Databases for Developers – Project hand-in

Mads Dam Eckardt Jensen - 060691-1149

Table of Contents

Introduction

This rapport explains the the development of an application which models and stores e-books in multiple databases from Project Gutenberg. Furthermore does it make use of geo-spatial data of cities in order to find occurrences of these in the books.

Hardware and database applications

I've primary used a working station with the following specifications:

- Intel(r) Core(TM) i7-4790S CPU @3.20 GHz
- 8 GB DDR3 SDRAM
- Samsung SSD 840 250 GB
- Ubuntu 16.04.2 LTS (x86_x64)

I have chosen to use the document-oriented database MongoDB 3.4 (https://www.mongodb.com/) which stores its data in JSON-like documents and a PostgreSQL (https://www.postgresql.org/) database which a object-relational database.

To build the front end I created a Ruby on Rails 4.0.4 application which is a MVC (Model, View, Controller)-framework for the Ruby programming language. Furthermore are all scripts made during this project, running in the context of the Rails application to gain easy access to the database. I initially wanted to use Neo4j but I encountered problems during the integration to the spatial framework from Ruby which enables spatial operations on data. Therefore I chose to use MongoDB instead.

I wanted the application to feel dynamic without having the need for changing page all the time. In order to accomplish this, I made use of the AngularJS framework to perform AJAX queries against my server. Furthermore did I use Bootstrap for styling, Jquery for easy manipulating the DOM of the page and last but not least LeafletJS which is a library for interactive maps similar to Google Maps. The advantage of Leaflet was that I could avoid registration of my application – it is open-source.
The application also makes use of libraries supporting spatial queries against geographical data in PostgreSQL and MongoDB.

Collection of data

Collection of E-books

The books I have used for this project is the April 2010 DVD edition of the Project Gutenberg acquired from here:

https://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project.

This DVD contains roughly 29.500 archived e-books and these are contained inside a .ISO-file. I used the following script to unzip all the archives and move the files my home folder:

```
DIR="/home/mads/books"
UNZIP_DIR="${DIR}/files"
mkdir -p "${UNZIP_DIR}"
for ZIP_FILE in $(find ${ZIP_DIR} -name '*.zip')
do
  UNZIP_FILE=$(basename ${ZIP_FILE} .zip)
  UNZIP_FILE="${UNZIP_DIR}/${UNZIP_FILE}.txt"
  if [ ! -f "${UNZIP_FILE}" ] ; then
    unzip -o "${ZIP_FILE}" -d "${UNZIP_DIR}"
  else
    echo "${ZIP_FILE##*/} already unzipped. Skipping."
  fi
done
```

After looking through the files I discovered that collection contained multiple file types but for this project I’m only interested in .txt files, so I made the following snippet to clean the data a bit:

find . -name '*.jpg' -or -name '*.jpeg' -or -name '*.png' -or -name '*.html' -or -name '*.htm' -or -name '*.bmp' -or -name '*.gif' -delete

The above snippet deletes a lot of pictures, HTML-files etc. which are unnecessarily for the project.

Collection of cities

The city data is acquired from here: http://download.geonames.org/export/dump/cities15000.zip

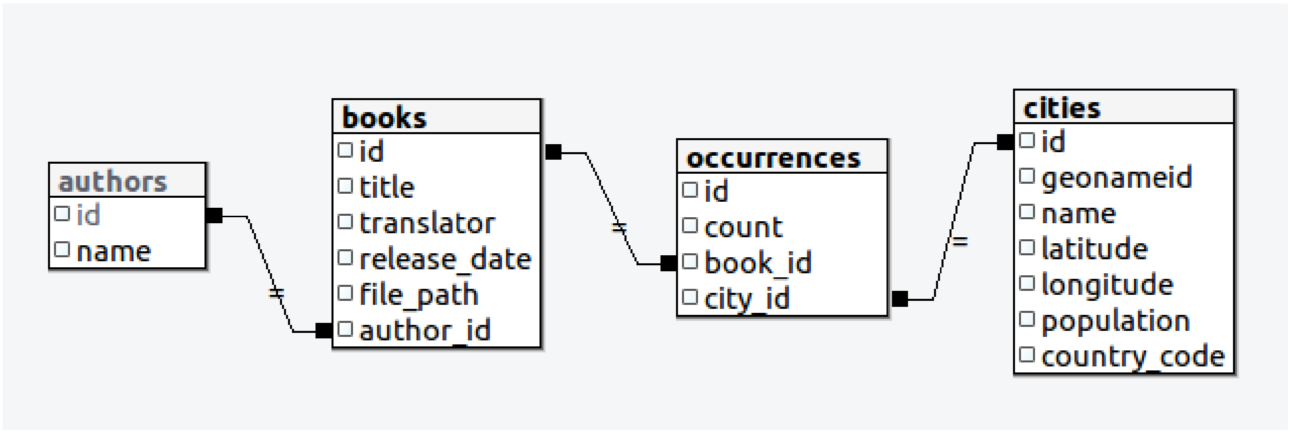
This file contains a tab-separated file with around 23.000 cities having a population more than 15.000 citizens. A description of the data contained is listed at the bottom of this site:

