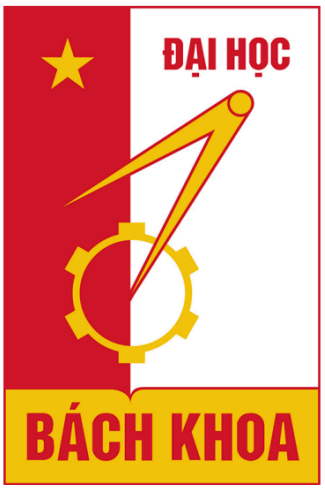


SOICT

REPORT

Project 3 - 733512

NEO4J QUERY SYSTEM USING CHATGPT MODEL

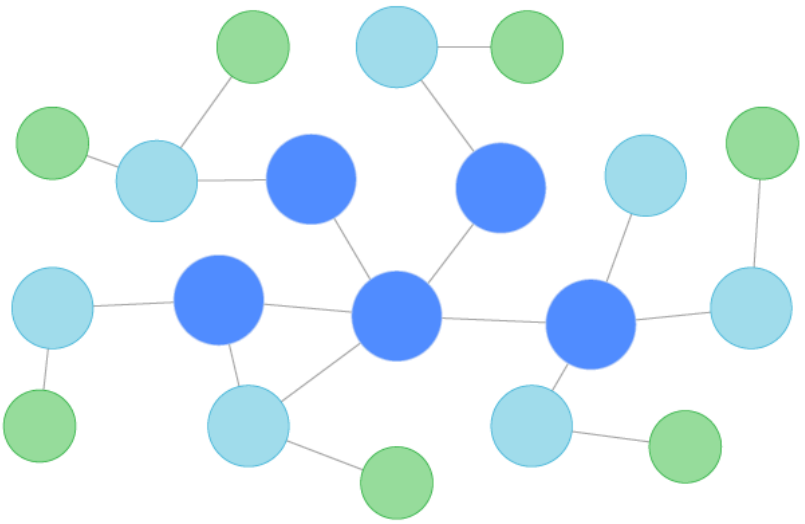
Name: Nguyen Huu Tuan Duy
ID: 20204907
Class: IT-E10-01

Instructor: Assoc. Prof. Dr. Cao Tuan Dung

1. Neo4j

Neo4j is one of the most popular and powerful graph database systems, designed to process and store data in the form of graphs.

1.1. What is a Graph Database?



A graph database is defined as a specialized, single-purpose platform for creating and manipulating graphs. Graphs contain nodes, edges, and properties, all of which are used to represent and store data in a way that relational databases are not equipped to do.

Graph analytics is another commonly used term, and it refers specifically to the process of analyzing data in a graph format using data points as nodes and relationships as edges. Graph analytics requires a database that can

support graph formats; this could be a dedicated graph database or a converged database that supports multiple data models, including graphs. Graph databases address big challenges many of us tackle daily. Modern data problems often involve many-to-many relationships with heterogeneous data that set up needs to:

- Navigate deep hierarchies
- Find hidden connections between distant items
- Discover inter-relationships between items

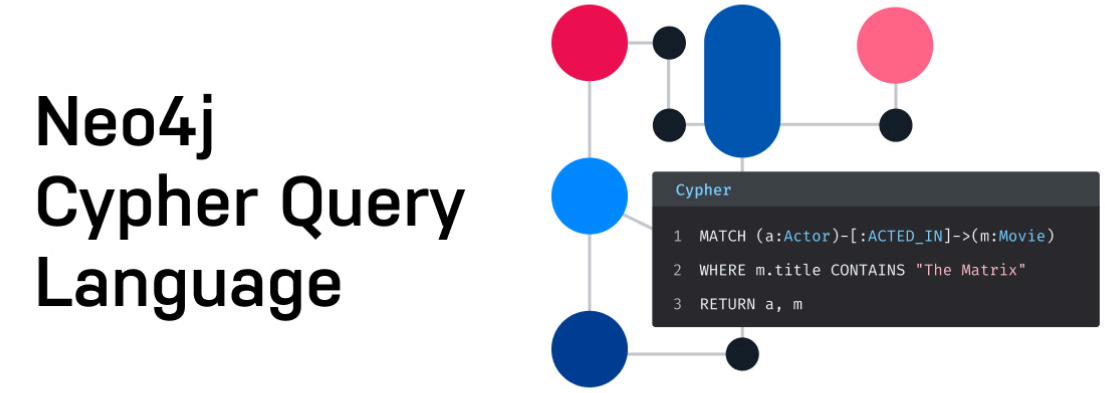
Whether it's a social network, payment network, or road network you'll find that everything is an interconnected graph of relationships. And when we want to ask questions about the real world, many questions are about the relationships rather than about the individual data elements.

