team058-dbquest
Ayushe Nagpal, Reva Jethwani, Achintya Sanjay, Karan Shah

AirDnB: All-in-One Platform

Stage 1

1. **Project Summary:**

AirDnB: A one-stop portal for users seeking to explore or stay in NYC, offering detailed insights into AirBnB accommodations in neighborhoods based on safety, transportation options, and local amenities. AirDnB is a play on words where DB stands for database.

2. **Description of an application of your choice. State as clearly as possible what you want to do. What problem do you want to solve, etc.?**

We want to provide a seamless experience for those seeking to explore or stay in New York City with safety, convenience, and local amenities in mind through a web application. By integrating comprehensive data on Airbnb listings, crime statistics, Citibike and public transportation options, as well as local stores and restaurants, our platform will provide users with a one-stop portal for informed decision-making. We are addressing the challenge of users scattering to different sources to do their research by aggregating all this data.

Safety is a paramount concern for anyone staying in a new city. Our application will integrate up-to-date crime statistics by neighborhood, offering users a clear view of the safety landscape of New York City. This feature will enable users to make informed decisions about where they choose to stay, prioritizing their safety and peace of mind. To facilitate easy navigation around the city, our platform will provide comprehensive data on Citibike rental stations and public transportation options, including subway lines, bus routes, and their schedules.

3. **What would be a good creative component (technically challenging function) that can improve the functionality of your application? (What is something cool that you want to include? How are you planning to achieve it?)**

The creative component that we will be displaying on our application is a map that updates as you change how you want to filter the results. It will display a user's chosen amount of AirBnBs on a map based on what filters the user selects. For example, if the user wishes to see 25 AirBnbs in the safest neighborhoods, it will display 25 markers of different AirBnB locations that are ranked by safety.

We are planning to achieve it by using some sort of map API. Once we calculate the top AirBnBs by some metric for each filter option we choose later on, we will query a new table of these AirBnBs and graph markers on the map for each one.

4. **Usefulness. Explain as clearly as possible why your chosen application is useful. What are the basic functions of your web application? (What can users of this website do? Which simple and complex features are there?). Make sure to answer the following questions: Are there any similar websites/applications out there? If so, what are they, and how is yours different?**

The chosen application is useful because it allows potential visitors of the city to easily gauge the safety of their specific destination. New York City is vast with many boroughs, and there are no accessible applications currently available that would allow people to comprehensively understand and compare the safety of AirBnB sites. While people can cross-check crime rates with neighborhoods, it can be difficult to pinpoint a specific location (where they would actually stay), making it difficult to accurately gauge safety.

Our web application will allow users to specifically check bike routes, nearby restaurants, Citi bike locations, and crime rates by choosing their AirBnb location and applying the respective filters. Basic features include identifying AirBnBs by price and location, and more complex features would include applying different filters and putting these locations on a map. Our application ultimately addresses the need for a holistic platform that combines safety, accommodation, transportation, and local activity in one intuitive interface. Users can plan with confidence with access to up-to-date crime statistics and transportation schedules to make informed decisions.

5. **Realness. We want you to build a real application. So, make sure to locate real datasets. Describe your data sources (Where is the data from? In what format [csv, xls, txt,...], data size [cardinality and degree], what information does the data source capture?). It would be hard to satisfy stage 2 requirements with one dataset. Thus, we strongly recommend identifying at least two different data sources for your project.**

    a. AirBnBs in NYC:
        i. http://insideairbnb.com/get-the-data
        ii. Format: CSV
        iii. Cardinality: 39,720
        iv. Degree: 18
        v. AirBnB Information for NYC
    b. Crime in NYC:

i. Data Source: City of New York Website https://data.cityofnewyork.us/Public-Safety/NYPD-Complaint-Data-Historic/qgea-i56i/data_preview
ii. Format: CSV
iii. Cardinality: 8359721 entries
iv. Degree: 35

c. Restaurants in NYC:
i. Data Source: City of New York Website https://data.cityofnewyork.us/Transportation/Open-Restaurant-Applications-Historic-/pitm-atqc/data_preview
ii. Format: CSV
iii. Cardinality: 14428 entries
iv. Degree: 35

d. Subway Station Locations in NYC
i. Data Source: https://data.ny.gov/widgets/i9wp-a4ja
ii. Format: CSV:
iii. Cardinality: 1868
iv. Degree: 20

6. **A detailed description of the functionality that your website offers. This is where you talk about what the website delivers. Talk about how a user would interact with the application (i.e., things that one could create, delete, update, or search for).**

The user can interact with the website by changing what filters they would like to use when searching for an AirBnb in New York City. They can also change how many of the top AirBnBs they would like to view when seeing the top recommendations. For example, if the user wishes to see 25 AirBnbs in the safest neighborhoods (using crime database), it will display 25 markers of different AirBnB locations that are ranked by safety.

The website offers a map that will allow users to visualize the AirBnBs in relation to whatever filter they placed.

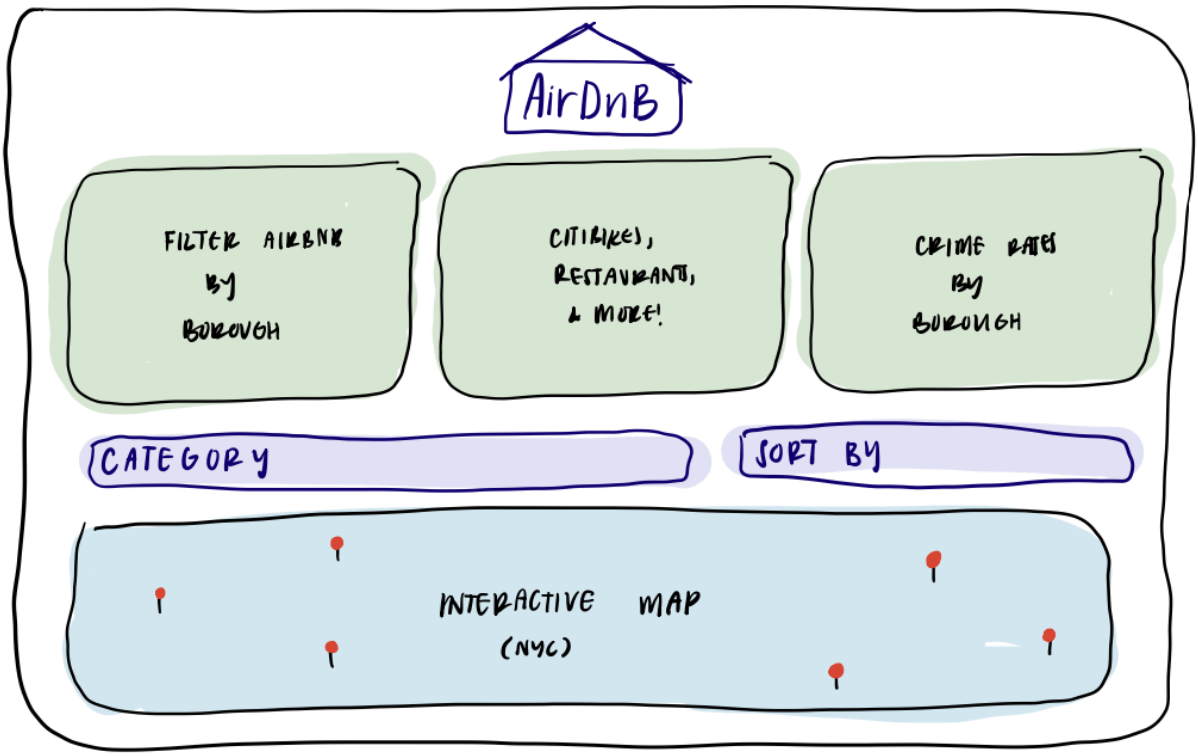
Customize Searches: Based on safety, transportation options, and proximity to amenities.
Interact with Dynamic Maps: Showcasing Airbnb listings, Citibike stations/paths, public transport options, and local businesses and update according to filters.

