CIS 450/550 Final Project
Alex Matthys
Drew Boyette
Archit Sharma
Lucas Tai-MacArthur

Write-up

Abstract

Inspired by the massive amounts of data collected by US government regarding various important social factors, our group decided to come up with an application which would combine these massive datasets and provide data regarding various factors with respect to different states in America.

We have developed a web application which displays an interactive map of the United States. The user can interact with the application by clicking on a particular state. When the user clicks the map they will be presented with interesting facts about the state selected, generated from querying datasets about broad categories provided by the US government from data.gov and from the homeland infrastructure foundation-level data.

The motivation behind this idea was to choose a topic that has a plethora of databases with information relating to this topic, and states in America is certainly one of them. We chose attributes like hospitals, obesity, cities, population, sport venues and universities because these are very general categories that should have relationships to one another. For example, you might expect a state with a higher population to have a higher number of universities, or a state with a higher obesity rate to have fewer hospitals. This idea was implemented by finding databases containing this information and populating an AWS MySQL instance.

Modules, Architecture, and Technical Specifications

The architecture of the system was designed with extensibility and modularity in mind. It was meant such that if one aspect of the system changed (database provider, database type), the amount of work necessary to bring the system back to full user functionality was minimized. To this end, the system was designed in three parts. The **frontend server**, the **content database**, and the **login database**.

Starting with the frontend server, this component of the system acted as the intermediary between client and database services. It interpreted the client requests and marshalled and modified them in the correct way such that they could be interpreted correctly by external database services. It runs as a Node.js server on a Heroku managed instance. The client facing views are generated on the server by the Express.js view engine and handled on the client machine itself through jQuery. To manage the different database interactions, the application uses the MongoDB and Mysql drivers for Javascript to make requests to our AWS and MLab endpoints.

The second component of the system is the content database. It is here that our cleaned, indexed, and separated data about the United States (6 different tables) is stored and accessed. The database is of MySQL 5.7 type running on an AWS managed RDS instance. Our original datasets were in json, csv and xls formats which we cleaned using python scripts (as discussed later) and then imported into the AWS RDS instance using the Table Data Import Wizard feature of MySQLWorkbench. The queries used to access these datasets are all SQL queries and we created a view to support one of our more complex queries.

The final component of the system is the login database. This database, as its name implies, servers as the touchpoint for information about users, namely their usernames and passwords. It is accessed through the login system, which both queries the data already in the database, but also registers new users by adding new documents. We validate data against an email/password syntax to ensure that we aren't adding any junk data into our database. In specifics, we are using a small MongoDB instance running on MLab, a managed Mongo provider. We can and have imported and exported new tables of users in JSON format to bulk add or bulk remove the created users who are using the application. On the serverside, we use the MongoDB driver for Javascript to make asynchronous queries and additions to the instance.

