CS 316 Project Report

Nathaniel Brooke, Gouttham Chandrasekar, Harry Liu,
August Ning, Gabriela Rodriguez-Florido

December 10th 2018

We have adhered to the Duke Community Standard in completing this assignment.

1 Introduction

Our project is called Health Map, a health data visualizer that presents health data on various diseases / conditions from states in the USA and different countries, and will color in a map depending the statistics of those locations depending on the incidence, prevalence, mortality, etc., of the chosen disease / condition.

The inspiration of this project stems from the sheer amount of data available on health statistics. In the USA, health and health care is widely available from data.gov, and there is good documentation on a wide variety of diseases and conditions. In addition, there is a large amount of data from around the world from the World Health Organization. The health data is also many times split by various demographic factors, such as age, race, gender, and year filters on each data entry.

Another inspiration is that we have a premed on our team, and we wanted a way to bridge the gap between the layman's understanding of diseases and health conditions with technical classifications that already exist but hard to understand in the medical world. We also want to know, for example, how many people have died from Syphilis in North Carolina in 2005 between the ages of middle age individuals who are women and also African American. We can do this with our project. This helps anyone understand large datasets without having to tediously parse it.

With the sheer number of disease and location combinations, there are a large number of unique visuals that can be displayed on map APIs. Afterwards, the demographic data can be passed through a combination of filters that can be applied to a disease location combo, and the applied filters will be reflected on the map. The utility is flexible, and hopefully can add additional features such as more conditions, data on more locations, and any new demographic data that correspond to the diseases to make new filters.

Our goal is to create a data visualization tool for health data. Although we do not do significant data processing or metric generation, we do provide and clean visual and intuitive interface to explore the current health data in our database. It is helpful for curious individuals to explore these health data sets. The visual display and filters abstract the numbers to make the large health data sets more digestible.

Health data visualization is not new, but we wanted to make our own interface, especially one that uses plot.ly and a map API. There is a website called healthmap.org which we found out about after we bought our own website domain name, health-map.org (no relation). healthmap.org dynamically shows outbreaks around the world, and many websites have various visualizations of data, but usually is specialized. Our focus is on the data(base), so our goal is to gain a sizable amount of data and let the user work with the data and play with the visuals.

2 System Overview

The general system overview of our project can be broken down into a simple front end and back end. The back end is based off PostgreSQL and front end is off of Python Flask. Our database is set up according to the schema outlined in our original report.

Once the data was cleaned and formatted according our outlined schema, we imported it into PostgreSQL, which allows us to do standard SQL queries. Data is first selected by by disease / condition filters, and then depending on either global or USA data, the data is first graphed on a map and filters are available for users to modify to change the data presented. This plotting is set up with Python Plot.ly and supported by the Flask framework to set up the website.

The backend takes in default values from the user, and computes the SQL queries to return the data to Python using Psycopg2. The inputs are cleaned by Psycopg2 to prevent injection attacks, and allows us to use the SQL we learned in class. The backend in Python is then responsible for using the map and plotting API to display the values onto the map. When a user updates a value, the backend will re-query the database with the updated values with demographic filters, and the Python backend will re-plot the maps accordingly.

The frontend is in charge of interacting the with the user, setting up the website / organization, and the general user experience. Using Plot.ly, we can plot our data from the database and present it using the map API. The map uses colors to represent count, which helps abstract out the numbers into more visual incident/mortality data by location. The frontend also allows the user to filter out demographic data on the conditions, and will pass the user's updates to the backend to recalculate the values, and display them.

3 Design Choices

There are four important design choices that we considered when completing this project. They can be broken down into choosing PostgreSQL as our backend database manager, using a relational database instead of a tagged/structured database, using Flask for our website/frontend, and comint up with the structure of our schema.

3.1 PostgreSQL

One of the most influential reasons we choose PostgreSQL/relational DBMS mostly since it is what we have learned in class, and we found that using well structured DBMS made it easier for us to traverse data and make queries, compared to a less structured schema/tagged no SQL DBMS. It was more straightforward to start querying since we already knew the syntax and didn't have to learn a new syntax at the point we started

the project. After learning more about tagged / semi structured data layout, we still believed in using PostgreSQL because having relations for each condition, state, values, etc., allow us to quickly change the join conditions or tables that we cross together in our SQL queries and still have a simple template for the majority of our most of our desired queries: cross the relevant tables with the desired join conditions, and project out the relevant columns.

With tagged data like XML or JSON, the data will already have a semi structure, but also a defined nested hierarchy, which can make different kinds of querying more difficult. For example, if we want to find the number of people with a certain disease in a state, depending on how your nodes are structured, you might traverse into a state node, then to a disease node, and then count all the cases. However, if you have diseases as your top node, you will have to first traverse to diseases then to states then count, and if you only have diseases top level nodes, and all cases are inside there with states at attributes, you will need to do a different kind of query.