Data Integration:

- Airbnb Listings: Users can search and filter Airbnb options by price, location, amenities, and availability, making it easier to find the perfect stay.

- Crime Statistics: Updated crime data by neighborhood allows users to assess the safety of potential stays, ensuring peace of mind.
- Transportation Data: Detailed information on Citibike rental stations, subway lines, bus routes, and schedules helps users plan their travel within the city efficiently.
- Local Amenities: Information on nearby stores, restaurants, and attractions enables users to explore what each neighborhood has to offer.

a. A low-fidelity UI mockup: What do you imagine your final application's interface might look like? A PowerPoint slide or a pencil sketch on a piece of paper works!



b. Project Distribution:

i. Overall work will be shared throughout, but the parts below are what each person is responsible for

ii. Frontend:

1. Ayushe Nagpal
    a. User Interface Design
2. Reva Jethwani
    a. Dynamic Components

iii. Backend:

1. Achintya Sanjay
    a. Creating functions for filtering options
    b. Building a relational database management system
2. Karan Shah
    a. Setting up SQL Database, joining different tables, producing map output table

7. CRUD, Search, and Application Functions

Create
Functionality: Users can create new accounts/profiles.
This allows users to save their preferences, favorite listings, or previous searches for future reference. For example, they might want to save a list of favorite Airbnbs or preferred filters for quicker access.

Read
Functionality: Users can view listings, crime statistics, transportation options, and local amenities.
This is the core functionality of the application – users can read information about Airbnbs, crime statistics, transportation options, and local amenities to make informed decisions about where to stay in NYC.

Update
Functionality: Users can update their profile information or preferences.
This allows users to modify their saved preferences, such as updating their preferred filters or changing their saved favorite listings.

Delete
Functionality: Users can delete their accounts/profiles or remove saved preferences/favorite listings.
This gives users control over their data and preferences. For example, they might want to delete their account if they don't plan to use the platform or remove outdated saved preferences after traveling.

Search
Functionality: Users can search for Airbnbs based on various criteria such as price, location, amenities, safety (crime statistics), transportation options, and proximity to local amenities.
Users can search for specific types of Airbnbs that meet their criteria, such as finding accommodations in safe neighborhoods with easy access to transportation and nearby amenities.

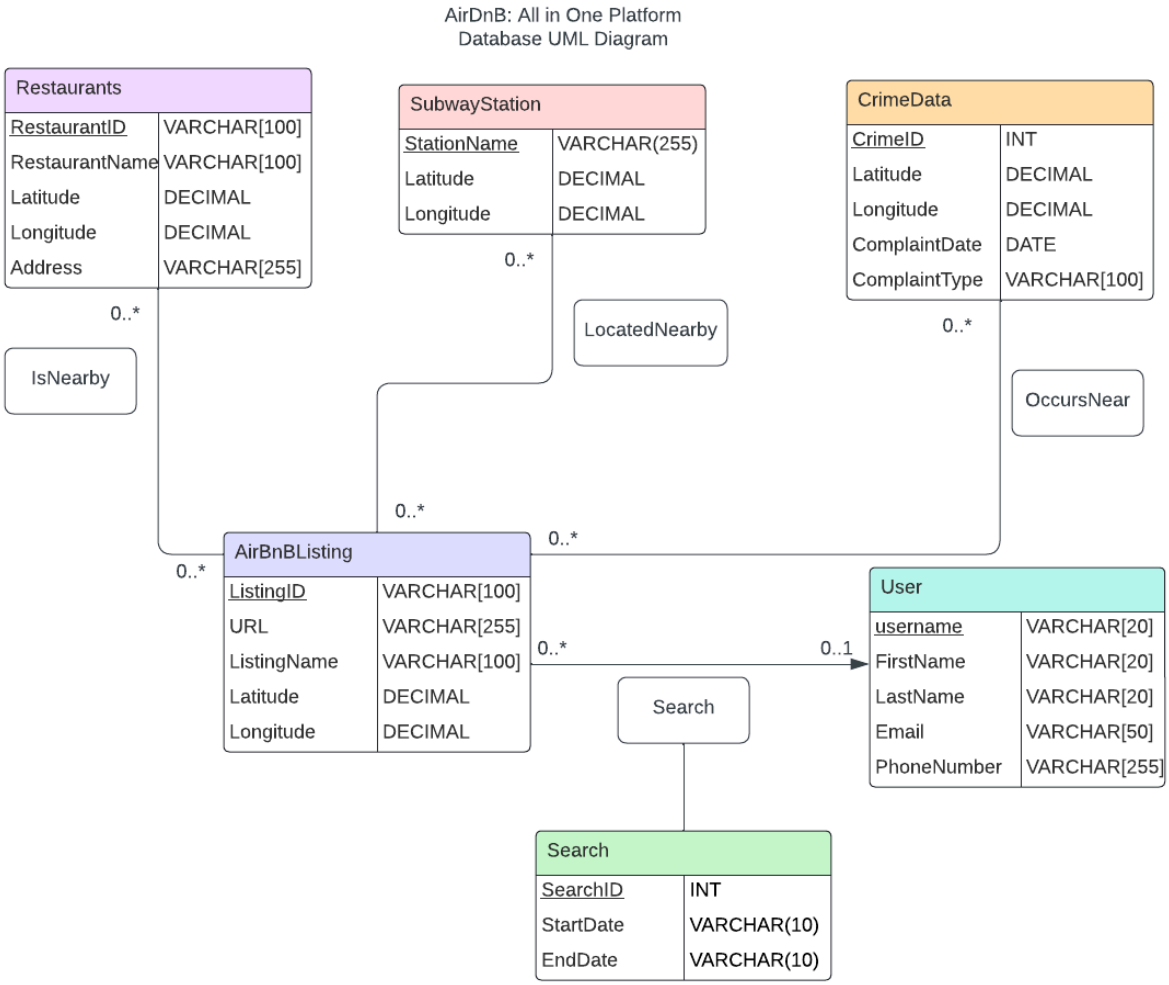
Potential Application

Functionality: Display a map that updates based on user-selected filters, showing Airbnb listings, public transport options, and local restaurants.

A map enhances the user experience by providing a visual representation of the search results, so users can see the spatial distribution of Airbnbs, transportation options, and local amenities on a map, making it easier to understand the geographical context of their search results.

Stage 2

1. UML Database Design



2. Entities and Relationships Assumptions

Entities and Their Assumptions:

- **User**: Represents individuals using the application, either as Airbnb hosts or guests. Assumed to be a central entity for personalizing the application experience, and to track user metrics.
- **AirbnbListing**: Represents the accommodations available. It's a distinct entity due to its complex attributes, including location, price, and amenities – this is the crux of our application, as AirBnB listing maps to all other entities.

- **CrimeData**: Represents crime incidents in NYC, which is essential for assessing the safety of areas surrounding Airbnb listings.
- **SubwayStation**: These represent all available subway stations. These are crucial for assessing the accessibility of Airbnb listings.
- **Restaurants**: Represents local businesses. It's an entity because each business has unique attributes (e.g., type, rating) and contributes differently to the appeal of an area – this allows users to cross reference safety with things to do in the area.

Relationships and Cardinality:
User → AirbnbListing: Modeled as a one-to-many relationship because one user can search multiple listings.

AirbnbListing → CrimeData: A many-to-many relationship, assuming that each listing is affected by multiple crime incidents based on its location. This relationship helps calculate a safety rating for each listing.

AirbnbListing → SubwayStation: A many-to-many relationship, as a listing can be near multiple stations and a station can serve multiple listings. This highlights the accessibility of the listing.

AirbnbListing → Restaurants: Also a many-to-many relationship, reflecting the proximity of multiple businesses to a listing and vice versa, affecting the listing's desirability.

