DETAILED PROJECT REPORT OF DATABASE MOD B

EVENT MANAGEMENT SYSTEM

Prof. ARMANDO RUGGERI

SASHANK N
522502

INTRODUCTION

Small context about data :

In this gigantic world, which contains humans, animals, different natural resources and many more things. Data is one that tops the list with an enormous amount of 147 zettabytes(approximately). Since gigabyte is familiar to us in our day to day life, “1 Zettabyte is 1 trillion gigabytes” . And with just a month left to enter into 2025 this number is set to increase to 181 zettabytes. Having this much amount of data it is definitely neccessary to properly manage the storage and usage of data for our works and as a DATA ANALYST it is the primary task to take care of it.

Database & DBMS :

So as we’ve seen above that the amount of data is too large hence it’s impossible to physically store the data(as hard copies) and it is where the concept of **database** arises. A database is a structured collection of data that is stored and managed electronically. It allows for the efficient storage, retrieval, modification, and deletion of data, and it organizes the data in a way that enables easy access and management. Databases are used in various applications where large volumes of information need to be handled effectively. Databases are managed using **Database Management Systems (DBMS)**, which provide tools and interfaces for database creation, management, and operation. They are completely safe and maintain the privacy and integrity of the data.

Events Data :

Many organizations and institutions maintain databases for their data and one such field where the use of databases is strongly required is event organization. Every event organizing company will definitely maintain the track of events data for many various reasons and as mentioned earlier maintaining records of that much data physically is highly impossible and so comes the need to make usage of the databases and this project is also a small display of how the different data of many events is stored, managed and sorted out using various queries.

Problems faced in event management system :

Event management systems face challenges such as outdated methods for managing attendees, scheduling, and resources, which reduce efficiency. Modern systems with advanced scheduling, resource tracking, and search capabilities can address these issues. Accessibility is another concern, especially when participants lack remote access to event details. Web-based portals and mobile apps providing real-time updates, ticketing, and resource sharing can help. Managing diverse data like attendee details, schedules, and vendor contracts requires robust database systems for effective tracking and centralized management.

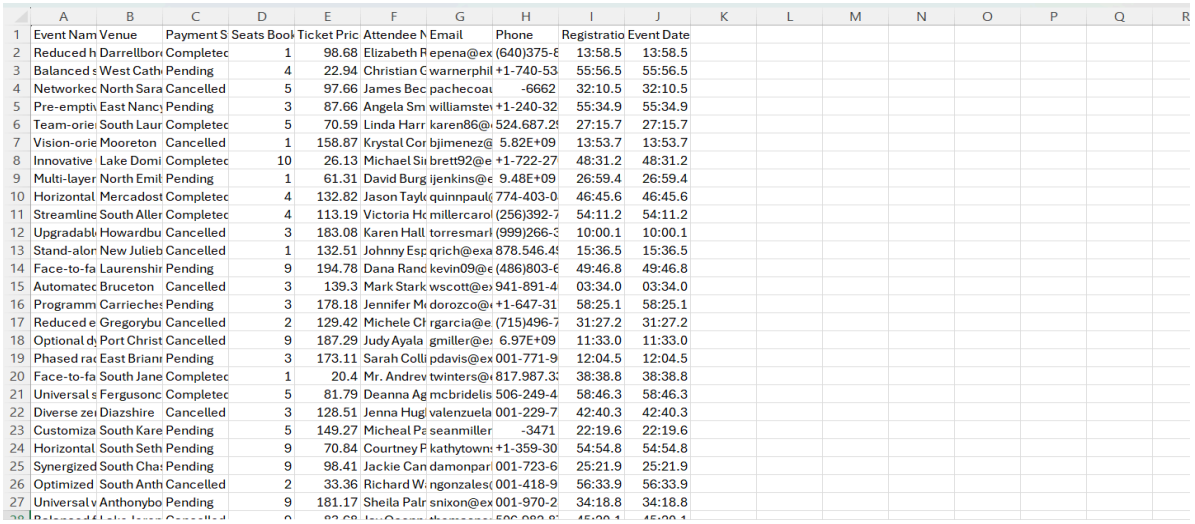
Purpose of Evaluation :

This report aims to present a comprehensive comparison of various database systems for an event management system. The database systems under consideration include NoSQL databases like **Redis, Cassandra, Neo4j, MongoDB**, and the traditional relational database **MySQL**. The objective is to evaluate the performance, scalability, flexibility, and query capabilities of these systems in managing event-related data.

DATASET AND EXPERIMENT DESIGN

A dataset representative of an event management system has been constructed. It includes information about attendees, events, venues, schedules, attendees contact details, payment status and few other entities. To ensure thorough analysis, we will generate datasets of varying sizes, ranging from **250,000 to 1,000,000 records**. These datasets will be loaded into the selected databases, ensuring consistent informational content across them.

To evaluate the efficiency and effectiveness of each database, we will design and execute a series of queries. These queries will use various entities and selection filters, giving us a better understanding of how each database manages more complex processes.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Event Nam	Venue	Payment S	Seats Book	Ticket Pric	Attendee N	Email	Phone	Registratio	Event Date								
2	Reduced h	Darrellbor	Completec	1	98.68	Elizabeth R	epena@ex	(640)375-8	13:58.5	13:58.5								
3	Balanced s	West Cath	Pending	4	22.94	Christian C	warnerphil	+1-740-53	55:56.5	55:56.5								
4	Networkec	North Sara	Cancelled	5	97.66	James Bec	pachecoa	-6662	32:10.5	32:10.5								
5	Pre-emptiv	East Nancy	Pending	3	87.66	Angela Sm	williamste	+1-240-32	55:34.9	55:34.9								
6	Team-orie	South Laur	Completec	5	70.59	Linda Harr	karen86@	524.687.2	27:15.7	27:15.7								
7	Vision-orie	Mooreton	Cancelled	1	158.87	Krystal Cor	bjimenez@	5.82E+09	13:53.7	13:53.7								
8	Innovative	Lake Domi	Completec	10	26.13	Michael Sir	brett92@e	+1-722-27	48:31.2	48:31.2								
9	Multi-layer	North Emil	Pending	1	61.31	David Burg	ijenkins@e	9.48E+09	26:59.4	26:59.4								
10	Horizontal	Mercadost	Completec	4	132.82	Jason Tayl	quinnpaul(	774-403-0	46:45.6	46:45.6								
11	Streamline	South Aller	Completec	4	113.19	Victoria Hc	millercarol	(256)392-7	54:11.2	54:11.2								
12	Upgradabl	Howardbu	Cancelled	3	183.08	Karen Hall	torresmarl	(999)266-3	10:00.1	10:00.1								
13	Stand-alor	New Julieb	Cancelled	1	132.51	Johnny Esp	qrich@exa	878.546.45	15:36.5	15:36.5								
14	Face-to-fa	Laurenshir	Pending	9	194.78	Dana Rand	kevin09@e	(486)803-6	49:46.8	49:46.8								
15	Automatec	Bruceton	Cancelled	3	139.3	Mark Stark	wscott@ex	941-891-4	03:34.0	03:34.0								
16	Programm	Carrieches	Pending	3	178.18	Jennifer Mc	dorozco@e	+1-647-31	58:25.1	58:25.1								
17	Reduced e	Gregorybu	Cancelled	2	129.42	Michele Cl	rgarcia@e	(715)496-7	31:27.2	31:27.2								
18	Optional dy	Port Christ	Cancelled	9	187.29	Judy Ayala	gmiller@ex	6.97E+09	11:33.0	11:33.0								
19	Phased rac	East Brian	Pending	3	173.11	Sarah Colli	pdavis@ex	001-771-9	12:04.5	12:04.5								
20	Face-to-fa	South Jane	Completec	1	20.4	Mr. Andrev	twinters@e	817.987.3	38:38.8	38:38.8								
21	Universal s	Fergusonc	Completec	5	81.79	Deanna Ag	mcbridelis	506-249-4	58:46.3	58:46.3								
22	Diverse zer	Diazshire	Cancelled	3	128.51	Jenna Hug	valenzuela	001-229-7	42:40.3	42:40.3								
23	Customiza	South Kare	Pending	5	149.27	Micheal Pa	seanmiller	-3471	22:19.6	22:19.6								
24	Horizontal	South Seth	Pending	9	70.84	Courtney P	kathytown	+1-359-30	54:54.8	54:54.8								
25	Synergized	South Cha	Pending	9	98.41	Jackie Can	damonpar	001-723-6	25:21.9	25:21.9								
26	Optimized	South Anth	Cancelled	2	33.36	Richard W	ngonzales(	001-418-9	56:33.9	56:33.9								
27	Universal v	Anthonybo	Pending	9	181.17	Sheila Palr	snixon@ex	001-970-2	34:18.8	34:18.8								

To evaluate the efficiency and effectiveness of each database, we will design and execute a series of queries. These queries will use various entities and selection filters, giving us a better understanding of how each database manages more complex processes.

Each test will be run 30 times to ensure statistical reliability. Performance metrics like query execution time will be measured and histograms will be plotted using these saved query execution times. By analyzing the data, we will determine the database that exhibits superior query execution times under equal hardware and software conditions.

DATASET GENERATION

Our main objective is to compare databases and so we require diverse sets of simulated data ranging from 250,000 to 1,000,000 and for that purpose I've used a python module named FAKER in order to generate realistic fake data and this generation is done in Google Collab.

```python
import csv
from faker import Faker
import random
from datetime import timedelta

# Set the number of records for the dataset
dataset_size = 250000

# Initialize Faker
fake = Faker()

# Create a list to store the generated data
data = []

# Generate the data
for _ in range(dataset_size):
    # Generate fake event data
    event_name = fake.catch_phrase()
    event_venue = fake.city()

    # Generate fake attendee data
    attendee_name = fake.name()
    attendee_email = fake.email()
    attendee_phone = fake.phone_number()

    # Generate fake registration and event data
    registration_date = fake.date_time_this_year()
    event_date = registration_date + timedelta(days=random.randint(1, 90))

    # Generate payment and ticket data
    payment_status = random.choice(["Completed", "Pending", "Cancelled"])
    seats_booked = random.randint(1, 10)
    ticket_price = round(random.uniform(20, 200), 2)  # Ticket price between $20 and $200

    # Add the data to the dataset list
    data.append([event_name, event_venue, payment_status, seats_booked, ticket_price,
```

```python
    # Add the data to the dataset list
    data.append([event_name, event_venue, payment_status, seats_booked, ticket_price,
                 attendee_name, attendee_email, attendee_phone,
                 registration_date, event_date])

# Define the filename
filename = "event_management_dataset.csv"

# Save the dataset as a CSV file
with open(filename, 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Event Name', 'Venue', 'Payment Status', 'Seats Booked', 'Ticket Price',
                     'Attendee Name', 'Email', 'Phone', 'Registration Date', 'Event Date'])  # Write header
    writer.writerows(data)

print(f"Dataset of size {dataset_size} saved as {filename}.")
```

From the above pictures you can see the code that's used to generate the dataset required for the comparison of the databases.

