Crimebnb

Travel Safe, Travel More, Travel Better

Tommy Drenis

Zach Duey

Sungbum Hong

Kun Hwi Ho

# Introduction

Travelers are increasingly using services like Airbnb to find short-term rentals both in the United States and abroad when traveling for both leisure and pleasure. Although Airbnb provides users with advanced search functionality for finding places to stay, one area that is lacking is that users must leverage other resources to incorporate the safety of the neighborhood where the rental is located into their decision. Our application is a proof-of-concept that fills this gap by integrating publicly-available crime and Airbnb listing data in New York City, enabling users to make smarter, safety-conscious decisions about where to stay when visiting the city.

The application allows users to identify the best listing for their needs from two angles. Users who are primarily concerned about the safety of the location of the listing can use the neighborhood path, while users who are primarily interested in features of the listing can use the listings path. Both paths ultimately allow users to drill down into the single-best listing for their stay.

Users primarily interested in safety start by navigating to the **all boroughs** landing page, which displays the top 100 crime hot spots based on the number of crimes committed at that location (red markers) alongside the top 100 listings with a minimum of 25 reviews ranked by average overall rating (blue markers). From this page, users can use the tabs at the top of the page to zoom in on each individual borough. If interested, users can then drill further down to the **single borough page** and view additional summary information about the borough such as the number of listings, the average and median price of listings, safest neighborhoods within the borough, and crime statistics since 2015. Users who need more information to make a decision can then use the **borough comparison page** to determine which borough they should focus on in their listing search. By exploring the different borough pages, users can make a decision about which borough best fits their needs. From here, they have two options. The quick option is to use the provided map to explore highly-rated listings by clicking on the listing of interest and viewing the listing detail page (discussed later). The second option is to navigate to the listing search page.

The **listing search page** allows users to identify listings within a single borough that meet their specific criteria. For example, users can filter listings based on the highest price they are willing to pay, the number of bedrooms and bathrooms, and the cleanliness and location ratings of the listings. When users submit their preferences, they are provided with two ways to explore the results. In the map view, they can see a (hopefully familiar now) map of the listings that meet their criteria. By clicking on an icon, they can then drill down further to the listing details. Alternatively, users can view a list of results that takes into account listing features as well as crimes that have been committed near the listing. This allows users a simple way to find listings that meet their budget, amenity, and safety requirements. By clicking on a particular listing, users can navigate to the listing detail page, which provides an even finer-grained view of the listing.

The **listing detail page** gives users fine-grained information designed to help them make a final decision about whether or not to book their stay. At the top of the page, users are given summary information about the listing (room type, neighborhood) that complements the criteria they used to identify the listing. It also provides a detailed breakdown of the average ratings summarized on a 1-5 star (fractions allowed) scale. Crime information is provided in two ways. Precinct-level crime information is summarized in a plot of the top 5 crimes in the precinct in which the listing is located. Nearby crimes are displayed alongside the location of the listing (light blue marker) within the map (red markers), aggregated by location. Finally, users are provided with a list of similar listings close to the current listing so that they can easily hop around to other listings without needing to navigate to other pages.

# Architecture

Our project architecture is divided into three components: data processing pipeline, application backend, and application frontend. Our data processing pipeline leverages python and SQL. The source data (described in the following section) was acquired in CSV format and stored in a [MySQL](https://www.mysql.com/) database. Airbnb data was parsed and cleaned using the [pandas](https://pandas.pydata.org/) library. Due to the size of the crime data, both [dask](https://dask.org/) and pandas were used for parsing and cleaning. We used [sqlalchemy](https://www.sqlalchemy.org/) and pandas to write records into the database. Our application backend consists of two components: a MySQL database and an API written using [node express](https://expressjs.com/) that is responsible for interacting with the database and returning results as json. Finally, our application frontend is written using the [React.js](https://reactjs.org/) framework with charts generated using [Chart.js](https://www.chartjs.org/). The frontend gathers crime and listing data using the API provided by our node-express-based backend service. Finally, we render interactive maps using [mapboxgl](https://docs.mapbox.com/mapbox-gl-js/api/), which provides a javascript interface for interacting with the [Mapbox API](https://docs.mapbox.com/api/overview/).