3. Normalization

Functional Dependencies:

RestaurantID → RestaurantName, Latitude, Longitude, Address
StationName → Latitude, Longitude
SearchId, ListingId, Username → StartDate, EndDate
CrimeID → Latitude, Longitude, ComplaintDate, ComplaintType
ListingId → URL, Name, Latitude, Longitude
Username → FirstName, LastName, Email, PhoneNumber

Minimal Basis:

RestaurantID → RestaurantName
RestaurantID → Latitude
RestaurantID → Longitude
RestaurantID → Address
StationName → Latitude

StationName → Longitude
SearchId, ListingId, Username → StartDate
SearchId, ListingId, Username → EndDate
CrimeID → Latitude
CrimeID → Longitude
CrimeID → ComplaintDate
CrimeID → ComplaintType
ListingId → URL
ListingId → Name
ListingId → Latitude
ListingId → Longitude
Username → FirstName
Username → LastName
Username → Email
Username → PhoneNumber

3NF:

User(Username, FirstName, LastName, Email, PhoneNumber)
Restaurants(RestaurantName, Latitude, Longitude, Address)
SubwayStation(StationName, Latitude, Longitude)
CrimeData(CrimeId, Latitude, Longitude, ComplaintDate, ComplaintType)
AirBnBListing(ListingID, URL, Name, Latitude, Longitude)
SearchFunction(SearchId, ListingId, Username, StartDate, EndDate)
CandidateKey(RestaurantId, StationName, ListingId, UserName, CrimeId, SearchId)

4. **Relational Schema**

User(username: VARCHAR(20) [PK], FirstName: VARCHAR(20), LastName: VARCHAR(20), Email: VARCHAR(50), PhoneNumber: VARCHAR(255))

AirBnBListing(ListingID: VARCHAR(100) [PK], URL: VARCHAR(255), ListingName: VARCHAR(100), Latitude: DECIMAL, Longitude: DECIMAL)

CrimeData(CrimeID: INT [PK], Latitude: DECIMAL, Longitude: DECIMAL, ComplaintDate: DATE, ComplaintType: VARCHAR(100))

SubwayStation(StationName: VARCHAR(255) [PK], Latitude: DECIMAL, Longitude: DECIMAL)

Restaurants(RestaurantID: VARCHAR(100) [PK], RestaurantName: VARCHAR(100), Latitude: DECIMAL, Longitude: DECIMAL, Address: VARCHAR(255))

IsNearby(Restaurants.RestaurantID: VARCHAR(100) [FK to Restaurants.RestaurantID], AirBnBListing.ListingID: VARCHAR(100) [FK to AirBnBListing.ListingID])

LocatedNearby(SubwayStations.StationName: VARCHAR(100) [FK to

SubwayStations.StationName], AirBnBListing.ListingID: VARCHAR(100) [FK to AirBnBListing.ListingID])

OccursNear(CrimeData.CrimeID: INT [FK to CrimeData.CrimeID], AirBnBListing.ListingID: VARCHAR(100) [FK to AirBnBListing.ListingID])

Search(SearchId: INT [PK], User.Username: VARCHAR(20) [FK to User.Username], AirBnBListing.ListingID: VARCHAR(100) [FK to AirBnBListing.ListingID], StartDate: VARCHAR(10), EndDate: VARCHAR(10))

5. **Stage 1 Changes**

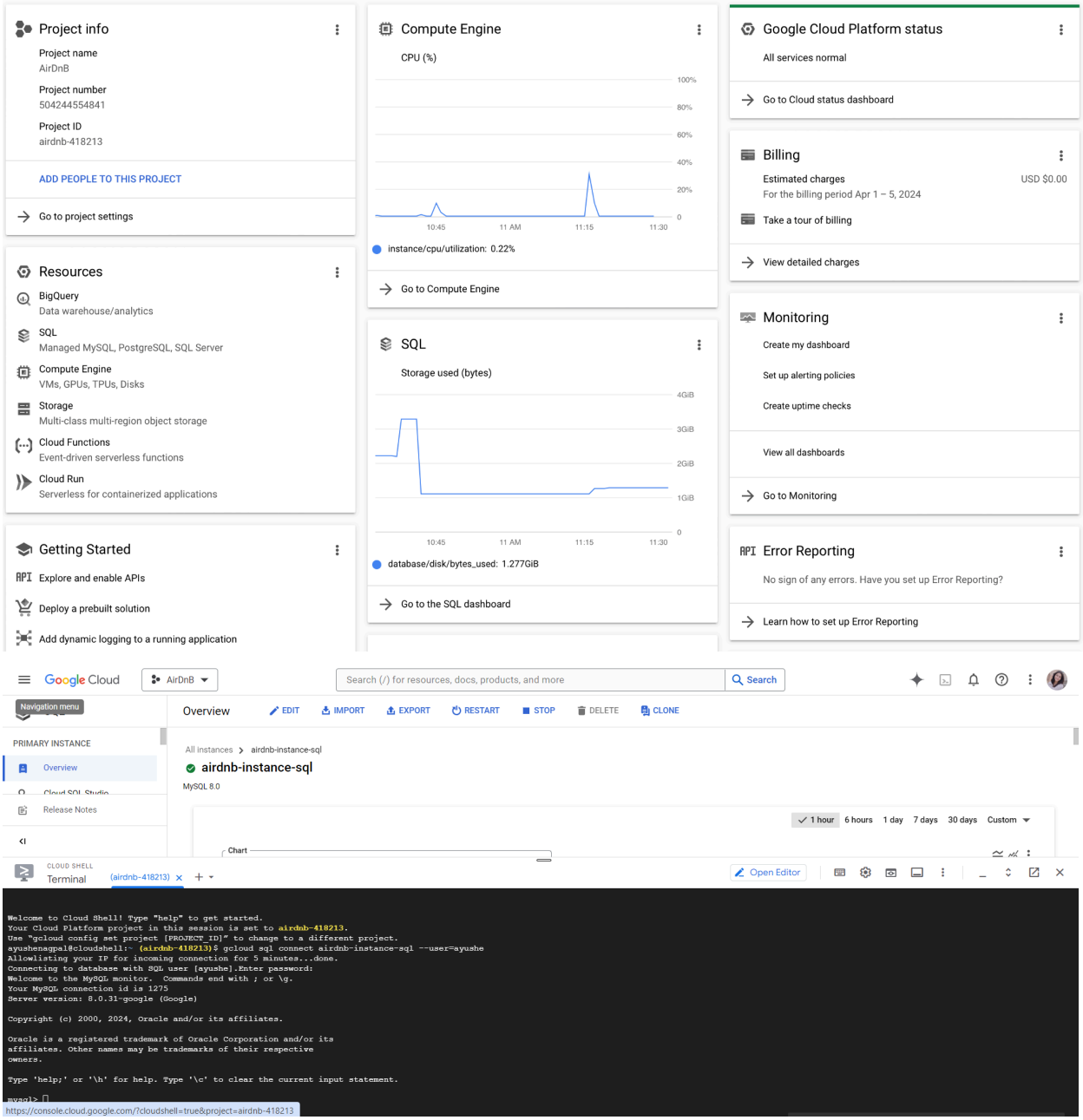
Within stage one, we implemented some changes based on the comments given. First, we wrote out “AirDnB” explicitly as “AirBnB Database Project,” as it is a play on words. We also explicitly outlined the connection between CRUD, search and our application functions, explaining how a user might apply these functions for their own personal use.

