MASTER LEVEL PROJECT

PYGRAPHDB: HARNESSING PYTHON AND NEO4J FOR DYNAMIC DATA MODELING

SOHEIL NEZAKAT | TU CLAUSTHAL | APRIL 2024

ABSTRACT

This project, titled "PyGraphDB: Harnessing Python and Neo4j for Dynamic Data Modeling," focused on developing a graph database system using Neo4j, facilitated by Python programming. The main goal was to create a flexible and powerful database capable of managing complex relationships between data, which is crucial for understanding and analyzing information effectively.

Using Python's versatile programming capabilities alongside the specialized features of Neo4j, a graph database, we designed a system that allows for easy data management and efficient retrieval of information through complex queries. The project involved setting up a database schema that mirrors real-world interactions and using Python to handle operations like adding new data, updating existing information, and executing search queries.

The outcome of this project is a user-friendly graph database system that efficiently organizes data and speeds up information retrieval. This system is particularly useful for applications that require a deep understanding of data connections, such as social networks, recommendation engines, and other data-intensive applications. The project demonstrated the effective use of Python and Neo4j in creating a scalable and adaptable database solution and laid a strong foundation for further development in the field of graph databases.

INTRODUCTION

Background

In our digital world, we often need to understand how different pieces of information are connected, like how friends are linked in a social network. Traditional ways of storing data can't always handle these connections very well. This project uses a special kind of database called Neo4j, which is great at showing how things are linked, and Python, a programming language that makes it easy to work with data. The goal was to combine Python and Neo4j to build a system that helps us see and understand these connections easily and quickly. This system can be really useful in areas where understanding relationships in data is important, like in social media or when recommending products to users.

Problem statement

In a university environment, managing and understanding the complex relationships between students, courses, faculty, and departments is crucial but challenging. Traditional databases struggle to efficiently map and query these relationships, which can hinder academic advising, course scheduling, and research collaboration. For instance, it can be difficult to quickly identify which students are taking multiple courses from the same instructor, or how different departments interact through interdisciplinary programs. Our project aimed to tackle these issues by developing a database system specifically designed for the university's needs. By utilizing Neo4j, a graph database adept at handling complex relationships, combined with Python's powerful data manipulation capabilities, we created a system that can clearly and effectively visualize and manage the intricate web of connections within a university. This new system promises to enhance administrative efficiency and academic planning, providing deep insights that were previously difficult to obtain.

Project objectives

The primary objective of this project was to develop a dynamic, scalable, and comprehensive graph database for a university setting, utilizing Neo4j to manage complex data relationships effectively. A key goal was to automate the database creation process through Python scripts, enabling the system to dynamically adapt and scale as new data is introduced, or existing data is modified. This automation ensures that the database can effortlessly grow and adjust without manual intervention, accommodating changes such as new courses, student enrollments, or faculty assignments. Additionally, the database needed to be comprehensive, meaning it should accurately represent all pertinent relationships and data points relevant to the university's operational needs. The system’s design was aimed at providing clear, actionable insights into the interconnected nature of academic and administrative data, facilitating better decision-making and more efficient management of university resources.

METHODOLOGY

To achieve our project goals, we employed a structured methodology using Python and Neo4j. Initially, we designed a comprehensive schema that accurately represents the relationships within a university, such as connections between students, courses, and lecturer. Using Python, we developed scripts to automate the creation and updating of this graph database, ensuring it could dynamically adapt to changes in university data. These scripts handle data ingestion, integration, and transformation efficiently, making the system both dynamic and scalable. Throughout the development, we continuously tested the database with various data scenarios to ensure its robustness and reliability in a real-world university setting.

Technology Selection and Setup:

We opted for Python 3.11.5 for its latest features and robustness in handling data operations, particularly those involving complex data structures like graphs. Python scripts were developed within Microsoft Visual Studio Code, a popular Integrated Development Environment (IDE) known for its extensive support for Python and seamless integration with version control systems. This setup allowed us to leverage advanced coding tools and extensions tailored for Python development, enhancing productivity, and ensuring code quality.

