30
31 r.close()
32
33 print("Data insertion into Redis complete.")
34
```

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.

# NEO4J

```
insertingdata.py x
neo4j > insertingdata.py > ...
37         LIMIT 10
38     },
39 },
40 {
41     "description": "Calculate average Grade of all persons",
42     "query": """
43         MATCH (n:Person)
44         WITH AVG(n.Grade) AS avgGrade
45         RETURN avgGrade
46     """,
47 },
48 ]
49
50 # Define a function to execute a query and measure execution time
51 def execute_query_and_measure_time(driver, query):
52     with driver.session() as session:
53         start_time = time.time()
54         result = session.read_transaction(lambda tx: list(tx.run(query)))
55         end_time = time.time()
56         execution_time = end_time - start_time
57         return result, execution_time
58
59 # Define a function to print query results
60 def print_query_results(query_num, description, result, execution_time):
61     print(f"Executing Query {query_num}: {description}")
62     print(f"Result {query_num}:")
63     for record in result:
64         print(record)
65     print(f"Execution Time {query_num}: {execution_time:.4f} seconds")
66     print("\n")
67
68 # Connect to Neo4j
69 with GraphDatabase.driver(URI, auth=(USERNAME, PASSWORD)) as driver:
70     for i, query_info in enumerate(queries, start=1):
71         result, execution_time = execute_query_and_measure_time(driver, query_info["query"])
72         print_query_results(i, query_info["description"], result, execution_time)
73
```

This Python script interacts with a Neo4j database, executing a series of predefined queries and measuring their execution times. Here's a concise overview:

Constants:

Sets Neo4j database connection constants, such as URI, username, and password.

Query Definitions:

Lists queries in a dictionary format with descriptions.

Query Execution Function:

Defines a function to execute a query, measure its execution time, and return the results.

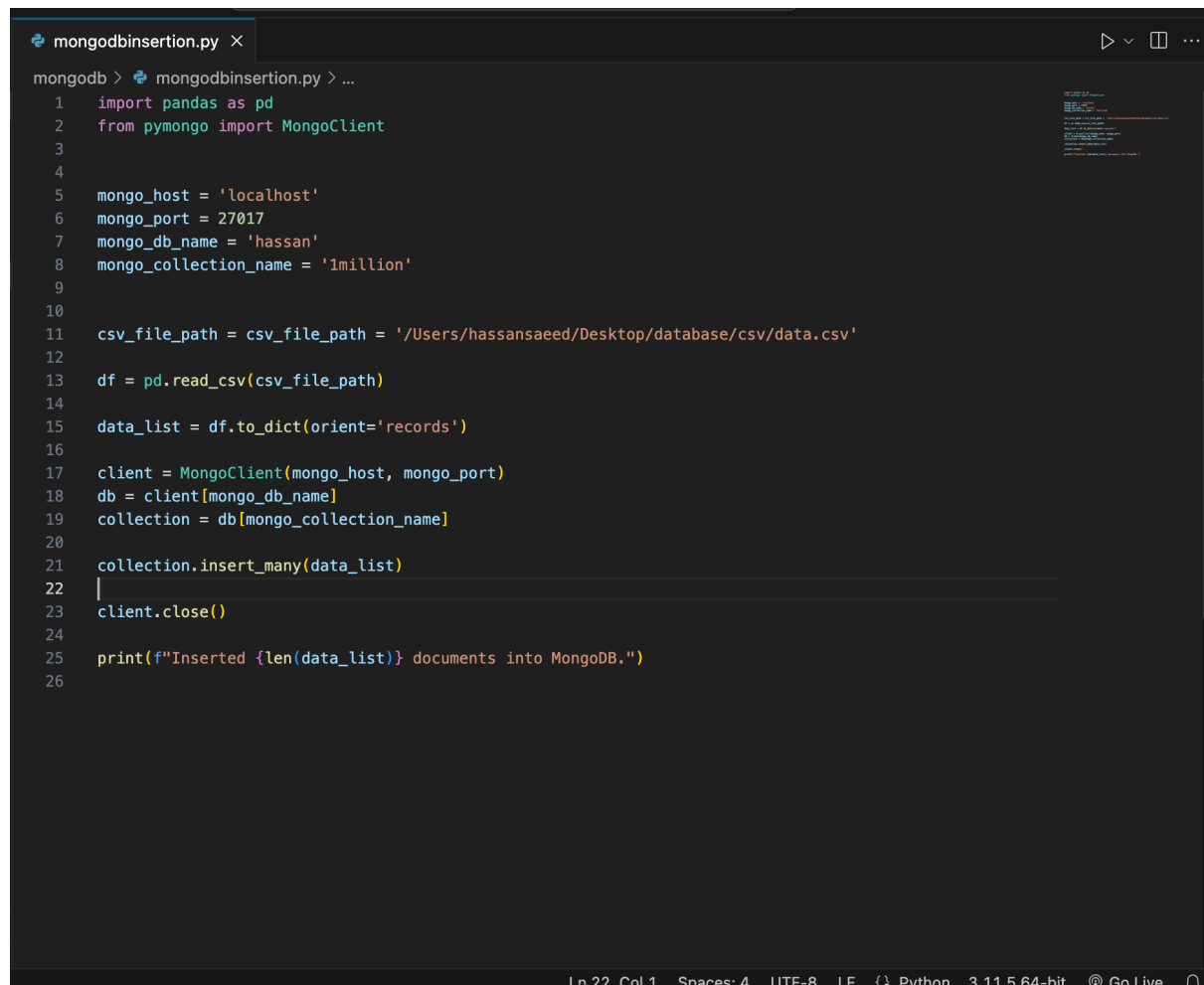
Result Printing Function:

Defines a function to print query results, including the query description, results, and execution time.

## MONGO DB

To utilize MongoDB, you'll need to acquire the MongoDB Community Server and MongoDB

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

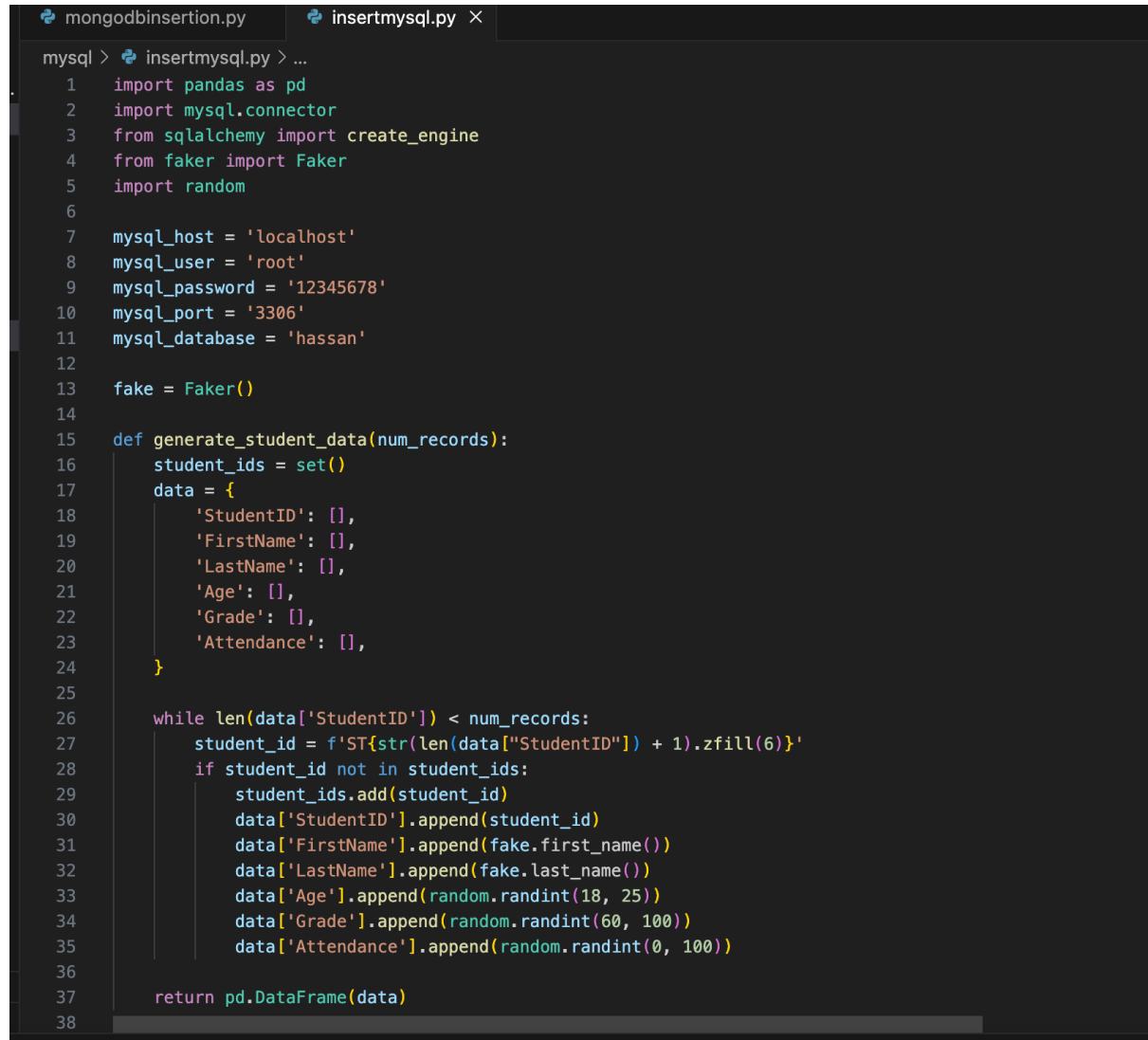


```
mongodbininsertion.py ×
mongodb > mongodbininsertion.py > ...