http://download.geonames.org/export/dump/

Importing data to PostgreSQL

Initially I created migrations in order to form the schema which would be used for the PostgreSQL database. As stated by rubyonrails.org: "*Migrations are a feature of Active Record that allows you to evolve your database schema over time*", which means that for each change that you want to do to the schema, another migration gets added so you can go back and forth in time to see how the schema developed. Furthermore does Rails applications offer a easy syntax for editing the schema. The migrations are contained in /db/migrate/ folder.

When the migrations are run, a db/schema.rb file is generated which contains an overview of the schema in its current state. Here is an overview of the schema made with PgAdmin III:



After I made the schema I created models which represented the objects in the database. In all Rails applications exists a built-in ORM(Object-relational mapper) which binds the models to the tables in the database. It makes creating, updating and deleting of objects very easy. I created models for books, authors, cities and occurrences. The models for the PostgresSQL database is located in /app/models/postgresql/.

In order to find city names in all the books, I had to import the collection of cities first. The script I created for the task can be found in lib/load_cities.rb. The script assumes that the data file is located at /data/cities/cities15000.txt.
The script essentially reads each line of the file and splits it into an array with a tab-character as delimiter. I chose to import the following columns: geonameid, name, latitude, longitude, population, country_code.

```ruby
#!/usr/bin/env ruby
file_path = Rails.root.join('data', 'cities','cities15000.txt')
count = 1
File.open(file_path, "r") do |f|
  f.each_line do |line|
    arr = line.split(/\t/)

    city = City.new
    city.geonameid = arr[0]
    city.name = arr[1]
    city.latitude = arr[4].to_f
    city.longitude = arr[5].to_f
    city.population = arr[14]
    city.country_code = arr[8]
    city.save
    puts "Line: " + count.to_s if count % 500 == 0
    count = count + 1
  end
end
```

After I imported all the cities, it was time to process the books. For this purpose I made the following script /lib/load_books.rb.
This script contains multiple safety features because I experienced a lot of errors while parsing the data the first couple of times. Before the script starts to read the books, it creates an index of all the files that it has to parse to a file /lib/files_index.txt. This ensures that the order of the files are the same if I had to re-run it. I further more keep an index of the current file number, so I know where to continue from if I encounter a problem.

The script runs through every line of every book and looks for meta data if it was not yet found. This meta contained information about the title, author, release date and translator.

Initially I looked up every word in the database one by one, but found out that it would take around 22 days to complete on the system. For each book i took the completion time and calculated an average, so I could see how well my optimizations worked.

The next thing I did was to ask the database for cities where the city name was in an array. The array being the current line split by a space character. That reduced the estimated time of the whole script to around 32 hours. It drastically reduced the amount of queries against the database, but that was still not enough.

The next thing I did was to create a bit heavy regular expression in order to detect the city names. By doing this, I also found out that if I had continued with the previous approaches I would have missed a lot of cities as many of the city names consists of one or more words like New York City, Andorra la Vella, Ras al-Khaimah etc. The previous approaches only worked for city names containing exactly one word. My regular expression could handle this.
Furthermore I created an in-memory index of the city names together with the id of row in the database. For each match that the regular expression returned I immediately had the id of the city in the database. This spared the database for a lot search queries. My calculations estimated that the whole process time was reduced to around 6.5 – 7 hours which I found satisfying.

One thing to note is that there are cities which have exactly the same names. I only register occurrences for the first city that I match, so if two cities are named Paris, only the first occurrence of that city would be used.

