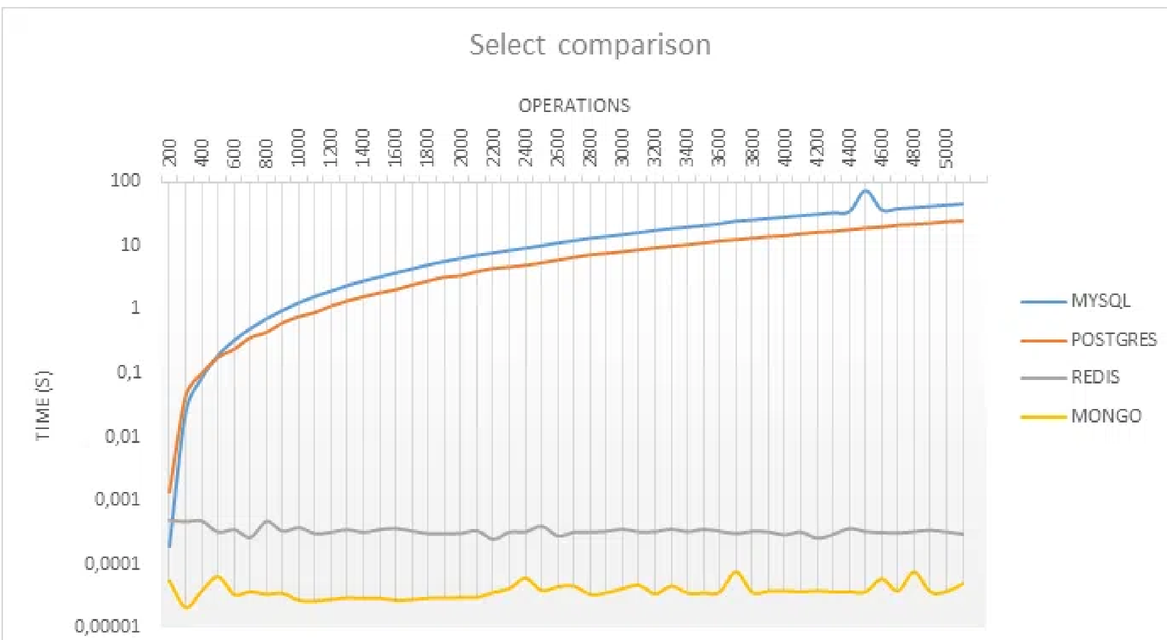
NoSQL Database Labs: BigTable and Neo4J

Overview

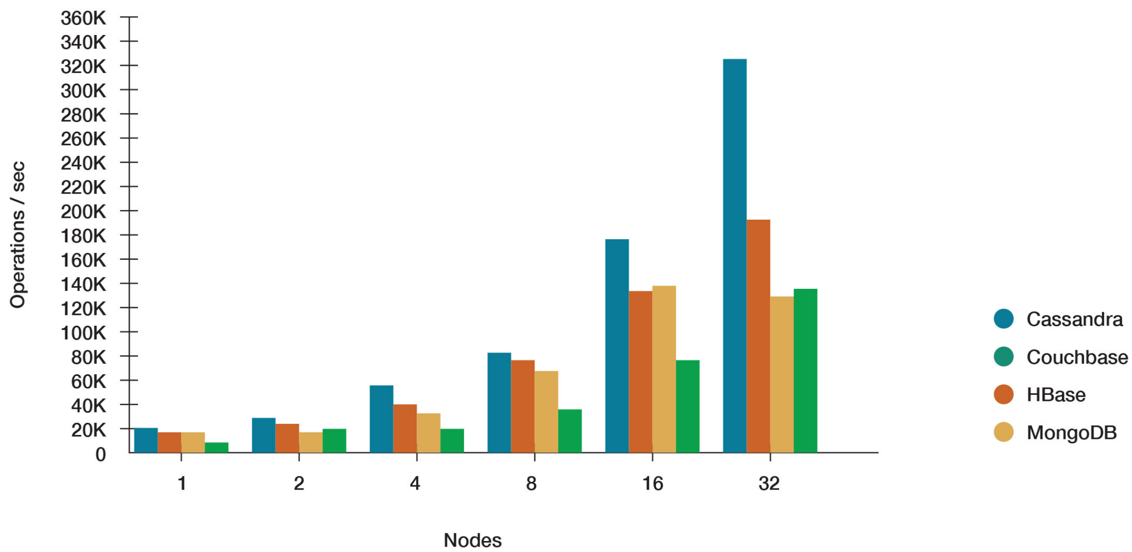
- What is BigTable?
- BigTable Storage Model
- Cloud Execution
- Neo4J
- Neo4J on the Cloud
- Cypher Scripts
- Neo4J in Python
- Readings

