Università degli Studi di Messina

2/12/2024

Database Mod B

Professor: Armando Ruggeri

ADEEBULLAH HAMIDY 539745

Online Admissions System

Contents

Introduction

The project is aimed to execute queries on different databases and different datasizes in order to compare the timing and efficiency of different databases and to delve deeper into pros and cons of different databases which are: Mysql, Mongodb, Neo4j, Cassandra and Redis. The dataset is generated using faker library in python and the queries are all written in python with database-related languages. The queries have been executed in the same software/hardware so the runtime will be comparable. At the end, histograms are captured from each database and dataset to compare all the runtime from different databases and at the end select one database considering its UI and speed in order to work with.

Database used in the project:

1. Mysql:

MySQL (/ˌmaɪˌɛsˌkjuːˈɛl/) is an open-source relational database management system (RDBMS).Its name is a combination of "My", the name of co-founder Michael Widenius's daughter My, and "SQL", the acronym for Structured Query Language. A relational database organizes data into one or more data tables in which data may be related to each other; these relations help structure the data. SQL is a language that programmers use to create, modify and extract data from the relational database, as well as control user access to the database. In addition to relational databases and SQL, an RDBMS like MySQL works with an operating system to implement a relational database in a computer's storage system, manages users, allows for network access and facilitates testing database integrity and creation of backups.

MySQL is free and open-source software under the terms of the GNU General Public License, and is also available under a variety of proprietary licenses. MySQL was owned and sponsored by the Swedish company MySQL AB, which was bought by Sun Microsystems (now Oracle Corporation). In 2010, when Oracle acquired Sun, Widenius forked the open-source MySQL project to create MariaDB.

MySQL has stand-alone clients that allow users to interact directly with a MySQL database using SQL, but more often, MySQL is used with other programs to implement applications that need relational database capability. MySQL is a component of the LAMP web application software stack (and others), which is an acronym for Linux, Apache, MySQL, Perl/PHP/Python. MySQL is used by many database-driven web applications, including Drupal, Joomla, phpBB, and WordPress. MySQL is also used by many popular websites, including Facebook, Flickr, MediaWiki, Twitter, and YouTube.

2. Mongodb:

mongoDB

MongoDB is an open source NoSQL database management program. NoSQL (Not only SQL) is used as an alternative to traditional relational databases. NoSQL databases are quite useful for working with large sets of distributed data. MongoDB is a tool that can manage document-oriented information, store or retrieve information.

MongoDB is used for high-volume data storage, helping organizations store large amounts of data while still performing rapidly. Organizations also use MongoDB for its ad-hoc queries, indexing, load balancing, aggregation, server-side JavaScript execution and other features.

Structured Query Language (SQL) is a standardized programming language that is used to manage relational databases. SQL normalizes data as schemas and tables, and every table has a fixed structure.

Instead of using tables and rows as in relational databases, as a NoSQL database, the MongoDB architecture is made up of collections and documents. Documents are made up of Key-value pairs -- MongoDB's basic unit of data. Collections, the equivalent of SQL tables, contain document sets. MongoDB offers support for many programming languages, such as C, C++, C#, Go, Java, Python, Ruby and Swift.

3. Cassandra:

Apache Cassandra is an open-source, NoSQL database designed to store data for applications that require fast read and write performance. For example, you can use Cassandra to store user profile information for online video games, device metadata for internet of things (IoT) applications, or records for events.

Cassandra is a nonrelational data store. You organize Cassandra data into keyspaces and tables. Keyspaces are collections of tables that you can use to represent table groupings, such as different system components, applications, or environments. Each table in Cassandra has a schema. Schemas define the columns within a table, and the data type of each column. Cassandra stores data within tables in rows. Each row in a table has a primary key which uniquely identifies the row within the table. Primary keys are composed of partition keys and optional clustering columns. Partition keys can be simple and made up of a single column, or rows can have compound partition keys composed from multiple columns. Rows also can have clustering columns. Cassandra uses clustering columns to sort rows within the same partition.

4. Redis:

What is Redis?

Redis (for REmote DIctionary Server) is an open source, in-memory, NoSQL key/value store that is used primarily as an application cache or quick-response database. Because it stores data in memory, rather than on a disk or solid-state drive (SSD), Redis delivers unparalleled speed, reliability, and performance.

When an application relies on external data sources, the latency and throughput of those sources can create a performance bottleneck, especially as traffic increases or the application scales. One way to improve performance in these cases is to store and manipulate data in-memory, physically closer to the application. Redis is built to this task: It stores all data in-memory—delivering the fastest possible performance when reading or writing data—and offers built-in replication capabilities that let you place data physically closer to the user for the lowest latency.

Other Redis characteristics worth noting include support for multiple data structures, built-in Lua scripting, multiple levels of on-disk persistence, and high availability.

5. Neo4j:

Fundamentally, graph databases store data in the form of graphs. A graphs is a mathematical concept that classifies elements in terms of vertices (nodes) and edges (relationships) to understand connections and patterns within the information being studied. When using a graph database like Neo4j, these graphs are often represented visually.

Graph databases are a relatively new class of database leveraged for use cases that are particularly focused on connectedness within data. In other words, while graph databases store data like nodes and edges, they focus more heavily on the relationships that are often hidden among the many elements within masses of data. In a graph database, relationships are first-class citizens along with data objects.

Generating Data Using Faker Library:

What is faker library:

The Faker library is a Python library used to generate fake data for a variety of purposes, such as testing, populating databases with dummy data, and creating realistic-looking sample data for data analysis or visualization tasks. It provides a wide range of methods to generate fake data for various data types, including names, addresses, dates, numbers, and more.

Here are some key features of the Faker library:

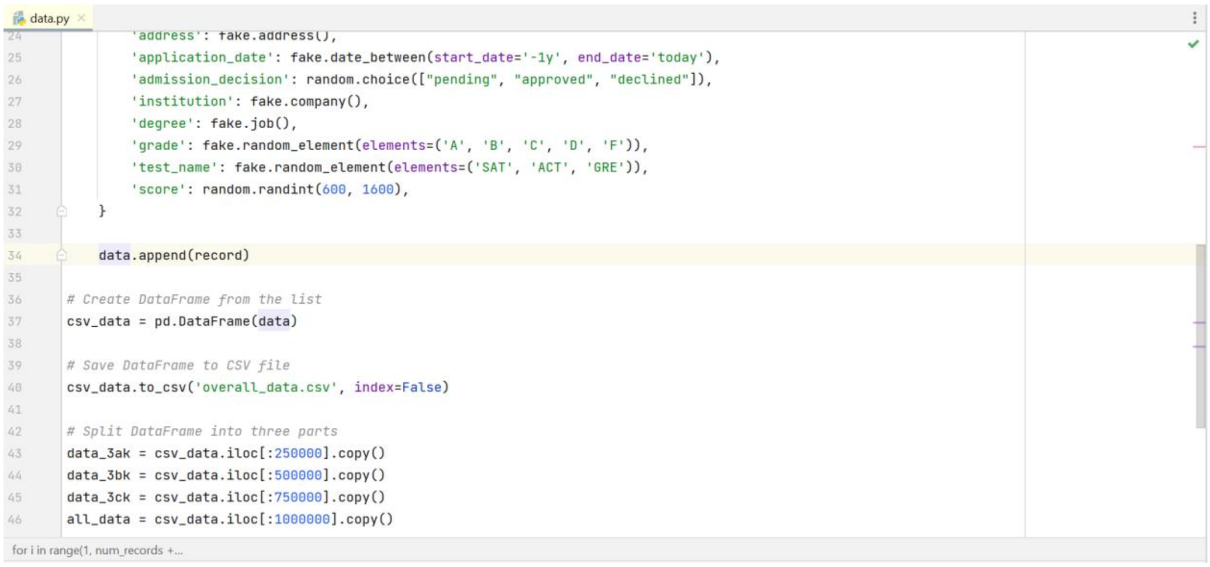
Localized Data: Faker supports multiple locales and can generate data that matches the language and cultural conventions of different regions.

Customization: Users can customize the generated data by specifying the format, pattern, or constraints for each data type.

Extensibility: Faker is highly extensible, allowing users to define their own custom data providers or extend existing ones.

Easy to Use: The library provides a simple and intuitive API, making it easy to generate fake data with just a few lines of code.

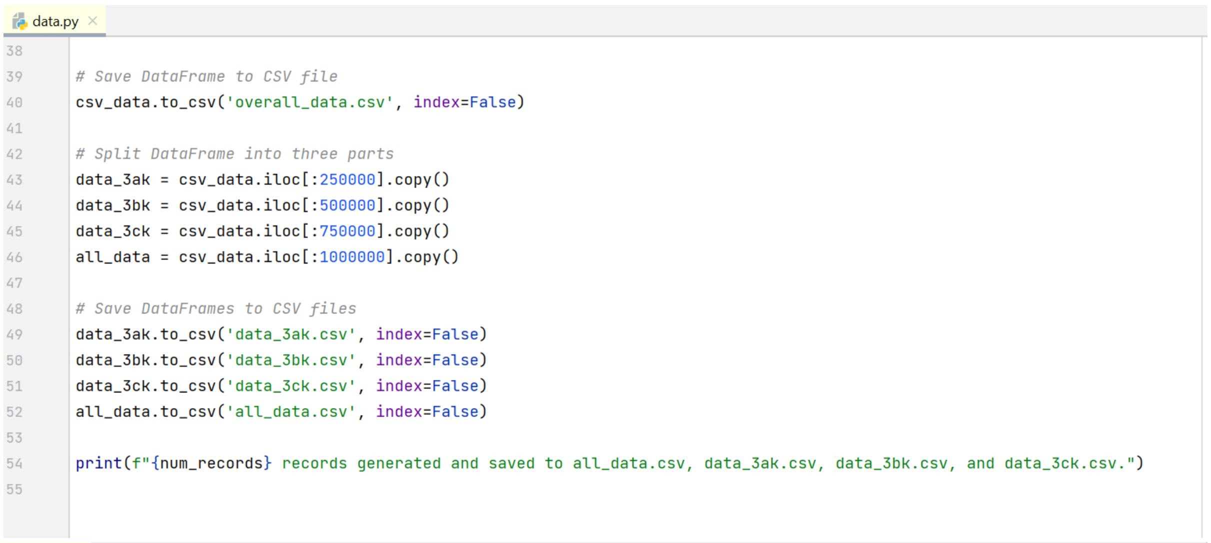
Overall, we needed the faker library in order to generate dataset consisting of 1 million data so to generate a big data with our custom column names faker library comes handy. The following is the code to generate 1 million data in 4 part: 250k, 500k, 750k and one million. The data is the online system admission data which consists of columns like: name, test, score, status of student, university, address and etc.