---

team058-dbquest
Ayushe Nagpal, Reva Jethwani, Achintya Sanjay, Karan Shah

Stage 3

1. GCP SetUp



2. DDL Commands & Data Insertion

- User

```
CREATE TABLE User (
   username VARCHAR(20) PRIMARY KEY,
   FirstName VARCHAR(20),
   LastName VARCHAR(20),
   Email VARCHAR(50),
   PhoneNumber VARCHAR(255)
);
```

```
mysql> DESCRIBE User;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| username    | varchar(20)  | NO   | PRI | NULL    |       |
| FirstName   | varchar(20)  | YES  |     | NULL    |       |
| LastName    | varchar(20)  | YES  |     | NULL    |       |
| Email       | varchar(50)  | YES  |     | NULL    |       |
| PhoneNumber | varchar(255) | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM User;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.01 sec)
```

The number of rows in the User table is currently 0 since it needs to be populated once we interact with the interface and create user accounts.

- AirbnbListing

```
CREATE TABLE AirBnBListing (
   ListingID VARCHAR(100) PRIMARY KEY,
   URL VARCHAR(255),
   ListingName VARCHAR(255),
   Latitude DECIMAL(9,6),
   Longitude DECIMAL(9,6)
```

);

```
mysql> DESCRIBE AirBnBListing;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| ListingID   | varchar(100) | NO   | PRI | NULL    |       |
| URL         | varchar(255) | YES  |     | NULL    |       |
| ListingName | varchar(255) | YES  |     | NULL    |       |
| Latitude    | decimal(9,6) | YES  |     | NULL    |       |
| Longitude   | decimal(9,6) | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM AirBnBListing;
+----------+
| COUNT(*) |
+----------+
|    39203 |
+----------+
1 row in set (0.01 sec)
```

SELECT COUNT(*) FROM CrimeData;

- Crime Data

CREATE TABLE CrimeData (
  CrimeID INT PRIMARY KEY,
  ComplaintDate DATE,
  ComplaintType VARCHAR(100),
  Latitude DECIMAL(9,6),
  Longitude DECIMAL(9,6)
);

```
mysql> DESCRIBE CrimeData;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| CrimeID      | int          | NO   | PRI | NULL    |       |
| Latitude     | decimal(10,0) | YES  |     | NULL    |       |
| Longitude    | decimal(10,0) | YES  |     | NULL    |       |
| ComplaintDate | date         | YES  |     | NULL    |       |
| ComplaintType | varchar(100) | YES  |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM CrimeData;
+----------+
| COUNT(*) |
+----------+
|   998861 |
+----------+
1 row in set (0.08 sec)
```

- Subway Station

```
CREATE TABLE SubwayStation (
   StationName VARCHAR(255) PRIMARY KEY,
   Latitude DECIMAL(9,6),
   Longitude DECIMAL(9,6)
);
```

```
mysql> DESCRIBE SubwayStation;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| StationName | varchar(255) | NO   | PRI | NULL    |       |
| Latitude    | decimal(10,0) | YES  |     | NULL    |       |
| Longitude   | decimal(10,0) | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM SubwayStation;
+----------+
| COUNT(*) |
+----------+
|      357 |
+----------+
1 row in set (0.00 sec)
```

- Restaurants

```
CREATE TABLE Restaurants (
        RestaurantID VARCHAR(255) PRIMARY KEY,
        RestaurantName VARCHAR(255),
        Address VARCHAR(255),
        Latitude DECIMAL(9,6),
        Longitude DECIMAL(9,6)
);
```

```
mysql> DESCRIBE Restaurants;
+----------------+--------------+------+-----+---------+-------+
| Field          | Type         | Null | Key | Default | Extra |
+----------------+--------------+------+-----+---------+-------+
| RestaurantID   | varchar(100) | NO   | PRI | NULL    |       |
| RestaurantName | varchar(100) | YES  |     | NULL    |       |
| Latitude       | decimal(10,0)| YES  |     | NULL    |       |
| Longitude      | decimal(10,0)| YES  |     | NULL    |       |
| Address        | varchar(255) | YES  |     | NULL    |       |
+----------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM Restaurants;
+----------+
| COUNT(*) |
+----------+
|    14426 |
+----------+
1 row in set (0.00 sec)
```

```
CREATE TABLE IsNearby (
   RestaurantID VARCHAR(100),
```

```
   ListingID VARCHAR(100),
   PRIMARY KEY(RestaurantID, ListingID),
   FOREIGN KEY (RestaurantID) REFERENCES Restaurants(RestaurantID),
   FOREIGN KEY (ListingID) REFERENCES AirBnBListing(ListingID)
);
```

```
mysql> DESCRIBE IsNearby
    -> ;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| RestaurantID | varchar(100) | NO   | PRI | NULL    |       |
| ListingID    | varchar(100) | NO   | PRI | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM IsNearby;
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+
1 row in set (0.00 sec)

mysql>
```

```
CREATE TABLE LocatedNearby (
   StationName VARCHAR(100),
   ListingID VARCHAR(100),
   PRIMARY KEY(StationName, ListingID),
   FOREIGN KEY (StationName) REFERENCES SubwayStation(StationName),
   FOREIGN KEY (ListingID) REFERENCES AirBnBListing(ListingID)
);

CREATE TABLE OccursNear (
   CrimeID INT,
   ListingID VARCHAR(100),
   PRIMARY KEY(CrimeID, ListingID),
   FOREIGN KEY (CrimeID) REFERENCES CrimeData(CrimeID),
   FOREIGN KEY (ListingID) REFERENCES AirBnBListing(ListingID)
);
```

```
INSERT INTO OccursNear (CrimeID, ListingID)
SELECT c.CrimeID, l.ListingID
FROM CrimeData c
JOIN AirBnBListing l ON
   (6371 * acos(
      cos(radians(c.Latitude)) * cos(radians(l.Latitude)) *
      cos(radians(l.Longitude) - radians(c.Longitude)) +
      sin(radians(c.Latitude)) * sin(radians(l.Latitude))
   )) < 1;
```

```
CREATE TABLE Search (
   SearchId INT PRIMARY KEY,
   StartDate VARCHAR(10),
   EndDate VARCHAR(10),
   Username VARCHAR(20),
   ListingID VARCHAR(100),
   FOREIGN KEY (Username) REFERENCES User(Username),
   FOREIGN KEY (ListingID) REFERENCES AirBnBListing(ListingID)
);
```

```
mysql> DESCRIBE Search;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| SearchId  | int          | NO   | PRI | NULL    |       |
| StartDate | varchar(10)  | YES  |     | NULL    |       |
| EndDate   | varchar(10)  | YES  |     | NULL    |       |
| Username  | varchar(20)  | YES  | MUL | NULL    |       |
| ListingID | varchar(100) | YES  | MUL | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

This will currently be 0 rows since this depends on User entries

3. Advanced Queries

1. SELECT

```
  b.Borough,
  COUNT(l.ListingID) AS NumberOfAirbnbListings
FROM
  (
    SELECT 'Bronx' AS Borough, 40.8082 AS MinLat, 40.9093 AS MaxLat, -73.911 AS
MinLong, -73.804 AS MaxLong
    UNION ALL
    SELECT 'Queens', 40.5915, 40.771, -73.956, -73.76286
    UNION ALL
    SELECT 'Staten Island', 40.5083, 40.6393, -74.2010, -74.0562
    UNION ALL
    SELECT 'Manhattan', 40.7059, 40.8164, -73.978, -73.928
    UNION ALL
    SELECT 'Brooklyn', 40.5876, 40.7277, -74.007, -73.8776
  ) AS b
LEFT JOIN
  AirBnBListing AS l ON (
    l.Latitude BETWEEN b.MinLat AND b.MaxLat
    AND l.Longitude BETWEEN b.MinLong AND b.MaxLong
  )
GROUP BY
  b.Borough;
```