After the whole process was done I had imported the following data:

Cities	23.602
Books	32.472
Authors	11.597
Occurrences	1.705.156

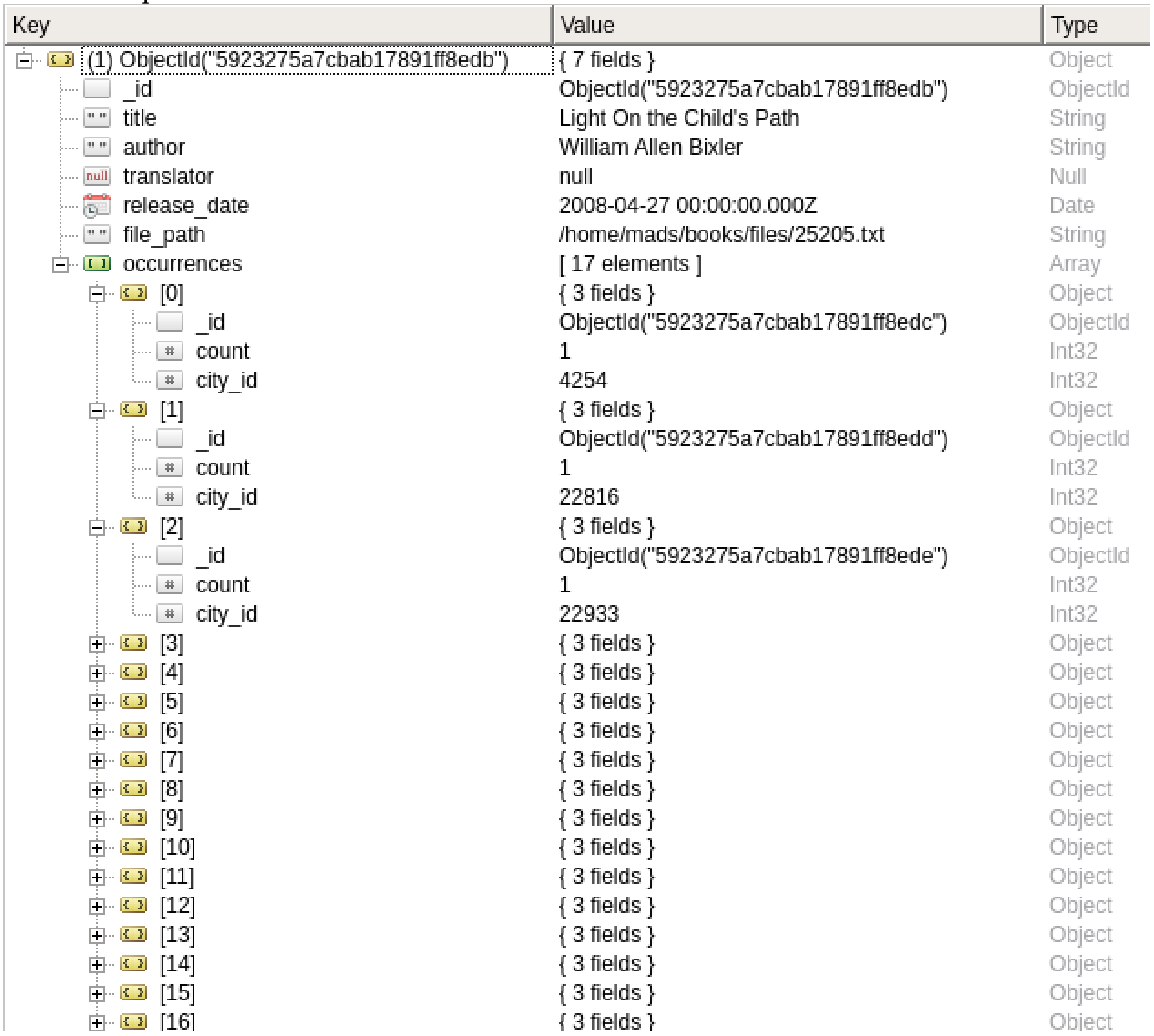
When I developed the script I encountered multiple problems, some of which I have listed here:

- In case book title not found, the program would initially crash as no occurrences could be added, due to the book is not uniquely identifiable. Later I accepted that I might not find the book title, so in order to identify the book in the database, I used the file name as the identifier.
- If the same book occurred more than once, for example at a re-run, the count of the occurrences would be off as additional occurrences just got added to the existing. I made a failsafe so that if book occurred again – the counts of occurrences would reset for the book.
- The release date have multiple formats for example "March 2, 2017", "3/2/2017", "2-mar-2017" etc. so in order to handle this, I had to test for multiple ways of writing the date. I used regular expressions for the most frequent ones and in case it was unable to match the date, I simple skipped it.
- Invalid byte sequence UTF-8. Some of the files contained multiple errors according to UTF-8 encoding, and I handled this by stripping invalid characters from each line before parsing.
- Multiple occurrences of lines which started with ”Author: ..” ”Translator: ...” etc., which initially would result in an overwrite. Now the meta data is only stored if its missing.

