Database Project Final Report: SeeClickFix

by Emon Datta, William Dower, and Kyle VanderWerf

I. Background & Motivation

SeeClickFix is a “communications platform for citizens to report non-emergency issues, and governments to track, manage, and reply — ultimately making communities better through transparency, collaboration, and cooperation.” SeeClickFix has been around for years and has helped cities more quickly and transparently resolve infrastructural issues, and has information for millions of issues in its repositories. However, the organization has not had a chance to analyze its rich data set despite its potential to reveal illuminating information on infrastructural quality in different sectors, efficacy and timeliness of different city departments (by issue resolvement time), and the like, all of which could be used to help improve and prioritize city infrastructure issues, as well as hold government bodies accountable.

Through Yale’s Community Based Learning program, we contacted SeeClickFix’s New Haven arm to help begin this process by creating a system to analyze historical issue data, beginning in New Haven, that could be easily expanded/generalized to other cities. Our system scrapes the online API to retrieve and store historical issue data, and allows users to query these tables for resolve-time information on issues by type and location category. This gives end users — specifically, city officials and SeeClickFix employees — insight into the distribution of issues by type in different areas as well as information on the time taken to resolve these issues.

II. Working Split

Emon

I primarily worked on creating and designing the databases themselves from the SeeClickFix data. This was done in two phases.

A. In the first, I cleaned and parsed an exported CSV dump of New Haven's historical issue data that an employee at SeeClickFix sent me. I then imported this data into a table I created on PostgreSQL Server. Since there was a functional dependency between request_type_id → request_type_name, I decomposed that data into a separate table, with request_type_id being a foreign key in the issues table into the request_type table. (See *Appendix*.)

B. Since the CSV data dump was quite unclean and did not encompass the full set of available data, and also to make this a tool that will be useful for the future when the available historical data changes, I created a second set of tables from data that I scraped from SeeClickFix's online API. This data included more information on issues, and furthermore, expanded information on users (e.g., the issue reporter and assignee), comments, and questions. In my scraping program, I added this information on users, comments, and questions into 3 separate tables for a total of 5 tables (in addition to the main issues and request_types tables); this was done to reduce redundancy in the tables, since one issue might have several questions/comments, and one user may be a reporter/assignee for several issues, and it would be redundant to have separate tuples in a one large, all-encompassing table for all this information for each issue. (See *Appendix*.)

Will

I wrote the front-end GUI through which the user can access the database. To do so, I used the NetBeans IDE to assemble a GUI using Java's swing library. This involved learning the basics of how a GUI works, such as working with frames, windows, dialog boxes,

buttons, tables, and so on, and how to knit all the components together into a usable and aesthetically magnificent executable.

Kyle

I had two responsibilities. First, I wrote code that used Google's Reverse Geocoding API to get addresses/neighborhoods/localities from the longitude and latitude coordinates provided in the CSV file (which code can be found in the folder GeocodingIssues on GitHub) and add them to the database. Second, after Will wrote each part of the GUI, I imported JDBC and wrote the SQL query functionality into it. Essentially, my work connected Will's work (the GUI) to Emon's work (the database itself).

III. System Functionality

The system can store and display metadata about entries in New Haven's SeeClickFix website. The data is drawn from a .csv file given to us by SeeClickFix, stored in our database, and accessed by our GUI. The GUI prompts the user to specify an area for which it should display statistics about the types of problems people report there. The user can specify neighborhoods in the city or specific addresses (featuring an address-search bar that auto-completes the streets of New Haven), and a timespan. The data can be displayed sorted by number of separate instances, total instance time, or average instance time. The system can determine what instances happen within certain areas because each incident reported to SeeClickFix has a latitude and longitude, which we then use with Reverse Geocoding to find the address nearest to the incident. The system can thereby give a high-level view of what kinds of problems occur in different areas of the city during different blocks of time.

For example, if you wanted to know the number of problems reported to SeeClickFix on Prospect Street for the year 2010, you could launch the program, select search-by-address, look for Prospect Street in the

search bar, choose the start and end dates, and click ‘Run!’ to see a table containing each category’s data.

IV. System Limitations

A. At the moment, difficulties with Google’s Geocoding tool makes it so that we do not have data more current than 2012 in the database. The Reverse Geocoding API only allows for a certain number of queries to be run per day, and with our many thousands of issues, it took days/weeks to get this information even for the initial set of issues from the CSV. Due to this, we were only able to use this smaller set of data and the original tables from the CSV, since it would not be possible to Reverse Geocode all the data from the API-scraped data in time for the project deadline. However, the same logic can be used for information in the newer, API-scraped data tables, making migration very simple.

