Professor : Armando Ruggeri

Made by:

Arya Khosravirad

Università
degli Studi di
Messina

comparison of databases

student information system

TABLE OF CONTENTS

INTRODUCTION

Student Information Systems (SIS) are vital for edu

cational institutions but face challenges like outdated data management and limited accessibility.

This report compares diverse database systems for SIS, including Redis, Cassandra, Neo4j, MongoDB, and MySQL. We assess performance, scalability, flexibility, and query capabilities within the SIS context.

We use a representative dataset with information about students, courses, grades, and academic records, with record sizes of 250,000, 500,000, 750,000, and 1 million. Multiple complex queries are executed and automated for reliable data, each test run at least 30 times.

Our findings are presented through histograms, showing response times in milliseconds. The goal is to identify the database system demonstrating superior query execution under consistent hardware and software conditions.

EXPLANATION OF DATABASES

MySQL is an open-source relational database management system (RDBMS) that is widely used for storing and managing structured data. It is a popular choice for web applications and is often used in conjunction with scripting languages like PHP, Python, and Ruby.

MongoDB is a popular open-source NoSQL (non-relational) database management system. It is designed to handle large volumes of unstructured and semi-structured data, making it particularly suitable for modern applications that require flexible and scalable data storage.

Redis is a high-speed, open-source, in-memory data store known for its lightning-fast data access and low-latency performance. It serves as a versatile solution for various applications, including caching, real-time analytics, and messaging. Redis's support for advanced data structures and ease of use make it a valuable asset for enhancing the efficiency of modern software systems.

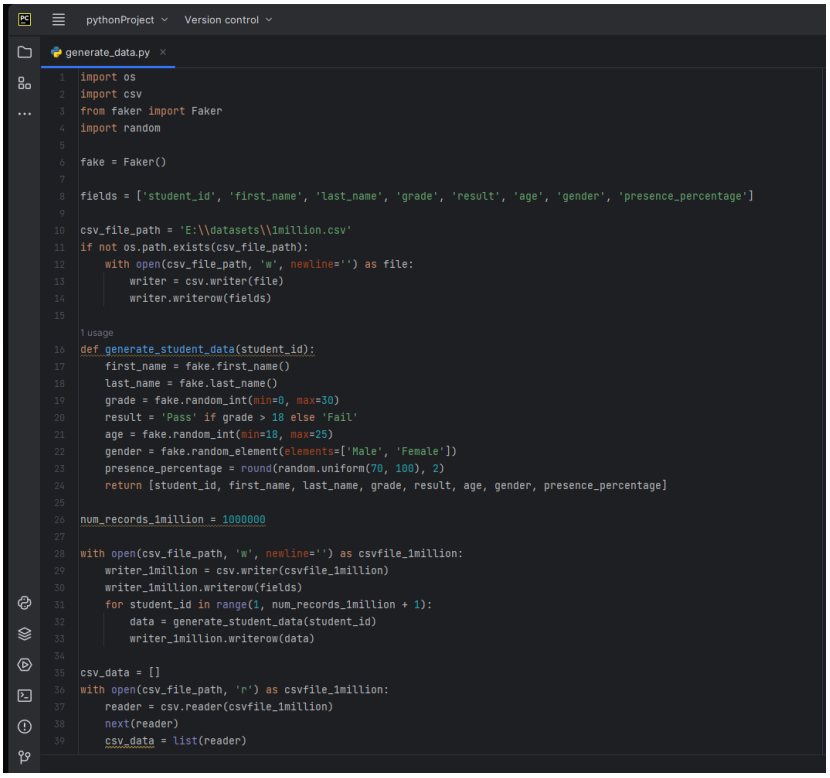
Cassandra is a highly scalable, distributed NoSQL database system known for its exceptional fault tolerance and data availability. This open-source solution is tailored for handling large datasets and distributed data management. Developed by Facebook and open-sourced, Cassandra is favored in applications demanding real-time big data analytics, time-series data, and content management systems. It offers a resilient and versatile solution for modern data storage and management.

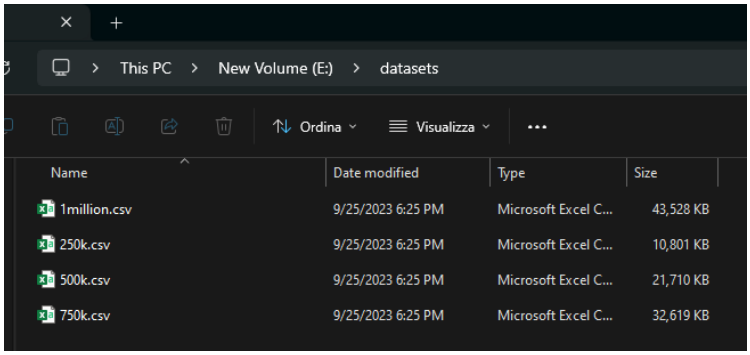
Neo4j is a distinguished graph database management system, purpose-built for handling data as nodes and relationships. It excels in managing complex, interconnected data, making it an ideal choice for applications involving intricate relationships, such as social networks and recommendation systems. Neo4j's graph-based model allows for efficient and insightful data analysis, setting it apart as a leading solution for industries that require deep exploration of connections within their datasets.

GENERATING DATA

This script generates synthetic student data for our Student Information System (SIS) and organizes it into separate CSV files. It utilizes the 'Faker' library to create fictional student information, including 'student_id,' 'first_name,' 'last_name,' 'grade,' 'result,' 'age,' 'gender,' and 'presence_percentage.' The script verifies the existence of the primary CSV file, '1million.csv,' and creates it if absent, including a header row to label the fields.

The core functionality involves generating data for one million students, capturing details such as grades, results, age, gender, and presence percentage using a dedicated function. This data is then divided into three distinct CSV files—'750k.csv,' '500k.csv,' and '250k.csv,' each tailored to house a specific number of records. This script streamlines the data preparation process for our SIS, enabling testing and development with substantial and realistic datasets. Upon completion, the script confirms the successful generation and distribution of synthetic data.





as you see the fake data is generated and saved as 4 csv files in E:\\datasets\\1million.csv

INSERTING DATA

MySQL:

A local MySQL server connection is established with host, user, password, port, and 'students_db' database parameters.

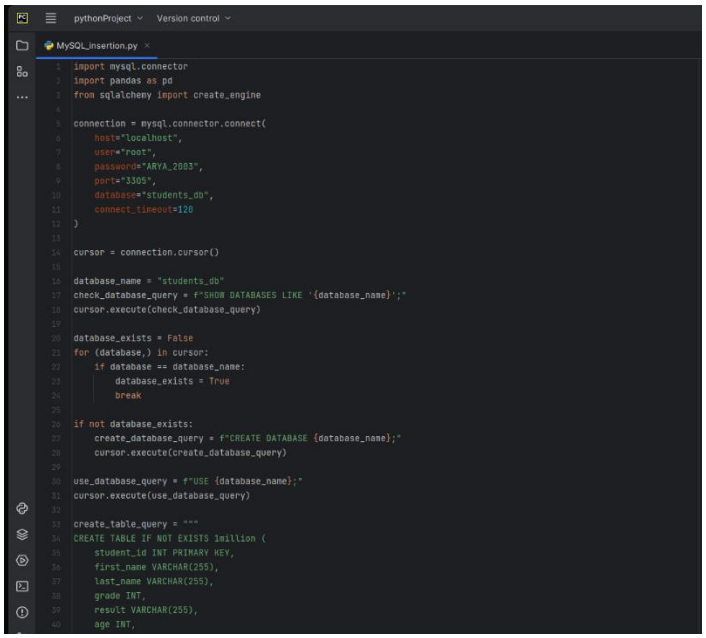
If the database doesn't exist, it's created. Inside 'students_db,' four tables ('1million,' '750k,' '500k,' and '250k') are made for CSV data.

The script's core function is importing data from CSV files into MySQL tables using Pandas and SQLAlchemy for insertion, with error handling for data integrity.

Successful insertions are committed, and the script closes the cursor and connection for efficiency.

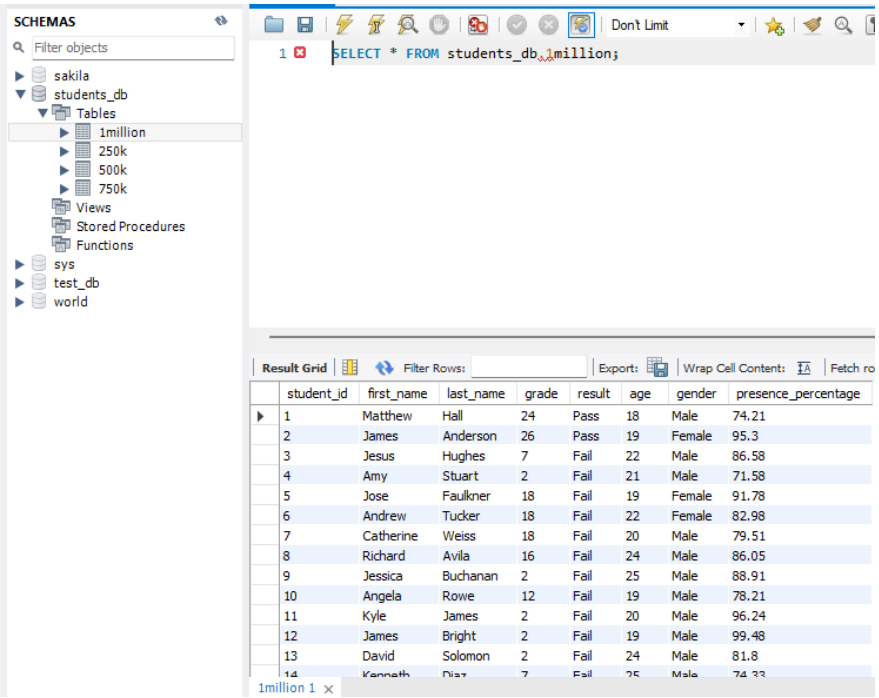
This script is vital for managing diverse data in our Student Information System.