# Data

The New York City crime data comes from [Kaggle](https://www.kaggle.com/mrmorj/new-york-city-police-crime-data-historic). This dataset covers the time period from 2006 to 2017 and includes 35 columns and more than 6 million rows describing various aspects of reported crime incidents across the city. For each crime, demographic information is included about both the victim(s) and suspect(s). In addition, the type of crime committed is described and assigned a proper code, as well as a description of any weapons used. This data is used to render crime hot spots in map on the **all boroughs page**, is the source data for the crime statistics charts on the **single borough page**, rank boroughs based on criminal activity in the **borough comparison page**, rank lististings on the **listing search page**, and is the source data for the precinct-level crime summary on the **single listing page**.

The Airbnb New York City listings data also comes from [Kaggle](https://www.kaggle.com/samyukthamurali/airbnb-ratings-dataset?select=NY_Listings.csv&fbclid=IwAR1DBTEqKLdki9QFfuF-TBHq1aRFTvO0TQ5XEBNTkwxNP7tipqTX6TbUfdo). The data includes listings in NYC which were last reviewed by a customer between 2017 and 2020. The data includes 35 columns and 75,750 rows containing details about each listing. Listing details include the price, number of bedrooms and bathrooms, available amenities, as well as latitude/longitude coordinates and neighborhood in which the listing is located. The data also includes the number of reviews per month and a review score breakdown by accuracy, cleanliness, check-in, communication, location and value. This data is used render the listing locations in the map on the **all boroughs page**, is the source data for the listing statistics charts on the **single borough page**, rank boroughs by average listing rating on the **borough comparison page**, filter listings (and display them on the map) by amenity and rating on the **listing comparison page,** and is the primary source of information as well as the recommendation engine on the **single listing page**.

# Database

As with most Kaggle datasets, the original data was relatively well-organized, however each data source provided different data processing and normalization challenges.

Pandas was the main tool used for our data parsing process. We began by dropping unnecessary columns that are of little use to our application, made appropriate type conversions, and removed duplicates as necessary. For example, when creating a data frame to view the unique neighborhoods within New York, keeping duplicate instances of neighborhoods makes little sense.

One difficulty for the data parsing process was the amount of string parsing that had to be involved. For instance, we wanted to create a separate csv file and table for different amenities, but this required delimiting semi-colon separated strings like “Cable TV; Wireless Internet; Kitchen; Free Parking”. In another example, we wanted to combine columns representing dates (YYYY/MM/DD) with those representing time (HH:MM:SS) into a single datetime column.

Another difficulty regarding the data parsing process was fitting the data into memory. We used an external library called Dask to handle this issue. In the end, we took 2 csv files and broke them up into 16 smaller datasets.

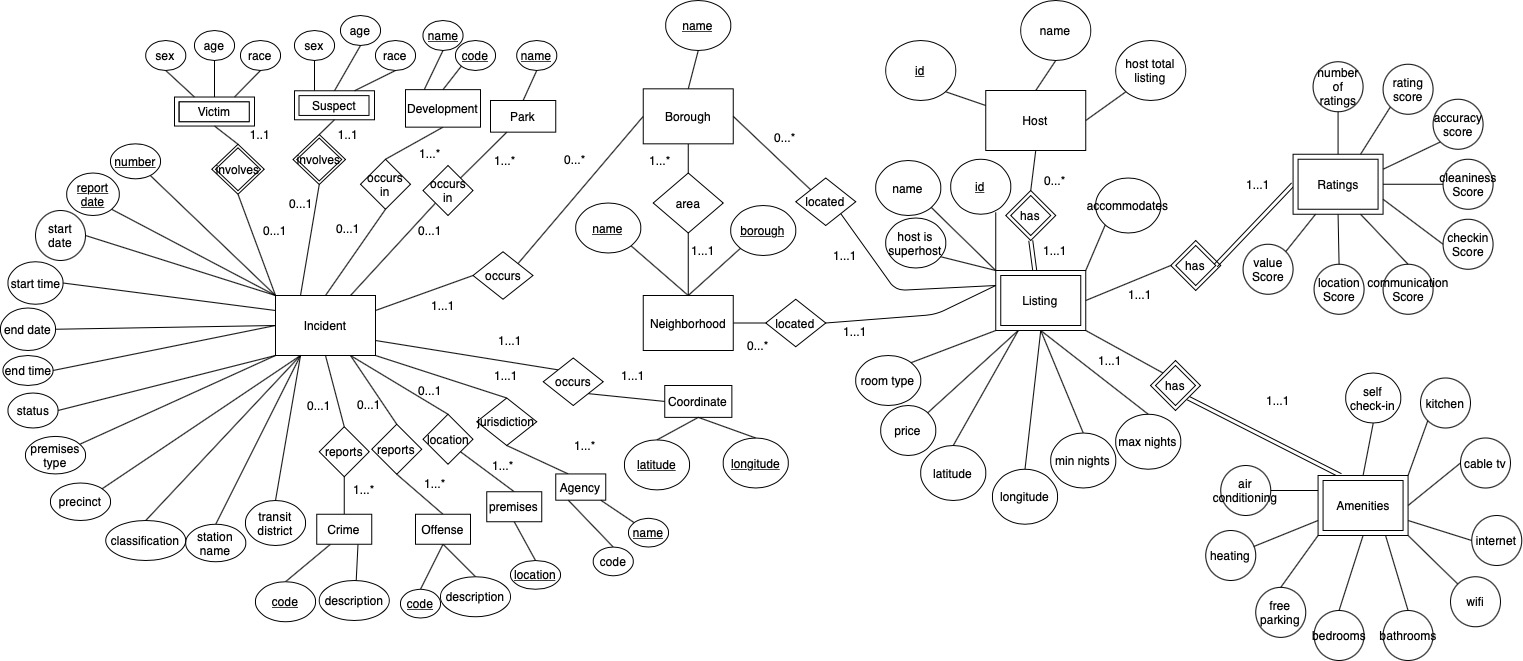
As for inserting the data into the MySQL database, the csv files are unfortunately too large to effectively use something like the MySQL Workbench “csv import” functionality. Instead, for the listings data, we used a Python script to generate SQL insertion statements. This proved to be slightly more efficient as SQL insert statements generally have a faster execution speed than csv conversions. For the crimes data, we optimized the speed even more through the use of the to\_sql() functionality in Pandas to inject the parsed data from the Python script straight into our AWS S3 instance. This was necessary as the crimes dataset proves to have more tuples than the listings dataset.