```
mysql> SELECT
    ->     b.Borough,
    ->     COUNT(l.ListingID) AS NumberOfAirbnbListings
    -> FROM
    ->     (
    ->         SELECT 'Bronx' AS Borough, 40.8082 AS MinLat, 40.9093 AS MaxLat, -73.911 AS MinLong, -73.804 AS MaxLong
    ->         UNION ALL
    ->         SELECT 'Queens', 40.5915, 40.771, -73.956, -73.76286
    ->         UNION ALL
    ->         SELECT 'Staten Island', 40.5083, 40.6393, -74.2010, -74.0562
    ->         UNION ALL
    ->         SELECT 'Manhattan', 40.7059, 40.8164, -73.978, -73.928
    ->         UNION ALL
    ->         SELECT 'Brooklyn', 40.5876, 40.7277, -74.007, -73.8776
    ->     ) AS b
    -> LEFT JOIN
    ->     AirBnBListing AS l ON (
    ->         l.Latitude BETWEEN b.MinLat AND b.MaxLat
    ->         AND l.Longitude BETWEEN b.MinLong AND b.MaxLong
    ->     )
    -> GROUP BY
    ->     b.Borough;
+---------------+-------------------------+
| Borough       | NumberOfAirbnbListings |
+---------------+-------------------------+
| Bronx         |                    1012 |
| Queens        |                   14097 |
| Staten Island |                     311 |
| Manhattan     |                    9298 |
| Brooklyn      |                   16443 |
+---------------+-------------------------+
```

2. SELECT
b.Borough,
COUNT(s.StationName) AS NumberOfSubwayStations
FROM
(
SELECT 'Bronx' AS Borough, 40.8082 AS MinLat, 40.9093 AS MaxLat, -73.911 AS MinLong, -73.804 AS MaxLong
UNION ALL
SELECT 'Queens', 40.5915, 40.771, -73.956, -73.76286
UNION ALL
SELECT 'Staten Island', 40.5083, 40.6393, -74.2010, -74.0562
UNION ALL
SELECT 'Manhattan', 40.7059, 40.8164, -73.978, -73.928
UNION ALL
SELECT 'Brooklyn', 40.5876, 40.7277, -74.007, -73.8776
) AS b
LEFT JOIN
SubwayStation AS s ON (
s.Latitude BETWEEN b.MinLat AND b.MaxLat
AND s.Longitude BETWEEN b.MinLong AND b.MaxLong
)
GROUP BY
b.Borough;

```
mysql> SELECT
    ->     b.Borough,
    ->     COUNT(s.StationName) AS NumberOfSubwayStations
    -> FROM
    ->     (
    ->         SELECT 'Bronx' AS Borough, 40.8082 AS MinLat, 40.9093 AS MaxLat, -73.911 AS MinLong, -73.804 AS MaxLong
    ->         UNION ALL
    ->         SELECT 'Queens', 40.5915, 40.771, -73.956, -73.76286
    ->         UNION ALL
    ->         SELECT 'Staten Island', 40.5083, 40.6393, -74.2010, -74.0562
    ->         UNION ALL
    ->         SELECT 'Manhattan', 40.7059, 40.8164, -73.978, -73.928
    ->         UNION ALL
    ->         SELECT 'Brooklyn', 40.5876, 40.7277, -74.007, -73.8776
    ->     ) AS b
    -> LEFT JOIN
    ->     SubwayStation AS s ON (
    ->         s.Latitude BETWEEN b.MinLat AND b.MaxLat
    ->         AND s.Longitude BETWEEN b.MinLong AND b.MaxLong
    ->     )
    -> GROUP BY
    ->     b.Borough;
+---------------+-------------------------+
| Borough       | NumberOfSubwayStations |
+---------------+-------------------------+
| Bronx         |                      52 |
| Queens        |                     129 |
| Staten Island |                      16 |
| Manhattan     |                      44 |
| Brooklyn      |                     128 |
+---------------+-------------------------+
```

3. SELECT
b.Borough,
COUNT(r.RestaurantID) AS NumberOfRestaurants
FROM
(
SELECT 'Bronx' AS Borough, 40.8082 AS MinLat, 40.9093 AS MaxLat, -73.911 AS MinLong, -73.804 AS MaxLong
UNION ALL
SELECT 'Queens', 40.5915, 40.771, -73.956, -73.76286
UNION ALL
SELECT 'Staten Island', 40.5083, 40.6393, -74.2010, -74.0562
UNION ALL
SELECT 'Manhattan', 40.7059, 40.8164, -73.978, -73.928
UNION ALL
SELECT 'Brooklyn', 40.5876, 40.7277, -74.007, -73.8776
) AS b
LEFT JOIN
Restaurants AS r ON (
r.Latitude BETWEEN b.MinLat AND b.MaxLat
AND r.Longitude BETWEEN b.MinLong AND b.MaxLong
)
GROUP BY
b.Borough;

```
mysql> SELECT
    ->     b.Borough,
    ->     COUNT(r.RestaurantID) AS NumberOfRestaurants
    -> FROM
    ->     (
    ->         SELECT 'Bronx' AS Borough, 40.8082 AS MinLat, 40.9093 AS MaxLat, -73.911 AS MinLong, -73.804 AS MaxLong
    ->         UNION ALL
    ->         SELECT 'Queens', 40.5915, 40.771, -73.956, -73.76286
    ->         UNION ALL
    ->         SELECT 'Staten Island', 40.5083, 40.6393, -74.2010, -74.0562
    ->         UNION ALL
    ->         SELECT 'Manhattan', 40.7059, 40.8164, -73.978, -73.928
    ->         UNION ALL
    ->         SELECT 'Brooklyn', 40.5876, 40.7277, -74.007, -73.8776
    ->     ) AS b
    -> LEFT JOIN
    ->     Restaurants AS r ON (
    ->         r.Latitude BETWEEN b.MinLat AND b.MaxLat
    ->         AND r.Longitude BETWEEN b.MinLong AND b.MaxLong
    ->     )
    -> GROUP BY
    ->     b.Borough;
+---------------+---------------------+
| Borough       | NumberOfRestaurants |
+---------------+---------------------+
| Bronx         |                 550 |
| Queens        |                3306 |
| Staten Island |                 158 |
| Manhattan     |                2498 |
| Brooklyn      |                4300 |
+---------------+---------------------+
```

4. SELECT
a.ListingID,
a.ListingName,
a.Latitude,
a.Longitude,
c.CrimeID,
MAX(c.ComplaintDate) AS RecentComplaintDate,
c.ComplaintType
FROM
AirBnBListing a
LEFT JOIN
CrimeData c ON (
6371 * acos(
cos(radians(a.Latitude)) * cos(radians(c.Latitude)) * cos(radians(c.Longitude) - radians(a.Longitude)) + sin(radians(a.Latitude)) * sin(radians(c.Latitude))) <= 0. AND c.ComplaintDate >= DATE_SUB(CURRENT_DATE(), INTERVAL 1 MONTH))
GROUP BY
a.ListingID, a.ListingName, a.Latitude, a.Longitude, c.CrimeID, c.ComplaintType
ORDER BY
a.ListingID, RecentComplaintDate DESC
LIMIT 15;

