Neo4j

Assignment 4

Papadatos Ioannis (t8190314)
Panourgia Evangelia (t8190130)
Professor: Chatziantoniou Damianos

Latest Update: May 14, 2022

School of Management Science and Technology,
Athens University of Economics and Business

Contents

Introduction

Neo4j is an open-source, NoSQL, native graph database. A graph database stores nodes and relationships instead of tables, or documents. Your data is stored without restricting it to a pre-defined model, allowing a very flexible way of thinking about and using it.

Cypher is the primary interface for Neo4j. Cypher is a powerful, intuitive, graph-optimized query language that understands, and takes advantage of, data connections. When trying to find patterns or insights within data, Cypher queries are much simpler and easier to write than massive SQL joins.

Installation

We used Neo4j's docker image to start a container by executing the following command:

```
docker run -d --name neo4j -p 7474:7474 -p 7687:7687 \
        -v neo4j-data:/data \
        -e NEO4J_AUTH=neo4j/secret_password \
        neo4j
```

We also created the equivalent docker-compose.yml file for the above command:

```yaml
version: "3.8"

services:
  neo4j:
    image: neo4j
    container_name: neo4j
    ports:
      - "7474:7474"
      - "7687:7687"
    volumes:
      - neo4j-data:/data
    env_file:
      - ./env/neo4j.env
    restart: unless-stopped

volumes:
  neo4j-data:
```

Importing Data in Neo4j

The script main.py was written for importing the data (https://snap.stanford.edu/data/act-mooc.html) in Neo4j.

```
import os
from dotenv import load_dotenv
from neo4j import GraphDatabase
```

The get_driver function uses the load_dotenv function from the dotenv module in order to load the environment variables specified in the .env file (that exists in the same directory). Then, it reads those environment variables and uses them to create a driver that can be used to establish a connection with the database.

```
def get_driver():
    load_dotenv() # Parses a .env file and loads the environment variables.
    scheme = os.getenv('SCHEME')
    host_name = os.getenv('HOST_NAME')
    port = os.getenv('PORT')
    uri = f"{scheme}://{host_name}:{port}"
    user = os.getenv('USER')
    password = os.getenv('PASSWORD')
    return GraphDatabase.driver(uri, auth=(user, password))
```

The create_user function is used to create a node with label User and a property user_id (the value of which is received as an argument), if it does not already exist.

```
def create_user(tx, user_id):
    tx.run("MERGE (u:User { user_id: $user_id })", user_id=user_id)
```

The create_target function is used to create a node with label Target and a property target_id (the value of which is received as an argument), if it does not already exist.

```
def create_target(tx, target_id):
    tx.run("MERGE (t:Target { target_id: $target_id })", target_id=target_id)
```

The create_action function is used to create a relationship with label Action and the properties: timestamp, feature_0, feature_1, feature_2, feature_3 and label. The user_id is used to indicate the start node & the target_id is used to indicate the end node of the relationship.

```
def create_action(tx, user_id, target_id, action_id, timestamp,
        feature_0, feature_1, feature_2, feature_3, label):
    tx.run("MATCH (u:User { user_id: $user_id }), (t:Target { target_id: $target_id }) "
           "CREATE (u) -[:Action { action_id: $action_id, timestamp: $timestamp, "
           "feature_0: $feature_0, feature_1: $feature_1, feature_2: $feature_2, "
           "feature_3: $feature_3, label: $label }]-> (t)",
           user_id=user_id, target_id=target_id, action_id=action_id, timestamp=timestamp,
           feature_0=feature_0, feature_1=feature_1, feature_2=feature_2,
           feature_3=feature_3, label=label)
```

