

DS-5420 Database Management

Project 2

NYPD Crime Complaint Data Historic (2006-2019)

<https://www.kaggle.com/brunacmendes/nypd-complaint-data-historic-20062019>

Ty Painter & Tonnar Castellano

Introduction

For this project we were tasked with finding a data set that was over 30 columns and 100MB to create a front-end application with a back-end in MySQL. In order to complete this task, we selected a dataset from kaggle.com that revolved around complaint data reported to New York City Police Department (NYPD). Specifically, the dataset focused on various complaints, the associated crime type, basic description of the victim and suspects, and the location. The original data set was 2.22GB with 35 columns and almost seven million observations.

One of the biggest problems with this dataset is due to the size of the file, because the dataset is so large searching or storing the information is exceptionally difficult. The other major problem was the amount of dirty data in the general corpus; many columns in the dataset had missing values and data was entered in the wrong format. To rectify this problem, many cleaning steps were taken such as fixing date formats and the database was broken into more manageable and intuitive sections. This process will be described more in the final implementation.

The application that we created will be centered around giving police easy access to crime data with various features such as viewing, updating, inserting, and deleting new complaint numbers and the associated crimes. The application could also allow the public to view specific crime related information along with general summary statistics. Currently, data of this magnitude is hard to come by often due to paywalls, this front-end application will help to democratize the data. Hopefully, this application, if given to the NYPD, would allow them to manage the current crime database in a more efficient manner from both an instinctive and functional standpoint. Also, the application would allow for better information sharing between the police and the public which is important in today's political climate. However, the main goal of this project was just to give the NYC police an easier way to organize data; the idea of sharing with the public would be a later add-on to the work done in this project.

Final Implementation

The final implementation of our database was to establish a platform that allows the NYPD to be able to easily interact with complaint records received by the NYPD. The database accommodates various functionalities including searching, inserting,

updating, and deleting police records in regard to the greater New York City Area. The database's back-end system is supported by MySQL Workbench and the front-end platform is a web application written in PHP.

The user-interface portion of the front-end application has 6 different web tabs in coordination with DML (insert, update, delete, and select) commands. Each tab uses an advanced feature to complete the DML actions. The "Search Complaint" tab uses a stored procedure to display various columns of information, from multiple tables about a specific "complaint_num." The "Delete Crime" tab incorporates another stored procedure to delete a record by having the user enter a "complaint_num." The "Insert Crime" and "Update Crime" tabs are also completed by stored procedures, but also cause various triggers to run to ensure update and insert commands are in agreement with data types and functional dependencies. The last two tabs, "Crime by Borough" and "Crime by Type" use the view functionality, which requires no user input. These tabs will be described in more detail in the implementation walkthrough section.

The final database design is a product of extensive data cleaning and manipulation. The two major problems we encountered were products of poor data quality. The first was how missing information was handled and the second was column data types. The missing data was made uniform by converting each column's "NA" and blank values into "NULL" values to satisfy specific data types. Date columns also had to be read in as VARCHAR and then converted to a DATE format. The next major problem with the data was that some of the data had inconsistencies with potential identifiers. In order to handle this, we were forced to write SQL queries to find the invalid data then sanity check it before adding it to the main tables.

The next step is to make the database into third normal form for ease and clarity of use. This was achieved by assigning data to 8 different tables: complaint_info, victim_info, suspect_info, pd, crime_info, extra_info, pd, jurisdiction, offense, plus a "dirty data" table. Complaint number is the primary key of our central table, complaint_info, but is also the primary key for all other tables except pd, jurisdiction, and offense.

