**Challenge**

# Automated Crime Data Analysis

## Background

Municipal police departments have historically kept *police blotters*, records of the location, time, and type of each crime reported in their jurisdiction. As they modernize their data systems, they often release these blotters on the Internet, allowing citizens to map and analyze crime in their cities. Police departments also use extensive analysis of their own crime data to guide deployment and strategy, directing officers to "hotspots" where crime seems to be flaring up.

Some cities are farther ahead in this process than others. The New York City Police Department, for example, led the way with its CompStat system, now adopted by many other cities. CompStat involves making weekly reports on crime levels in each of NYPD's precincts, comparing the past week to previous averages and other precincts. Police officials then determine which precincts are falling behind and need more attention and hold precinct commanders accountable for their performance.

## Task

Sophia, a recently graduated MSP, works for a police department trying to implement CompStat. The crime records office sends her weekly lists of all reported crimes, in the form of Excel spreadsheets, and she needs to process these spreadsheets, filter out unimportant crime reports (e.g. spitting or false burglar alarms), load the useful data into an SQL database, and produce a report summarizing changes in crime for each of the town's six police zones.

To complicate matters, data is sometimes updated after the fact: perhaps a victim dies in the hospital, and an assault is upgraded to a murder. Or an initial report was incorrect and needs to be corrected. So, weeks after an incident, the records office sometimes sends Sophia extra spreadsheets with lists of updated records.

Sophia's predecessor simply used Excel to handle the whole process. To compile reports for each police zone, he copied and pasted six different sets of tables and charts from an Excel spreadsheet into a Word file, occasionally introducing errors by forgetting a table or pasting it in the wrong place.

Sophia would like to automate this process. By using an SQL database, she will be able to do much more advanced analysis of crimes, and deal with a larger set of records than Excel would allow her to use. (Eventually, she hopes to analyze all 911 calls as well as crime records.) By using shell scripts and careful programming, she hopes to automate the entire process, from receipt of the data files to production of the final reports. Once her scripts are complete, she'll be able to spend her time applying to PhD programs instead of copying and pasting tables from Excel.

Your task is to write the scripts and code Sophia needs to avoid Excel purgatory.

## The Database

There are two main data files provided by the records office.

**Blotter** A row for each crime, indicating the type of crime, time and date, street address, police zone, neighborhood, and other associated metadata.

**Neighborhoods** A list of all neighborhoods, their names, and their locations. Provided as `Data/crime-data/police-neighborhoods.csv`.

Neighborhoods do not change often, so you just need to load them once and forget about them. But the blotter is updated weekly with new spreadsheets.

You must design a schema to represent each type of data, using whatever table structure you deem appropriate. Make liberal use of foreign keys, column constraints, and other SQL features to ensure the data is valid. Provide your `CREATE TABLE` statements in a `schema.sql` file as part of your submission. Using `COPY` commands or a script, load the neighborhood data into your tables. Include the commands in `schema.sql` or provide the script file in your submission.

## Input Data

Weekly police blotter data contains the following columns:

**_id** A unique integer ID for each incident.

**REPORT_NAME** `ARREST` indicates an arrest report, `OFFENSE 2.0` a report of a crime. Sometimes this is missing and should be treated as `OFFENSE 2.0`.

**SECTION** The section of the Pennsylvania penal code alleged to be violated.

**DESCRIPTION** A textual description of the type of incident, e.g. “Criminal Mischief”.

**ARREST_TIME** Date and time of the arrest or incident, in ISO 8601 format (e.g. “2015-03-10T01:30:00” for 1:30 AM on March 10, 2015).

**ADDRESS** Textual description of the location (e.g. “5000 block Forbes Ave”).

**NEIGHBORHOOD** Name of the neighborhood this address is in.

**ZONE** Number of the police zone this address is in. Police zones are how police work is split across Pittsburgh, like precincts in other cities, so it’s important for reports to be made separately for each zone.

*However,* you must be aware of potential issues in the data:

1. Some crimes occur outside Pittsburgh (e.g. a Pittsburgh resident threatened someone outside of town, and Pittsburgh police were asked to arrest the offender), so they have no zone or no neighborhood.