To manage the relationships and entities within the university, we used Neo4j 5.18.1, a powerful graph database that excels in storing and querying connected data. The local deployment of Neo4j was done via the Neo4j Desktop application, which provided a user-friendly interface for database management and visualization. This setup not only facilitated the development and testing of our database model but also allowed us to utilize Neo4j's robust querying capabilities.

Python Libraries and Neo4j Plugins:

Key Python libraries were integrated to streamline the process of data manipulation and interaction with the Neo4j database. Libraries such as neo4j, which provides tools for interfacing with Neo4j from Python scripts, were crucial for executing Cypher queries and managing database transactions efficiently. Additionally, we used Neo4j plugins to extend the database functionality, enabling more complex queries and data analytics directly within the Neo4j environment.

Database Schema and Structure:

The schema of our graph database was intricately designed to mirror the organizational structure and academic processes of a university. Our database includes the following nodes:

Student: Represents the individuals enrolled in the university.

Course: Courses offered by the university.

Lecturer: Faculty members who deliver lectures for courses.

Tutor: Teaching assistants or tutors associated with specific courses.

Assignments: Academic tasks assigned within courses.

Department: Academic departments within the university.

Institute: Higher organizational units that encompass several departments.

The relationships between these nodes were carefully modeled to represent real-life academic interactions:

enrolled_in: Links students to the courses they are taking.

assignment_of: Connects assignments to the courses they are part of.

tutor_of: Associates tutors with the courses they assist in.

lecturer_of: Connects lecturers to the courses they teach.

published_by: Shows which department offers a particular course.

hiwi_of: Represents tutors who work within a specific department.

work_for: Links lecturers to their respective departments.

hired_by: Connects lecturers to the institutes that employ them.

belongs_to: Demonstrates the affiliation of departments to a higher institute.

This structure not only facilitates effective data retrieval and visualization but also supports complex queries that mirror multifaceted interactions within a university. This dynamic and scalable database solution is designed to grow and evolve, accommodating changes in the university's structure and offerings without the need for manual updates or restructuring.

To establish the foundational components of our graph database, we first downloaded the free version of Neo4j from the official Neo4j website. This ensured we had a legitimate and stable version of the software suited for our developmental needs. Following the download, we proceeded with the installation of Neo4j on our local machine, a straightforward process that integrates the database management system (DBMS) into our development environment. Once installed, we created a new database using version 5.18.1 of Neo4j, which is the latest at the moment, offering advanced features and improved performance.

For the database naming, we chose "TUC" (Technische Universität Clausthal), although this name can be customized based on the preferences or requirements of the deployment experts. Additionally, to enhance the database's functionality and allow for advanced data manipulation and analysis, we installed the APOC (Awesome Procedures On Cypher) library. APOC is a collection of pre-built procedures and functions that extend the capabilities of Neo4j, enabling us to perform complex operations more efficiently. By incorporating APOC, we significantly boosted the potential of our database, allowing us to leverage a wide range of additional features that are not available in the standard Neo4j setup.

In this project, Python served as the key tool for scripting and managing interactions with our Neo4j database, using Microsoft Visual Studio Code (VSCode) as the development environment. To facilitate these interactions, we installed several Python packages that were critical for our objectives. The neo4j package from the Neo4j Python driver was essential as it allowed us to create a Python class specifically designed for connecting to and manipulating the Neo4j database. This setup enabled our scripts to execute queries, update the graph, and manage transactions seamlessly.

Additionally, we used the pandas library for its powerful data manipulation features, allowing us to read and transform local data files efficiently, which is crucial for feeding data into the Neo4j database dynamically. The time library was incorporated into our scripts to manage operation timing and provide delays when necessary, ensuring that our data operations were synchronized and executed smoothly. We also included the random library to introduce randomness into our scripts, which was particularly useful for simulating real-world scenarios and testing the robustness of our database under various data conditions.

By integrating these packages, our Python scripts became highly dynamic, capable of handling complex data interactions and providing the flexibility needed to adapt to any changes in the database structure or data itself. This approach not only enhanced the functionality of our Neo4j database but also ensured that our system could operate effectively in a real-world university environment.