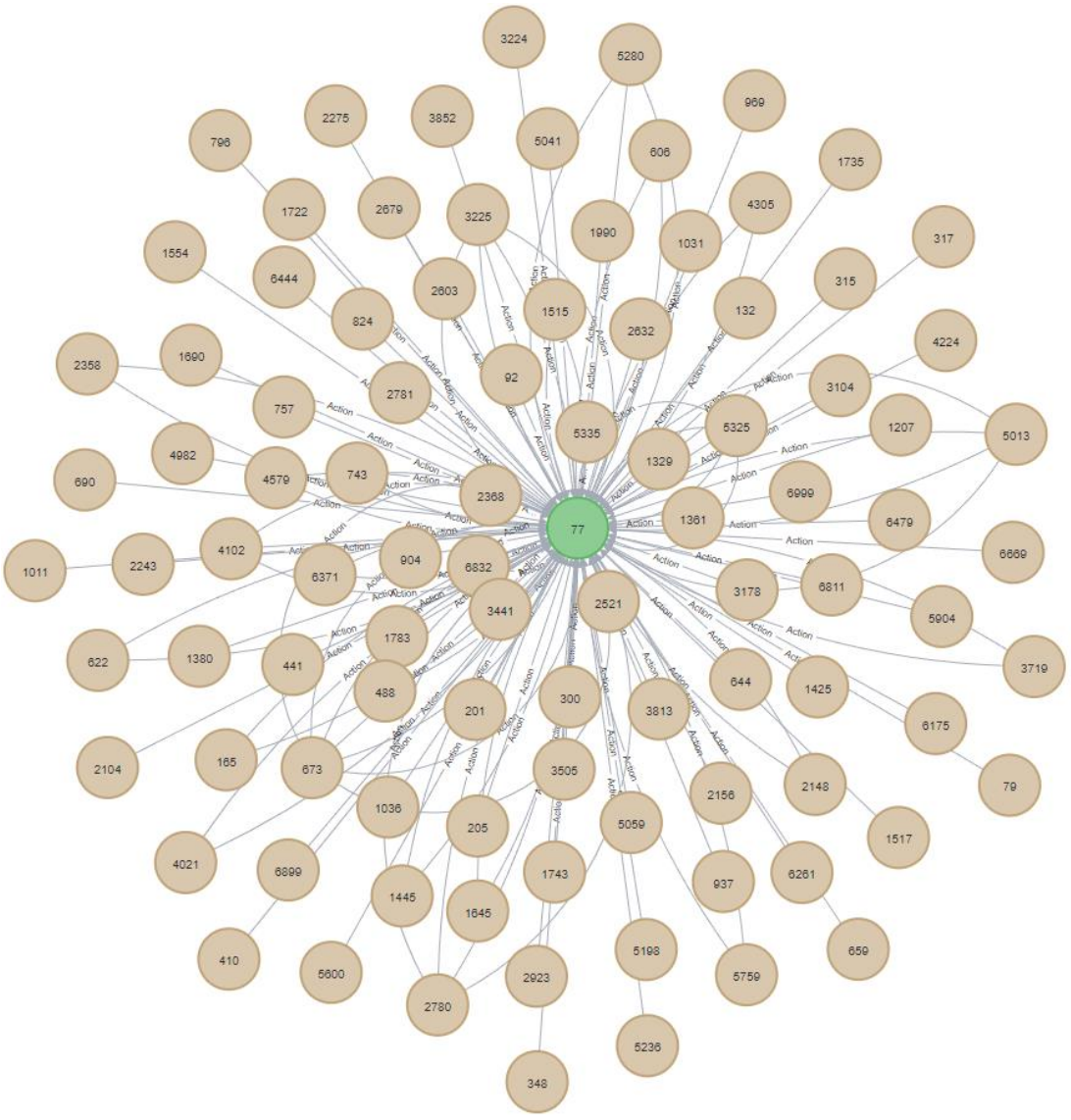
When the script is executed, it opens the three files containing the data. Each file contains a line with the column labels that is skipped and 411749 lines of the actions (which are sorted in DESC order based on the ACTIONID column). Therefore, we can use the built-in zip function to iterate the lines of those files in parallel, having access to all of the required properties in the same iteration, that allows us to create the nodes (Users & Targets) and the relationships (Actions).

```
if __name__ == "__main__":
    driver = get_driver()

    with open('../data/mooc_actions.tsv') as actions_f, \
            open('../data/mooc_action_features.tsv') as features_f, \
            open('../data/mooc_action_labels.tsv') as labels_f, \
            driver.session() as session:
        next(actions_f) # Skips the 1st line containing the column labels.
        next(features_f) # Skips the 1st line containing the column labels.
        next(labels_f) # Skips the 1st line containing the column labels.

        for line_1, line_2, line_3 in zip(actions_f, features_f, labels_f):
            line_1_elements = line_1.strip().split('\t')
            timestamp = float(line_1_elements[-1])
            action_id, user_id, target_id = [int(e) for e in line_1_elements[:-1]]

            line_2_elements = line_2.strip().split('\t')
            feature_0, feature_1, feature_2, feature_3 = \
                [float(e) for e in line_2_elements[1:]]

            line_3_elements = line_3.strip().split('\t')
            label = int(line_3_elements[-1])

            session.write_transaction(create_user, user_id)
            session.write_transaction(create_target, target_id)
            session.write_transaction(create_action, user_id, target_id, action_id, timestamp,
                feature_0, feature_1, feature_2, feature_3, label)

    driver.close()
```