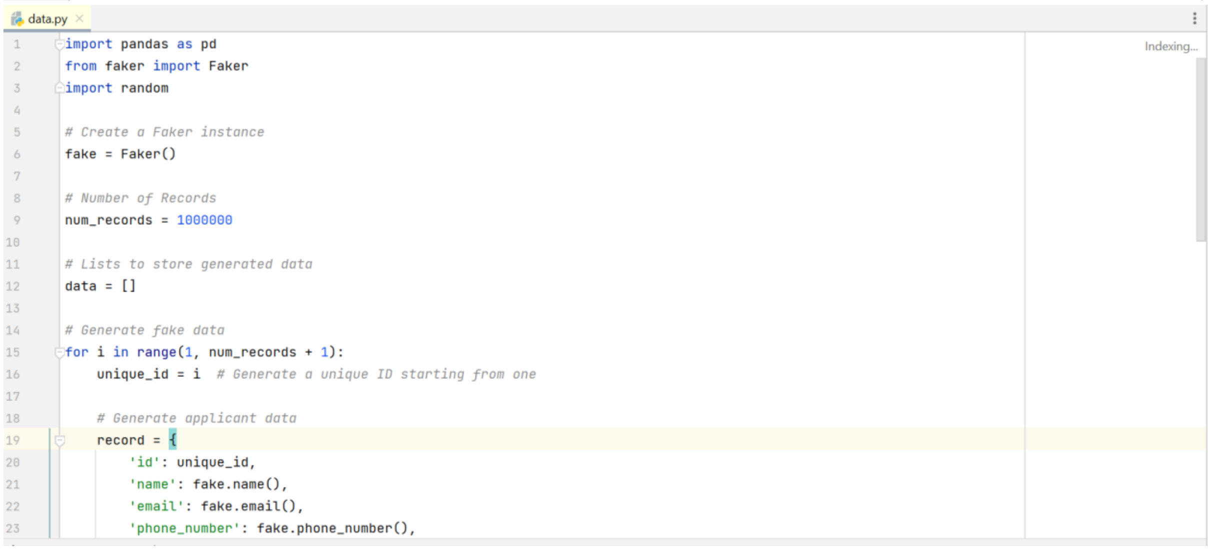
data.py

```
            'address': fake.address(),
            'application_date': fake.date_between(start_date='-1y', end_date='today'),
            'admission_decision': random.choice(["pending", "approved", "declined"]),
            'institution': fake.company(),
            'degree': fake.job(),
            'grade': fake.random_element(elements=('A', 'B', 'C', 'D', 'F')),
            'test_name': fake.random_element(elements=('SAT', 'ACT', 'GRE')),
            'score': random.randint(600, 1600),
        }

    data.append(record)

# Create DataFrame from the list
csv_data = pd.DataFrame(data)

# Save DataFrame to CSV file
csv_data.to_csv('overall_data.csv', index=False)

# Split DataFrame into three parts
data_3ak = csv_data.iloc[:250000].copy()
data_3bk = csv_data.iloc[:500000].copy()
data_3ck = csv_data.iloc[:750000].copy()
all_data = csv_data.iloc[:1000000].copy()
```

for i in range(1, num_records +...



```python

# Save DataFrame to CSV file
csv_data.to_csv('overall_data.csv', index=False)

# Split DataFrame into three parts
data_3ak = csv_data.iloc[:250000].copy()
data_3bk = csv_data.iloc[:500000].copy()
data_3ck = csv_data.iloc[:750000].copy()
all_data = csv_data.iloc[:1000000].copy()

# Save DataFrames to CSV files
data_3ak.to_csv('data_3ak.csv', index=False)
data_3bk.to_csv('data_3bk.csv', index=False)
data_3ck.to_csv('data_3ck.csv', index=False)
all_data.to_csv('all_data.csv', index=False)

print(f"{num_records} records generated and saved to all_data.csv, data_3ak.csv, data_3bk.csv, and data_3ck.csv.")

```



```python
import pandas as pd
from faker import Faker
import random

# Create a Faker instance
fake = Faker()

# Number of Records
num_records = 1000000

# Lists to store generated data
data = []

# Generate fake data
for i in range(1, num_records + 1):
    unique_id = i  # Generate a unique ID starting from one

    # Generate applicant data
    record = {
        'id': unique_id,
        'name': fake.name(),
        'email': fake.email(),
        'phone_number': fake.phone_number(),
```

The above are all the screenshots from the code that I used for generating fake data using fake library.

At first I imported the libraries then I specified the number of record that I want. Secondly, with faker methods I created 12 columns for my data as indicated in the pics. And lastly, I saved all the data in 4 separated csv file for the project that I used in the databases later.

Inserting data into libraries

I used different methods for different libraries as the following:

1. Mysql insertion:

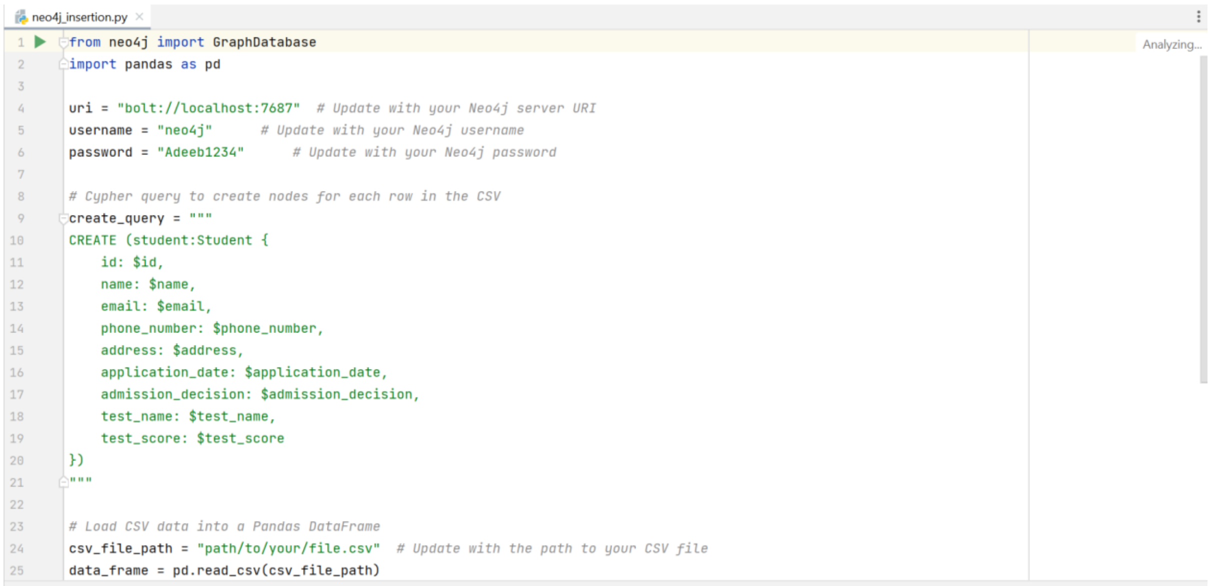
For the mysql I just used the graphical user interface in order to add all the data at first I changed the limit of the mysql from limited to unlimited then I imported my data which was fast and easy.

2. Cassandra:

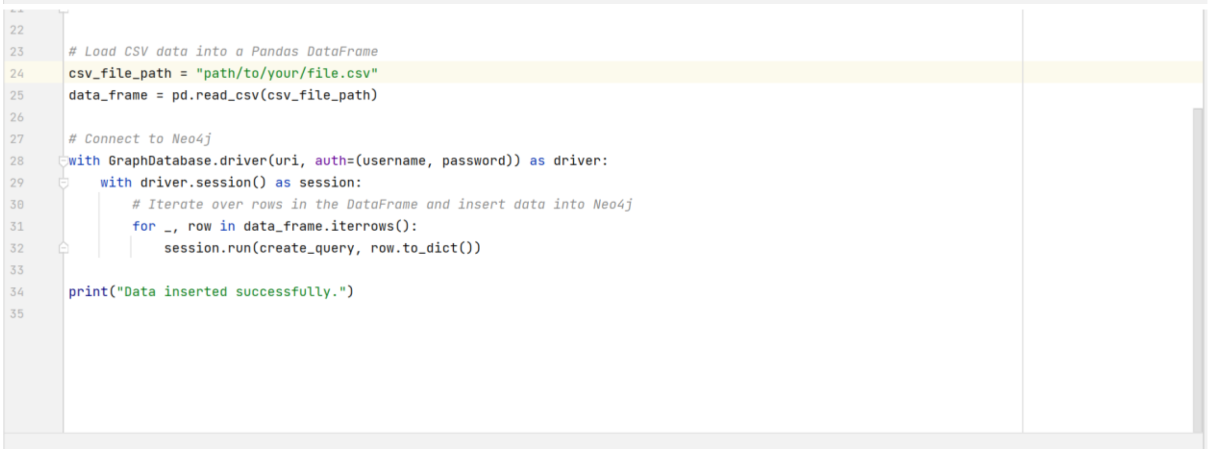
I have configured cassandra into dockers which is easy to use and I used DSbulk for the insertion which was an easy method for the cassandra. However, I tried inserting data with python but it needed approximately 1 day to insert 1 million data and its part separately but with dsbulk I inserted all the data in just 20 minutes.

3. Neo4j:

For the neo4j graph database I used python for the insertion which was an easy method and the codes are the following:



```python
from neo4j import GraphDatabase
import pandas as pd

uri = "bolt://localhost:7687"  # Update with your Neo4j server URI
username = "neo4j"      # Update with your Neo4j username
password = "Adeeb1234"      # Update with your Neo4j password

# Cypher query to create nodes for each row in the CSV
create_query = """
CREATE (student:Student {
    id: $id,
    name: $name,
    email: $email,
    phone_number: $phone_number,
    address: $address,
    application_date: $application_date,
    admission_decision: $admission_decision,
    test_name: $test_name,
    test_score: $test_score
})
"""

# Load CSV data into a Pandas DataFrame
csv_file_path = "path/to/your/file.csv"  # Update with the path to your CSV file
data_frame = pd.read_csv(csv_file_path)
```



```python

# Load CSV data into a Pandas DataFrame
csv_file_path = "path/to/your/file.csv"
data_frame = pd.read_csv(csv_file_path)

# Connect to Neo4j
with GraphDatabase.driver(uri, auth=(username, password)) as driver:
    with driver.session() as session:
        # Iterate over rows in the DataFrame and insert data into Neo4j
        for _, row in data_frame.iterrows():
            session.run(create_query, row.to_dict())

print("Data inserted successfully.")
```

This Python code connects to a Neo4j database and inserts data from a CSV file into the database. Here's a summary of the code:

Neo4j Connection Information:

uri: The URI of the Neo4j server, typically in the format "bolt://localhost:7687".

username: The username used to authenticate with the Neo4j server.

password: The password used to authenticate with the Neo4j server.

Cypher Query to Create Nodes:

The create_query variable contains a Cypher query string that defines how nodes will be created in the Neo4j database. It uses parameters to insert values from the CSV file into the node properties.

