Sergey Makarov

Report

To test:

Beforewards, you need to install packages `neo4j` and `psycopg2` with `pip`. Then, run `docker-compose` up from the root directory of the assignment. This will create 2 containers with postgre and Neo4j respectively. Afterwards, you can run `python3 migrate.py`, which will migrate all data from postgre to Neo4j (you may need to run it with `sudo`, as docker usually protects mounted volumes from third-parties). After that, there is a script that runs all queries and shows their time to execute, named `runallqueries.sh`, but if you want to test each query manually, each query is in separate python file in the `./queries/python` directory. However, as I use relative path in this queries to save `.csv` results, it is better to run from root directory to save results in `./tables/queries` instead of `./queries/python/tables/queries`.

Process and adjustments:

As soon as Neo4j has a built-in functional to load `.csv` files, it was very easy to move all data from the relational one. I generate this `.csv` file in `migrate.py` and then just load with simple cypher query. However, in order to make queries look simple and work fast, we need to convert all foreign keys into a new relation. Here is an example:

```
MATCH (f:Film),(l:Language)
WHERE f.language_id = l.language_id
CREATE (l)-[r:film_language]->(f)
```

Nothing special, but this should be performed several times, so this is actually the biggest part of code. As for `film_actor` and `film_category` relations, I decided not to make anything special and just to write separate functions for importing them to Neo4j(where they became just a relation), but if there were more such relations, it is easy to write general function to work with all of them, as I did for all other tables. I also created index for each primary key for fast import and fast queries. There is a special relation `watch` which is used in fourth query. In fact, it is not needed, but query itself becomes shorter.

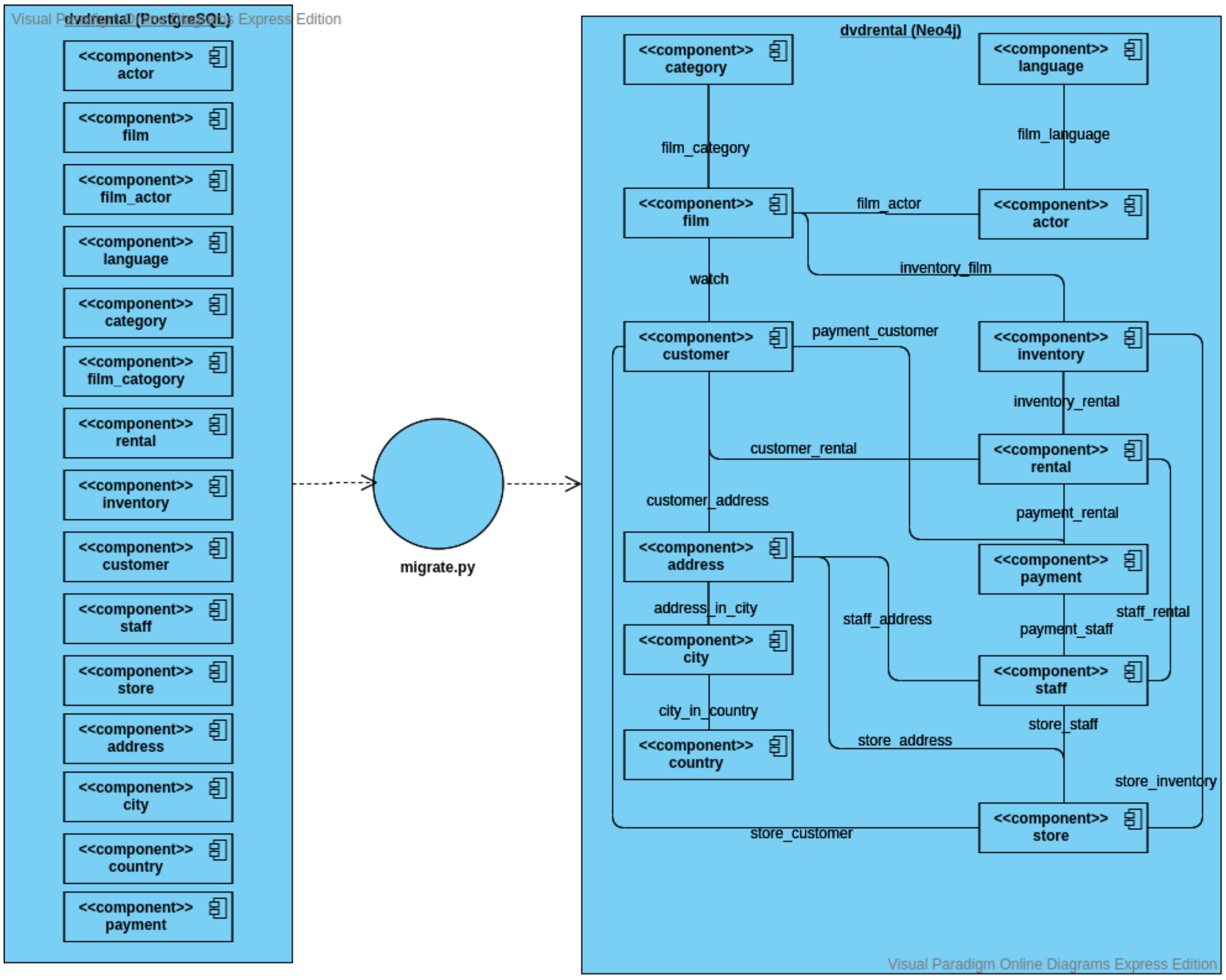
Challanges:

Before I found out Neo4j can directly import data from `.csv`, I implemented solution where for each record in each table I make a transaction to Neo4j. This solution taked triple times more code and worked ~8 minutes, so I decided to rewrite it.

Queries:

Queries that are present in the assignment are extremely easy to implement with Neo4j. All of them take just a couple of lines. 1, 3, 5 are easy, I implemented them in cypher and use python only to generate `.csv` report. Second is a bit harder, because there is no simple way to build 200x200 table in Neo4j, so I return a cartesian product from a cypher query (40000 lines like "actor 1, actor 2, count") and then convert them to 200x200 table using python. Because of this cartesian product query works for quite a long time, 3-4 seconds. As for fourth query, which is also implemented in cypher directly, I'd like to introduce my metric: for each customer we determine some customer which has common films. We divide this intersection of watched films over count of films first customer watched, and this will be our metric for all films that is watched by second, but not watched by first. Then we sum this metric for certain film for every possible "second" customer. We can use sum, but I think it is better to watch film that many people recommend, than film that is recommended by one person, even though this person has similar preferences. As for performance, all queries run really fast, except second one due to cartesian product.

Component diagram:


dvdrental (PostgreSQL)
<<component>> actor
<<component>> film
<<component>> film_actor
<<component>> language
<<component>> category
<<component>> film_catogory
<<component>> rental
<<component>> inventory
<<component>> customer
<<component>> staff
<<component>> store
<<component>> address
<<component>> city
<<component>> country
<<component>> payment
migrate.py
dvdrental (Neo4j)
<<component>> category
<<component>> language
film_category
film_language
<<component>> film
film_actor
<<component>> actor
inventory_film
watch
<<component>> customer
payment_customer
<<component>> inventory
inventory_rental
customer_rental
<<component>> rental
customer_address
payment_rental
<<component>> address
<<component>> payment
address_in_city
staff_address
staff_rental
payment_staff
<<component>> city
<<component>> staff
city_in_country
store_staff
store_address
<<component>> country
store_inventory
<<component>> store
store_customer
Visual Paradigm Online Diagrams Express Edition