Chicago Traffic Crashes

Jorge Fioranelli, William Hendrickson, Brendan Veit, Justin Waldman
University of Pennsylvania

August 7, 2025

1 Introduction

Our aim was to find a topic that had a vast amount of data to choose from and would allow us to develop a tool to help the public. We immediately were drawn to traffic incidents as we were able to find a rich dataset to work with. The city of Chicago is one of the largest cities in the US and they efficiently track their traffic incidents with great detail. Through the Chicago Data Portal, they provide many different datasets including "Traffic Crashes - Crashes", "Traffic Crashes - Vehicles" and "Traffic Crashes - People". These are the datasets we chose for our project.

Nearly every person in the US commutes in some form. Many people have heard that driving can be dangerous, but they often lack data to show how it can be dangerous. Our visualization tool provides clear information to a user and helps them understand how to be safer on the road. For example, one of the top reasons why crashes occur is that a driver fails to yield the right of way. With this information, driving schools and parents can spend more time educating younger drivers about when they are required to yield to traffic.

Users initially land on our homepage, which visualizes multiple forms of traffic data. We designed each page to provide unique functionality which serves the end goal of helping educate our users. From the homepage, the user can navigate to a crash search page where they can filter the crash data to refine their search. Once they find a particular crash incident of interest, they can click on the crash id field to be taken to a page detailing all information about the crash. Our final page is a user quiz page. This page is designed to challenge a user to search for the correct information within the site.

2 Architecture

Each member of our group had little to no prior experience with web development. Due to this, we decided to use Homework 3 as a starting template for our website. As in the homework, we used node js and react to implement our site. We used AWS to host our Postgres database. We stuck closely to the template format by maintaining our routes and server files which contained nearly all of our backend code. Our frontend consisted of multiple pages and utilized a few custom components such as CrashCard and VehicleInfo.
Below are brief descriptions of each of the distinct pages of our application:

Chicago Crash Safety Dashboard (Homepage)
This page serves as the initial landing page for a user. It presents many different graphs which help visualize the data to the user. The user can immediately interact with the graphs by selecting different options from one of the drop-down menus. The selection triggers a back-end query which is then visualized in the relevant graphs.

Search Crashes
This page allows a user to search for a specific crash by entering the crash id. Alternatively, a user can search for a specific crash or set of crashes via the slider selection and/or drop-down box selection. After pressing search, a query is triggered on the backend to pull up all information per the user input. Some relevant information about the crash is then output in a table for the user to view. A user can click on the crash id to be taken to the Crash Information Page.

Crash Information
This page displays information associated with a crash record. It provides a more detailed view of an incident including which vehicles were involved, the weather conditions when the crash occurred, and a summary of injury information.

Quiz
This page serves to encourage the user to explore the website in an effort to obtain the answers to all five questions. The user is presented with a question and is asked to select the correct answer for each question. We keep track of the number of correct answers, then display the result to the user as they answer the last question. The user is able to retake the quiz at any time.

3 Data

The project utilized traffic crash data sourced from the Chicago Data Portal. This data encompasses incidents that took place on streets in Chicago and fall under the purview of the Chicago Police Department (CPD). The dataset is organized into three major files: "Traffic_Crashes_-_Crashes.csv", "Traffic_Crashes_-_Vehicles.csv", and "Traffic_Crashes_-_People.csv". To access the most up-to-date version of these files, you may visit `https://data.cityofchicago.org/Transportation/Traffic-Crashes-Crashes/85ca-t3if/about_data`.

The initial file, and the most important, contains a compilation of the crashes. The second file encompasses data concerning the vehicles implicated in these crashes, while the final file provides details about the individuals involved. Each of these files includes an attribute named "CRASH_RECORD_ID," which links them together. This report will outline the key aspects of the data, and you can access the full Jupyter notebook here: `https://github.com/jorgefio/cis5500-group4/blob/main/data_cleaning_and_db_population_v5.ipynb`

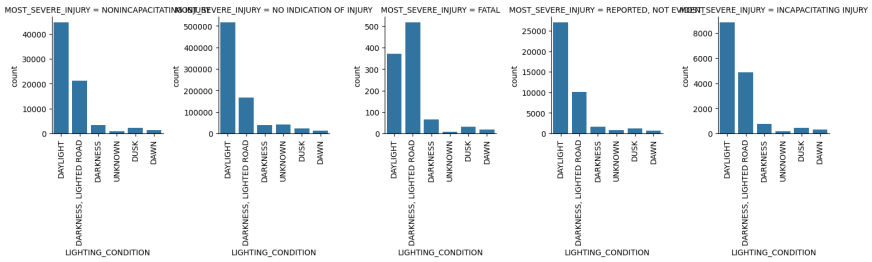


Figure 1: Crashes EDA

3.1 Crashes Dataset

The raw Crashes file consists of approximately 1 million rows and 48 columns, with many columns having numerous missing values (see our Appendix for more information).

Throughout the exploratory data analysis (EDA), multiple interesting relationships between the attributes were identified. For instance, Figure 1 illustrates the correlation between various "LIGHTING_CONDITION" values and "MOST_SEVERE_INJURY".

3.2 Vehicles Dataset

This dataset refers to the vehicles involved in the crashes. It comprises approximately 2 million rows and 71 columns. Similar to the earlier dataset, numerous columns include a significant amount of missing data (see our Appendix for more details).

During our analysis, the distribution of vehicle years caught our attention; notably, there's a distinct peak for vehicles that are approximately 10 years old, corresponding to the years 2014 to 2017. Refer to Figure 2.a for further information.

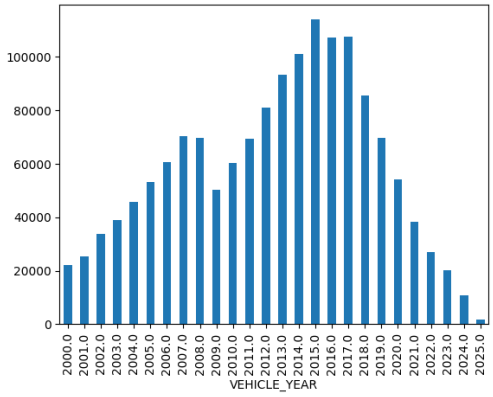
3.3 People Dataset

The final dataset in our collection provides information about individuals involved in the crashes. It consists of more than 2 million rows and includes 29 columns. Similar to the previous datasets, numerous columns in this dataset have sparse data (see Appendix for details).

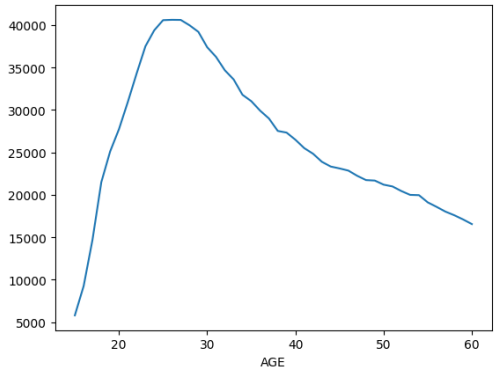
Upon examining the people dataset, we observed that the majority of accidents involve individuals in their mid-twenties (refer to Figure 2.b).