We decided on complaint info as the central table because that would allow the police to have a main table to which everything else would be attached i.e., with complaint number. We then decided to pull out victim info and suspect info into its own

tables for considerations in terms of data access. This allows for managing access in the future to who is able to see/edit who the suspect is or whom the victim is from a database administration perspective helping to enforce privacy. Also, with regards to cardinalities each complaint number can belong to many victims and/or suspects but each victim or suspect can only have one complaint number for that instance of crime. From a logical perspective, this makes sense because each suspect/victim would have a single complaint number, even if there were multiple suspects/victims involved with a specific crime to keep the data unique to each combination of complaint number and suspect/victim description.

As mentioned above the main purpose of the database is to try to create a way of organizing data with respect to crime. As such, crime info was also broken up into a separate table in order to help manage the amount of information in one table. We decided on a one-to-one relationship, where each complaint number belongs to one reported crime and each reported crime has one complaint number. We did this because it would be impossible to have the same type of crime, occur at the same time, at the same location, with the same people involved. This would enforce the one-to-one relationship between complaint number and crime info.

The 3 tables below were formed after running many functional dependency tests and finding 3 distinctive functional dependencies. To abide by 3rd Normal Form rules, we created three tables: pd, jurisdiction, and offense. In the pd table, "pd_code" determines "pd_description." In the jurisdiction table, "jurisdiction_code" determines "jurisdiction_desc." In offense, "ky_code" determines "offense_desc." All 3 of these primary keys are used as foreign keys in the central, complaint info table. Also, all three of these have a one-to-many relationship with complaint numbers; each complaint number can have one of each of these three codes and each of these codes can be used by many complaint numbers.

jurisdiction_code Functional Dependency

```
40 • SELECT count(*) AS count, jurisdiction_code
41 FROM
42 (SELECT count(*) as count, jurisdiction_code, jurisdiction_desc
43 FROM police_mega
44 GROUP BY jurisdiction_code, jurisdiction_desc
45 ORDER BY jurisdiction_code) as new_table
46 GROUP BY jurisdiction_code
47 HAVING count > 1; # null
```

100% 7:39

Result Grid Filter Rows: Search Export:

count	jurisdiction_code
4	NULL

ky_code Functional Dependency

```
11 • SELECT ky_code, count(*) AS count
12 FROM
13 (SELECT count(*) as count, ky_code, offense_desc
14 FROM police_mega
15 GROUP BY ky_code, offense_desc
16 HAVING offense_desc IS NOT NULL
17 ORDER BY ky_code) as new_table
18 GROUP BY ky_code
19 HAVING count > 1;
```

100% 18:19

Result Grid Filter Rows: Search Export:

ky_code	count
116	2
120	2
124	3
125	2
343	2
345	2
364	3
677	2

pd_code Functional Dependency

```
26 • SELECT pd_code, count(*) AS count
27 FROM
28 (SELECT count(*) as count, pd_code, pd_desc
29 FROM police_mega
30 GROUP BY pd_code, pd_desc
31 ORDER BY pd_code) as new_table
32 GROUP BY pd_code
33 HAVING count > 1;
```

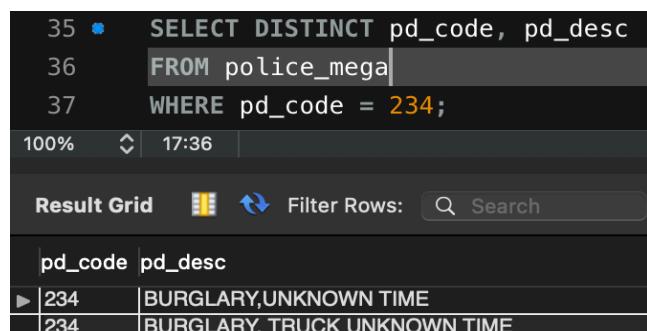
100% 18:33

Result Grid Filter Rows: Search Export:

pd_code	count
234	2
694	2

There was additional data cleaning done to enforce the functional dependencies in the pd and offense tables. There seemed to be some data entry errors where a certain code produced two descriptions that implied the same meaning on a few different occasions. The two different descriptions were conformed into one, consistent result to enforce the functional dependencies. This was the case for both ky_code and pd_code. The jurisdiction_code issue involved NULL values and was resolved by moving the NULL values into the dirty_data table.