INFO ABOUT VARIOUS DBMS's

Cassandra :

Apache Cassandra is an open-source, distributed NoSQL database designed for handling large amounts of data across multiple servers with high availability and fault tolerance. It uses a decentralized architecture, ensuring no single point of failure. Cassandra is optimized for high scalability and write-intensive workloads, making it ideal for big data applications. It provides tunable consistency, allowing developers to balance performance and data accuracy. Cassandra is known for its reliability in managing real-time, large-scale data.

MongoDB :

MongoDB is an open-source, document-oriented NoSQL database designed for scalability and high performance. It stores data in JSON-like documents, which makes it flexible and easy to model complex data. MongoDB is ideal for managing large volumes of unstructured or semi-structured data and is widely used in applications like content management and real-time data processing.

MySQL :

MySQL is an open-source, relational database management system (RDBMS) that uses structured query language(SQL) for managing and querying data. It is widely known for its reliability, performance, and ease of use. MySQL is commonly used in web applications, particularly for managing structured data and supporting large-scale websites.

Neo4j :

Neo4j is an open-source, graph database designed to efficiently store and query highly connected data. It uses a graph structure with nodes, relationships, and properties, which makes it ideal for handling complex relationships such as social networks, recommendation engines, and fraud detection. Neo4j excels in traversing relationships and performing operations like pathfinding, which is challenging for traditional relational databases.

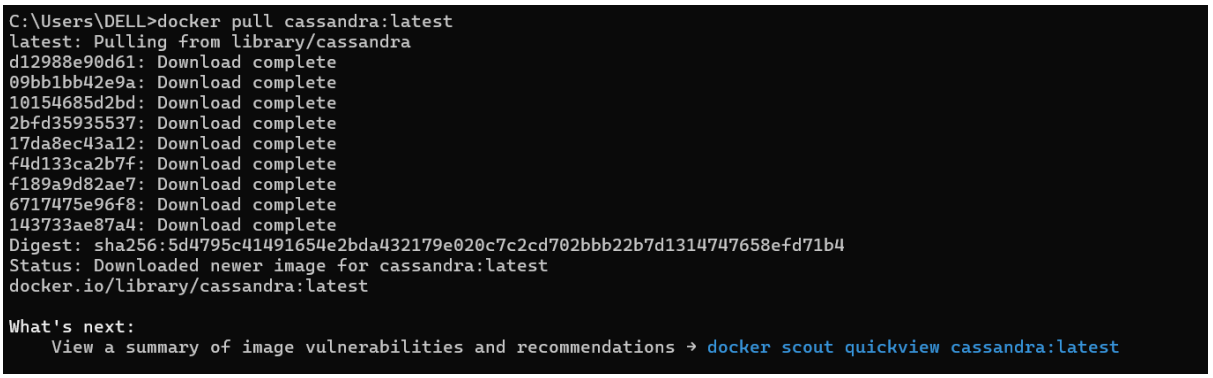
Redis :

Redis is an open-source, in-memory key-value database that supports a wide range of data. Unlike traditional databases, Redis stores data primarily in memory, enabling ultra-fast operations. Redis is designed for high-performance and low-latency applications. It is well-suited for scenarios requiring rapid access to frequently used data.

ESTABLISHING CONNECTION WITH DBMS AND DATA INSERTION

Cassandra :

To establish connection with Cassandra I found out using DOCKER would be a more efficient way.



Using command prompt or windows terminal I've pulled out the latest stable version of the cassandra image. Using the following code I've started a new container named "my cassandra".

"docker run --name my-cassandra -d cassandra:latest"

And with the help of below code I've started the boot process to start up and run it.

"**docker logs my-cassandra"**

Once the container is running, you can connect to Cassandra's CQL shell(cqlsh) to interact with the database.

"docker exec -it my-cassandra cqlsh"

This command opens an interactive shell inside the Cassandra container, and we can start writing Cassandra queries.

Once the connection is established now it's time to enter data into the container/server and it is done with the help of the following code.

```python
from cassandra.cluster import Cluster
from cassandra.query import BatchStatement
from datetime import datetime
import csv

# Set up connection to Cassandra on localhost
cluster = Cluster(['localhost'])
session = cluster.connect()

# Create keyspace if it doesn't exist
session.execute(
    """
    CREATE KEYSPACE IF NOT EXISTS event_management
    WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 }
    """
)

# Set the keyspace
session.set_keyspace('event_management')

# Create table if it doesn't exist
session.execute(
    """
    CREATE TABLE IF NOT EXISTS events (
        event_name text,
        venue text,
        payment_status text,
        seats_booked int,
        ticket_price decimal,
        attendee_name text,
        email text,
        phone text,
        registration_date timestamp,
        event_date timestamp,
        PRIMARY KEY (event_name, attendee_name)
    )
    """
```

```python
    )
    """
)

# Prepare insert statement
insert_statement = session.prepare(
    """
    INSERT INTO events (event_name, venue, payment_status, seats_booked, ticket_price,
                        attendee_name, email, phone, registration_date, event_date)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    """
)

# Configurable batch size to optimize performance
batch_size = 50
filename = r'C:\Users\DELL\Downloads\event_management_dataset (1).csv'

with open(filename, 'r') as file:
    csv_data = csv.reader(file)
    next(csv_data)  # Skip header row if present

    batch = BatchStatement()
    count = 0  # Track the number of rows in the current batch

    for row in csv_data:
        try:
            # Unpack and process row data
            (event_name, venue, payment_status, seats_booked, ticket_price,
             attendee_name, email, phone, registration_date, event_date) = row

            seats_booked = int(seats_booked)
            ticket_price = float(ticket_price)

            # Adjust the date format to handle microseconds
            registration_date = datetime.strptime(registration_date, '%Y-%m-%d %H:%M:%S.%f')
            event_date = datetime.strptime(event_date, '%Y-%m-%d %H:%M:%S.%f')
```

```python
            # Add to batch
            batch.add(insert_statement, (event_name, venue, payment_status, seats_booked, ticket_price,
                                         attendee_name, email, phone, registration_date, event_date))
            count += 1

            # Execute and clear the batch when reaching the batch size
            if count % batch_size == 0:
                session.execute(batch)
                print(f"Batch of {batch_size} rows inserted.")
                batch.clear()  # Clear the batch for the next set of rows

        except Exception as e:
            print(f"Error processing row {row}: {e}")
            continue  # Skip to next row on error

    # Execute any remaining rows in the final batch
    if count % batch_size != 0:
        try:
            session.execute(batch)
            print("Final batch insert completed successfully.")
        except Exception as e:
            print(f"Error during final batch insert: {e}")

# Retrieve and display one row from the events table
row = session.execute("SELECT * FROM events LIMIT 1").one()

if row:
    print(f"Event Name: {row.event_name}")
    print(f"Venue: {row.venue}")
    print(f"Payment Status: {row.payment_status}")
    print(f"Seats Booked: {row.seats_booked}")
    print(f"Ticket Price: {row.ticket_price}")
    print(f"Attendee Name: {row.attendee_name}")
    print(f"Email: {row.email}")
    print(f"Phone: {row.phone}")
    print(f"Registration Date: {row.registration_date}")
```

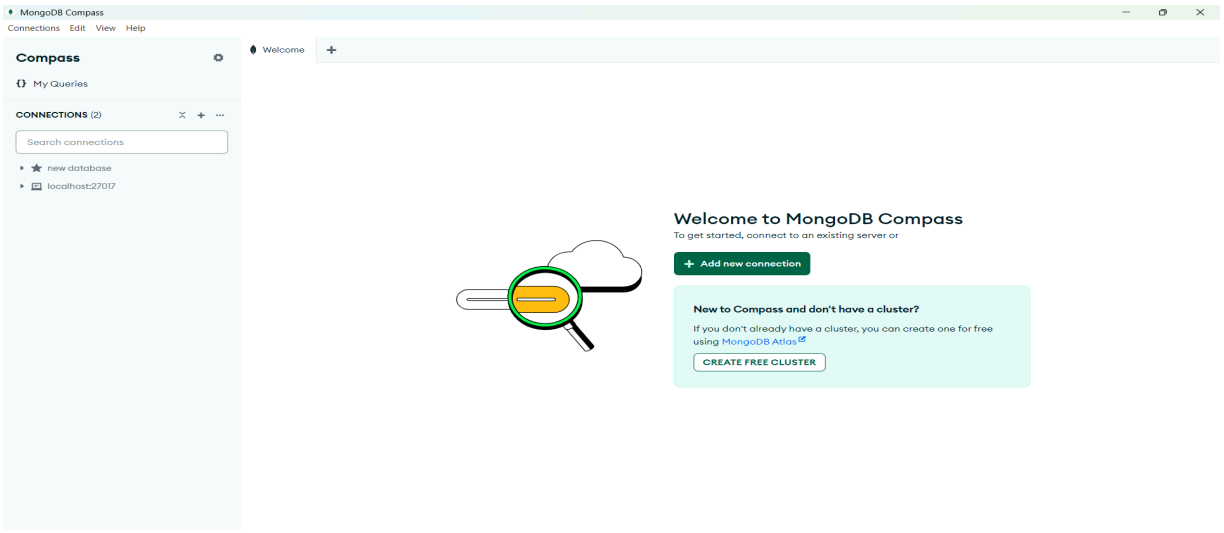
```
    print(f"Event Name: {row.event_name}")
    print(f"Venue: {row.venue}")
    print(f"Payment Status: {row.payment_status}")
    print(f"Seats Booked: {row.seats_booked}")
    print(f"Ticket Price: {row.ticket_price}")
    print(f"Attendee Name: {row.attendee_name}")
    print(f"Email: {row.email}")
    print(f"Phone: {row.phone}")
    print(f"Registration Date: {row.registration_date}")
    print(f"Event Date: {row.event_date}")
else:
    print("No data found.")

# Close the Cassandra session and cluster
session.shutdown()
cluster.shutdown()
```