3.4 Data Cleaning

In our data cleaning process, we initially eliminated columns with over 30% missing values and subsequently removed NAs. Reversing this order would have significantly reduced the number of rows available. We opted to rename a mislabeled column "VEHICLE_ID" in the vehicles_df because the true identifier is the "CRASH_UNIT_ID" column. Although the dataset sheet lacked a description for this column, it was notable that the people_df dataset similarly features a "VEHICLE_ID" column that corresponds to "CRASH_UNIT_ID" (source: `https://data.cityofchicago.org/Transportation/Traffic-Crashes-People/`



(a) Vehicles EDA



(b) People EDA

Figure 2: Comparing with and without caching

`u6pd-qa9d/about_data`). To prevent confusion, we eliminated VEHICLE_ID and renamed CRASH_UNIT_ID to VEHICLE_ID.

We also discarded orphan entries. Since NAs were removed, we had to verify that the datasets no longer contained invalid references. We also adjusted the column types in accordance with the datasheet.

As a result, ended up with these new numbers:

- crashes_df: 1M rows -> 900K rows, 48 columns -> 37 columns
- vehicles_df: 2M rows -> 1.5M rows, 71 columns -> 16 columns
- people_df: 2M rows -> 700K rows, 29 columns -> 17 columns

4 Database

In AWS, we set up a PostgreSQL database and imported each of the three data files into corresponding tables: crash, person, and vehicle. We also created tables for their reference data.

4.1 Entity Resolution

In the course of our EDA, we identified that the MAKE and CITY columns were populated with numerous typographical errors. Initially, we attempted to establish mappings for each column; however, given the extensive dataset and the multitude of error combinations, this method proved unsustainable. After conducting further research, we chose to apply the Jaro-Winkler Similarity algorithm to assess the closeness of the entries. Nonetheless, we quickly found that simply utilizing Jaro-Winkler was insufficient, as the algorithm could not understand which among similar entries was correct (e.g., CHICAGO versus CHCAGO). Therefore, we revised the logic to incorporate the dataset's row count to deduce the most frequent terms as the correct ones (see our create_similar_word_mappings function in the

referenced Jupyter notebook). It is important to note that fully automating typo correction is highly challenging. We experimented with various methods and thresholds to achieve the best results.

4.2 Database Normalization

To comply with 3NF principles, we needed to decompose our three primary tables, as several columns contained redundant data, breaching 3NF rules. For this, we developed a function named 'create_reference_table' (refer to the Jupyter notebook for more information), which helped us isolate the duplicated values into separate reference tables, each with its own Primary Key. Subsequently, we incorporated Foreign Keys in the original three tables, linking them to these reference tables. As a result, we generated an additional 36 tables, making a total of 39, including the initial three (see Appendix for more details).

4.3 Database Population

Considering the large volume of our datasets, filling the database proved challenging. We invested numerous hours dealing with repeated timeouts and failures throughout the task. After multiple efforts, we managed to perform the process on a local level, filling a local database, exporting every table to a file, and subsequently using the PostgreSQL COPY command to populate the remote database. After finishing the process, we conducted queries on each table to ensure the row counts matched those in our local database. For more information, see the "Populate the Database" section in our Jupyter notebook.

5 Queries

As described before, the main focus of our application was to show interesting insights related to the car crashes. Therefore, many of our queries required heavy use of aggregation over one or multiple datasets.

To keep things concise, this section only features two queries, with the remaining ones available in our Appendix. Be aware that these queries are the originals, and you can also view the updated materialized views in the Appendix.

5.1 Query 1: Crash Severity Index per City

This query calculates a severity index for each city, showing fatalities relative to crash volume. Cities with a high index have deadlier crashes on average, even if their overall crash counts are lower.

```
WITH crash_city AS (
    SELECT DISTINCT
        c.crash_record_id,
        p.city_id
    FROM crash c
    JOIN person p ON c.crash_record_id = p.crash_record_id
    WHERE p.city_id IS NOT NULL
),
```

```
city_stats AS (
    SELECT
        ci.name AS city_name,
        COUNT(*) AS total_crashes,
        SUM(c.injuries_fatal) AS total_fatalities
    FROM crash_city cc
    JOIN crash c ON cc.crash_record_id = c.crash_record_id
    JOIN city ci ON cc.city_id = ci.id
    GROUP BY ci.name
)
SELECT
    city_name,
    total_crashes,
    total_fatalities,
    ROUND(
        CASE WHEN total_crashes > 0
            THEN total_fatalities::NUMERIC / total_crashes
            ELSE 0 END,
        3
    ) AS severity_index
FROM city_stats
ORDER BY severity_index DESC
LIMIT 10;
```

5.2 Query 2: Top Vehicle Types in Fatal Crashes with Average Age

This query finds the top five vehicle types most frequently involved in fatal crashes and calculates the average age of their drivers. It helps identify whether certain vehicles are linked to fatal incidents and if driver age plays a role.

```
WITH fatal_crashes AS (
    SELECT DISTINCT crash_record_id
    FROM crash
    WHERE injuries_fatal > 0
)
SELECT vt.name AS vehicle_type,
    COUNT(*) AS fatal_crash_count,
    ROUND(AVG(p.age), 1) AS avg_driver_age
FROM fatal_crashes fc
JOIN vehicle v ON fc.crash_record_id = v.crash_record_id
JOIN vehicle_type vt ON v.vehicle_type_id = vt.id
JOIN person p ON v.vehicle_id = p.vehicle_id
JOIN person_type pt ON p.person_type_id = pt.id
WHERE pt.name = 'DRIVER'
GROUP BY vt.name
ORDER BY fatal_crash_count DESC
LIMIT 5;
```

6 Performance Evaluation

Our initial effort to execute our main page using the above queries ranged from 44 seconds to 1 minute and 5 seconds because of:

- Slow query performance
- Sequential execution (no parallelization)
- No caching

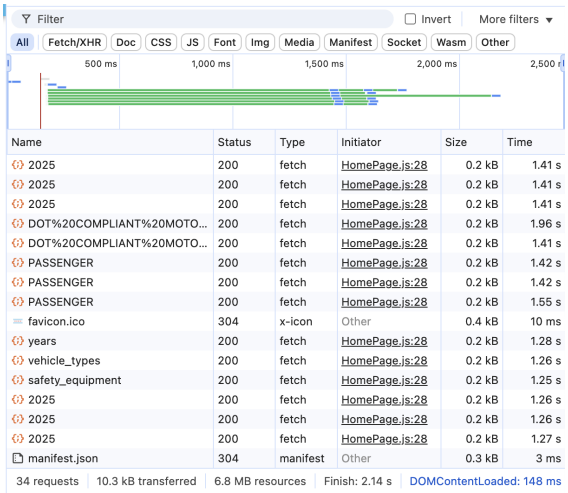
Initially, we enhanced our query performance by incorporating 28 indexes (see the list of indexes in our appendix), driving our loading times to a new range: from 21 seconds to 38 seconds. To improve further, we incorporated parallelization with Promise.all in React, enabling the queries to execute concurrently. Our implementation can be viewed in this file: `https://github.com/jorgefio/cis5500-group4/blob/main/client/src/pages/HomePage.js`

The updated loading times ranged from 8 to 14 seconds, displaying positive improvements, yet still remaining slow for users. It became evident that the primary bottlenecks were the queries involving aggregations. Given the large row counts, indexes by themselves were insufficient, so we decided to implement filtering conditions (such as year) and develop materialized views. We are showing one example here, but you can find more in our Appendix.