Reminder: SQL vs. NoSQL Performance at SELECT Queries



Database Comparison from Profil website, accessible from here

MongoDB does not have the best performance!

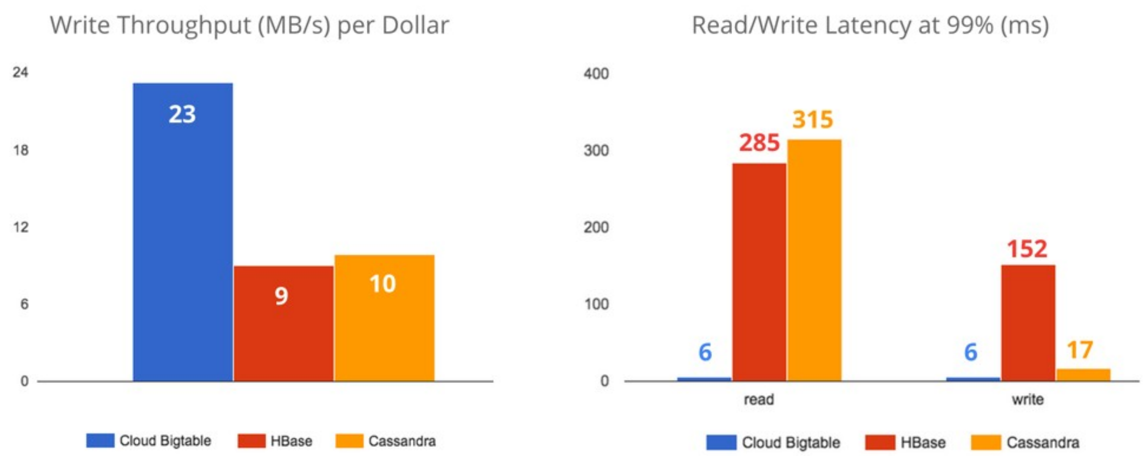


NoSQL DB Comparison, featured in the details of this article

BigTable

- Bigtable is engineered to handle structured data at a vast scale, managing petabytes across thousands of servers for high-demand applications
- Bigtable supports diverse workloads from high-throughput batch processing to real-time, latency-sensitive data serving. This includes 60+ Google products such as Google Analytics and Google Earth
- Bigtable clusters vary in size, from a few to thousands of servers, efficiently managing hundreds of terabytes of data to meet the unique needs of each Google service.

BigTable Performance Statistics



Methodology: Read/Write: 120 total client threads from 10 n1-standard-8s (12 threads/client). Databases loaded with 1TB of data, then immediately tested. YCSB Workload A (50/50 mix 1k read/write; zipfian request distribution). Write throughput: average throughput while loading 1TB of data during the test.