Load CSV Data into Pandas DataFrame:

The csv_file_path variable holds the path to the CSV file containing the data to be inserted into Neo4j.

The pd.read_csv() function from the Pandas library reads the CSV file into a DataFrame named data_frame.

Connect to Neo4j:

The GraphDatabase.driver() function from the neo4j library connects to the Neo4j database using the provided URI, username, and password.

The driver.session() function creates a session object for interacting with the database.

Insert Data into Neo4j:

Inside a with block, the code iterates over each row in the DataFrame using data_frame.iterrows().

For each row, the row.to_dict() method converts the row into a dictionary where the keys are column names and the values are the corresponding values from the row.

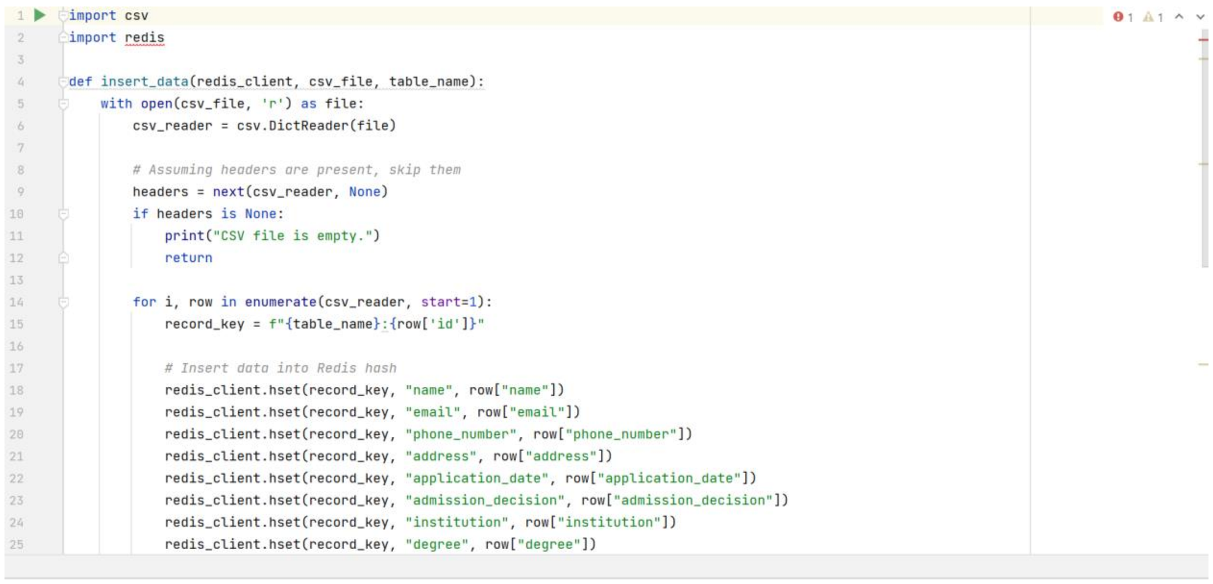
The session.run() method executes the Cypher query (create_query) with the row data as parameters, inserting a node into the Neo4j database for each row in the CSV.

Confirmation Message:

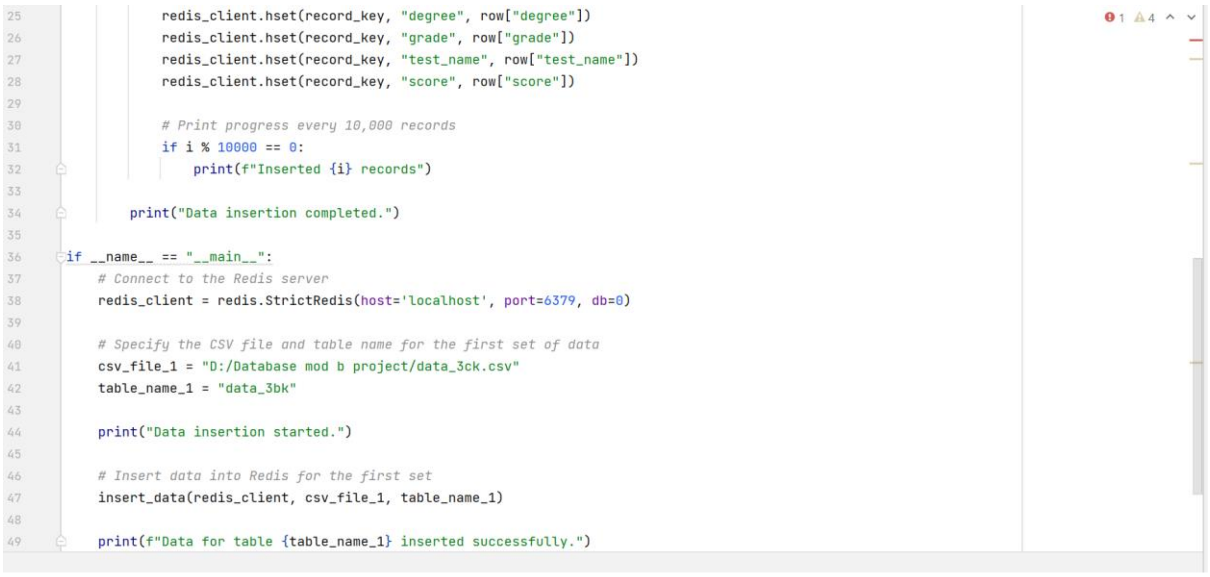
After the data insertion process completes successfully, the code prints a message indicating that the data was inserted successfully.

Overall, this code demonstrates how to connect to a Neo4j database, load data from a CSV file into a Pandas DataFrame, and insert the data into the database as nodes using Cypher queries.

4. Redis:



```python
import csv
import redis

def insert_data(redis_client, csv_file, table_name):
    with open(csv_file, 'r') as file:
        csv_reader = csv.DictReader(file)

        # Assuming headers are present, skip them
        headers = next(csv_reader, None)
        if headers is None:
            print("CSV file is empty.")
            return

        for i, row in enumerate(csv_reader, start=1):
            record_key = f"{table_name}:{row['id']}"

            # Insert data into Redis hash
            redis_client.hset(record_key, "name", row["name"])
            redis_client.hset(record_key, "email", row["email"])
            redis_client.hset(record_key, "phone_number", row["phone_number"])
            redis_client.hset(record_key, "address", row["address"])
            redis_client.hset(record_key, "application_date", row["application_date"])
            redis_client.hset(record_key, "admission_decision", row["admission_decision"])
            redis_client.hset(record_key, "institution", row["institution"])
            redis_client.hset(record_key, "degree", row["degree"])
```



```python
            redis_client.hset(record_key, "degree", row["degree"])
            redis_client.hset(record_key, "grade", row["grade"])
            redis_client.hset(record_key, "test_name", row["test_name"])
            redis_client.hset(record_key, "score", row["score"])

            # Print progress every 10,000 records
            if i % 10000 == 0:
                print(f"Inserted {i} records")

        print("Data insertion completed.")

if __name__ == "__main__":
    # Connect to the Redis server
    redis_client = redis.StrictRedis(host='localhost', port=6379, db=0)

    # Specify the CSV file and table name for the first set of data
    csv_file_1 = "D:/Database mod b project/data_3ck.csv"
    table_name_1 = "data_3bk"

    print("Data insertion started.")

    # Insert data into Redis for the first set
    insert_data(redis_client, csv_file_1, table_name_1)

    print(f"Data for table {table_name_1} inserted successfully.")
```

This Python script inserts data from a CSV file into a Redis database. Key points include:

Function Definition (insert_data):

- Defines a function to insert CSV data into Redis.
- Takes parameters for the Redis client, CSV file path, and table name.
- Reads CSV rows as dictionaries and inserts each row into Redis as a hash.

Main Function:

- Connects to Redis using the Redis client.
- Specifies CSV file path and Redis table name.
- Calls insert_data to insert data into Redis.
- Prints progress messages during data insertion.
- Data Insertion Process:

- Reads CSV rows as dictionaries.
- Constructs Redis keys using table name and "id" column value.
- Inserts each row's data into Redis as a hash using constructed keys.
- Prints progress messages every 10,000 records inserted.

Redis Connection:

- Establishes a connection to the Redis server.
- Uses the specified host, port, and database number.
- Data Insertion:

- CSV data is inserted into Redis as hash structures.
- Each row in the CSV becomes a hash entry in Redis.
- Keys and values in Redis correspond to CSV headers and row values.

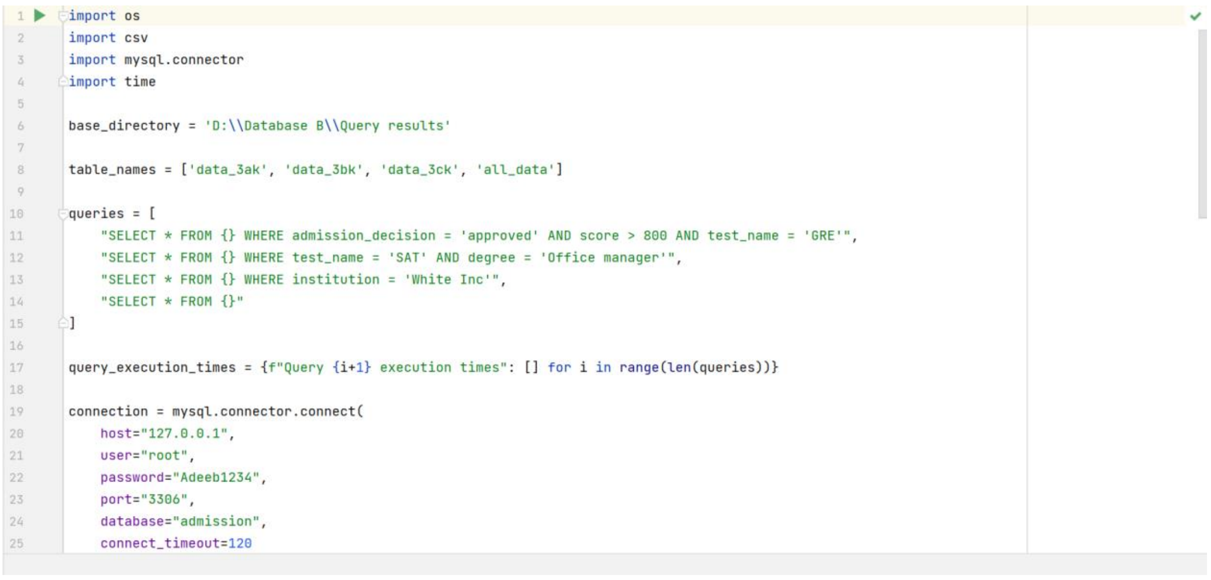
Overall, the script efficiently transfers structured data from a CSV file to a Redis database, enabling easy storage and retrieval of information.