In Neo4j, information is organized as nodes, relationships, and properties.

- **Nodes** are the entities in the graph.
    - Nodes can be tagged with labels, representing their different roles in your domain. (For example, Person).
    - Nodes can hold any number of key-value pairs, or properties. (For example, name)
    - Node labels may also attach metadata (such as index or constraint information) to certain nodes.

- **Relationships** provide directed, named, connections between two node entities (e.g. Person LOVES Person).
    - Relationships always have a direction, a type, a start node, and an end node, and they can have properties, just like nodes.
    - Nodes can have any number or type of relationships without sacrificing performance.
    - Although relationships are always directed, they can be navigated efficiently in any direction.

1.2. Cypher Query Language



In Neo4j, we use Cypher Query Language for the graph database. Neo4j is designed to be intuitive and efficient for working with graph data structures, consisting of nodes, relationships, and properties.

a. MATCH

Find nodes with specific properties

```
MATCH (c:City {name: "London"})
RETURN c.population_size;
```

- **MATCH (c: City {name: ”London”})**: the MATCH clause specifies a node pattern with the label **City**, filters the matched results to those with a **name** property with value London and assigns the matches to variable c.
- **RETURN c.population_size**: the RETURN clause is used to request specific results.

Find nodes with specific relationships

```
MATCH (city:City {name:
"London"})-[:IN]->(country:Country)
RETURN country.name;
```

- **MATCH (city: City {name: “London”})-[: IN]->(country: Country)**: the **MATCH** clause specifies a node and relationship pattern with two connected nodes, labeled **City** and **Country**, connected by a relationship of type **IN**.

Match labels

```
MATCH (c:City)
RETURN c;
```

- **MATCH (c:City)**: the MATCH clause specifies a node labeled **City**

Match multiple labels

```
MATCH (c:City:Country)
RETURN c;
```

- **MATCH (c:City:Country)**: the MATCH clause specifies a node labeled both **City** and **Country**

b. OPTIONAL MATCH

The **MATCH** clause can be modified by prepending the **OPTIONAL** keyword. **OPTIONAL MATCH** clause behaves the same as a regular **MATCH**, but when it fails to find the pattern, missing parts of the pattern will be filled with null values.

Get optional relationships

```
MATCH (c1:Country {name: 'France'})OPTIONAL MATCH
(c1)--(c2:Country {name: 'Germany'})
RETURN c2;
```

- Using **OPTIONAL MATCH** when returning a relationship that doesn't exist will return the default value **NULL** instead. The returned property of an optional element that is **NULL** will also be **NULL**

Optional typed and named relationships

The **OPTIONAL MATCH** clause allows you to use the same conventions as **MATCH** when it comes to handling variables and relationship types

```
MATCH (c:Country {name: 'United Kingdom'})OPTIONAL MATCH
(c)-[r:LIVES_IN]->()
RETURN c.name, r;
```

c. WHERE

Specifying properties can also be done with the **WHERE** clause.

Find nodes with specific properties

```
MATCH (c:City)WHERE c.name = "London"
RETURN c.population_size;
```

Find nodes with specific relationships

```
MATCH (city:City)-[:IN]->(country:Country)WHERE
city.name = "London"
RETURN country.name;
```

Match multiple labels

```
MATCH (c)WHERE c:City AND c:Country
RETURN c;
```

Matching nodes with properties in a range

```
MATCH (c:City)WHERE c.population_size >= 1000000 AND
c.population_size <= 2000000
RETURN c;
```

d. WITH

The **WITH** clause is used to chain together parts of a query, piping the results from one to be used as a starting point of criteria in the next query.

Filter on aggregate functions

Aggregated results have to pass through a **WITH** clause if you want to filter them:

```
MATCH (p:Person {name: 'John'})--(person)-->()WITH
person, count(*) AS foafWHERE foaf > 1
RETURN person.name;
```

Sorting unique aggregated results can be done with **DISTINCT** operator in the aggregation function which can be then filtered:

```
MATCH (p:Person {name: 'John'})--(person)-->(m)WITH
person, count(DISTINCT m) AS foafWHERE foaf > 1
RETURN person.name;
```

Sorting results

The **WITH** clause can be used to order results before using **collect()** on them:

```
MATCH (n)WITH n
ORDER BY n.name ASC LIMIT 3
RETURN collect(n.name);
```