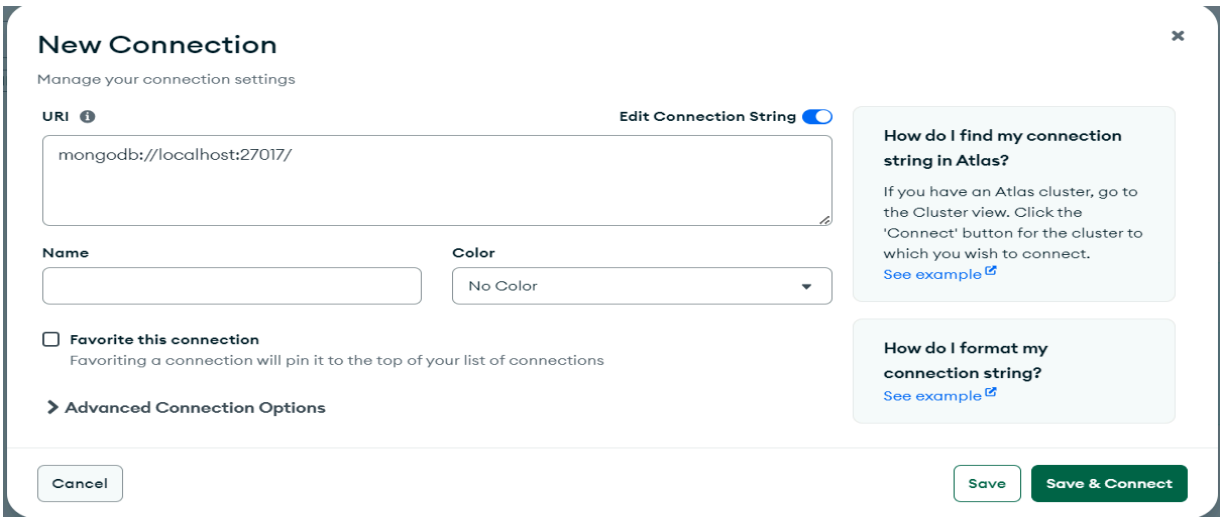
```
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
Batch of 50 rows inserted.
```

As you can see that a dataset is being inserted in batches and the main reason to do it is to efficiently process the data. The same process is done to insert the data from all the four datasets of differing sizes. This is how connection is established with the cassandra server and data is loaded into it to perform queries.

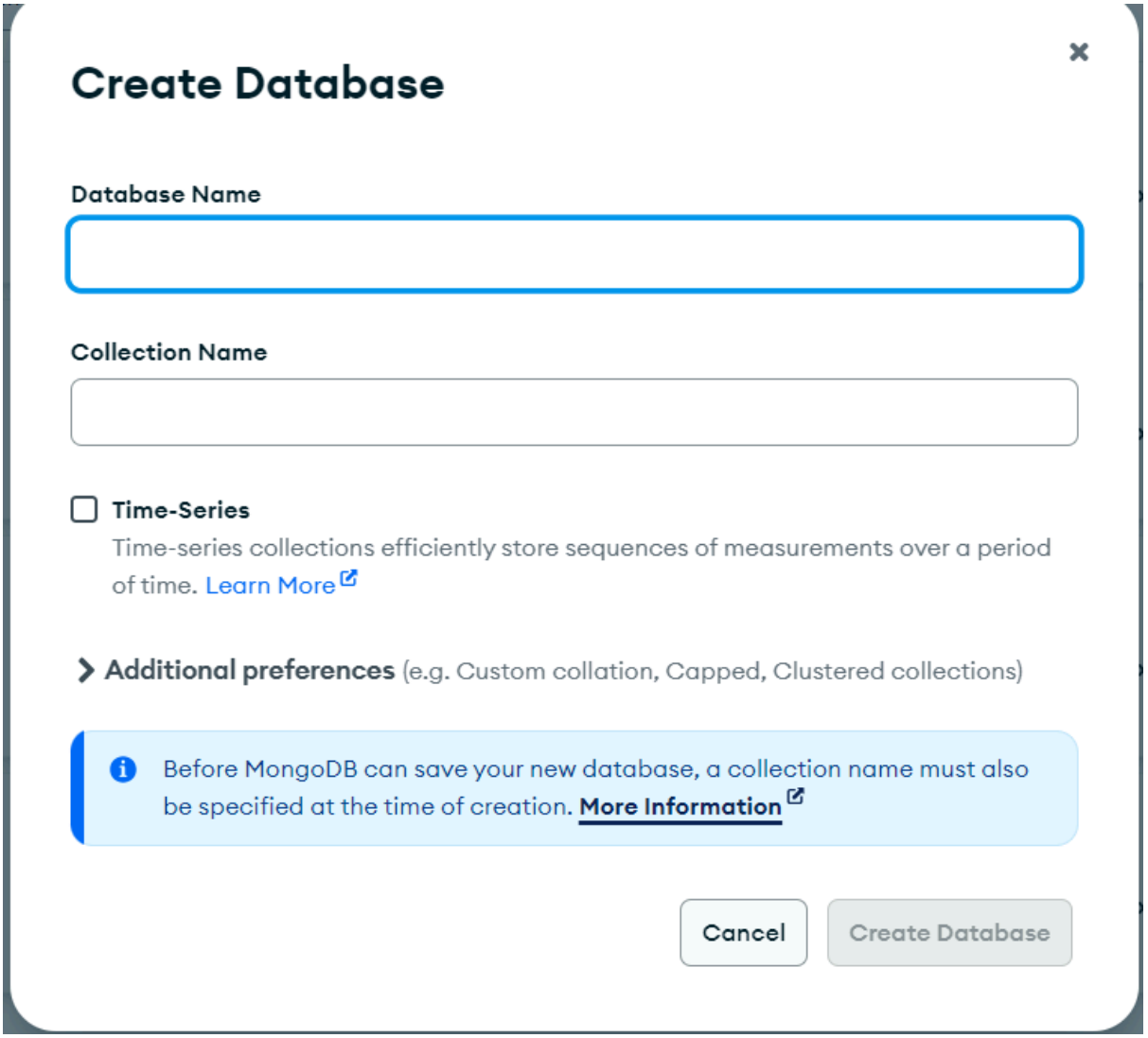
MongoDB :

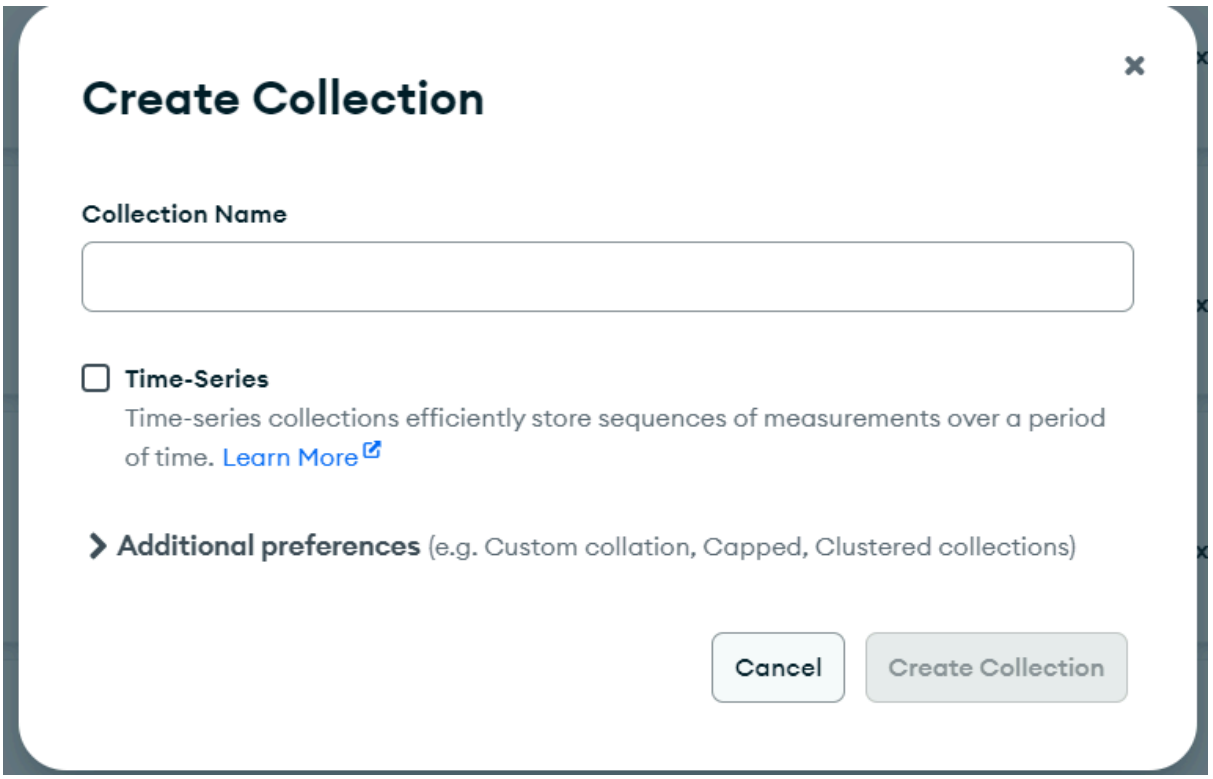
After downloading the MongoDB Compass application on to the PC, I've added a new connection