```
mysql> SELECT
    ->     a.ListingID,
    ->     a.ListingName,
    ->     a.Latitude,
    ->     a.Longitude,
    ->     c.CrimeID,
    ->     MAX(c.ComplaintDate) AS RecentComplaintDate,
    ->     c.ComplaintType
    -> FROM
    ->     AirBnBListing a
    -> LEFT JOIN
    ->     CrimeData c ON (
    ->         6371 * acos(
    ->             cos(radians(a.Latitude)) * cos(radians(c.Latitude))
    ->             * cos(radians(c.Longitude) - radians(a.Longitude))
    ->             + sin(radians(a.Latitude)) * sin(radians(c.Latitude))
    ->         ) <= 0.1  -- This represents a distance of 1 kilometer
    ->         AND c.ComplaintDate >= DATE_SUB(CURRENT_DATE(), INTERVAL 1 MONTH)
    ->     )
    -> GROUP BY
    ->     a.ListingID, a.ListingName, a.Latitude, a.Longitude, c.CrimeID, c.ComplaintType
    -> ORDER BY
    ->     a.ListingID, RecentComplaintDate DESC
    -> LIMIT 15;
+--------------------+-----------------------------------------------------------------------------+----------+-----------+------------+---------------------+-----------------------------+
| ListingID          | ListingName                                                                 | Latitude | Longitude | CrimeID    | RecentComplaintDate | ComplaintType               |
+--------------------+-----------------------------------------------------------------------------+----------+-----------+------------+---------------------+-----------------------------+
| 10000070           | Rental unit in Brooklyn · ★4.65 · 1 bedroom · 1 bed · 1 shared bath         | 40.649050 | -73.969050 |       NULL | NULL                | NULL                        |
| 100002             | Rental unit in New York · ★4.69 · 1 bedroom · 2 beds · 1 shared bath        | 40.854330 | -73.932400 |       NULL | NULL                | NULL                        |
| 10000231489203763325 | Rental unit in New York · 1 bedroom · 1 bed · 1 bath                      | 40.728970 | -73.985310 |       NULL | NULL                | NULL                        |
| 10000276575127671600 | Rental unit in Brooklyn · ★2.88 · 1 bedroom · 2 beds · 1 bath             | 40.673831 | -73.938128 |       NULL | NULL                | NULL                        |
| 10000416611791818460 | Rental unit in New York · ★4.69 · 1 bedroom · 2 beds · 1 bath             | 40.771846 | -73.959741 |       NULL | NULL                | NULL                        |
| 10001022           | Rental unit in Queens · 1 bedroom · 1 bed · 1 bath                          | 40.760740 | -73.923300 |       NULL | NULL                | NULL                        |
| 10001031242444467969 | Rental unit in Brooklyn · 2 bedrooms · 2 beds · 1 bath                    | 40.730351 | -73.954892 | 284533567  | 2024-03-29          | CRIMINAL MISCHIEF & RELATED OF |
| 10001144077655282 59 | Bed and breakfast in Brooklyn · 1 bedroom · 1 bed · Half-bath             | 40.647891 | -73.950709 |       NULL | NULL                | NULL                        |
| 10001263137624180 98 | Rental unit in New York · 2 bedrooms · 3 beds · 1 bath                    | 40.719457 | -73.989828 | 284524871  | 2024-03-29          | POSSESSION OF STOLEN PROPERTY |
| 10001280453869997 00 | Rental unit in Queens · 1 bedroom · 1 bed · 1 shared bath                 | 40.740520 | -73.894240 |       NULL | NULL                | NULL                        |
| 10001401684226010 37 | Rental unit in Queens · 1 bedroom · 1 bed · 1 shared bath                 | 40.744212 | -73.901662 |       NULL | NULL                | NULL                        |
| 10001559702377329 64 | Home in Brooklyn · ★5.0 · 2 bedrooms · 3 beds · 1 bath                    | 40.670753 | -73.942801 |       NULL | NULL                | NULL                        |
| 10001593088114835 83 | Rental unit in Queens · 1 bedroom · 1 bed · 1 shared bath                 | 40.742370 | -73.901781 |       NULL | NULL                | NULL                        |
| 10001901192352654 02 | Rental unit in Queens · 1 bedroom · 1 bed · 1.5 baths                     | 40.747809 | -73.901538 |       NULL | NULL                | NULL                        |
| 10002801660407279 10 | Rental unit in Brooklyn · 1 bedroom · 1.5 shared baths                    | 40.677964 | -73.918229 |       NULL | NULL                | NULL                        |
+--------------------+-----------------------------------------------------------------------------+----------+-----------+------------+---------------------+-----------------------------+
```

4. Final Index Design

1st query

EXPLAIN ANALYZE:

```
-> Table scan on <temporary>  (actual time=102.780..102.781 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=102.774..102.774 rows=5 loops=1)
        -> Left hash join (no condition), extra conditions: (l.Latitude between b.MinLat and b.MaxLat) and (l.Longitude between b.MinLong and b.MaxLong)  (cost=20563.79 rows=195785)
(actual time=23.628..81.915 rows=41161 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.109..0.114 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.106..0.106 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Hash
                -> Table scan on l  (cost=816.83 rows=39157) (actual time=0.060..13.729 rows=39203 loops=1)
```

Before indexing,
Left join: cost = 20563.79
Lower bound = 23.628
Upper bound = 81.915

First Index:

```
mysql> CREATE INDEX idx_airbnb ON AirBnBListing (Latitude, Longitude);
Query OK, 0 rows affected, 1 warning (0.71 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
-> Table scan on <temporary>  (actual time=148.857..148.859 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=148.853..148.853 rows=5 loops=1)
        -> Nested loop left join  (cost=19750.02 rows=195785) (actual time=0.267..128.595 rows=41161 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.082..0.092 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.080..0.080 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((l.Latitude between b.MinLat and b.MaxLat) and (l.Longitude between b.MinLong and b.MaxLong))  (cost=816.83 rows=39157) (actual time=0.199..25.180 rows=8232
loops=5)
                -> Index range scan on l (re-planned for each iteration)  (cost=816.83 rows=39157) (actual time=0.059..15.426 rows=39203 loops=5)
```

This query creates a composite index named idx_airbnb_listing_latlong on the Latitude and Longitude columns of the AirBnBListing table. As shown above, this index improves the performance of the range filtering conditions and slightly lowers the cost to 19750.02.

Second Index:

```
mysql> CREATE INDEX idx1 ON AirBnBListing (Latitude);
Query OK, 0 rows affected, 1 warning (0.29 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> CREATE INDEX idx2 ON AirBnBListing (Longitude);
Query OK, 0 rows affected, 1 warning (0.28 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
| -> Table scan on <temporary>  (actual time=146.626..146.627 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=146.624..146.624 rows=5 loops=1)
        -> Nested loop left join  (cost=19750.02 rows=195785) (actual time=0.200..126.348 rows=41161 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.021..0.028 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.019..0.019 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((l.Latitude between b.MinLat and b.MaxLat) and (l.Longitude between b.MinLong and b.MaxLong))  (cost=816.83 rows=39157) (actual time=0.182..24.715 rows=8232
loops=5)
            -> Index range scan on l (re-planned for each iteration)  (cost=816.83 rows=39157) (actual time=0.039..14.917 rows=39203 loops=5)
```

These queries create two separate indices: idx_airbnb_listing_latitude on the Latitude column and idx_airbnb_listing_longitude on the Longitude column of the AirBnBListing table. If there was filtering on individual columns, this would be helpful, but since we are not the cost does not change. However, it is still valuable to analyze the efficiency of one overall index over 2 separate indices, as it allows us to analyze how different indexes would have different speeds.

Third Index:

```
mysql> CREATE INDEX listing_name ON AirBnBListing (ListingName);
Query OK, 0 rows affected, 1 warning (0.81 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
| -> Table scan on <temporary>  (actual time=166.355..166.356 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=166.345..166.345 rows=5 loops=1)
        -> Nested loop left join  (cost=19750.02 rows=195785) (actual time=0.262..143.120 rows=41161 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.038..0.047 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.034..0.034 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((l.Latitude between b.MinLat and b.MaxLat) and (l.Longitude between b.MinLong and b.MaxLong))  (cost=816.83 rows=39157) (actual time=0.225..28.039 rows=8232
loops=5)
            -> Index range scan on l (re-planned for each iteration)  (cost=816.83 rows=39157) (actual time=0.082..17.570 rows=39203 loops=5)
```

This statement creates an index named idx_airbnb_listing_name on the ListingName column of the AirBnBListing table. While the original query doesn't directly filter or sort by the ListingName column, creating an index on this column helps in cases when we have queries that only need to retrieve data from the ListingName column (which we do in our main application), and reduces the need to access the actual table data and improving query performance.

2nd query

EXPLAIN ANALYZE:

