The New York Experience

Aakash Jajoo, Karan Jaisingh, Kush Pandey and Yathushan Nadanapathan.

Abstract

This report explores the development of 'The New York Experience', a full-stack web application built on Javascript, Node, React and SQL that enables individuals wishing to visit New York to book an AirBNB that suits every need of theirs, from basic accommodation necessities to the amount of nightlife activity that they want to be surrounded by. It tackles the issue of AirBNB not taking important information related to each city into account in its search features - in the case of New York, it fails to provide fields directed towards suiting every visitor's nightlife needs. This report analyses the problem at hand, provides an overview of our solution, examines the data worked with and

I. Introduction

Airbnb is a rapidly growing company that provides users with the opportunity to rent houses/apartments on short notice. With the rapidly increasing size of the company, users today have many options for places to rent. Due to the quantity of results they are met with though, it often becomes a challenge to determine which place is the best for a specific user: certain users may look for cheaper spots, others may look for highly eloquent spots, while others may simply care about the rating of the host. The company, however, does not take into account the context of the city that it tends to - and this is a problem that our project wishes to solve.

Our website, titled 'The New York Experience', focuses on optimising the AirBNB searching process for individuals who visit New York City - also known as the home of nightlife in the United States. It enables individuals to search for AirBNBs in the city based on the kind of nightlife activity they desire - what kind of bar culture, party activity or even tranquility they wish to have during their trip to the Big Apple.

If a young group of students is looking to have a fun weekend in Manhattan, our website would suggest to them homes near areas known to be near many bars, and even in areas known to receive many noise complaints due to parties. Meanwhile, if an older couple is looking for a relaxing, tranquil stay in the city, we would suggest an AirBNB in a more quiet neighborhood.

II. System Architecture

Technology

Our application is built as a full-stack web app. It uses a variety of programming languages, libraries and technologies to achieve its appearance and functionality.

- HTML / CSS: The baseplate of the frontend utilises these two languages to provide the main content and provide several basic components.

- React: The frontend is built on top of the React framework, utilising a frontend style that mimics the Document Object Model.
- Javascript: The backend of the application utilises Javascript, in tandem with Node.js and Express to assist with middleware, in order to manage communication between client and server.
- SQL: The tables part of the database for this project are held in a SQL database that is hosted on a remote Amazon Web Service (AWS) backend server. They are accessed by establishing a remote connection to the server with a secure username and password.
- Additional Libraries:
    - Bootstrap: Simple and reusable components are imported from Bootstrap to avoid code repetition for trivial components.
    - Shard React: More advanced components not available in raw HTML or Bootstrap are implemented through Shard React.
    - React Vis: Used to implement informational diagrams and graphs.

Application Functionality

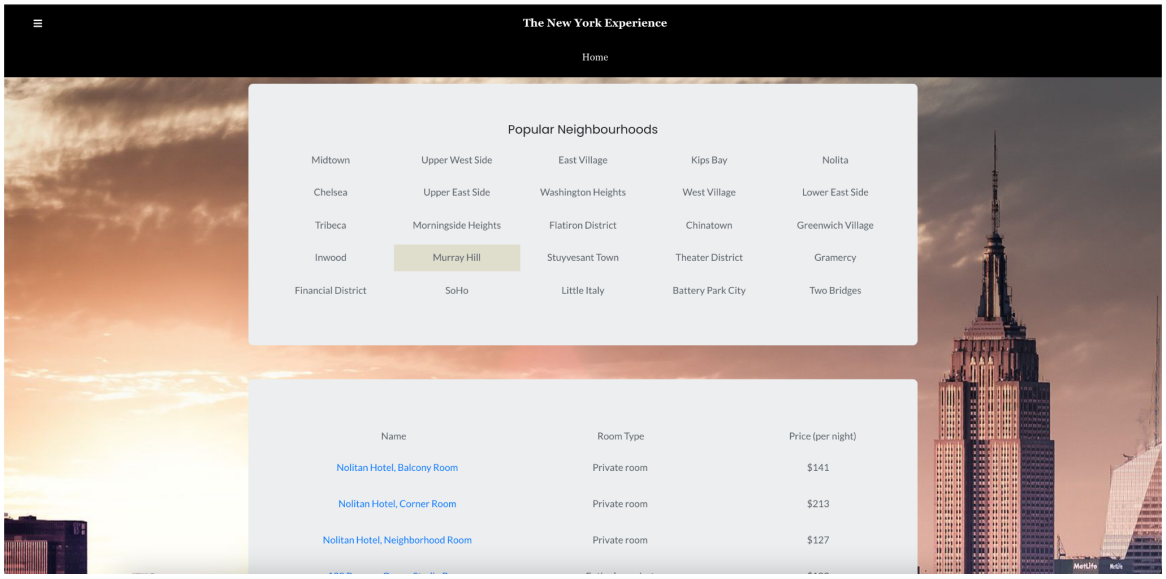
The functionality of the application is split into two primary sections - finding an AirBNB in New York based on home-related properties, and finding an AirBNB based on the nightlife that one desires.