BigTable Performance on 8 vCPU Machine, accessible from here

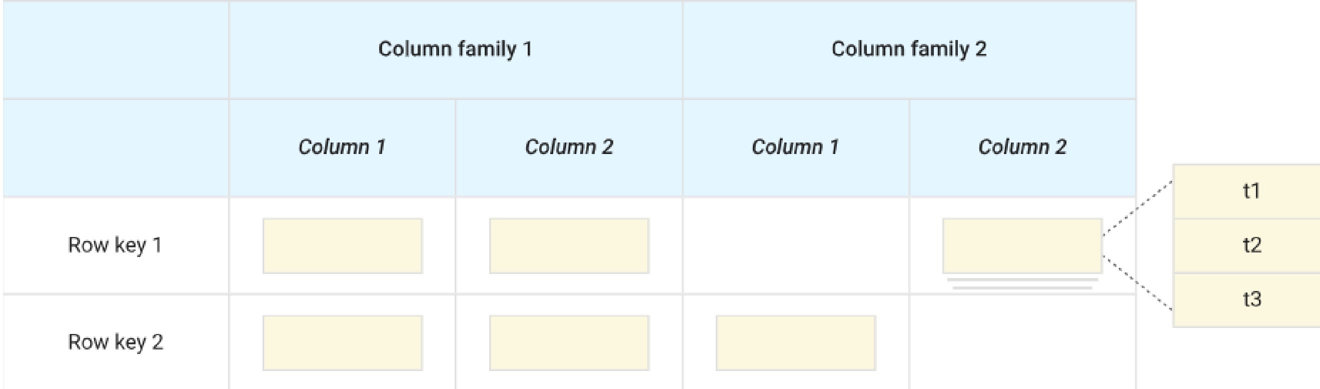
Important Notes

- Bigtable is suitable for tables with sparse data tables are sparse
- if a column is not used in a particular row, it does not take up any space.

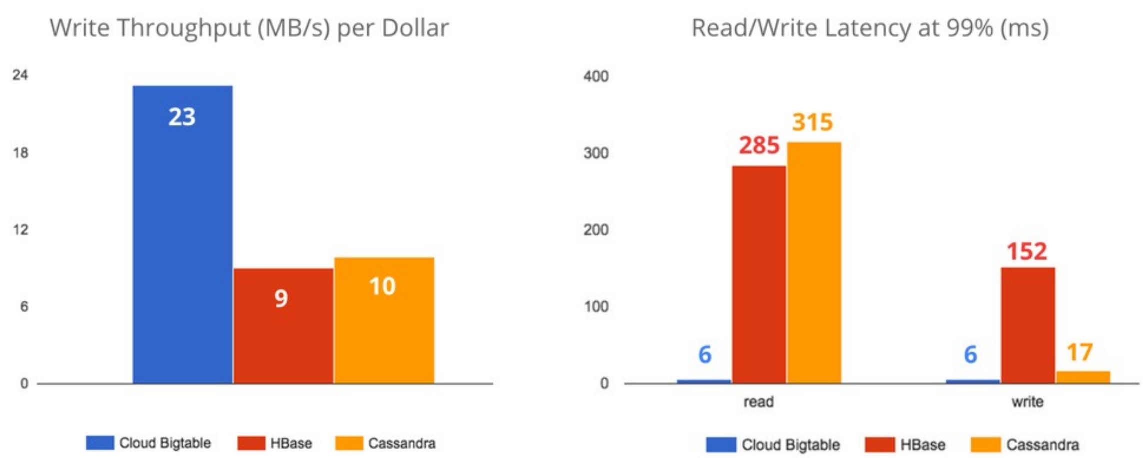
BigTable Storage Model

- Bigtable stores data in massively scalable tables
- Each table is a sorted key/value map.
- The table is composed of rows, each of which typically describes a single entity, and columns, which contain individual values for each row.
- Each row is indexed by a single row key, and columns that are related to one another are typically grouped into a column family.
- Each column is identified by a combination of the column family and a column qualifier, which is a unique name within the column family.

- Each row/column intersection can contain multiple cells.
- Each cell contains a unique timestamped version of the data for that row and column.
- Storing multiple cells in a column provides a record of how the stored data for that row and column have changed over time.



BigTable Performance Statistics



Methodology: Read/Write: 120 total client threads from 10 n1-standard-8s (12 threads/client). Databases loaded with 1TB of data, then immediately tested. YCSB Workload A (50/50 mix 1k read/write; zipfian request distribution). Write throughput: average throughput while loading 1TB of data during the test.