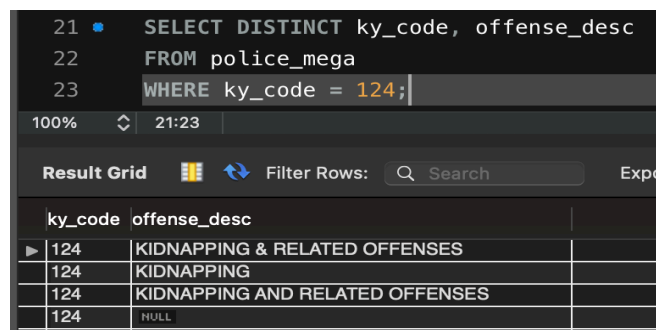
pd_code Cleaning



The screenshot shows a SQL query in a dark-themed editor. The query is: `SELECT DISTINCT pd_code, pd_desc FROM police_mega WHERE pd_code = 234;`. Below the query, a 'Result Grid' is displayed with two columns: 'pd_code' and 'pd_desc'. There are two rows of data.

pd_code	pd_desc
234	BURGLARY, UNKNOWN TIME
234	BURGLARY, TRUCK UNKNOWN TIME

ky_code Cleaning



The screenshot shows a SQL query in a dark-themed editor. The query is: `SELECT DISTINCT ky_code, offense_desc FROM police_mega WHERE ky_code = 124;`. Below the query, a 'Result Grid' is displayed with two columns: 'ky_code' and 'offense_desc'. There are four rows of data, including NULL values.

ky_code	offense_desc
124	KIDNAPPING & RELATED OFFENSES
124	KIDNAPPING
124	KIDNAPPING AND RELATED OFFENSES
124	NULL

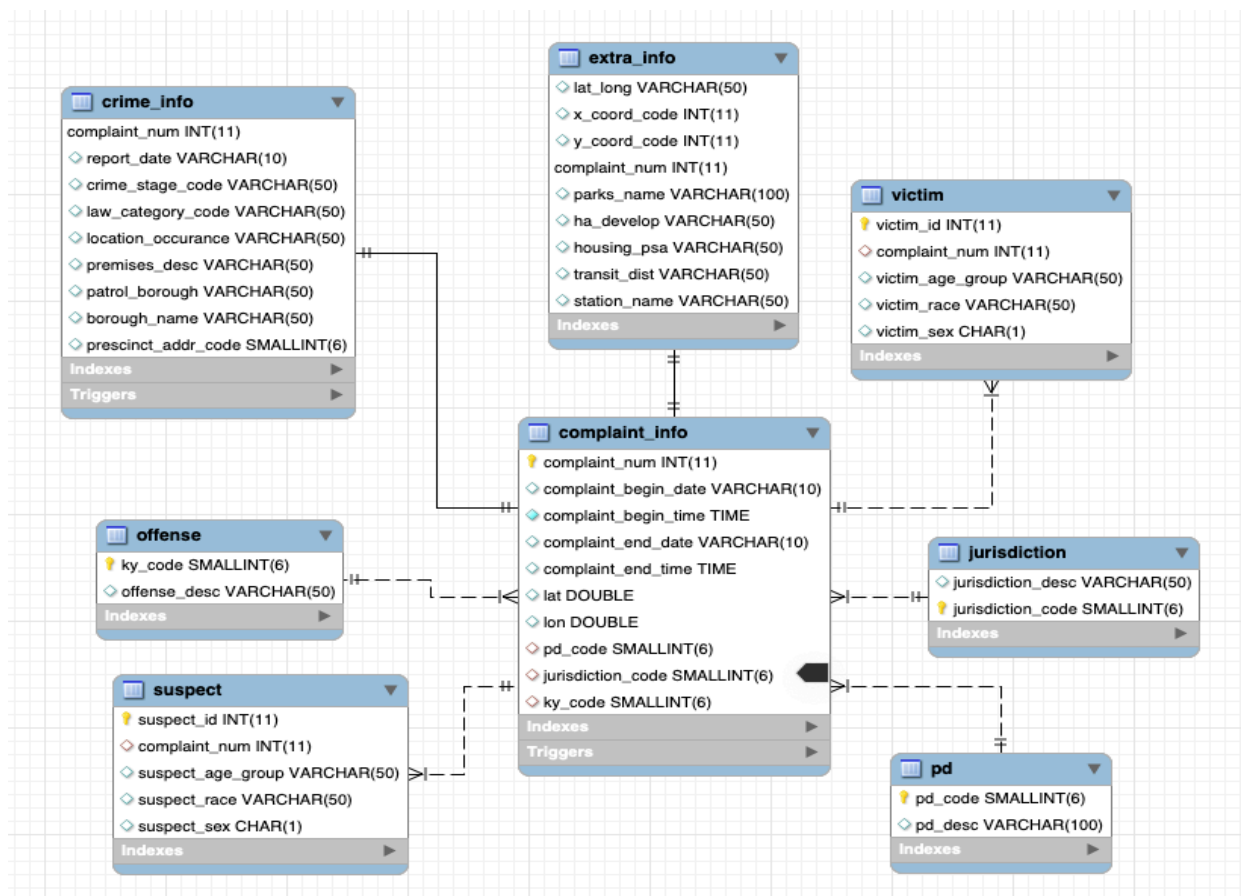
The extra_info table includes columns that could not be deciphered without a data dictionary, or whose purpose is not useful for this application. The dirty_data table consists of duplicate rows and rows with invalid primary keys. In order to preserve data integrity, we established a one-to-one relationship here between this table and the central complaint info table.