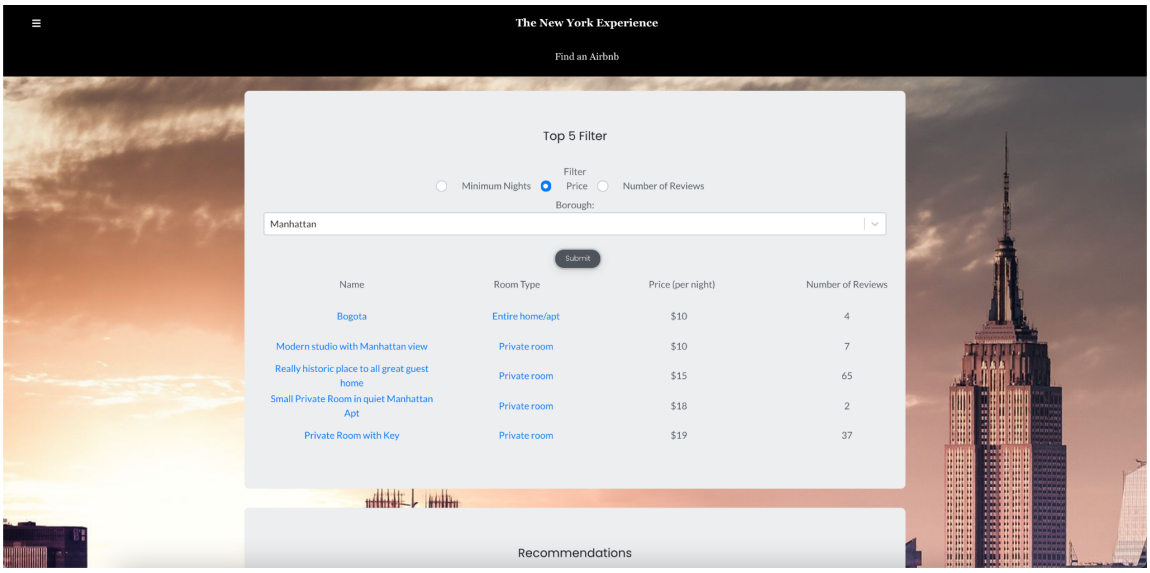
If users wish to find an AirBNB based on a series of metadata, they can do so through our application. They are allowed to search through AirBNBs based on keyword, borough of New York, type of amenities they provide as well as price point. Though this type of search feature is provided by AirBNB, our application builds on this by providing aspects such as a verification system for marked amenities associated with listings by ensuring that either the description or ratings mention their presence, given the increasingly prevalent problem of listed amenities not being truly present in properties. Furthermore, this system provides searchable listings based on several other criteria not present in AirBNB itself - including searching for top hosts in a particular based on their host reviews, as well as searching for the neighbourhoods within a borough with the largest number of available properties.

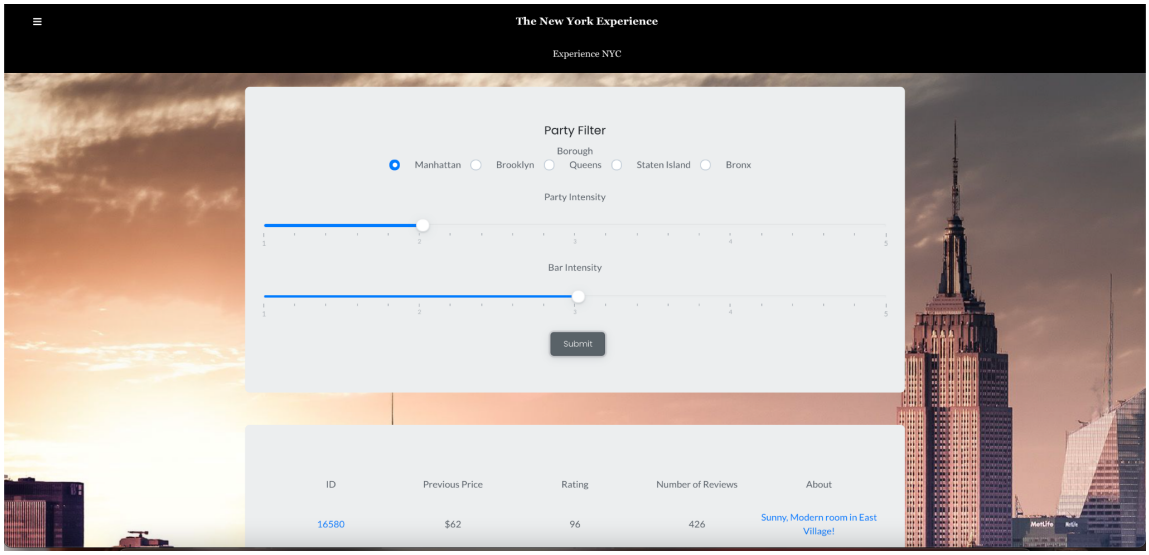
However, most users visiting our site are likely to wish to use it to determine the kind of AirBNB listing they should book based on the kind of nightlife they wish to have in the city. Enter our party and bar activity recommendation system - this uses the degree of nightlife prevalence sought after by individuals coming to New York, and returns suitable AirBNBs based on this, as well as basic factors such as the borough which is desired. It does so through a density algorithm that measures the prevalence of both bars and parties (identified by party-related noise complaints) throughout the area mapped. For each listing, important information such as nearby bars and recent reviews are further displayed in order to provide more context about the listing to a user of the website. The application further provides intuitive displays for listings known to be in party-centric locations, as well as more tranquil listings.

Application Visualisation

Though the associated video provides a more in-depth overview of the visual presentation of the web application, the main pages can be seen below.


