Data Modelling and Databases II

Assignment 1

Magomed Magomedov, BS18-05

Migration

Database migration is implemented via *migration.py* script which can be divided into 3 parts:

1. Executing *restore.sql* file to populate PostgreSQL database.
2. Obtaining a list of tables from the populated database.
3. For each table create a MongoDB collection while moving data row by row.

There were different thoughts about whether the initial schema should be changed or not. On one hand, the initial schema is really good, it is not redundant and it is normalized. On another hand, clearly this relational schema might have not been suitable for a NoSQL database as is. After researching possible solutions and realizing that I will have to enforce redundancy anyway by implementing nested BSON objects, I have decided to perform the transfer without any changes to the initial PostgreSQL schema.

Performance

My query implementation process was facilitated by automatic SQL to PyMongo by Studio3T converter. But since NoSQL query languages generally contain less functionality than their SQL counterparts, a huge amount of query logic was implemented as python code. Still, performance of the queries cannot be assessed as being fast, due to the large amount of necessary INNER JOIN operators.

Queries

1. Perform INNER JOIN all the way from *Customer* to *Rental* to *Inventory* to *Film* to *Film_Category*. Match rental year to 2006, and select users with count of film categories greater than 2. This is the only query where the entire selection is in the pymongo pipeline.
2. Perform INNER JOIN of *Actor* with itself while counting common films. This will create an array of 3 element tuples containing two actors' ids and number of films they co-starred in. Then in python I go over the NxN grid, row by row, and if there is such a tuple where the coordinates are equal to the ids', I put that number, otherwise - 0.
3. Here I make 2 separate subqueries: INNER JOIN on *Film*, *Category*, *Film_category* and another INNER JOIN on *Customer*, *Rental*, *Inventory*. Then in python I go through every record in the first query, write its film's title, category, and number of inventories with the same film_id from the second query.
4. Similarly to the previous query I INNER JOIN *Customer*, *Rental* and *Inventory*. CD denotes normalized contribution of a customer. KD denotes normalized contribution of each matched film. Their multiplication gives the total contribution of each film of this customer, which I then sum into the final dictionary. The more films you share with some customer, the more recommended to you the other movies he watched.
5. Similarly the 2nd query, create a list of 2-item actor1-actor2 ids tuples, forming a simple graph. Then in python perform depth-first recursive search, overwriting saved distance when a shortest path was found.