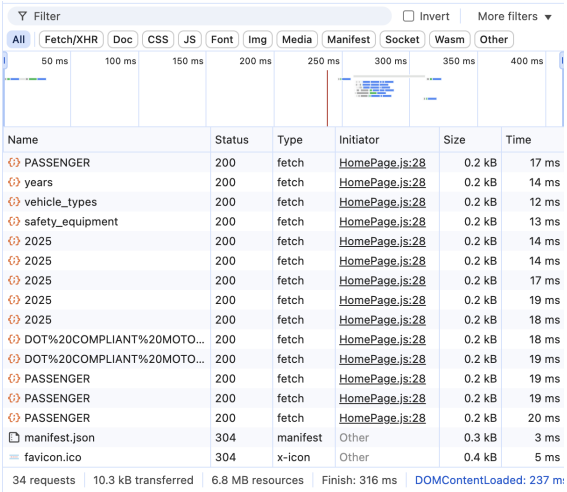
```
CREATE MATERIALIZED VIEW mv_city_crash_severity AS
SELECT c.crash_year,
    ci.name AS city_name,
    COUNT(DISTINCT c.crash_record_id) AS total_crashes,
    SUM(c.injuries_fatal) AS total_fatalities,
    ROUND(SUM(c.injuries_fatal)::NUMERIC / COUNT(DISTINCT c.crash_record_id), 3) AS severity_index
FROM crash c
JOIN person p ON c.crash_record_id = p.crash_record_id
JOIN city ci ON p.city_id = ci.id
WHERE p.city_id IS NOT NULL
GROUP BY c.crash_year, ci.name
HAVING SUM(c.injuries_fatal) > 0;
```

Following these changes, the loading times improved to a range of 1 to 3 seconds. Our last step in optimization involved implementing caching for 60 minutes in the server layer, detailed at: `https://github.com/jorgefio/cis5500-group4/blob/main/server/routes.js`.

With caching turned on, the latest load times decreased to between 300ms and 400ms. Refer to Figure 3.a and Figure 3.b for images showing the first loading without caching and the second with caching, respectively.



(a) Without Caching



(b) With Caching

Figure 3: Comparing with and without caching

In conclusion, we reduced the initial homepage loading time from over 44 seconds to less than 400 milliseconds (approximately 100 times faster). To accomplish this, we employed

techniques such as parallelization, indexing, modifying queries, creating materialized views, and implementing caching.

Query	Initial Time	Optimized Time	Optimization Technique
Query 1	19.4 s	169 ms	indexes, materialized view and caching *
Query 4	12.5 s	345 ms	indexes, materialized view and caching *
Query 3	11.2 s	605 ms	indexes and caching *
Query 2	9.94 s	396 ms	indexes, materialized view and caching *

Table 1: Recorded timings before and after optimization

(*) As previously mentioned, we employed a mix of indexes, materialized views, and caching techniques, as our intensive queries aggregated a substantial volume of rows. Our caching has a 60-minute expiration, thus it is beneficial only once the results are cached for specific parameters. For a comparison of all query times before and after optimization, see our performance enhancement Jupyter notebook at `https://github.com/jorgefio/cis5500-group4/blob/main/query-optimization.ipynb`.

7 Technical Challenges

During the project, we faced multiple technical challenges:

- Misleading Column Name
- Entity Resolution
- Normalization
- Database Population
- Aggregation Queries
- Bugs

7.1 Misleading Column Name

As mentioned before, the vehicles_df dataset contained a misleading column labeled VEHICLE_ID, which initially appeared to be the primary key of the dataset. Surprisingly, this column lacked any description in the accompanying data sheet, and its values did not follow the pattern of other identifiers. As we encountered difficulties linking datasets, we discovered that the true identifier was actually the column "CRASH_UNIT_ID". To add to the confusion, the people_df dataset also included a column named "VEHICLE_ID", yet this referred to "CRASH_UNIT_ID", according to their data sheet: "VEHICLE ID: The corresponding CRASH UNI ID from the Vehicles dataset" (see `https://data.cityofchicago.org/Transportation/Traffic-Crashes-People/u6pd-qa9d/about_data`). To eliminate any misconceptions, we removed VEHICLE_ID and renaming CRASH_UNIT_ID to VEHICLE_ID.

7.2 Entity Resolution

Our dataset contained 3 columns that were candidates for entity resolution: CITY, MAKE, and MODEL. These columns often contained repetitive values, caused by typos or variations in name representation. For instance, "UNIVERISTY PARK" instead of "UNIVERSITY PARK", or "TOYOTA" and "TOYOTA MOTOR COMPANY, LTD."

Initially, we attempted to create mappings for each variation, but given the dataset's size and the extensive combinations of errors, this approach wasn't feasible. Through further research, we discovered the 'jellyfish' library, which effectively compares strings using Jaro-Winkler Similarity. However, adjusting it to achieve optimal results for our dataset was challenging.

Ultimately, we devised our solution by integrating Jaro-Winkler Similarity with frequency tracking of each occurrence. The main challenge was that the standard Jaro-Winkler method didn't consistently select the right match. By analyzing the frequency of each option, we managed to map all variations correctly. To view our solution, visit `https://github.com/jorgefio/cis5500-group4/blob/main/data_cleaning_and_db_population_v5.ipynb`.

Additionally, we considered employing entity resolution for the MODEL column, but realized that model names can often be very similar, even within the same brand, making it difficult to distinguish a typo from an actual model name (e.g., Honda CR-V compared to Honda HR-V)

7.3 Normalization

As mentioned in the Data section, our three datasets were quite extensive and de-normalized, meaning they contained numerous columns with duplicate entries instead of using identifiers to link to other datasets. We discussed whether or not to fully normalize the database, considering the substantial work involved and potential performance downgrades due to the additional joins required.

Ultimately, we chose to normalize to full 3NF and leave any necessary de-normalization for later performance tuning. Given the requirement for a significant number of reference tables (36), we developed two utility functions: "create_reference_table" and "findFK". The first automatically inserted values along with their primary keys and kept the mapping in a dictionary, while the second utilized this mapping to retrieve the assigned key when inserting dependent rows. More details can be found here: `https://github.com/jorgefio/cis5500-group4/blob/main/data_cleaning_and_db_population_v5.ipynb`.

7.4 Database Population

Due to the size of our datasets, populating the database proved challenging. We dedicated numerous hours, dealing with several timeouts and process disruptions. Initially, we attempted to insert data row by row; however, the process was repeatedly interrupted by connectivity failures.

Following multiple unsuccessful tries, we considered batch processing. We tested various batch sizes to find a balance between delay and timeouts. This method worked with some datasets, but failed with the Crash dataset.

In the end, we overcame this issue by first inserting data into a local PostgreSQL database (normalized), and then utilizing PostgreSQL's COPY command to efficiently bulk upload to AWS.

7.5 Aggregation Queries

Our application's primary goal was to present insights into car accidents in Chicago, necessitating numerous queries that aggregate vast amounts of data, sometimes across different tables, leading to slow response times.

Normalizing our database exacerbated this situation due to the significant increase in the number of joins. Although we considered de-normalizing the database to boost query performance, the use of indexes, materialized views, query filters, and caching significantly improved efficiency.