To keep consistency amongst the datasets, entity resolution was deemed necessary. The most effective case for entity resolution was for the “borough” table as this was a commonly used join key for the listings and crime data. We kept the borough names as all capital letters to ensure this consistency. Another case of entity resolution was for the latitude and longitude numbers. We decided to keep the precision of these values at 8 decimal points to ensure consistency when plotting these coordinates to our Mapbox API.

After a successful insertion to the AWS hosted database, we had the following tables and their corresponding number of rows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Table Name** | **# of Tuples** | **Table Name** | **# of Tuples** |
| agency | 22 | listing | 75,746 |
| amenities | 75,746 | neighborhood | 240 |
| borough | 5 | offense | 72 |
| coordinate | 78,302 | park | 463 |
| crime | 387 | premises | 5 |
| development | 764 | ratings | 75,746 |
| host | 56,678 | suspect | 500,000 |
| incident | 499,945 | victim | 500,000 |

The resulting ER diagram is as below, but we have also provided a link [here](https://github.com/zduey/cis550-41-project/blob/main/overview/diagram/ER_Diagram.jpg) for better readability.



We can further examine if the individual tables are a good decomposition by showing that the individual tables come from a 3NF decomposition.

Let us start by examining the original listings data and any functional dependencies that might exist. Columns that are not used for our application have been removed from the schema.

* Original Schema: Listings(listing\_id, listing\_name, host\_id, hostname, host\_total\_listing, host\_is\_superhost, borough, neighborhood, latitude, longitude, room\_type, accommodates, (all amenities…), price, min\_nights, max\_nights, number\_of\_ratings, (all rating scores…))
* Functional Dependencies = {

host\_id → hostname, host\_total\_listing;

listing\_id, host\_id → listing\_name, host\_is\_superhost, borough, neighborhood,

latitude, longitude, room\_type, accommodates, (all amenities...), price, min\_nights,

max\_nights, number\_of\_ratings, (all ratings...);

}

Before conducting the actual decomposition, there are a few key things to note. First off, it might be intuitive to think that the neighborhood name entails which borough it is in. However, this is not the case as duplicate neighborhood names exist.

It might also be easy to imagine that the listings id is functionally dependent on the host id. Again, while this is normally the case, we acknowledged that a duplicate listing id could exist if the listing id is under two different hosts. This happens when a listing is outdated, and a new host decides to reuse the same listing id.

With the given assumptions, we execute the 3NF decomposition.

Step 1) Normalize the right side of each dependency.