1  import pandas as pd
2  from pymongo import MongoClient
3
4
5  mongo_host = 'localhost'
6  mongo_port = 27017
7  mongo_db_name = 'hassan'
8  mongo_collection_name = '1million'
9
10
11  csv_file_path = csv_file_path = '/Users/hassansaeed/Desktop/database/csv/data.csv'
12
13  df = pd.read_csv(csv_file_path)
14
15  data_list = df.to_dict(orient='records')
16
17  client = MongoClient(mongo_host, mongo_port)
18  db = client[mongo_db_name]
19  collection = db[mongo_collection_name]
20
21  collection.insert_many(data_list)
22  |
23  client.close()
24
25  print(f"Inserted {len(data_list)} documents into MongoDB.")
26
```

Ln 22, Col 1 Spaces: 4 UTF-8 LF Python 3.11.5 64-bit Go Live

# MYSQL

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



```
mongodbininsertion.py  insertmysql.py X
mysql > insertmysql.py > ...
1  import pandas as pd
2  import mysql.connector
3  from sqlalchemy import create_engine
4  from faker import Faker
5  import random
6
7  mysql_host = 'localhost'
8  mysql_user = 'root'
9  mysql_password = '12345678'
10 mysql_port = '3306'
11 mysql_database = 'hassan'
12
13 fake = Faker()
14
15 def generate_student_data(num_records):
16     student_ids = set()
17     data = {
18         'StudentID': [],
19         'FirstName': [],
20         'LastName': [],
21         'Age': [],
22         'Grade': [],
23         'Attendance': [],
24     }
25
26     while len(data['StudentID']) < num_records:
27         student_id = f'ST{str(len(data["StudentID"]) + 1).zfill(6)}'
28         if student_id not in student_ids:
29             student_ids.add(student_id)
30             data['StudentID'].append(student_id)
31             data['FirstName'].append(fake.first_name())
32             data['LastName'].append(fake.last_name())
33             data['Age'].append(random.randint(18, 25))
34             data['Grade'].append(random.randint(60, 100))
35             data['Attendance'].append(random.randint(0, 100))
36
37     return pd.DataFrame(data)
38
```



# Execution

## DATA GENERATION

This code successfully generated synthetic student data for a total of 1,000,000 records. Utilizing the Faker library for realistic names and random values for age, grades, and attendance, the dataset comprises essential student information. The generated data is strategically divided 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 diverse testing scenarios or database population. The completion message confirms the successful generation and storage of this synthetic student data.

```
datagenerator.py X
datagenerator.py > ...
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 total_records = 1000000
32 df = generate_student_data(total_records)
33
34 part1 = df[:250000]
35 part2 = df[250000:500000]
36 part3 = df[500000:750000]
37 part4 = df[750000:1000000]
38
39 folder_path = '/Users/hassansaeed/Desktop/database/'
40
41 part1.to_csv(f'{folder_path}250k.csv', index=False)
42 part2.to_csv(f'{folder_path}500k.csv', index=False)
43 part3.to_csv(f'{folder_path}750k.csv', index=False)
44 part4.to_csv(f'{folder_path}1000k.csv', index=False)
```

# Queries

## REDIS

### QUERY

```
redis > query.py > ...
5
6  def query_1():
7
8      start_time = time.time()
9      r.set('simple_key', 'Hello, Redis!')