the engine is a central component of SQLAlchemy that provides a standardized and efficient way to connect to databases, execute SQL queries, and manage database connections.



```
pythonProject  Version control
MySQL_insertion.py
```

```python
import mysql.connector
import pandas as pd
from sqlalchemy import create_engine

connection = mysql.connector.connect(
    host="localhost",
    user="root",
    password="ARYA_2003",
    port="3305",
    database="students_db",
    connect_timeout=120
)

cursor = connection.cursor()

database_name = "students_db"
check_database_query = f"SHOW DATABASES LIKE '{database_name}';"
cursor.execute(check_database_query)

database_exists = False
for (database,) in cursor:
    if database == database_name:
        database_exists = True
        break

if not database_exists:
    create_database_query = f"CREATE DATABASE {database_name};"
    cursor.execute(create_database_query)

use_database_query = f"USE {database_name};"
cursor.execute(use_database_query)

create_table_query = """
CREATE TABLE IF NOT EXISTS 1million (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    grade INT,
    result VARCHAR(255),
    age INT,
    gender VARCHAR(255),
    presence_percentage DECIMAL(5, 2)
);
"""
cursor.execute(create_table_query)

create_table_query = """
CREATE TABLE IF NOT EXISTS 500k (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    grade INT,
    result VARCHAR(255),
    age INT,
    gender VARCHAR(255),
    presence_percentage DECIMAL(5, 2)
);
"""
cursor.execute(create_table_query)

create_table_query = """
CREATE TABLE IF NOT EXISTS 750k (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    grade INT,
    result VARCHAR(255),
    age INT,
    gender VARCHAR(255),
    presence_percentage DECIMAL(5, 2)
);
"""
cursor.execute(create_table_query)

create_table_query = """
CREATE TABLE IF NOT EXISTS 250k (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    grade INT,
    result VARCHAR(255),
    age INT,
    gender VARCHAR(255),
    presence_percentage DECIMAL(5, 2)
);
"""
cursor.execute(create_table_query)


csv_files = [
    'E:\\datasets\\1million.csv',
    'E:\\datasets\\750k.csv',
    'E:\\datasets\\500k.csv',
    'E:\\datasets\\250k.csv'
]

engine = create_engine('mysql+mysqlconnector://root:ARYA_2003@localhost:3305/students_db')

for i, csv_file_path in enumerate(csv_files):
    table_name = ['1million', '750k', '500k', '250k'][i]

    df = pd.read_csv(csv_file_path)

    try:
        df.to_sql(table_name, engine, if_exists='replace', index=False)
        print(f"Data inserted successfully into table: {table_name}")
    except Exception as e:
        print(f"Error inserting data into table: {table_name}. Error: {str(e)}")

connection.commit()

cursor.close()
connection.close()
```



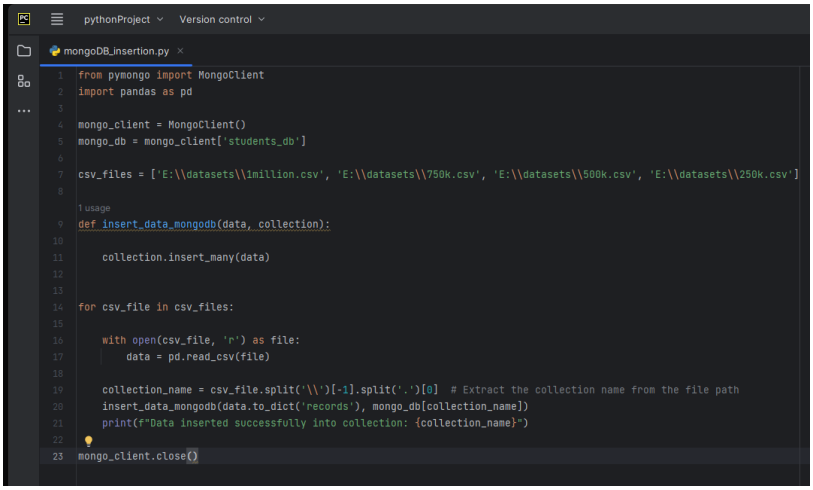
Now you can see the output that 4 tables are created and the data is inserted into each table

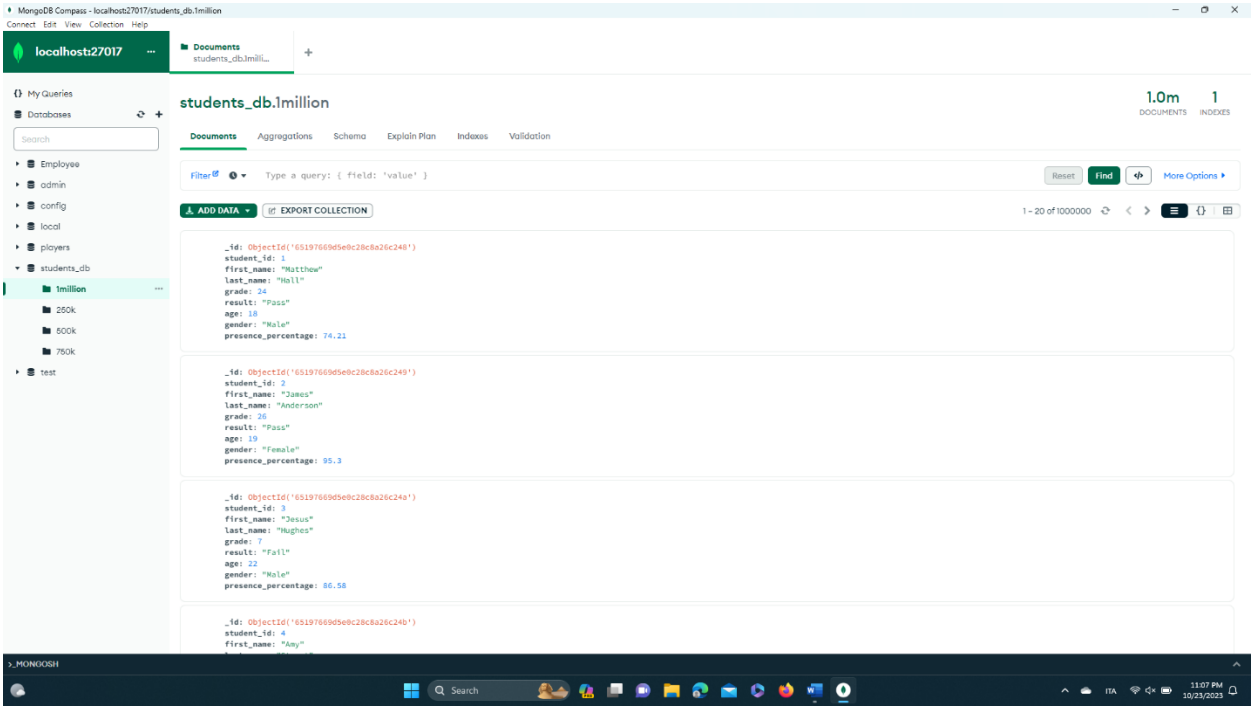
mongoDB:

This script facilitates the insertion of data from CSV files into a MongoDB database. Using the 'pymongo' library, it establishes a connection to a MongoDB server and creates or accesses the 'students_db' database. The script then defines a list of CSV file paths, each representing a different dataset.

The core function 'insert_data_mongodb' is responsible for inserting data from each CSV file into the MongoDB database. The script iterates through the list of CSV files, reads the data using Pandas, extracts the collection name from the file path, and inserts the data into the corresponding MongoDB collection.

Upon successful data insertion, the script efficiently closes the MongoDB connection. This script streamlines the process of populating a MongoDB database with data from various CSV sources, making it a valuable tool for data management in your project.





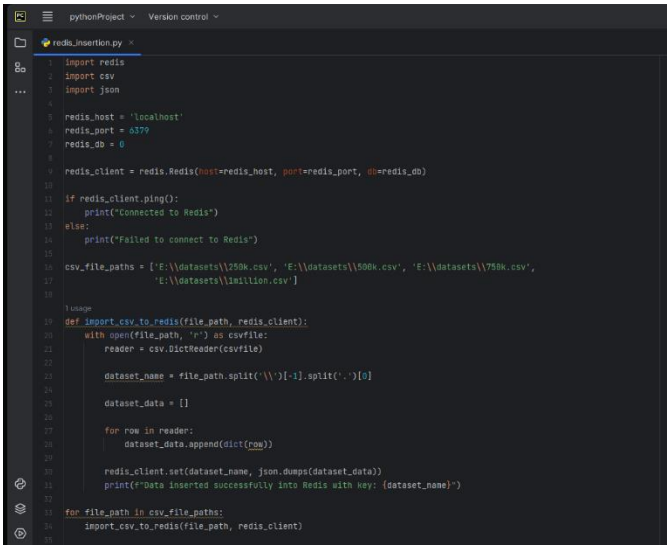
Now we have for collections are created with the corresponding data.

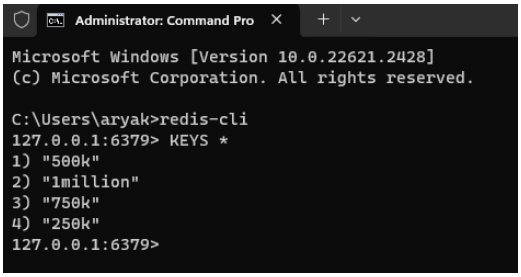
Redis:

This script streamlines the process of inserting data from CSV files into a Redis database. It establishes a connection with Redis using the provided connection details and checks for a successful connection. The CSV file paths are specified, representing distinct datasets.

The core function 'import_csv_to_redis' reads each CSV file, extracts the dataset name from the file path, and stores the data as a JSON string in Redis with a specific key. This approach enables the script to insert data into Redis with clear distinction. After successful insertion, it provides a confirmation message.

By looping through the CSV files, the script imports each dataset into Redis with separate keys. Subsequently, it allows data retrieval from Redis based on the designated keys. This script simplifies the process of integrating data from various sources into Redis, offering an efficient and organized method for data storage and retrieval in your project.