* Functional Dependencies = {

host\_id → hostname / host\_total\_listing;

listing\_id, host\_id → listing\_name / host\_is\_superhost / borough / neighborhood /

latitude / longitude / room\_type / accommodates / (all amenities...) / price /

min\_nights / max\_nights / number\_of\_ratings / (all ratings...);

}

Slashes here represent different dependencies (e.g. listing\_id, host\_id → listing\_name; listing\_id, host\_id → borough; …).

Step 2) Remove redundant dependencies.

In this case, none exist.

Step 3) Reduce the left side of any dependency.

In this case, “listing\_id, host\_id” cannot be reduced through Armstrong’s Axioms.

The resulting schema is:

* host(host\_id, hostname, host\_total\_listing)
* listing(host\_id, listing\_id, listing\_name, host\_is\_superhost, borough, neighborhood, latitude, longitude, room\_type, accommodates, (all amenities…), price, min\_nights, max\_nights, number\_of\_ratings, (all ratings…))

Although this is the resulting schema from the 3NF decomposition, we break up the amenities and ratings into separate tables for better readability. This results in:

* host(host\_id, hostname, host\_total\_listing)
* listing(host\_id, listing\_id, listing\_name, host\_is\_superhost, borough, neighborhood, latitude, longitude, room\_type, accommodates, price, min\_nights, max\_nights, number\_of\_ratings)
* amenities(host\_id, listing\_id, bathrooms, bedrooms, kitchen, cable\_tv...)
* ratings(host\_id, listing\_id, number\_of\_ratings, rating\_score, accuracy\_score...)

Now let us observe the original crime data and any functional dependencies that might exist. The right side of the functional dependencies here have already been normalized and are separated by slashes.

* Original Schema: Crimes(cmplnt\_num, rpt\_dt, crm\_atpt\_cptd\_cd, law\_cat\_cd, cmplnt\_fr\_dt, cmplnt\_fr\_tm, cmplnt\_to\_dt, cmplt\_to\_tm, prem\_typ\_desc, addr\_pct\_cd, patrol\_boro, station\_name, transit\_district, pd\_cd, pd\_desc, parks\_nm, hadevelopt, housing\_psa, juris\_desc, jurisdiction\_code, latitude, longitude, susp\_age\_group, susp\_race, susp\_sex, vic\_age\_group, vic\_race, vic\_sex, ky\_cd, ofns\_desc, loc\_of\_occur\_desc)
* Functional Dependencies = {

ky\_cd → ofns\_desc; pd\_cd → pd\_desc; juris\_desc → jurisdiction\_code;

cmplnt\_num → susp\_age\_group / susp\_race /susp\_sex / vic\_age\_group / vic\_race,

vic\_sex;

cmplnt\_num, rpt\_dt → crm\_atpt\_cptd\_cd / law\_cat\_cd / cmplnt\_fr\_dt /

cmplnt\_fr\_tm / cmplnt\_to\_dt / cmplt\_to\_tm / prem\_typ\_desc /addr\_pct\_cd

/patrol\_boro / station\_name / transit\_district / pd\_cd / pd\_desc / parks\_nm /

hadevelopt / housing\_psa / juris\_desc / jurisdiction\_code / latitude / longitude /

susp\_age\_group / susp\_race / susp\_sex / vic\_age\_group / vic\_race / vic\_sex /

ky\_cd / ofns\_desc / loc\_of\_occur\_desc;

}

Step 2) Remove redundant dependencies.

* Functional Dependencies = {

ky\_cd → ofns\_desc; pd\_cd → pd\_desc; juris\_desc → jurisdiction\_code;

cmplnt\_num → susp\_age\_group / susp\_race /susp\_sex / vic\_age\_group / vic\_race,

vic\_sex;

cmplnt\_num, rpt\_dt → crm\_atpt\_cptd\_cd / law\_cat\_cd / cmplnt\_fr\_dt /

cmplnt\_fr\_tm / cmplnt\_to\_dt / cmplt\_to\_tm / prem\_typ\_desc /addr\_pct\_cd

/patrol\_boro / station\_name / transit\_district / pd\_cd / parks\_nm /

hadevelopt / housing\_psa /jurisdiction\_code / latitude / longitude / ky\_cd /

loc\_of\_occur\_desc;

}

Step 3) Reduce the left side of any dependency.

In this case, none exist.