7.6 Bugs

While simultaneously working on the project, our team occasionally encountered disruptions due to conflicts in our code base, blocking each other in real-time. For instance, when implementing caching across all backend queries, we unintentionally broke one of the search queries, resulting in a challenging bug that required several hours to fix.

7.7 Demo

During the presentation with the TAs, we received advice on further enhancing the search page's performance. By incorporating extra indexes, we successfully decreased the loading time to approximately 450ms. For a detailed list of the indexes and a screenshot of the page's performance, please see the Appendix.

8 Optional Extra Credit

Our team integrated Google Maps into the Crash Info Page of our site. The crash location is displayed on an interactive map under the "Crash Location" section of the page. This integration utilizes "react-google-maps".

We also created unittests for all our frontend pages and components, achieving over 80% of coverage.

```
-------------------|---------|----------|---------|---------|-------------------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------|---------|----------|---------|---------|-------------------
All files          |   92.24 |       80 |   81.48 |   92.85 |
 components        |     100 |      100 |     100 |     100 |
  GoogleMap.js     |     100 |      100 |     100 |     100 |
  NavBar.js        |     100 |      100 |     100 |     100 |
  VehicleInfo.js   |     100 |      100 |     100 |     100 |
 pages             |   92.09 |    78.84 |   80.76 |    92.7 |
  CrashInfoPage.js |   98.18 |    60.71 |    90.9 |   98.18 | 89
  ...SearchPage.js |   87.01 |      100 |    75.6 |    85.5 | ...10,223,236,268
  HomePage.js      |   84.48 |      100 |      64 |   89.09 | 84-343
  QuizPage.js      |     100 |      100 |     100 |     100 |
-------------------|---------|----------|---------|---------|-------------------

Test Suites: 7 passed, 7 total
Tests:       21 passed, 21 total
Snapshots:   0 total
Time:        2.965 s
```

Appendix

A Datasets

A.1 Crashes Dataset

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 934459 entries, 0 to 934458
Data columns (total 48 columns):
 #   Column                          Non-Null Count   Dtype
---  ------                          --------------   -----
 0   CRASH_RECORD_ID                 934459 non-null  object
 1   CRASH_DATE_EST_I                68625 non-null   object
 2   CRASH_DATE                      934459 non-null  object
 3   POSTED_SPEED_LIMIT              934459 non-null  int64
 4   TRAFFIC_CONTROL_DEVICE          934459 non-null  object
 5   DEVICE_CONDITION                934459 non-null  object
 6   WEATHER_CONDITION               934459 non-null  object
 7   LIGHTING_CONDITION              934459 non-null  object
 8   FIRST_CRASH_TYPE                934459 non-null  object
 9   TRAFFICWAY_TYPE                 934459 non-null  object
 10  LANE_CNT                        199023 non-null  float64
 11  ALIGNMENT                       934459 non-null  object
 12  ROADWAY_SURFACE_COND            934459 non-null  object
 13  ROAD_DEFECT                     934459 non-null  object
 14  REPORT_TYPE                     904711 non-null  object
 15  CRASH_TYPE                      934459 non-null  object
 16  INTERSECTION_RELATED_I          214881 non-null  object
 17  NOT_RIGHT_OF_WAY_I              42403 non-null   object
 18  HIT_AND_RUN_I                   293101 non-null  object
 19  DAMAGE                          934459 non-null  object
 20  DATE_POLICE_NOTIFIED            934459 non-null  object
 21  PRIM_CONTRIBUTORY_CAUSE         934459 non-null  object
 22  SEC_CONTRIBUTORY_CAUSE          934459 non-null  object
 23  STREET_NO                       934459 non-null  int64
 24  STREET_DIRECTION                934455 non-null  object
 25  STREET_NAME                     934458 non-null  object
 26  BEAT_OF_OCCURRENCE              934454 non-null  float64
 27  PHOTOS_TAKEN_I                  12944 non-null   object
 28  STATEMENTS_TAKEN_I              21778 non-null   object
 29  DOORING_I                       2914 non-null    object
 30  WORK_ZONE_I                     5127 non-null    object
 31  WORK_ZONE_TYPE                  3942 non-null    object
 32  WORKERS_PRESENT_I               1320 non-null    object
 33  NUM_UNITS                       934459 non-null  int64
 34  MOST_SEVERE_INJURY              932405 non-null  object
 35  INJURIES_TOTAL                  932419 non-null  float64
 36  INJURIES_FATAL                  932419 non-null  float64
 37  INJURIES_INCAPACITATING         932419 non-null  float64
 38  INJURIES_NON_INCAPACITATING     932419 non-null  float64
 39  INJURIES_REPORTED_NOT_EVIDENT   932419 non-null  float64
 40  INJURIES_NO_INDICATION          932419 non-null  float64
 41  INJURIES_UNKNOWN                932419 non-null  float64
 42  CRASH_HOUR                      934459 non-null  int64
 43  CRASH_DAY_OF_WEEK               934459 non-null  int64
 44  CRASH_MONTH                     934459 non-null  int64
 45  LATITUDE                        927565 non-null  float64
 46  LONGITUDE                       927565 non-null  float64
 47  LOCATION                        927565 non-null  object
dtypes: float64(11), int64(6), object(31)
memory usage: 342.2+ MB
```

A.2 Vehicles Dataset

```
CRASH_RECORD_ID                 172.874915
LOCATION                         89.989762
CRASH_TYPE                       76.342301
PRIM_CONTRIBUTORY_CAUSE          75.417576
MOST_SEVERE_INJURY               74.599820
DATE_POLICE_NOTIFIED             73.822261
CRASH_DATE                       73.822261
SEC_CONTRIBUTORY_CAUSE           71.438939
ALIGNMENT                        70.035114
REPORT_TYPE                      68.883605
TRAFFICWAY_TYPE                  66.413611
DEVICE_CONDITION                 66.162923
FIRST_CRASH_TYPE                 65.867324
TRAFFIC_CONTROL_DEVICE           64.708541
DAMAGE                           64.124594
LIGHTING_CONDITION               63.402232
STREET_NAME                      63.244009
ROAD_DEFECT                      62.075166
WEATHER_CONDITION                58.270070
ROADWAY_SURFACE_COND             56.718262
STREET_DIRECTION                 54.198518
HIT_AND_RUN_I                    37.523314
INTERSECTION_RELATED_I           35.489594
CRASH_DATE_EST_I                 31.686938
NOT_RIGHT_OF_WAY_I               31.005166
STATEMENTS_TAKEN_I               30.468916
PHOTOS_TAKEN_I                   30.239232
WORK_ZONE_TYPE                   30.044111
WORK_ZONE_I                      30.035990
DOORING_I                        29.978452
WORKERS_PRESENT_I                29.937008
CRASH_DAY_OF_WEEK                 7.475672
INJURIES_UNKNOWN                  7.475672
CRASH_HOUR                        7.475672
INJURIES_REPORTED_NOT_EVIDENT     7.475672
CRASH_MONTH                       7.475672
LONGITUDE                         7.475672
LATITUDE                          7.475672
INJURIES_NO_INDICATION            7.475672
STREET_NO                         7.475672
INJURIES_NON_INCAPACITATING       7.475672
INJURIES_INCAPACITATING           7.475672
INJURIES_FATAL                    7.475672
INJURIES_TOTAL                    7.475672
NUM_UNITS                         7.475672
BEAT_OF_OCCURRENCE                7.475672
LANE_CNT                          7.475672
POSTED_SPEED_LIMIT                7.475672
Index                             0.000128
dtype: float64
```

A.3 People Dataset