2. The weekly blotters are made by copying and pasting data, so sometimes data is accidentally included from previous weeks. This means there may be duplicate IDs. Duplicate data should be ignored. You should not write a query to check if a crime is already in the database, as this is wasteful; instead, you need to use exceptions or conditions to catch the PostgreSQL error indicating the ID is duplicated, and handle it appropriately.

3. Some neighborhoods in the blotter may not match the neighborhoods data file: a neighborhood may be misspelled, abbreviated, or altered (e.g. "Squirrel Hill South" instead of "Squirrel Hill"). When inserting data, your script needs to try to match these altered neighborhoods to the correct neighborhood. You should try several approaches in sequence:

   - Find neighborhoods containing a string: `SELECT * FROM neighborhoods WHERE neighborhood LIKE '%Squirrel Hill%'` will automatically find any neighborhood name containing the phrase "Squirrel Hill". This will work if the blotter neighborhood is a substring of the correct neighborhood name.
   - Find neighborhoods which are a substring of a string: `SELECT * FROM neighborhoods WHERE 'Squirrel Hill South' LIKE format('%%%s%%', neighborhood)` finds any neighborhood which is a substring of "Squirrel Hill South". This will work if the blotter neighborhood *contains* the correct neighborhood name.
   - Optionally, use methods like the Levenshtein distance to find neighborhoods similar to the string in the blotter file—see the text processing example in the databases lecture notes.

   Be sure your matching scheme can also catch neighborhoods written in a different case, like "squirrel hill". If there are multiple different matching neighborhoods and it is not clear which to use, you should generate an appropriate warning message to print and save to the log file. Substitutions should also be logged so the analyst can check if any were incorrect.

A base set of data is provided as `Data/crime-data/crime-base.csv`. There are four weeks of additional data provided, labeled `Data/crime-data/crime-week-1.csv` through `Data/crime-data/crime-week-4.csv`. You ***may not*** edit the data files manually to change any entries or fix incorrect data. Your scripts must robustly handle incorrect data ***without*** special cases for specific rows.

## Ingesting data

You must provide two scripts to handle this data. The first script should accept the name of the blotter CSV file as a command-line argument and filter the data, only including `OFFENSE 2.0` reports of the types indicated in Table 1, and skipping crimes with no zone. It should print the result as a CSV file to STDOUT. The second script should read this CSV from STDIN and load the incidents into the table you defined in `schema.sql`.

| Section | Crime type |
|---|---|
| 3304 | Criminal mischief |
| 2709 | Harassment |
| 3502 | Burglary |
| 13(a)(16) | Possession of a controlled substance |
| 13(a)(30) | Possession w/ intent to deliver |
| 3701 | Robbery |
| 3921 | Theft |
| 3921(a) | Theft of movable property |
| 3934 | Theft from a motor vehicle |
| 3929 | Retail theft |
| 2701 | Simple assault |
| 2702 | Aggravated assault |
| 2501 | Homicide |

Table 1: The types of `OFFENSE 2.0` reports you should ingest into the database for later reporting.

(R users can read and write to STDIN and STDOUT with commands like `read.csv("stdin")`, which reads from standard input, or `write.csv(data)`, which writes to STDOUT by default when not given a filename.)

If the second script encounters data that is ill-formed, such as a row with no ID, an invalid zone, or other problems, it must catch the error, print an appropriate message identifying the row that is invalid, and continue its work with the next row, rather than crashing. The error messages should also be logged to a file, specifying enough diagnostic information that a user can easily read the file and identify the problem with the data.

The purpose of splitting the two scripts is to separate filtering from database manipulation, and make it possible to automate the process, for example by writing

```
Rscript filter_data.R crime-week-1.csv | Rscript ingest_data.R
```

## Patch Files

As mentioned above, sometimes old data needs to be corrected to reflect updated information or to fix errors. For some weeks, you will receive "patch files", which are CSV files in the same format as the crime blotter, but containing crimes that happened in *previous* weeks. You need to write a script to automatically update the database to replace the old incident data with the information in the patch file. Four patch files are in the `Data/crime-data` folder, from `crime-week-1-patch.csv` through `crime-week-4-patch.csv`. Each patch file updates the previous data, so `crime-week-4-patch.csv` updates data from files *preceding* `crime-week-4.csv`.