Importing data to MongoDB

I chose to use the already parsed data that I had in my PostgreSQL database instead of creating a new script for reading all the books. I though about the export from PostgreSQL to MongoDB and how the data could be structured. An easy way to do it would be to map the tables in the PostgreSQL to collections in MongoDB. By doing this the data would be imported, but it would properly not change anything in performance as the way to lookup the data, would be more or less the same. But this is only a guess!
So I decided to change the data a bit before importing into MongoDB in the hopes of it could result in performance gain.

I rearranged the data so all the occurrences were embedded in the book document, but still with a reference to the city document. My goal was to try to reduce the amounts of joins which were performed by the PostgreSQL. That resulted in two collections in the MongoDB database: cities and books. Furthermore did I include the name of the author in the book document directly.
This is a sample from the books collection:

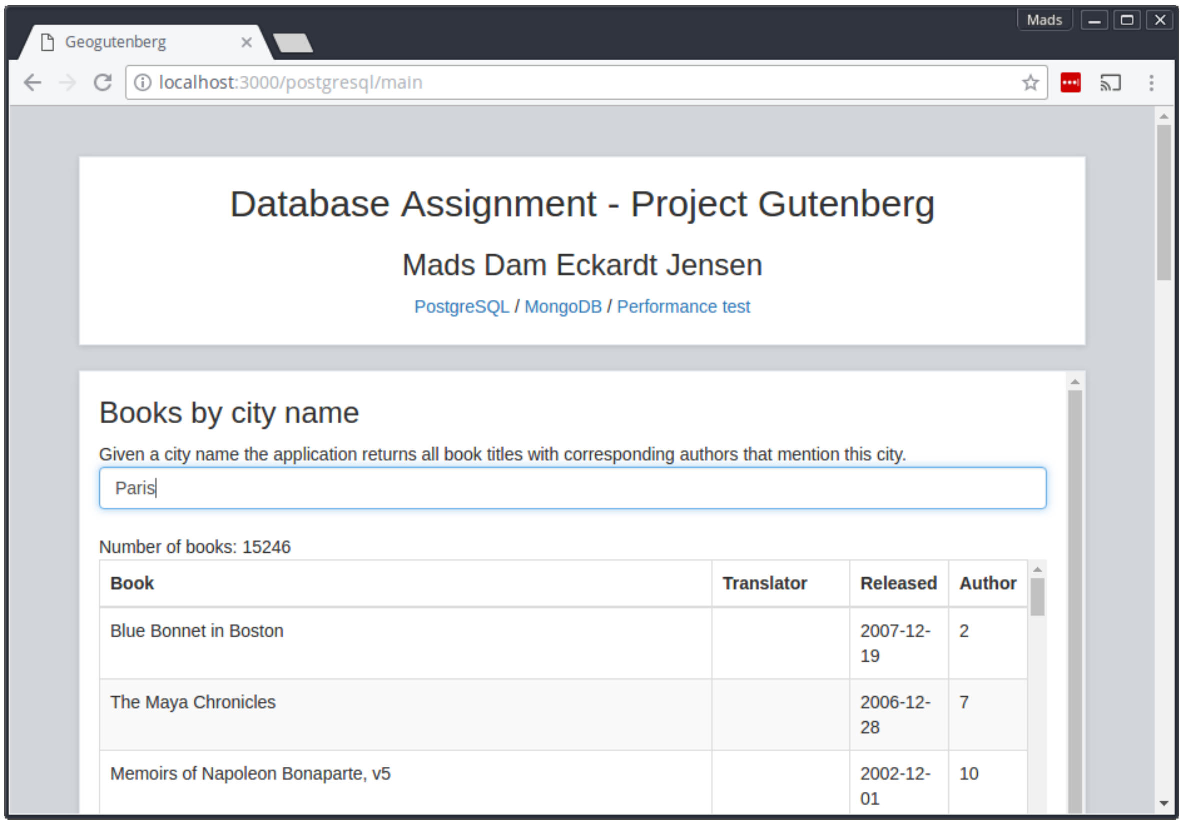


Key	Value	Type
(1) ObjectId("5923275a7cbab17891ff8edb")	{ 7 fields }	Object
_id	ObjectId("5923275a7cbab17891ff8edb")	ObjectId
title	Light On the Child's Path	String
author	William Allen Bixler	String
translator	null	Null
release_date	2008-04-27 00:00:00.000Z	Date
file_path	/home/mads/books/files/25205.txt	String