```
CRASH_RECORD_ID          379.561355
INJURY_CLASSIFICATION    164.020585
CRASH_DATE               162.082957
SAFETY_EQUIPMENT         146.675797
AIRBAG_DEPLOYED          146.202102
BAC_RESULT               132.641702
PERSON_ID                131.943782
PERSON_TYPE              130.635471
EJECTION                 124.897113
DRIVER_VISION            122.124796
DRIVER_ACTION            121.085078
```

```
SEX                     118.093828
PHYSICAL_CONDITION      117.098737
CITY                    114.303781
ZIPCODE                 106.924259
STATE                   106.596224
DRIVERS_LICENSE_STATE    98.106344
DRIVERS_LICENSE_CLASS    91.616070
HOSPITAL                 76.892022
EMS_AGENCY               71.773435
PEDPEDAL\_VISIBILITY      67.562772
PEDPEDAL_ACTION          67.465286
PEDPEDAL_LOCATION        67.092355
EMS_RUN_NO               66.599832
CELL_PHONE_USE           65.684016
AGE                      16.413464
SEAT_NO                  16.413464
VEHICLE_ID               16.413464
BAC_RESULT VALUE         16.413464
Index                     0.000128
dtype: float64
\begin{verbatim}
```

B Created Indexes

```
CREATE INDEX idx_person_crash_city ON person(crash_record_id, city_id) WHERE city_id IS NOT NULL;
CREATE INDEX idx_city_name ON city(name);
CREATE INDEX idx_crash_injuries ON crash(crash_record_id, injuries_fatal);
CREATE INDEX idx_vehicle_crash_type_id ON vehicle(crash_record_id, vehicle_type_id, vehicle_id);
CREATE INDEX idx_person_vehicle_driver ON person(vehicle_id, person_type_id, age);
CREATE INDEX idx_person_type_driver ON person_type(id) WHERE name = 'DRIVER';
CREATE INDEX idx_vehicle_type_name ON vehicle_type(id, name);
CREATE INDEX idx_person_driver_safety ON person(person_type_id, safety_equipment_id, injury_classification_id, crash_record_id);
CREATE INDEX idx_crash_year ON crash(crash_year);
CREATE INDEX idx_vehicle_travel_crash ON vehicle(travel_direction_id, crash_record_id);
CREATE INDEX idx_vehicle_covering ON vehicle(travel_direction_id, crash_record_id, vehicle_id);
CREATE INDEX idx_travel_direction_covering ON travel_direction(id, name);
CREATE INDEX idx_vehicle_type_crash ON vehicle(vehicle_type_id, crash_record_id);
CREATE INDEX idx_vehicle_type ON vehicle(vehicle_type_id);
CREATE INDEX idx_vehicle_age_crash ON vehicle(crash_record_id, vehicle_year, vehicle_type_id);
CREATE INDEX idx_crash_year_fatal ON crash(crash_record_id, crash_year, injuries_fatal);
CREATE INDEX idx_crash_type_fatalities ON crash(first_crash_type_id, injuries_fatal);
CREATE INDEX idx_first_crash_type_id_name ON first_crash_type(id, name);
CREATE INDEX idx_crash_weather_fatalities ON crash(weather_condition_id, injuries_fatal);
CREATE INDEX idx_weather_condition_id_name ON weather_condition(id, name);
CREATE INDEX idx_crash_filters ON crash(posted_speed_limit, injuries_total, crash_hour, crash_record_id);
CREATE INDEX idx_weather_condition_name ON weather_condition(name);
CREATE INDEX idx_lighting_condition_name ON lighting_condition(name);
CREATE INDEX idx_first_crash_type_name ON first_crash_type(name);
CREATE INDEX idx_trafficway_type_name ON trafficway_type(name);
CREATE INDEX idx_roadway_surface_condition_name ON roadway_surface_condition(name);
CREATE INDEX idx_damage_name ON damage(name);
CREATE INDEX idx_safety_equipment_name ON safety_equipment(name);
CREATE INDEX idx_crash_search_filters ON crash(posted_speed_limit, injuries_total, crash_hour);
CREATE INDEX idx_crash_record_id_text ON crash(crash_record_id text_pattern_ops);
CREATE INDEX idx_crash_weather_condition_id ON crash(weather_condition_id);
CREATE INDEX idx_crash_lighting_condition_id ON crash(lighting_condition_id);
CREATE INDEX idx_crash_crash_type_id ON crash(crash_type_id);
CREATE INDEX idx_crash_trafficway_type_id ON crash(trafficway_type_id);
CREATE INDEX idx_crash_roadway_surface_cond_id ON crash(roadway_surface_cond_id);
CREATE INDEX idx_crash_damage_id ON crash(damage_id);
CREATE INDEX idx_crash_search_filters ON crash(posted_speed_limit, injuries_total, crash_hour);
CREATE INDEX idx_crash_record_id_text ON crash(crash_record_id text_pattern_ops);
CREATE INDEX idx_crash_weather_condition_id ON crash(weather_condition_id);
CREATE INDEX idx_crash_lighting_condition_id ON crash(lighting_condition_id);
CREATE INDEX idx_crash_crash_type_id ON crash(crash_type_id);
```

```
CREATE INDEX idx_crash_trafficway_type_id ON crash(trafficway_type_id);
CREATE INDEX idx_crash_roadway_surface_cond_id ON crash(roadway_surface_cond_id);
CREATE INDEX idx_crash_damage_id ON crash(damage_id);
```

C Materialized Views

```
CREATE MATERIALIZED VIEW mv_city_crash_severity AS
SELECT c.crash_year,
    ci.name AS city_name,
    COUNT(DISTINCT c.crash_record_id) AS total_crashes,
    SUM(c.injuries_fatal) AS total_fatalities,
    ROUND(SUM(c.injuries_fatal)::NUMERIC / COUNT(DISTINCT c.crash_record_id), 3) AS severity_index
FROM crash c
JOIN person p ON c.crash_record_id = p.crash_record_id
JOIN city ci ON p.city_id = ci.id
WHERE p.city_id IS NOT NULL
GROUP BY c.crash_year, ci.name
HAVING SUM(c.injuries_fatal) > 0;

CREATE MATERIALIZED VIEW mv_vehicle_type_fatal_driver_stats AS
SELECT vt.name AS vehicle_type,
    COUNT(*) AS fatal_crash_count,
    ROUND(AVG(p.age), 1) AS avg_driver_age
FROM crash c
JOIN vehicle v ON c.crash_record_id = v.crash_record_id
JOIN vehicle_type vt ON v.vehicle_type_id = vt.id
JOIN person p ON v.vehicle_id = p.vehicle_id
JOIN person_type pt ON p.person_type_id = pt.id
WHERE c.injuries_fatal > 0
AND pt.name = 'DRIVER'
GROUP BY vt.name;

CREATE MATERIALIZED VIEW mv_travel_direction_stats AS
SELECT c.crash_year,
    td.name AS travel_direction,
    COUNT(*) AS vehicle_count,
    SUM(c.injuries_fatal) AS total_fatalities
FROM vehicle v
JOIN travel_direction td ON v.travel_direction_id = td.id
JOIN crash c ON v.crash_record_id = c.crash_record_id
GROUP BY c.crash_year, td.name;