The resulting schema is:

* offense(ky\_cd, ofns\_desc)
* crime(pd\_cd, pd\_desc)
* agency(jurisdiction\_code, juris\_desc)
* parties(cmplnt\_num, susp\_age\_group, susp\_race, susp\_sex, vic\_age\_group, vic\_race, vic\_sex)
* incident(cmplnt\_num, rpt\_dt, crm\_atpt\_cptd\_cd, law\_cat\_cd, cmplnt\_fr\_dt, cmplnt\_fr\_tm, cmplnt\_to\_dt, cmplt\_to\_tm, prem\_typ\_desc, addr\_pct\_cd, patrol\_boro, station\_name, transit\_district, pd\_cd, parks\_nm, hadevelopt, housing\_psa, jurisdiction\_code, latitude, longitude, ky\_cd, loc\_of\_occur\_desc)

Although this is the resulting schema from the 3NF decomposition, we renamed most of the column names and broke up certain tables for better readability. This results in:

* offense(code, description)
* crime(code, description)
* agency(name, code)
* suspect(incident\_number, sex, age, race)
* victim(incident\_number, sex, age, race)
* premises(location)
* park(name)
* development(name, code)
* incident(number, report\_date, status, classification, start\_date, start\_time, end\_date, end\_time, premises\_type, precinct, patrol\_borough, station\_name, transit\_district, crime, offense, park, premises, development\_code, development\_name, agency, latitude, longitude)

# Queries And Performance Evaluation

For this project, the primary means of optimization our group used were the following : pushing selection and projection to bottom most layer, user smaller table as outer join, reusing temp table for duplicate query and finally left-side oriented join structure for pipelining.

Following are four complex queries that are used to evaluate the performance of our optimization. Also, note that in order to fully showcase impact of the optimization, pre-optimization time was calculated based on query that performed no selection until the very last moment, with join order with zero regards to table size and left-oriented pipeline join.

|  |  |  |
| --- | --- | --- |
| Name | Pre-Optimization (s) | Post-Optimization (s) |
| getListingCrimeSummary | 1.77 | 0.94 |
| getBoroughAllCrimeInfo | 1.96 | 1.25 |
| getBoroughBestListingInfo | 0.21 | 0.11 |
| getRatingRank | 0.42 | 0.29 |
| getListings | 1.72 | 1.13 |

**Note: code for all queries below is in appendix attached to the end of this report**

1) getListingCrimeSummary

* Purpose
  + List all crime that occurred near given listing since 2015
* Implementation
  + First extract longitude and latitude information from the listing with given listing id
  + Then, find closest precinct to the listing by using longitude, latitude and incident table
  + Finally, print out precinct, crime type and number of occurance of each crime.

2) getBoroughBestListingInfo

* Purpose
  + Find neighborhood in borough with most listing and find neighborhood in borough with highest average listings rating
* Implementation
  + First, create a listing info table which groups neighborhoods in a given borough by average rating and number of listings.
  + From there, create two tables listing by count and listing by rating which extract the top five neighborhoods with the most listing count and highest average rating.
  + Then unionize both listing by count and listing by rating table to return one table which encompass both data

3) getBoroughAllCrimeInfo

* Purpose
  + Find total number of crime for a borough since 2015 and also list top five most common crime of the borough
  + Also return yearly count of top five most common crime to use it on graph display.
* Implementation
  + Create table of crime since 2015 with code, description and year as attribute by joining crime table and incident table
  + Then, create the most\_reported\_crime table that counts all crimes by description and extract the top five crimes with the highest count.
  + Also, we created four separate tables that catalogue the number of crimes that occured in 2015, 2016, 2017 and 2018 respectively.
  + Finally, use join to merge all these tables into one so it contains total count of top five most reported crime and respective occurrence of these crime in 2015, 2016, 2017 and 2018
  + Note that we join these table on most\_reported\_crime’s description attribute so only yearly count for top 5 crime are reported.

4) getRatingRank

* Purpose
  + Rank all borough based on average rating of the listings that matches user configuration on price, number of nights, and amenities
* Implementation
  + First filter listing based on price and number of nights user want to stay since all these information is stored in listing table
  + Then join filtered listing table with amenities table and cull out listing that doesn’t match user configuration
  + Finally, calculate average of each listing by borough and rank them in descending order

5) getListings

* Purpose
  + Retrieve all listings which meet price, accommodations, bedrooms, bathrooms, location rating and cleanliness rating criteria set by user. Rank all listings by number of felonies committed in nearby vicinity of listing since 2010.
* Implementation
  + First filter listings based on price, accommodations, bedrooms, bathrooms, location rating and cleanliness rating
  + Then, join with incident table on longitude and latitude so as to find nearby crimes for each listing.
  + Group results listing name, count number of crimes and arrange listings by number of crimes in ascending order.