```
-> Table scan on <temporary>  (actual time=298.648..298.650 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=298.643..298.643 rows=5 loops=1)
        -> Left hash join (no condition), extra conditions: (r.Latitude between b.MinLat and b.MaxLat) and (r.Longitude between b.MinLong and b.MaxLong)  (cost=7738.62 rows=73760) (
actual time=44.501..174.201 rows=20458 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.076..0.086 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.074..0.074 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Hash
                -> Table scan on r  (cost=306.30 rows=14752) (actual time=0.066..33.157 rows=14426 loops=1)
```

Before indexing,
Left join: cost = 7738.62
Lower bound = 44.501
Upper bound = 174.201

First Index:

```
mysql> CREATE INDEX idx_subway_station ON SubwayStation (Latitude, Longitude);
Query OK, 0 rows affected, 1 warning (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
-> Table scan on <temporary>  (actual time=1.603..1.604 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=1.602..1.602 rows=5 loops=1)
        -> Nested loop left join  (cost=192.81 rows=1890) (actual time=0.102..1.403 rows=369 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.020..0.022 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.019..0.019 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((s.Latitude between b.MinLat and b.MaxLat) and (s.Longitude between b.MinLong and b.MaxLong))  (cost=7.71 rows=378) (actual time=0.055..0.267 rows=74 loops=5
)
                -> Index range scan on s (re-planned for each iteration)  (cost=7.71 rows=378) (actual time=0.033..0.163 rows=378 loops=5)
```

This creates a composite index idx_subway_station_latlong on the Latitude and Longitude columns of the SubwayStation table. This index is effective for the range filtering conditions in the JOIN clause, allowing the query optimizer to quickly locate the relevant rows within the specified latitude and longitude ranges for each borough – as you can see above, it *significantly* lowers the cost to 192.81.

Second Index:

```
mysql> ALTER TABLE SubwayStation ADD COLUMN Borough VARCHAR(20) AS (
    ->     CASE
    ->         WHEN Latitude BETWEEN 40.8082 AND 40.9093 AND Longitude BETWEEN -73.911 AND -73.804 THEN 'Bronx'
    ->         WHEN Latitude BETWEEN 40.5915 AND 40.771 AND Longitude BETWEEN -73.956 AND -73.76286 THEN 'Queens'
    ->         WHEN Latitude BETWEEN 40.5083 AND 40.6393 AND Longitude BETWEEN -74.2010 AND -74.0562 THEN 'Staten Island'
    ->         WHEN Latitude BETWEEN 40.7059 AND 40.8164 AND Longitude BETWEEN -73.978 AND -73.928 THEN 'Manhattan'
    ->         WHEN Latitude BETWEEN 40.5876 AND 40.7277 AND Longitude BETWEEN -74.007 AND -73.8776 THEN 'Brooklyn'
    ->         ELSE NULL
    ->     END
    -> );
ERROR 1060 (42S21): Duplicate column name 'Borough'
mysql>
mysql>
mysql> CREATE INDEX borough ON SubwayStation (Borough);
Query OK, 0 rows affected, 1 warning (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
| -> Table scan on <temporary>  (actual time=1.549..1.550 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=1.548..1.548 rows=5 loops=1)
        -> Nested loop left join  (cost=192.81 rows=1890) (actual time=0.098..1.358 rows=369 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.020..0.022 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.019..0.019 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((s.Latitude between b.MinLat and b.MaxLat) and (s.Longitude between b.MinLong and b.MaxLong))  (cost=7.71 rows=378) (actual time=0.048..0.261 rows=74 loops=5
)
                -> Index range scan on s (re-planned for each iteration)  (cost=7.71 rows=378) (actual time=0.030..0.171 rows=378 loops=5)
 |
```

The ALTER TABLE statement adds a new computed column named Borough to the SubwayStation table. The computed column is created, which calculates the borough based on the latitude and longitude ranges. After adding the computed column, we created a regular index named idx_subway_station_borough on the Borough column.

By creating the computed column and indexing it, the query optimizer can utilize the index to quickly locate the relevant rows in the SubwayStation table based on the calculated Borough values, without having to perform the range filtering calculations for every row.

Third Index:
CREATE INDEX subway ON SubwayStation (Longitude, Latitude);

```
| -> Table scan on <temporary>  (actual time=1.573..1.574 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=1.573..1.573 rows=5 loops=1)
        -> Nested loop left join  (cost=192.81 rows=1890) (actual time=0.111..1.365 rows=369 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.021..0.022 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.020..0.020 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((s.Latitude between b.MinLat and b.MaxLat) and (s.Longitude between b.MinLong and b.MaxLong))  (cost=7.71 rows=378) (actual time=0.051..0.262 rows=74 loops=5
)
                -> Index range scan on s (re-planned for each iteration)  (cost=7.71 rows=378) (actual time=0.032..0.165 rows=378 loops=5)
 |
```

This statement creates a composite index idx_subway_station_longlatrev on the SubwayStation table, but with the columns in reverse order (Longitude followed by Latitude). For certain

queries that first iterate over longitude, this index is more efficient than the composite index with the columns in the original order.

3rd query

EXPLAIN ANALYZE:

```
-> Table scan on <temporary>  (actual time=38.275..38.276 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=38.271..38.271 rows=5 loops=1)
        -> Left hash join (no condition), extra conditions: (r.Latitude between b.MinLat and b.MaxLat) and (r.Longitude between b.MinLong and b.MaxLong)  (cost=7738.62 rows=73760) (actual time=10.909..33.234 rows=10812 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.042..0.044 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.039..0.039 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Hash
                -> Table scan on r  (cost=306.30 rows=14752) (actual time=0.060..4.697 rows=14426 loops=1)
```

Before indexing,
Left join: cost = 7738.62
Lower bound = 10.909
Upper bound = 33.234

First Index:

```
mysql> CREATE INDEX idx_restaurants_lat_long ON Restaurants(Latitude, Longitude);
Query OK, 0 rows affected (0.26 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
-> Table scan on <temporary>  (actual time=50.920..50.922 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=50.917..50.917 rows=5 loops=1)
        -> Nested loop left join  (cost=7435.38 rows=73760) (actual time=0.116..45.584 rows=10812 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.020..0.027 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.018..0.018 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((r.Latitude between b.MinLat and b.MaxLat) and (r.Longitude between b.MinLong and b.MaxLong))  (cost=306.30 rows=14752) (actual time=0.065..8.926 rows=2162 loops=5)
                -> Index range scan on r (re-planned for each iteration)  (cost=306.30 rows=14752) (actual time=0.043..5.451 rows=14426 loops=5)
```

This composite index helps speed up the filtering of restaurants based on their geographical location. Since the advanced query filters restaurants within specific latitude and longitude ranges, this index allows the database to quickly locate restaurants within these ranges. However, the cost difference is minimal, brought down to 7435.38.

Second Index:

```
mysql> CREATE INDEX idx_restaurants_id_lat_lng ON Restaurants (RestaurantID, Latitude, Longitude);
Query OK, 0 rows affected (0.28 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Table scan on <temporary>  (actual time=52.625..52.627 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=52.623..52.623 rows=5 loops=1)
        -> Nested loop left join  (cost=7435.38 rows=73760) (actual time=0.073..47.068 rows=10812 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.018..0.023 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.017..0.017 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((r.Latitude between b.MinLat and b.MaxLat) and (r.Longitude between b.MinLong and b.MaxLong))  (cost=306.30 rows=14752) (actual time=0.
061..9.265 rows=2162 loops=5)
                -> Index range scan on r (re-planned for each iteration)  (cost=306.30 rows=14752) (actual time=0.039..5.591 rows=14426 loops=5)
 |
```

Third Index:

```
mysql> CREATE INDEX idx_restaurants_id ON Restaurants (RestaurantID);
Query OK, 0 rows affected (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Table scan on <temporary>  (actual time=50.558..50.559 rows=5 loops=1)
    -> Aggregate using temporary table  (actual time=50.556..50.556 rows=5 loops=1)
        -> Nested loop left join  (cost=7435.38 rows=73760) (actual time=0.074..45.220 rows=10812 loops=1)
            -> Table scan on b  (cost=1.01..3.06 rows=5) (actual time=0.018..0.023 rows=5 loops=1)
                -> Union all materialize  (cost=0.50..0.50 rows=5) (actual time=0.017..0.017 rows=5 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
                    -> Rows fetched before execution  (cost=0.00..0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=1)
            -> Filter: ((r.Latitude between b.MinLat and b.MaxLat) and (r.Longitude between b.MinLong and b.MaxLong))  (cost=306.30 rows=14752) (actual time=0.
055..8.898 rows=2162 loops=5)
                -> Index range scan on r (re-planned for each iteration)  (cost=306.30 rows=14752) (actual time=0.033..5.366 rows=14426 loops=5)
 |
```

4th query

EXPLAIN ANALYZE:

```
| -> Limit: 15 row(s)  (actual time=31511.973..31511.978 rows=15 loops=1)
    -> Sort: a.ListingID, RecentComplaintDate DESC, limit input to 15 row(s) per chunk  (actual time=31511.972..31511.975 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=31495.534..31502.770 rows=39203 loops=1)
            -> Aggregate using temporary table  (actual time=31495.531..31495.531 rows=39203 loops=1)
                -> Left hash join (no condition), extra conditions: ((6371 * acos((((cos(radians(a.Latitude)) * cos(radians(c.Latitude))) * cos((radians(c.Longitude) - radians(a.Lon
gitude)))) + (sin(radians(a.Latitude)) * sin(radians(c.Latitude)))))) <= 0)  (cost=236732734.74 rows=2367158121) (actual time=28.760..31252.707 rows=39203 loops=1)
                    -> Table scan on a  (cost=4083.95 rows=39157) (actual time=0.263..47.872 rows=39203 loops=1)
                    -> Hash
                        -> Filter: (c.ComplaintDate >= <cache>((curdate() - interval 1 month)))  (cost=0.59 rows=60453) (actual time=27.023..27.518 rows=830 loops=1)
                            -> Table scan on c  (cost=0.59 rows=60453) (actual time=0.066..23.476 rows=63622 loops=1)
 |
```

Before indexing,
Left join: cost = 236732734.74
Lower bound = 28.760

Upper bound = 31252.707

First Index:

```
mysql> CREATE INDEX idx_crimedata_complaintdate ON CrimeData (ComplaintDate);
Query OK, 0 rows affected (0.34 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (actual time=29967.564..29967.568 rows=15 loops=1)
    -> Sort: a.ListingID, RecentComplaintDate DESC, limit input to 15 row(s) per chunk  (actual time=29967.563..29967.566 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=29952.150..29958.754 rows=39203 loops=1)
            -> Aggregate using temporary table  (actual time=29952.147..29952.147 rows=39203 loops=1)
                -> Left hash join (no condition), extra conditions: ((6371 * acos((((cos(radians(a.Latitude)) * cos(radians(c.Latitude))) * cos((radians(c.Longitude) - radians(a.Lon
gitude)))) + (sin(radians(a.Latitude)) * sin(radians(c.Latitude))))) <= 0)  (cost=14635611.08 rows=32500310) (actual time=2.624..29788.189 rows=39203 loops=1)
                    -> Table scan on a  (cost=4083.95 rows=39157) (actual time=0.040..37.447 rows=39203 loops=1)
                    -> Hash
                        -> Filter: (c.ComplaintDate >= <cache>((curdate() - interval 1 month)))  (cost=290.76 rows=830) (actual time=0.048..1.681 rows=830 loops=1)
                            -> Index range scan on c using idx_crimedata_complaintdate over ('2024-03-30' <= ComplaintDate)  (cost=290.76 rows=830) (actual time=0.045..1.579 rows=83
0 loops=1)
```

After this index,
Left join: cost = 14635611.08
Lower bound = 2.624
Upper bound = 29788.189

Since the query filters the CrimeData table based on the ComplaintDate column (c.ComplaintDate >= DATE_SUB(CURRENT_DATE(), INTERVAL 1 MONTH)), indexing the ComplaintDate column improves the performance of this filtering operation, bringing the cost down to 14635611.08.

Second Index:

```
mysql> CREATE INDEX crimedata ON CrimeData (CrimeID, ComplaintType);
Query OK, 0 rows affected, 1 warning (0.49 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
| -> Limit: 15 row(s)  (actual time=30198.823..30198.827 rows=15 loops=1)
    -> Sort: a.ListingID, RecentComplaintDate DESC, limit input to 15 row(s) per chunk  (actual time=30198.822..30198.824 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=30183.177..30189.778 rows=39203 loops=1)
            -> Aggregate using temporary table  (actual time=30183.175..30183.175 rows=39203 loops=1)
                -> Left hash join (no condition), extra conditions: ((6371 * acos((((cos(radians(a.Latitude)) * cos(radians(c.Latitude))) * cos((radians(c.Longitude) - radians(a.Lon
gitude)))) + (sin(radians(a.Latitude)) * sin(radians(c.Latitude))))) <= 0)  (cost=14635611.08 rows=32500310) (actual time=2.158..30016.379 rows=39203 loops=1)
                    -> Table scan on a  (cost=4083.95 rows=39157) (actual time=0.033..37.747 rows=39203 loops=1)
                    -> Hash
                        -> Filter: (c.ComplaintDate >= <cache>((curdate() - interval 1 month)))  (cost=290.76 rows=830) (actual time=0.071..1.238 rows=830 loops=1)
                            -> Index range scan on c using idx_crimedata_complaintdate over ('2024-03-30' <= ComplaintDate)  (cost=290.76 rows=830) (actual time=0.067..1.153 rows=83
0 loops=1)
```

After this index, the cost did not go down because

Third Index:

```
mysql> CREATE INDEX idx_crimedata_latitude ON CrimeData (Latitude);
Query OK, 0 rows affected (0.31 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_crimedata_longitude ON CrimeData (Longitude);
Query OK, 0 rows affected (0.31 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
-> Limit: 15 row(s)  (actual time=30083.674..30083.678 rows=15 loops=1)
    -> Sort: a.ListingID, RecentComplaintDate DESC, limit input to 15 row(s) per chunk  (actual time=30083.673..30083.676 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=30067.768..30074.647 rows=39203 loops=1)
            -> Aggregate using temporary table  (actual time=30067.765..30067.765 rows=39203 loops=1)
                -> Left hash join (no condition), extra conditions: ((6371 * acos((((cos(radians(a.Latitude)) * cos(radians(c.Latitude))) * cos((radians(c.Longitude) - radians(a.Lon
gitude)))) + (sin(radians(a.Latitude)) * sin(radians(c.Latitude))))) <= 0)  (cost=14635611.08 rows=32500310) (actual time=2.224..29900.901 rows=39203 loops=1)
                    -> Table scan on a  (cost=4083.95 rows=39157) (actual time=0.062..38.740 rows=39203 loops=1)
                    -> Hash
                        -> Filter: (c.ComplaintDate >= <cache>((curdate() - interval 1 month)))  (cost=290.76 rows=830) (actual time=0.031..1.268 rows=830 loops=1)
                            -> Index range scan on c using idx_crimedata_complaintdate over ('2024-03-30' <= ComplaintDate)  (cost=290.76 rows=830) (actual time=0.027..1.172 rows=83
0 loops=1)
```

Stage 4

1. **Stage 3 Changes**

The indexing design now includes a screenshot of ANALYZE of each indexing design, by choosing the correct attributes to build indexes, with different indexing variants for each advanced query. We also changed our advanced queries so they are more relevant to our application and are practical for any user on our site.