5. Mongodb:

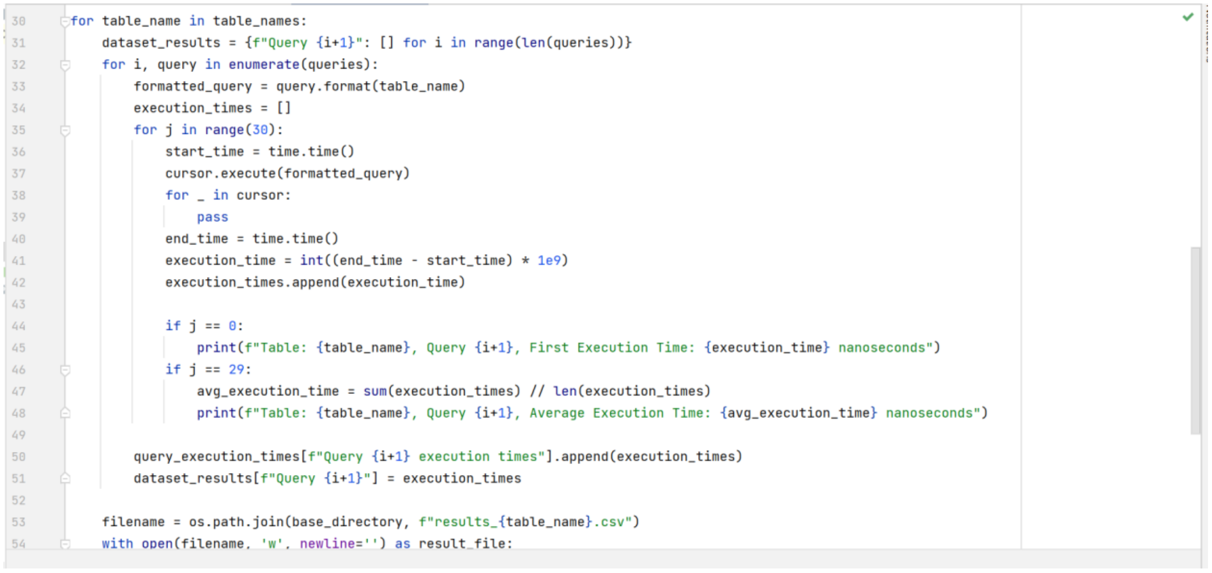
I used the graphical user interface for the mongodb which was fast and easy method. However, I tried inserting all the data through the python but it took long time but graphically with its desktop application it took just small amount of time.

Queries for the Databases

1. Mysql:



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

base_directory = 'D:\\Database B\\Query results'

table_names = ['data_3ak', 'data_3bk', 'data_3ck', 'all_data']

queries = [
    "SELECT * FROM {} WHERE admission_decision = 'approved' AND score > 800 AND test_name = 'GRE'",
    "SELECT * FROM {} WHERE test_name = 'SAT' AND degree = 'Office manager'",
    "SELECT * FROM {} WHERE institution = 'White Inc'",
    "SELECT * FROM {}"
]

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

connection = mysql.connector.connect(
    host="127.0.0.1",
    user="root",
    password="Adeeb1234",
    port="3306",
    database="admission",
    connect_timeout=120
```



```python
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:
```

```python
    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()
```

This Python script executes SQL queries against a MySQL database and measures their execution times. Key points include:

Setup:

- Connects to a MySQL database using mysql.connector.
- Defines base directory, table names, and SQL queries to execute.

Execution Loop:

- Iterates over each table and query combination.
- Executes each query multiple times (30 times) to get accurate execution times.
- Measures execution times using time.time() before and after query execution.
- Calculates the average execution time for each query.

Data Storage:

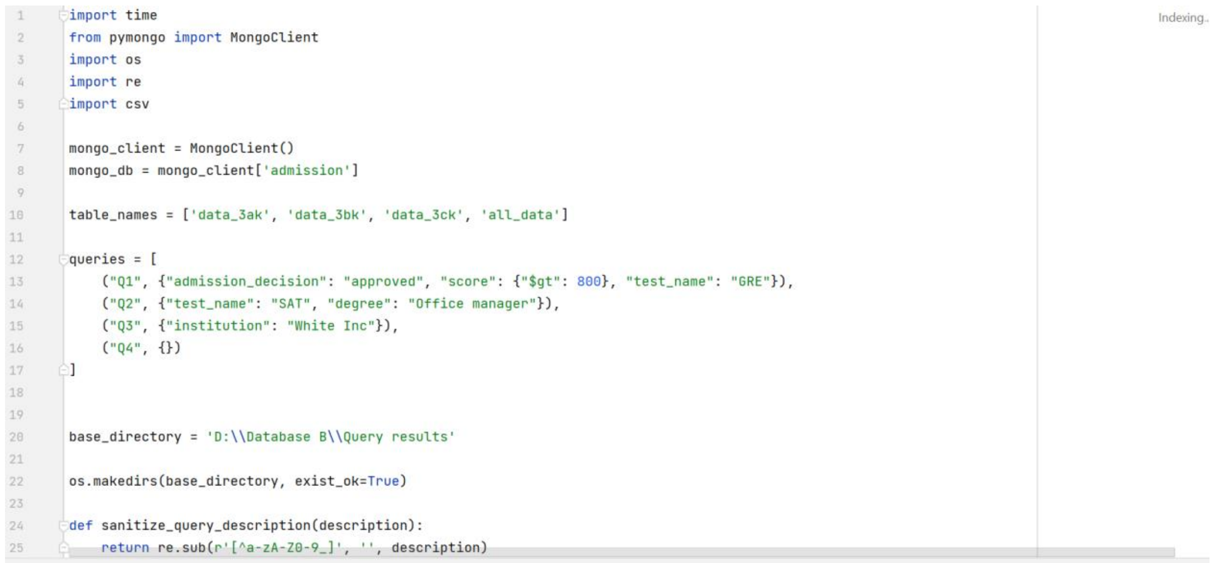
- Stores the results in a dictionary (query_execution_times) for later analysis.
- Writes the results to CSV files named according to the table name and query.

Cleanup:

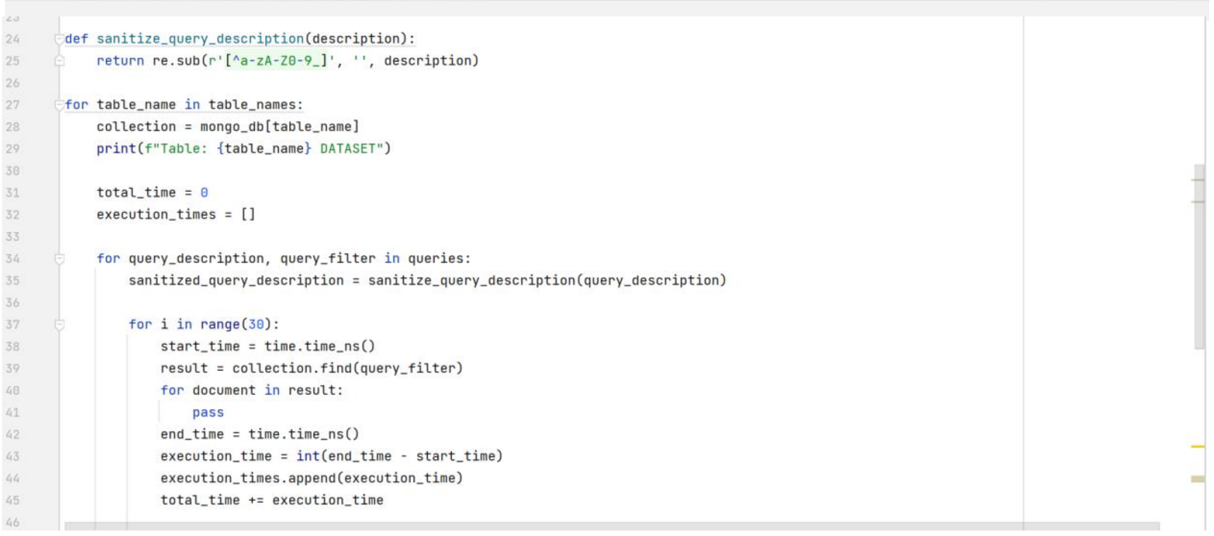
- Closes the cursor and database connection after executing all queries.

Overall, the script efficiently benchmarks the performance of SQL queries against the MySQL database and stores the results for further analysis.

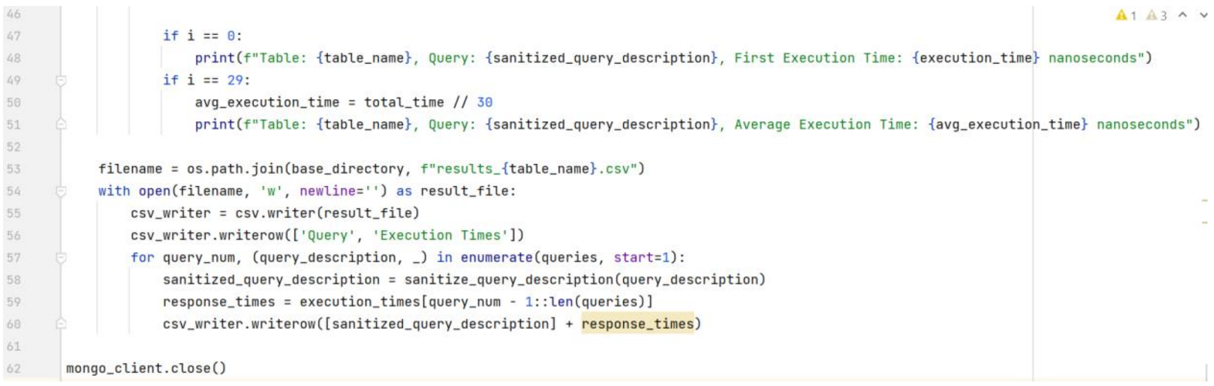
2. Mongodb:



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

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

table_names = ['data_3ak', 'data_3bk', 'data_3ck', 'all_data']

queries = [
    ("Q1", {"admission_decision": "approved", "score": {"$gt": 800}, "test_name": "GRE"}),
    ("Q2", {"test_name": "SAT", "degree": "Office manager"}),
    ("Q3", {"institution": "White Inc"}),
    ("Q4", {})
]


base_directory = 'D:\\Database B\\Query results'

os.makedirs(base_directory, exist_ok=True)

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