If you want to **collect()** only unique values:

```
MATCH (n)WITH n
ORDER BY n.name ASC LIMIT 3
RETURN collect(DISTINCT n.name) as unique_names;
```

Limited path searches

The **WITH** clause can be used to match paths, limit to a certain number, and then match again using those paths as a base:

```
MATCH (p1 {name: 'John'})--(p2)WITH p2
ORDER BY p2.name ASC LIMIT 1
MATCH (p2)--(p3)RETURN p3.name;
```

e. Creating

Create a node

```
CREATE (c:City {name: "Zagreb", population_size:
1000000});
```

- **c:City**: creates a new node with the label **City** and assigns it to variable **c** (which can be omitted if it's not needed).
- **{name: "Zagreb", population_size: 1000000}**: the newly created node has two properties, one with a string value and another with an integer value.

Create nodes with relationships

```
CREATE (c1:City {name: "UK"}),
     (c2:City {name: "London", population_size: 9000000})
     (c1)<-[r:IN]-(c2)
RETURN c1, c2, r;
```

The **CREATE** clause creates two new nodes and a directed relationship between them.

Create a relationship between existing nodes

```
MATCH (c1), (c2)
WHERE c1.name = "UK" AND c2.name = "London"
CREATE (c2)-[:IN]->(c1);
```

This will create a directed relationship of type **IN** between two existing nodes. If such a relationship already exists, this query will result in a duplicate. To avoid this, you can use the **MERGE** clause:

```
MATCH (c1), (c2)WHERE c1.name = "UK" AND c2.name =
"London"
MERGE (c2)-[:IN]->(c1);
```

f. Updating

Add or update node properties

```
MATCH (c:Country {name: "UK"})
SET c.name = "United Kingdom";
```

If you use the **SET** clause on a property that doesn't exist, it will be created.

Replace all node properties

```
MATCH (c:Country)
WHERE c.name = "United Kingdom"
SET c = {name: "UK", population_size: "66650000"};
```

- **SET c = {name: "UK" ...}**: this **SET** clause will delete all existing properties and create the newly specified ones.

Update multiple node properties

```
MATCH (c:Country)
WHERE c.name = "United Kingdom"
SET c += {name: "UK", population_size: "66650000"};
```

- **SET c += {name: "UK" ...}**: this **SET** clause will add new properties and update existing ones if they are already set.

Check if a property exists and update it

```
MATCH (c:Country)
WHERE c.name = "Germany"
AND c.language IS NULL
SET c.language = "German";
```

Because the **WHERE** clause contains the statement **c.language IS NULL**, the node will only be matched if it doesn't have a **language** property.

Rename a property

```
MATCH (c:Country)WHERE c.official_language IS nullSET
c.official_language = c.languageREMOVE c.language;
```

- **WHERE c.official_language IS null**: the **WHERE** clause makes sure that you only create the new property in nodes that don't have a property with the same name.

- **SET n.official_language = n.language**: you are technically not renaming a property but rather creating a new one with a different name and the same value.
- **REMOVE n.language**: the **REMOVE** clause is used to delete the old property.

g. Deleting

Delete a node

```
MATCH (c)-[r]-()
WHERE c.name = "US"
DELETE r, c;
```

- **DELETE r, c**: before you can delete a node, all of its relationships must be deleted first.

This query can be rewritten with the **DETACH** clause to achieve the same result.

```
MATCH (c)WHERE c.name = "US"
DETACH DELETE c;
```

Delete a property

```
MATCH (c:Country)
WHERE c.name = "US" AND c.language IS NOT null
DELETE c.language;
```

This query will delete the property **language** from a specific node.

Delete label in every node

```
MATCH (c)
DELETE c:Country;
```

This query will delete the label **Country** from every node.

Delete one of multiple labels

```
MATCH (c)
WHERE c:Country:CityREMOVE c:City;
```

This will delete the label **City** from every node that has the labels **Country** and **City**.

Delete all nodes and relationships

```
 MATCH (n)
 DETACH DELETE n;
```

This query will delete the whole database.

h. CALL

Cartesian product

CALL subquery is executed once for each incoming row. If multiple rows are produced from the **CALL** subquery, the result is a Cartesian product of results. It is an output combined from 2 branches, one being called the **input branch** (rows produced before calling the subquery), and the **subquery branch** (rows produced by the subquery). Imagine the data includes two **:Person nodes**, one named **John** and one named **Alice**, as well as two **:Animal** nodes, one named **Rex** and one named **Lassie**.