The New York Experience
Home
Popular Neighbourhoods
Midtown
Upper West Side
East Village
Kips Bay
Nolita
Chelsea
Upper East Side
Washington Heights
West Village
Lower East Side
Tribeca
Morningside Heights
Flatiron District
Chinatown
Greenwich Village
Inwood
Murray Hill
Stuyvesant Town
Theater District
Gramercy
Financial District
SoHo
Little Italy
Battery Park City
Two Bridges
Name
Room Type
Price (per night)
Nolitan Hotel, Balcony Room
Private room
$141
Nolitan Hotel, Corner Room
Private room
$213
Nolitan Hotel, Neighborhood Room
Private room
$127


The New York Experience
Find an Airbnb
Top 5 Filter
Filter
Minimum Nights
Price
Number of Reviews
Borough:
Manhattan
Submit
Name
Room Type
Price (per night)
Number of Reviews
Bogota
Entire home/apt
$10
4
Modern studio with Manhattan view
Private room
$10
7
Really historic place to all great guest home
Private room
$15
65
Small Private Room in quiet Manhattan Apt
Private room
$18
2
Private Room with Key
Private room
$19
37
Recommendations


The New York Experience
Experience NYC
Party Filter
Borough
Manhattan
Brooklyn
Queens
Staten Island
Bronx
Party Intensity
Bar Intensity
Submit
ID
Previous Price
Rating
Number of Reviews
About
16580
$62
96
426
Sunny, Modern room in East Village!

III. Database

Data Ingestion

Data preprocessing was conducted primarily in Python. This involved using libraries such as Pandas, Numpy and Matplotlib. A pipeline was created through which each dataset used in the application was passed through, and then would be inserted directly into the AWS server.

Initially, each dataset went through a process of dealing with missing values - certain reasonably irrelevant features with many missing values were immediately discarded. After this, numerical features with few missing values had their missing values replaced with zero. Further irrelevant columns were then dropped in each table in order to save space in the table - for example, the initial listings table had over 50 columns. Finally, instances which still contained null fields were discarded as these null fields were likely not resolvable.

Following this, we removed entries in the database that had characters which could not be recognised. We also ran functions that removed impossible values like negative price and negative number of reviews. Only after this were we able to insert the table into the database.

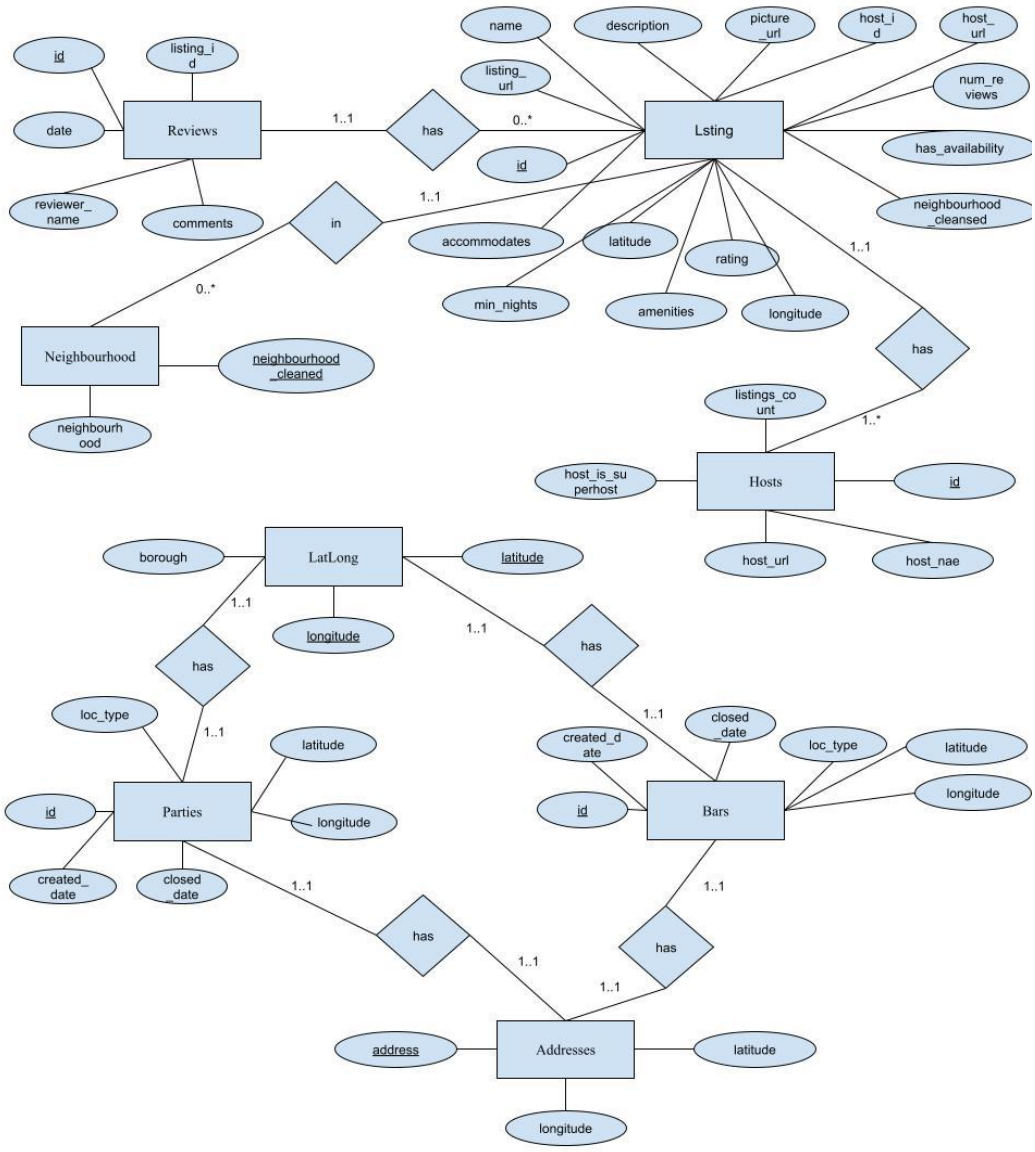
Database

Our database consisted of the following tables, some of which were simply downloaded and preprocessed while others which were self-created. Citations for all datasets derived from online sources can be found in the Bibliography.