CREATE MATERIALIZED VIEW mv_vehicle_fatality_stats AS
    SELECT vt.name AS v,
        COUNT(*) AS vehicle_count,
        SUM(c.injuries_fatal) AS total_fatalities,
        ROUND(100.0 * SUM(c.injuries_fatal) / COUNT(*), 2) AS fatal_rate_pct
    FROM vehicle v
    JOIN vehicle_type vt ON v.vehicle_type_id = vt.id
    JOIN crash c ON v.crash_record_id = c.crash_record_id
    GROUP BY vt.name;

CREATE MATERIALIZED VIEW mv_vehicle_age_stats AS
SELECT vt.name AS vehicle_type,
    CASE
        WHEN c.crash_year - v.vehicle_year < 5 THEN 'Under 5'
        WHEN c.crash_year - v.vehicle_year BETWEEN 5 AND 10 THEN '5-10'
        WHEN c.crash_year - v.vehicle_year BETWEEN 10 AND 25 THEN '10-25'
        ELSE '25+' END AS vehicle_age,
    COUNT(*) AS vehicle_count,
    SUM(c.injuries_fatal) AS total_fatalities,
    ROUND(100.0 * SUM(c.injuries_fatal) / COUNT(*), 2) AS fatal_rate_pct
FROM vehicle v
JOIN vehicle_type vt ON v.vehicle_type_id = vt.id
JOIN crash c ON v.crash_record_id = c.crash_record_id
GROUP BY vt.name, vehicle_age;
```

```
CREATE MATERIALIZED VIEW mv_safety_equipment_driver_stats AS
SELECT se.name AS safety_equipment,
    COUNT(*) AS total_drivers,
    SUM(CASE WHEN ic.name = 'FATAL' THEN 1 ELSE 0 END) AS fatal_drivers
FROM person p
JOIN safety_equipment se ON p.safety_equipment_id = se.id
JOIN injury_classification ic ON p.injury_classification_id = ic.id
WHERE p.person_type_id = (SELECT id FROM person_type WHERE name = 'DRIVER')
GROUP BY se.name;

CREATE MATERIALIZED VIEW mv_primary_cause_fatal_stats AS
SELECT c.crash_year,
    pcc.name AS primary_cause,
    COUNT(*) AS fatal_crashes
FROM crash c
JOIN prim_contributory_cause pcc ON c.prim_contributory_cause_id = pcc.id
WHERE c.injuries_fatal > 0
GROUP BY c.crash_year, pcc.name;

CREATE MATERIALIZED VIEW mv_weather_condition_stats AS
SELECT c.crash_year,
    wc.name AS weather_condition,
    COUNT(*) AS total_crashes,
    SUM(c.injuries_fatal) AS fatal_crashes
FROM crash c
JOIN weather_condition wc ON c.weather_condition_id = wc.id
GROUP BY c.crash_year, wc.name;

CREATE MATERIALIZED VIEW mv_crash_type_fatality_stats AS
SELECT c.crash_year,
    ct.name AS crash_type,
    COUNT(*) AS total_crashes,
    SUM(c.injuries_fatal) AS total_fatalities,
    ROUND(100.0 * SUM(c.injuries_fatal) / COUNT(*), 2) AS fatality_rate_pct
FROM crash c
JOIN first_crash_type ct ON c.first_crash_type_id = ct.id
GROUP BY c.crash_year, ct.name;
```

D Initial Queries

D.1 Query 1: Crash Severity Index per City

This query calculates a severity index for each city, showing fatalities relative to crash volume. Cities with a high index have deadlier crashes on average, even if their overall crash counts are lower.

```
WITH crash_city AS (
    SELECT DISTINCT
        c.crash_record_id,
        p.city_id
    FROM crash c
    JOIN person p ON c.crash_record_id = p.crash_record_id
    WHERE p.city_id IS NOT NULL
),
city_stats AS (
    SELECT
        ci.name AS city_name,
        COUNT(*) AS total_crashes,
        SUM(c.injuries_fatal) AS total_fatalities
    FROM crash_city cc
    JOIN crash c ON cc.crash_record_id = c.crash_record_id
    JOIN city ci ON cc.city_id = ci.id
```

```
    GROUP BY ci.name
)
SELECT
    city_name,
    total_crashes,
    total_fatalities,
    ROUND(
        CASE WHEN total_crashes > 0
            THEN total_fatalities::NUMERIC / total_crashes
            ELSE 0 END,
        3
    ) AS severity_index
FROM city_stats
ORDER BY severity_index DESC
LIMIT 10;
```

D.2 Query 2: Top Vehicle Types in Fatal Crashes with Average Age

This query finds the top five vehicle types most frequently involved in fatal crashes and calculates the average age of their drivers. It helps identify whether certain vehicles are linked to fatal incidents and if driver age plays a role.

```
WITH fatal_crashes AS (
    SELECT DISTINCT crash_record_id
    FROM crash
    WHERE injuries_fatal > 0
)
SELECT vt.name AS vehicle_type,
    COUNT(*) AS fatal_crash_count,
    ROUND(AVG(p.age), 1) AS avg_driver_age
FROM fatal_crashes fc
JOIN vehicle v ON fc.crash_record_id = v.crash_record_id
JOIN vehicle_type vt ON v.vehicle_type_id = vt.id
JOIN person p ON v.vehicle_id = p.vehicle_id
JOIN person_type pt ON p.person_type_id = pt.id
WHERE pt.name = 'DRIVER'
GROUP BY vt.name
ORDER BY fatal_crash_count DESC
LIMIT 5;
```

D.3 Query 3: Driver Age Groups and Safety Equipment Effectiveness

This query shows how effective different safety equipment types are for drivers across age groups by comparing fatality counts. It helps uncover whether certain age groups benefit less from protective measures.

```
WITH drivers AS (
    SELECT p.*, se.name AS safety_equipment, ic.name AS injury_level
    FROM person p
    JOIN person_type pt ON p.person_type_id = pt.id
    JOIN safety_equipment se ON p.safety_equipment_id = se.id
    JOIN injury_classification ic ON p.injury_classification_id = ic.id
    WHERE pt.name = 'DRIVER'
)
SELECT
    CASE
        WHEN age < 25 THEN 'Under 25'
```

```
        WHEN age BETWEEN 25 AND 64 THEN '25-64'
        ELSE '65+' END AS age_group,
    safety_equipment,
    COUNT(*) AS driver_count,
    SUM(CASE WHEN injury_level = 'FATAL' THEN 1 ELSE 0 END) AS fatalities
FROM drivers
GROUP BY age_group, safety_equipment
ORDER BY age_group, fatalities DESC;
```

D.4 Query 4: Safety Equipment vs. Injury Outcomes

This query groups drivers by the type of safety equipment (e.g. seat belt, helmet) they were using and counts how many sustained fatal injuries. It joins person to person_type to filter only drivers, and to safety_equipment and injury_classification for equipment and injury status, as well as crash for context. Note: this query takes some time to run, we will add some indexes to improve its performance.