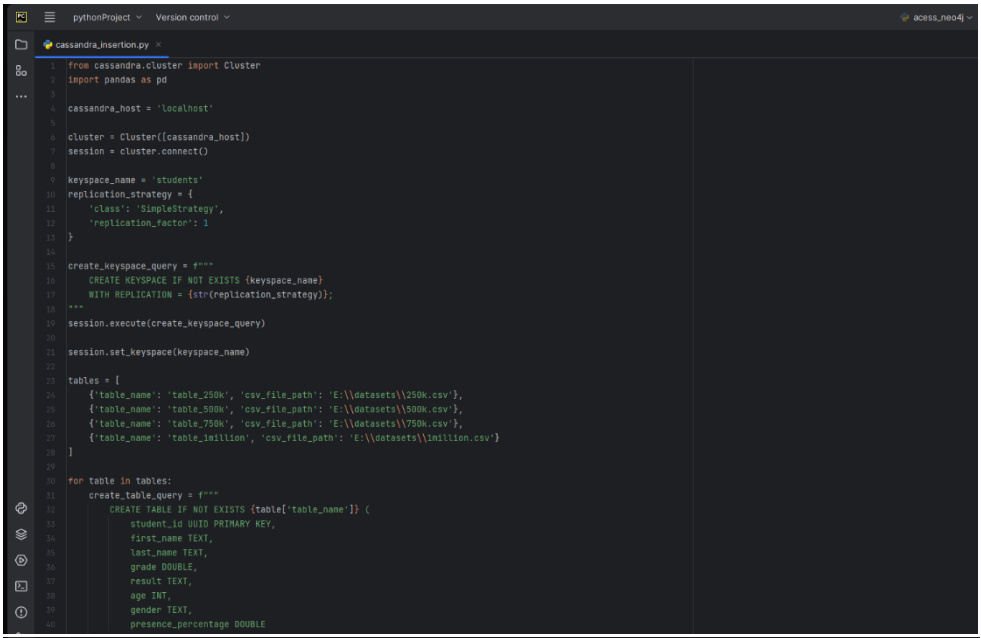
After the code execution you can run redis client in command prompt to see the created keys

CASSANDRA:

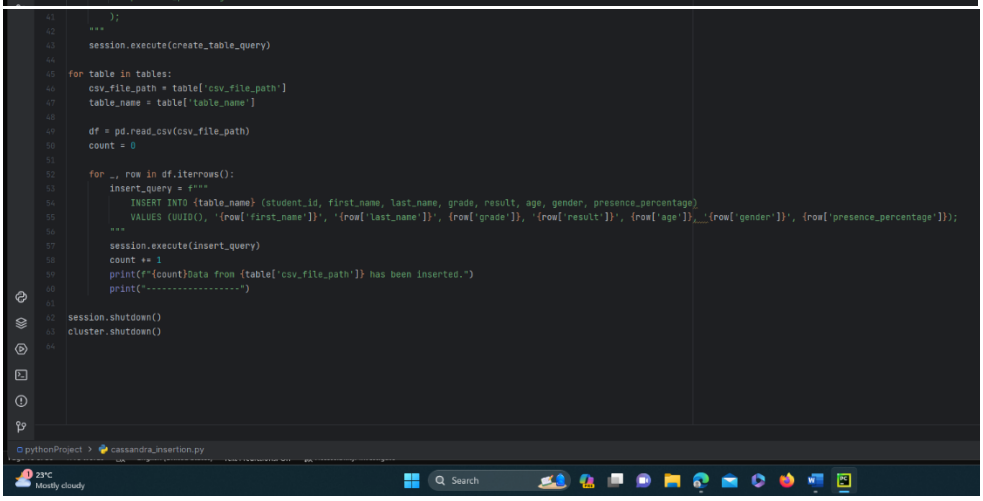
This script simplifies the process of inserting data into a Cassandra database, leveraging Docker for containerization. It connects to Cassandra, defines a keyspace, and creates tables to accommodate data from various CSV files. The CSV data is then loaded using Pandas, and the script inserts the data into the corresponding Cassandra tables.

To ensure data consistency and integrity, each table's schema is predefined, and data is inserted while adhering to the defined structure. The script neatly handles the creation of keyspace, tables, and data insertion for different datasets. Upon successful insertion, it provides informative progress messages, streamlining the data integration process in your project. It's a valuable tool for efficiently managing data in a Cassandra database, especially when dealing with multiple datasets.

Cassandra uses a replication strategy to determine how data is distributed across multiple nodes in a cluster. In this code, a simple replication strategy (SimpleStrategy) is used. This strategy is suitable for single-node or small-scale deployments. The replication_factor is set to 1, which means that each piece of data is replicated to one node in the cluster.



```python
from cassandra.cluster import Cluster
import pandas as pd

cassandra_host = 'localhost'

cluster = Cluster([cassandra_host])
session = cluster.connect()

keyspace_name = 'students'
replication_strategy = {
    'class': 'SimpleStrategy',
    'replication_factor': 1
}

create_keyspace_query = f"""
    CREATE KEYSPACE IF NOT EXISTS {keyspace_name}
    WITH REPLICATION = {str(replication_strategy)};
"""
session.execute(create_keyspace_query)

session.set_keyspace(keyspace_name)

tables = [
    {'table_name': 'table_250k', 'csv_file_path': 'E:\\datasets\\250k.csv'},
    {'table_name': 'table_500k', 'csv_file_path': 'E:\\datasets\\500k.csv'},
    {'table_name': 'table_750k', 'csv_file_path': 'E:\\datasets\\750k.csv'},
    {'table_name': 'table_1million', 'csv_file_path': 'E:\\datasets\\1million.csv'}
]

for table in tables:
    create_table_query = f"""
        CREATE TABLE IF NOT EXISTS {table['table_name']} (
            student_id UUID PRIMARY KEY,
            first_name TEXT,
            last_name TEXT,
            grade DOUBLE,
            result TEXT,
            age INT,
            gender TEXT,
            presence_percentage DOUBLE
        );
    """
    session.execute(create_table_query)

for table in tables:
    csv_file_path = table['csv_file_path']
    table_name = table['table_name']

    df = pd.read_csv(csv_file_path)
    count = 0

    for _, row in df.iterrows():
        insert_query = f"""
            INSERT INTO {table_name} (student_id, first_name, last_name, grade, result, age, gender, presence_percentage)
            VALUES (UUID(), '{row['first_name']}', '{row['last_name']}', {row['grade']}, '{row['result']}', {row['age']}, '{row['gender']}', {row['presence_percentage']});
        """
        session.execute(insert_query)
        count += 1
        print(f"{count}Data from {table['csv_file_path']} has been inserted.")
        print("------------------")

session.shutdown()
cluster.shutdown()
```



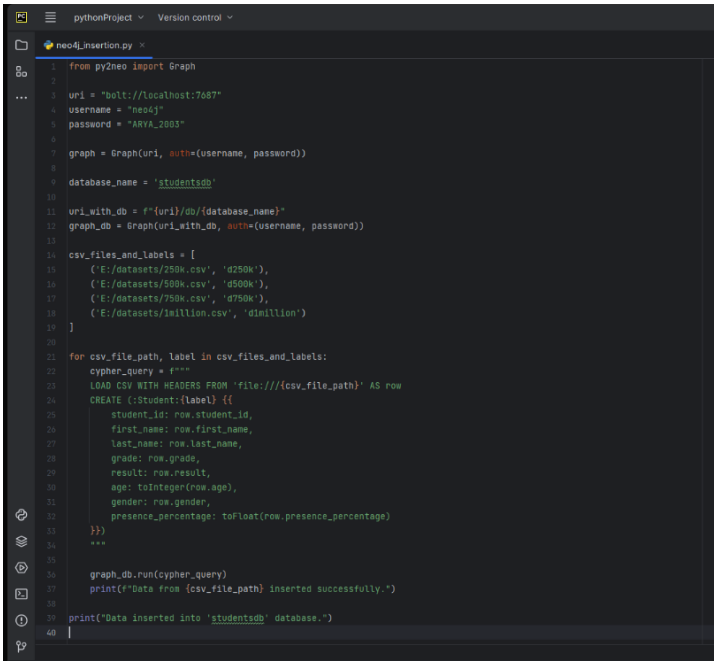
```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\aryak>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES

C:\Users\aryak>docker exec -it cassandra-container cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.3 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> DESCRIBE KEYSPACES;

students     system_distributed  system_views
system       system_schema       system_virtual_schema
system_auth  system_traces       your_keyspace_name

cqlsh> USE students;
cqlsh:students> DESCRIBE TABLES;

table_1million  table_250k  table_500k  table_750k

cqlsh:students> SELECT * FROM table_500k;

 student_id                           | age | first_name | gender | grade | last_name | presence_percentage | result
--------------------------------------+-----+------------+--------+-------+-----------+---------------------+--------
 bb1dcfbe-fed5-4e71-9bdc-7ae99998a92b |  25 |  Alexandra |   Male |    30 |  Thompson |               78.15 |   Pass
 54ec6ca0-74b7-433b-8cfc-8873a359bdd0 |  18 |      Sara |   Male |     5 |      Rice |               88.24 |   Fail
 a1ca2c95-3428-413e-a97c-3858b6f5e6d0 |  24 |     Justin | Female |    27 |   Johnson |               72.58 |   Pass
 9163e85c-decb-4e72-a38f-12d339bfc896 |  18 |      Kevin |   Male |    27 |    Valdez |               85.82 |   Pass
 c61ca94c-2fdf-46c0-9817-5b353f1d27b1 |  19 |      Kathy |   Male |     6 |    Garcia |               78.31 |   Fail
 1325de82-3bf1-4439-8c86-8c457c543126 |  23 |    Shannon |   Male |     3 |    Powell |               87.21 |   Fail
 374bea46-6f97-47e3-8974-c7089b509046 |  25 |       Tina | Female |    23 |   Johnson |               99.34 |   Pass
 de25bc99-d27c-43f8-b6a7-e71695c66b63 |  22 |    Katelyn |   Male |    25 |   Douglas |               98.13 |   Pass
 8780b139-b9e9-4b01-a758-d0d1f962e2dd |  21 |      David | Female |    10 |    Orozco |               95.43 |   Fail
 fd04766a-9c56-4a7d-94d8-85b6b0d170a2 |  23 |  Stephanie | Female |    15 |    Vaughn |               74.32 |   Fail
 d802af70-542b-4421-bd49-5a241c298791 |  18 |     Victor |   Male |    17 |    Peters |               83.13 |   Fail
 8598e6c1-0305-4bc0-b4c0-873d60743e5d |  19 |      April |   Male |     8 |     Smith |               84.75 |   Fail
 760c1968-5376-4414-a4d5-3475abcfa4ef |  20 |  Alexander | Female |    24 |   Barnett |               82.47 |   Pass
 c1daecb5-3655-4835-a597-f870de6d54f8 |  23 |     Pamela | Female |    20 |     Wells |               74.41 |   Pass
 494ded42-1955-4b71-accd-f89ae507dd07 |  24 |    Jasmine | Female |    20 |     Evans |               85.64 |   Pass
 216dffca-7de9-4151-afe8-bee17ab1821  |  22 |    Heather | Female |    16 |      Neal |               70.58 |   Fail
```

After executing the code you can connect your container to cqlsh in command prompt and to see the keyspaces, tables and retrieving the data in them

Neo4j:

This script serves as a fundamental tool for inserting data into a Neo4j database. It establishes a connection to the Neo4j instance, creates a database named 'studentsdb,' and imports data from CSV files into it. The script utilizes the Cypher query language with the 'LOAD CSV' command to seamlessly integrate data from the CSV files into the Neo4j database.