- LatLong: This table was derived using a geolocating Python that utilised the lat-long coordinates in each of Bars and Parties to form a reference table that mapped Lat-Long coordinates to boroughs in New York.
- Bars: This table was derived from the same Kaggle source as the parties table, and contains both bar locations as well as the number of noise complaints that they have garnered in recent times, providing an indication of how active they are. Again, it is used to help assess how lively surrounding locations are for a particular listing.
- Parties: The dataset for this table was found on Kaggle, and contained details about numerous parties hosted in New York, such as the timing of them, their location and the type of party that it was. It was used as one part of determining the more lively parts of New York when finding suitable AirBNBs.
- Lsting: This dataset was derived from Inside AirBNB, and consisted of metadata for thousands of listings in the New York region. It was used as our primary table, as nearly all queries conducted returned home results from this table.
- Hosts: This dataset was derived using redundant information in the Lsting table, and aggregating results over hosts with the same ID. It was used to ensure 3NF compliance, and now contains vital information about each host.
- Reviews: This dataset was similarly derived from Inside AirBNB, and consists of ratings associated with various AirBNB properties in the New York area. This table was frequently joined with the listings table, followed by a group by, to associate reviews with listings.

- Addresses: Though the Bars and Parties tables had the geographical coordinates of each bar, they had no associated street address or even name. Hence, a new dataset was created for each instance in the given datasets, as well as the street address it corresponded to, using a Python API that generated this information from geographical data input.
- Neighbourhood: This table was also derived from Inside AirBNB, and contains mappings from neighbourhoods in New York to boroughs that they are part of. It is crucial for all joins in queries that take in a borough as input, as this allows for the borough of each listing to be obtained.

ER Diagram



Database Breakdown

Table Name	Number of Features	Number of Instances
LatLong	3	137,879
Bars	5	2441
Parties	6	225,415
Lsting	17	27,186
Hosts	5	3428
Reviews	5	846,587
Addresses	3	2441
Neighbourhood	2	68

3NF Schema

LatLong (latitude, longitude, borough);

Bars(id, location_type, latitude, longitude, num_calls)

- FOREIGN KEY(latitude, longitude) REFERENCES LatLong(latitude, longitude)
- FOREIGN KEY(latitude, longitude) REFERENCES Addresses(latitude, longitude)

Parties(id, created_date, closed_data, location_type, latitude, longitude)

- FOREIGN KEY(latitude, longitude) REFERENCES LatLong(latitude, longitude)
- FOREIGN KEY(latitude, longitude) REFERENCES Addresses(latitude, longitude)

Lsting(id, listing_url, name, picture_url, latitude, longitude, description, host_id, accommodates, amenities, price, minimum_nights, has_availability, number_of_reviews, rating, neighbourhood_cleansed, reviews_per_month);

- FOREIGN KEY host_id REFERENCES Hosts(id)
- FOREIGN neighbourhood_cleansed REFERENCES Neighbourhood(neighbourhood_cleansed)

Hosts(id, host_url, host_name, host_is_superhost, host_total_listings_count);

Reviews(id, listing_id, date, reviewer_name, comments);

- FOREIGN KEY listing_id REFERENCES Lsting(id)

Addresses(latitude, longitude, address);

- FOREIGN KEY(latitude, longitude) REFERENCES LatLong(latitude, longitude)

Neighbourhood(neighbourhood, neighbourhood_cleansed);

3NF Proof

1. Lsting:

Functional Dependencies:

- id → listing_url, name, description, picture_url, host_id, host_url, host_name, host_is_superhost, host_picture_url, host_total_listings_count, neighbourhood, latitude, longitude, accommodates, amenities, price, minimum_nights, has_availability, rating, neighbourhood_cleansed

- listing_url → id, name, description, picture_url, host_id, host_url, host_name, host_is_superhost, host_picture_url, host_total_listings_count, neighbourhood, latitude, longitude, accommodates, amenities, price, minimum_nights, has_availability, rating, neighbourhood_cleansed
- picture_url → id, name, description, listing_url, host_id, host_url, host_name, host_is_superhost, host_picture_url, host_total_listings_count, neighbourhood, latitude, longitude, accommodates, amenities, price, minimum_nights, has_availability, rating, neighbourhood_cleansed
- latitude, longitude → id, name, description, picture_url, listing_url, host_id, host_url, host_name, host_is_superhost, host_picture_url, host_total_listings_count, neighbourhood, accommodates, amenities, price, minimum_nights, has_availability, rating, neighbourhood_cleansed

Candidate Keys: {id}, {listing_url}, {picture_url}, {latitude, longitude}
Looking at each of the dependencies, we see that for each X → A, X is always a superkey of the entire table R.

- In dependency 1, id is a superkey.
- In dependency 2, listing_url is a superkey
- In dependency 3, picture_url is a superkey
- In dependency 4, latitude, longitude is a superkey

Thus, there is no 3NF violation.

