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.

1.3 Unstructured Data in a NoSQL Database

NEED SOME HELP HERE!

2 Data Preparation

2.1 Designing a Relational Database Schema

The principle dataset used for the project was sourced from Montgomery County, Pennsylvania and is accessible here [4]. 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 Pandas dataframe [7] and inspect the types of data 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.

2.2 Dashboard REST API and Application

In order to better demonstrate and a feasible use-case for this dashboard, our group will create a mock-up of an application. The framework selected for this project was Django [5]—a Python object-relational mapping (ORM) framework. Django also has a wonderful API to many commonly used SQL databases which we planned to leverage. Specifically, the models will be first drafted as classes and subsequently migrated to

For demonstration purposes, our group decided to use a simple sqlite database however the django framework itself is database agnostic. Django also has some great packages for developing a REST API, specifically, we leveraged classes from the Django REST Framework [6] package.

On the frontend of the application, this tool will use a combination of django templates and JavaScript (JS) libraries to present end-users with information from the dashboard. The JS libraries in conjunction with the REST API is a great way to provide users with a reactive experience.

The Django web-application will also be used to define a Parser class that will be used to ingest the CSV data. Th

The mock application will be hosted in on a free-tier cloud platform in order to illicit feedback from stakeholders.

Finally, all the code for this report and for the application is store in this publicly accessible git repository on GitHub: `hhttps://github.com/davjfish/WatspeedBigDataGroupProject/`.

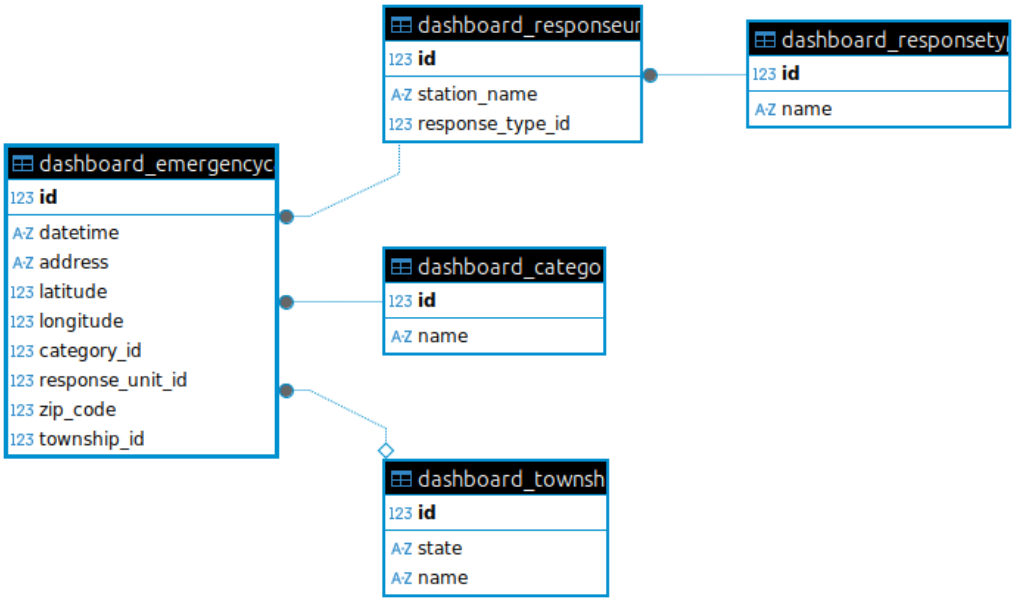

Figure 1: A visual depiction of the five tables in our SQL database design.

3 Results

3.1 Relational Database Schema

The principle dataset used for the project was sourced from Montgomery County, Pennsylvania and is accessible here [4]. 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.

After an initial inspection of the data using exploratory tools from Pandas [7], 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.

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, Figure 1 presents a visual portrayal of the five above table and the relationships between them.

gains in efficiency

Specifically, in addition to HTML Django templates, the reactive frontend of the application will be developed using Vue.js [3], Leaflet [2] and Charts.js [1].

On the backend of the application a parser class was defined in order to handle the incoming CSV and its digestion into the database. - leveraging bulk creates

The Django web-application will also be used to define a Parser class that will be used to ingest the CSV data. Th

3.2 Methodology

4 Conclusions

4.1 Future Development

- 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*

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] Vue.JS - The Progressive JavaScript Framework.

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

[5] Django Software Foundation. Django.

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

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