Cypher Queries

Query 1: Show a small portion of your graph database (screenshot)

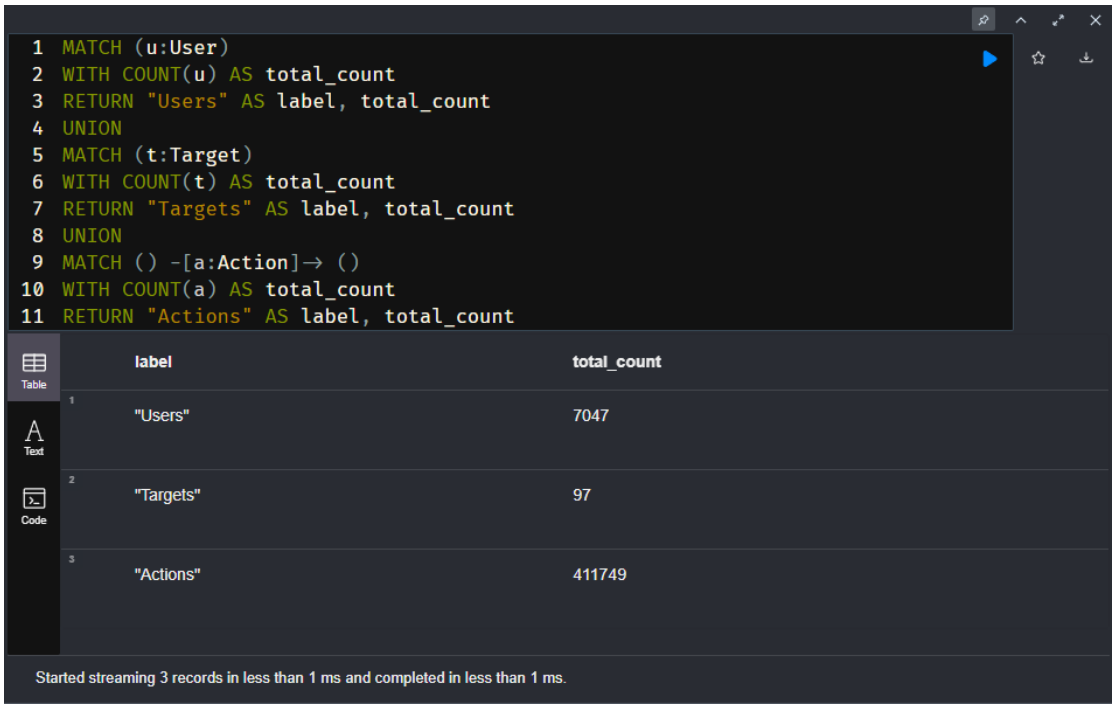
```
MATCH (u:User) -[a:Action]→ (t:Target { target_id: 77 })
RETURN u, a, t
```



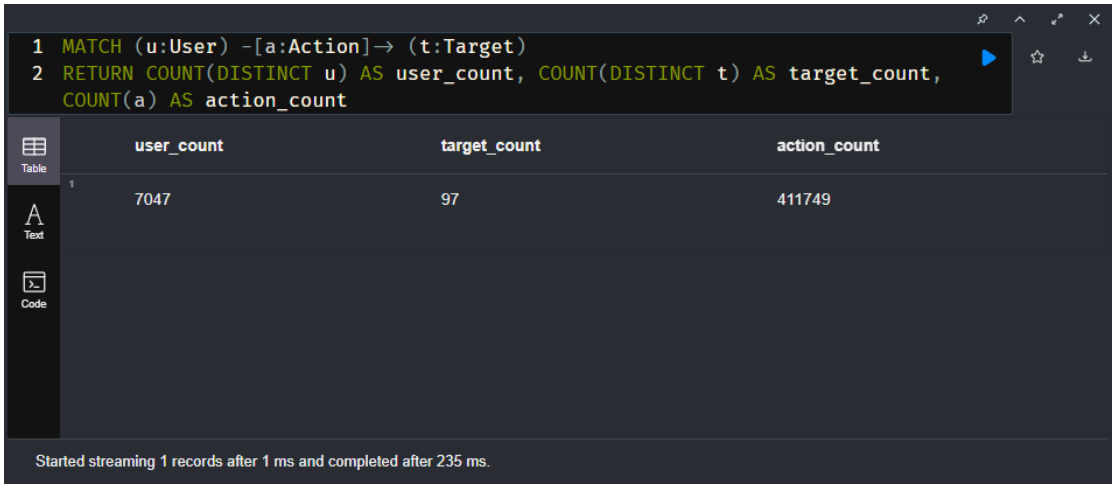
(In this screenshot we can see all the users that have performed at least one action on the target with target_id = 77)