2. Hosts:
Functional Dependencies:

1) id → host_url, host_is_superhost, host_name, host_total_listings_count
2) host_url → id, host_is_superhost, host_name, host_total_listings_count

Candidate Keys: {id}, {host_url}
Looking at each of the dependencies, we see that for each X → A, X is always a superkey of the entire table R.

- In dependency 1, id is a superkey of R.
- In dependency 2, host_url is a superkey of R.

Thus, there is no 3NF violation.

3. Neighbourhood:
Functional Dependencies:

- neighbourhood_cleansed → neighbourhood

Candidate Keys: {neighbourhood_cleansed}
Since neighbourhood_cleansed is a superkey of R, there is no 3NF violation.

4. Reviews:
Functional Dependencies:

- id → listing_id, date, reviewer_name, comments

Candidate Keys: {id}
Since id is a superkey of R, there is no 3NF violation.

5. Addresses:
Functional Dependencies:

- Address → latitude, longitude

Candidate Keys: {Address}
Since Address is a superkey of R

6. Bars:
Functional Dependencies:

- id → location_type, latitude, longitude, num_calls

Candidate Keys: {id}
Since id is a superkey of R, there is no 3NF violation.

7. Parties:
Functional Dependencies:

- Id → created_date, closed_date, Location_type, latitude, longitude

Candidate Keys: {id}
Since id is a superkey of R, there is no 3NF violation.

8. LatLong:
Functional Dependencies:

- Latitude, longitude → borough

Candidate Keys: {latitude, longitude}
Since {latitude, longitude} is a superkey, there is no 3NF violation.

IV. Queries

Our optimisation timings in this section were all found by observing their execution timing in the MySQL Workbench, which provides an automated timing indicator for all queries.

Query 1

Purpose: find relevant AirBNBs based on party intensity and bar intensity scores.
Runtimes:

- Pre-Optimisation: 2.417 seconds
- Post-Optimisation: 0.544 seconds

Optimisation Techniques:

- To begin with, the pre-optimized query is already greatly optimized; we needed to do optimizations to have it run - otherwise, it would time out. We created 7 CTES with the goal of reducing the size of the intermediate queries - that is, the CTEs have selections and other filtering to reduce the size of the table using in our joins. All of these optimisations are reflected in the pre-optimized query.
- Following the optimisations made above, we still needed 2.417 seconds to run the query. The cause for this was due to us joining on latitude and longitude, which is not a key in any table. To speed up the joins, we created 6 indices (B+ trees because we are joining on an inequality not a strict equality): for Bars, Parties, and Lsting there are 2 new indices: 1 on latitude and 1 on longitude. This change resulted in the reduction of runtime as seen above. Lastly, note that the use of this index sped up numerous queries made throughout our project.

Query 2

Purpose: find listings and description of the listings closest to locations with the highest amount of party and bar activity in Manhattan.
Runtimes:

- Pre-Optimisation: 12.715 seconds
- Post-Optimisation: 1.407 seconds

Optimisation Techniques:

- First, we reduced the size of our first CTE which was used in later queries. We did this by reducing the limit clause from 7 to 3. Next, we realized that we could push our selections even deeper. Specifically, at the end of our pre-optimised query, we filtered the results based on what words appeared in the description. However, in our fourth CTE (tab4), we have a correlated subquery. As with any correlated subquery, this is a very expensive operation that we need to perform. By moving our filtering of descriptions before this correlated subquery, we are reducing the number of times the correlated subquery is being performed. The restructuring of the query (by moving the selection/filtering of descriptions to an earlier spot) is the leading cause for the time reduction.
- One optimization that we wanted to perform was to create a Hash Index on Parties(latitude, longitude) because we do a group by on these attributes (strict equality so the Hash Index would outperform the B+ tree). However, our database does not support the creation of hash indexes due to it viewing it as too space extensive.

Query 3

Purpose: find listings and description of the listings closest to locations with the lowest amount of party and bar activity in Manhattan.
Runtimes:

- Pre-Optimisation: 7.625 seconds
- Post-Optimisation: 0.048 seconds

Optimisation Techniques:

- We created a Materialized View for the query since it was a static, complex query that didn't depend on any user based inputs
    - Unfortunately, MySQL does not permit Materialized Views and only allows for Virtual Views. However, Virtual Views are pointless for our case since we want the query to be precomputed for our webpage. To get around this, we created a new table (which serves as a Materialized View) to store the result of the query.
- Prior to creating the materialized view, we performed the regular optimisation techniques to reduce the runtime of the query. This includes creating CTEs where we performed numerous selections to reduce the size of intermediate tables. Since there was no need for excessive results, we also added limit clauses to the CTEs to restrain the size of intermediate tables.

Query 4

Purpose: find the AirBNB listings based on several search criteria (size, superhost, amenities, price, etc) as well as conduct amenity verification in the process.
Runtimes:

- Pre-Optimisation: 29.594 seconds.
- Post-Optimisation: 1.609 seconds.