Data Instances

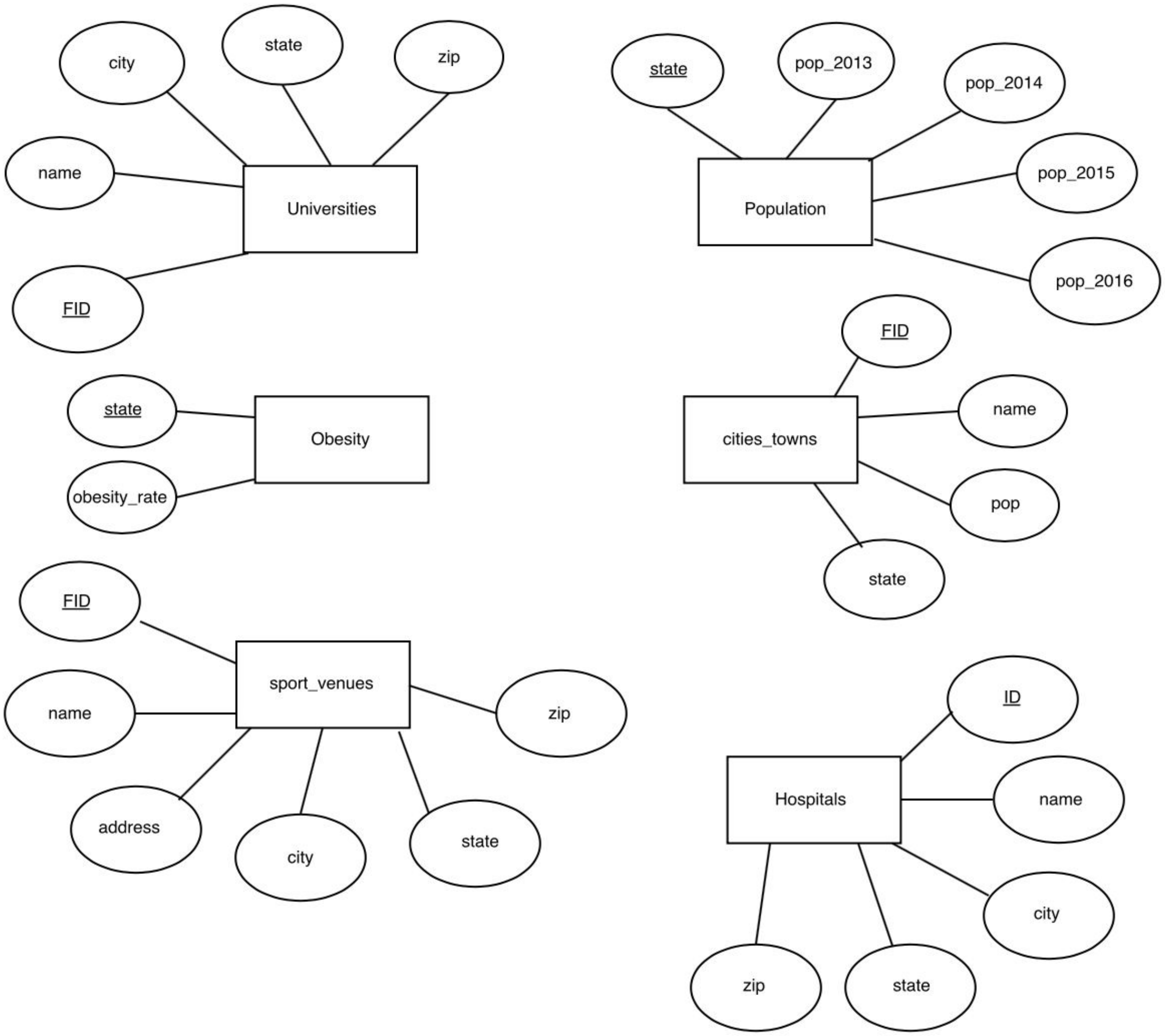
Here we haved used the following datasets:

- Universities data by state (source) - found and imported into MySQL in Csv format
- Population data by state (source) - found and imported into MySQL in Xls format
- Obesity data per state (source) - found and imported into MySQL in Json format
- Cities and Towns in the US data (source) - found and imported into MySQL in Csv format
- Sports Venues per state (source) - found and imported into MySQL in Csv format
- Hospitals in the US per state (source) - found and imported into MySQL in Xls format

Our normalized relations schema is:

Universities(FID, name, city, state, zip)
Population(state, pop_2013, pop_2014, pop_2015, pop_2016)
Obesity(state, obesity_rate)
cities_towns(FID, name, pop, state)
sport_venues(FID, name, address, city, state, zip)
Hospitals(ID, name, city, state, zip)

ER Diagram:



We are storing the user login information (username, password) in MongoDB on Mlab, the login details for which are as follows:
Mlab credentials (source)
username: cis-550-finalproject
password : cis450550

To Connect to our AWS MySQL Database:
Endpoint: **cis550sql.cl12qlaqfgyt.us-west-2.rds.amazonaws.com:3306**
Username: cis550krewsql

Password: cis450550team

Data Cleaning

We retrieved five of our six datasets in either csv or xls form, and one dataset (on Obesity rates) in Json form. Because the Obesity dataset only had 50 rows, it was very easy to quickly scan through it and ensure that the data was clean and formatted correctly. However, for the other five datasets, this was not the case. For each of the five datasets in either csv or xls form, we wrote unique python scripts tailored to a particular dataset to read in all rows, store values in a dictionary or a set, and then check for missing or poorly formatted values. While checking for duplicate rows, duplicate primary keys, empty/null values, and special characters, some of the specific data formatting issues which were caught by our Python scripts include:

- When cleaning the Cities and Towns dataset (38,000+ rows!), we discovered 26 duplicate entries. The output of running the cleaning script 'datacleaning.py' on 'Cities_and_Towns_NTAD.csv' resulted in 26 entries formatted as follows:
    - We have found a duplicate: ('Gibson City', '3407', 'IL') at row(s) ['11960', '35041']
- When cleaning the Universities dataset, we found 53 empty cells in the datatable. The output of running the script 'datacleaning_POST_SECONDARY_UNIVERSITIES.py' on "POST_SECONDARY_UNIVERSITIES.xls' resulted in the following, with 53 more 'empty cell' entries:
    - done checking duplicate university names
    - done checking primary keys
    - We have found an empty cell at location: (341, 6)
- When cleaning the Universities, Hospitals, Sports Venues, and Cities and Towns datasets, we had to clean out any special characters (such as city names with an accent mark over a letter).