Sometimes the patch file will contain crimes that *weren't* previously in the data, perhaps because a crime was discovered long after it was committed. You should insert these into the database directly.

You should filter the patch files in the same way you filtered the original data, because patch files may refer to crimes not listed in Table 1. You can use a shell pipeline to do so, for example by writing

```
Rscript filter_data.R crime-week-1-patch.csv | Rscript patch_data.R
```

## Generating Reports

After new data is loaded, the police department would like to make simple reports of crime trends.

Write a script which uses a SQL queries to:

- Produce a table of counts of crimes each week, split up by type of crime.
- Graph the total number of crimes per day over the past month.
- List the change in number of crimes between this week and last week, split up both by neighborhood and by police zone (in separate tables).

These calculations should be done primarily in SQL, and *not* by loading all the data into your scripts and post-processing it there.

Your script should automatically use the last week of data in the database, so it can be run after each week's data is received to generate the latest reports. It should output a report in an easy-to-read format, such as an HTML file with tables and figures or a nicely formatted text file.

For R users, R Markdown (with the knitr package) is a good choice here. Python users can simply print out the report to a text file (and may find the tabulate package useful to create plain-text tables).

The report should be easily generated from the command line. Knitr users can knit an Rmd file using R code like

```
library(knitr)
knit('report.Rmd')
```

Include a driver script so the report can be easily generated with a command like `Rscript report.R` or `./make-report.sh`.

### Example

The overall process, from ingestion to report generation, should be as simple as a few commands:

```
Rscript filter_data.R crime-week-1-patch.csv | Rscript patch_data.R
Rscript filter_data.R crime-week-1.csv | Rscript ingest_data.R
Rscript report.R
```

or the equivalent for your programming language. (Notice that patching goes first, since the patch files correct *previous* data.) If you are ambitious, write a shell script which can run all three commands, e.g. by running `./ingest.sh crime-week-1.csv crime-week-1-patch.csv`.

## Questions

Briefly answer the following questions in comments on your GitHub pull request.

1. What is the benefit of writing a set of scripts to be used from the command line, instead of a single large script? How easy was it to load multiple weeks of data and generate new reports?

2. Clear all crime data from your database table and use your scripts to ingest the base data, the patches, and each week's crime data in the correct sequence. After all data is ingested, report in your pull request

   - the total number of crimes loaded in the database
   - the number of crimes skipped due to invalid data
   - the number of crimes whose data was automatically corrected (e.g. a mistyped neighborhood matched to the correct one)
   - the number of crimes whose records were patched
   - the number of crimes which were not patched because the patch data was invalid

   You may need to alter your scripts to automatically report these numbers.

## Requirements

A submission which has **Mastered** the assignment will:

☐ Provide `schema.sql` with your `CREATE TABLE` commands, along with a command or separate script to load the neighborhood data. Make sure your SQL is clear and readable and your schema is DRY.

☐ Provide a script to filter data and one to ingest it into the database. Make sure they are well-documented and tested as needed, and can ingest all data from weeks 1 through 4.

☐ Provide a `README.txt` file specifying the commands to be used to ingest data and generate a report.

☐ Include an example report for week 4 (i.e. after week 4's data has been loaded).

☐ Answer the questions stated above.

☐ Supply a test suite (formal or informal) for your functions.

Additionally, a submission which is **Sophisticated** will:

☐ Use excellent variable names, code style, organization, and comments, so the code is clear and readable throughout.

- ☐ Robustly handle malformed CSV input files, producing sensible error messages and continuing without crashing.
- ☐ Generate a particularly thorough and well-presented report.
- ☐ Supply a command-line driver that runs all your tests (e.g. `Rscript run_tests.R`) and report any errors.

You should also remember that **peer review is an essential part of this assignment.** You will be asked to review another student's submission, and another student will review yours. You will then revise your work based on their comments. You should provide a clear, comprehensive, and helpful review to your classmate.