BigTable Performance on 8 vCPU Machine, accessible from here

Run BigTable Locally via Docker

You can run a BigTable emulator Locally. To do so, you need to pull Google Cloud SDK Container and run it. More information can be found here

- Set the **BIGTABLE_EMULATOR_HOST** environment variable to **localhost:8086**. More information can be found on https://cloud.google.com/bigtable/docs/emulator
- Run the docker pull command for the image: **docker pull google/cloud-sdk**
- Execute the docker run command: **docker run -p 127.0.0.1:8086:8086 --rm -ti google/cloud-sdk gcloud beta emulators bigtable start --host-port=0.0.0.0:8086**
- In your terminal, run **docker container ls** or check your docker desktop to see if the container is running

Install Required Packages

In [ ]:

```
!pip install google-cloud-bigtable
!pip install google-cloud-happybase
```

Initialize the Application

In [ ]:

```
from google.cloud import bigtable
from google.cloud import happybase
from google.cloud.bigtable import column_family

#Populate project_id and instance_id if you are running on the cloud
project_id = ""
instance_id = ""

client = bigtable.Client(project=project_id, admin=True)
instance = client.instance(instance_id)
```

Create Tables

In [ ]:

```
table_id = 'test'
print("Creating the {} table.".format(table_id))
table = instance.table(table_id)

print("Creating column family cf1 with Max Version GC rule...")
# Create a column family with GC policy : most recent N versions
# Define the GC policy to retain only the most recent 2 versions
max_versions_rule = column_family.MaxVersionsGCRule(2)
column_family_id = "cf1"
column_families = {column_family_id: max_versions_rule}
if not table.exists():
    table.create(column_families=column_families)
else:
    print("Table {} already exists.".format(table_id))
```

Insert Rows into Tables

In [ ]:

```
import datetime

print("Writing some greetings to the table.")
greetings = ["Hello World!", "Hello Cloud Bigtable!", "Hello Python!"]
rows = []
column = "greeting".encode()
for i, value in enumerate(greetings):
    # Note: This example uses sequential numeric IDs for simplicity,
    # but this can result in poor performance in a production
    # application.  Since rows are stored in sorted order by key,
    # sequential keys can result in poor distribution of operations
    # across nodes.
    #
    # For more information about how to design a Bigtable schema for
    # the best performance, see the documentation:
    #
    #     https://cloud.google.com/bigtable/docs/schema-design
    row_key = "greeting{}".format(i).encode()
    row = table.direct_row(row_key)
    row.set_cell(
        column_family_id, column, value, timestamp=datetime.datetime.utcnow()
    )
    rows.append(row)
table.mutate_rows(rows)
```

Retrieve All Rows in BigTable Table!

In [ ]:
```
print("Scanning for all greetings:")
partial_rows = table.read_rows()

for row in partial_rows:
    cell = row.cells[column_family_id][column][0]
    print(cell.value.decode("utf-8"))
```

Delete Tables

In [ ]:
```
print("Deleting the {} table.".format(table_id))
table.delete()
```

Take Home Excercise

Read the Example for More Inforamtion https://cloud.google.com/bigtable/docs/samples-python-hello

BigTable Cloud Execution

To create Google BigTable instance on GCP, conduct the following activities:

- 1. Download and Configure Google Cloud SDK from this URL: https://cloud.google.com/sdk/docs/install
- 2. Enable the Billing on Your GCP Account
- 3. Ensure you have a project created
- 4. Enable the BigTable API and Create a BigTable Instance from https://console.cloud.google.com/bigtable/instances
- 5. Update your project ID and Instance ID in the code.

Graph Databases

Why Graph Databases/Stores?!

- Graph databases/Stores naturally represent entities and relationships, reducing **impedance mismatch** compared to relational databases.
- Graph DBs align closely with how applications and humans view data, making it easier to model real-world domains.
- No need for complex joins to reconstruct relationships, improving performance and simplicity.