The script is designed to efficiently handle data insertion for various datasets, and it is well-suited for scenarios where you need to work with different datasets in Neo4j. It simplifies the process of data integration and ensures the seamless storage of information in your Neo4j 'studentsdb' database, making it a valuable asset for your project's data management.



QUERIES

The provided code samples implement query execution on four different database systems: MySQL, MongoDB, Redis, and Cassandra. Each code sample contains four progressively complex queries designed to retrieve and analyze data from their respective databases. These queries are designed to assess and optimize the performance of each database in handling different levels of complexity in data retrieval and analysis.

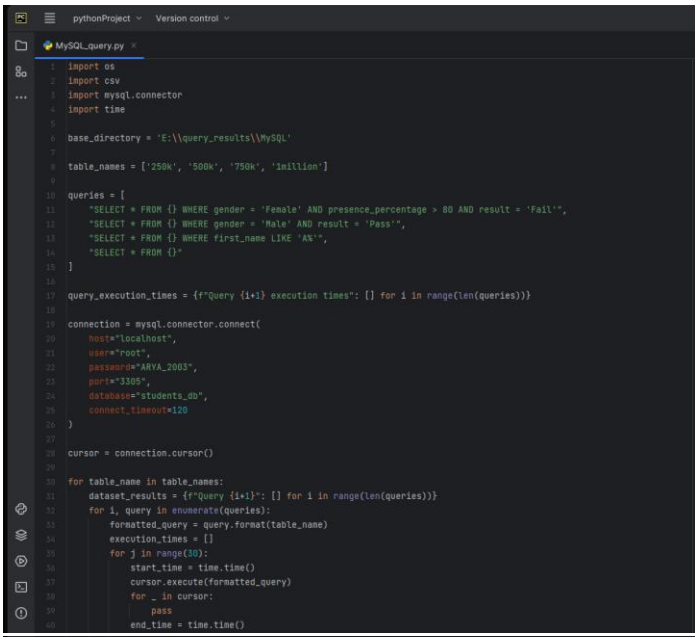
The queries used in all databases are as follows:

1. **Query 1: Retrieve records for females with a presence percentage greater than 80% who failed: This query selects all columns from a specified table where the gender is 'Female,' the presence percentage is greater than 80, and the result is 'Fail.'**
2. **Query 2: Retrieve records for males who passed: This query selects all columns from a specified table where the gender is 'Male' and the result is 'Pass.'**
3. **Query 3: Retrieve records for individuals whose first names start with 'A': This query selects all columns from a specified table where the first name starts with the letter 'A.'**
4. **Query 4: Retrieve all records from the specified table: This query selects all columns from a specified table without applying any specific conditions, essentially fetching all available records.**

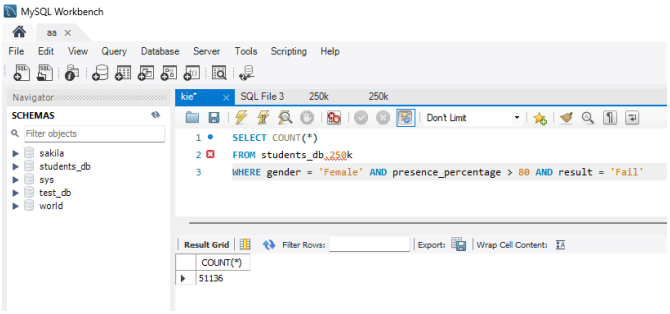
These queries collectively provide a comprehensive assessment of each database's ability to handle varying levels of data retrieval complexity.

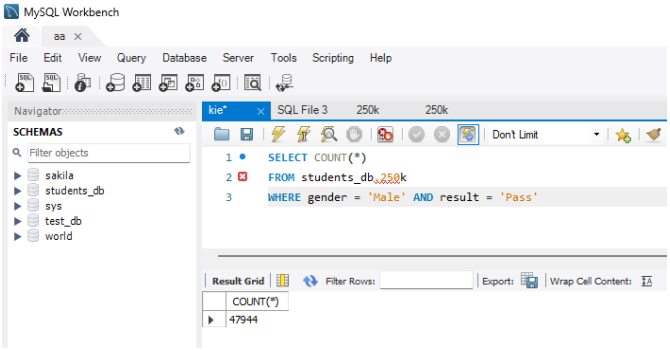
MySQL Code:

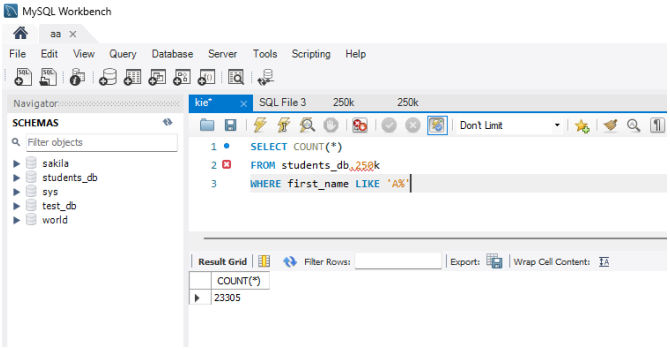
- **The MySQL code connects to a MySQL database and executes four progressively complex SQL queries. It measures query execution times on different tables, allowing for performance evaluation and optimization.**

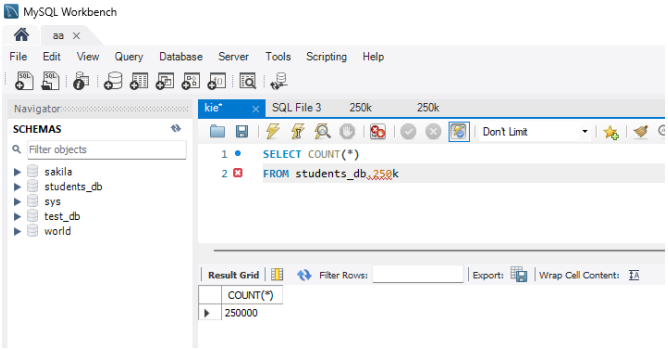


```python
import os
import csv
import mysql.connector
import time

base_directory = 'E:\\query_results\\MySQL'

table_names = ['250k', '500k', '750k', '1million']

queries = [
    "SELECT * FROM {} WHERE gender = 'Female' AND presence_percentage > 80 AND result = 'Fail'",
    "SELECT * FROM {} WHERE gender = 'Male' AND result = 'Pass'",
    "SELECT * FROM {} WHERE first_name LIKE 'A%'",
    "SELECT * FROM {}"
]

query_execution_times = {f"Query {i+1} execution times": [] for i in range(len(queries))}

connection = mysql.connector.connect(
    host="localhost",
    user="root",
    password="ARYA_2003",
    port="3305",
    database="students_db",
    connect_timeout=120
)

cursor = connection.cursor()

for table_name in table_names:
    dataset_results = {f"Query {i+1}": [] for i in range(len(queries))}
    for i, query in enumerate(queries):
        formatted_query = query.format(table_name)
        execution_times = []
        for j in range(30):
            start_time = time.time()
            cursor.execute(formatted_query)
            for _ in cursor:
                pass
            end_time = time.time()
            execution_time = int((end_time - start_time) * 1e9)
            execution_times.append(execution_time)

            if j == 0:
                print(f"Table: {table_name}, Query {i+1}, First Execution Time: {execution_time} nanoseconds")
            if j == 29:
                avg_execution_time = sum(execution_times) // len(execution_times)
                print(f"Table: {table_name}, Query {i+1}, Average Execution Time: {avg_execution_time} nanoseconds")

        query_execution_times[f"Query {i+1} execution times"].append(execution_times)
        dataset_results[f"Query {i+1}"] = execution_times

    filename = os.path.join(base_directory, f"results_{table_name}.csv")
    with open(filename, 'w', newline='') as result_file:
        csv_writer = csv.writer(result_file)
        csv_writer.writerow(['Query', 'Execution Times'])
        for query_num in range(1, 5):
            query_label = f"Query {query_num}"
            response_times = dataset_results[query_label]
            csv_writer.writerow([query_label] + response_times)

cursor.close()
connection.close()
```


MySQL Workbench
File Edit View Query Database Server Tools Scripting Help
SCHEMAS
sakila
students_db
sys
test_db
world
SELECT COUNT(*)
FROM students_db.250k
WHERE gender = 'Female' AND presence_percentage > 80 AND result = 'Fail'
Result Grid
COUNT(*)
51136

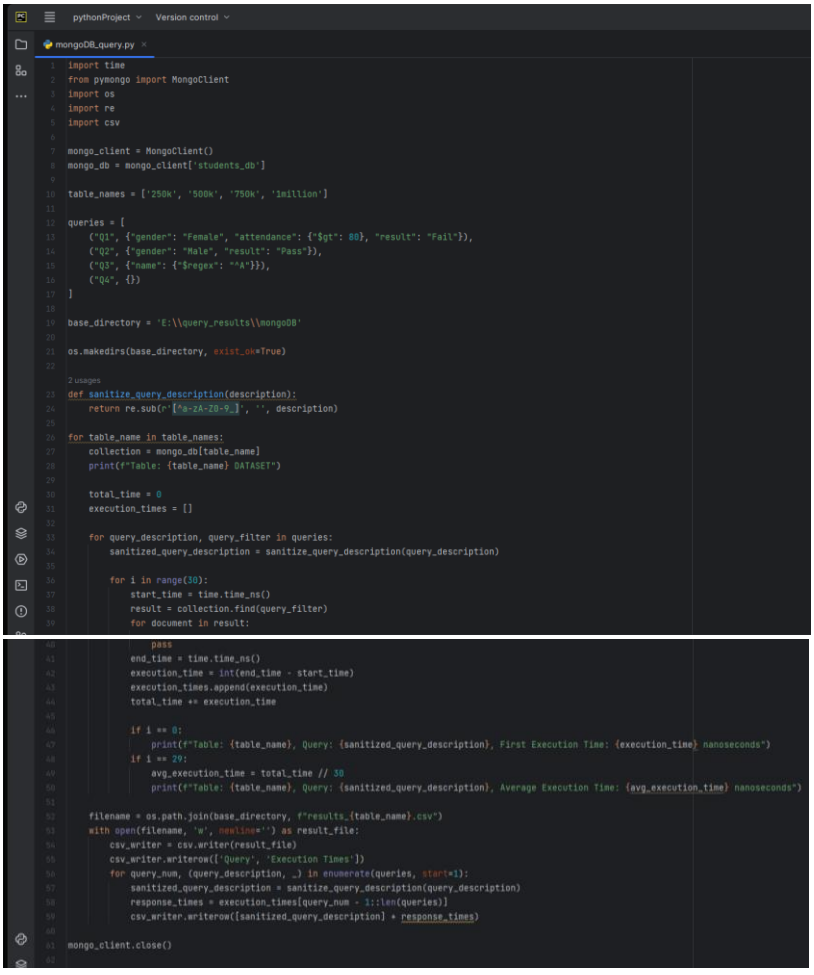

MySQL Workbench
File Edit View Query Database Server Tools Scripting Help
SCHEMAS
sakila
students_db
sys
test_db
world
SELECT COUNT(*)
FROM students_db.250k
WHERE gender = 'Male' AND result = 'Pass'
Result Grid
COUNT(*)
47944


