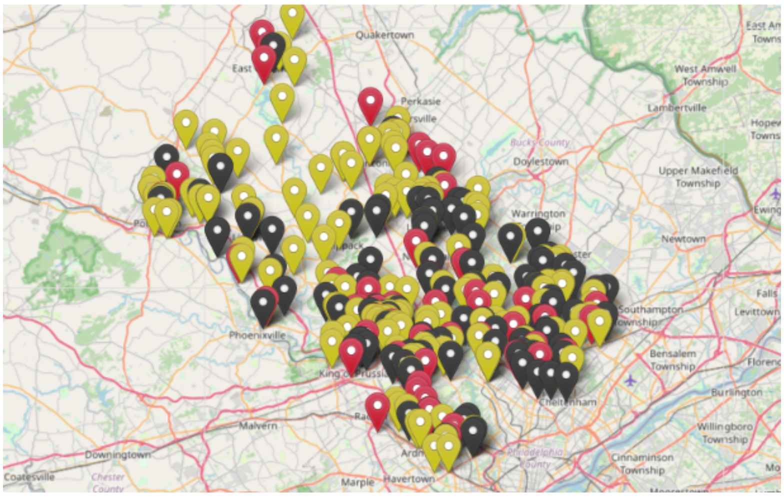
Development of an Emergency Response Dashboard and Associated Infrastructure - Final Report (Group 2)



Antoun, Fadi[1], Fishman, David[2], Haughton, Laurie[3], Jung, Myunghee[4], and McLennan, Dan[5]

Emails: [1]fadi.antoun.95@gmail.com, [2]davjfish@gmail.com, [3]fadi.antoun.95@gmail.com, [4]mundlejung@gmail.com, [5]DanJMcLennan@gmail.com

December 2025

Contents

1 Objectives

1.1 Rationale behind the analysis

Effective emergency response is paramount to public safety, yet emergency services are often collected and monitored by disparate systems. Understanding trends and patterns across different jurisdictions and agencies can help decision-makers and public administrators make better decision about city planning, coordination, resource allocation and response optimization. The challenge that this project is addressing is converting a high volume of raw, unstructured call data, such as that depicted in this dataset, into meaningful, operational intelligence.

In this project, we will discuss different approaches for the aggregation, and storage of these data and will outline some approaches for ways these data can be disseminated with stakeholders and administrators.

1.2 Structured Data in a SQL Database

Storing the emergency response data in a relational database offers several critical advantages for building an effective and reliable dashboard. The primary rationale is centered on data integrity, efficient analysis, and compatibility with standard business intelligence tools, specifically, web and mobile applications that stakeholders will want to use to view and interact with the data.

There are several key reasons why a SQL database is an appropriate choice for this project. First, most emergency call data will have a predicable and well-structured format, e.g., incident type, timestamp, location (zip code, township), latitude and longitude. Data from which these basic attributes cannot be extracted are going to be less useful to stakeholders. Next, data organized into a relational database can be stored much more effectively than a two-dimensional dataframe (assuming a well-designed schema and sufficient normalization). A relational database management system (RDBMS) will also give administrators the ability to implement constraints in order to maximize data integrity. This includes adding lookup tables (i.e, foreign key constraints) and enforcing uniqueness, either within a column (e.g., should only have a single entry for a zipcode in a ZipCode table) or between columns (e.g., combinations of ćity,́ śtateánd ćountryśhould be unique in an AdministrativeArea table).

Finally, a SQL database is highly compatible with REST APIs, and the two technologies are commonly used together in modern application architecture. A REST API would allow for the implementation of reactive tools for viewing and interacting with the data across a variety of platforms, such as browsers, alert-systems or mobile applications.

2 Methods

2.1 Designing a Relational Database Schema

The principle dataset used for the project was sourced from Montgomery County, Pennsylvania and is accessible here [5]. This structured, tabular dataset is provided as a flat CSV file and consists of over 600,000 records of emergency calls from December 2015 to April 2020. This dataset will be used to create an outline of the database schema that will be used to store the dashboard data. In order to achieve this, We will explore the dataset by importing it into a dataframe and inspect the data types being stored in each column. Subsequently, we will decide on the best way to separate the data into discrete tables. In addition to foreign key and uniqueness constraints, we will ensure the strategic implementation of database indexes in order to optimize database performance. Finally, for the sake of demonstration, we decided to utilize a simple sqlite database [3].