Query 2: Count all users, count all targets, count all actions

Neo4j maintains a transactional count store for holding count metadata for a number of things. Obtaining counts from the count store is constant-time, so if you want counts for something that is obtainable from the count store, it can be queried quickly. Due to limitations of the query planner, the count store will only be leveraged if the count() aggregation is alone on a WITH or RETURN. If any other variable is in scope along with the count() aggregation, the count store will not be used. Also, it is stated that the count store can't be used with labels present on both start and end nodes.
(https://neo4j.com/developer/kb/fast-counts-using-the-count-store/)



We can also use this less performant alternative (that don't leverages the count store), to get the counts in the same record:



Query 3: Show all actions (actionID) and targets (targetID) of a specific user (choose one)

```
1 MATCH (:User { user_id: 77 }) -[a:Action]→ (t:Target)
2 RETURN a.action_id AS action_id, t.target_id AS target_id
3 ORDER BY t.target_id
```

Table | Text | Code

	action_id	target_id
1	498	1
2	505	1
3	2755	3
4	2765	3
5	2771	3
6	3615	3
7		

Started streaming 59 records in less than 1 ms and completed after 3 ms.

Query 4: For each user, count his/her actions

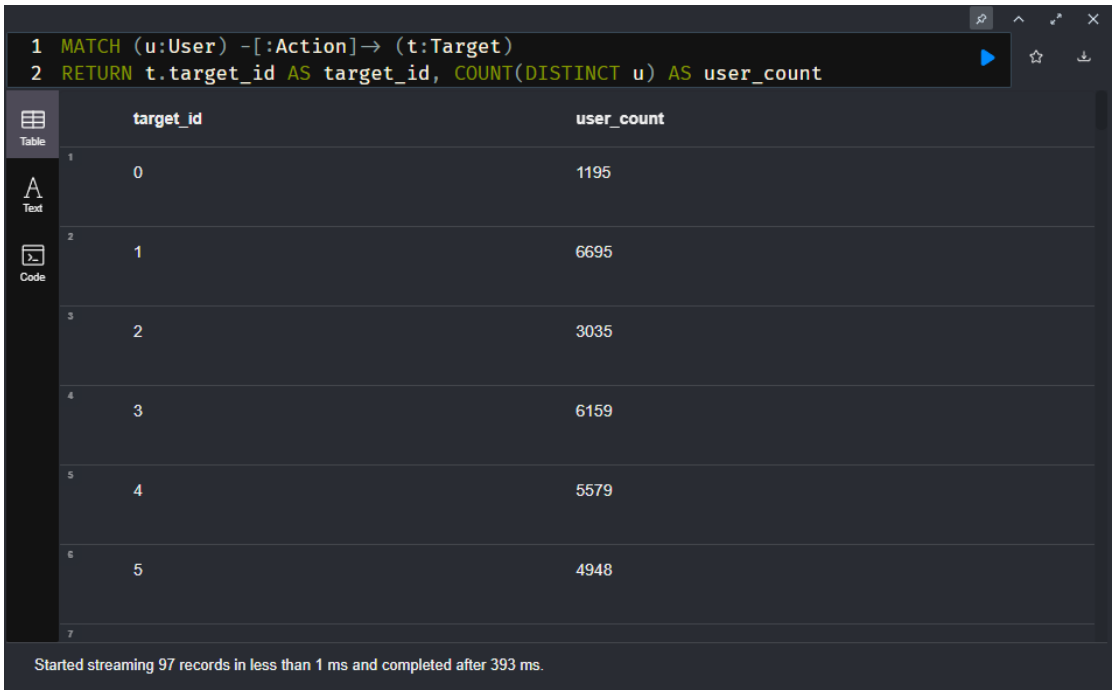
```
1 MATCH (u:User) -[:Action]→ ()
2 RETURN u.user_id AS user_id, COUNT(*) AS action_count
3 ORDER BY u.user_id
```