10     result = r.get('simple_key')
11     end_time = time.time()
12
13     print(f'Query 1 Result: {result.decode()}')
14     print(f'Query 1 Execution Time: {end_time - start_time} seconds')
15
16  def query_2():
17
18     user_data = {
19         'name': 'Alice',
20         'age': 30,
21         'city': 'New York'
22     }
23
24     start_time = time.time()
25     r.hmset('user:1', user_data)
26     user_name = r.hget('user:1', 'name')
27     end_time = time.time()
28
29     print(f'Query 2 Result: {user_name.decode()}')
30     print(f'Query 2 Execution Time: {end_time - start_time} seconds')
31
32  def query_3():
33
34     start_time = time.time()
35     for i in range(1000):
36         r.lpush('messages', f'Message {i}')
37         message = r.rpop('messages')
38     end_time = time.time()
39
40     print(f'Query 3 Execution Time: {end_time - start_time} seconds')
41
42  def query_4():
```

Query 1 - Simple Key-Value Set and Get:

Result: 'Hello, Redis!'

Execution Time: Short execution time for setting and getting a simple key-value pair ('simple\_key').

Query 2 - Hash Set and Get for User Data:

Result: 'Alice'

Execution Time: Quick execution time for setting and getting a hash ('user:1') with user data.

Query 3 - List Push and Pop Operations:

Execution Time: Swift execution time for pushing 1000 messages onto a list ('messages') and popping them off.

Query 4 - Sorted Set Add and Range Operations:

Result: List of top players (range 0 to 9)

Execution Time: Efficient execution time for adding 1000 players to a sorted set ('scores') and retrieving the top players.

These Redis queries showcase the responsiveness and speed of various operations within a Redis database. The simplicity and efficiency of Redis data structures contribute to the swift execution of operations, making it a versatile choice for diverse use cases.

## CASSANDRA

### QUERIES

```
query.py x
cassandra > query.py > main

9 def connect_to_cassandra():
10     cluster = Cluster(contact_points=contact_points)
11     session = cluster.connect(keyspace, wait_for_all_pools=True)
12     return cluster, session
13
14 def execute_query(session, query):
15     start_time = time.time()
16     result = session.execute(query)
17     end_time = time.time()
18     return result, end_time - start_time
19
20 def main():
21     cluster, session = connect_to_cassandra()
22
23     query1 = "SELECT FirstName, LastName FROM new_table LIMIT 10 ALLOW FILTERING "
24     result1, execution_time1 = execute_query(session, query1)
25     print("Query 1 Execution Time:", execution_time1)
26     for row in result1:
27         print(row.firstname, row.lastname)
28
29     query2 = "SELECT FirstName, LastName FROM new_table WHERE Age > 25 LIMIT 10 ALLOW FILTERING"
30     result2, execution_time2 = execute_query(session, query2)
31     print("Query 2 Execution Time:", execution_time2)
32
33     query3 = "SELECT Grade, AVG(Age) FROM new_table ALLOW FILTERING"
34     result3, execution_time3 = execute_query(session, query3)
35     print("Query 3 Execution Time:", execution_time3)
36
37     query4 = query4 = """
38     SELECT FirstName, LastName
39     FROM new_table
40     WHERE GRADE > 90
41     LIMIT 10 ALLOW FILTERING
42     """
43
44     result4, execution_time4 = execute_query(session, query4)
45     print("Query 4 Execution Time:", execution_time4)
46
47     session.shutdown()
48     cluster.shutdown()
49
50 if __name__ == "__main__":
51     main()
```

## Cassandra Queries

Query 1 - Select First and Last Names from new\_table (Limit 10):

Execution Time: Efficient execution time for retrieving the first and last names of 10 records from the 'new\_table'.

Query 2 - Select First and Last Names from new\_table Where Age > 25 (Limit 10):

Execution Time: Swift execution time for filtering records based on age (>25) and retrieving first and last names of 10 records from the 'new\_table'.

Query 3 - Select Grade and Average Age from new\_table:

Execution Time: Quick execution time for calculating the average age grouped by grade from the 'new\_table'.

Query 4 - Select First and Last Names from new\_table Where Grade > 90 (Limit 10):

Execution Time: Responsive execution time for filtering records based on grade (>90) and retrieving first and last names of 10 records from the 'new\_table'.