2.2 Dashboard REST API and Application

In order to provide stakeholders with a concrete vision of our dashboard, our group will create a mock-up of this application. The framework selected to do this was Django [6]—a Python object-relational mapping (ORM) web framework. Django also has a wonderful built-in API for working with the most common types of relation database management systems. This is great because the tables can be drafted as database-agnostic models and migrated to different systems in different environments.

Additionally, Django also has some great external packages for developing a REST API. In particular, the Django REST Framework [7] will be used to define endpoints that will be used for acquiring data from the application. This framework provides a quick and effective way of serializing the Django models (i.e., database tables) into a JSON response as well as writing incoming serialized data to the database.

On the frontend of the application, we will use a combination of HTML, the Django templating language and JavaScript to present end-users with information from the dashboard. In particular, the JavaScript library Vue.js [4] will be important in providing users with a reactive, modern experience. Aside from Vue.js, we will also utilize Leaflet [2] and Charts.js [1] for mapping and reactive graphing, respectively. The mock application will be hosted in on a free-tier cloud platform in order to illicit feedback from stakeholders.

2.3 Data Ingestion

One of the challenges we will face in this endeavor is how to load information coming in from disparate data sources. We will address this by creating a Python class for handling, parsing and ingesting serialized emergency call data. A separate Parser class will have to be developed for each schema of incoming data.

2.4 Code Repository

All the code for this report and for the application will be stored in a publicly accessible git repository on GitHub.

3 Results

3.1 Database Design and Implementation

After an initial inspection of the data using exploratory tools from Pandas [8], a SQL schema of five tables was devised and drafted. The five resulting tables and their descriptions are as follows:

- Category - categorical descriptions of the types of calls received (e.g., car accident)
- Township - township name and state (e.g., Kings Township, PA)
- ResponseUnit - complete list of units responding to emergency calls (e.g., Station 123, EMS)
- ResponseType - the response unit type (e.g., EMS, Traffic, or Fire)
- EmergencyCall - this is the primary data table and used to store information about emergency calls received. It has several links to the above tables.

Figure 1 presents a visual portrayal of the five above table and the relationships between them. In addition to the foreign key constraints between tables, indexes were added to each table to improve performance. Unique constraints were placed on fields in tables were duplication of data entry was not wanted. For example, we only wanted there to be a single entry for b̈ack painïn the Category table and only a single entry for ËMSïn the ResponseType table. The Township and ResponseUnit tables both had unique together constraints across two columns. In the case of the former, only a single entry for a combination of township name and state was desired and for the latter, only a single combination of response unit and station name was desired. Finally, the complete SQL definitions of the above tables and indices can be found in Appendix 1.

The file size of the original CSV downloaded from Kaggle was approximately 123 MB. Following the creation of the database and subsequent ingestion of the data, the file size was reduced to approximately 75 MB, including all the indices. This confirms that our schema has indeed achieved greater storage efficiency than the CSV flat file.

3.2 Emergency Response Dashboard Application

The mockup of the Emergency Response Application can be viewed on the internet here. The application is being hosted on the PythonAnywhere cloud with a free-tier account. While this is great for demonstration purposes, the free-tiered containers are low performant and thus very slow. The application has a landing page, a map, a data summary page and an administration page reserved for manually importing data. All of these pages are very basic and have the sole purpose of demonstrating a proof of concept.

A screenshot of the map, which utilizes Leaflet, can be seen in Figure 2. Since there is a large number of calls in the database (¿600,000), it would take too long to load this in a single call. This will become even more pronounced overtime as additional data is accumulated. Therefore, using a paginated API endpoint is of paramount importance—allowing users to load data incrementally. In our application, the users can decide on how many records to load at a time (up to a maximum page size of 10,000, see pagination class here). Additionally, users can hone in on specific data by using filters which are then passed to the API endpoint as query parameters. For example, a GET request to this endpoint would return a paginated JSON response of calls received between December 10, 2015 and December 10, 2016, and responded to by EMS Station 313A.