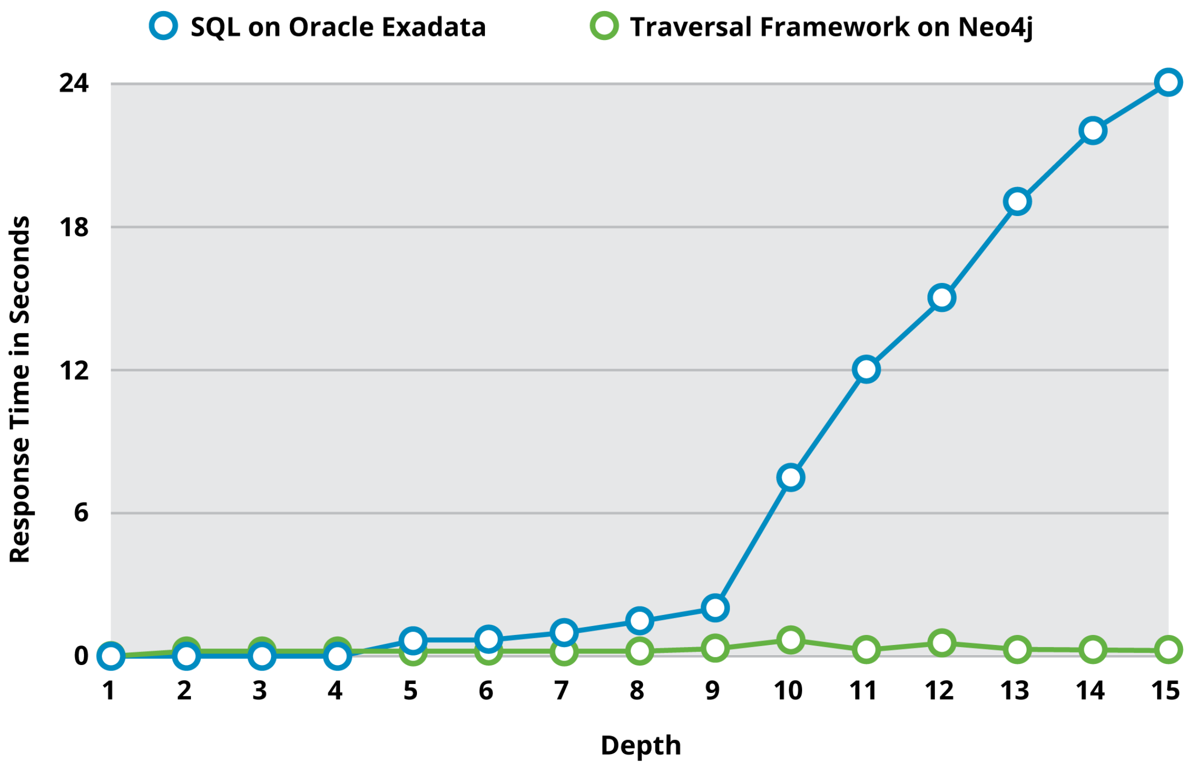
Graph DB Use Cases

- Cybercrime Networks: Graph databases map relationships in cybercrime activity, supporting investigators in analyzing attack patterns and identifying threats.
- Recommendation Engines: Graph databases excel in constructing recommendation systems by modeling intricate relationships between users, products, and preferences. This facilitates the discovery of complex patterns, such as "customers who purchased X also tend to purchase Y," enabling more refined and personalized recommendations.
- Fraud Detection: By mapping and analyzing the complex web of relationships between transactions, accounts, and entities, graph databases can detect sophisticated fraud patterns.

Neo4J

- Founded in 2000.
- Available in open-source and commercial editions.
- Highly scalable.
- Uses a powerful query lanaguage named Cypher which aids in traversing graphs smoothly.

Neo4J Performance Statistics



Oracle vs. Neo4J Query Performance for Finding all of a plant's ancestors at Monsanto, accessible from here