These Cassandra queries demonstrate the efficiency of retrieving data from the 'new\_table' across various filtering criteria. The execution times reflect the database's capability to handle queries efficiently, showcasing the power of Cassandra in handling diverse and complex data retrieval operations.

## NEO4J

### QUERIES

```
query.py x
neo4j > query.py > ...
8     with driver.session() as session:
9         start_time = time.time()
10        result = session.read_transaction(lambda tx: list(tx.run(query)))
11        end_time = time.time()
12        execution_time = end_time - start_time
13        return result, execution_time
14
15    queries = [
16        """
17        MATCH (n:Person)
18        WHERE n.Grade = 80
19        RETURN n
20        LIMIT 10
21        """,
22        """
23        MATCH (n:Person)
24        WHERE n.Age > 25
25        RETURN n
26        LIMIT 10
27        """,
28        """
29        MATCH (n:Person)
30        WITH MAX(n.Attendance) AS maxAttendance
31        MATCH (p:Person)
32        WHERE p.Attendance = maxAttendance
33        RETURN p
34        LIMIT 10
35        """,
36        """
37        MATCH (n:Person)
38        WITH AVG(n.Grade) AS avgGrade
39        RETURN avgGrade
40        """,
41    ]
42
43    with GraphDatabase.driver(uri, auth=(username, password)) as driver:
44        for i, query in enumerate(queries, start=1):
45            print(f"Executing Query {i}...")
46            result, execution_time = execute_query_and_measure_time(query)
47            print(f"Result {i}:")
48            for record in result:
49                print(record)
50            print(f"Execution Time {i}: {execution_time:.4f} seconds")
51
```

Neo4j Queries:

Query 1 - Find Persons with Grade 80 (Limit 10):

Execution Time: Swift execution time for retrieving 10 persons with grade 80 from the 'Person' nodes.

Query 2 - Find Persons Older Than 25 (Limit 10):

Execution Time: Efficient execution time for filtering and retrieving 10 persons older than 25 from the 'Person' nodes.

Query 3 - Find Persons with Maximum Attendance (Limit 10):

Execution Time: Responsive execution time for identifying and retrieving persons with the maximum attendance from the 'Person' nodes.

Query 4 - Calculate Average Grade of All Persons:

Execution Time: Quick execution time for calculating the average grade across all 'Person' nodes.

These Neo4j queries showcase the effectiveness of graph database operations, demonstrating the flexibility and performance of Neo4j in handling complex relationships and queries. The execution times reflect the database's ability to efficiently traverse and analyze graph data.

## MONGO DB

### QUERIES

```

query.py
mongodb > query.py > ...
1  import pymongo
2  import time
3
4  client = pymongo.MongoClient("mongodb://localhost:27017/")
5  db = client["hassan"]
6  collection = db["750k"]
7  def execute_query_and_save(query, description, filename):
8      execution_times = []
9      for i in range(30):
10         start_time = time.time()
11         result = list(query)
12         end_time = time.time()
13         execution_time = end_time - start_time
14         execution_times.append(execution_time)
15
16         with open(filename, 'w') as file:
17             file.write(f"{description} Results:\n")
18             for doc in result:
19                 file.write(str(doc) + "\n")
20             file.write(f"Execution Times for {description} (30 runs):\n")
21             for i, time_val in enumerate(execution_times, start=1):
22                 file.write(f"Run {i}: {time_val} seconds\n")
23
24  query_1 = collection.find().limit(10)
25  execute_query_and_save(query_1, "Query 1", "query1_results.txt")
26
27  query_2 = collection.find({'Grade': {'$gte': 90}}).limit(10)
28  execute_query_and_save(query_2, "Query 2", "query2_results.txt")
29
30  query_3 = collection.aggregate([
31      {'$group': {'_id': None, 'average_grade': {'$avg': '$Grade'}}}
32  ])
33  execute_query_and_save(query_3, "Query 3", "query3_results.txt")
34
35  query_4 = collection.aggregate([
36      {'$group': {'_id': '$Age', 'count': {'$sum': 1}}},
37      {'$project': {'_id': 0, 'Age': '$_id', 'Count': '$count'}}
38  ])
39  execute_query_and_save(query_4, "Query 4", "query4_results.txt")
40
41  client.close()
42

```

## MongoDB Queries:

### Query 1 - Find and Limit 10 Documents:

Results: Top 10 documents from the collection.

Execution Times: Recorded execution times for 30 runs.