occurrences	[17 elements]	Array
[0]	{ 3 fields }	Object
_id	ObjectId("5923275a7cbab17891ff8edc")	ObjectId
count	1	Int32
city_id	4254	Int32
[1]	{ 3 fields }	Object
_id	ObjectId("5923275a7cbab17891ff8edd")	ObjectId
count	1	Int32
city_id	22816	Int32
[2]	{ 3 fields }	Object
_id	ObjectId("5923275a7cbab17891ff8ede")	ObjectId
count	1	Int32
city_id	22933	Int32
[3]	{ 3 fields }	Object
[4]	{ 3 fields }	Object
[5]	{ 3 fields }	Object
[6]	{ 3 fields }	Object
[7]	{ 3 fields }	Object
[8]	{ 3 fields }	Object
[9]	{ 3 fields }	Object
[10]	{ 3 fields }	Object
[11]	{ 3 fields }	Object
[12]	{ 3 fields }	Object
[13]	{ 3 fields }	Object
[14]	{ 3 fields }	Object
[15]	{ 3 fields }	Object
[16]	{ 3 fields }	Object

Front-end application

The front-end application consists of multiple views where the user can perform the different types of queries which are stated the project description. For all the types of queries are tables presented with data some of which contains links which can be clicked. For the last three types of queries is the user presented with a world map which plots most of the data.

The following are screen shots from the running application:



Database Assignment - Project Gutenberg

Mads Dam Eckardt Jensen

PostgreSQL / MongoDB / Performance test

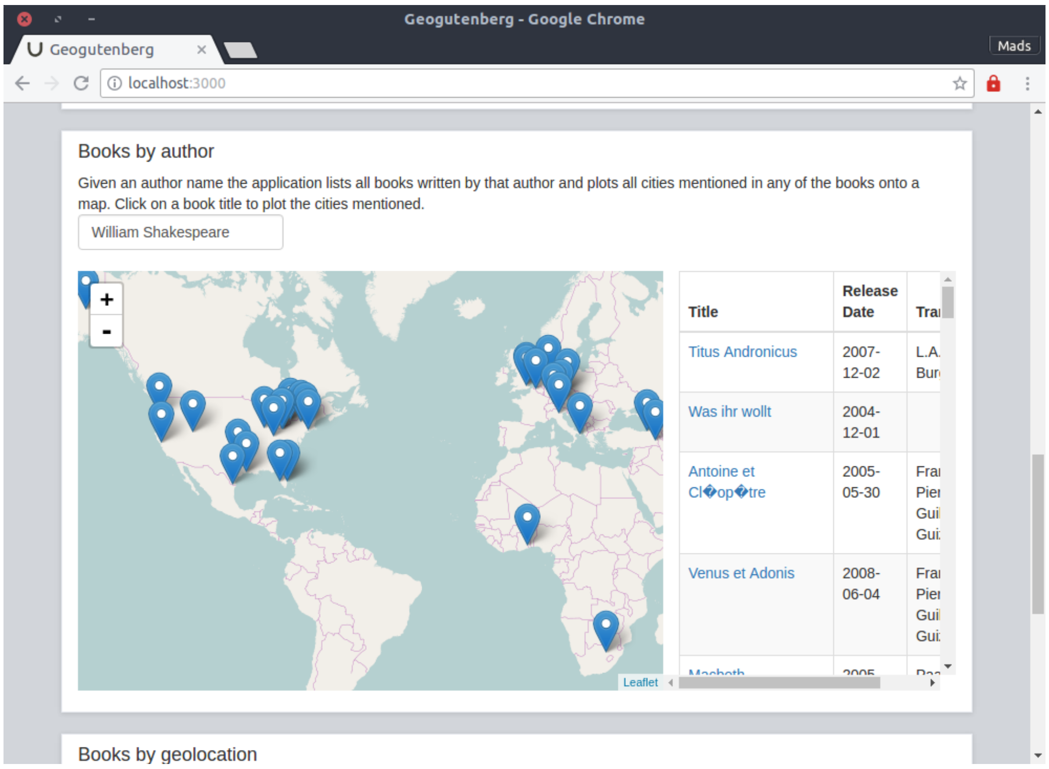
Books by city name

Given a city name the application returns all book titles with corresponding authors that mention this city.