Optimisation Techniques:

- In the original query, we were exploring the entire data repeatedly to find amenities and comments that matched what the user wanted. However, after restructuring the query, only the rows which fulfilled the first criteria, that it had one of the amenities, were searched and we checked if they had the second amenity and so on.
- The main idea behind the restructuring was to reduce the amount of data that is being processed after each subquery has been run in order to reduce the runtime.
- We used relational algebra in order to see how we could push down selections and projections to ensure that the query becomes more efficient and the number of joins used within the query decreases. The pre-optimized query had most of the projections pushed down so the optimisation was mainly focused on pushing down the selections. Following this process helped reduce the runtime of the query significantly.
- Relational Algebra Pre-Optimisation:

  $\Pi_{id,\, name,\, price,\, listingUrl,\, roomType,\, hostName,\, hostUrl,\, pictureUrl}(T3 \times T6 \times T9 \times T12)$
  - Here, T3 = $\Pi_{id,\, name,\, price,\, listingUrl,\, roomType,\, hostName,\, hostUrl,\, pictureUrl}(T1 \times T2)$
  - T1 = $\Pi_{id,\, name,\, price,\, listingUrl,\, roomType,\, hostName,\, hostUrl,\, pictureUrl}\sigma_{amenities\ LIKE\ 'A'\ \&\ description\ LIKE\ 'd'\ \&\ price <= p\ \&\ superhost = s\ \&\ accommodates >=}$
  - $Hosts\ JOIN_{Hosts.id = Lsting.hostID} Lsting$
  - T2 = $\Pi_{listingID}\sigma_{comments\ LIKE\ 'A'}$
  - Note that T6, T9, and T12 are similar to T3 and are made up of (T4, T5), (T7, T8), (T10, T11) respectively. (T4, T5), (T7, T8), (T10, T11) are similar to T1 and T2. They have not been written down to make the algebra look clean.
- Relational Algebra Post-Optimisation:
  - $\Pi_{id,\, name,\, price,\, listingUrl,\, roomType,\, hostName,\, hostUrl,\, pictureUrl}$
  - $\sigma_{amenities\ LIKE\ 'A'\ AND\ (comments\ LIKE\ 'X'\ OR\ comments\ LIKE\ 'Y'\ OR\ comments\ LIKE\ 'Z'\ OR\ comments\ LIKE\ 'W')}$
  - $(\Pi_{id,\, name,\, price,\, listingUrl,\, roomType,\, hostName,\, hostUrl,\, pictureUrl}\sigma_{amenities\ LIKE\ 'B'})$
  - $(\Pi_{id,\, name,\, price,\, listingUrl,\, roomType,\, hostName,\, hostUrl,\, pictureUrl}\sigma_{amenities\ LIKE\ 'C})$
  - $(\Pi_{id,\, name,\, price,\, listingUrl,\, roomType,\, hostName,\, hostUrl,\, pictureUrl}$
  - $\sigma_{Lsting.price <= 500\ AND\ Lsting.host\_is\_superhost = 0\ \ AND\ Lsting.accommodates >= 1\ AND\ description\ LIKE\ 'XYZ'}$
  - $((Lsting\ JOIN_{Lsting.id = reviews.ListingID} Reviews) JOIN_{Hosts.id = Lsting.HostID} Hosts)$

<u>Query 5</u>

Purpose: find the top hosts in a particular borough based on the reviews provided to accommodations listed under their name.

Runtimes:

- Pre-Optimisation: 0.123 seconds

- Post-Optimisation: 0.059 seconds

Optimisation Techniques:

- The pre-optimised query was being broken down into sub-queries, and there was a new table being created for each subquery. To get the final output, a join had to be performed.
- However, by restructuring the query and adding all the constraints in the original query, there was no join that had to be performed and as a result, the runtime decreased significantly.

V. Evaluation

Technical Challenges

Throughout the duration of this project, we faced many technical challenges - some were dealt with, some were not and some had partial-solutions found for them. To preface these technical challenges, it is important to note that besides Homework 2, no member in our group had experience with either frontend or backend web development, hence most non-SQL parts of the application did involve initial challenges for us.

There was a steep learning challenge associated for us with getting acquainted with project management tools like Github. We were initially faced with severe merge conflicts, which prevented us from making steady progress initially. As a result, we began communicating clearly to one another about which components of the project we would each modify, which involved defining our roles more clearly. It also made us learn how to work with distinct branches, as this would ensure that conflicts were seen prior to their emergence, ensuring that no destructive writes were conducted to the repository.

Another issue revolved around the size of the datasets being worked with - given that our backend was hosted on AWS, certain queries often took extremely long to execute, causing performance issues. To mitigate these, we firstly worked on optimising these queries, but other issues still persisted as certain requests were timing out. To prevent this, we noticed that we intermittently had to clear dead accesses that were still running, as this created a significant time lag.

Further Work

Though our application does tackle the initial problem that we had set out to solve, there are several extensions that could be added in the future if we as a group decide to pursue its development.

Firstly, it is clear that the application is limited in its scope, given that it only tackles the problem of finding suitable AirBNBs in New York. However, extending it to other cities in the US should be a reasonably simple process, given that similar data is available for all major cities within the datasets utilised, though in separate files. This would enable us to broaden our scope, attract more users and potentially even implement some sort of self-reviewing system for parties that would enable further data generation.

Next, personalising an application to a user is a crucial strategy to retain their long-term interest. As a result, implementing some kind of login system would be extremely beneficial - this would allow for

users to have personalised recommendations based on past searches of theirs, save listings they like and be notified about things like price discounts through direct emails. Linking this to their social media accounts such as Facebook would further allow for them to potentially see the kind of filters or listings that their friends are looking for, adding a social aspect to the application.

Lastly, the application UI could be completely redefined to mimic the look and feel of AirBNB to some extent. This would be beneficial as it is clear that users of this application are familiar with AirBNB - should our application follow the same data flow process as theirs, it would reduce the learning curve. This would involve adding visual components like interactive maps, modifying the entire colour scheme and centralising all search into a single page.

Extra Credit

- The implementation of a backend hosted on Amazon Web Services (AWS).