One significant challenge we encountered in the development process was establishing a robust and scalable connection to the Neo4j database that would not only support our current project requirements but also accommodate potential future expansion as the project grows. Ensuring that the connection method could handle increased data loads and more complex queries without performance degradation was crucial. Among the different connection methods available for

interacting with Neo4j, we explored using the direct driver connection, which allows for session-based interactions with the database.

After evaluating our options, we decided to implement a custom class using the GraphDatabase driver imported from the neo4j package in Python. This approach enabled us to encapsulate the connection logic within a class structure, making our code cleaner, more maintainable, and easier to scale. The class provided methods to connect to the database, perform transactions, and disconnect safely, ensuring efficient management of database sessions. This setup not only addressed our immediate connectivity needs but also laid a foundation for handling more extensive data interactions, which could be crucial as the database grows in complexity and size. However, perfecting this setup required careful handling of connection pools and session management to prevent resource leaks and ensure optimal performance.

IMPLEMENTATION

In the implementation phase, we prioritized a dynamic approach to effectively manage and utilize our data. To begin, we compiled a comprehensive basket of raw data, which served as the foundation for all subsequent data manipulations. This data was meticulously organized into an Excel file where each property was placed in its own sheet. This organization not only facilitated easy access and manipulation of data properties but also allowed for future expansion as needed.

From this collected data, we created a base node named 'namelist' within our Neo4j database. This node encompassed all the raw data that would later be used as properties for other nodes in the database, such as 'student', where attributes like first name and last name were to be randomly selected to ensure diversity and realism in our data modeling.

As the project progressed, we utilized the properties stored in the 'namelist' base node to dynamically create other nodes within the database. This was achieved through various methods implemented in Python, where we constructed precise Cypher queries to be executed within the Neo4j database. Additionally, to enhance flexibility and maintainability of our code, we stored some of these Cypher queries in a text file. This file was then read by our Python script, which executed the stored queries on the Neo4j database. This method not only streamlined the process of node creation but also allowed for easy updates and modifications to our database queries, supporting the dynamic nature of our project's requirements.

Before exploring the intricacies of our Python script, it is essential to understand its pivotal role in the project. The script serves as the backbone of our database operations, seamlessly bridging the gap between raw data management and dynamic database interactions. Developed in a robust and flexible manner, it enables the precise execution of complex tasks essential for maintaining the integrity and utility of our Neo4j graph database.

key libraries are imported to enhance functionality and efficiency. The neo4j package is crucial for establishing a connection with the Neo4j database, enabling the execution of Cypher queries and transaction management. pandas is employed for its powerful data handling capabilities, allowing for efficient reading, cleaning, and preparation of data from various sources. The time module is used to manage operation timing and ensure smooth execution of tasks by adding necessary delays. Lastly, the random module introduces randomness into the process, which is particularly useful for creating realistic test scenarios and data variability. Together, these libraries equip the script to effectively manage data interactions and optimize the performance of database operations.

```
from neo4j import GraphDatabase
import pandas as pd
import time
import random
```

The Neo4jConnection class manages interactions with a Neo4j database. It initializes the database connection using credentials and URI, handles errors during driver setup, and provides methods for executing and closing queries. The constructor sets up the connection, while the close method ensures resources are properly released. The query method executes queries within managed sessions, handling exceptions and session closures efficiently. Additionally, the run_query method offers a straightforward way to execute queries, automatically managing the session lifecycle for simpler and more robust database operations.

```
class Neo4jConnection:

    def __init__(self, uri, user, password):
        self.__uri = uri
        self.__user = user
        self.__password = password
        self.__driver = None
```

```
        try:
            self.__driver = GraphDatabase.driver(
                self.__uri, auth=(self.__user, self.__password)
            )
        except Exception as e:
            print("Failed to create the driver:", e)

    def close(self):
        if self.__driver is not None:
            self.__driver.close()

    def query(self, query, parameters=None, db=None):
        assert self.__driver is not None, "Driver not initialized!"
        session = None
        response = None
        try:
            session = (
                self.__driver.session(database=db)
                if db is not None
                else self.__driver.session()
            )
            response = list(session.run(query, parameters))
        except Exception as e:
            print("Query failed:", e)
        finally:
            if session is not None:
                session.close()
        return response

    def run_query(self, query, parameters=None):
        with self.driver.session() as session:
            result = session.run(query, parameters)
            return result
```