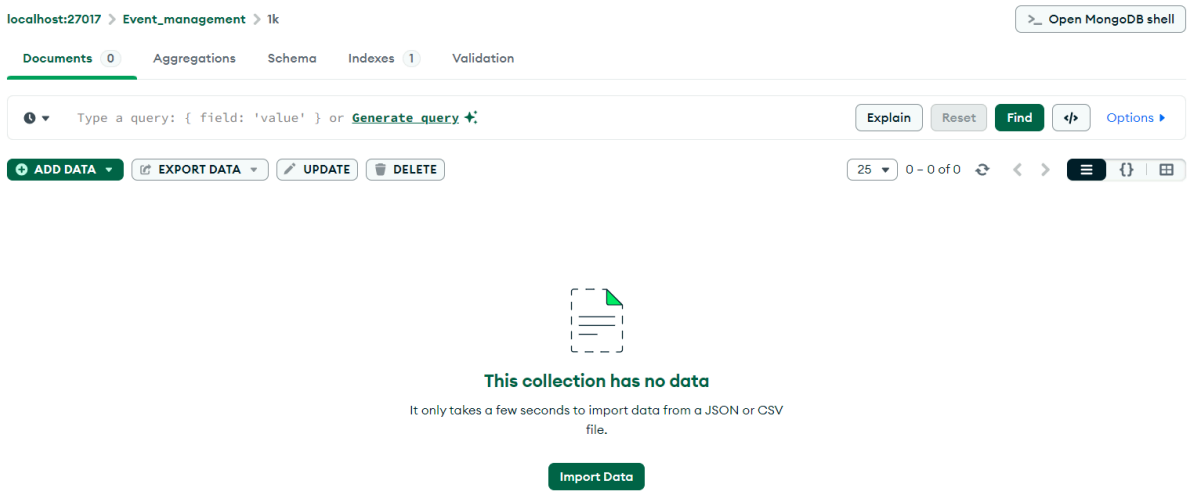


I then created a database followed by different collections where I need to upload all the data from the datasets.

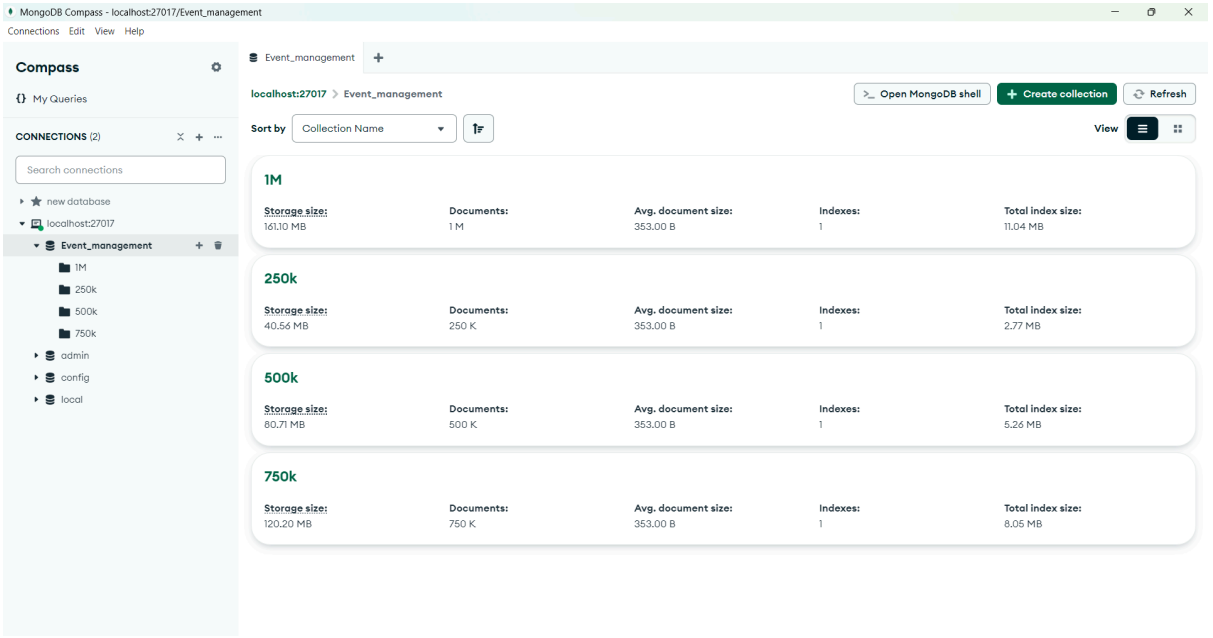




After creating the database and collection as you can see below we can import the data into these collections that we've created.



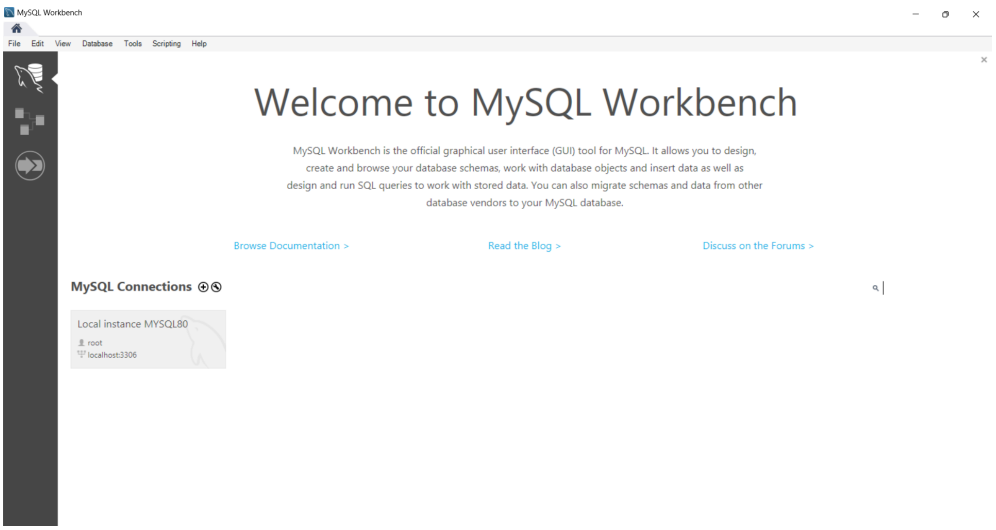
Once all the process is done you can see the following.



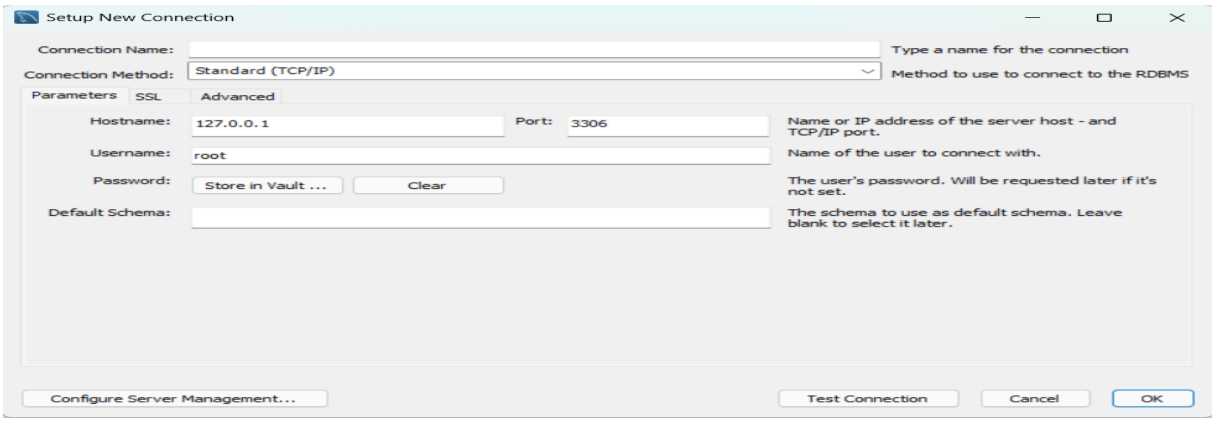
And with this you can understand that the connection is established and data is inserted into the database as well. Now the required queries can be performed.

MySQL :

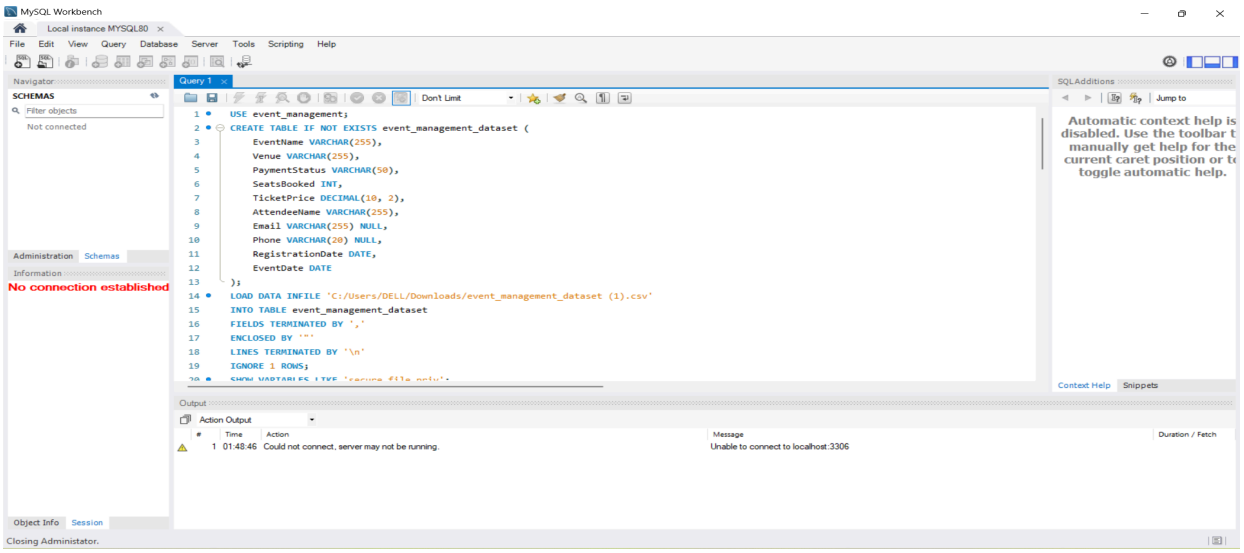
As we've previously studied that MySQL being a relational database management system the data insertion into it has to be done by creating tables as the data inserted into it will be stored as rows and columns. And for this I've used MySQL workbench(both for creating and establishing a connection and to create tables to insert data into it).