Running the following query would produce the output below:

```
MATCH (p:Person)
CALL {
   MATCH (a:Animal)
   RETURN a.name as animal_name
}
RETURN p.name as person_name, animal_name
```

Output:

person_name	animal_name

'John'	'Rex'
'John'	'Lassie'
'Alice'	'Rex'
'John'	'Rex'

Cartesian products with bounded symbols

To reference variables from the outer scope in the subquery, start the subquery with the **WITH** clause. It allows using the same symbols to expand on the neighborhood of the referenced nodes or relationships. Otherwise, the subquery will behave as it sees the variable for the first time. In the following query, the WITH clause expanded the meaning of the variable person to the node with the label **:Person** matched in the outer scope of the subquery:

```
MATCH (person:Person)
CALL {
   WITH person
   MATCH (person)-[:HAS_PARENT]->(parent:Parent)
   RETURN parent
}
RETURN person.name, parent.name
```

Output:

person_name	parent_name
'John'	'John Sr.'
'John'	'Anna'
'Alice'	'Roxanne'
'Alice'	'Bill'

i. LOAD CSV

The **LOAD CSV** clause enables you to load and use data from a **CSV** file of your choosing in a row-based manner within a query. We support the Excel CSV dialect, as it's the most commonly used one.

The syntax of the clause is:

```
LOAD CSV FROM <csv-location> ( WITH | NO ) HEADER [IGNORE
BAD] [DELIMITER <delimiter-string>] [QUOTE
<quote-string>] [NULLIF <nullif-string>] AS
<variable-name>
```

Below is an example of a query using the **LOAD CSV** clause:

```
LOAD CSV FROM "/people.csv" WITH HEADER AS row
CREATE (p:People) SET p += row;
```

j. INDEXES

- Indexes are not created automatically.
- You can explicitly create indexes on a data with a specific label or label-property combination using the **CREATE INDEX ON** syntax.

Create a label index

To optimize queries that fetch nodes by label, you need to create a label index:

```
CREATE INDEX ON :Person;
```

Creating an index will optimize the following type of queries:

```
MATCH (n:Person) RETURN n;
```

Create a label-property index

To optimize queries that fetch nodes with a certain label and property combination, you need to create a label-property index. For the best performance, create indexes on properties containing unique integer values.

For example, to index nodes that are labeled as **:Person** and have a property named **age**:

```
CREATE INDEX ON :Person(age);
```

Creating an index will optimize the queries that need to match a specific label and property combination:

```
MATCH (n :Person {age: 42}) RETURN n;
```

The index will also optimize queries that filter labels and properties with the **WHERE** clause:

```
MATCH (n) WHERE n:Person AND n.age = 42 RETURN n;
```

Be aware that since the filter inside **WHERE** can contain any kind of an expression, the expression can be so complicated that the index doesn't get used. If there is any suspicion that an index isn't used, we recommend writing labels and properties inside the **MATCH** pattern.

Check indexes

To check all the labels and label-property pairs that Memgraph currently indexes, use the following query:

```
SHOW INDEX INFO;
```

The query displays a table of all label and label-property indexes presently kept by Memgraph, ordered by index type, label, property and count.

Delete an index

Created indexes can be deleted using the following syntax:

```
DROP INDEX ON :Label;

DROP INDEX ON :Label(property);
```

These queries instruct all active transactions to abort as soon as possible. Once all transactions have finished, the index will be deleted.

k. Constraints

Create a uniqueness constraint

```
CREATE CONSTRAINT ON (c:City) ASSERT c.location IS
UNIQUE;
```

This query will make sure that every node with the label **City** has a unique value for the **location** property.

Create an existence constraint

```
CREATE CONSTRAINT ON (c:City)ASSERT exists (c.name);
```

This query will make sure that every node with the label **City** has the property **name**.

Check constraints

```
SHOW CONSTRAINT INFO;
```

This query will list all active constraints in the database.

Drop a uniqueness constraint

```
DROP CONSTRAINT ON (c:City)ASSERT c.location IS UNIQUE;
```

This query will remove the specified uniqueness constraint.

Drop an existence constraint

```
DROP CONSTRAINT ON (c:City)ASSERT exists (c.name);
```

This query will remove the specified existence constraint.

I. Other Useful Cypher Queries

Count all nodes

```
MATCH (n)RETURN count(n);
```

This query will return the number of nodes in the database.

Count all relationships

```
MATCH ()-->()RETURN count(*);
```

This query will return the number of relationships in the database.

Limit the number of returned results

```
MATCH (c:City)
RETURN c
LIMIT 5;
```

LIMIT 5: this will limit the number of returned nodes to 5.