MySQL Workbench
File Edit View Query Database Server Tools Scripting Help
SCHEMAS
sakila
students_db
sys
test_db
world
SELECT COUNT(*)
FROM students_db.250k
WHERE first_name LIKE 'A%'
Result Grid
COUNT(*)
23305

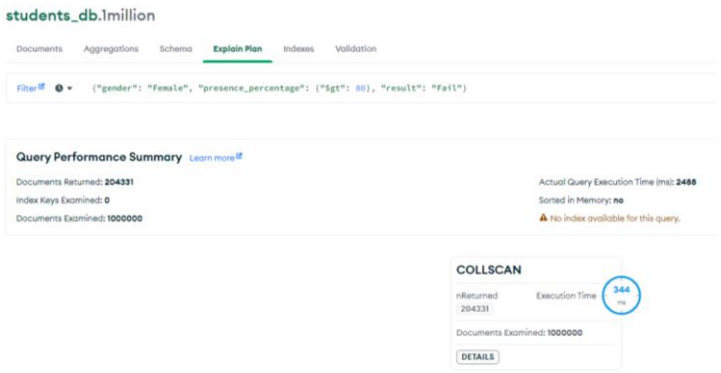

MySQL Workbench
File Edit View Query Database Server Tools Scripting Help
SCHEMAS
sakila
students_db
sys
test_db
world
SELECT COUNT(*)
FROM students_db.250k
Result Grid
COUNT(*)
250000

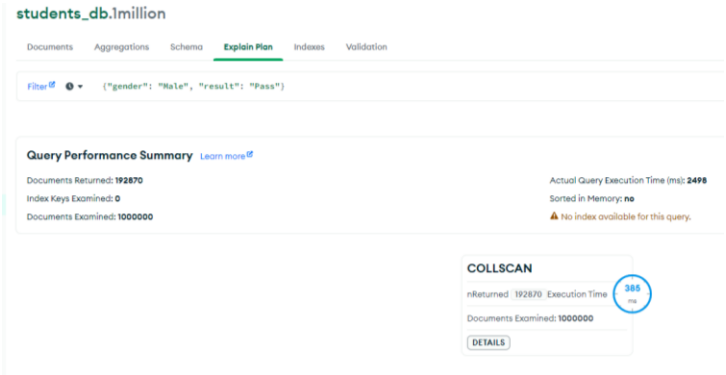
MongoDB Code:

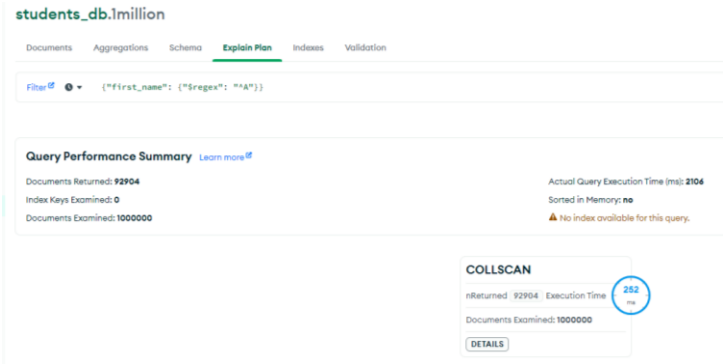
- **The MongoDB code connects to a MongoDB database and runs four queries with increasing complexity. It times the execution of these queries and saves the results for performance analysis.**

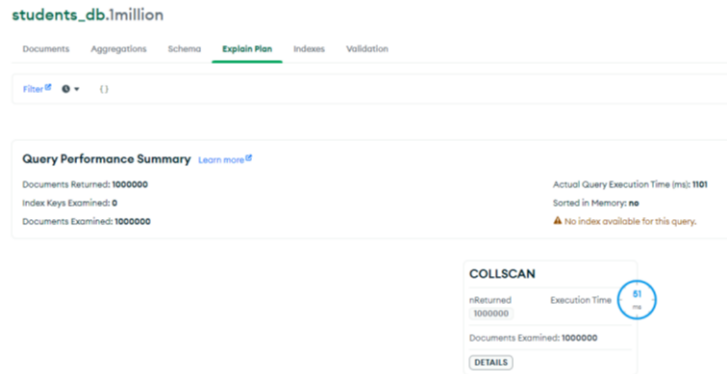


```python
import time
from pymongo import MongoClient
import os
import re
import csv

mongo_client = MongoClient()
mongo_db = mongo_client['students_db']

table_names = ['250k', '500k', '750k', '1million']

queries = [
    ("Q1", {"gender": "Female", "attendance": {"$gt": 80}, "result": "Fail"}),
    ("Q2", {"gender": "Male", "result": "Pass"}),
    ("Q3", {"name": {"$regex": "^A"}}),
    ("Q4", {})
]

base_directory = 'E:\\query_results\\mongoDB'

os.makedirs(base_directory, exist_ok=True)

2 usages
def sanitize_query_description(description):
    return re.sub(r'[^a-zA-Z0-9_]', '', description)

for table_name in table_names:
    collection = mongo_db[table_name]
    print(f"Table: {table_name} DATASET")

    total_time = 0
    execution_times = []

    for query_description, query_filter in queries:
        sanitized_query_description = sanitize_query_description(query_description)

        for i in range(30):
            start_time = time.time_ns()
            result = collection.find(query_filter)
            for document in result:
                pass
            end_time = time.time_ns()
            execution_time = int(end_time - start_time)
            execution_times.append(execution_time)
            total_time += execution_time

            if i == 0:
                print(f"Table: {table_name}, Query: {sanitized_query_description}, First Execution Time: {execution_time} nanoseconds")
            if i == 29:
                avg_execution_time = total_time // 30
                print(f"Table: {table_name}, Query: {sanitized_query_description}, Average Execution Time: {avg_execution_time} nanoseconds")

    filename = os.path.join(base_directory, f"results_{table_name}.csv")
    with open(filename, 'w', newline='') as result_file:
        csv_writer = csv.writer(result_file)
        csv_writer.writerow(['Query', 'Execution Times'])
        for query_num, (query_description, _) in enumerate(queries, start=1):
            sanitized_query_description = sanitize_query_description(query_description)
            response_times = execution_times[query_num - 1::len(queries)]
            csv_writer.writerow([sanitized_query_description] + response_times)

mongo_client.close()
```


students_db.1million
Documents
Aggregations
Schema
Explain Plan
Indexes
Validation
Filter
{"gender": "Female", "presence_percentage": {"$gt": 80}, "result": "Fail"}
Query Performance Summary
Learn more
Documents Returned: 204331
Index Keys Examined: 0
Documents Examined: 1000000
Actual Query Execution Time (ms): 2488
Sorted in Memory: no
No index available for this query.
COLLSCAN
nReturned
204331
Execution Time
344
ms
Documents Examined: 1000000
DETAILS