Paris

Number of books: 15246

Book	Translator	Released	Author
Blue Bonnet in Boston		2007-12-19	2
The Maya Chronicles		2006-12-28	7
Memoirs of Napoleon Bonaparte, v5		2002-12-01	10



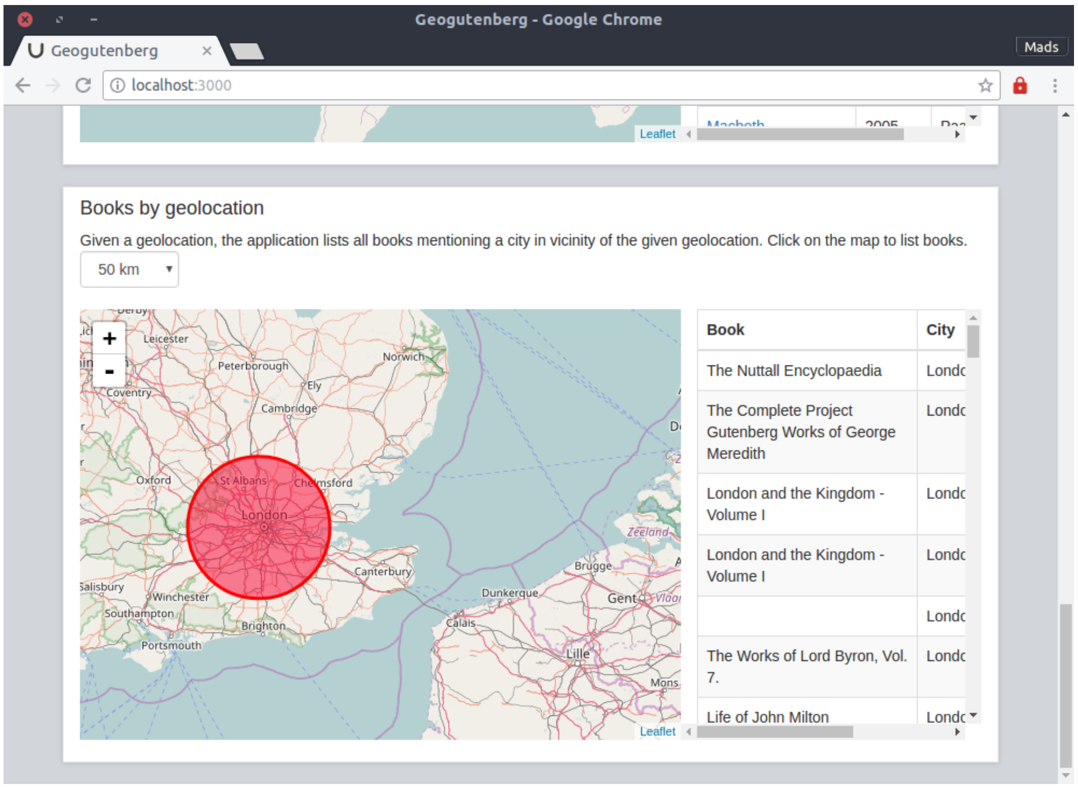
Books by author

Given an author name the application lists all books written by that author and plots all cities mentioned in any of the books onto a map. Click on a book title to plot the cities mentioned.

William Shakespeare

Title	Release Date	Tra
Titus Andronicus	2007-12-02	L.A. Bur
Was ihr wollt	2004-12-01	
Antoine et Cl�op�tre	2005-05-30	Fra Pie Gui Gui
Venus et Adonis	2008-06-04	Fra Pie Gui Gui
Macbeth	2005	Da