As you can see, I've already established a connection to the host. If you want to establish a new connection we need to click on the '+' sign and enter the necessary username, password, connection name and method of connection as well.



I've created a table for the instance of inserting the data into that table. Each individual table has been created to insert data of different dataset sizes.



Once these tables are created I inserted the data into these tables using the following code.

```python
import csv
import mysql.connector

# MySQL server configuration
mysql_config = {
    'host': '127.0.0.1',
    'user': 'root',
    'password': 'Sanee@2003',
    'database': 'event_management'
}

# Path to your CSV file
csv_file_path = r'C:\Users\DELL\Downloads\event_management_dataset (1).csv'

# Create MySQL connection
def create_connection():
    try:
        conn = mysql.connector.connect(**mysql_config)
        return conn
    except mysql.connector.Error as err:
        print("Error connecting to MySQL:", err)
        return None

# Batch insert function to handle large datasets
def import_csv_to_mysql_batch(conn, csv_file_path, batch_size=1000):
    if not conn:
        return

    try:
        with open(csv_file_path, 'r') as csv_file:
            csv_reader = csv.DictReader(csv_file)
            cursor = conn.cursor()

            # Prepare the SQL query for batch insertion
            insert_query = """
                INSERT INTO event_management_dataset
                (EventName, Venue, PaymentStatus, SeatsBooked, TicketPrice, AttendeeName, Email, Phone, RegistrationDate, EventDa
                VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
            """

            batch_data = []
            for idx, row in enumerate(csv_reader):
                # Collect the data in batch
                batch_data.append((
                    row['Event Name'], row['Venue'], row['Payment Status'], row['Seats Booked'],
                    row['Ticket Price'], row['Attendee Name'], row['Email'], row['Phone'],
                    row['Registration Date'], row['Event Date']
                ))

                # Insert data in batches
                if len(batch_data) >= batch_size:
                    cursor.executemany(insert_query, batch_data)
                    conn.commit()
                    batch_data = []  # Reset batch after insert

            # Insert any remaining rows
            if batch_data:
                cursor.executemany(insert_query, batch_data)
                conn.commit()

            print("CSV data imported successfully in batches!")

    except mysql.connector.Error as err:
        print("Error importing CSV data:", err)

    finally:
        if conn.is_connected():
            cursor.close()
            conn.close()

if __name__ == "__main__":
    connection = create_connection()
```

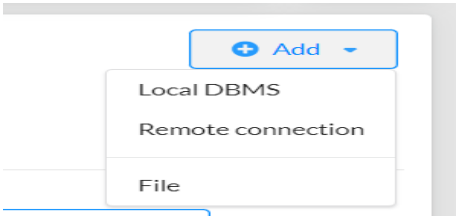
```python
if __name__ == "__main__":
    connection = create_connection()
    import_csv_to_mysql_batch(connection, csv_file_path)
```

```
CSV data imported successfully in batches!
```

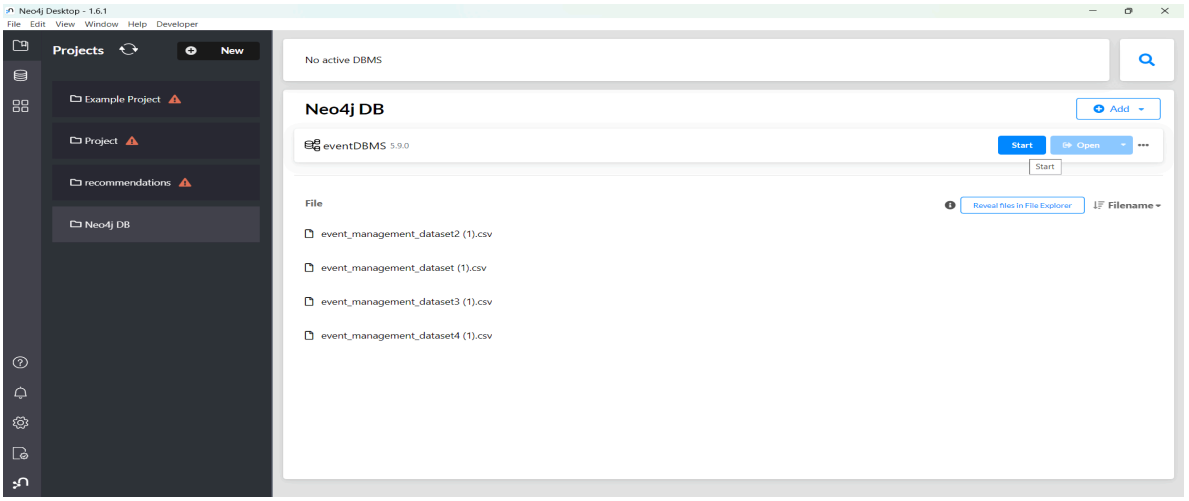
Once we see the output that CSV data is imported successfully, we are ready to implement our queries.

Neo4j :

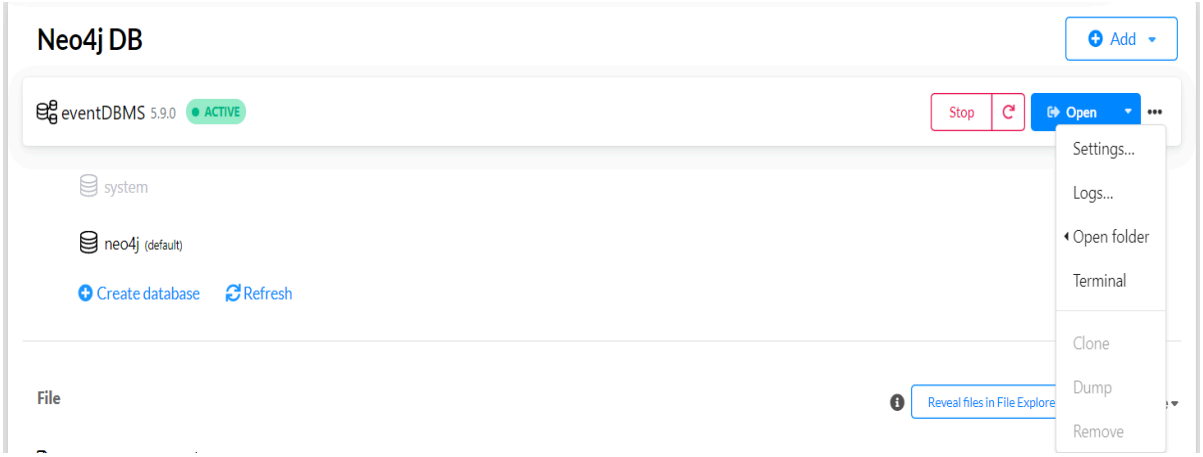
Once the Neo4j desktop application is downloaded and good to go, we can start proceeding with establishing a remote connection from the given list when you click on the ADD option on the Neo4j home screen.

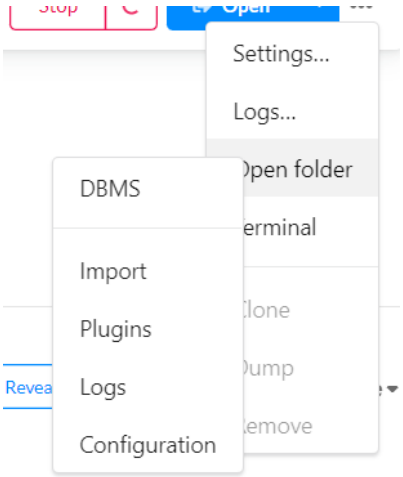


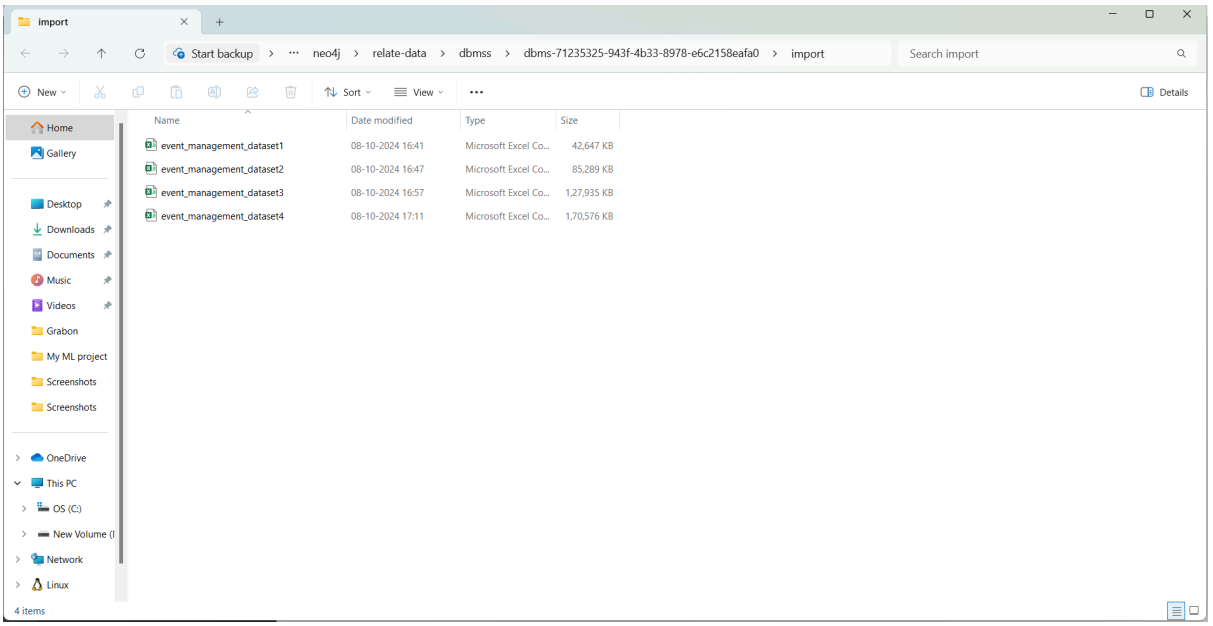
You can proceed with presenting the database name and once it is done you can click on start to establish a connection with the host.