```
SELECT se.name AS safety_equipment,
    COUNT(*) AS total_drivers,
    SUM(CASE WHEN ic.name = 'FATAL' THEN 1 ELSE 0 END) AS fatal_drivers
FROM person p
JOIN person_type pt ON p.person_type_id = pt.id
JOIN crash c ON p.crash_record_id = c.crash_record_id
JOIN safety_equipment se ON p.safety_equipment_id = se.id
JOIN injury_classification ic ON p.injury_classification_id = ic.id
WHERE pt.name = 'DRIVER'
GROUP BY se.name;
```

D.5 Query 5: Crashes by Travel Direction

Groups vehicles by their travel direction at the time of crash (e.g. N, S). It counts the number of vehicles and sums fatalities for each direction category.

```
SELECT td.name AS travel_direction,
    COUNT(*) AS vehicle_count,
    SUM(c.injuries_fatal) AS total_fatalities
FROM public.vehicle v
JOIN public.travel_direction td ON v.travel_direction_id = td.id
JOIN public.crash c ON v.crash_record_id = c.crash_record_id
GROUP BY td.name
ORDER BY vehicle_count DESC;
```

D.6 Query 6: Crashes per Vehicle Type

Measures how often each vehicle type (passenger, van/mini-van, bus, etc.) is involved in fatal crashes. It counts vehicles by type and sums the fatalities in their crashes, computing the fatality rate per vehicle.

```
SELECT vt.name AS v,
    COUNT(*) AS vehicle_count,
    SUM(c.injuries_fatal) AS total_fatalities,
    ROUND(100.0 * SUM(c.injuries_fatal) / COUNT(*), 2) AS fatal_rate_pct
FROM public.vehicle v
JOIN public.vehicle_type vt ON v.vehicle_type_id = vt.id
JOIN public.crash c ON v.crash_record_id = c.crash_record_id
GROUP BY vt.name
ORDER BY fatal_rate_pct DESC;
```

D.7 Query 7: Average Age of Vehicles

Computes the average age of vehicles (at the time of the crash) grouped by vehicle type. For each unit type (motorcycle, truck, etc.), it subtracts the vehicle's model year from the crash year and averages the result.

```
SELECT vt.name AS vehicle_type,
    ROUND(AVG(EXTRACT(YEAR FROM v.crash_date) - v.vehicle_year), 2) AS avg_vehicle_age
FROM public.vehicle v
JOIN public.vehicle_type vt ON v.vehicle_type_id = vt.id
GROUP BY vt.name
ORDER BY avg_vehicle_age DESC;
```

D.8 Query 8: Total Crashes and Fatalities

Calculates the total crashes and fatalities for each crash type, then computes the percentage of crashes that were fatal. This highlights which types of crashes (e.g. head on, rear end, overturned) are the deadliest.

```
SELECT ct.name AS crash_type, COUNT(*) AS total_crashes, SUM(c.injuries_fatal) AS total_fatalities,
    ROUND(100.0 * SUM(c.injuries_fatal) / COUNT(*), 2) AS fatality_rate_pct
FROM public.crash c
JOIN public.first_crash_type ct ON c.first_crash_type_id = ct.id
GROUP BY ct.name
ORDER BY fatality_rate_pct DESC;
```

D.9 Query 9: Top Primary Contributory Causes

Identifies the top primary contributory causes of crashes that resulted in at least one fatality. By filtering on injuries_fatal larger than 0, it counts how many fatal crashes were attributed to each primary cause (like exceeding the authorized speed limit, physical condition of the driver, etc.).

```
SELECT pcc.name AS primary_cause, COUNT(*) AS fatal_crashes
FROM public.crash c
JOIN public.prim_contributory_cause pcc ON c.prim_contributory_cause_id = pcc.id
WHERE c.injuries_fatal > 0
GROUP BY pcc.name
ORDER BY fatal_crashes DESC;
```

D.10 Query 10: Weather Conditions

Joins crashes to the weather condition at the time of each crash, counting crashes (and summing fatal injuries) per weather category. This reveals which weather (rain, snow, fog/smoke/haze, etc.) produces the most crashes and fatalities.

```
SELECT wc.name AS weather_condition, COUNT(*) AS total_crashes, SUM(c.injuries_fatal) AS fatal_crashes
FROM public.crash c
JOIN public.weather_condition wc ON c.weather_condition_id = wc.id
GROUP BY wc.name
ORDER BY total_crashes DESC;
```

E Final Queries

E.1 Query 1: Crash Severity Index per City

This query calculates a severity index for each city, showing fatalities relative to crash volume. Cities with a high index have deadlier crashes on average, even if their overall crash counts are lower.

```
SELECT city_name, total_crashes, total_fatalities, severity_index
FROM mv_city_crash_severity
WHERE crash_year = '2017'
ORDER BY severity_index DESC
LIMIT 5;
```

E.2 Query 2: Top Vehicle Types in Fatal Crashes with Average Age

This query finds the top five vehicle types most frequently involved in fatal crashes and calculates the average age of their drivers. It helps identify whether certain vehicles are linked to fatal incidents and if driver age plays a role.

```
SELECT vehicle_type, fatal_crash_count, avg_driver_age
FROM mv_vehicle_type_fatal_driver_stats
ORDER BY fatal_crash_count DESC;
```

E.3 Query 3: Driver Age Groups and Safety Equipment Effectiveness

This query shows how effective different safety equipment types are for drivers across age groups by comparing fatality counts. It helps uncover whether certain age groups benefit less from protective measures.

```
WITH drivers AS (
     SELECT p.*, se.name AS safety_equipment, ic.name AS injury_level
     FROM person p
     JOIN person_type pt ON p.person_type_id = pt.id
     JOIN safety_equipment se ON p.safety_equipment_id = se.id
     JOIN injury_classification ic ON p.injury_classification_id = ic.id
     WHERE pt.name = 'DRIVER' AND se.name = 'SAFETY BELT NOT USED'
 )SELECT
          CASE
              WHEN age < 25 THEN 'Under 25'
              WHEN age BETWEEN 25 AND 64 THEN '25-64'
              ELSE '65+' END AS age_group,
          COUNT(*) AS driver_count,
          SUM(CASE WHEN injury_level = 'FATAL' THEN 1 ELSE 0 END) AS fatalities
      FROM drivers
      GROUP BY age_group
      ORDER BY fatalities DESC;
```

E.4 Query 4: Safety Equipment vs. Injury Outcomes

This query groups drivers by the type of safety equipment (e.g. seat belt, helmet) they were using and counts how many sustained fatal injuries. It joins person to person_type

to filter only drivers, and to safety_equipment and injury_classification for equipment and injury status, as well as crash for context. Note: this query takes some time to run, we will add some indexes to improve its performance.

```
SELECT travel_direction, vehicle_count, total_fatalities
FROM mv_travel_direction_stats
WHERE crash_year = '2025'
ORDER BY vehicle_count DESC
```

E.5 Query 5: Crashes by Travel Direction

Groups vehicles by their travel direction at the time of crash (e.g. N, S). It counts the number of vehicles and sums fatalities for each direction category.

```
SELECT * FROM mv_vehicle_fatality_stats
WHERE v = 'PASSENGER'
ORDER BY fatal_rate_pct DESC;
```

E.6 Query 6: Crashes per Vehicle Type

Measures how often each vehicle type (passenger, van/mini-van, bus, etc.) is involved in fatal crashes. It counts vehicles by type and sums the fatalities in their crashes, computing the fatality rate per vehicle.