The script sets up a connection to a Neo4j graph database using specified URI, user, and password credentials. It initializes an instance of the Neo4jConnection class to manage this connection. Additionally, it employs the GraphDatabase.driver for direct driver access. A query is executed through the conn instance to delete all nodes and their relationships in the database using the apoc.periodic.iterate method. This method batches the deletion process for efficiency, processing nodes in groups of 1000, ensuring the operation's performance is optimized while managing larger volumes of data.

```
uri = "bolt://localhost:7687"
user = "neo4j"
password = "123456789"
conn = Neo4jConnection(uri=uri, user=user, password=password)
driver = GraphDatabase.driver(uri, auth=(user, password))
remove = conn.query(
     "CALL apoc.periodic.iterate('MATCH (n) RETURN n', 'DETACH DELETE n', {batchSize:1000,
iterateList:true})"
```

```
)
```

By using sleep for 5 seconds we can view in our database, that the all previous data are deleted.

```
time.sleep(5)
```

In this implementation, a method has been developed to ascertain the desired quantity of nodes, specifically for entities such as Student, Lecturer, and Tutor nodes. The method stipulates a minimum threshold of 10 nodes and a maximum limit of 1000 nodes; these parameters are adjustable to accommodate varying requirements. This uniform approach is applied across different node types to maintain consistency in the data collection process.

```
while True:
    try:
        j = int(
            input(
                "How many students node would you like to be created? Please enter an integer number greater than 10 less than 1000: "
            )
        )
        if 1000 > j > 10:
            break
        else:
            print("The number must be in the range. Please try again.")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")
```

In below script. Excel file get read by pandas. The excel file contains all the necessary raw data, and store all the sheets name. Therefore, as how many as you would like, you can add new sheet and data in excel file, without major touch in the script.

```
xls = pd.ExcelFile("D:\\VSCode\\Neo4j project\\raw_data.xlsx")
sheet_names = xls.sheet_names
```

The script iterates over each sheet in an Excel file, reading the contents of the first column into a DataFrame using Pandas. It then converts the DataFrame column into a list and dynamically creates a global variable with a name based on the sheet name. This variable holds the list of values from the corresponding sheet, with spaces replaced by underscores and all characters converted to lowercase for consistency.

```
for sheet_name in sheet_names:
    df = pd.read_excel(xls, sheet_name=sheet_name, usecols=[0], header=None)
    globals()[sheet_name.replace(" ", "_").lower() + "_list"] = df.iloc[:, 0].tolist()
```

In this part of python script, we executing a cypher which creates our base node, including all the raw data, fetched from excel file. The parameters in the cypher are replaced by the lists, created from excel file.

```
with driver.session() as session:
    result = session.run(
          "CREATE (nl:NameList) SET nl.institute = $instituteList , nl.Dept= $department , nl.coursename3= $Course_Politics ,nl.coursename2 = $Course_Math , nl.coursename = $Course_Computer ,nl.semester = $semester , nl.firstNames = $First_Name , nl.lastNames = $Last_Name , nl.email = $Email, nl.city = $City , nl.street= $Street , nl.housn= $House_no , nl.gender = $Gender , nl.specialization = $Specialization, nl.assignment = $Assignment ",
        instituteList=institute_list,
        department=department_list,
        Course_Politics=course_politics_list,
        Course_Math=course_math_list,
        Course_Computer=course_computer_list,
        semester=semester_list,
        First_Name=first_name_list,
        Last_Name=last_name_list,
        Email=email_list,
        City=city_list,
        Street=street_list,
        House_no=house_no_list,
        Gender=gender_list,
        Specialization=specialization_list,
        Assignment=assignment_list,
    )
```