Neo4J on the Cloud

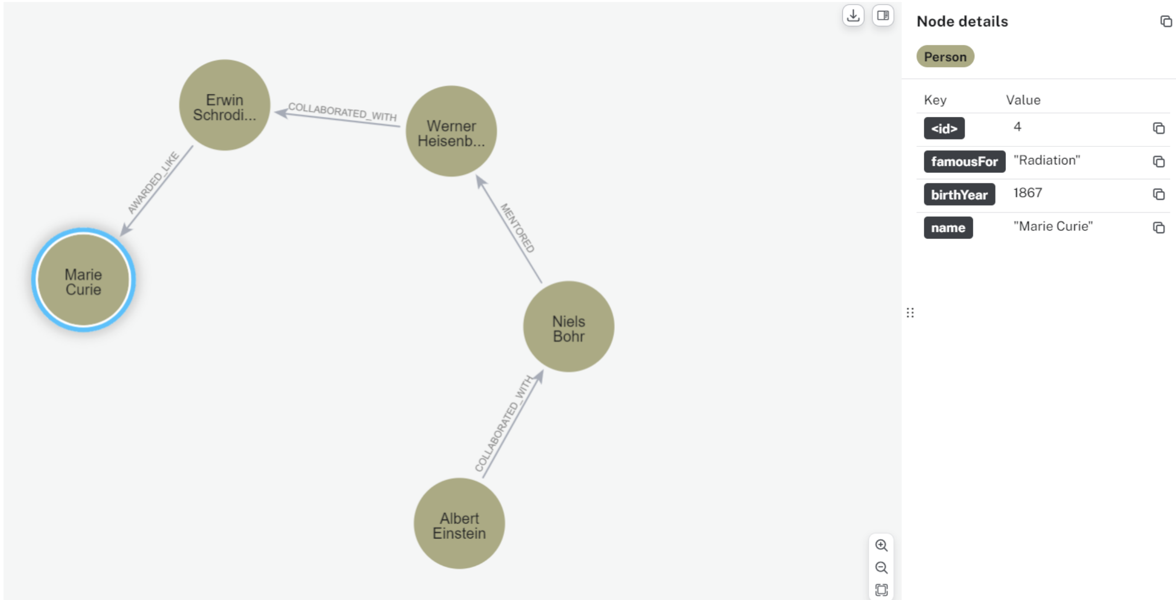
We will use AuraDB which offers Free Tier option

- Register for a new Neo4J account via this URL: https://neo4j.com/product/auradb/
- Verify Your Email. Upon verifying your email (on the same tab of email verification), navigate to the console page, accept the privacy policy, and sign up for a free tier.
- Download the Instance Password as a text file to your machine.
- Open the Instance.
- Click Connect (and save the connection URI).

Cypher

- Cypher streamlines querying by focusing on what to retrieve or create, abstracting the how for seamless interaction with the graph.
- Cypher uses intuitive symbols—parentheses for nodes and arrows for relationships—offering a clear, visual representation of graph structures
- Check Cypher documentation here

Example: Track Inter-relationships between Select Persons



Graph diagram for the target entities and inter-relationships

Create a new Graph

In [ ]:
```
CREATE (einstein:Person {name: "Albert Einstein", birthYear: 1879, famousFor: "Physics"}
       (bohr:Person {name: "Niels Bohr", birthYear: 1885, famousFor: "QuantumMechanics"}
       (heisenberg:Person {name: "Werner Heisenberg", birthYear: 1901}),
       (shchrodinger:Person {name: "Erwin Schrodinger", birthYear: 1887}),
       (curie:Person {name: "Marie Curie", birthYear: 1867, famousFor: "Radiation"}),

       (einstein)-[:COLLABORATED_WITH {domain:"QuantumMechanics"}]-> (bohr),
       (bohr)-[:MENTORED]-> (heisenberg),
       (heisenberg)-[:COLLABORATED_WITH]-> (shchrodinger),
       (shchrodinger)-[:AWARDED_LIKE]-> (curie);
```

Display All Nodes in a Graph

In [ ]:
```
MATCH (n)
RETURN n;
```

Filter by the Existence of an Attribute

Display Famous Persons

In [ ]:
```
MATCH (n:Person)
WHERE (n.famousFor IS NOT NULL)
RETURN n;
```

Retrieve an Attribute for a Matching Node

Find The Birth Year for Albert Einstein

In [ ]:
```
MATCH (n:Person {name: "Albert Einstein"})
RETURN n.birthYear;
```

Find the Nodes with a Specific Relationship

Find all the mentors in our Graph

In [ ]:
```
MATCH (n:Person) -[:MENTORED]-> (m:Person)
Return n
```

Filter Nodes Based on Relationship Attributes

Find all the collaborators in the Quantum Mechanics Domain

In [ ]:
```
MATCH (n:Person) -[collaboration:COLLABORATED_WITH]-> (m:Person)
WHERE collaboration.domain = "QuantumMechanics"
Return n,m
```