Table | Text | Code

	user_id	action_count
1	0	76
2	1	26
3	2	189
4	3	6
5	4	9
6	5	88
7		

Started streaming 7047 records in less than 1 ms and completed in less than 1 ms, displaying first 1000 rows.

Query 5: For each target, count how many users have done it

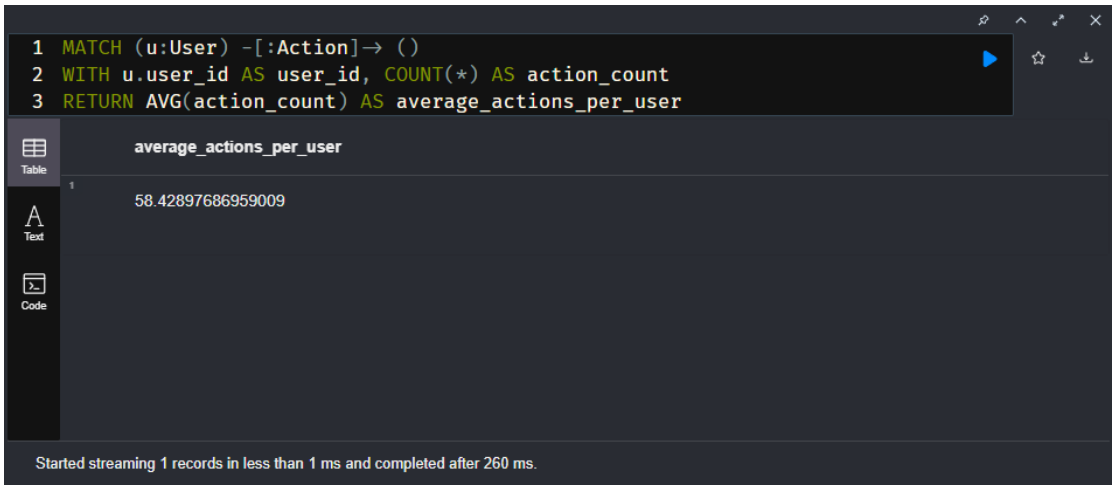


```
MATCH (u:User) -[:Action]→ (t:Target)
RETURN t.target_id AS target_id, COUNT(DISTINCT u) AS user_count
```

	target_id	user_count
1	0	1195
2	1	6695
3	2	3035
4	3	6159
5	4	5579
6	5	4948
7		

Started streaming 97 records in less than 1 ms and completed after 393 ms.

Query 6: Count the average actions per user



```
MATCH (u:User) -[:Action]→ ()
WITH u.user_id AS user_id, COUNT(*) AS action_count
RETURN AVG(action_count) AS average_actions_per_user
```

	average_actions_per_user
1	58.42897686959009

Started streaming 1 records in less than 1 ms and completed after 260 ms.

Query 7: Show the userID and the targetID, if the action has positive Feature2

```
MATCH (u:User) -[a:Action]→ (t:Target)
WHERE a.feature_2 > 0.0
RETURN u.user_id AS user_id, t.target_id AS target_id
```

	user_id	target_id
1	5658	0
2	6725	0
3	195	0
4	1783	0
5	5593	0
6	6571	0
7		

Started streaming 287960 records in less than 1 ms and completed after 3 ms, displaying first 1000 rows.

Query 8: For each targetID, count the actions with label “1”

```
MATCH (:User) -[a:Action { label: 1 }]→ (t:Target)
RETURN t.target_id AS target_id, COUNT(*) AS positive_action_label_count
```

	target_id	positive_action_label_count
1	0	31
2	1	64
3	2	49
4	3	128
5	4	80
6	5	113
7		

Started streaming 85 records in less than 1 ms and completed after 459 ms.

References

- https://neo4j.com/
- https://www.docker.com/