The script engages in the process of file reading, specifically from a designated file containing Cypher queries, with a strict adherence to "read-only" mode. In tandem with file access, leading and trailing whitespaces within the queries are systematically removed to ensure data integrity and fidelity. Subsequently, a line counter mechanism is deployed to discern the current line position within the file, a pivotal aspect given the one-to-one correspondence between each Cypher query and its respective line. Integral to this operation is the dynamic substitution of the user-specified number of desired nodes into the queries, thereby customizing the queries according to the user's input. Finally, the script executes the Cypher queries, conditioned upon the non-emptiness of the respective line, thus orchestrating the precise execution of each query within the file.

```
try:
    with open("D:\\VSCode\\Neo4j project\\cyphers.txt", "r") as file:
        line_count = 0
        for line in file:
            line = line.strip()
            if line:
                if line_count == 0:
                    line = line.replace("=x", f"{j}")
                    print(line)
                elif line_count == 1:
                    line = line.replace("=y", f"{i}")
                    print(line)
                elif line_count == 2:
                    line = line.replace("=z", f"{k}")
                    print(line)
                line_count += 1
                result = conn.query(line)
```

```
finally:
    print("Created successfully")
```

To ensure the comprehensive integration of robustness and randomness into our system, a sophisticated method is employed to establish relations randomly. This method transcends conventional iterative approaches, incorporating the utilization of Python's built-in random function. It dynamically determines the number of relations created within the Neo4j database, enhancing the versatility and adaptability of the system. Specifically, this intricate algorithm is applied to govern the creation of relations such as 'ENROLLED_IN', 'LECTURER_OF', 'TUTOR_OF', and 'ASSIGNMENT_OF', ensuring a nuanced and dynamic relational framework within the database.

```
while True:
        result = session.run(
            "MATCH ()-[s:ENROLLED_IN]->() RETURN count(s) AS NumberOfStudents"
        )
        number_of_enrolled_rel = result.single().value()
        print(number_of_enrolled_rel)
        if (number_of_enrolled_rel) == 450:
            print(f"You entered: {number_of_enrolled_rel}")
            break
        result = session.run(
            "MATCH (s:Student {studentID: $student_id})  MATCH (n:Course {courseID: $course_id
}) MERGE (s)-[:ENROLLED_IN]->(n)",
            student_id=random.randint(10010001, 10010000 + j),
            course_id=random.randint(4001001, 4001200),
        )
```

The other relations, are created, by a fixed cypher, by containing a feature like, that create a relation between to content-related nodes. E.g.:

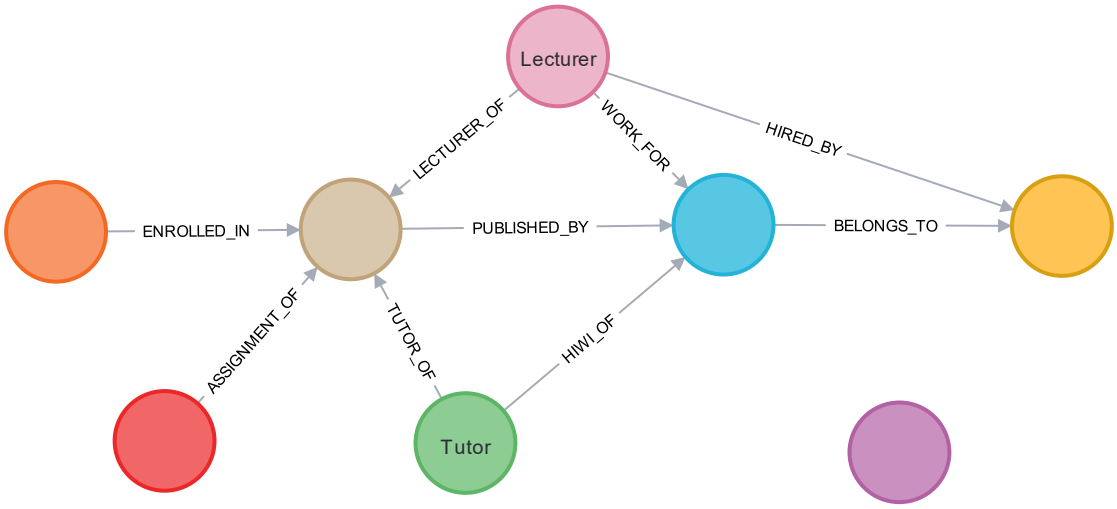
```
result = session.run(

                  "match    (s:Student)-[r:ENROLLED_IN]-(c:Course),    (d:Department)    where
c.department=d.departmentname merge (c)-[:PUBLISHED_BY]-(d)"
    )
```