VI. Bibliography

"Express 4.x - API Reference." Express - Node.js Web Application Framework, expressjs.com/en/api.html.

"Getting Started – React." reactjs.org/docs/getting-started.html.

"Inside Airbnb. Adding Data to the Debate." Inside Airbnb, insideairbnb.com/get-the-data.html.

"New York, New York (NY) Profile." City-Data.com - Stats About All US Cities, www.city-data.com/city/New-York-New-York.html.

"Nightlife in New York City | NYCgo." NYCgo.com, 23 Mar. 2021, www.nycgo.com/things-to-do/nightlife.

Otto, Mark, et al. "Introduction." Bootstrap · The Most Popular HTML, CSS, and JS Library in the World, getbootstrap.com/docs/4.1/getting-started/introduction/.

"Parties in New York." Kaggle: Your Machine Learning and Data Science Community, www.kaggle.com/somesnm/partynyc.

"Shards React." DesignRevision - Free & Premium Bootstrap Templates and Themes, designrevision.com/docs/shards-react/getting-started.

VII. Appendix

Query 1: Pre-Optimisation

```
WITH tab1 AS (
    SELECT latitude, longitude, COUNT(*) as
NumParties
    FROM Parties JOIN LatLong ON
Parties.latitude = LatLong.latitude AND
Parties.longitude = LatLong.longitude
WHERE Borough LIKE 'Manhattan'
    GROUP BY latitude, longitude),
  tab2 AS
```

Query 1: Post-Optimisation

```
WITH tab1 AS (
    SELECT latitude, longitude, COUNT(*) as
NumParties
    FROM Parties JOIN LatLong ON
Parties.latitude = LatLong.latitude AND
Parties.longitude = LatLong.longitude
WHERE Borough LIKE 'Manhattan'
    GROUP BY latitude, longitude),
  tab2 AS
```

```
    (SELECT * FROM tab1
    WHERE NumParties > 0 and NumParties <=
10),
   tab3 AS
    (SELECT *
    FROM Lsting JOIN Neighbourhood on
Lsting.neighbourhood_cleaned =
Neighbourhood.neighbourhood_cleansed
    WHERE neighbourhood LIKE 'Manhattan'
ORDER BY number_of_reviews * rating),
   tab4 AS
    (SELECT id, name, listing_url, price, rating,
number_of_reviews, picture_url
    FROM tab3 JOIN tab2 ON ABS(tab3.latitude
- tab2.latitude) <= .001 AND ABS(tab3.longitude
- tab2.longitude) <= .001
    LIMIT 1000),
   tab5 AS
    (SELECT * FROM Bars JOIN LatLong ON
Bars.latitude = LatLong.latitude AND
Bars.longitude = LatLong.longitude
    WHERE Borough LIKE 'Manhattan' and
num_calls > 0 and num_calls <= 10),
   tab6 AS
    (SELECT tab3.id, name, listing_url, price,
rating, number_of_reviews, picture_url
    FROM tab3 JOIN tab5 ON ABS(tab3.latitude
- tab5.latitude) <= .001 AND ABS(tab3.longitude
- tab5.longitude) <= .001
    LIMIT 1000),
   tab7 AS
    (SELECT * FROM tab4
    UNION
    SELECT * FROM tab6)
   SELECT DISTINCT id, name, listing_url, price,
rating, number_of_reviews, picture_url FROM
tab7
   ORDER BY number_of_reviews * rating DESC
   LIMIT 2000;
```

Query 2: Pre-Optimisation

```
WITH tab1 AS (
SELECT Borough, latitude, longitude, COUNT(*)
```

```
    (SELECT * FROM tab1
    WHERE NumParties > 0 and NumParties <=
10),
   tab3 AS
    (SELECT *
    FROM Lsting JOIN Neighbourhood on
Lsting.neighbourhood_cleaned =
Neighbourhood.neighbourhood_cleansed
    WHERE neighbourhood LIKE 'Manhattan'
ORDER BY number_of_reviews * rating),
   tab4 AS
    (SELECT id, name, listing_url, price, rating,
number_of_reviews, picture_url
    FROM tab3 JOIN tab2 ON ABS(tab3.latitude
- tab2.latitude) <= .001 AND ABS(tab3.longitude
- tab2.longitude) <= .001
    LIMIT 1000),
   tab5 AS
    (SELECT * FROM Bars JOIN LatLong ON
Bars.latitude = LatLong.latitude AND
Bars.longitude = LatLong.longitude
    WHERE Borough LIKE 'Manhattan' and
num_calls > 0 and num_calls <= 10),
   tab6 AS
    (SELECT tab3.id, name, listing_url, price,
rating, number_of_reviews, picture_url
    FROM tab3 JOIN tab5 ON ABS(tab3.latitude
- tab5.latitude) <= .001 AND ABS(tab3.longitude
- tab5.longitude) <= .001
    LIMIT 1000),
   tab7 AS
    (SELECT * FROM tab4
    UNION
    SELECT * FROM tab6)
   SELECT DISTINCT id, name, listing_url, price,
rating, number_of_reviews, picture_url FROM
tab7
   ORDER BY number_of_reviews * rating DESC
   LIMIT 2000;
```