Based on various pre-optimization and post optimization trials that were performed, the most notable performance change usually occurred when selection and projection was pushed to the bottom most layer. The most logical explanation for this phenomenon is the copious size of our data table. In our current dataset, incident and listings tables host the bulk of our 2GB data set. Both of these queries hold at minimum 13 attributes and unfortunately they are the most frequently used table in our list of queries. Consequently, complex queries that perform no low level projection had to hold on average 10 attributes that weren’t used in selection and caused bottlenecks in I/O for join.

Compared to selection and projection scheduling, join order showcased smaller impact on decrease in time consumed for queries to finish. However, that doesn’t mean it should completely be disregarded. In all cases, join using a smaller table as outer returned slightly faster performance. Furthermore, it is likely that impact of join ordering was further diminished due to lower size difference between table after selection and projection. For example, if low level selection and projection was performed on one table while other remain untouched, the larger size difference between tables would have caused far more noticeable differences in performance. Thus, although it is true that selection and projection had larger effects on performance optimization than join ordering, we deduced that join order was still a very viable optimization technique that would work better if our data set size were more unbalanced.

Initially, we considered using indexing to further enhance performance of our queries. Most of our queries had some range search for attributes such as price, number of nights and year.

However, some unexpected error in AWS server prevented us from creating indexes. Whenever the program tried to create an index, sql server took an exorbitant amount of time and halted consecutive queries from executing. Even if a user waited for the index to be created, sql server eventually timed out. This error was most likely caused due to the sheer number of data entries in the database and limitation on AWS server. Thus, we decided to forgo indexing in our queries.

On the other hand, caching was used in some limited fashion. Although the structure of our queries were quite similar, each query either operated on a different data set or had slightly different where conditions. Thus, the only avenue in which cache was used was to generate a filler table that can be used to left join with the actual query. For example, in case neighborhoodRanking query only returns four boroughs instead of five because one borough simply didn’t match any where condition, this filler table with value 0 as stat can be joined with the result query to fill in the missing borough.

# Technical Challenges

Our group faced two primary technical challenges: integrating the interactive map into the application using Mapbox, and preprocessing the NYC crime data.

We knew from the start of the project that we would like to incorporate maps into the application, similar to how Airbnb works. However, none of us had experience working with geospatial data and interfaces like Google Maps or Mapbox. We were able to overcome this challenge by creating a proof-of-concept application that displayed a map on a web page using React.js by following a [tutorial](https://docs.mapbox.com/help/tutorials/use-mapbox-gl-js-with-react/) provided by Mapbox. This effort surfaced a second challenge, which was that the template code provided in the course included weback settings that prevented the map from rendering correctly. This necessitated starting our application largely from scratch, which was both instructive and challenging.

The second major technical challenge was interacting with the NYC crime data. The source data is a 2GB compressed CSV file. Early on, we found that we could not use pandas’ read csv functionality to load the data into memory. After some research, we determined that this is because the way that pandas parses CSV files results in far more RAM usage than what is shown on disk. This occurs for a couple of reasons. First, the uncompressed representation is obviously larger than the compressed one. Second, since CSVs are text-based, all data is initially read in as strings, which requires more memory than a more specific data type (e.g. the string “3” requires more space than the integer 3). Finally, pandas attempts to guess and convert the data types on-read, which can result in multiple copies of individual data elements existing in memory at any given time. We overcame this challenge in a few ways. To perform exploratory analysis, we switched to using the python library dask, which mirrors the pandas API, but operates over the data in batches without ever loading the full file into memory. However, we realized that this was untenable to perform the necessary data cleaning and preparation because the dask API is not as complete as pandas. To get around this issue, we decided to first convert the 2GB compressed CSV file to parquet using dask, and then use pandas for the data cleaning. Because parquet is a binary and typed file format, this resulted in a more compact representation both on disk and in memory that avoided the issues we initially experienced working with the data. The final challenge was around uploading records to the database. Generating a sql script with 6 million insert statements was untenable, so we used pandas’ to\_sql functionality. By default, this method inserts a single record at a time, waits for an ACK over the network sent from the DB, and then proceeds to the next insert. This was also untenably slow due to network I/O rather than I/O on the database side. We found a flag (multi=True), which causes the to\_sql method to batch inserts records. This approach was feasible, but still took about 12 hours to insert 2 million records. Therefore, we ultimately created a random sample of 500,000 records and only inserted those.