Specify an alias for results

```
MATCH (c:Country)
WHERE c.name = "US"
RETURN c.population_size AS population
```

By using AS with the RETURN clause, the property population_size will be returned with an alias.

1.3. Neosemantics (n10s)

Neosemantics (n10s) is a Neo4j plugin that enables the use of the Resource Description Framework (RDF) in Neo4j and is part of the Neo4j Labs program.

RDF is a W3C standard model for data interchange.
Some key features of neosemantics are:

- Store RDF data in Neo4j in a lossless manner (imported RDF can subsequently be exported without losing a single triple in the process)
- On-demand export property graph data from Neo4j as RDF

2. Wikidata

Wikidata is a free, collaborative, multilingual, secondary database, collecting structured data to provide support for Wikipedia, Wikimedia Commons, the other wikis of the Wikimedia movement, and to anyone in the world. Wikidata provides its data in RDF (Resource Description Framework) format.

2.1. Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a standard model for data interchange on the Web, developed by the World Wide Web Consortium (W3C). It's particularly designed for representing information about resources in a structured and interlinked way, aligning with the concepts of the Semantic Web. RDF allows data to be shared and reused across application, enterprise, and community boundaries.

Key features of RDF:

- **Triple Structure**: The basic structure of RDF is a set of triples, each consisting of a subject, a predicate, and an object. This simple structure enables the representation of semantic statements about resources.
    - Subject: The entity or thing being described.
    - Predicate: A trait or aspect of the subject, or a relationship between the subject and the object.
    - Object: A specific value or another resource that is related to the subject.
- **Graph-Based Model**: RDF triples can be visualized as a directed graph, where resources are nodes and predicates are the edges that connect these nodes. This makes RDF excellent for modeling complex interrelationships between entities.
- **URI-Based Identifiers**: Resources are typically identified using Uniform Resource Identifiers (URIs), ensuring unique identification across the web. This includes resources that are not directly retrievable via the web (like abstract concepts or non-digital entities).
- **Extensibility**: RDF is designed to be extensible. It allows the use of various vocabularies and ontologies to express data more precisely.
- **Data Interchange**: RDF provides a common framework for data interchange, which can be leveraged in various data formats, including RDF/XML, N-Triples, Turtle, and JSON-LD.

2.2. SPARQL

RDF data is often queried using SPARQL, a powerful RDF query language that allows for complex querying and manipulation of RDF graphs.

a. SELECT

```
SELECT ?variable1 ?variable2 ... WHERE {
# pattern matching
```

```
# conditions
}
```

- SELECT Clause: This specifies the variables to be returned in the query results. Variables in SPARQL are prefixed with a question mark (?).
- WHERE Clause: This contains a set of triple patterns and optional filters. It describes the pattern that must be matched in the RDF data for the variables in the SELECT clause to be returned.

b. WHERE

- The WHERE clause consists of a group of triple patterns, along with optional patterns, filters, and other constraints. These patterns are used to match specific subsets of data within the RDF graph.
- Triple Patterns: These are the core of the WHERE clause. A triple pattern has a similar structure to an RDF triple and consists of a subject, predicate, and object. These can be specific URIs, literals, blank nodes, or variables (indicated with a question mark, e.g., ?subject).
- Grouping of Patterns: Triple patterns within a WHERE clause are grouped together using curly braces { ... }.
- Dot Separator: Individual triple patterns are often separated by a period ., indicating the end of a pattern.

c. FILTER

- Variable Constraints: FILTER can be used to set conditions on variables. For example, FILTER (?age > 18) would restrict results to cases where the variable ?age is greater than 18.
- String Operations: You can perform operations on string variables, like FILTER (CONTAINS(?name, "Smith")) to select only those entries where the ?name variable contains the substring "Smith".
- Logical Expressions: FILTER supports logical operators (&& for AND, || for OR, ! for NOT). For instance, FILTER (?age > 18 && ?age < 65) would select ages between 18 and 65.
- Comparison Operators: It includes comparison operators like =, !=, >, <, >=, <=.
- Regular Expressions: FILTER can use regular expressions for complex pattern matching, for example, FILTER (REGEX(?email, "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")) to filter valid email addresses.

- Functions: SPARQL provides a variety of functions that can be used in FILTER, such as STR(), LANG(), DATATYPE(), and mathematical functions.

3. Fine-tune ChatGPT Model

The GPT-3.5 architecture is a neural network-based language model that has been trained on a massive amount of data. It is an advanced version of the GPT-3 architecture and is capable of performing a wide range of language tasks, including language translation, text completion, and question answering.
Fine-tuning improves on few-shot learning by training on many more examples than can fit in the prompt, letting us achieve better results on a wide number of tasks. Once a model has been fine-tuned, we won't need to provide as many examples in the prompt. This saves costs and enables lower-latency requests.