```python
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
```



```python

            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()
```

This Python script benchmarks MongoDB queries against different collections within the "admission" database. Key points include:

Setup:

- Imports necessary libraries (time, pymongo) and initializes a MongoDB client.
- Defines the MongoDB database (mongo_db) and table names (table_names).
- Specifies the queries to execute (queries), each with a descriptive label and corresponding filter.

Execution Loop:

- Iterates over each table and query combination.
- Executes each query multiple times (30 times) to measure execution times accurately.
- Measures execution times using time.time_ns() before and after query execution.

Data Storage:

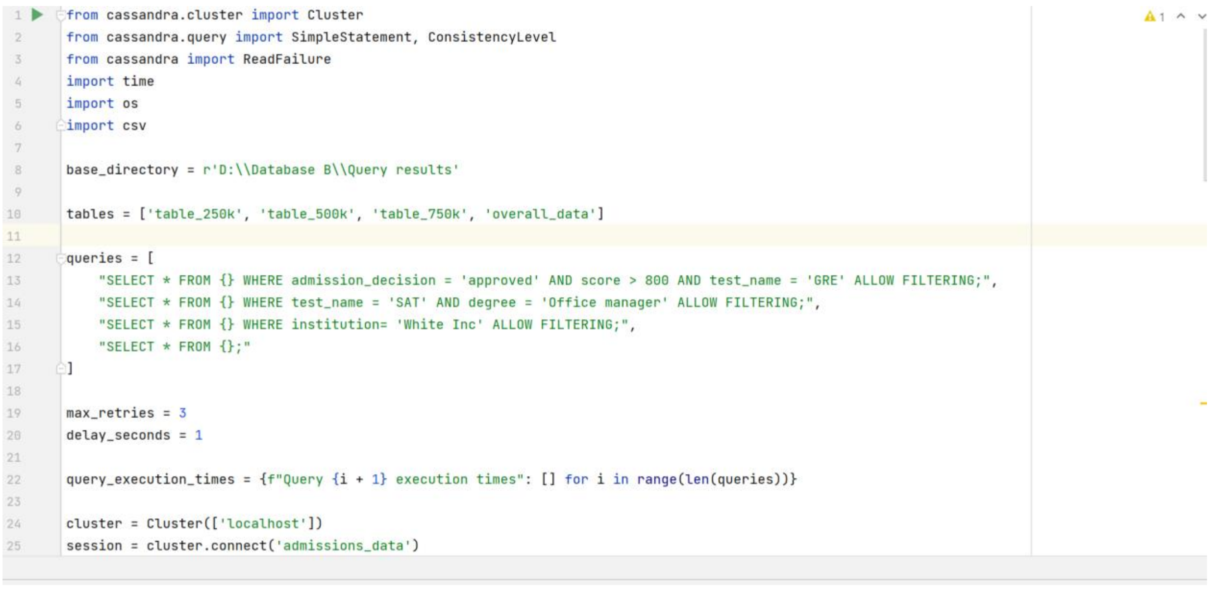
- Stores the results in a CSV file named according to the table name.
- Writes the execution times for each query to the CSV file.

Cleanup:

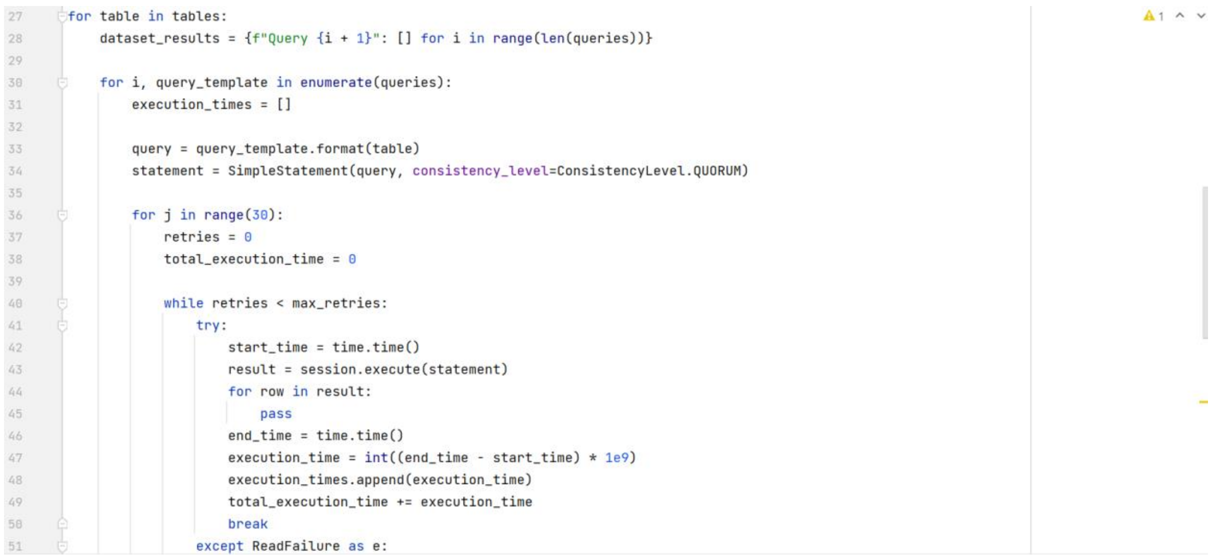
- Closes the MongoDB client after executing all queries.

Overall, the script efficiently benchmarks the performance of MongoDB queries against different collections and stores the results for further analysis.

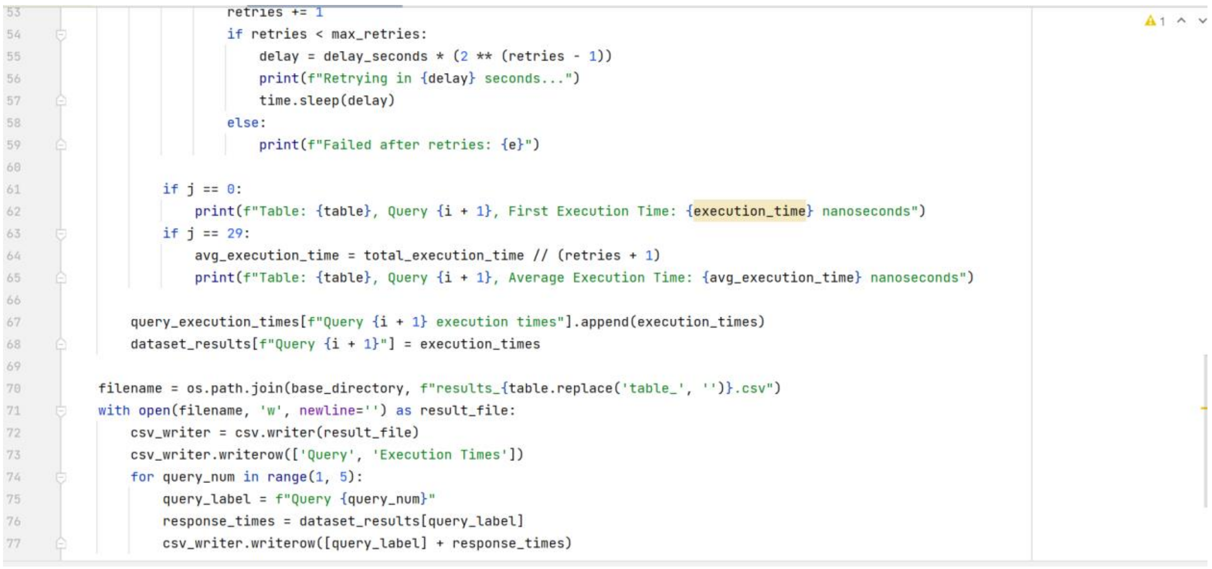
3. Cassandra:



```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'D:\\Database B\\Query results'

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

queries = [
    "SELECT * FROM {} WHERE admission_decision = 'approved' AND score > 800 AND test_name = 'GRE' ALLOW FILTERING;",
    "SELECT * FROM {} WHERE test_name = 'SAT' AND degree = 'Office manager' ALLOW FILTERING;",
    "SELECT * FROM {} WHERE institution= 'White Inc' 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('admissions_data')
```