As soon as you're done with establishing a connection you can import the dataset files into the database in the following way. You have to click on the three dots(menu option) that you can see right next to the open option and then you get a list of options as shown in the pictures below and from those you need to select open folder and it again opens one more dropdown list and in that select import which opens a folder where we can add these dataset csv files for them to be a part of the database server.

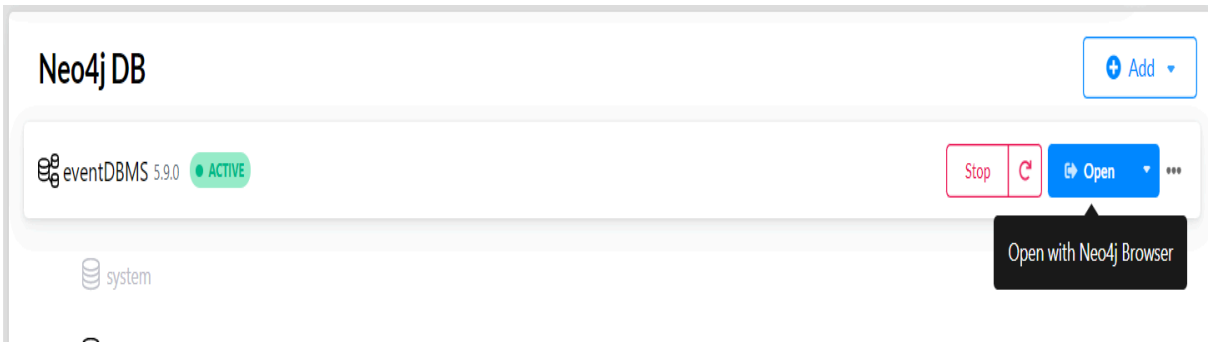






As you can see that I've already added my four dataset files here and thus you can perform the queries later.

You can even access the Neo4j browser as well.



Redis :

Using the following line of code**(“sudo service redis-server start”)** I started the redis local server to establish a connection and also to insert data into it to perform queries. And to confirm whether it’s running or not is a definite check using the following line code **“redis-cli ping”** and if the answer returns PONG it means that there isn’t any problem in the process and everything’s perfectly fine. For the insertion of data the following code is used.

```
import csv
import redis
import time

# Connect to Redis instance
r = redis.Redis(host='localhost', port=6379, db=0)

dataset_size = 250000

with open('Downloads/event_management_dataset (1).csv', 'r') as file:
    csv_data = csv.reader(file)

    # Skip header row if present
    next(csv_data)

    # Iterate over the rows and insert the data into Redis
    for row in csv_data:
        # Check if the row has the correct number of values
        if len(row) != 10:
            continue

        # Unpack event management system fields
        event_name, venue, payment_status, seats_booked, ticket_price, attendee_name, attendee_email, attendee_phone, registratic

        # Use a unique key for each attendee (combination of event name and attendee name)
        key = f'event:{event_name}:{attendee_name}'

        # Insert event data into Redis
        r.hset(key, 'Event Name', event_name)
        r.hset(key, 'Venue', venue)
        r.hset(key, 'Payment Status', payment_status)
        r.hset(key, 'Seats Booked', seats_booked)
        r.hset(key, 'Ticket Price', ticket_price)
        r.hset(key, 'Attendee Name', attendee_name)
        r.hset(key, 'Email', attendee_email)
        r.hset(key, 'Phone', attendee_phone)
        r.hset(key, 'Registration Date', registration_date)
```

The following code is inserting the dataset of size 250,000 and the same process goes for all the remaining datasets as well. Once the insertion of data is done you are good to go with the execution of the queries.

QUERY EXECUTION

Cassandra :

Query 1 :

The following query selects all the columns from the events table in the cassandra database where event_name matches the given key. This loop

runs 30 times and measures the time taken for each execution, storing the results in the query_times list.

```
# Performing the query 30 times and compute the execution time for Cassandra
query_times = []
for _ in range(30):
    key = 'Networked next generation implementation'
    start_time = time.time()

    # Execute the SELECT query
    query = f"SELECT * FROM events WHERE event_name = '{key}'"
    result = session.execute(query)

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)

# Write the execution times to a result file
with open(f'results_query1_cassandra_event250k.txt', 'w') as result_file:
    result_file.write(f"Cassandra Query 1 execution times: {query_times}\n")
```

Query 2 :

This query executes to retrieve a limited number of rows from the events table, specifically aiming to retrieve up to end_mark rows. And in the same way as the above query even this query runs 30 times and the results are stored in the given list.

```
query_times = []
for _ in range(30):
    start_rank = 0
    end_rank = 9
    start_time = time.time()

    # Execute a query to retrieve a limited number of events
    query = f"SELECT * FROM events LIMIT {end_rank}"
    result = session.execute(query)

    rows = []
    for row in result:
        rows.append(row)
        if len(rows) >= end_rank:
            break

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 3 :

The purpose of the following query is to measure the performance of executing a SELECT query on a Cassandra table(events_by_pattern)based on a specific search pattern(‘event:’) simulating data retrieval and measuring the time it takes to run the query multiple times.

```
query_times = []
for _ in range(30):
    pattern = 'event:'
    start_time = time.time()

    # Execute the SELECT query with the composite primary key
    query = f"SELECT * FROM events_by_pattern WHERE pattern = '{pattern}'"
    result = session.execute(query)

    # Iterate through result to simulate data retrieval
    for _ in result:
        pass

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 4 :

The primary goal of this code is to benchmark the performance of a query that retrieves rows based on both pattern and event_name from the events_by_name table.It simulates data processing by iterating over the query results and creating modified data which is stored in the results file. The execution time for each query (including the time to process the results) is measured and stored in the query_times list.

```
query_times = []

for _ in range(30):
    event_name = 'Annual Tech Conference'
    start_time = time.time()

    # Execute the SELECT query to retrieve all rows with the specified event_name
    query = f"SELECT * FROM events_by_name WHERE pattern = 'event:' AND event_name = '{event_name}'"
    rows = session.execute(query)

    results = []

    # Process the retrieved rows if needed
    for row in rows:

        result = {
            'Event ID': row.event_id,
            'Event Name': row.event_name,
            'Venue': row.venue,
            'Payment Status': row.payment_status,
            'Seats Booked': row.seats_booked,
            'Ticket Price': row.ticket_price,
            'Attendee Name': row.attendee_name,
            'Email': row.email,
            'Phone': row.phone,
            'Registration Date': row.registration_date,
            'Event Date': row.event_date,
            'Modified Field': 'Some modification',
            'Additional Field': 'Some additional data',
        }
        results.append(result)

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

MongoDB :

Query 1 :

The primary goal of this query is to measure the performance of retrieving a single document from MongoDB using a specific filter and projection by limiting the retrieved fields to only those relevant to the query. Executes the query 30 times to evaluate response times and store the execution times for performance analysis.

```
# Query filter to search for a specific event (e.g., by Event Name)
query_filter = {"Event Name": "Networked next generation implementation"}
projection = {
    "Event Name": 1,
    "Event Date": 1,
    "Venue": 1,
    "Attendee Name": 1,
    "Email": 1,
    "Phone": 1,
    "Registration Date": 1,
    "Ticket Price": 1,
    "Seats Booked": 1,
    "Payment Status": 1,
    "_id": 0,
}

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]

query_times = []

# Retrieve event information from MongoDB and measure the execution time
for _ in range(30):
    start_time = time.time()
    result = collection.find_one(query_filter, projection)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
    print(result)
```

Query 2 :

This MongoDB query benchmarks the performance of retrieving a paginated list of documents. It matches all documents using an empty filter, applies a projection to limit returned fields, sorts results by Event Date in descending order, and retrieves a specific range using pagination (skip and limit). Executed 30 times, it measures execution times to evaluate the efficiency and consistency of MongoDB's handling of sorting, pagination, and projection.

```
query_filter = {}
projection = {
    "Event Name": 1,
    "Event Date": 1,
    "Venue": 1,
    "Attendee Name": 1,
    "Email": 1,
    "Phone": 1,
    "Registration Date": 1,
    "Ticket Price": 1,
    "Seats Booked": 1,
    "Payment Status": 1,
    "_id": 0,
}

# Skip and limit values for pagination
start_rank = 0
limit = 10

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]

query_times = []

# Retrieve event information from MongoDB and measure the execution time
for _ in range(30):
    start_time = time.time()
    result = collection.find(query_filter, projection).sort(
        "Event Date", -1).skip(start_rank).limit(limit)  # Sorting by Event Date
    for record in result:
        print(record)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 3 :

This MongoDB query benchmarks the performance of retrieving documents with a specified projection that includes fields like Event Name, Event Date, Venue and Payment Status. It matches all documents using an empty filter and retrieves results without applying sorting whereas pagination is implemented with skip and limit to fetch a defined range of 10 documents. Executed 30 times, it measures the execution times to evaluate MongoDB's efficiency in processing and returning data with a limited field projection.

```
query_filter = {}
projection = {
    "Event Name": 1,
    "Event Date": 1,
    "Venue": 1,
    "Payment Status": 1,
}

# Skip and limit values for pagination
start_rank = 0
limit = 10

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]

query_times = []

# Retrieve event information from MongoDB and measure the execution time
for _ in range(30):
    start_time = time.time()
    result = collection.find(query_filter, projection).skip(start_rank).limit(limit)
    for record in result:
        print(record)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 4 :

This MongoDB query benchmarks the performance of retrieving a paginated list of documents. It uses an empty filter to match all documents and a projection that excludes the _id field. Pagination is applied with skip and limit to fetch a range of 22 documents. Executed 30 times, the query measures execution times to evaluate MongoDB's efficiency in handling pagination and minimal projections.