For the description of data used for testing we approached this in two separate ways. The first was to break the table into a much smaller subset to make sure that the database was acting as required i.e., 10 rows for the complaint table. Once here the next step was to test the most basic of the functionalities of the database which included a view of selected crime, total number of crimes by types, and the crimes by boroughs.

Once this section was completed and working the next step to complete was the proper insertion and deletion of data. Lastly, the ability to update data was added.

However, in order to make sure that the functionality and implementation was help to a high enough standard the ability to update the numbers columns increased, specifically, the end time and end date. The idea behind the added capability was because cops would need to be able to successfully “close” the complaint time. We also added various triggers such as making sure that the date for begin time could not be added before the current date.

UML Diagram



In order to test this data, we often used a complaint number of “1”. The reason for this was the ease of memory when checking that it was successfully entered. After correctly inserting this new row, we would then make sure it existed by using the view that was selected from the complaint information table. Next, we would delete this row and rerun the view. After we would add another row, select it, then update the row. During the update of the row, we would purposefully invoke the triggers by adding

information to columns that would call the trigger. Once we had confirmed this worked, we would review the data checking that the update and triggers actually ran. Lastly, we would redelete the data putting the database back to its original state.

Illustration of Functionality

In order to walk through the functionality of the basic database, the first thing that needs to be done is to add the “Front-end” folder to the MAMP/htdocs directory. This allows the front-end files access to the internet and creates the user interface. After completing this task, the next task is to launch the MAMP application. After launching MAMP, the tester should go to the address <http://localhost:80/application> or replace 80 with the user’s correct port. Once at the specific address the user should be able to see a general webpage called My Crime Database in blue letters. As a last side note, if the tester ran the original “project2” SQL file, all of the necessary triggers, stored procedures, etc. will have already been created.

The next tasks that will be described are the DML options as well as the views that were incorporated into the front-end application. The first DML option that will be discussed is delete, which can be found on the “Delete Crime” tab. Once there, a user is able to enter a “Complaint Number” to delete the crime instance from all tables within the database associated with that complaint number. If the deletion is successful, the page will display “Crime successfully deleted” and will return an error if the crime is not deleted.