Books by geolocation



Books by geolocation

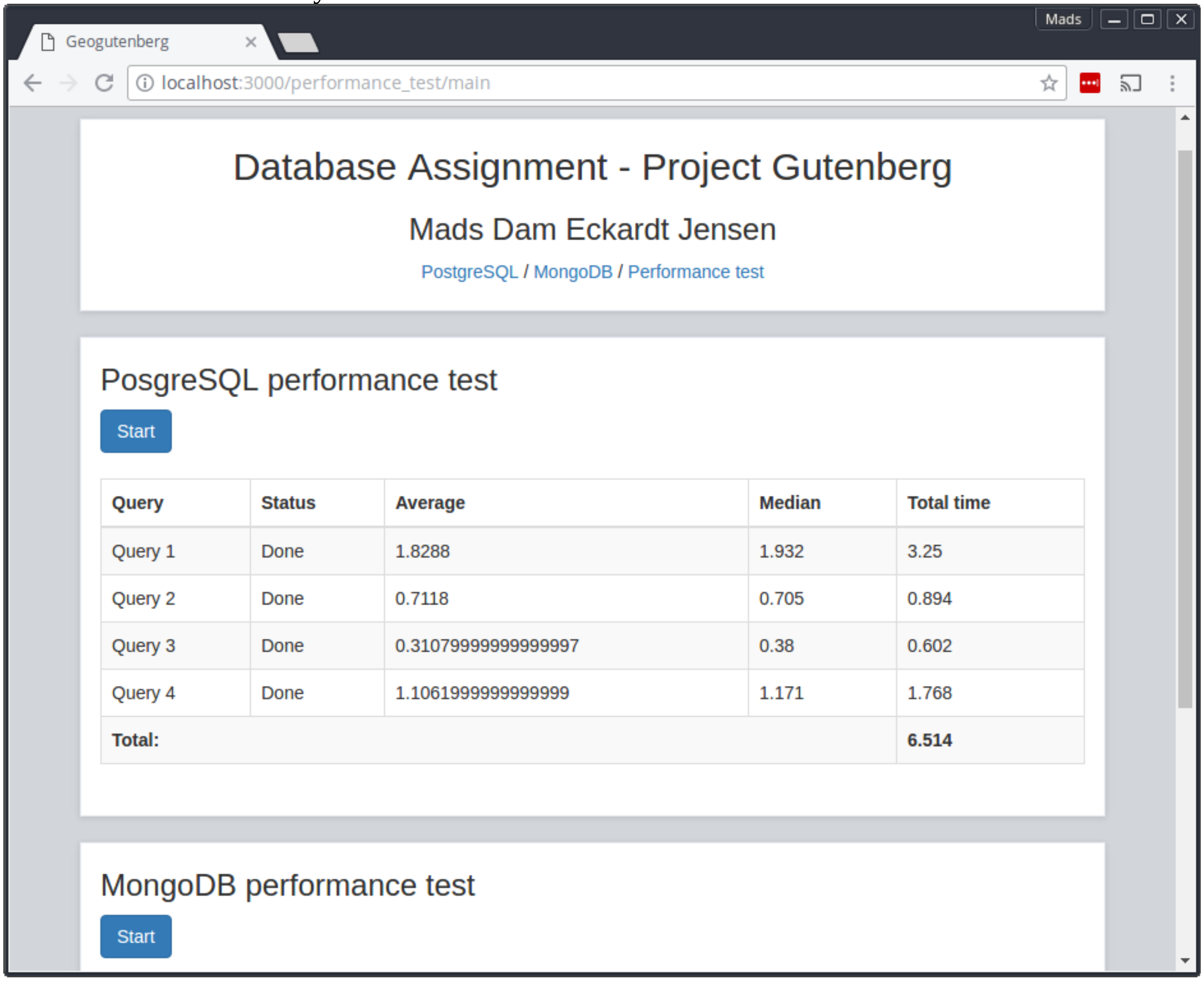
Given a geolocation, the application lists all books mentioning a city in vicinity of the given geolocation. Click on the map to list books.

50 km

Book	City
The Nuttall Encyclopaedia	Londo
The Complete Project Gutenberg Works of George Meredith	Londo
London and the Kingdom - Volume I	Londo
London and the Kingdom - Volume I	Londo
	Londo
The Works of Lord Byron, Vol. 7.	Londo
Life of John Milton	Londo

Performance tests

In order to measure the application behavior I decided to do the test from client side. I choose to do this because even though that query times on the two different databases could be interesting, it's important to keep in mind that the data they return are different. The way it is structured and the size of it. Therefore I believe that the overall experience with the application could differ in spite of the performance of the database.
Here is a screen shot of my test view:



Database Assignment - Project Gutenberg

Mads Dam Eckardt Jensen

PostgreSQL / MongoDB / Performance test

PosgreSQL performance test

Start

Query	Status	Average	Median	Total time
Query 1	Done	1.8288	1.932	3.25
Query 2	Done	0.7118	0.705	0.894
Query 3	Done	0.31079999999999997	0.38	0.602
Query 4	Done	1.1061999999999999	1.171	1.768
Total:				**6.514**

MongoDB performance test

Start

I chose to perform 5 queries for each of the types of queries specified in the project description for both databases. The view presented for the user, contains two boxes which each runs a set of queries against the different databases. They both ask for the same books, cities, authors etc., so the times measured are more comparable.