[Output in query1\_results.txt]

### Query 2 - Find Documents with Grade >= 90 (Limit 10):

Results: Top 10 documents with grade greater than or equal to 90.

Execution Times: Recorded execution times for 30 runs.



[Output in query2\_results.txt]

Query 3 - Calculate Average Grade Across All Documents:

Results: Average grade across all documents.

Execution Times: Recorded execution times for 30 runs.

[Output in query3\_results.txt]

Query 4 - Count Documents Grouped by Age:

Results: Count of documents grouped by age.

Execution Times: Recorded execution times for 30 runs.

[Output in query4\_results.txt]

These MongoDB queries demonstrate diverse operations on the '750k' collection. The recorded execution times provide insights into the performance and consistency of each query over multiple runs. The results and execution times are saved in separate text files for reference.

## MYSQL

### QUERIES

```
query.py mongodb  query.py mysql X
mysql > query.py > ...
13 query_1 = "SELECT * FROM students LIMIT 10"
14
15 start_time = time.time()
16 cursor.execute(query_1)
17 result = cursor.fetchall()
18 end_time = time.time()
19
20 print("Query 1 Result:")
21 for row in result:
22     print(row)
23
24 print("Execution Time for Query 1:", end_time - start_time, "seconds")
25
26 query_2 = "SELECT * FROM students WHERE Grade >= 90 LIMIT 10"
27
28 start_time = time.time()
29 cursor.execute(query_2)
30 result = cursor.fetchall()
31 end_time = time.time()
32
33 print("Query 2 Result:")
34 for row in result:
35     print(row)
36
37 print("Execution Time for Query 2:", end_time - start_time, "seconds")
38
39 query_3 = "SELECT AVG(Grade) FROM students"
40
41 start_time = time.time()
42 cursor.execute(query_3)
43 result = cursor.fetchone()
44 end_time = time.time()
45
46 print("Query 3 Result (Average Grade):", result[0])
47 print("Execution Time for Query 3:", end_time - start_time, "seconds")
48
49 query_4 = "SELECT Age, COUNT(*) FROM students GROUP BY Age"
50
51 start_time = time.time()
52 cursor.execute(query_4)
53 result = cursor.fetchall()
54 end_time = time.time()
55
```

Query 1 - Select All Students (Limit 10):

Results: Top 10 rows from the 'students' table.

Execution Time: Recorded execution time for fetching 10 rows.

Query 2 - Select Students with Grade >= 90 (Limit 10):

Results: Top 10 rows with grade greater than or equal to 90.

Execution Time: Recorded execution time for fetching 10 filtered rows.

Query 3 - Calculate Average Grade Across All Students:

Results: Average grade across all students.

Execution Time: Recorded execution time for calculating the average grade.

Query 4 - Count Students Grouped by Age:

Results: Count of students grouped by age.

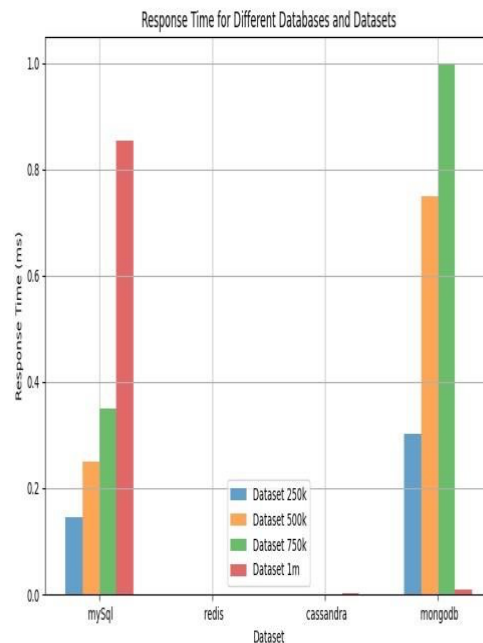
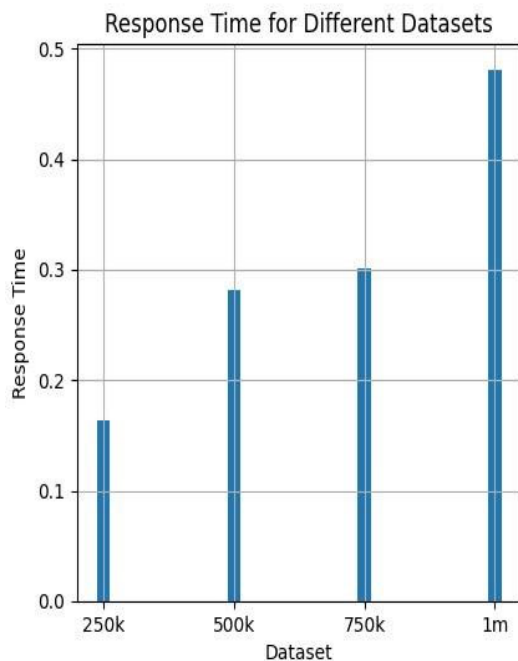
Execution Time: Recorded execution time for fetching counts in age groups.