# Extra Credit

We think that there are 4 aspects of our project that we think can be considered extra credit:

1. Mapbox API integration to render interactive maps with crime/listing information
2. Dask for out-of-memory data processing
3. Git LFS for storing and tracking changes to outputs from our data processing pipeline
4. Docker-based runtime environment to simplify the task of deploying the application

# Appendix

## Contact Information

|  |  |  |
| --- | --- | --- |
| **Name** | **Email** | **Github Username** |
| Tommy Drenis | tdrenis@seas.upenn.edu | tdrenis |
| Zach Duey | zduey@seas.upenn.edu | zduey |
| Sungbum Hong | sungbum@seas.upenn.edu | peter0135 |
| Kun Hwi Ko | kunko@seas.upenn.edu | kunhwiko |

## Queries

### getListingCrimeSummary

with listing\_location as (

select latitude as listing\_lat, longitude as listing\_lon, round(latitude, 2) as close\_lat, round(longitude, 2) as close\_lon

from crimebnb.listing l

where listing\_id = ${listing\_id}

),

closest\_precinct as (

select precinct

from crimebnb.incident i

join listing\_location l on round(i.latitude, 2) = l.close\_lat and round(i.longitude, 2) = l.close\_lon

order by sqrt(POWER((latitude - listing\_lat), 2) + power((longitude - listing\_lon), 2)) desc

limit 1

),

precinct\_crimes as (

select i.precinct, description as crime\_type, count(crime) as n\_reported\_incidents

from crimebnb.incident i

join crimebnb.crime c on c.code = i.crime

join closest\_precinct cp on cp.precinct = i.precinct

where year(report\_date) > 2015

group by description

order by count(crime) desc

limit 5

)

select \* from precinct\_crimes ;

### getBoroughBestLisingInfo

* Query

WITH listing\_info AS (

SELECT neighborhood, COUNT(neighborhood) AS cnt, AVG(rating\_score) AS avg\_ratings

FROM (

SELECT listing\_id, neighborhood

FROM crimebnb.listing

WHERE borough = "${currBorough}"

) L

JOIN (

SELECT listing\_id, rating\_score

FROM crimebnb.ratings

WHERE rating\_score > 0

) R ON L.listing\_id = R.listing\_id

GROUP BY neighborhood

), listing\_by\_count AS (

SELECT neighborhood, cnt, avg\_ratings

FROM listing\_info

ORDER BY cnt DESC

LIMIT 5

), listing\_by\_rating AS (

SELECT neighborhood, cnt, avg\_ratings

FROM listing\_info

WHERE cnt > 10

ORDER BY avg\_ratings DESC

LIMIT 5

)

SELECT neighborhood, cnt, avg\_ratings

FROM listing\_by\_count

UNION ALL

SELECT neighborhood, cnt, avg\_ratings

FROM listing\_by\_rating;

### getBoroughAllCrimeInfo

WITH crime\_date\_from\_2015 AS (

SELECT code, description, crime\_years

FROM (

SELECT code, description

FROM crimebnb.crime

) crime\_code

JOIN (

SELECT crime, YEAR(report\_date) AS crime\_years

FROM crimebnb.incident

WHERE patrol\_borough LIKE "${currBorough}%" AND report\_date >= DATE("2015-01-01") AND report\_date < DATE("2019-01-01")

) crimes ON crime\_code.code = crimes.crime

), most\_reported\_crimes AS (

SELECT description, COUNT(description) AS cnt

FROM crime\_date\_from\_2015

GROUP BY description

ORDER BY COUNT(description) DESC

LIMIT 5

), crimes\_2015 AS (

SELECT description, COUNT(crime\_years) AS cnt\_2015

FROM crime\_date\_from\_2015

WHERE crime\_years = 2015

GROUP BY description

), crimes\_2016 AS (

SELECT description, COUNT(crime\_years) AS cnt\_2016

FROM crime\_date\_from\_2015

WHERE crime\_years = 2016

GROUP BY description

), crimes\_2017 AS (

SELECT description, COUNT(crime\_years) AS cnt\_2017

FROM crime\_date\_from\_2015

WHERE crime\_years = 2017

GROUP BY description

), crimes\_2018 AS (

SELECT description, COUNT(crime\_years) AS cnt\_2018

FROM crime\_date\_from\_2015

WHERE crime\_years = 2018

GROUP BY description

)