3.1. Dataset

We use Wikidata Query Service to get the dataset of 100 music songs and their properties:

- **songLabel**: Name of the song
- **releaseDate**: The year the song was written
- **artistLabel**: Name of the singer perform the song
- **composerLabel**: Name of the composer of the song
- **genreLabel**: The genre of the song

songLabel	releaseDate	artistLabel	composerLabel	genreLabel
Hey Hey, My My (Into the Black)	1979	Neil Young	Neil Young	rock music
3	2009	Britney Spears	Max Martin	synth-pop
You Rock My World	2001	Michael Jackson	Michael Jackson	contemporary R&B
You Rock My World	2001	Michael Jackson	Michael Jackson	post-disco
Hey Hey, My My (Into the Black)	1979	Neil Young	Neil Young	hard rock
Un-Break My Heart	1996	Toni Braxton	Diane Warren	rhythm and blues
Un-Break My Heart	1996	Toni Braxton	Diane Warren	rhythm and blues
California Gurls	2010	Snoop Dogg	Snoop Dogg	electropop
Space Oddity	1969	David Bowie	David Bowie	folk rock
Space Oddity	1969	David Bowie	David Bowie	space rock

3.2. Prepare the dataset

Once we have determined that fine-tuning is the right solution (i.e. we've optimized your prompt as far as it can take you and identified problems that the model still has), we will need to prepare data for training the model. We should create a diverse set of demonstration conversations that are similar to the conversations we will ask the model to respond to at inference time in

production.

Each example in the dataset should be a conversation in the same format as our Chat Completions API, specifically a list of messages where each message has a role, content, and optional name. At least some of the training examples should directly target cases where the prompted model is not behaving as desired, and the provided assistant messages in the data should be the ideal responses we want the model to provide.

The format of the dataset must be:

```
{"messages": [
            {"role": "system", "content": "Your task is to translate
natural language queries into Neo4j Cypher queries."},
            {"role": "user", "content": "Who performed My Life ?"},
            {"role": "assistant", "content": "MATCH
(a:Artist)-[:PERFORMED]->(s:Song {title: 'My Life'}) RETURN a.name"}
]}
```

A list of things you might want to fix includes:

- Making sure that your dataset does not contain duplicate examples.
- Making sure that your examples are utf-8 encoded.

3.3. Train a new fine-tuned model

Create a fine-tuned model

After ensuring we have the right amount and structure for our dataset, and have uploaded the file, the next step is to create a fine-tuning job using OpenAI JDK.

```
suffix_name = "richard-test"

response = openai.fine_tuning.jobs.create(
    hyperparameters={"n_epochs": 5,
                     "batch_size" : 16},
    training_file=training_file_id,
    validation_file=validation_file_id,
    model="gpt-3.5-turbo-1106",
    suffix=suffix_name,
)
```

- **model**: The name of the model we want to fine-tune
- **training_file**: the file ID that was returned when the training file was uploaded to the OpenAI API.
- **validation_file**: the file ID that was returned when the validation file was uploaded to the OpenAI API.
- **hyperparameters**: The hyperparameters used for the fine-tuning job.
    - **n_epochs**: The number of epochs to train the model for
    - **batch_size**: number of examples in each batch

After we have started a fine-tuning job, it may take some time to complete. Our job may be queued behind other jobs in the system, and training a model can take minutes or hours depending on the model and dataset size.

Use a fine-tuned model

When the job has succeeded, we will use the API from OpenAI platform to make requests to our fine-tuned model.

```
system_message = "Your task is to translate natural language queries into Neo4j Cypher queries."
test_messages = []
test_messages.append({"role": "system", "content": system_message})
user_message = question
test_messages.append({"role": "user", "content": user_message})

answer = openai.chat.completions.create(
model="ft:gpt-3.5-turbo-1106:personal:richard-test:8jNM1VGb", messages=test_messages, temperature=0, max_tokens=500
)
return answer.choices[0].message.content
```