```
query_filter = {}
projection = {
    "_id": 0,
}

# Skip and limit values for pagination
start_rank = 0
limit = 22

# Connect to MongoDB
client = MongoClient(mongodb_uri)
db = client[database_name]
collection = db[collection_name]

query_times = []

# Retrieve event information from MongoDB and measure the execution time
for _ in range(30):
    start_time = time.time()
    result = collection.find(query_filter, projection).skip(start_rank).limit(limit)
    for record in result:
        print(record)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

MySQL :

Query 1 :

This SQL query benchmarks the performance of retrieving records based on a specific email address. The query selects fields such as Event Name,Venue, Payment Status and more from the event_management_dataset table, filtering results where Email matches 'pachecoaustin@example.net'. Executed 30 times, the code measures execution times to evaluate the database's efficiency in handling filtered queries and data retrieval.

```
# MySQL query to fetch event details for a specific attendee based on their email
query = """
SELECT `EventName`, Venue, `PaymentStatus`, `SeatsBooked`, `TicketPrice`, `AttendeeName`, Email, Phone, `RegistrationDate`, `Eve
FROM event_management_dataset
WHERE Email = 'pachecoaustin@example.net'
"""

# Execute the query and measure the execution time
query_times = []
for _ in range(30):
    start_time = time.time()
    cursor.execute(query)
    result = cursor.fetchall()
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
    print(result)
```

Query 2 :

This SQL query benchmarks the performance of retrieving a paginated list of records ordered by the number of seats booked in descending order. It selects fields like event name, venue, payment status, and others from the event management dataset table, fetching the top 10 results with an offset of 0. Executed 30 times, the query measures execution times to assess the database's efficiency in handling sorting, pagination, and data retrieval.

```
query = """
SELECT `EventName`, Venue, PaymentStatus, SeatsBooked, TicketPrice, AttendeeName, Email, Phone, RegistrationDate, EventDate
FROM event_management_dataset
ORDER BY SeatsBooked DESC
LIMIT 10 OFFSET 0
"""

# Execute the query and measure the execution time
query_times = []
for _ in range(30):
    start_time = time.time()
    cursor.execute(query)
    result = cursor.fetchall()
    for record in result:
        print(record)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 3 :

This SQL query benchmarks the performance of retrieving a single record from a table. It selects fields like event name, venue, payment status, and others from the event management dataset table, limiting the result to just one record. Executed 30 times, the query measures execution times to assess the database's efficiency in handling data retrieval for a single entry.

```
# MySQL query to select event details
query = """
SELECT `EventName`, Venue, PaymentStatus, SeatsBooked, TicketPrice, AttendeeName, Email, Phone, RegistrationDate, EventDate
FROM event_management_dataset
LIMIT 1
"""

# Execute the query and measure the execution time
query_times = []
for _ in range(30):
    start_time = time.time()
    cursor.execute(query)
    result = cursor.fetchall()
    if result:
        print(result[0])  # Print the first event record
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 4 :

This SQL query benchmarks the performance of retrieving multiple records from a table. It selects fields like event name, venue, payment status, and seats booked from the event management dataset table. Executed 30 times, the query measures execution times to assess the database's efficiency in handling data retrieval for multiple entries.

```
# MySQL query to select event details
query = """
SELECT `EventName`, Venue, PaymentStatus, SeatsBooked
FROM event_management_dataset
"""

# Execute the query and measure the execution time
query_times = []
for _ in range(30):
    start_time = time.time()
    cursor.execute(query)
    result = cursor.fetchall()
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Neo4j :

Query 1 :

This Cypher query benchmarks the performance of retrieving event details from a Neo4j database. It matches an event with a specific event_id and selects various fields such as event name, venue, payment status, seats booked, ticket price, attendee details, and event dates. The query is executed 30 times, and each time, the execution time is measured to assess the database's efficiency in retrieving event information. The results are collected in a list for each iteration, and any exceptions encountered during the query execution are handled and reported.

```
# Define the Cypher query to retrieve an event's information
cypher_query = (
    "MATCH (e:Event_250k {event_id: 'event123'}) "
    "RETURN e.event_name AS event_name, e.venue AS venue, e.payment_status AS payment_status, "
    "e.seats_booked AS seats_booked, e.ticket_price AS ticket_price, e.attendee_name AS attendee_name, "
    "e.attendee_email AS attendee_email, e.attendee_phone AS attendee_phone, "
    "e.registration_date AS registration_date, e.event_date AS event_date"
)

query_times = []
results = []

# Retrieve an event's information from Neo4j and measure the execution time
for _ in range(30):
    try:
        start_time = time.time()
        result = graph.run(cypher_query).data()  # Fetch as list of dictionaries
        end_time = time.time()
        execution_time = end_time - start_time
        query_times.append(execution_time)
        results.append(result)  # Collect the result for each iteration throughout the loop
    except Exception as e:
        print(f"An error occurred: {e}")
```

Query 2 :

This Cypher query benchmarks the performance of retrieving a limited number of events from a Neo4j database. It selects the first 10 events from the database using the MATCH clause and limits the result to 10 entries. The query is executed 30 times, with execution times measured each time to assess the efficiency of retrieving a set of records. The results are printed for each iteration, and the execution times are stored in a list for analysis.

```
cypher_query = (
    "MATCH (e:Event) RETURN e LIMIT 10"
)

with driver.session() as session:
    # Execute the Cypher query and measure the execution time
    for _ in range(30):
        start_time = time.time()
        result = session.run(cypher_query)
        for record in result:  # Directly iterate over the Result object
            print(record)  # Access individual records
        end_time = time.time()
        execution_time = end_time - start_time
        query_times.append(execution_time)
```

Query 3 :

This Cypher query benchmarks the performance of retrieving a single event node from a Neo4j database. It uses the MATCH clause to find all Event nodes and returns the entire node. The query is executed 30 times, and for each iteration, it retrieves and prints the first event node. Execution times are measured to assess the efficiency of fetching individual event nodes, with the results stored in a list for analysis.

```
# Define the Cypher query to retrieve all Event nodes
cypher_query = (
    "MATCH (e:Event) "
    "RETURN e"  # Return the entire Event node
)

with driver.session() as session:
    # Execute the Cypher query and measure the execution time
    start_time = time.time()
    for _ in range(30):
        result = session.run(cypher_query)
        record = result.single()  # Retrieve a single record from the query result
        if record:
            # Access the Event node (e represents the 'Event' node)
            print(record['e'])  # Print the entire Event node
        end_time = time.time()
        execution_time = end_time - start_time
        query_times.append(execution_time)
```

Query 4 :

This Cypher query benchmarks the performance of retrieving detailed information about events from a Neo4j database. It matches all Event nodes and returns specific properties like event name, venue, payment status, seats booked, ticket price, attendee details, registration date, and event date. The

query is executed 30 times, and in each iteration, the results are collected into a list. The execution time for each query is measured and stored for analysis, providing insights into the efficiency of retrieving and processing event data.

```
# Define the Cypher query to retrieve event details
cypher_query = (
    "MATCH (e:Event) "
    "RETURN e.`Event Name` AS event_name, e.Venue AS venue, "
    "e.`Payment Status` AS payment_status, e.`Seats Booked` AS seats_booked, "
    "e.`Ticket Price` AS ticket_price, e.`Attendee Name` AS attendee_name, "
    "e.Email AS email, e.Phone AS phone, "
    "e.`Registration Date` AS registration_date, e.`Event Date` AS event_date"
)

with driver.session() as session:
    # Execute the Cypher query and measure the execution time
    for _ in range(30):
        start_time = time.time()
        result = session.run(cypher_query)
        results = []
        for record in result:
         results.append(record)
        end_time = time.time()
        execution_time = end_time - start_time
        query_times.append(execution_time)
```

Redis :

Query 1 :

This code benchmarks the performance of retrieving data from a Redis database using a hash key. It constructs a key based on a specific event name and attendee name, and then retrieves all fields associated with that key using hgetall. The query is executed 30 times, and the execution time for each retrieval is measured. These times are collected and stored in a list for performance analysis, providing insights into the efficiency of querying data from Redis based on a composite key.

```
# Benchmarking the query
query_times = []

# Example key for querying
sample_event_name = 'Networked next generation implementation'
sample_attendee_name = 'James Becker'
key = f'event:{sample_event_name}:{sample_attendee_name}'

for _ in range(30):
    start_time = time.time()
    value = r.hgetall(key)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 2 :

This code benchmarks the performance of retrieving the top 10 events based on ticket price from a Redis sorted set. It uses the zrange command to fetch the events with the highest ticket prices (specified by the rank range from start_rank to end_rank). The query is executed 30 times, and the execution time for each retrieval is measured and stored. This allows for performance analysis on how efficiently Redis retrieves the top events sorted by ticket price.

```
# Benchmarking query performance on sorted set
query_times = []

# Query top 10 events by ticket price
for _ in range(30):
    start_rank = 0
    end_rank = 9
    start_time = time.time()
    results = r.zrange('events:ticket_price', start_rank, end_rank, withscores=True)
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 3 :

This code benchmarks the performance of retrieving all keys that match a specific pattern from a Redis database. It uses the keys command with the pattern ‘event:*’ to fetch all keys related to events. The query is executed 30 times, and the execution time for each retrieval is measured and stored in a list. This provides insight into the efficiency of Redis when querying for multiple keys that match a given pattern.

```
# Benchmarking the query performance to retrieve all keys matching 'event:*'
query_times = []

for _ in range(30):
    pattern = 'event:*'
    start_time = time.time()
    keys = r.keys(pattern)  # Retrieve all keys matching the pattern
    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

Query 4 :

This code benchmarks the performance of retrieving event information from a Redis database based on a pattern match for keys (‘event:*’). It first retrieves all keys that match the pattern using the keys command, then iterates over each key to fetch event details using hgetall. For each event, it decodes the

byte strings to extract relevant data like event name, venue, payment status, seats booked, and ticket price, and adds additional fields. The query is executed 30 times, and the execution time for each retrieval is measured and stored in a list, allowing for analysis of Redis' efficiency when retrieving and processing multiple event records.

