IN4331 Web Data Management
Development Assignment Report

Piotr Tekieli
Xiaoxu Gao
Alexander Overvoorde
Tom Peeters

1. INTRODUCTION

The purpose of this assignment was connected with the creation of a movie information web service supporting multiple backend storage systems. For each technology, an appropriate API had to be developed, in order to enable efficient and simplified communication between client machines and the database itself. The starting point for the project was established with the development of PostgreSQL support module and the inclusion of data obtained from IMDB platform, which acted as a real-life testing source. Further, two other systems were implemented (i.e. MongoDB [document-based store] and Neo4J [graph-based database]), in order to present the perspective of NoSQL technologies on the matter of handling such data and contrast the differences in design and performance with respect to Relation Database approach. The whole project was realized in JAVA using JAX-RS API.

2. RELATIONAL DATABASE

A relational database is a collection of items organized as a set of well-structured tables from which data can be accessed and modified easily. Nowadays, it's still popular because of its flexibility and data normalization.

2.1 Pre-processing

When first populating a SQL database, we needed to insert a large amount of data. To smoothly play around with it later, some strategies were needed to make these query processes as efficient as possible. In order to achieve that, first, we used VACUUM module to collect obsolete garbage. In effect, after a while, we reclaimed some storage space occupied by dead tuples. Then, we performed the REINDEX operation, rebuilding indexes, by using the existing data stored in tables.

2.2 Data Model

The Relational Model organized data into several tables, representing respective entities, with a set of columns and rows, each equipped with a unique key (i.e. primary key of a row). The rows, on the other hand, represented instances of a particular entity and the column represented values attributed to these instances. The relationships (logical connections) between tables were created with the use of a set of foreign keys and associative entities.

2.3 Implementation

In this case, we managed to implement all service interfaces mentioned in the specification. The desing was based around three controllers: ActorController, GenreController and MovieController, which transformed client's API requests into SQL queries, sent them to the database and handled received responses. Each request concering the data located in a seprate entities, which could not "linked toghether" using the JOIN clause, was formulated as a separate SQL query. Therefore, in order to get Full Movie Info, we needed to send 5 queries to the DB. In case of statistic purposes, we used built-in functions.

2.4 Experience

By doing this project, we observed that there are both several advantages and disadvantages, when it comes to Relational Datases. First and foremost, this technology is quite flexible in aquiring any kind of information from the created database. The SQL statements used for this purpose are intiutive and easy to "forge". On the other hand, it's susceptible to impedence misatch with respect to used object oriented programming language.

3. NON-RELATIONAL DATABASE

3.1 MongoDB

MongoDB is a document oriented database that provides high performance, flexibility and easy scalability. It's based on the concept of collections of JSON documents.

3.1.1 Pre-processing and data model

The initial schema was created by simply exporting the PostgreSQL tables as CSV and importing them into MongoDB. This resulted in a collection for every SQL table filled with documents using the original columns as key/values. After that we normalized the smaller data like genres and keywords into arrays in the movies collection. Many of the original purely relational tables like acted_in was carried over to simulate joins between movies and actors. The decision to not denormalize this data was made because of anticipated storage and data updating costs. Indexes were added to the ID fields, relations like acted_in and the keyword/genre arrays to facilitate fast queries for the service endpoints.

3.1.2 Implementation

The general approach to the queries behind every service interface was two queries:

- A search query to get the IDs of the matched documents
- A bulk data query to retrieve all full documents matching these IDs

Sometimes a third query was required to, for example, fill in the details of actors in a sequence of movie documents. All of these steps were required because we did not fully denormalize the data, which turned out to not be such a great decision in the end.

Despite the downsides of artifically performing joins on the clientside, the performance of MongoDB using this query model turned out to be adequate. The statistics queries were implemented using MongoDB's aggregation pipelines. The sections below will shortly cover each service interface specifically.

SC1: Detailed movie information

A find query is run on the movies collection with a regular expression for the title field. The regular expression is set up such that it does not have a start or ending marker, so it can occur anywhere within a title (partial matching). If a year is specified, then an exact match for it is included in the find query. Once all of the basic movie documents have been retrieved that way, the information is augmented by the summaries of the actors. This is done with two more queries: acted_in with all of the matched movies as possible ids and then a query on the actors collection to find all of the actor documents using the ids found in the previous query. These actor documents are then merged with the movie documents. If a movie is queried by id instead, then the initial find query is much simpler: an exact match for the id on the idmovies field.

SC2: Detailed actor information

A find query on the actor id or a case-insensitive regular expression query on the first name, last name or both. The movies corresponding to these actors are subsequently found in the acted_in collection with one bulk query. All of the actor ids are then turned into documents with one final query. The information is then all merged clientside.

SC3: Short actor statistics

The document of the actor is looked up the same was as in the previous service interface and an aggregate query is then run on the acted_in collection to find the number of movies.

SC4: Genre exploration

A query is run on the movies collection that checks if the specified index occurs in the genres array of each movie (using an index). The find query optionally has more filters like a year range ($gt, $lt). The basic movie documents are then augmented with actor data by the previously described process.

SC5: Genre statistics

This query can be performed using a single aggregation query that essentially performs a group by sum operation on all movie documents and their genre arrays. Doing the group by directly on the array would yield invalid results, so it uses $unwind to treat each genre as its own value. This effectively duplicates every movie document for all of the genres contained within the array. An optional filter for the year is added to the query if it has been specified. The resulting totals are then returned to the user as movies per genre.

