Database Course - Final Project

in Collaboration with the Test Course.

This is our final project for the Database Course, which takes root in the “Project Gutenberg”; the first online provider of free eBooks. The final project contains all of the necessary requirements set by our teacher Rolf-Helge Pfeiffer, and is combined with the assignment criteria from the test course. We have been working on this project from the 8th of May to the 24th of May 2017.

We started off by setting up some initial learning goals for this project, and during the development process we have been attending review meetings, which were based on those learning goals with both Helge, and teachers from the test course respectively. We got some useful input from the meetings, and it allowed us to follow a more organized plan for the remainder of our project period.

It has been challenging to extract the relevant data from the Project Gutenberg files, as well as getting use to writing cypher queries for the Neo4J system, but a great learning process nonetheless.

Throughout this report we will cover areas of:
- Which database engines are used.
- How data is modeled.
- How data is imported.
- The behaviour of our query “test” sets.
- Our recommendation for the database to use for production.

Have fun!

Athinodoros Sgouromallis, cph-as293
Ben Jones, cph-bj120
Frederik Alexander Vygg Larsen, cph-fl64
Rihards Pacevics, cph-rp84

Lyngby, May 24th, 2017.

Which Database Engines Are Used.

In this section we will give out reasons for the choices we have made regarding database engines. Considering the tasks at hand, being creating a system that connects to a database service only containing large data sets of books and cities, we didn’t even find it relevant to discuss the use of a relational database in its traditional sense. Instead, we wanted to explore the competitiveness of a Document Oriented, and Graph database respectively.

MongoDB

Mongo is document oriented[1], classified as a NoSQL database, Mongo uses a JSON similar structure for its documents, making it easy to have a denormalized database quite simply because everything is stored together; in this case each book has city attributes. This enables us to write fast and simple queries to the database, we don’t have to write complex joins or alike because everything is retrieved upon id lookup, e.g. for the query 1, you look up a city, and once that city is found, you don’t need a second query to find the book then because it’s already within that document.

Part of what makes MongoDB an interesting choice over an Relational Database Management System (RDBMS) in this use case, is the aforementioned design architecture for denormalisation. In MongoDB, to get the entire entity, we have to perform:

1. One index lookup on the collection (assuming the query is fetching by an id).
2. Get the content of a single database page (the BSON object).

In an RDBMS like MySQL with, e.g., 10 - 20 tables we will have to perform:

1. One index lookup on the “root” table (same conditions as before apply).
2. 10 - 20+ range lookups (based on the entity’s, from step 1, primary key) for the PK’s value in order to return the result set.

So in total, if we would have made this with MySQL the execution time would be much longer due to the multiple I/O (input / output) calls to the database.

Neo4J Graph

Neo4J, being a graph database[2], uses a graph structure[3], i.e. it consists of a limited set of nodes together with a set of ordered pairs of edges. It’s not a relational database, but it *does* however relates its data items directly to each other (if the relation states so). This way is also exactly what makes it possible to retrieve that very same relationship between two nodes because they are linked together. Like mentioned before, with MongoDB, we avoid

[1] https://en.wikipedia.org/wiki/MongoDB
[2] https://en.wikipedia.org/wiki/Graph_database
[3] https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

making joins in Neo4J to retrieve some related data, hence partly the efficiency not seen elsewhere. In a large data set like the example with books and cities from our project, we set it up for a relationship looking like the following;

(:Book)-[:MENTIONS]->(:City)
example usage of Cypher Query Language for our project.

This simple structure is the very benefit of using a graph database like Neo4J.

How Data is Modelled.

For our data modelling on the MongoDB we have used the mongoose module to create two models. The City model contains all information about the cities, including the coordinates that can be used to plot the position of the cities on the map.

The Book model contains the book information as well as City objects of any cities that are mentioned in that particular book. We realise that this gives us a lot of redundant data and in ‘the real world’ we would only store the city ids so that they could be searched for from the City model.

For the Neo4j side of the project the we create nodes from the Cities and Books and use Mentions as the edges to link each book to the cities that it mentions. This way we can also find all books that mention a city.

How Data is Imported.

For our data importing we have tried to take some of the work from the database by creating functions to format the data in a way that we can import it into both databases from the same files. We have created a huge JSON file from the books that contains Book objects that contain City objects for each of the mentioned cities in that book.

Also, Athinodoros made a function that would find any duplicate cities that are mentioned and try to figure out which one is the most likely to be the city mentioned. The function looks at the nearby country codes and finds the closest ones that match that city. Then this one is added to the book object.

The Behaviour of our Query “Test” Sets.

…

Our Recommendation for the Database to Use for Production.

...