Insert a new Relationship in a Graph

Add New Relationships between Curie and Einstein

In [ ]:
```
MATCH (n1:Person {name: "Marie Curie"}), (n2:Person {name: "Albert Einstein"})
CREATE (n1)-[:AWARDED_LIKE]->(n2);
```

Delete all the Nodes in a Graph

In [ ]:
```
MATCH (n)
DETACH DELETE n;
```

Neo4J in Python

Install Dependencies

In [ ]:
```
!pip install neo4j
```

In [ ]:
```
from neo4j import GraphDatabase

class Neo4JConnection:
    def __init__(self, uri, user, password):
        """Initialize the Neo4j connection with URI, username, and password."""
        self.driver = GraphDatabase.driver(uri, auth=(user, password))

    def close(self):
        """Close the Neo4j connection."""
        if self.driver:
            self.driver.close()

    def execute_query(self, query, parameters=None):
        """Execute a Cypher query and return the result."""
        with self.driver.session() as session:
            result = session.run(query, parameters)
            return result.data()
```

In [ ]:
```
# Function to create nodes and relationships as per Query 1
def create_nodes_and_relationships(connection):
    query = """
    CREATE (einstein:Person {name: "Albert Einstein", birthYear: 1879, famousFor: "Physi
           (bohr:Person {name: "Niels Bohr", birthYear: 1885, famousFor: "QuantumMechani
           (heisenberg:Person {name: "Werner Heisenberg", birthYear: 1901}),
           (shchrodinger:Person {name: "Erwin Schrodinger", birthYear: 1887}),
           (curie:Person {name: "Marie Curie", birthYear: 1867, famousFor: "Radiation"})

           (einstein)-[:COLLABORATED_WITH {domain:"QuantumMechanics"}]-> (bohr),
           (bohr)-[:MENTORED]-> (heisenberg),
           (heisenberg)-[:COLLABORATED_WITH]-> (shchrodinger),
           (shchrodinger)-[:AWARDED_LIKE]-> (curie);
    """
    connection.execute_query(query)
    print("Nodes and relationships created.")
```

In [ ]:
```
# Function for adding a new relationship between Marie Curie and Albert Einstein
def add_awarded_like_relationship(connection):
    query = """
    MATCH (n1:Person {name: "Marie Curie"}), (n2:Person {name: "Albert Einstein"})
    CREATE (n1)-[:AWARDED_LIKE]->(n2);
    """
    connection.execute_query(query)
    print("AWARDED_LIKE relationship created between Marie Curie and Albert Einstein.")
```

In [ ]:

```python
# Function for returning people who mentored others
def get_mentors(connection):
    query = """
    MATCH (n:Person) -[:MENTORED]-> (m:Person)
    RETURN n;
    """
    result = connection.execute_query(query)
    for record in result:
        print(record)
```

In [ ]:

```python
# Function for deleting all nodes
def delete_all_nodes(connection):
    query = """
    MATCH (n)
    DETACH DELETE n;
    """
    connection.execute_query(query)
    print("All nodes deleted.")
```

In [ ]:

```python
def main():
    # Replace with your actual Neo4j AuraDB credentials
    uri = "neo4j+s://a3349088.databases.neo4j.io"
    user = "neo4j"
    password = "DNmGymR3u9A9Z95wu0-739aoziP8Qqb5i-r-7apmAM4"

    # Initialize Neo4j connection
    neo4j_conn = Neo4JConnection(uri, user, password)

    try:
        # Execute the queries
        delete_all_nodes(neo4j_conn)
        create_nodes_and_relationships(neo4j_conn)
        add_awarded_like_relationship(neo4j_conn)
        get_mentors(neo4j_conn)
        delete_all_nodes(neo4j_conn)
    finally:
        # Close the connection when done
        neo4j_conn.close()


if __name__ == "__main__":
    main()
```

Readings

- Usefule Tools for Neo4J: https://neo4j.com/docs/tools/
- More on Creating Nodes from here