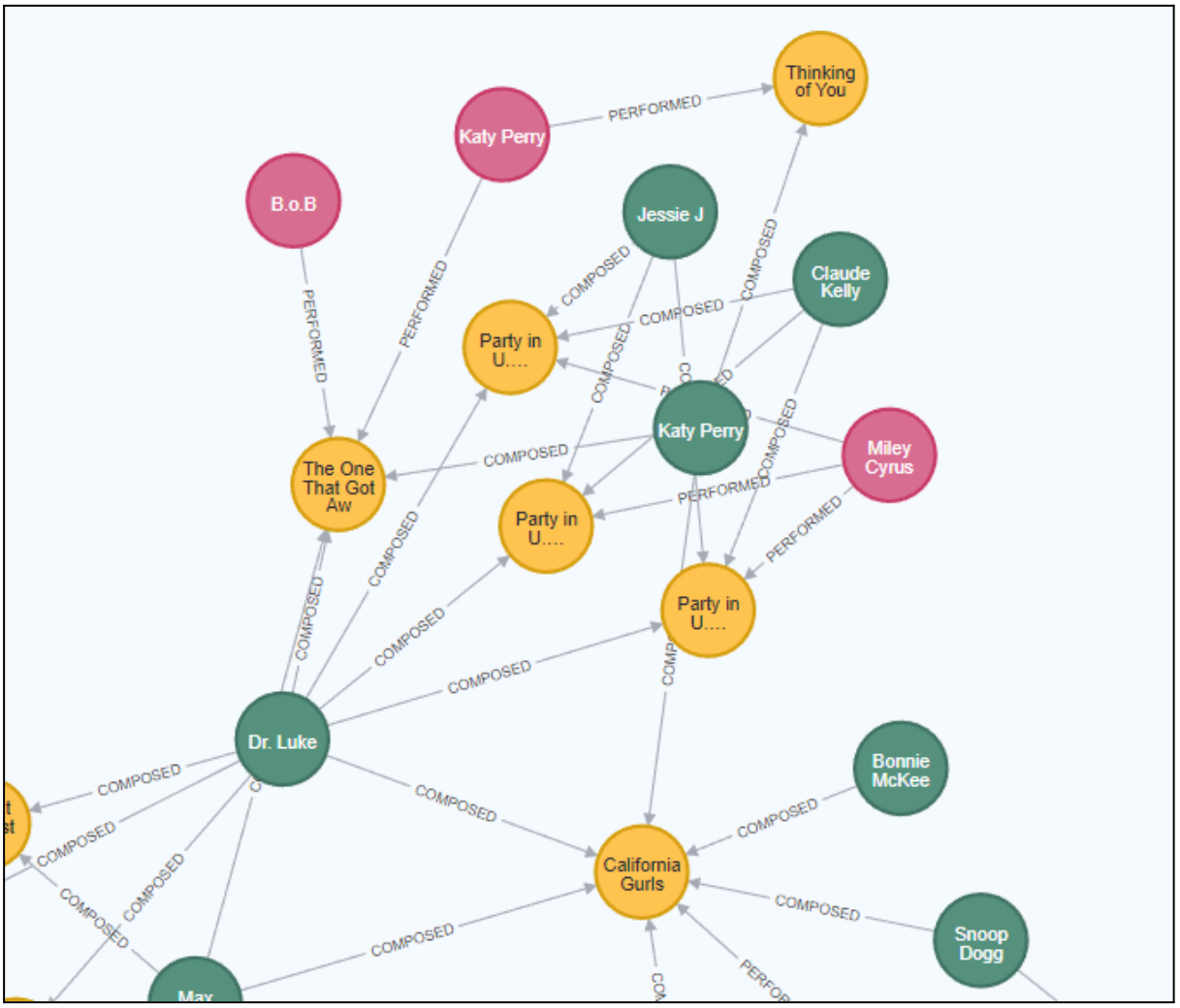
3.4. Evaluate the fine-tuned model

- The training loss is generally decreasing, which is a good sign as it indicates that the model is learning and improving its performance on the training data over time and the best result is about 0.15%.
- The validation loss is also decreasing, though not as smoothly as the training loss. This means the model is generalizing to new, unseen data, which is also positive and reaches 0.05% as the best result.

