DMD Assignment 1

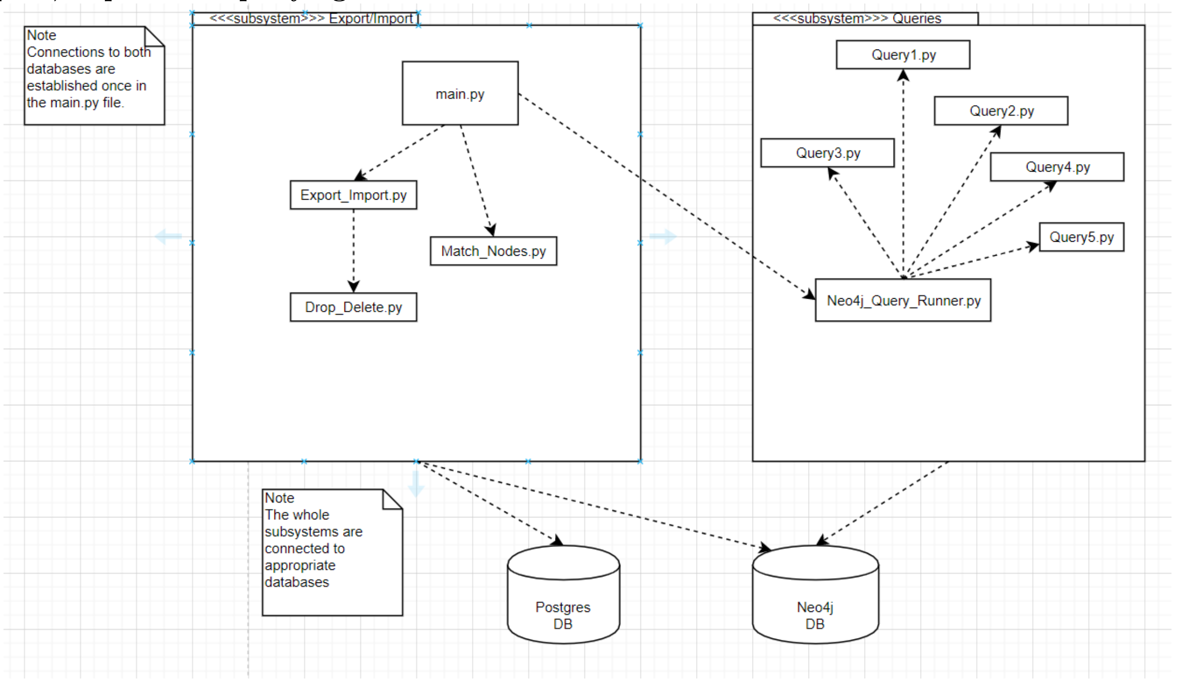
Marina Smirnova Group 6

March 2020

1 Introduction

The document contains overall description of used tools and techniques:
• The chosen NoSQL Database is Neo4j.
• The chosen programming language for scripts is Python.
• The script that executes the whole program (export, import, queries) is main.py.
• Results of all queries are displayed on the screen after execution of *execute_query()* in main.py.

The Component Diagram for all the scripts that were used for export, import and querying



2 Process of moving the database

Connections to Neo4j and Postgres Databases are established in main.py file. And the instances of cursor(for Postgres) and graph(for Neo4j) are passed to Export_Import.py file. Export and Import are done within a single file due to merged approach to doing them. The function to call from main.py is $execute\_export(cursor, tx, graph : Graph)$.

The file for drop of Neo4j Database(deletion of all nodes, relations and indexes) is imported immediately for simplicity. It is called Drop_Delete.py.

2.1 Export

For export of data from Postgres the usual operation $SELECT * FROMtable$; is used. So, all records from a certain table are fetched by cursor and then parsed for node's creation for Neo4j Database.

2.2 Import

At the very beginning, before import data records as nodes into Neo4j, indexes for labels of nodes should be created to speed up the overall process. Labels are equal to names of tables.

Then for each fetched record from the cursor a single node is created. Records are parsed according to attributes. For each attribute of the record, a property with the same name is created inside the node. Moreover, all operations with creation of nodes are done through transactions. It means: firstly, all statements for node's creation are containerized, and a final commit for all nodes is done.

Secondly, a file Matching_Nodes.py links together nodes according to necessary conditions on their properties. The function *execute_matching(graph: Graph)* is executed in the main.py to create all relations that are needed for execution of queries.

3 Adjustments for the new database

The list of made adjustments:
• Pictures in Staff table from Bytea format were converted into arrays of hexadecimal numbers.
• Two tables *film_actor* and *film_category* were implemented via relations between nodes with corresponding labels. And an attribute last_update is assigned to a relation between corresponding nodes.
• The most important adjustment is done over nodes of certain Labels: the relations are created to improve the speed of search over the Graph Database. All these relation types significantly improves execution of queries.
List of created relations:

1. $(actor) - [acts\_in] - > (film)$

2. $(film) - [is\_of\_category] - > (category)$
3. $(customer) - [ordered] - > (rental)$
4. $(rental) - [ordered\_inventory\_is] - > (inventory)$
5. $(inventory) - [contains\_film] - > (film)$
6. $(customer) - [watched] - > (film)$
7. $(customer) - [likes\_category] - > (category)$

4 Performance of the queries

4.1 Query 1

Execution of this query is quite fast if the technique of the shortest path is used. And if appropriate relations between nodes exist. Otherwise, if brute iteration over all customers, rentals, inventories and films is done, the execution time is enormous. So, there is a chain of links between nodes of listed above Labels. Also, the index on the rental date reduces the execution time.

Performance via Neo4j Browser: *Started streaming 21 records after 1 ms and completed after 80 ms.*

4.2 Query 2

This query is a little time-consuming because it is needed to match each actor to all others. And create a table 200 by 200 with results. The table is created by for loop and printed with help of tabulate library in python.

Performance via Neo4j Browser: *Started streaming 20868 records in less than 1 ms and completed after 2 ms, displaying first 1000 rows.*

4.3 Query 3

The query displays all films(and their categories) that were watched by a certain customer. It also shows how many times the film was rented by this customer.

The customer is chosen explicitly and given as a parameter to query.

Performance via Neo4j Browser (as an example for a customer with id = 5): *Started streaming 38 records after 1 ms and completed after 3 ms.*

4.4 Query 4

The list recommendations for a certain customer is based on customers that watched similar sets of movies. So, firstly, the top three most rented categories of films are taken (for the given customer). Then, up to 10 movies are found for each category (total 30 films at most). The conditions which taken films must satisfy: 1 - are of the category from top three; 2- were watched by other customers; 3 - were NOT watched by the given customer yet.

Performance via Neo4j Browser (as an example for a customer with id = 1): *Started streaming 30 records after 1 ms and completed after 10 ms.*

4.5 Query 5

The fixed actor for this quer has actor_id = 1. And another actor is specified via actor_id as a parameter to the query.

The query is executed with built-in functionality of a Graph Database. It's called shortestPath() algorithm. It helps to show the degree of separation between two actors. The relation $(actor) - [acts\_in]- > (film)$ is used to make a shortest possible path which goes through nodes of Labels: Actor and Film.

Performance via Neo4j Browser (as an example for actors with id = 1 and id = 2): *Displaying 5 nodes, 4 relationships. Completed in less than 1 ms.*