Each of the tests represents the same query against the server as the dedicated front-end for each database, but this is without a visual representation of the data. The data is received, but then tossed away.

The following are the results of the queries:

PostgreSQL

Query	**Status**	**Average**	**Median**	**Total time**
Query 1	Done	2.5996	3.034	4.597
Query 2	Done	0.8457999999999999	0.843	1.009
Query 3	Done	0.2112	0.252	0.409
Query 4	Done	1.0698	1.149	1.655
Total:				**7.67**

MongoDB

Query	**Status**	**Average**	**Median**	**Total time**
Query 1	Done	108.46340000000001	111.495	196.314
Query 2	Done	0.8484	0.835	1.069
Query 3	Done	10.9256	14.15	14.864
Query 4	Done	212.70459999999997	207.267	391.682
Total:				**603.929**

We see that PostgreSQL outperforms MongoDB in almost every query. Clearly something in the MongoDB does not perform well.
Query 1 starts off by using a predefined city_id for each of the 5 runs to return books which contains occurrences of the particular city. In the PostgreSQL database the query starts by finding occurrences in the occurrences table and returns book_id. The query then looks up books with those book ids and joins the author table to return the name of the author.
Even though the MongoDB has an index on "occurrences.city_id" the performance is very poor. Going through forums like Stackoverflow, I found that this behavior seems to be common with large arrays.
In our case, one book document holds up to thousands of occurrence objects in the current structure. This problem seems to be occurring throughout all the queries for the MongoDB database with the exception of query 2.
In this case, the query looks up a book and returns all the cities occurring in the book. When MongoDB have found the correct book, the occurrences of cities are already embedded in the document. The only thing needed afterwards is to look up the city name in the cities collection. This could be optimized even more by having the name of the city in the occurrence instance.

Generally did MongoDB behavior stay the same with and without the index on "occurrences.city_id", so assuming it is unused, the process of work for the 3 slowest queries are

presumably to go though every book and every array containing all occurrences. This adds up to roughly 32.500 books with over 1.7 million occurrences objects to go through for every query. The way I chose to structure the data in the MongoDB did clearly not pay off for the most part. I avoided multiple joins with the exception when the name of a city is required, but the way the database scans the data in the current state is too inefficient.
In order to optimize the database, I would first reallocate all the occurrences into its own collection to have a more similar structure to the PostgreSQL schema. I would assume that when doing so, I would achieve more or less the same performance at least.
It is currently on the todo list to see what is going on in the MongoDB queries. For this I'll make use of the built-in tools to explain the queries. So far I've established that queries in fact did make use of the index so there have to another underlaying problem.

Conclusion

So what did I learn making this application. First of, the fact that I chose to do a web application with typeaheads and maps challenged the data structure of both databases. In order to make a typeahead for example an author requires basically the name and an id for the author to work. In an already relational database this would presumable be in a table for it self. In my current MongoDB database the author name is an attribute on the book document. In order to get a list of authors the MongoDB have to go through every book to find the author and furthermore distinct the values returned as they appear redundant if an author has written multiple books.
This type of problem did I take into account when designing the relational schema for the PostgreSQL database. The same goes for city names.
If I have chosen to another type of application for example a command line interface, this structure would properly have been omitted. This would also have resulted in fewer joins in the PostgreSQL database.
Another thing to take into account is that this project is not intended to have data updated, created or deleted. In this project, we more or less knows everything about the data we are modeling, and we are not likely to add new attributes to any of our data. This knowledge suppresses some of the advantages of the document oriented database as the structure of these are easier to remodel. Further more does the lack of CRUD capabilities spare us for thinking about data consistency, transactional behavior and redundancy which would have been essential in a multiuser work flow.
In current setup, the MongoDB also have a problem if we were to store an author without referencing any books. That would not be possible with the current structure.
If we were to support that, the collections we would have to make, would become more structured like the relational database.

Despite the performance issues that I had with the indexes of the MongoDB database, I would still go with a relational database for project like this. This is due to that the well known structure of data fits well in tables, the ease of queering a single attribute of an object in relation to typeaheads, less redundancy in data and meta data of the database, better support for multiple user work flows working the data if that were to be supported and for the moment – the performance.
If the data were to contain unknown and and more differentiated data, I would properly reconsider a database like MongoDB.