SELECT most\_reported\_crimes.description, cnt, cnt\_2015, cnt\_2016, cnt\_2017, cnt\_2018

FROM most\_reported\_crimes

JOIN crimes\_2015 ON most\_reported\_crimes.description = crimes\_2015.description

JOIN crimes\_2016 ON most\_reported\_crimes.description = crimes\_2016.description

JOIN crimes\_2017 ON most\_reported\_crimes.description = crimes\_2017.description

JOIN crimes\_2018 ON most\_reported\_crimes.description = crimes\_2018.description

### getRatingRank

With listingFiltered as (SELECT listing.listing\_id, listing.borough

FROM crimebnb.listing

WHERE listing.min\_nights >= "${inputMin}" AND listing.max\_nights <= "${inputMax}"

AND listing.price <= "${inputMaxPrice}" AND listing.price >= "${inputMinPrice}"),

distinctBorough as (

select DISTINCT listing.borough

FROM crimebnb.listing

WHERE listing.borough IS NOT NULL

),

amenitiesFiltered as (SELECT amenities.listing\_id, listingFiltered.borough

FROM crimebnb.amenities JOIN listingFiltered ON amenities.listing\_id = listingFiltered.listing\_id

WHERE (amenities.kitchen LIKE "${inputKitchen}" OR "${inputKitchen}" LIKE "FLAG")

AND (amenities.cable\_tv LIKE "${inputCableTV}" OR "${inputCableTV}" LIKE "FLAG")

AND (amenities.internet LIKE "${inputInternet}" OR "${inputInternet}" LIKE "FLAG")

AND (amenities.wifi LIKE "${inputWIFI}" OR "${inputWIFI}" LIKE "FLAG")

AND (amenities.heating LIKE "${inputHeating}" OR "${inputHeating}" LIKE "FLAG")

AND (amenities.air\_conditioning LIKE "${inputAC}" OR "${inputAC}" LIKE "FLAG")

AND (amenities.free\_parking LIKE "${inputParking}" OR "${inputParking}" LIKE "FLAG")

AND amenities.bathrooms >= "${inputBath}"

AND amenities.bedrooms >= "${inputBed}"),

accommodationFiltered as (SELECT amenitiesFiltered.borough as name, AVG(ratings.rating\_score) as stat

FROM amenitiesFiltered JOIN crimebnb.ratings ON amenitiesFiltered.listing\_id = ratings.listing\_id

GROUP BY amenitiesFiltered.borough)

select distinctBorough.borough AS name, COALESCE(accommodationFiltered.stat, 0) as stat

FROM distinctBorough LEFT JOIN accommodationFiltered ON distinctBorough.borough = accommodationFiltered.name

ORDER BY stat DESC;

### getListings

WITH listings AS (

SELECT l.listing\_id, l.listing\_name, l.price, l.accommodates, l.borough, l.latitude, l.longitude, l.room\_type

FROM crimebnb.listing l

WHERE l.price < '${maxPrice}' AND l.price > 0 AND l.accommodates >= '${minAccommodates}' AND l.borough = '${borough}'

),

amenities AS (

SELECT a.listing\_id, a.bedrooms, a.bathrooms

FROM crimebnb.amenities a

WHERE a.bedrooms >= '${beds}' AND a.bathrooms >= '${baths}'

),

ratings AS (

SELECT r.listing\_id, r.location\_score, r.cleanliness\_score

FROM crimebnb.ratings r

WHERE r.location\_score >= '${locscore}' AND r.cleanliness\_score >= '${cleanscore}'

),

incident AS (

SELECT i.latitude, i.longitude, i.start\_date

FROM crimebnb.incident i

),

tmp AS (

SELECT DISTINCT l.listing\_id, l.listing\_name, l.price, l.accommodates, l.borough, l.latitude, l.longitude, l.room\_type

FROM listings l JOIN amenities a ON l.listing\_id = a.listing\_id JOIN ratings r ON l.listing\_id = r.listing\_id

ORDER BY l.price ASC

)

SELECT DISTINCT listing\_id, listing\_name, price, accommodates, room\_type, borough, tmp.latitude, tmp.longitude, COUNT(\*) as num\_crimes

FROM tmp JOIN incident ON ROUND(tmp.latitude,3) = ROUND(incident.latitude, 3) AND ROUND(tmp.longitude,3) = ROUND(incident.longitude, 3)

WHERE start\_date > '2010-01-01'

GROUP BY listing\_name

ORDER BY num\_crimes ASC