In a similar vein, the dashboard models how a stakeholder can explore data summaries and statistics in real time. To achieve this, we used Charts.js in conjunction with a custom-paired API endpoint. A screenshot of this page is presented in Figure 3. The main difference between this view and the map is that the API is calling aggregation functions instead of returning individual call records. By doing so, user are able to aggregate vast quantities of data and leverage the table indices as well as the powerful SQL engine. As noted above, these endpoint are going to be custom-paired with specific summaries or analysis. In the example shown here, this endpoint returns a response that is highly tailored to the Charts.js bar chart being used on this page. The response is generated by first applying any filters to the queryset and then grouping the results by category and call count. Next, the data is sorted into two lists, one corresponding to category names and the other to the respective call counts. These step can be viewed in our application's code here.

3.3 Importing CSV Data

One of the challenges we will face in this endeavor is how to load information coming in from disparate data sources. We will address this by creating a

Python class for handling, parsing and ingesting serialized emergency call data. A separate Parser class will have to be developed for each type of incoming data that is anticipated. These parsers will be written as Python classes and stored in a file called parsers.py

The parser class called `PA911CSVParser` is what we developed to parse and ingest data that is structured like that found in the Pennsylvania dataset. The class has one obligatory initial argument called `file`, which is a BytesIO object held in memory. when the `parse()` method is called, the import sequence is then triggered. First, the data is read into a Pandas DataFrame and cleaned using the following steps:

1. The response unit type (i.e., EMS, Traffic or Fire) is extracted from the title and stored in a separate column. This is strait-forward because the response unit type is always prefixed in the record's title followed by a colon.

2. The incident category is extracted from the title and stored in a separate column. This step is also pretty simple since the category of incident always proceeds the type.

3. If possible, find a station name in the record's description. This is a more complicated step that involves the use of RegEx to detect the different ways in which the station name is provided. An additional challenge for this step is that sometimes the description contained an address with a street name called S̈tation Rd.̈

Next, we iterate through a unique list of all the townships in the data. Since we are not expecting there to be too many townships, it is safe to proceed item by item in the unique array. For each item, we strip trailing whitespace from the strings and convert them to uppercase in order to minimize the potential for duplicate entries. Then, we initialize a `Township` object and append it to a python list. At the end of that loop, we execute a bulk create query to add all the townships in a single transaction. Since there is a unique constraint on a Township's name and state, duplicate entries will be rejected. However, if we use the `INSERT or IGNORE` statement to the bulk create, the transaction will still be accepted. Then, we create a lookup dictionary of township name to database primary key and map this dictionary to the original dataframe—resulting in each row having a Township ID. The same approach was repeated for ResponseType, Category and ResponseUnit.

Finally, we are ready to create the individual entries for each emergency call. Since we are now dealing with a much higher quantity of records, we cannot simply iterate through each row of the database. Instead, we will leverage parallel processing by using the Pandas DataFrame apply method. We start by defining a protected-member method called `_make_call_from_row` that assembles an `EmergencyCall` object from a row in the dataframe and appending it to a list. Then, just as above we will execute a SQL bulk create query to write all the objects to the database. Running the parser locally on a standard desktop computer takes about 12 minutes to ingest over 600K records.