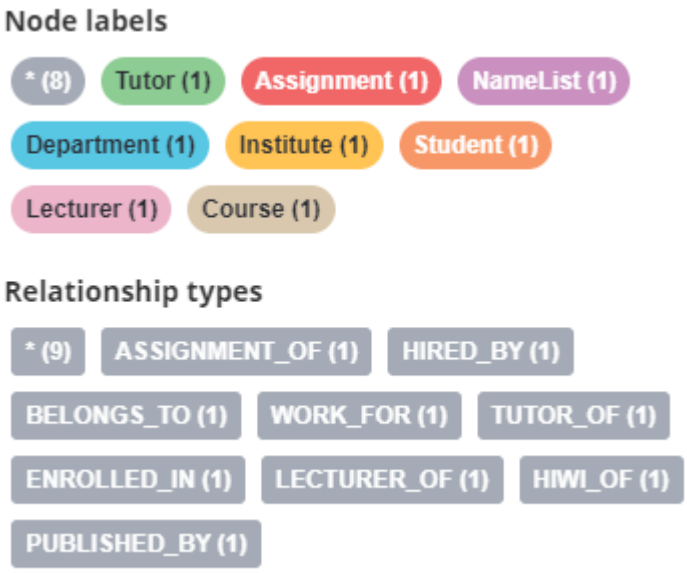
At the last couple of lines, connection and driver get closed.

```
driver.close()
conn.close()
```

DATA MODEL

The DBMS schema:





In below nodes and their properties are listed.

Student [dateofbirth, last name, phone, address, studentid, email, gender, firstnname, enrollmentdate]

Lecturer [lastname, qualification, lecturerid, department, hiredate, email, gender, firstname, specialization]

Tutor [lastname, tutored, department, hiredate, email, gender, firstname]

Course [coursed, department, credit, coursename, semester]

Assignment [assignment_title, assignmentid, credit, semester]

Department [departmentname, departmentid]

Institute [instituteid, institutename]