```python
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:
```



```
                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)
```

This Python script benchmarks Cassandra queries against different tables in the "admissions_data" keyspace. Key points include:

Setup:

- Imports necessary libraries (Cluster, SimpleStatement, ConsistencyLevel, ReadFailure, time, os, csv).
- Defines the base directory for query results (base_directory), table names (tables), and queries to execute (queries).

Execution Loop:

- Establishes a connection to the Cassandra cluster and session.
- Iterates over each table and query combination.
- Executes each query multiple times (30 times) to measure execution times accurately.
- Uses exponential backoff for retrying failed queries (max_retries, delay_seconds) to handle read failures.

Data Storage:

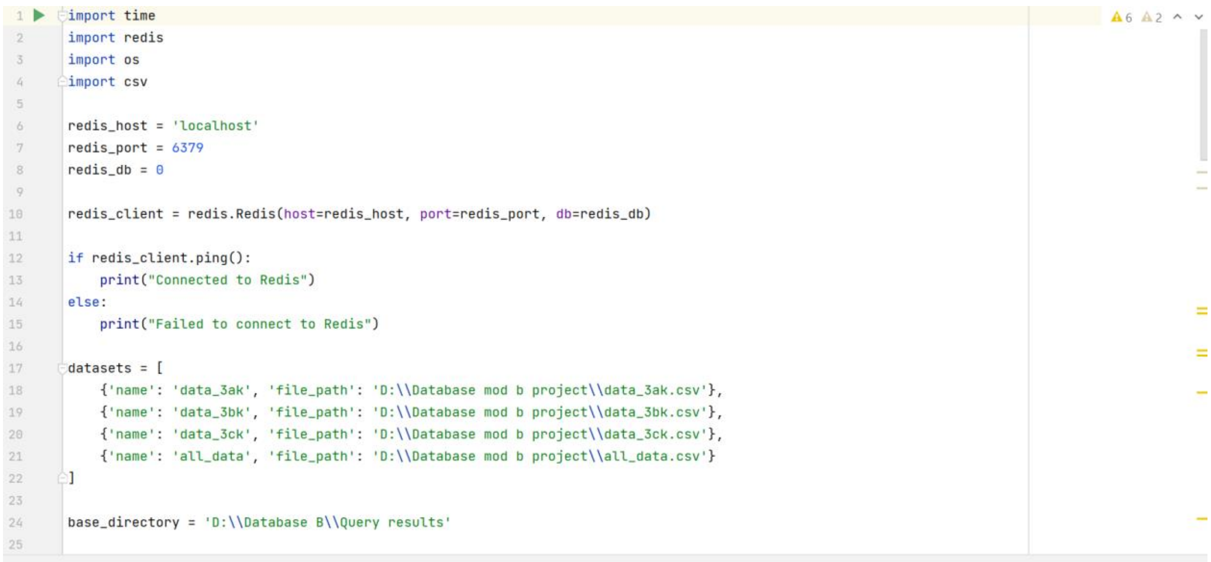
- Stores the results in a CSV file named according to the table name, with execution times for each query.

Cleanup:

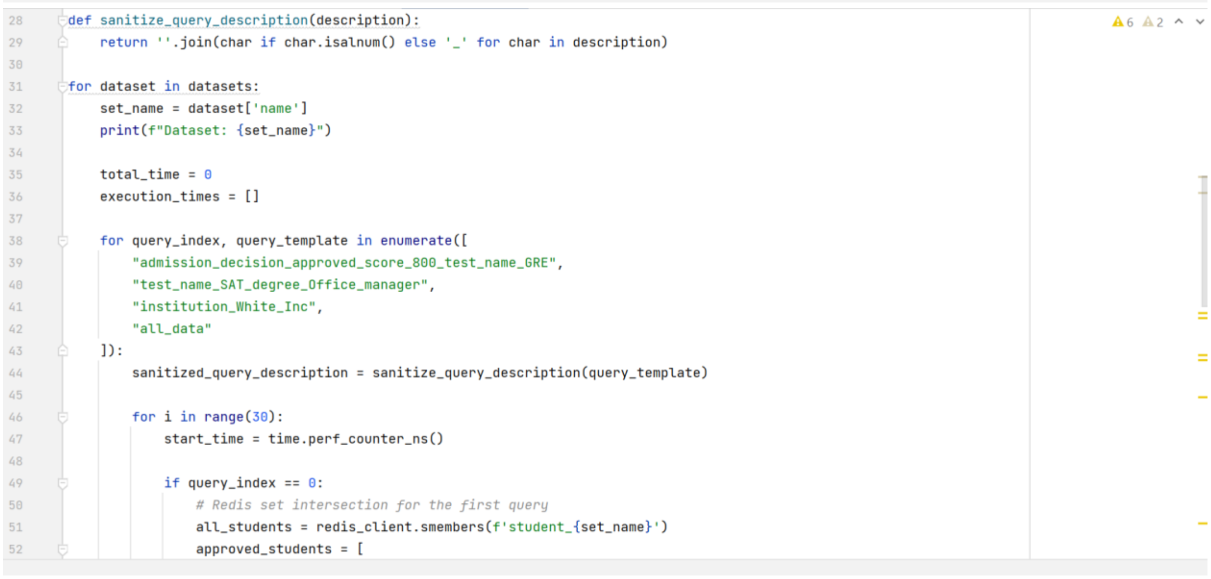
- Shuts down the session and cluster after executing all queries.

Overall, the script effectively benchmarks the performance of Cassandra queries against different tables and stores the results for further analysis. It also implements retry logic to handle transient failures gracefully.

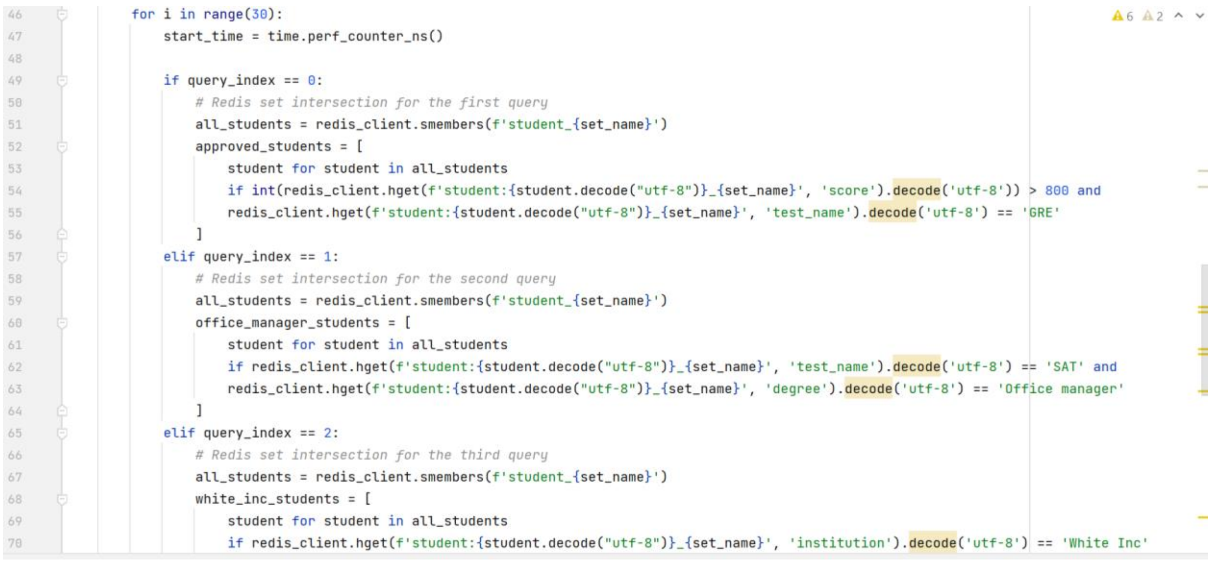
4. Redis:



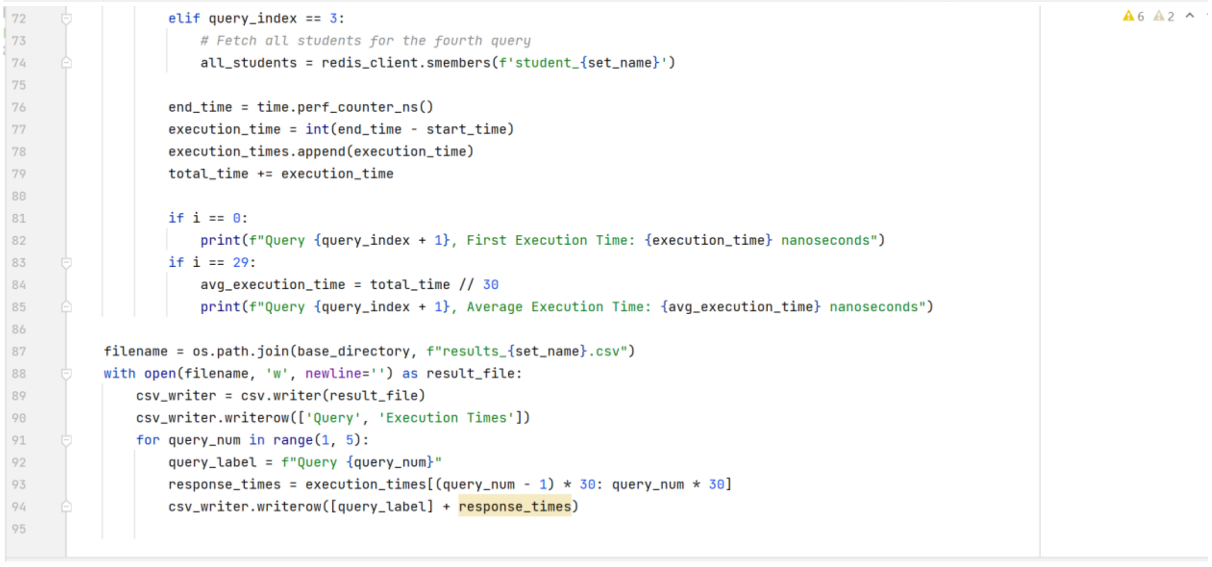
```python
import time
import redis
import os
import csv

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': 'data_3ak', 'file_path': 'D:\\Database mod b project\\data_3ak.csv'},
    {'name': 'data_3bk', 'file_path': 'D:\\Database mod b project\\data_3bk.csv'},
    {'name': 'data_3ck', 'file_path': 'D:\\Database mod b project\\data_3ck.csv'},
    {'name': 'all_data', 'file_path': 'D:\\Database mod b project\\all_data.csv'}
]

base_directory = 'D:\\Database B\\Query results'

```



```python
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_template in enumerate([
        "admission_decision_approved_score_800_test_name_GRE",
        "test_name_SAT_degree_Office_manager",
        "institution_White_Inc",
        "all_data"
    ]):
        sanitized_query_description = sanitize_query_description(query_template)

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

            if query_index == 0:
                # Redis set intersection for the first query
                all_students = redis_client.smembers(f'student_{set_name}')
                approved_students = [
```



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

            if query_index == 0:
                # Redis set intersection for the first query
                all_students = redis_client.smembers(f'student_{set_name}')
                approved_students = [
                    student for student in all_students
                    if int(redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'score').decode('utf-8')) > 800 and
                    redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'test_name').decode('utf-8') == 'GRE'
                ]
            elif query_index == 1:
                # Redis set intersection for the second query
                all_students = redis_client.smembers(f'student_{set_name}')
                office_manager_students = [
                    student for student in all_students
                    if redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'test_name').decode('utf-8') == 'SAT' and
                    redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'degree').decode('utf-8') == 'Office manager'
                ]
            elif query_index == 2:
                # Redis set intersection for the third query
                all_students = redis_client.smembers(f'student_{set_name}')
                white_inc_students = [
                    student for student in all_students
                    if redis_client.hget(f'student:{student.decode("utf-8")}_{set_name}', 'institution').decode('utf-8') == 'White Inc'
```



```python
            elif query_index == 3:
                # Fetch all students for the fourth query
                all_students = redis_client.smembers(f'student_{set_name}')

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

            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)
```

This Python script performs benchmarking of Redis queries against different datasets. Here's a summary:

Redis Connection Setup:

Establishes a connection to the Redis server using the redis library, with host, port, and database specified.

Dataset and Query Definitions:

- Defines a list of datasets, each containing a name and file path.
- Defines four queries, each corresponding to a specific dataset operation.

Execution Loop:

- Iterates over each dataset, performing operations and measuring execution times for each query.

For each dataset:

- Iterates over each query, executing it 30 times for accurate time measurements.
- Calculates and prints the first execution time and average execution time for each query.
- Stores execution times in a list for each query.
- Writes results to CSV files named according to the dataset.

Query Execution:

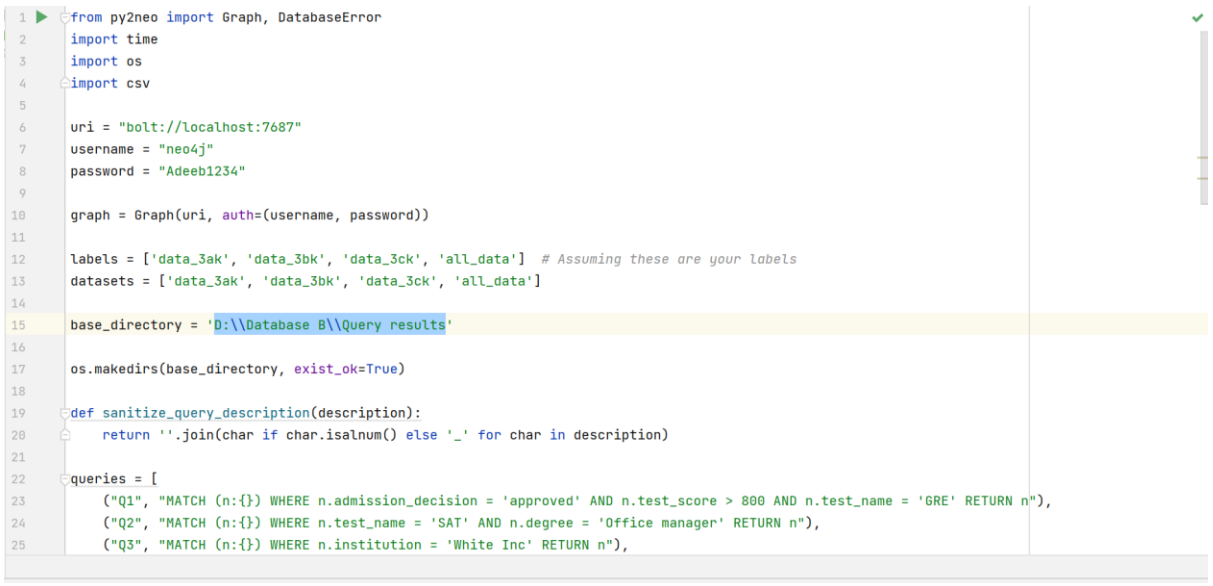
- Executes different Redis set operations for each query
- Set intersection to filter students based on conditions specified in the queries.
- Fetches all students for the fourth query.