```
SELECT * FROM mv_vehicle_age_stats ORDER BY total_fatalities DESC;
```

E.7 Query 7: Average Age of Vehicles

Computes the average age of vehicles (at the time of the crash) grouped by vehicle type. For each unit type (motorcycle, truck, etc.), it subtracts the vehicle's model year from the crash year and averages the result.

```
SELECT crash_type, total_crashes, total_fatalities, fatality_rate_pct
FROM mv_crash_type_fatality_stats
WHERE crash_year = '2017'
ORDER BY fatality_rate_pct DESC
LIMIT 5;
```

E.8 Query 8: Total Crashes and Fatalities

Calculates the total crashes and fatalities for each crash type, then computes the percentage of crashes that were fatal. This highlights which types of crashes (e.g. head on, rear end, overturned) are the deadliest.

```
SELECT wc.name AS weather_condition, COUNT(*) AS total_crashes, SUM(c.injuries_fatal) AS fatal_crashes
FROM public.crash c
JOIN public.weather_condition wc ON c.weather_condition_id = wc.id
GROUP BY wc.name
ORDER BY total_crashes DESC
LIMIT 5;
```

E.9 Query 9: Top Primary Contributory Causes

Identifies the top primary contributory causes of crashes that resulted in at least one fatality. By filtering on injuries_fatal larger than 0, it counts how many fatal crashes were attributed to each primary cause (like exceeding the authorized speed limit, physical condition of the driver, etc.).

```
SELECT primary_cause, fatal_crashes
FROM mv_primary_cause_fatal_stats
WHERE crash_year = '2025'
ORDER BY fatal_crashes DESC
LIMIT 5;
```

E.10 Query 10: Weather Conditions

Joins crashes to the weather condition at the time of each crash, counting crashes (and summing fatal injuries) per weather category. This reveals which weather (rain, snow, fog/smoke/haze, etc.) produces the most crashes and fatalities.

```
SELECT weather_condition, total_crashes, fatal_crashes
FROM mv_weather_condition_stats
WHERE crash_year = '2025'
ORDER BY total_crashes DESC
LIMIT 5;
```

F Search Page Performance

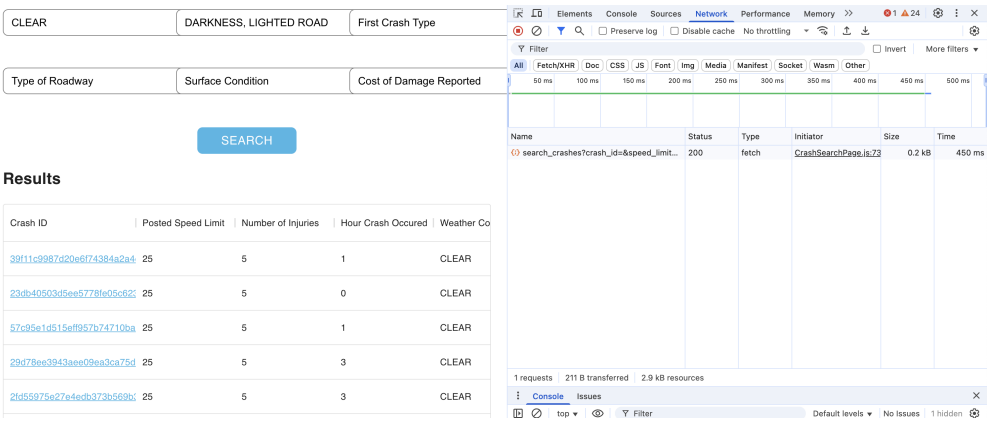


Figure 4: Search Page Improvements

G Database Schema

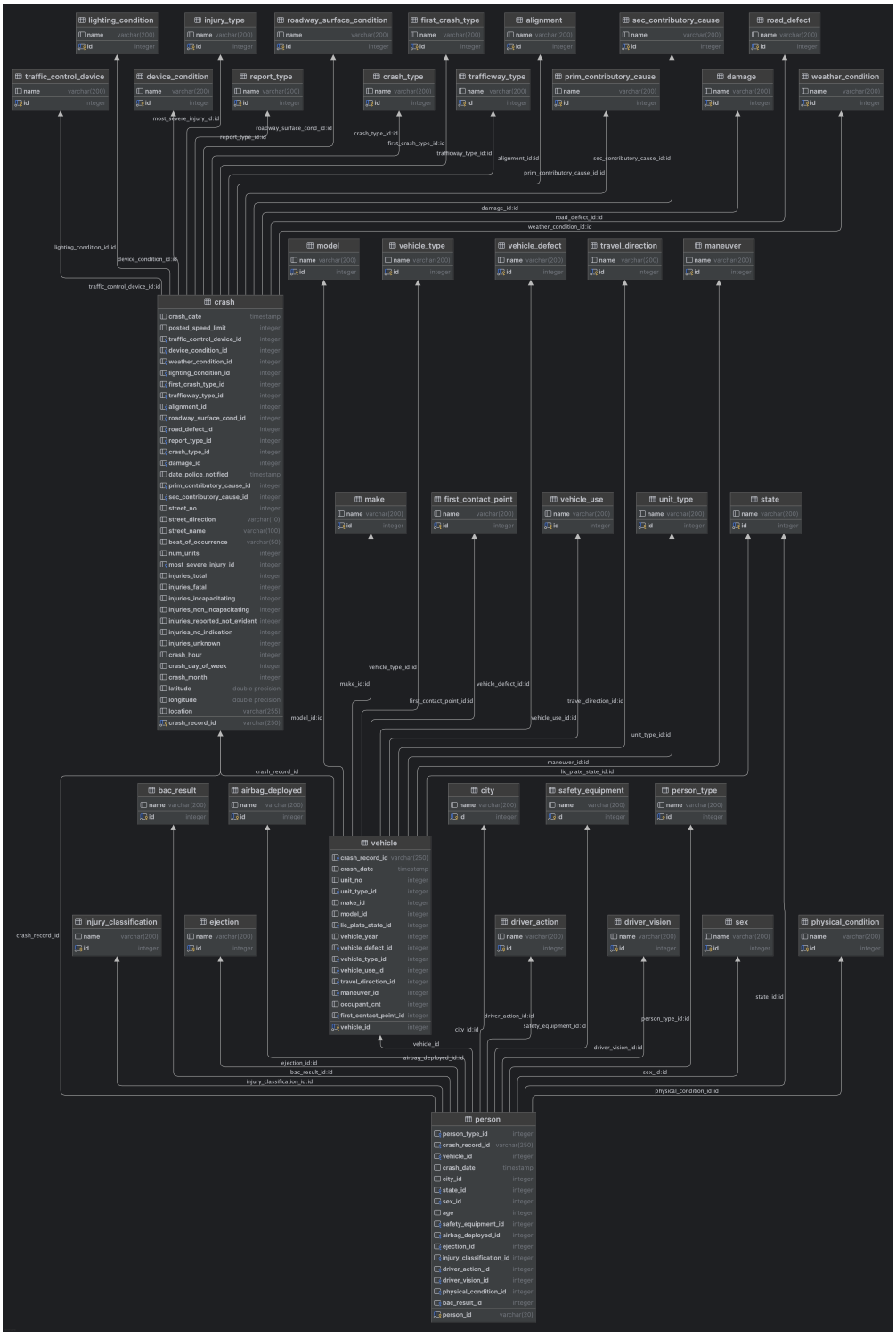


Figure 5: Database Schema