PYTHON SCRIPT

```
from neo4j import GraphDatabase
import pandas as pd
import time
import random


class Neo4jConnection:

    def __init__(self, uri, user, password):
        self.__uri = uri
        self.__user = user
        self.__password = password
        self.__driver = None
        try:
            self.__driver = GraphDatabase.driver(
                self.__uri, auth=(self.__user, self.__password)
            )
        except Exception as e:
            print("Failed to create the driver:", e)

    def close(self):
        if self.__driver is not None:
            self.__driver.close()

    def query(self, query, parameters=None, db=None):
        assert self.__driver is not None, "Driver not initialized!"
        session = None
        response = None
        try:
            session = (
                self.__driver.session(database=db)
                if db is not None
                else self.__driver.session()
            )
            response = list(session.run(query, parameters))
        except Exception as e:
            print("Query failed:", e)
        finally:
            if session is not None:
                session.close()
        return response

    def run_query(self, query, parameters=None):
        with self.driver.session() as session:
            result = session.run(query, parameters)
            return result
```

```
uri = "bolt://localhost:7687"
user = "neo4j"
password = "123456789"
conn = Neo4jConnection(uri=uri, user=user, password=password)
driver = GraphDatabase.driver(uri, auth=(user, password))
remove = conn.query(
     "CALL  apoc.periodic.iterate('MATCH  (n)  RETURN  n',  'DETACH  DELETE  n',  {batchSize:1000,
iterateList:true})"
)

time.sleep(5)

while True:
    try:
        j = int(
            input(
                "How many students node would you like to be created? Please enter an integer
number greater than 10 less than 10000: "
            )
        )
        if 10000 > j > 10:
            break
        else:
            print("The number must be in the range. Please try again.")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")

while True:
    try:
        i = int(
            input(
                "How many lecturer node would you like to be created? Please enter an integer
number greater than 5 and less than 1000: "
            )
        )
        if 1000 > i > 5:
            break
        else:
            print("The number must be in the range. Please try again.")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")

while True:
    try:
        k = int(
            input(
                 "How many tutor node would you like to be created? Please enter an integer
number greater than 2 and less than 100: "
            )
```

```
        )
        if 100 > k > 2:
            break
        else:
            print("The number must be in the range. Please try again.")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")


xls = pd.ExcelFile("D:\\VSCode\\Neo4j project\\raw_data.xlsx")
sheet_names = xls.sheet_names

for sheet_name in sheet_names:
    df = pd.read_excel(xls, sheet_name=sheet_name, usecols=[0], header=None)
    globals()[sheet_name.replace(" ", "_").lower() + "_list"] = df.iloc[:, 0].tolist()

with driver.session() as session:
    result = session.run(
          "CREATE (nl:NameList) SET nl.institute = $instituteList , nl.Dept= $department ,
nl.coursename3=  $Course_Politics  ,nl.coursename2  =  $Course_Math  ,  nl.coursename  =
$Course_Computer ,nl.semester = $semester , nl.firstNames = $First_Name , nl.lastNames =
$Last_Name , nl.email = $Email, nl.city = $City , nl.street= $Street , nl.housn= $House_no ,
nl.gender = $Gender , nl.specialization = $Specialization, nl.assignment = $Assignment ",
        instituteList=institute_list,
        department=department_list,
        Course_Politics=course_politics_list,
        Course_Math=course_math_list,
        Course_Computer=course_computer_list,
        semester=semester_list,
        First_Name=first_name_list,
        Last_Name=last_name_list,
        Email=email_list,
        City=city_list,
        Street=street_list,
        House_no=house_no_list,
        Gender=gender_list,
        Specialization=specialization_list,
        Assignment=assignment_list,
    )

try:
    with open("D:\\VSCode\\Neo4j project\\cyphers.txt", "r") as file:
        line_count = 0
        for line in file:
            line = line.strip()
            if line:
                if line_count == 0:
                    line = line.replace("=x", f"{j}")
                    print(line)
```

```
                elif line_count == 1:
                    line = line.replace("=y", f"{i}")
                    print(line)
                elif line_count == 2:
                    line = line.replace("=z", f"{k}")
                    print(line)
                line_count += 1
                result = conn.query(line)
finally:
    print("Created successfully")

with driver.session() as session:
    while True:
        result = session.run(
            "MATCH ()-[s:ENROLLED_IN]->() RETURN count(s) AS NumberOfStudents"
        )
        number_of_enrolled_rel = result.single().value()
        print(number_of_enrolled_rel)
        if (number_of_enrolled_rel) == 450:
            print(f"You entered: {number_of_enrolled_rel}")
            break
        result = session.run(
            "MATCH (s:Student {studentID: $student_id})  MATCH (n:Course {courseID: $course_id
}) MERGE (s)-[:ENROLLED_IN]->(n)",
            student_id=random.randint(10010001, 10010000 + j),
            course_id=random.randint(4001001, 4001200),
        )

    while True:
        result = session.run(
            "MATCH ()-[s:LECTURER_OF]->() RETURN count(s) AS NumberOfStudents"
        )
        number_of_lecturer_of = result.single().value()