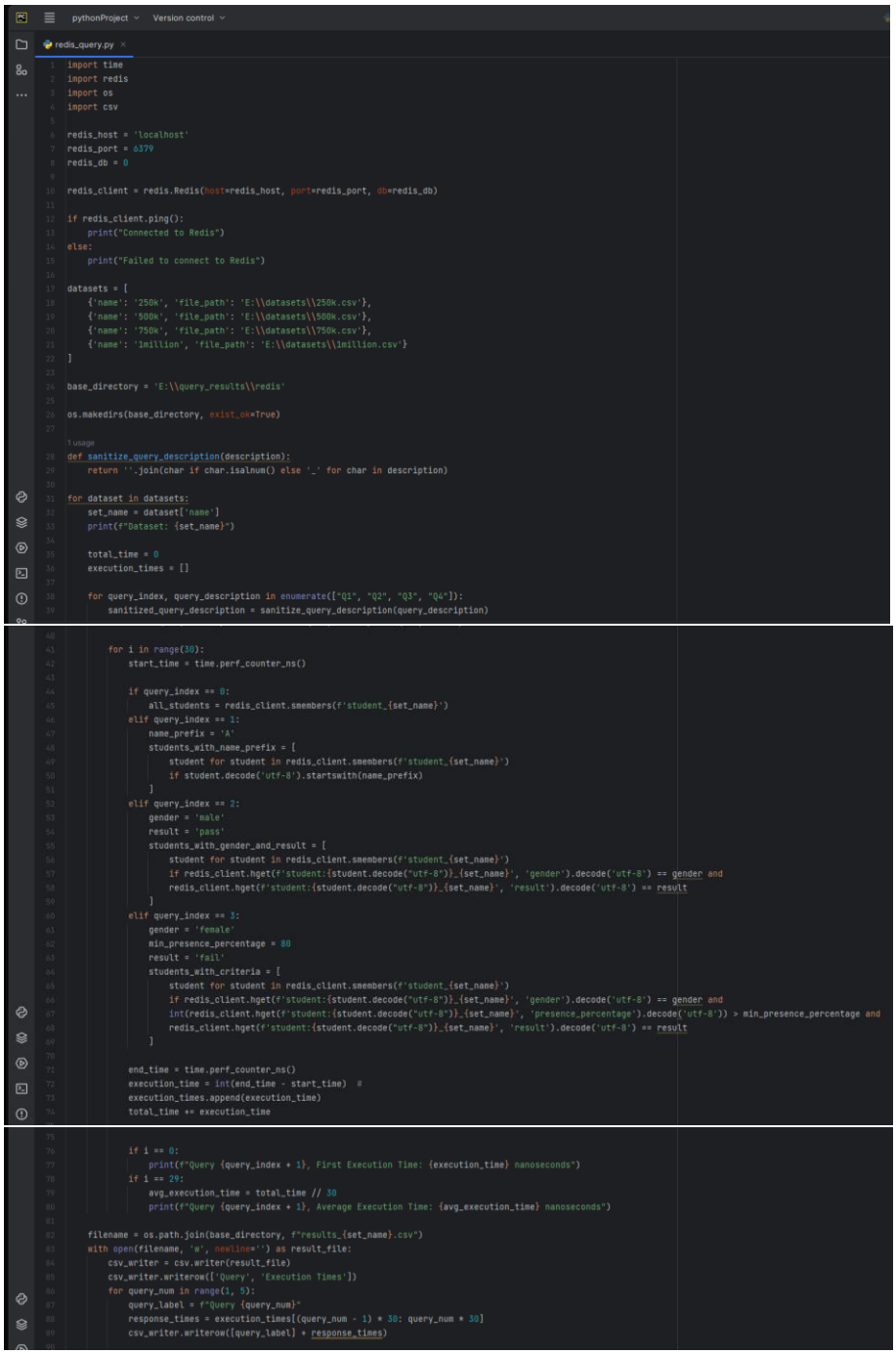

students_db.1million
Documents
Aggregations
Schema
Explain Plan
Indexes
Validation
Filter
{"gender": "Male", "result": "Pass"}
Query Performance Summary
Learn more
Documents Returned: 192870
Index Keys Examined: 0
Documents Examined: 1000000
Actual Query Execution Time (ms): 2498
Sorted in Memory: no
No index available for this query.
COLLSCAN
nReturned
192870
Execution Time
385
ms
Documents Examined: 1000000
DETAILS


students_db.1million
Documents
Aggregations
Schema
Explain Plan
Indexes
Validation
Filter
{"first_name": {"$regex": "^A"}}
Query Performance Summary
Learn more
Documents Returned: 92904
Index Keys Examined: 0
Documents Examined: 1000000
Actual Query Execution Time (ms): 2106
Sorted in Memory: no
No index available for this query.
COLLSCAN
nReturned
92904
Execution Time
252
ms
Documents Examined: 1000000
DETAILS


students_db.1million
Documents
Aggregations
Schema
Explain Plan
Indexes
Validation
Filter
{}
Query Performance Summary
Learn more
Documents Returned: 1000000
Index Keys Examined: 0
Documents Examined: 1000000
Actual Query Execution Time (ms): 1101
Sorted in Memory: no
No index available for this query.
COLLSCAN
nReturned
1000000
Execution Time
81
ms
Documents Examined: 1000000
DETAILS

Redis Code:

- **The Redis code connects to a Redis database and executes four queries. It measures query execution times for various datasets, aiding in performance assessment.**



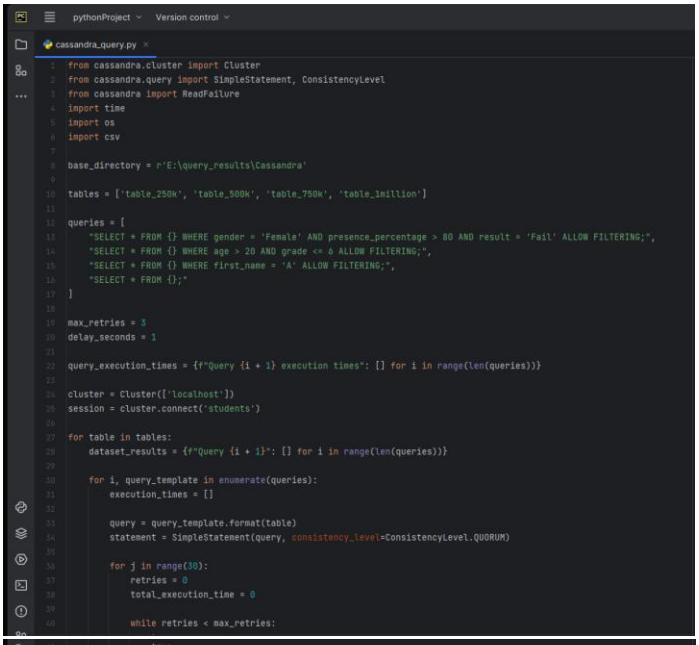
```python
import time
import redis
import os
import csv

redis_host = 'localhost'
redis_port = 6379
redis_db = 0

redis_client = redis.Redis(host=redis_host, port=redis_port, db=redis_db)

if redis_client.ping():
    print("Connected to Redis")
else:
    print("Failed to connect to Redis")

datasets = [
    {'name': '250k', 'file_path': 'E:\\datasets\\250k.csv'},
    {'name': '500k', 'file_path': 'E:\\datasets\\500k.csv'},
    {'name': '750k', 'file_path': 'E:\\datasets\\750k.csv'},
    {'name': '1million', 'file_path': 'E:\\datasets\\1million.csv'}
]

base_directory = 'E:\\query_results\\redis'

os.makedirs(base_directory, exist_ok=True)

1 usage
def sanitize_query_description(description):
    return ''.join(char if char.isalnum() else '_' for char in description)

for dataset in datasets:
    set_name = dataset['name']
    print(f"Dataset: {set_name}")

    total_time = 0
    execution_times = []

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

```python
        for i in range(30):
            start_time = time.perf_counter_ns()

            if query_index == 0:
                all_students = redis_client.smembers(f'student_{set_name}')
            elif query_index == 1:
                name_prefix = 'A'
                students_with_name_prefix = [
                    student for student in redis_client.smembers(f'student_{set_name}')
                    if student.decode('utf-8').startswith(name_prefix)
                ]
            elif query_index == 2:
                gender = 'male'
                result = 'pass'
                students_with_gender_and_result = [
                    student for student in redis_client.smembers(f'student_{set_name}')
                    if redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'gender').decode('utf-8') == gender and
                    redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'result').decode('utf-8') == result
                ]
            elif query_index == 3:
                gender = 'female'
                min_presence_percentage = 80
                result = 'fail'
                students_with_criteria = [
                    student for student in redis_client.smembers(f'student_{set_name}')
                    if redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'gender').decode('utf-8') == gender and
                    int(redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'presence_percentage').decode('utf-8')) > min_presence_percentage and
                    redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'result').decode('utf-8') == result
                ]

            end_time = time.perf_counter_ns()
            execution_time = int(end_time - start_time)  #
            execution_times.append(execution_time)
            total_time += execution_time
```

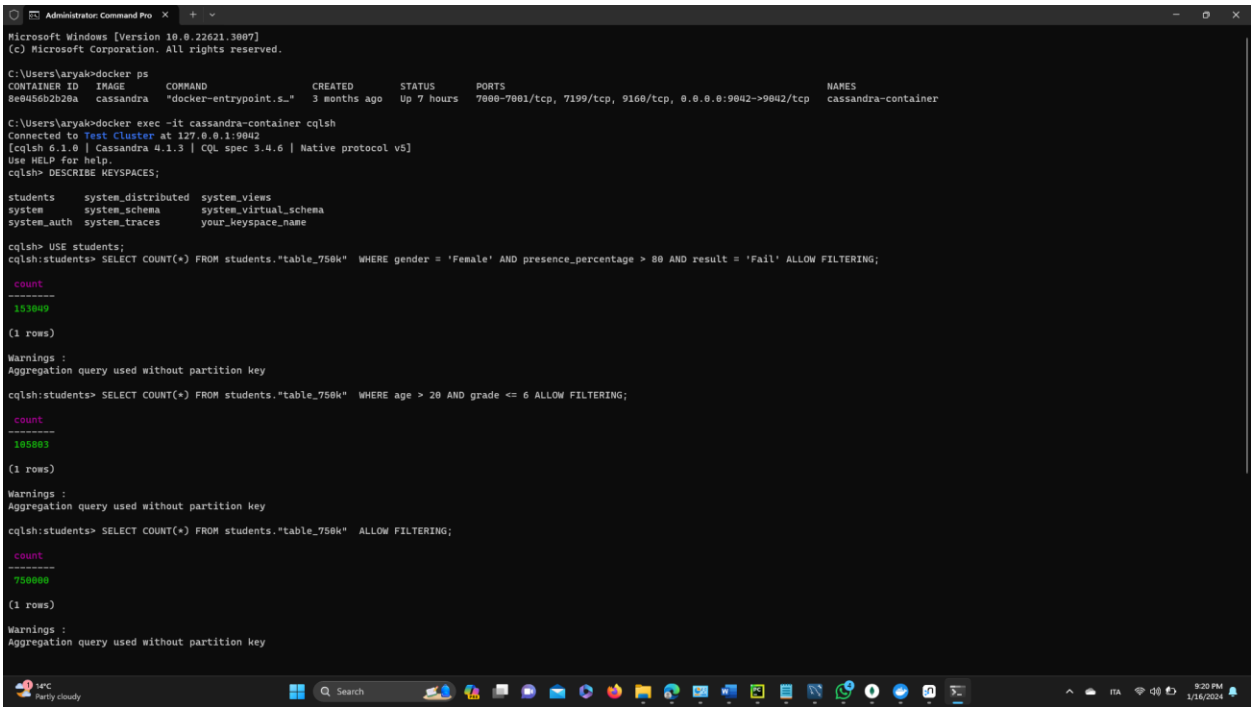
```python
            if i == 0:
                print(f"Query {query_index + 1}, First Execution Time: {execution_time} nanoseconds")
            if i == 29:
                avg_execution_time = total_time // 30
                print(f"Query {query_index + 1}, Average Execution Time: {avg_execution_time} nanoseconds")

    filename = os.path.join(base_directory, f"results_{set_name}.csv")
    with open(filename, 'w', newline='') as result_file:
        csv_writer = csv.writer(result_file)
        csv_writer.writerow(['Query', 'Execution Times'])
        for query_num in range(1, 5):
            query_label = f"Query {query_num}"
            response_times = execution_times[(query_num - 1) * 30: query_num * 30]
            csv_writer.writerow([query_label] + response_times)
```

Cassandra Code:

- **The Cassandra code connects to a Cassandra cluster and performs four queries, each with a different level of complexity. It measures the execution times and handles query retries for performance evaluation.**



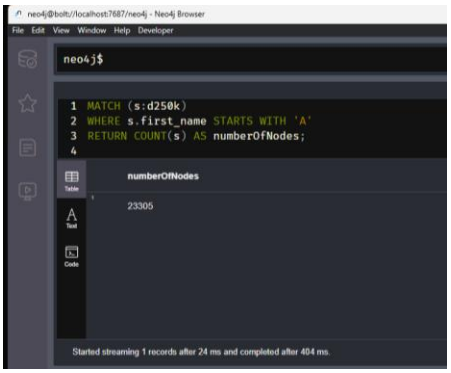
```python
from cassandra.cluster import Cluster
from cassandra.query import SimpleStatement, ConsistencyLevel
from cassandra import ReadFailure
import time
import os
import csv