Data Storage:

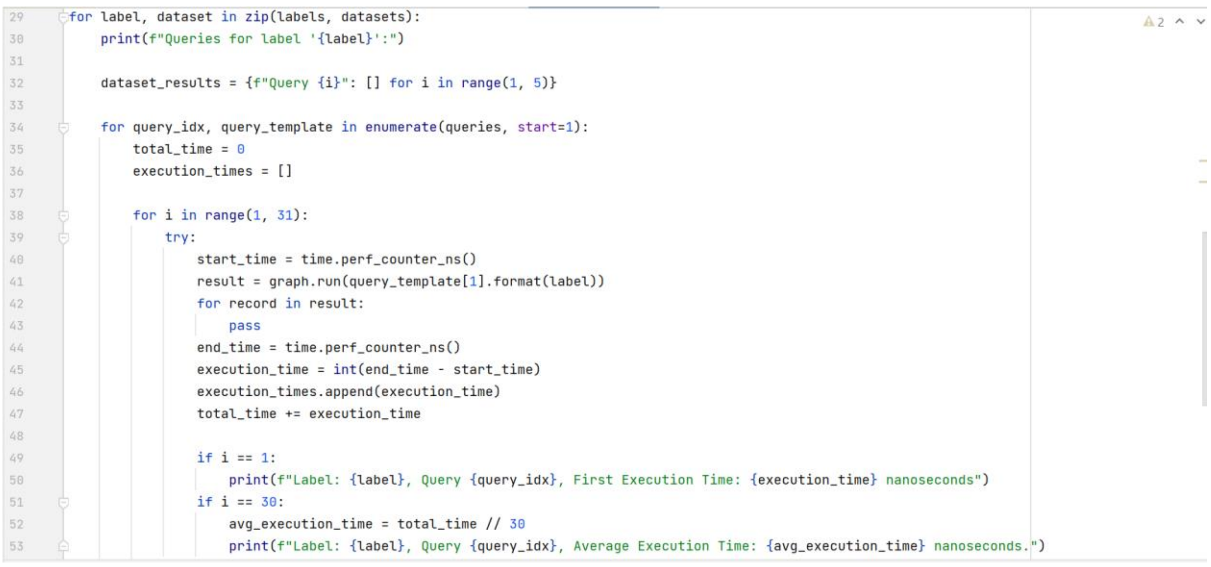
- Stores the results in CSV files named after each dataset, with execution times for each query.

Overall, the script effectively benchmarks Redis queries against various datasets and stores the results for further analysis. It leverages Redis set operations to perform filtering based on specified criteria in the queries.

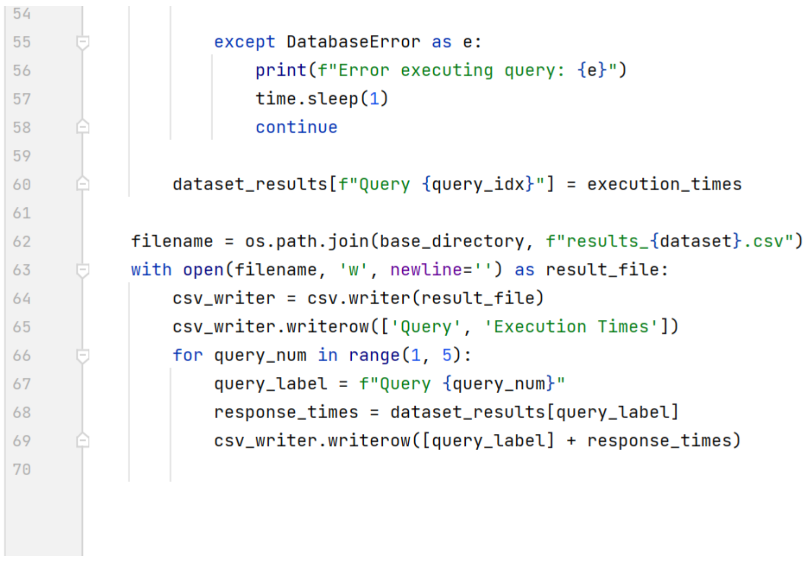
5. Neo4j:



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

uri = "bolt://localhost:7687"
username = "neo4j"
password = "Adeeb1234"

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

labels = ['data_3ak', 'data_3bk', 'data_3ck', 'all_data']  # Assuming these are your labels
datasets = ['data_3ak', 'data_3bk', 'data_3ck', 'all_data']

base_directory = 'D:\\Database B\\Query results'

os.makedirs(base_directory, exist_ok=True)

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

queries = [
    ("Q1", "MATCH (n:{}) WHERE n.admission_decision = 'approved' AND n.test_score > 800 AND n.test_name = 'GRE' RETURN n"),
    ("Q2", "MATCH (n:{}) WHERE n.test_name = 'SAT' AND n.degree = 'Office manager' RETURN n"),
    ("Q3", "MATCH (n:{}) WHERE n.institution = 'White Inc' RETURN n"),
```