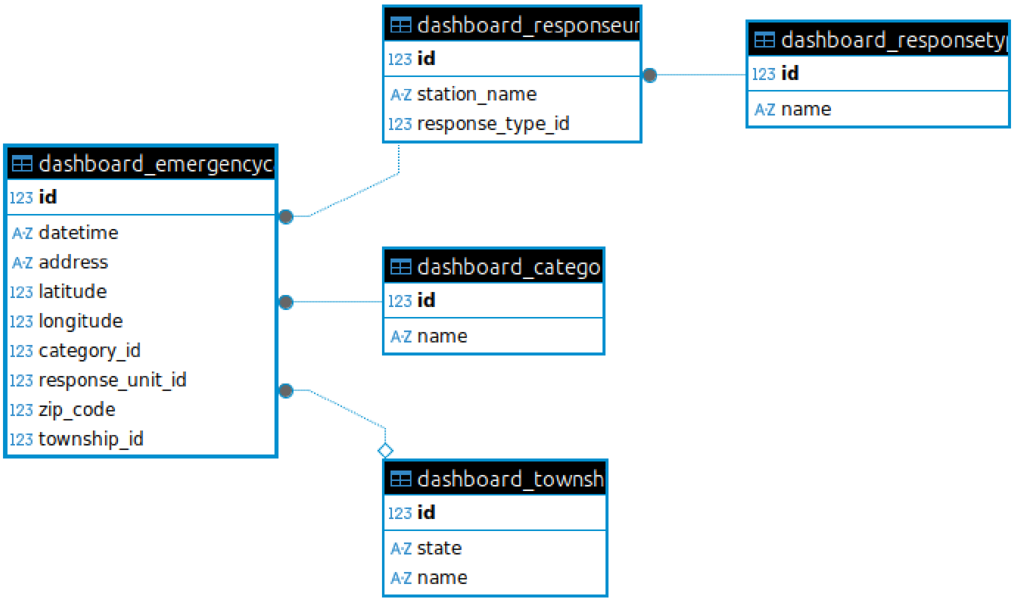


Figure 1: A visual depiction of the five tables in our SQL database design.

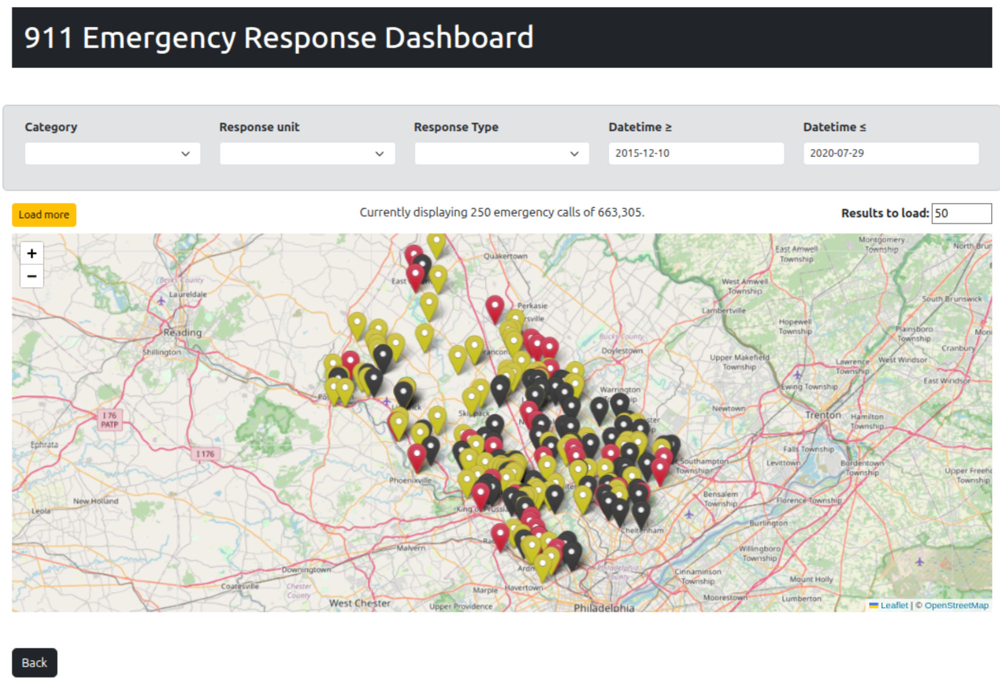


Figure 2: The Emergency Dashboard application's map.