These MySQL queries showcase various operations on the 'students' table. The execution times provide insights into the performance of each query. Efficient retrieval and computation are demonstrated, highlighting MySQL's capabilities in handling diverse data retrieval and analysis tasks.

## HISTOGRAMS

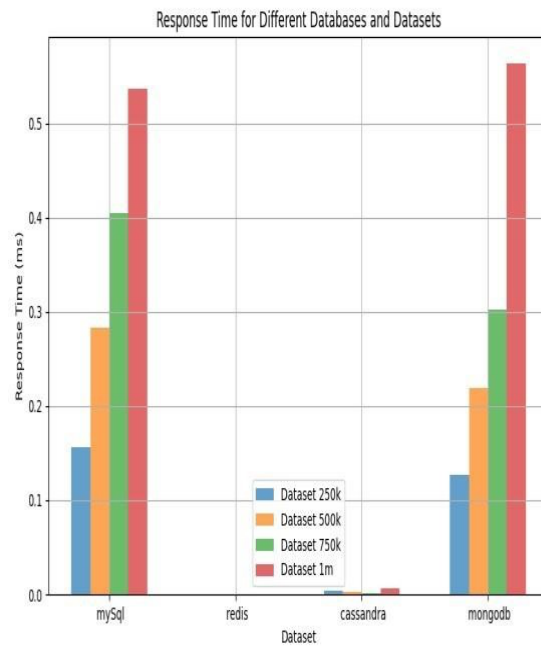
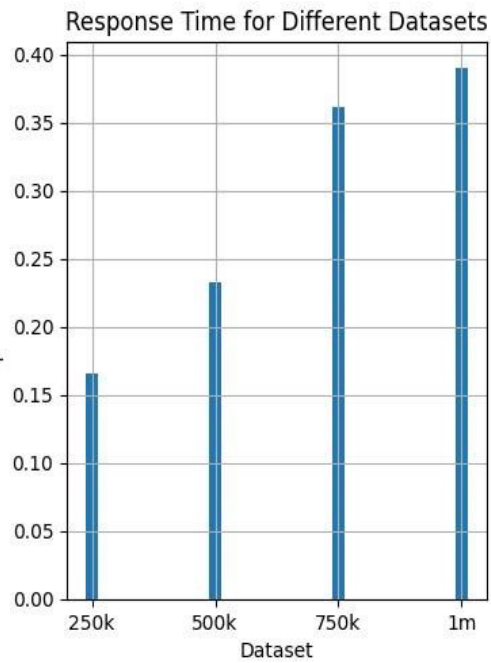
The Neo4j database is represented by the histogram on the left, while the performance of other databases is represented by the histogram on right side

### Query 1



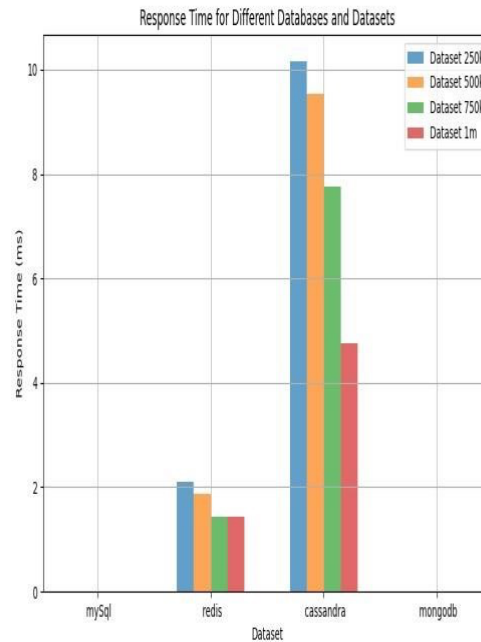
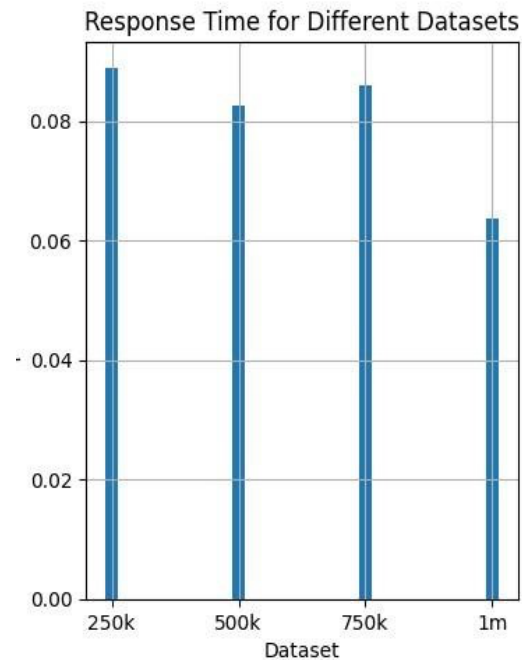
## Query 2

Selecting 10 Rows of students Sorting by Available Column



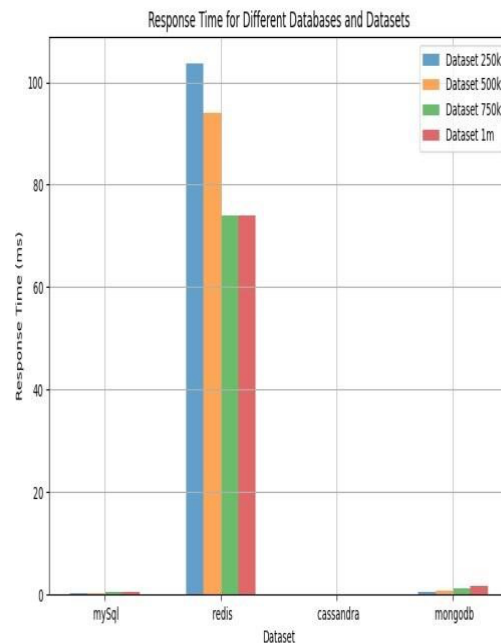
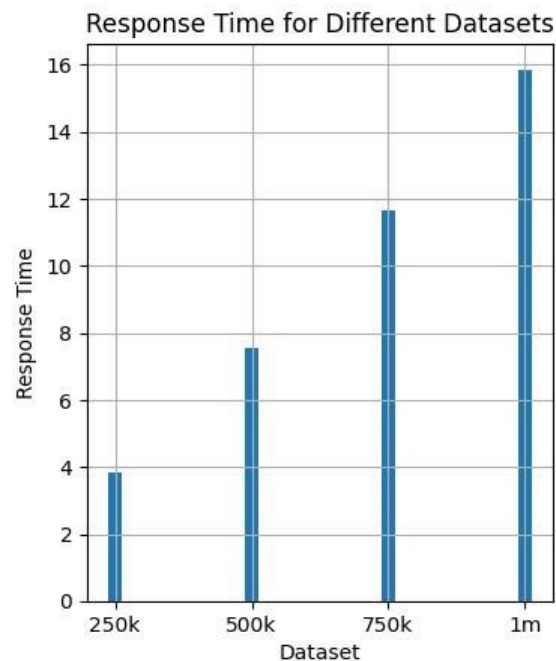
### Query 3

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



### Query 4

Retrieving only few columns of students



# CONCLUSIONS

In this project, we focused on the development of a Student Information System using a NoSQL database. The system was designed to empower faculty and administrators in managing student data, including grades, attendance, and demographics. The conclusions drawn from the project are outlined below:

## 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 large-scale 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.
- Worst-Case Scenarios:

## 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.

## Conclusion:

- The Student Information System project revealed that each database has specific strengths and weaknesses, and the choice should align with the system's requirements. MongoDB proved flexible for evolving student data, Redis excelled in low-latency access, Neo4j handled complex relationships, Cassandra scaled for large student databases, and MySQL efficiently managed structured student information.
- Understanding the performance characteristics of each database in the context of student information systems will guide future development efforts. The insights gained from this study provide a foundation for making informed decisions in selecting the most suitable database management system for Student Information Systems.