3.1.3 Experience

The advantage of MongoDB's document model is that it maps very nicely to Java objects, especially when using proper denormalization. Unfortunately we made the decision to use only very limited denormalization, which resulted in a lot of extra work on the clientside.

One of the main drives behind this decision was the atomic transaction model of MongoDB. It would require changing relations like actors in movies to update multiple documents at the same time, which could lead to inconsistencies. This disadvantage is most likely outweighed by the fact that reads would be much more frequent and much faster, however.

Still, even if we had properly denormalized the database, we would still consider MongoDB unappropriate for this task for several reasons:

- It requires you to have all of the ways you want to access your data in mind from the very beginning; it does not support flexibly querying.
- Denormalization can lead to inconsistencies because there are many relations in the data.
- MongoDB requires client-side data validation because of its flexible schema, creating more work for the programmer. Essentially, it requires defensive programming while interfacing with the database whereas it should be the database's job to keep the data consistent and orderly.

If we had to do this assignment again, we would definitely embed the arrays of actors and movies within each other's objects. That would have made the queries quite a bit faster and would have made the deserialization code faster. It would on the other hand also make the import process more difficult, because it would require significant preprocessing.

3.2 Neo4j

3.2.1 Pre-processing

Several steps needed to be performed to convert from a relational database into a Neo4j database. First, PostgreSQL's data was exported to CSV files using commands. Afterwards, the data was imported into Neo4j and indices were created. Finally, the relationships between the nodes were added. Relationships were created using the CSV files as in-between file. This is because the foreign keys were not stored within the Neo4j nodes, as the existence of relationships makes them redundant. Additionally, Neo4j standard approach is to load the actions of the entire operation into RAM before executing it when an in-between file is not used.

3.2.2 Data Model

Neo4j uses a graph data model. It describes an arbitrary domain as a connected graph of nodes and relationships. Both nodes and relationships can contain properties. Nodes are often used to represent entities. A label is a named graph construct that is used to group nodes into sets. In total, the converted Neo4j database contains 7 node labels and 6 relationship types.

The SQL tables `movies`, `actors`, `series`, `genres`, `keywords`, `aka_titles` and `aka_names` were converted into identically named Neo4j nodes, while the SQL tables `movies_genres`, `movies_keywords` and `acted_in` were converted into similarly named Neo4j relationships. Of particular note is the `acted_in` (named `AS_CHARACTER` forming a relationship from actor to movie) relationship which is the only relationship which has properties, namely the `billing_position` and `character` properties.

Additionally, it should be noted that the data of Neo4j and PostgreSQL is not perfectly identical.
Firstly, because Neo4j relationships do not allow `NULL` properties the `billing_position` was defaulted to '0' if set to `NULL` on PostgreSQL. Similarly, the `character` property was defaulted to `N/A`. Note that there were already `N/A` fields in PostgreSQL's data, so the PostgreSQL `NULL` fields cannot be perfectly retraced in Neo4j.
Additionally, as mentioned earlier Neo4j does not have the foreign keys present in PostgreSQL. Furthermore, while this is far from a bad thing, Neo4j automatically removed a number of duplicate relationships that were present in the PostgreSQL database. Finally, the primary key values for the three tables that were converted into relationships were not included in Neo4j either, as they were both redundant and prevented Neo4j from detecting duplicate relationships.

3.2.3 Implementation

All required functionality was successfully implemented in Neo4j, with queries resolving within half a second. The query design is exceedingly similar to that of PostgreSQL. Though the syntax of Neo4j's Cypher is clearly different to SQL, the operations are generally very comparable. For example:

- `OFFSET` is renamed to `SKIP`
- `SELECT` is renamed to `RETURN` and moved to the back of the query.
- `FROM` is renamed to `MATCH` and moved to the front of the query.
- `x JOIN y` is changed to `(x)-[relationship]->(y)` to indicate relationship matching. While the syntax of this is very different, the thought pattern is largely the same.

Whenever specific data is requested through the API (e.g. number of movies for a genre, actors for a movie) the result is returned using a single query. When the full data of an object is requested several queries are required, up to 5 in the case of movies. In theory this could be done in a single query, however the current implementation does not do this in order to allow the reuse of the individual data functions.

3.2.4 Experience

Overall, while Neo4j's performance was certainly not poor, within this use case there was no reason to use it over PostgreSQL. Unfortunately, the quantity and complexity of the relationships were not enough for Neo4j to truly shine and PostgreSQL's performance ended up being superior.

Additionally, while developing with Neo4j the immaturity of the program occasionally showed. Primarily this was evident due to the lack of parametrized queries. Within the scope of this project it was not a cause for concern, but in a commercial product this is absolutely required. While it is possible to write code to safely handle input arguments manually, the fact there is no native support available is a poor selling point. Furthermore, currently Neo4j only supports basic data types (floats, doubles, string, integers, etc.) and does not allow for further constraints to be placed on properties, nor the use of some advanced data types (e.g. timestamps).

Regardless, none of these concerns are major concerns. We believe that as long as the use case provides sufficient opportunity for Neo4j to shine, meaning there are abundant relationships present, then it is certainly wise to take Neo4j into consideration as a database choice.