Most of the health data came from csv files, which was straight forward to convert into relation tables, and PostgreSQL's queries made it straightforward to do dynamic demographic filters, since the user can select the desired demographic combination and values were simply inserted as madlibs into the query.

With datasets this small, the speed of the queries are essentially marginal, so we choose the DBMS that we were most comfortable with.

3.2 Flask

Flask was used since it was recommended to us by Prof. Jun Yang when we were working on our project at HackDuke. It was also helpful that Prof. Yang had Flask templates and example code that we built our website off of. Most of us in our group did not have a lot of experience with web development, so Flask seemed like an intuitive way to start our project since we did have experience in Python.

Flask, being Python framework, also made it easy to communicate with our DMBS, since we could query our PostgreSQL with Psycopg2. This allowed us to keep all our coding in Python, incorporating the features in our project that our database requires. Since everything can be developed in Python, it gives us a lot more control over implementation details.

Some conveniences of Flask are that we can use Python to generate pages and links. Since we want to have a page for each disease/condition, it would be tedious to generate all pages manually. With Python, we can generate all the pages in a loop, and can do it dynamically when we get new conditions, since we can get the names from the database from Psycopg2. Since our project is based on the Model-View-Controller architecture, Flask and Psycopg2 makes it easy to update and query information from database to

backend to frontend. Flask also helps keep presentation and control flow separate, leading to good encapsulation, but also make it clear how these communicate among each other.

3.3 Schema

Our schema is implemented such that most relations are based of number ids for entries on diseases/conditions and locations. With these ids, we made various look up tables for condition names to id, and replaced many of the condition names of incident/mortality data points with the ids.

Since there is standardization ids among all the data sets, querying is now more straight forward since for any condition location combo that a user selects, we just need to look up the id of the condition and location, select for those, and project the count of number of occurrences for that pair.

With this id system, it is also flexible to expand to new conditions, locations, and additional data points. In addition, we broke down our schema to BCNF to reduce any redundancy, so the only datasets that have to grow are the data entries for various conditions, locations, demographics, etc.

4 Approaches and Algorithms

For our project, we did not create any novel algorithms that explicitly manipulate the data, and most of the analytics that we present are all implemented in the PostgreSQL queries. Our approach is to abstract out large data sets and turn them in to simple visuals to help the user make relations between location and conditions.

Some algorithms that were implemented mostly involved data parsing and renaming, in order to make the data fit our schema, and scripts to parse csv files and add them into our database. Using Java and Python, we would read in csv files from different sources (mostly from data.gov or WHO) and rewrite them to fit our schema, which may involve rearranging columns, removing invalid points or data that will break our database, or renaming conditions/locations to the corresponding condition id / location id.

We also wrote scripts that would put the data into our database. Since a lot of the csv files were not formatted to exactly fit our schema, the script can correct them and insert the data points into our schema. This can be expanded dynamically if we were to want to add more data, or even if a user would like to add data and visualize it.

5 System Evaluation

The goal of our project is to make it easier to understand large health and location data sets. We wanted to make an intuitive application to for people to explore data that is commonly available from data.gov, WHO, and on the internet. However, due to the sheer amount of data, it can quickly become difficult to process or easily draw conclusions with. With good visuals and clear interface, users can easily explore the data and draw their own conclusions.

Compared to other health data visualizers, we are not the only application out there, and honestly there are some better visualizers on similar data. However, we hope to leverage the flexibility of our framework and develop it in the future. Hopefully, when new data comes out from data.gov, users can upload their own data or the csv file and as long as the csv formats are similar or rearranged, the user can quickly get their data visualized. Although we currently do not have this feature, the Python framework and scripts can easily make this possible, which is something competing visualizers do not offer.

6 Issues / Future

Some issues with our implementation is that we are limited by the number of graphs we can create per day due to our map API. Using plot.ly, we only get 25 free graphs per day, so if you graph too many times, the plots won't be able to show up anymore. This can be solved by buying the pro version, but that is also $100.

Another issue is that we did not process any world data or economic data, such as GDP and population. The framework is set up to accept this data, and can easily be implemented, but we ran out of time, since it takes a very long time to upload all our data files into PostgreSQL. If we had time, this could be implemented on the front end and integrated with plot.ly, due to the flexibility of our parsing script, which would provide more insight on the health data and how it correlates to other factors (like income) that other competitors don't have.

In the future, there is lot of potential development for this project. Since our data and demographic filters are updated dynamically from the schema of the data itself, we can potentially have users upload their own data into a database and allow them to analyze their own data. We will first start by putting more work into the integrating GDP data and world health data into our database, and making sure that our backend can scale to new data.

Afterwards we can let users upload data to visualize, and then also allow users to log data to contribute to our main database to overall increase the amount of data we have available.