4. Application

4.1. Create the graph database

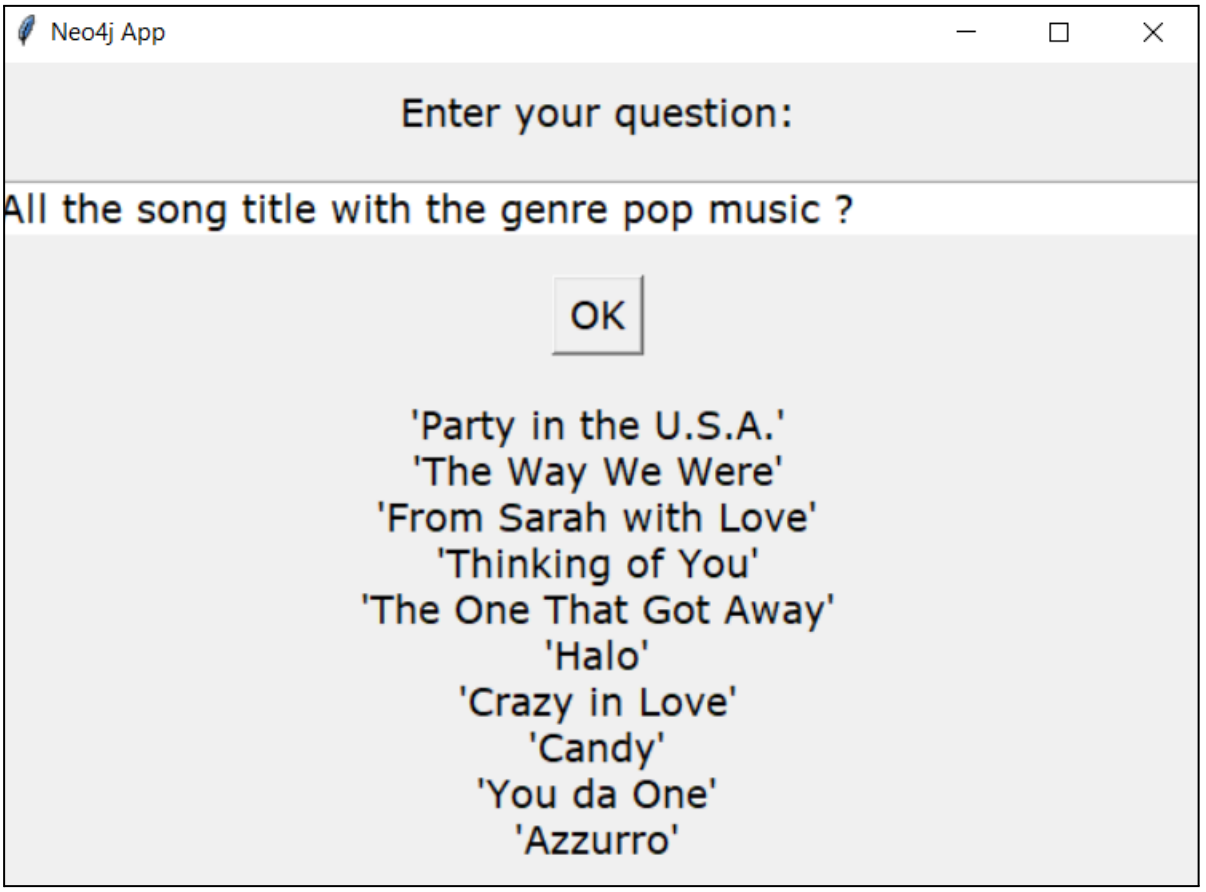
Firstly, we need to import the dataset we have from Wikidata into Neo4j, then create the entities and the relationships among them.
Here are the visualization of our graph database:



- Our graph database has 3 main entities:
  - **Song**: The song properties
  - **Artist**: The person who sing the song
  - **Composer**: The person who write the song
- The composer has COMPOSED relationship to the song then write
- The artist has PERFORMED relationship to the song they sing

4.2. Create the application GUI

For this part, we will use a package of Python called Tkinter to create the GUI for the application.



There will be a field for you to enter your natural sentences, then press the OK button and the result will be shown as in the image.

4.3. Explanation

```python
class Neo4jConnection:
    def __init__(self, uri, user, pwd):
        self.__uri = uri
        self.__user = user
        self.__password = pwd
        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()
```

The Neo4jConnection Class helps us to connect to the graph database in Neo4j using username and password

```python
def get_cypher_query(question):
    try:
        system_message = "Your task is to translate natural language queries into Neo4j Cypher queries."
        test_messages = []
        test_messages.append({"role": "system", "content": system_message})
        user_message = question
        test_messages.append({"role": "user", "content": user_message})

        answer = openai.chat.completions.create(
        model="ft:gpt-3.5-turbo-1106:personal:richard-test:8jNM1VGb", messages=test_messages, temperature=0, max_tokens=500
        )
        return answer.choices[0].message.content
    except Exception as e:
        print(f"Error: {e}")
        return None
```

The get_cypher_query( ) function will return the cypher query equivalent to the given natural sentences.

```python
def query(self, query, parameters=None, db=None):
    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
```

The query( ) function will take the given Cypher query into Neo4j to get the desired result.