base_directory = r'E:\query_results\Cassandra'

tables = ['table_250k', 'table_500k', 'table_750k', 'table_1million']

queries = [
    "SELECT * FROM {} WHERE gender = 'Female' AND presence_percentage > 80 AND result = 'Fail' ALLOW FILTERING;",
    "SELECT * FROM {} WHERE age > 20 AND grade <= 6 ALLOW FILTERING;",
    "SELECT * FROM {} WHERE first_name = 'A' ALLOW FILTERING;",
    "SELECT * FROM {};"
]

max_retries = 3
delay_seconds = 1

query_execution_times = {f"Query {i + 1} execution times": [] for i in range(len(queries))}

cluster = Cluster(['localhost'])
session = cluster.connect('students')

for table in tables:
    dataset_results = {f"Query {i + 1}": [] for i in range(len(queries))}

    for i, query_template in enumerate(queries):
        execution_times = []

        query = query_template.format(table)
        statement = SimpleStatement(query, consistency_level=ConsistencyLevel.QUORUM)

        for j in range(30):
            retries = 0
            total_execution_time = 0

            while retries < max_retries:
                try:
                    start_time = time.time()
                    result = session.execute(statement)
                    for row in result:
                        pass
                    end_time = time.time()
                    execution_time = int((end_time - start_time) * 1e9)
                    execution_times.append(execution_time)
                    total_execution_time += execution_time
                    break
                except ReadFailure as e:
                    print(f"ReadFailure on execution {j + 1} of Query {i + 1}: {e}")
                    retries += 1
                    if retries < max_retries:
                        delay = delay_seconds * (2 ** (retries - 1))
                        print(f"Retrying in {delay} seconds...")
                        time.sleep(delay)
                    else:
                        print(f"Failed after retries: {e}")

            if j == 0:
                print(f"Table: {table}, Query {i + 1}, First Execution Time: {execution_time} nanoseconds")
            if j == 29:
                avg_execution_time = total_execution_time // (retries + 1)
                print(f"Table: {table}, Query {i + 1}, Average Execution Time: {avg_execution_time} nanoseconds")

        query_execution_times[f"Query {i + 1} execution times"].append(execution_times)
        dataset_results[f"Query {i + 1}"] = execution_times

    filename = os.path.join(base_directory, f"results_{table.replace('table_', '')}.csv")
    with open(filename, 'w', newline='') as result_file:
        csv_writer = csv.writer(result_file)
        csv_writer.writerow(['Query', 'Execution Times'])
        for query_num in range(1, 5):
            query_label = f"Query {query_num}"
            response_times = dataset_results[query_label]
            csv_writer.writerow([query_label] + response_times)

session.shutdown()
cluster.shutdown()
```

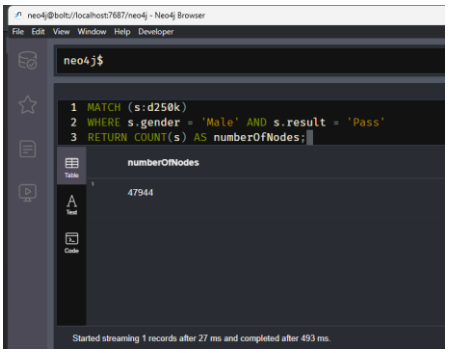

Administrator: Command Pro
Microsoft Windows [Version 10.0.22621.3007]
(c) Microsoft Corporation. All rights reserved.
C:\Users\aryak>docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
8e0456b2b20a cassandra "docker-entrypoint.s…" 3 months ago Up 7 hours 7000-7001/tcp, 7199/tcp, 9160/tcp, 0.0.0.0:9042->9042/tcp cassandra-container
C:\Users\aryak>docker exec -it cassandra-container cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.3 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> DESCRIBE KEYSPACES;
students system_distributed system_views
system system_schema system_virtual_schema
system_auth system_traces your_keyspace_name
cqlsh> USE students;
cqlsh:students> SELECT COUNT(*) FROM students."table_750k" WHERE gender = 'Female' AND presence_percentage > 80 AND result = 'Fail' ALLOW FILTERING;
count
153049
(1 rows)
Warnings :
Aggregation query used without partition key
cqlsh:students> SELECT COUNT(*) FROM students."table_750k" WHERE age > 20 AND grade <= 6 ALLOW FILTERING;
count
105803
(1 rows)
Warnings :
Aggregation query used without partition key
cqlsh:students> SELECT COUNT(*) FROM students."table_750k" ALLOW FILTERING;
count
750000
(1 rows)
Warnings :
Aggregation query used without partition key
14°C
Partly cloudy
Search
ITA
9:20 PM
1/16/2024

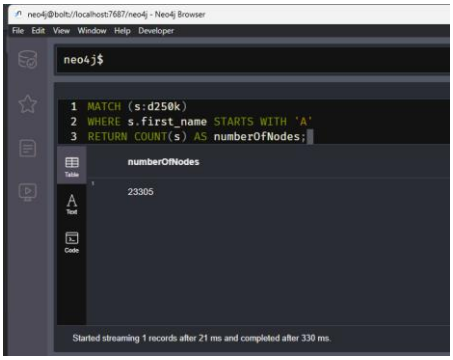
Neo4j Code:

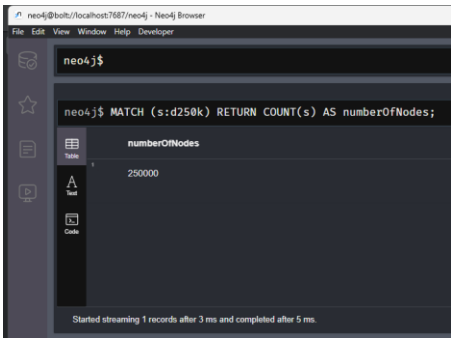
- **The Neo4j code connects to a Neo4j database and runs four queries on nodes with different labels. It times the execution of these queries and saves the results for performance analysis.**

```
from py2neo import Graph, DatabaseError
import time
import os
import csv

uri = "bolt://localhost:7687"
username = "neo4j"
password = "ARYA_2003"

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

labels = ['d250k', 'd500k', 'd750k', 'd1million']
datasets = ['250k', '500k', '750k', '1million']

base_directory = 'E:\\query_results\\neo4j'

os.makedirs(base_directory, exist_ok=True)

def sanitize_query_description(description):
    return ''.join(char if char.isalnum() else '_' for char in description)

queries = [
    "MATCH (s:Student:{}) WHERE s.gender = 'Female' AND s.presence_percentage > 80 AND s.result = 'Fail' RETURN s",
    "MATCH (s:Student:{}) WHERE s.gender = 'Male' AND s.result = 'Pass' RETURN s",
    "MATCH (s:Student:{}) WHERE s.first_name STARTS WITH 'A' RETURN s",
    "MATCH (s:Student:{}) RETURN s"
]

for label, dataset in zip(labels, datasets):
    print(f"Queries for label '{label}':")

    dataset_results = {f"Query {i}": [] for i in range(1, 5)}

    for query_idx, query_template in enumerate(queries, start=1):
        total_time = 0
        execution_times = []

        for i in range(1, 31):
            try:
                start_time = time.perf_counter_ns()
                result = graph.run(query_template.format(label))
                for record in result:
                    pass
                end_time = time.perf_counter_ns()
                execution_time = int(end_time - start_time)
                execution_times.append(execution_time)
                total_time += execution_time

                if i == 1:
                    print(f"Label: {label}, Query {query_idx}, First Execution Time: {execution_time} nanoseconds")
                if i == 30:
                    avg_execution_time = total_time // 30
                    print(f"Label: {label}, Query {query_idx}, Average Execution Time: {avg_execution_time} nanoseconds.")

            except DatabaseError as e:
                print(f"Error executing query: {e}")
                time.sleep(1)
                continue

