Student Name: Iliadis Viktoras
Student ID: 8180026

**Big Data Management Systems**
Assignment 5 - MapReduce/Hadoop

---

All the code and files can be found in the following repository like all the previous assignments. Some of the output files were too big to be included in this report. To showcase the outputs, they have been included in the deliverable as separate files.

## System Schema

The `load_data` Python script is used to load data into a Neo4j graph database. Below are some key decisions and observations I made when writing the script.

- **UNWIND**: The `UNWIND` clause is used to transform a list of values into individual rows. In this script, `UNWIND` is used to process a list of users, targets, and actions in batches. This is more efficient than processing each user, target, and action individually, as it reduces the overhead of communicating with the Neo4j server. For example, the line `UNWIND $users AS user_id MERGE (:User {id: user_id})` processes each user in the `users` list, creating a new `User` node for each user.
- **Indexes**: Indexes are used to speed up the lookup of nodes in a graph. In this script, indexes are created on the `id` properties of the `User` and `Target` nodes. This makes the `MATCH` operations in the `MERGE` clause faster, as Neo4j can use the index to find the nodes instead of scanning all the nodes in the graph. The lines `CREATE INDEX FOR (n:User) ON (n.id)` and `CREATE INDEX FOR (n:Target) ON (n.id)` create these indexes.
- **MERGE vs CREATE**: In Neo4j, `CREATE` and `MERGE` are two different commands used to create nodes and relationships. `CREATE` is used to create a new node or relationship without checking if an identical node or relationship already exists, leading to potential duplicates. On the other hand, `MERGE` is used to create a node or relationship if it doesn't already exist, and if it does exist, `MERGE` will match it instead of creating a new one. This ensures that there are no duplicates. In this script, `MERGE` is used to create the `User`, `Target`, and `ACTION` nodes and relationships, ensuring that each is unique.
- **Number of Actions**: The number of `ACTION` relationships in the graph is less than the number of actions in the data because `MERGE` is used to create the `ACTION` relationships. `MERGE` only creates a new relationship if an identical one doesn't already exist. If there are actions in the data that have the same `USERID`, `TARGETID`, `ACTIONID`, `FEATURE2`, and `LABEL`, `MERGE` will match the existing `ACTION` relationship instead of creating a new one. This is why the number of `ACTION` relationships in the graph is less than the number of actions in the data.

In summary, the `UNWIND` clause, indexes, and the `MERGE` command are used in this script to optimize the data loading process, making it faster and more efficient, and ensuring the uniqueness of nodes and relationships.

```
%run load_data.py

Inserting users
Inserting targets
Creating indexes
Inserting actions
```

## Tasks

Below are the queries used to answer the questions in the tasks.

```python
# Connect to Neo4j
from py2neo import Graph

graph = Graph("bolt://localhost:7687", auth=("neo4j", "123456789"))  # replace with your actual password
```

Listing 1: Connecting to Neo4j

### Task 1

Using the Neo4j browser, we can see the graph database we created. The following is a screenshot of the graph database, after executing the query:

```
> MATCH (n)-[r]->(m) \
RETURN n, r, m \
LIMIT 25
```

**Note** The screenshot is also in the deliverable and in the repository.

```python
# show the Screenshot.png
from IPython.display import Image
Image(filename='Screenshot.png')
```

Listing 2: Python code to show graph database

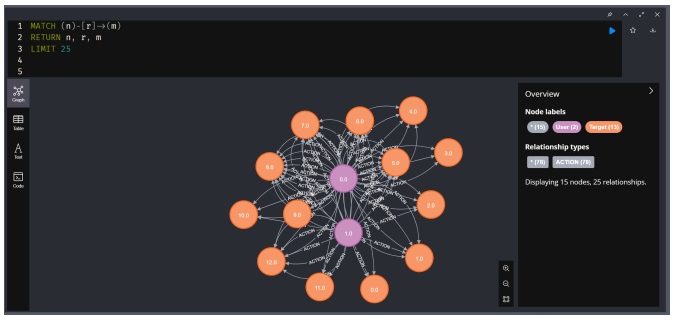


Figure 1: A beautiful picture.

### Task 2: Count all users, count all targets, count all actions

```
# Count all users
num_users = graph.run("MATCH (n:User) RETURN count(n)").evaluate()
print("Number of users: ", num_users)

# Count all targets
num_targets = graph.run("MATCH (n:Target) RETURN count(n)").evaluate()
print("Number of targets: ", num_targets)

# Count all actions
num_actions = graph.run("MATCH ()-[r:ACTION]->() RETURN count(r)").evaluate()
print("Number of actions: ", num_actions)
```

Listing 3: Task 2

```
Output:
Number of users:  7047
Number of targets:  97
Number of actions:  396712
```

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

```
# Choose a specific user
user_id = 3

# Get all actions and targets of the user
results = graph.run("""
MATCH (u:User {id: $user_id})-[a:ACTION]->(t:Target)
RETURN a.id AS actionID, t.id AS targetID
""", user_id=user_id)

for result in results:
    print("Action ID: ", result['actionID'], ", Target ID: ", result['targetID'])
```

Listing 4: Task 3

```
Output:
Action ID:  101294.0 , Target ID:  3.0
Action ID:  101293.0 , Target ID:  8.0
Action ID:  36.0 , Target ID:  13.0
Action ID:  30.0 , Target ID:  3.0
Action ID:  29.0 , Target ID:  10.0
Action ID:  26.0 , Target ID:  1.0
```

### Task 4: For each user, count his/her actions

```
results = graph.run("""
MATCH (u:User)-[a:ACTION]->()
RETURN u.id AS userID, count(a) AS numActions
""")

for result in results:
    print("User ID: ", result['userID'], ", Number of actions: ", result['numActions'])
```

Listing 5: Task 4

The output can be found on the file task4Output

### Task 5: For each target, count how many users have done this target

```
results = graph.run("""
MATCH (u:User)-[a:ACTION]->(t:Target)
RETURN t.id AS targetID, count(DISTINCT u) AS numUsers
""")

for result in results:
    print("Target ID: ", result['targetID'], ", Number of users: ", result['numUsers'])
```

Listing 6: Task 5

The output can be found on the file task5Output

### Task 6: Count the average actions per user

```
avg_actions_per_user = graph.run("""
MATCH (u:User)-[a:ACTION]->()
WITH u, count(a) AS actionsPerUser
RETURN avg(actionsPerUser) AS avgActionsPerUser
""").evaluate()

print("Average actions per user: ", avg_actions_per_user)
```

Listing 7: Task 6

```
Output:
Average actions per user:  56.29516106144467
```

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

```
results = graph.run("""
MATCH (u:User)-[a:ACTION]->(t:Target)
WHERE a.feature2 > 0
RETURN u.id AS userID, t.id AS targetID
""")

for result in results:
    print("User ID: ", result['userID'], ", Target ID: ", result['targetID'])
```

Listing 8: Task 7

The output can be found on the file task7Output

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

```
results = graph.run("""
MATCH ()-[a:ACTION {label: 1}]->(t:Target)
RETURN t.id AS targetID, count(a) AS numActions
""")

for result in results:
    print("Target ID: ", result['targetID'], ", Number of actions with label '1': ",
    result['numActions'])
```

Listing 9: Task 7

The output can be found on the file task8Output