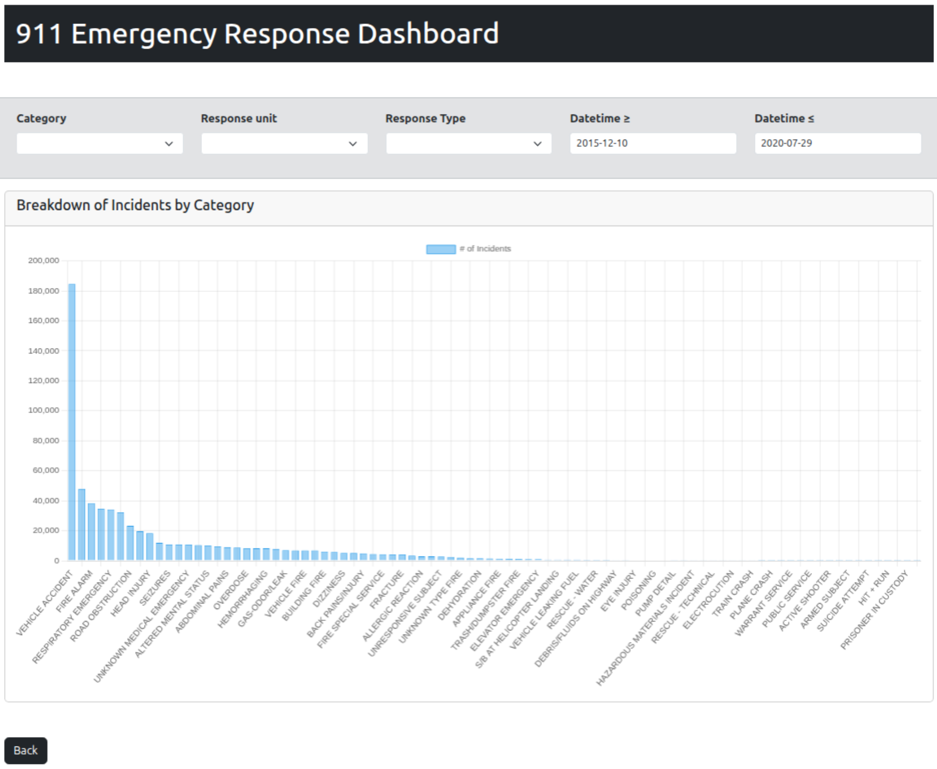


Figure 3: The Emergency Dashboard application's charts page.

4 Conclusions

This project helped develop an appreciation of the complexity of integrating

4.1 Future Development

Scratch ideas:

- it would be great to handle the ingestion the csv data using async, maybe something like celery where it can happen in the background - database routers - metadata fields such as created$_{a}tandcreated_{b}y-sqlitenotacceptabletouseinaproductioner$ $--thisissuitableofaproofofconcepthoweverthiswouldnotbeanacceptabletooltouseinaproduction$

a discussion about the parser classes and how they can be called in the context of uploading a CSV or a web scrapper that is interacting with an API for an outside system

it would probably make sense to put a lot of investment in the rest api to maximize interoperability between systems

one drawback is what happens if the data schema from a source changes.

5 Appendix 1: SQL Schema of Emergency Response Database

```
CREATE TABLE IF NOT EXISTS "EmergencyCall" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "datetime" datetime NOT NULL,
    "address" text NULL,
    "latitude" real NOT NULL,
    "longitude" real NOT NULL,
    "category_id" bigint NOT NULL
        REFERENCES "Category" ("id"),
    "response_unit_id" bigint NOT NULL
        REFERENCES "ResponseUnit" ("id"),
    "zip_code" smallint NULL, "township_id" bigint NULL
        REFERENCES "Township" ("id")
);

CREATE INDEX "emergencycall_category_id_28afcc20"
    ON "EmergencyCall" ("category_id");

CREATE INDEX "emergencycall_response_unit_id_f0a9566e"
    ON "EmergencyCall" ("response_unit_id");

CREATE INDEX "emergencycall_township_id_c7779d84"
    ON "EmergencyCall" ("township_id");

CREATE TABLE IF NOT EXISTS "Township" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "state" varchar(10) NOT NULL,
    "name" varchar(255) NOT NULL
);

CREATE UNIQUE INDEX "township_state_name_a30a5e69_uniq"
    ON "Township" ("state", "name");
```

```
CREATE TABLE IF NOT EXISTS "ResponseUnit" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "station_name" varchar(255) NOT NULL,
    "response_type_id" bigint NOT NULL
        REFERENCES "ResponseType" ("id")
);

CREATE UNIQUE INDEX "responseunit_response_type_id_station_name_25efee89_uniq"
    ON "ResponseUnit" ("response_type_id", "station_name");

CREATE INDEX "responseunit_response_type_id_21e2bd85"
    ON "ResponseUnit" ("response_type_id");

CREATE TABLE IF NOT EXISTS "Category" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "name" varchar(255) NOT NULL UNIQUE
);

CREATE TABLE IF NOT EXISTS "ResponseType" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "name" varchar(255) NOT NULL UNIQUE
);
```

References

[1] Chart.js — Simple yet flexible JavaScript charting library for the modern web. .

[2] Leaflet—A JavaScript Library for Interactive Maps.

[3] SQLite.

[4] Vue.JS - The Progressive JavaScript Framework.

[5] Mike Chirico. Emergency - 911 calls, 2020.

[6] Django Software Foundation. Django.

[7] Encode (main developer Tom Christie) and contributors. Django REST framework.

[8] NumFOCUS. pandas. `https://pandas.pydata.org/pandas-docs/stable/index.html`, 2023. Accessed on 2023-12-11.