B. Sometimes there isn’t much helpful data to display. For example, one of the most prominent categories of problems reported to the website is simply ‘Other.’ We can only display data that SeeClickFix has actually collected.

C. We wanted to add in the ability to search for problems by ZIP code (there are several different codes within New Haven), which would work by the same principle that lookup by address and neighborhood work, but we didn’t quite get around to it.

V. Challenges

A. The original CSV file of issue data had many issues and was difficult to parse to be consistent without losing information. Later, when scraping the issue data from the online API, it was somewhat difficult to decompose the data on the spot into separate tables (with the main issue tables and also request_type, users, comments, and questions tables), particularly because I had to deal with potentially NULL values

as well as the fact that some users and request_type data may already have been seen in a previous issue, in which case we did not want to try to re-add it to the table when unnecessary.

B. Writing the SQL query to aggregate the data was difficult. Since we wanted to aggregate the same data in multiple different ways, and then combine all those results into the same table, the form of the SQL expression got fairly complicated.

C. The NetBeans IDE in which the GUI was created can be extremely difficult to work with. For example, we originally wanted to enable the table that displays the data to be able to sort itself correctly when the user clicks a column header -- but this would require editing the IDE's auto-generated code, which it didn't like. We wound up just putting in drop-down menus for the user to specify how they wanted the table sorted instead. I used NetBeans due to its interactive GUI-building functionality, which could auto-generate a good deal of the trivial but still complicated Java code for GUI components, but it seems that feature was a double-edged sword.

VI. **Contributions**

The current SeeClickFix system only allows users to find and list data about individual issues. It doesn't have the ability to make queries about aggregate issue data, or trends in the issues. Our application allows users (which are expected to be government employees or SeeClickFix staff) to determine which types of issues are/were most prevalent in a specified time period. The app measures this in three ways: number of issues of a given type, total time to resolution of all the issues by type, and average time to resolution. Users can filter these queries by neighborhood or by street, and can sort them by any one of these metrics. With our app, local government can see trends in infrastructural issues over time, which can assist in decisions such as future budget allocation.

VII. **Selling Points**

A. Our system sorts data as specified in each query in the database itself. This is in general more efficient than sorting the data after it has been extracted, and hence our system's queries return results more quickly. Furthermore, the SQL query that aggregates the data outputs precisely the table seen in the GUI. We did not have to do any query processing in Java after the fact, thus improving efficiency further.

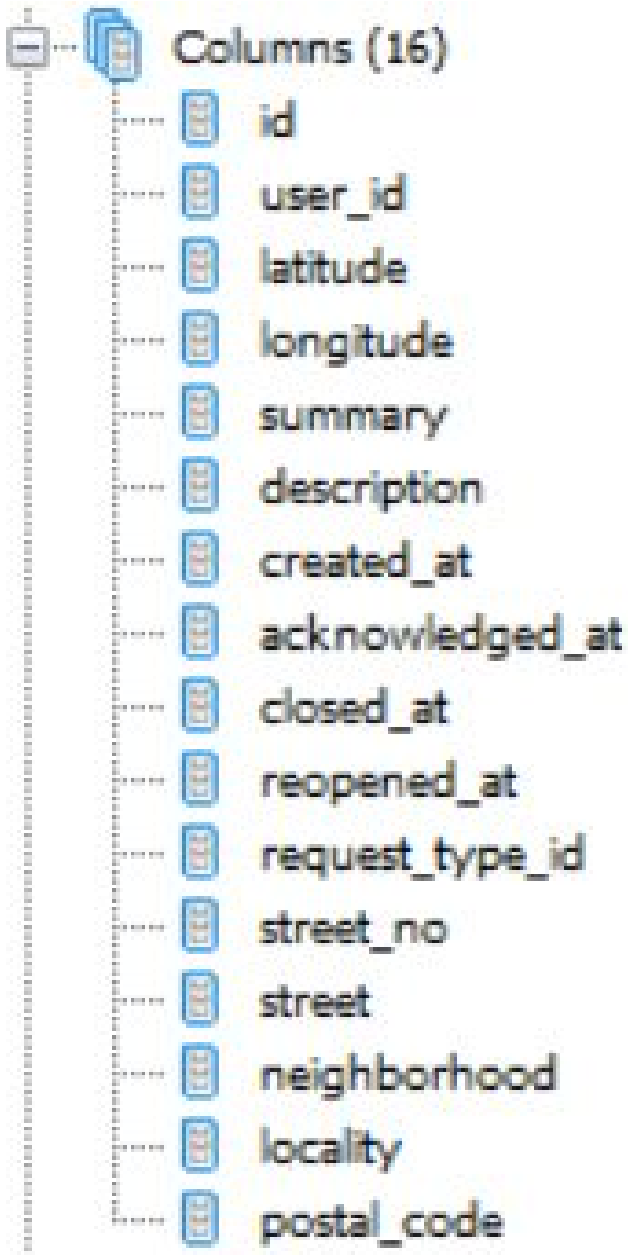
B. Our database uses null values to its advantage — for each attribute that has null values, the existence of a null value means something specific, and hence ambiguity is avoided. For example, a null value in the attribute "closed_at" means that the issue has not yet been resolved.