Optimization Techniques

In order to greatly improve the efficiency of our sql queries, we used several techniques to optimize it. Firstly, one of our queries was to select universities in cities that contain the maximum number of sport venues. One solution would be to create a table holding venue counts for every city with every query run, however this adds an unnecessary cost to our sql query and drastically increases the runtime. We decided to create a separate view stored in our database called VenueCount, which allowed us to access the number of venues per city very quickly and decrease the query runtime. The second optimization technique we used was creating indices on the tables. We noticed many queries were joining tables on city name, and considering these tables have thousands of entries, this results in a extremely large runtime. To improve this, we create indices on the following tables:

- Index on (city) for table Universities
- Index on (city) for table Sports Venues
- Index on (city) for table Hospitals
- Index on (name) for table Cities and Towns

This reduced the number of data pages the number of database data pages that have to be visited/scanned, hence reducing the runtime time of any query joining on this index.

After the user clicks on a state, our backend first runs all sql queries and stores the results. Before using indices, it took 6.7 seconds for our web app to execute all 14 queries. After creating these indices, it now takes 1.4 seconds to execute all 14 sql queries. Lastly, some of the more complicated queries required access to many of the databases we were using, such as**:** For state ‘x’, return the five largest cities by population size, the number of universities in each city, the number of hospitals in each city. One implementation of this query would be to create a separate table for every state, and for each query run determine which table to use. Although fairly efficient in terms of runtime, this takes up a large amount of unnecessary space. We came up with an alternative solution by creating nested SELECT FROM statements which essentially created temporary tables with every query run, and only tables that we require for the query. This optimization technique also greatly improved the runtime and space of our queries.

Technical Challenges

We experienced many technical challenges which we had to work together to find ways to overcome. Some of the more notable challenges we experienced were:

- Special characters when uploading data through MySQLWorkBench to our MySQL database. For some reason, MySQLWorkBench does not accept text with special characters (such as accents), and as such when we were importing a csv file, the data import wizard repeatedly crashed. It took some investigation to finally realize that ten thousand rows into our data set, there were a few city names which included accents. After discovering this, we were able to write python scripts to quickly check and clean these out.
- We originally created an Oracle relational database in AWS. However, we were using the node homework assignment completed in class as a good template for how to connect, and the homework assignment had us connecting to a MySQL database, not an Oracle database. As such, we had to delete our Oracle database and instead create a MySQL database hosted on AWS to allow for easier connecting between the front end and the back end.

Future Extensions and Use Cases

We have many different options when it comes to brainstorming future extensions on our project. Each of the different extensions we could implement would cause a unique use case. Our favorite idea of a future extension is:

- We could implement the map such that each state displays important information about tax rates, businesses existing, government restrictions, and other state-level legal fees which each state has regarding starting a business. Thus, a use case for this future extension would be when a user is planning to start a business, he/she can refer to our map to quickly compare the ease of starting a business in different states, and as such come to a decision about which state the business should be founded in.

Additional Features

1) The first additional feature of note is a live twitter feed displayed on the map page. It displays the feed of the US State Department (@StateDept) which we thought was fitting for a display of data collected from the various entities within the United States Government.
2) The second feature of note is the map entity. This is a jQuery plugin which allows us to display a highlighted and clickable map. By building HTTP requests which pull data from the map and send it to our Node.js server, we are able to incorporate it as a fun and responsive way for a user to select a state they have questions about
3) The third feature worth mentioning is the selectable query system. We have a number of queries of interesting data (obesity, education, healthcare) that we wanted to showcase to provide users insights within the states themselves, but also in differences between them. By providing us with a selectable query which (through clientside Javascript) can then query the database and display the results, we provide an easy way for information to be displayed.
4) The final feature of note is the login system. This system queries a database and allows only registered users to access the system. It includes full support for both ensuring that a user is valid, but also for the arbitrary registration of new users. Our serverside javascript talks through the MongoDB driver to an instance hosted on MLab

Division of Work

Alex Matthys

- Data cleaning (python scripts)
- Data import
- ER Diagram
- Mysql queries
- Data base research
- Testing

Archit Sharma

- Mysql queries
- Query performance and optimization
- Data analytics
- Database research
- Debugging

Drew Boyette

- Data import
- Mysql queries
- Database reasearch
- Mysql database management
- Testing

Lucas Tai-MacArthur

- Front end (including the graphical design)
- MongoDB database management
- Incorporation of of twitter feed