One of the best ways to test whether or not the delete statement successfully works is to use the “Search Crime” tab. This can be done in three simple steps. The first would be to select a “Complaint Number” to use. We recommend using 100005047. After viewing the complaint number and confirming its existence in the database, we move onto the second step which is to delete the same complaint number mentioned above, 100005047. After deleting the complaint number, the user should go back to the search page and enter the same complaint number once again. If the delete command executed successfully, then an error saying “The crime does not exist” will appear. If not, the user should still be able to see the complaint number.

The next feature of the application is the “Insert Crime” tab. This feature allows the user, specifically a police officer, to insert a new crime into the database via the

application window. In order to do this, much information is needed because of data integrity constraints on the complaint info to keep good data quality. For this example, we recommended that the tester uses a “Offense Code” of 113, a “Crime Code” of 729, and a “Jurisdiction Code” of 0. To prevent failure, the “Begin Time” must be entered in the MM/DD/YYYY format with “Complaint Number” can be any number. We recommend using an easy to remember complaint number of the tester’s choice. After adding the crime, the user can navigate back to the “Search Crime” tab and enter the created complaint number to ensure the record was created.

The “Update Crime” portion of the web application allows users to enter a crime end date. When a crime is reported, there is a discrepancy between the date when the crime occurs, and the date when the complaint is actually closed and no longer an open case. The update feature allows the user to enter a “Complaint Number” and an “End Date” in the MM/DD/YYYY format. Any other format will display an error message. Also, any future date that is entered will be adjusted to the current date. The user can navigate back to the “Search Crime” tab to check if the update worked by entering the complaint number that was used in the update to see the new end date. This allows for a fully integrated walk through with the database.

The last 2 capabilities of the front-end application involve views. The “Crime by Borough” and “Crime by Type” tabs display filtered views that give generalized summaries of types of crime and where they are committed. These view tabs require no user input besides just clicking on the tab button.

Summary Discussion

The status of our application is a complete and fully functioning database system that is connected to a front-end application. The front-end application allows users to search, insert, delete, and update crime records. The most challenging part of the project was dealing with the size and quality of our data set. Our data set had almost 7 million rows so evaluating certain columns for keys and functional dependencies required the most attention in the early stages of the product. To solve these challenges, we ensured functional dependencies by cleaning up data entry errors and also subset the mega table to help reduce runtime.

In terms of status, the database would be completely ready for the NYPD to use for an alpha test run. Ultimately, the database is not perfect from a polished consumer ready project, but the project is absolutely useful and would make data management easier for the police department which was the goal. As mentioned in the introduction allowing access to the public would be the logical conclusion for this project. The next steps would be to restrict who can access certain functionalities of the front-end application, specifically only allowing the public access to the “Search Crime,” “Crime by Borough,” and “Type of Crimes” tabs.

The work done on this project was broken down fairly equally as both team members worked on both parts of the database. Both Tonnar and Ty took on equal work when dividing the database into the overall structure, third normal form, and deciding on functional dependencies. However, certain parts were focused on more by different members. Tonnar’s main priority was focusing on creating and connecting the front-end application to the back-end database and also creating advanced features in MySQL that will interact with each user-interface page.

Ty’s main responsibility was loading, cleaning the data, updating the data, and other data concerns with the SQL back-end. This is an equal split because while cleaning and loading the data was our main concern at first, PHP is a new language to the both of us so learning and comprehending that code also took some additional focus and time. In regard to the final report, we split up the 4 sections evenly with Tonnar completing the Introduction and Illustration Functionality parts, and Ty writing the Final Implementation and Summary Discussion components. In addition to this each group member read over and edited the other’s work to make sure that everything was agreed upon by the other team member as well as for quality control.