```python
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[1].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)
```

This Python script benchmarks Neo4j queries using the py2neo library against different datasets. Here's a summary:

Neo4j Connection Setup:

- Establishes a connection to the Neo4j database using the Bolt protocol with authentication.

Dataset and Query Definitions:

- Defines a list of labels representing different datasets.
- Defines a list of queries, each containing a query identifier and the Cypher query template.

Execution Loop:

- Iterates over each dataset and its corresponding label.

For each label:

- Iterates over each query, executing it 30 times for accurate time measurements.
- Calculates and prints the first execution time and average execution time for each query.
- Stores execution times in a dictionary for each query.

Query Execution:

- Executes Cypher queries against the Neo4j database using the provided templates.
- Measures execution times using the time.perf_counter_ns() function.

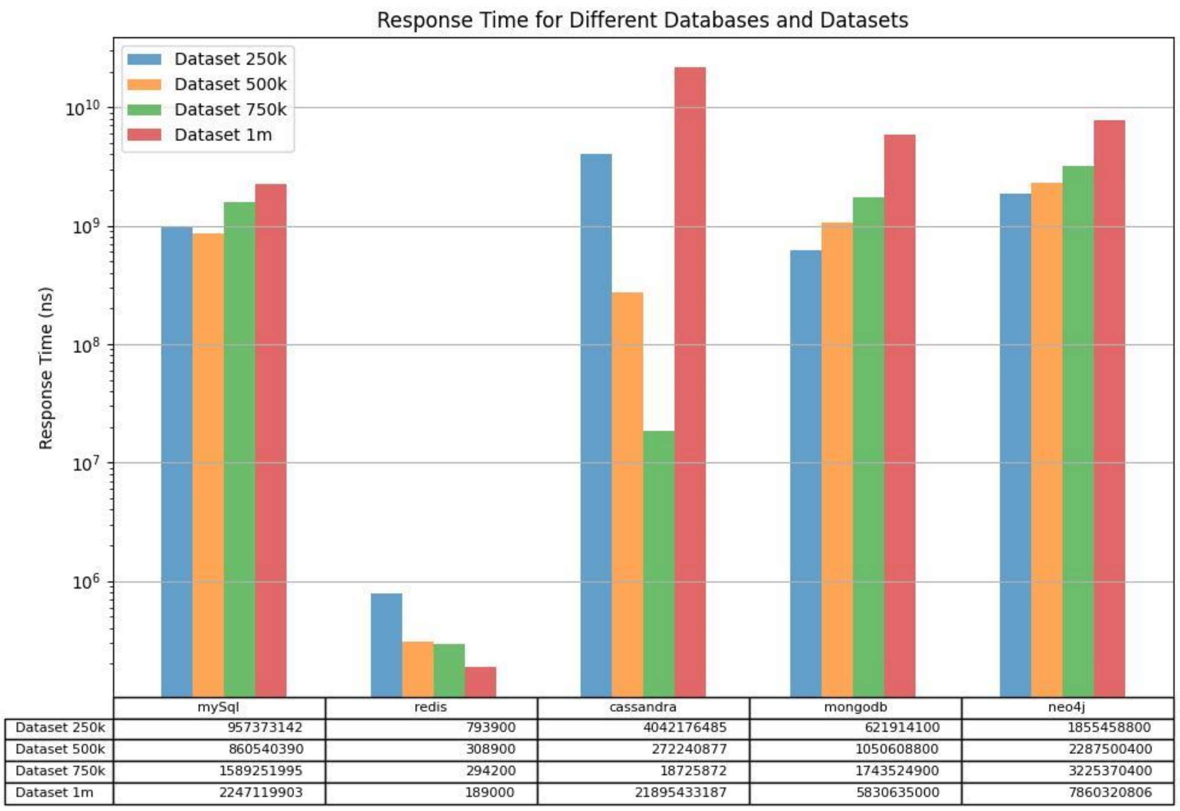
Data Storage:

- Writes the results to CSV files named after each dataset, with execution times for each query.

Overall, the script effectively benchmarks Neo4j queries against various datasets and stores the results for further analysis. It leverages the py2neo library to interact with the Neo4j database and measures execution times using Python's timing functionalities.

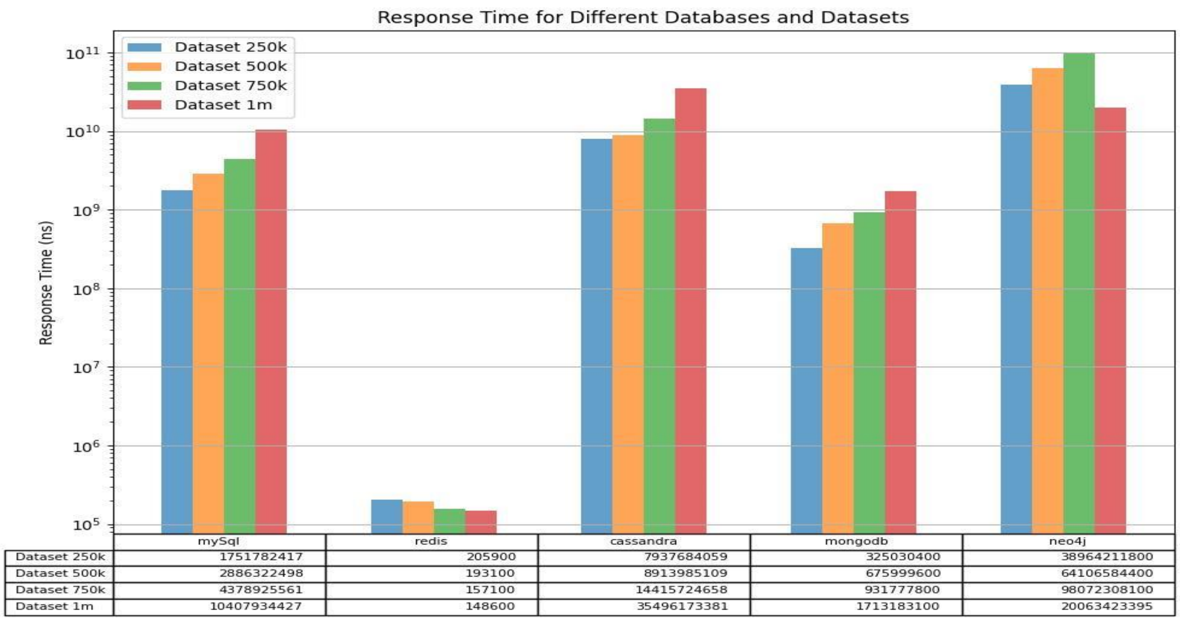
Histogram

First query:retriving the record of students who have been approved and they have more than 800 score in GRE.



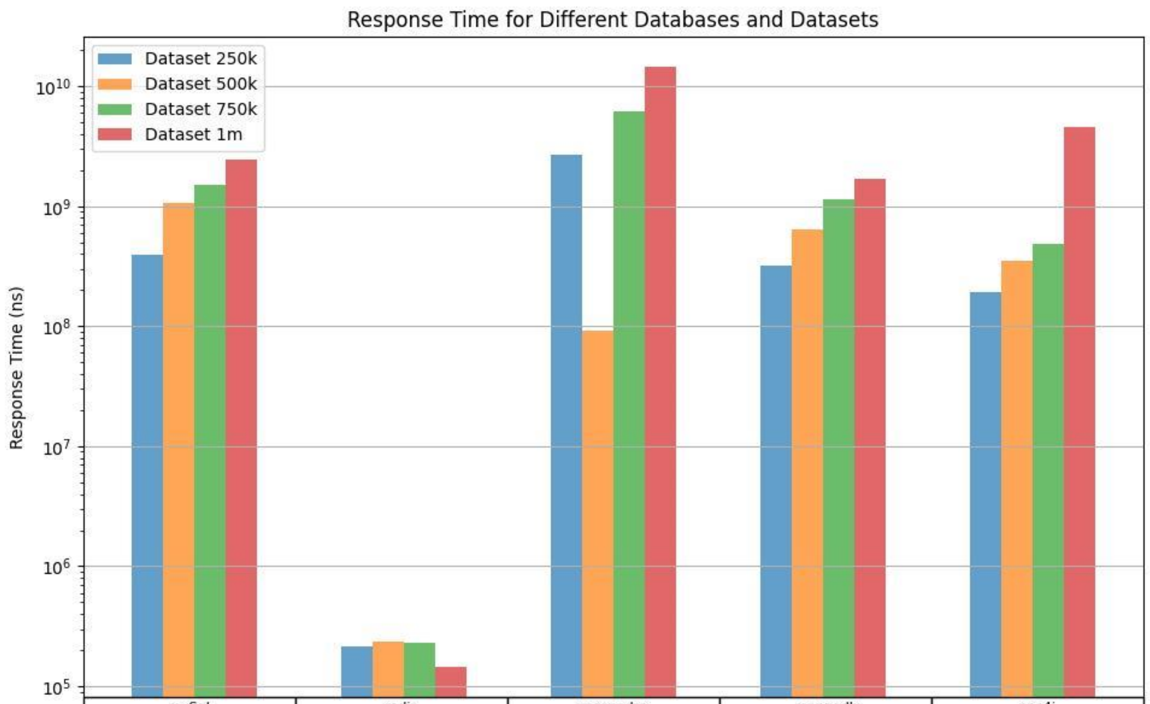
	mySql	redis	cassandra	mongodb	neo4j
Dataset 250k	957373142	793900	4042176485	621914100	1855458800
Dataset 500k	860540390	308900	272240877	1050608800	2287500400
Dataset 750k	1589251995	294200	18725872	1743524900	3225370400
Dataset 1m	2247119903	189000	21895433187	5830635000	7860320806

Second query: retriving the list of students who have degree in managing office and they have pased the test ‘SAT’.



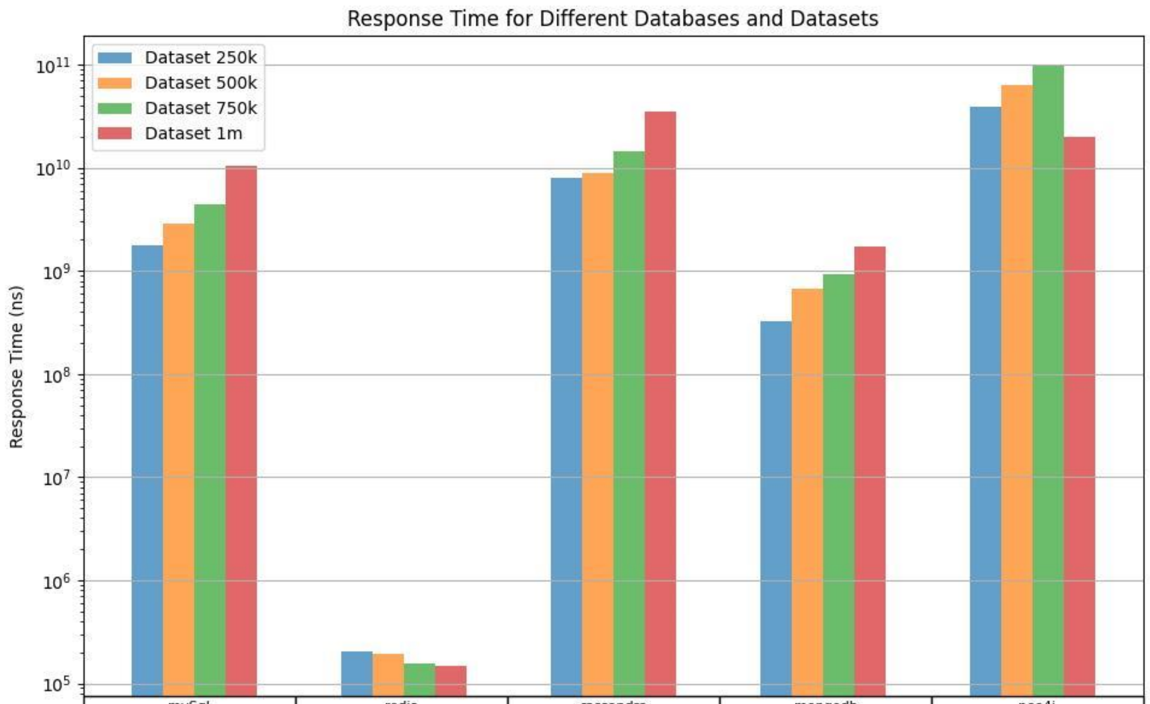
	mySql	redis	cassandra	mongodb	neo4j
Dataset 250k	1751782417	205900	7937684059	325030400	38964211800
Dataset 500k	2886322498	193100	8913985109	675999600	64106584400
Dataset 750k	4378925561	157100	14415724658	931777800	98072308100
Dataset 1m	10407934427	148600	35496173381	1713183100	20063423395

Third query: retriving the list of students who have selected the White Inc institution.



	mySql	redis	cassandra	mongodb	neo4j
Dataset 250k	394734144	213500	2719006538	319297100	192155000
Dataset 500k	1056546449	236900	92654228	636006200	353982100
Dataset 750k	1508355617	230300	6113925552	1147060100	488106000
Dataset 1m	2459102630	145500	14621105194	1678146200	4621105194

Fourth query: retriving the list of all students.



	mySql	redis	cassandra	mongodb	neo4j
Dataset 250k	1751782417	205900	7937684059	325030400	38964211800
Dataset 500k	2886322498	193100	8913985109	675999600	64106584400
Dataset 750k	4378925561	157100	14415724658	931777800	98072308100
Dataset 1m	10407934427	148600	35496173381	1713183100	20063423395

Conclusion

The project involves benchmarking different database systems using various datasets and query scenarios. Here are the key points and observations:

Database Systems:

- You have evaluated multiple database systems, including MySQL, Redis, Cassandra, MongoDB, and Neo4j.
- Each database system offers different strengths and is suitable for specific use cases. For example:
- MySQL is a relational database well-suited for structured data and complex queries.
- Redis is an in-memory data store known for its high-performance caching and data structure support.
- Cassandra is a distributed NoSQL database optimized for high availability and scalability.
- MongoDB is a document-oriented database that provides flexibility and scalability for unstructured data.
- Neo4j is a graph database designed for managing and querying highly connected data.

Benchmarking Methodology:

- You have developed benchmarking scripts for each database system to measure query performance under various conditions.
- The scripts include functionalities to execute queries, measure execution times, and store results for analysis.
- Each benchmarking script is tailored to the specific characteristics and query languages of the corresponding database system.

Query Scenarios:

- You have defined different query scenarios to assess the performance of each database system.
- These scenarios include filtering data based on specific criteria, performing aggregations, and retrieving results from different datasets.
- By testing a diverse range of queries, you can evaluate how each database system handles various workload types.

Data Analysis:

- The benchmarking scripts generate output files containing execution times for each query and dataset combination.
- You can analyze these results to compare the performance of different database systems across various scenarios.
- Key metrics such as average execution time, throughput, and resource utilization can provide insights into the strengths and limitations of each database system.

Future Considerations:

- You may further expand your benchmarking efforts by including additional database systems or optimizing query performance.
- Fine-tuning database configurations, indexing strategies, and hardware resources can potentially enhance performance and scalability.
- Continuously monitoring and benchmarking database performance ensures that your system meets evolving requirements and scales effectively.

In conclusion, your project provides valuable insights into the performance characteristics of different database systems, enabling informed decision-making for selecting the most suitable database technology based on specific use cases and requirements.

References

1. https://en.wikipedia.org/wiki/Neo4j

2. https://www.techtarget.com/searchdatamanagement/definition/MongoDB

3. https://en.wikipedia.org/wiki/MySQL

4. https://aws.amazon.com/keyspaces/what-is-cassandra/

5. https://www.ibm.com/topics/redis