C. Our database design balances the competing concerns of good design and good query performance. It is in Boyce-Codd Normal Form with respect to the attributes provided in the CSV file and those in the API, since we decomposed information such as request_type, users, comments, and questions. However, we intentionally add location-related attributes to the main issues table — such as street address, locality, neighborhood, and zip code — which are functionally dependent on the longitude and latitude. This made the table no longer in BCNF. However, since almost every issue has a unique longitude and latitude and our queries needed to retrieve this location information for each issue, decomposing it would have made queries much less efficient without saving much space.

Appendix

Tables from CSV scrape of data

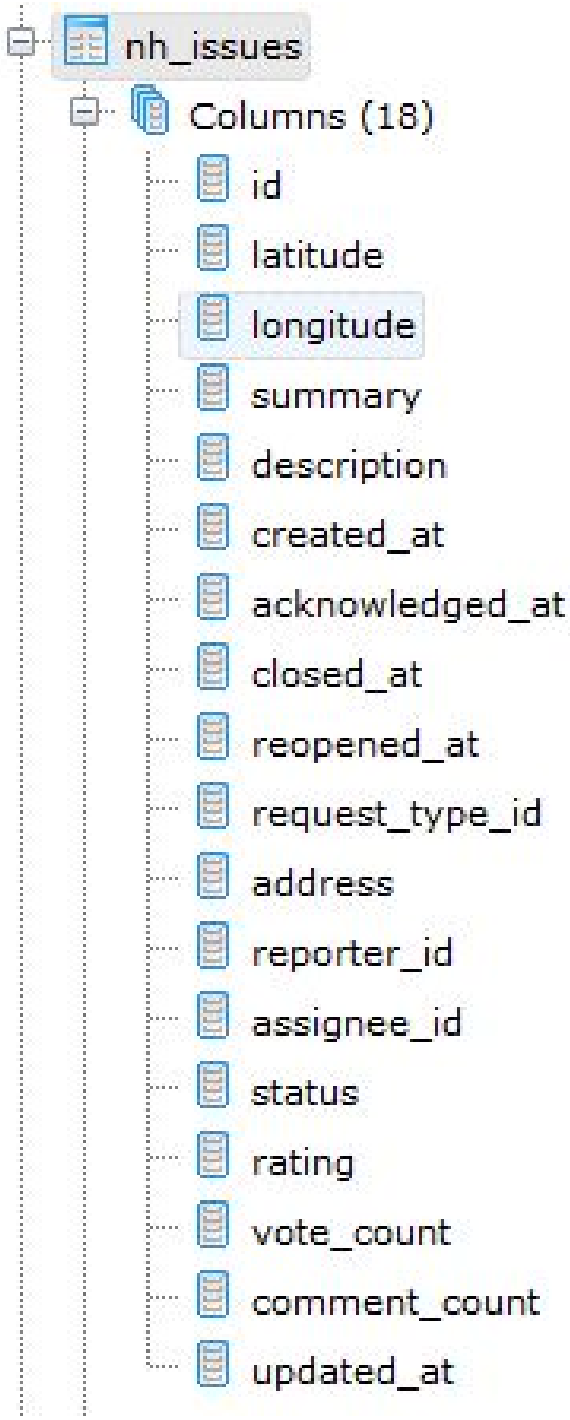
Issues:



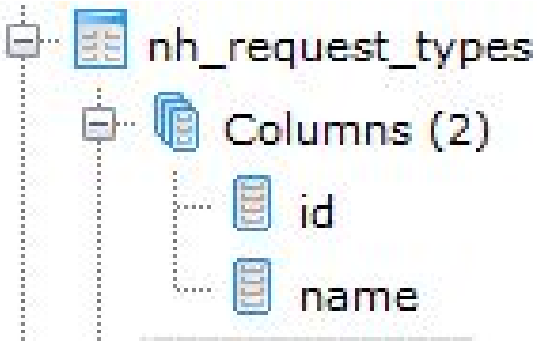
Request types:

Tables from API scrape of data

Issues: *(does not include Reverse Geocoding info due to API limits)*

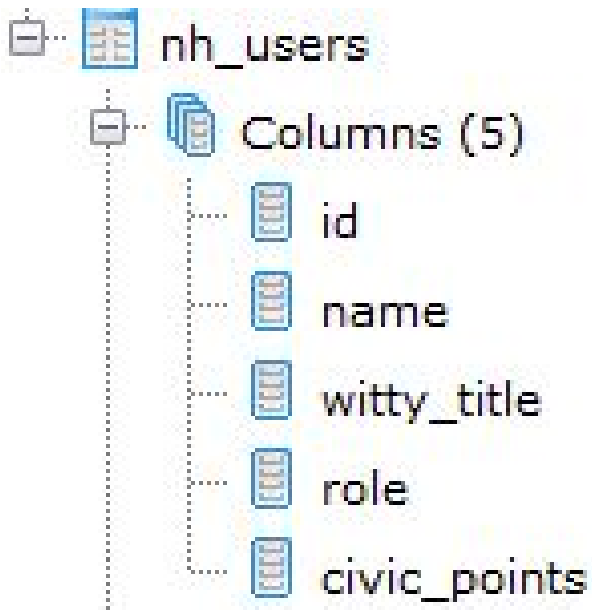


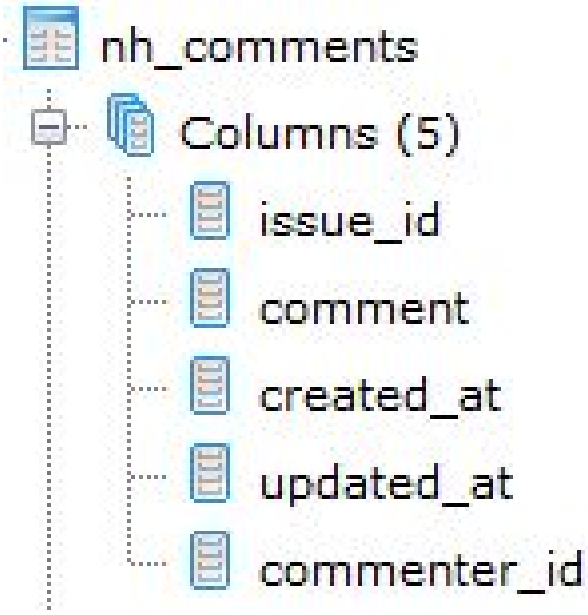
Request types:



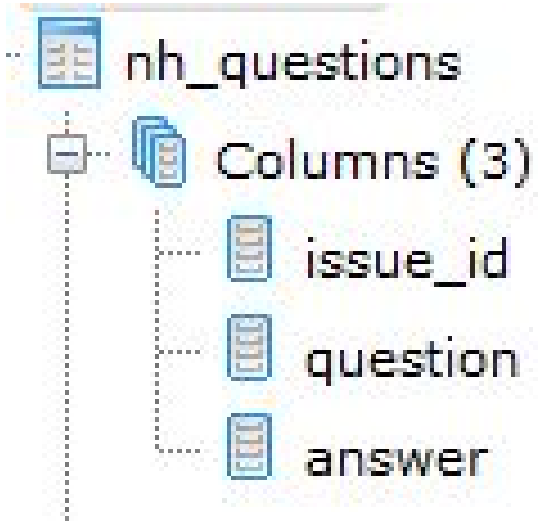
User Info:

Comments:


nh_users
Columns (5)
id
name
witty_title
role
civic_points


nh_comments
Columns (5)
issue_id
comment
created_at
updated_at
commenter_id

Questions:


nh_questions
Columns (3)
issue_id
question
answer