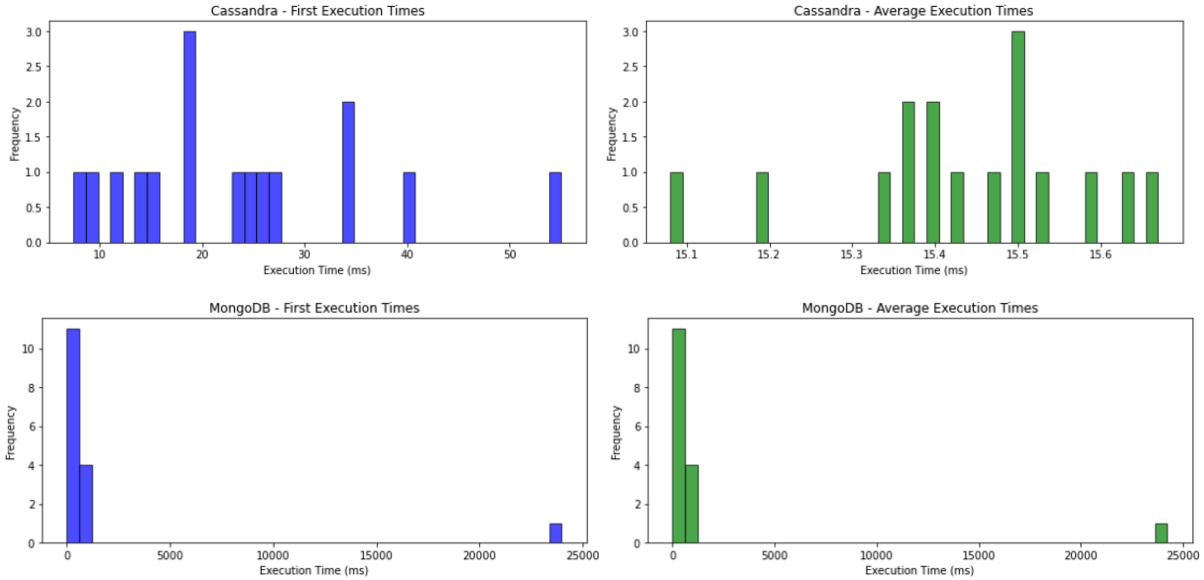
```
query_times = []

for _ in range(30):
    pattern = 'event:*'
    start_time = time.time()

    # Retrieve all keys matching the pattern
    keys = r.keys(pattern)
    results = []

    # Iterate over keys and retrieve event information
    for key in keys:
        key = key.decode()  # Decode the byte string to regular string
        event_info = r.hgetall(key)
        if event_info is not None:
            # Decode all the byte strings for the event information
            event_name = event_info.get(b'Event Name', b'').decode()
            venue = event_info.get(b'Venue', b'').decode()
            payment_status = event_info.get(b'Payment Status', b'').decode()
            seats_booked = event_info.get(b'Seats Booked', b'').decode()
            ticket_price = event_info.get(b'Ticket Price', b'').decode()

            # Populate results with the retrieved data and additional fields
            result = {
                'Event Name': event_name,
                'Venue': venue,
                'Payment Status': payment_status,
                'Seats Booked': seats_booked,
                'Ticket Price': ticket_price,
                'Modified Field': 'Some modification',
                'Additional Field': 'Some additional data',
            }
            results.append(result)

    end_time = time.time()
    execution_time = end_time - start_time
    query_times.append(execution_time)
```

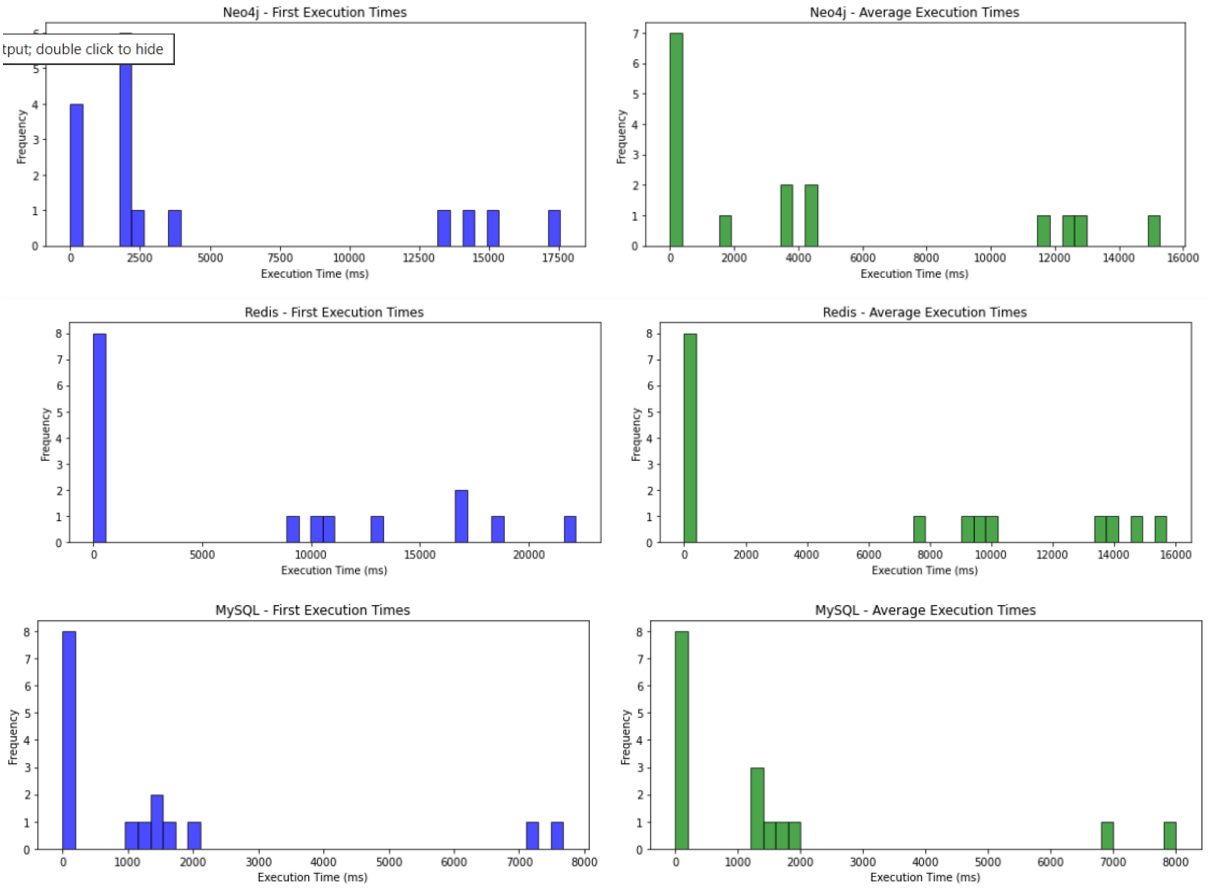
HISTOGRAMS

Histograms are nothing but the visual representation of the distribution of a set of data. It shows how often different values (or ranges of values) appear in your data. Here from the above queries we got the execution time of each query and they are stored as text files and now we make use of those results to plot the graph of those execution times. I have given those text files where execution times are saved and then used this code to plot the histograms.

```python
def load_execution_times(file_name):
    with open(file_name, 'r') as file:
        content = file.read()
    times = re.findall(r'\d+\.\d+', content)  # Extract all floating-point numbers
    return np.array(times, dtype=float) * 1000  # Convert to milliseconds
# Process data for all databases
database_data = {}
for db_name, dataset_sizes in files.items():
    first_exec_times = []
    avg_exec_times = []
    for size, file_list in dataset_sizes.items():
        for file in file_list:
            exec_times = load_execution_times(file)
            first_exec_times.extend(exec_times[:1])  # First execution time
            avg_exec_times.append(np.mean(exec_times[1:31]))  # Average of next 30 executions
    database_data[db_name] = {
        "first": first_exec_times,
        "average": avg_exec_times
    }
# Plot histograms grouped by database
fig, axes = plt.subplots(5, 2, figsize=(16, 20))  # 5 databases × 2 histograms
fig.suptitle('Grouped Histograms of Execution Times by Database', fontsize=18)

for i, (db_name, data) in enumerate(database_data.items()):
    # First execution times
    axes[i, 0].hist(data["first"], bins=bins, color='blue', alpha=0.7, edgecolor='black')
    axes[i, 0].set_title(f'{db_name} - First Execution Times', fontsize=12)
    axes[i, 0].set_xlabel('Execution Time (ms)', fontsize=10)
    axes[i, 0].set_ylabel('Frequency', fontsize=10)
    # Average execution times
    axes[i, 1].hist(data["average"], bins=bins, color='green', alpha=0.7, edgecolor='black')
    axes[i, 1].set_title(f'{db_name} - Average Execution Times', fontsize=12)
    axes[i, 1].set_xlabel('Execution Time (ms)', fontsize=10)
    axes[i, 1].set_ylabel('Frequency', fontsize=10)
# Adjust layout to prevent overlap
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

These histograms plotted shows the execution time in milliseconds. Two histograms have been plotted for each individual query(one is the plotting of the first execution time and the second is the plotting of the average of all the 30 execution times).





And this in short is the matter about histograms.

CONCLUSION

After carrying out the various tests on queries of completely different complexity and datasets of increasing size 250000, 500000, 750000 and 1000000, we can draw a series of conclusions.

From small datasets to large datasets, we can see the performance difference in the Cassandra, MongoDB, MySQL, Neo4j and Redis databases. Cassandra performing well in one query for one dataset is not up to the mark when compared to other databases in other datasets but on an average I can grab

an opinion that Cassandra has a good performance followed by MongoDB, Neo4j, Redis and MySQL being at last.

What I've noticed is a key takeaway that the speed of the query depends on the size of the document for almost all the DBMS.

Through this comparative analysis, we have established a solid foundation for selecting the most suitable Database Management System (DBMS) for our event management system. The insights gained from this study will inform future development and optimization efforts, enabling us to achieve optimal performance and deliver a seamless user experience in our event management application.

THANK YOU