        print(number_of_lecturer_of)
        if (number_of_lecturer_of) == 20:
            print(f"You entered: {number_of_lecturer_of}")
            break
        result = session.run(
                "MATCH (l:Lecturer {lecturerID: $lecturer_id})  MATCH (n:Course {courseID:
$course_id }) MERGE (l)-[:LECTURER_OF]->(n)",
            lecturer_id=random.randint(2001001, 2001000 + i),
            course_id=random.randint(4001001, 4001200),
        )

    while True:
        result = session.run(
            "MATCH ()-[s:TUTOR_OF]->() RETURN count(s) AS NumberOfStudents"
        )
        number_of_tutor_of = result.single().value()
```

```
        print(number_of_tutor_of)
        if (number_of_tutor_of) == 8:
            print(f"You entered: {number_of_tutor_of}")
            break
        result = session.run(
             "MATCH (t:Tutor {tutorID: $tutor_id})  MATCH (n:Course {courseID: $course_id }) MERGE (t)-[:TUTOR_OF]->(n)",
            tutor_id=random.randint(300101, 300100 + k),
            course_id=random.randint(4001001, 4001200),
        )

    while True:
        result = session.run(
            "MATCH ()-[s:ASSIGNMENT_OF]->() RETURN count(s) AS NumberOfStudents"
        )
        number_of_assignment_of = result.single().value()
        print(number_of_assignment_of)
        if (number_of_assignment_of) == 45:
            print(f"You entered: {number_of_assignment_of}")
            break
        result = session.run(
             "MATCH (a:Assignment {assignmentID: $assignment_id})  MATCH (n:Course {courseID: $course_id }) where a.semester=n.semester MERGE (a)-[:ASSIGNMENT_OF]->(n)",
            assignment_id=random.randint(7001001, 7001050),
            course_id=random.randint(4001001, 4001200),
        )

    result = session.run(
                  "match    (s:Student)-[r:ENROLLED_IN]-(c:Course),    (d:Department)    where c.department=d.departmentname merge (c)-[:PUBLISHED_BY]-(d)"
    )
    result = session.run(
        'MATCH (d:Department {departmentname: "Department of Sociology"})  MATCH (i:Institute {institutename: "Institute of Advanced Sciences and Humanities" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
          'MATCH (d:Department {departmentname: "Department of Political Science"})  MATCH (i:Institute {institutename: "Institute of Biological and Environmental Sciences" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
         'MATCH (d:Department {departmentname: "Department of Biology"})  MATCH (i:Institute {institutename: "Institute of Economic and Business Analysis" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
        'MATCH (d:Department {departmentname: "Department of Economics"})  MATCH (i:Institute {institutename: "Institute of Psychological Sciences" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
```

```
    result = session.run(
        'MATCH (d:Department {departmentname: "Department of Psychology"})  MATCH (i:Institute {institutename: "Institute of Literature and Cultural Studies" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
         'MATCH (d:Department {departmentname: "Department of English Literature"})  MATCH (i:Institute {institutename: "Institute of Historical and Anthropological Research" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
        'MATCH (d:Department {departmentname: "Department of History"})  MATCH (i:Institute {institutename: "Institute of Technology and Informatics" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
         'MATCH (d:Department {departmentname: "Department of Computer Science"})  MATCH (i:Institute {institutename: "Institute of Advanced Sciences and Humanities" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
        'MATCH (d:Department {departmentname: "Department of Mathematics"})  MATCH (i:Institute {institutename: "Institute of Theoretical and Applied Mathematics" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
        'MATCH (d:Department {departmentname: "Department of Physics"})  MATCH (i:Institute {institutename: "Institute of Global Studies and Governance" }) MERGE (d)-[:BELONGS_TO]->(i)'
    )
    result = session.run(
            "match(l:Lecturer)-[r:LECTURER_OF]-(c:Course)-[p:PUBLISHED_BY]-(d:Department)-[q:BELONGS_TO]-(i:Institute) MERGE (l)-[:HIRED_BY]->(i) MERGE (l)-[:WORK_FOR]-(d)"
    )
    result = session.run(
         "match(t:Tutor)-[r:TUTOR_OF]-(c:Course)-[p:PUBLISHED_BY]-(d:Department) MERGE (t)-[:HIWI_OF]->(d)"
    )

driver.close()
conn.close()
```

CONCLUSION

In conclusion, I am immensely grateful for the invaluable guidance and mentorship provided by Professor Dr. Sven Hartmann throughout the Data courses. Their expertise and encouragement have been instrumental in shaping my understanding and proficiency in database management and graph theory, enabling me to successfully navigate the intricacies of this endeavor. Additionally, I extend my heartfelt appreciation to my supervisors for their unwavering support and insightful feedback, which has been pivotal in overcoming challenges and realizing the project's objectives. Looking ahead, there are exciting prospects for further advancement in this project. Future work could explore the integration of advanced algorithms for data analysis and visualization, harnessing the power of machine learning and graph analytics to extract deeper insights from the database. Additionally, enhancements in user interface design and scalability measures could elevate the usability and efficiency of the system, paving the way for broader applications in academic and research settings. With continued dedication and collaboration, this project has the potential to make significant strides in revolutionizing database management within educational institutions and beyond.