        dataset_results[f"Query {query_idx}"] = execution_times

    filename = os.path.join(base_directory, f"results_{dataset}.csv")
    with open(filename, 'w', newline='') as result_file:
        csv_writer = csv.writer(result_file)
        csv_writer.writerow(['Query', 'Execution Times'])
        for query_num in range(1, 5):
            query_label = f"Query {query_num}"
            response_times = dataset_results[query_label]
            csv_writer.writerow([query_label] + response_times)
```


neo4j@bolt://localhost:7687/neo4j - Neo4j Browser
File Edit View Window Help Developer
neo4j$
1 MATCH (s:d250k)
2 WHERE s.first_name STARTS WITH 'A'
3 RETURN COUNT(s) AS numberOfNodes;
4
Table
Text
Code
numberOfNodes
1
23305
Started streaming 1 records after 24 ms and completed after 404 ms.

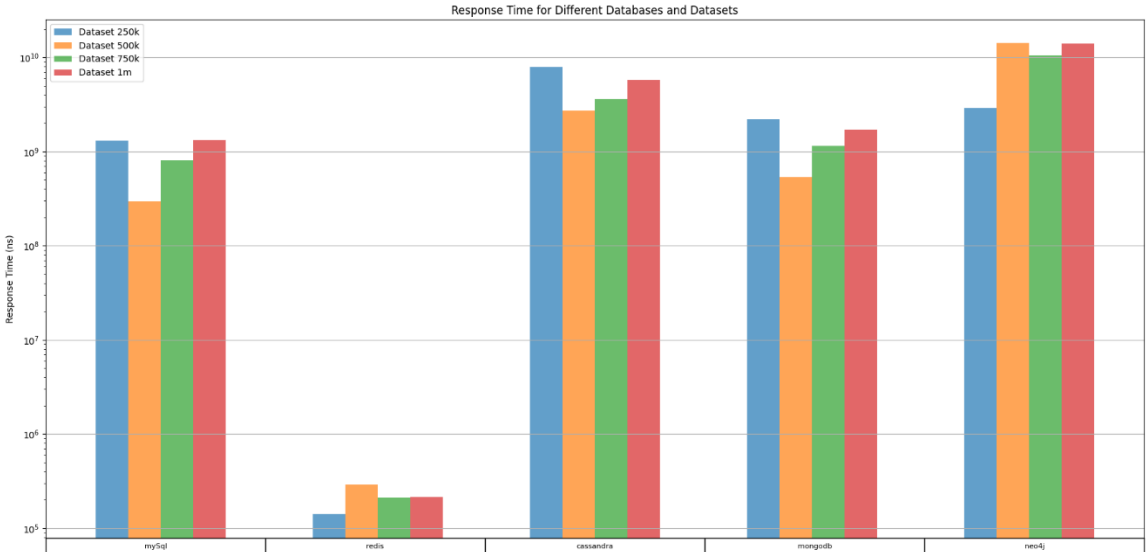

neo4j@bolt://localhost:7687/neo4j - Neo4j Browser
File Edit View Window Help Developer
neo4j$
1 MATCH (s:d250k)
2 WHERE s.gender = 'Male' AND s.result = 'Pass'
3 RETURN COUNT(s) AS numberOfNodes;
Table
Text
Code
numberOfNodes
1
47944
Started streaming 1 records after 27 ms and completed after 493 ms.


neo4j@bolt://localhost:7687/neo4j - Neo4j Browser
File Edit View Window Help Developer
neo4j$
1 MATCH (s:d250k)
2 WHERE s.first_name STARTS WITH 'A'
3 RETURN COUNT(s) AS numberOfNodes;
Table
Text
Code
numberOfNodes
1
23305
Started streaming 1 records after 21 ms and completed after 330 ms.


neo4j@bolt://localhost:7687/neo4j - Neo4j Browser
File Edit View Window Help Developer
neo4j$
neo4j$ MATCH (s:d250k) RETURN COUNT(s) AS numberOfNodes;
Table
Text
Code
numberOfNodes
1
250000
Started streaming 1 records after 3 ms and completed after 5 ms.

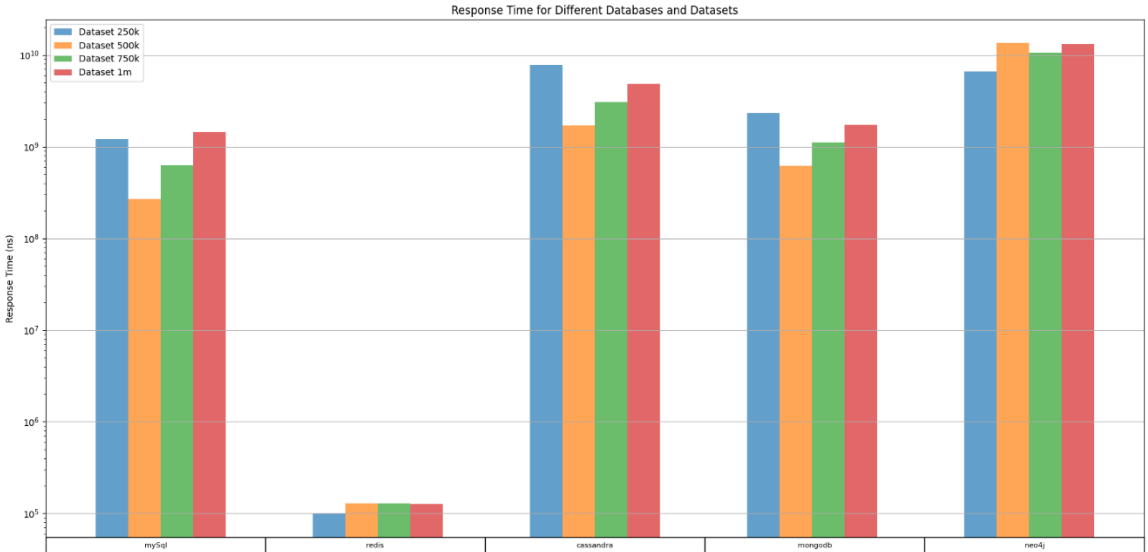
HISTOGRAMS

Query 1: Retrieve records for females with a presence percentage greater than 80% who failed :



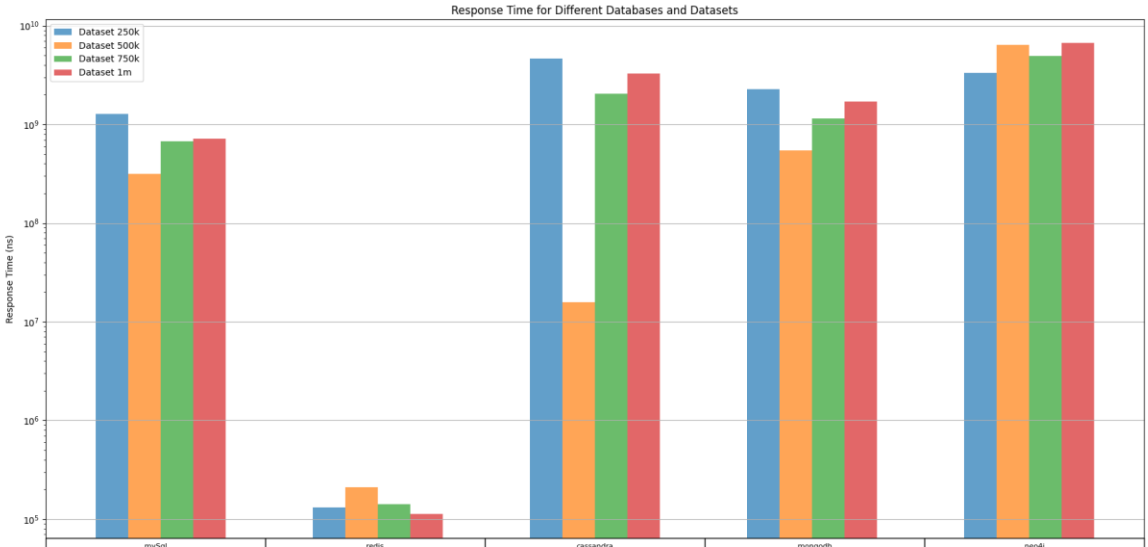
	mySql	redis	cassandra	mongodb	neo4j
Dataset 250k	1310356616	140700	7860320806	2192306900	2876127500
Dataset 500k	294651031	290600	2706038475	536941300	14176234200
Dataset 750k	809846639	212600	3607409954	1157385000	10587613900
Dataset 1m	1334067344	213300	5696088790	1716898100	14083852000

Query 2: Retrieve records for males who passed:



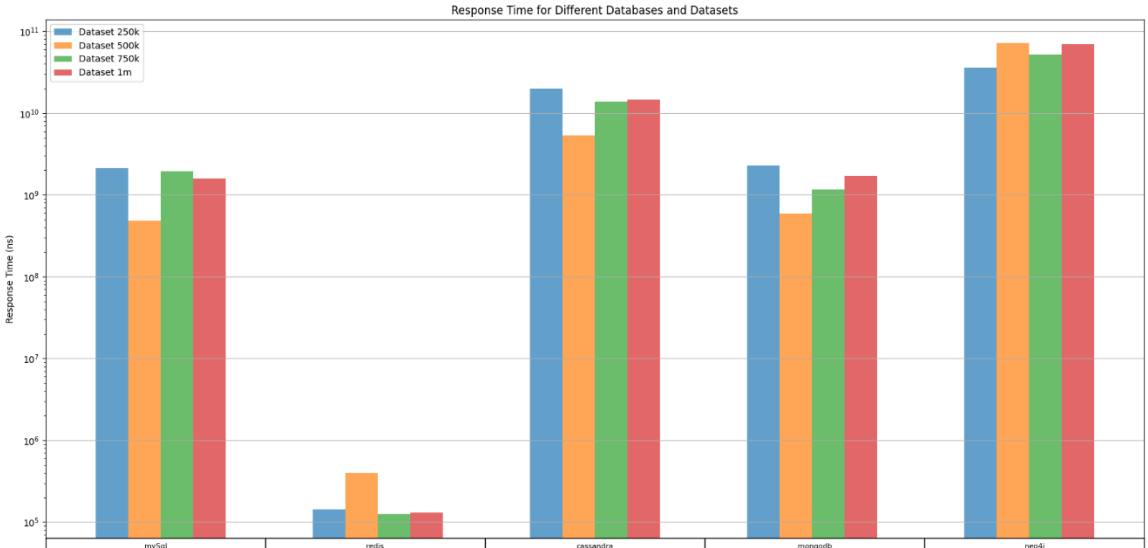
	mySql	redis	cassandra	mongodb	neo4j
Dataset 250k	1210894584	100000	7727785348	2319521900	6570884100
Dataset 500k	267594099	128500	1699537515	618698400	13501794700
Dataset 750k	629316091	128400	3080528497	1120181600	10592835700
Dataset 1m	1445851564	126800	4817503690	1731279800	13064315000

Query 3: Retrieve records for individuals whose first names start with 'A':



	mySql	redis	cassandra	mongodb	neo4j
Dataset 250k	1285355091	131000	4621105194	2289433400	3326304200
Dataset 500k	315414905	211200	15684843	542409800	6436652200
Dataset 750k	677571535	141400	2030141353	1148216200	4910278700
Dataset 1m	721738100	112000	3256367444	1701563400	6667711800

Query 4: Retrieve all records from the specified table:



	mySql	redis	cassandra	mongodb	neo4j
Dataset 250k	2125322103	143000	20063423395	2286756500	36223234000
Dataset 500k	486298799	399400	5303692817	594328300	71811610600
Dataset 750k	1945503950	124500	13865430355	1157951300	51917116600
Dataset 1m	1593664884	129800	14732636451	1699974900	69533956200

CONCLUSION

The project's graphs reveal clear trends in the performance of different databases—Redis emerges as the fastest, followed by MongoDB and MySQL, which show comparable speeds. Cassandra performs a bit slower than Redis, MongoDB, and MySQL. Notably, Neo4j appears to be the slowest among the databases.

For handling larger amounts of data (1 million entries), Cassandra shows reasonable scalability, but it's crucial to consider factors like data complexity when deciding on a database.

The bar graphs provide a quick way to see which databases are consistently faster. These visualizations assist decision-makers in understanding performance patterns and making informed choices.

REFERENCES

1. MySQL Documentation
2. MongoDB Documentation
3. Redis Documentation
4. Neo4j Documentation
5. Cassandra Documentation
6. PyCharm Documentation
7. Python Official Documentation