Query 2: Post-Optimisation

```
 WITH tab1 AS (
   SELECT Borough, latitude, longitude,
```

```
as NumParties
FROM Parties JOIN LatLong ON Parties.latitude
= LatLong.latitude AND Parties.longitude =
LatLong.longitude
WHERE Borough = 'MANHATTAN'
GROUP BY Borough, latitude, longitude
ORDER BY NumParties DESC
LIMIT 7),
tab2 AS
(SELECT A.listing_url, A.id, A.latitude,
A.longitude, A.description, A.rating,
A.neighbourhood
FROM Lsting as A JOIN Neighbourhood on
A.neighbourhood_cleaned =
Neighbourhood.neighbourhood_cleansed ),
tab3 AS
(SELECT A.listing_url, A.id, A.latitude,
A.longitude, A.description, A.rating,
A.neighbourhood, B.latitude as Blat, B.longitude
as Blon
FROM tab1 as B,tab2 as A),
tab4 AS
        (SELECT DISTINCT listing_url,
description, neighbourhood, (latitude -
Blat)*(latitude - Blat) + (longitude -
Blon)*(longitude - Blon) as distance
        FROM tab3
        WHERE neighbourhood = 'Manhattan'
AND EXISTS
                (SELECT *
                FROM Bars
                WHERE ABS(Bars.latitude -
tab3.latitude) <= .005 AND ABS(Bars.longitude -
tab3.longitude) <= .005)
        ORDER BY (latitude - Blat)*(latitude -
Blat) + (longitude - Blon)*(longitude - Blon)
DESC)
SELECT DISTINCT listing_url, description
FROM tab4
WHERE neighbourhood = 'Manhattan' AND
description LIKE '%Parties%' AND description
NOT LIKE '%no%' AND description NOT LIKE
'%spartan%' AND description NOT LIKE
'%soho%' AND description NOT LIKE '%queen%'
AND description NOT LIKE '%quiet%'
AND LENGTH(description) < 1000;
```

Query 3: Pre-Optimisation

```
WITH tab1 AS (
COUNT(*) as NumParties
   FROM Parties JOIN LatLong ON
Parties.latitude = LatLong.latitude AND
Parties.longitude = LatLong.longitude
   WHERE Borough = 'MANHATTAN'
   GROUP BY, Borough, latitude, longitude
   ORDER BY NumParties DESC
   LIMIT 3),
   tab2 AS
   (SELECT listing_url, id, latitude, longitude,
description, rating, neighbourhood
   FROM Lsting JOIN Neighbourhood on
Lsting.neighbourhood_cleaned =
Neighbourhood.neighbourhood_cleansed
WHERE neighbourhood LIKE 'Manhattan'),
   tab3 AS
   (SELECT A.listing_url, A.id, A.latitude,
A.longitude, A.description, A.rating,
A.neighbourhood, B.latitude as Blat, B.longitude
as Blon
   FROM tab1 as B,tab2 as A),
   tab4 AS
    (SELECT DISTINCT listing_url, description,
(latitude - Blat)*(latitude - Blat) + (longitude -
Blon)*(longitude - Blon) as distance
     FROM tab3
     WHERE neighbourhood = 'Manhattan' AND
description LIKE '%Parties%' AND description
NOT LIKE '%no%' AND description NOT LIKE
'%spartan%' AND description NOT LIKE
'%soho%' AND description NOT LIKE '%queen%'
AND description NOT LIKE '%quiet%'
    AND LENGTH(description) < 1000
    AND EXISTS
      (SELECT *
      FROM Bars
      WHERE ABS(Bars.latitude - tab3.latitude)
<= .005 AND ABS(Bars.longitude -
tab3.longitude) <= .005)
    ORDER BY (latitude - Blat)*(latitude - Blat) +
(longitude - Blon)*(longitude - Blon) DESC)
   SELECT DISTINCT listing_url, description
   FROM tab4;
```

Query 3: Post-Optimisation

```
CREATE Table peacefulBnbs AS (pre-optimisation
query)
```

```
        SELECT Borough, latitude, longitude,
COUNT(*) as NumParties
        FROM Parties
JOIN LatLong ON Parties.latitude =
LatLong.latitude AND Parties.longitude =
LatLong.longitude
        WHERE Borough = 'MANHATTAN'
        GROUP BY Borough, latitude, longitude
        ORDER BY NumParties DESC),
tab2 AS (
SELECT Bars.id, tab1.Borough, tab1.latitude,
tab1.longitude, MIN(num_calls) as numberCalls,
SUM(NumParties) as numParties,
MIN(num_calls) + SUM(NumParties) as issues
FROM Bars JOIN tab1 ON Bars.latitude =
tab1.latitude AND Bars.longitude =
tab1.longitude
GROUP BY Bars.id, tab1.Borough, tab1.latitude,
tab1.longitude
ORDER by issues ASC
LIMIT 25),
tab3 AS (
SELECT A.name, A.neighbourhood_cleansed,
A.latitude, A.longitude, A.room_type, A.price,
A.number_of_reviews, B.latitude as Blat,
B.longitude as Blong, issues
FROM tab2 as B, Lsting as A
),
tab4 AS(
SELECT name, neighbourhood, latitude,
longitude, room_type, price, number_of_reviews,
(latitude - Blat)*(latitude - Blat) + (longitude -
Blong)*(longitude - Blong) as distance, issues
FROM tab3
ORDER BY distance ASC
LIMIT 250),
tab5 AS(
SELECT description, listing_url
FROM Lsting JOIN tab3 ON tab3.latitude =
Lsting.latitude AND tab3.longitude =
Lsting.longitude)
SELECT distinct description, listing_url
FROM tab5;
```

```
SELECT * FROM peacefulBnbs
WHERE description LIKE '%peaceful%' AND
description LIKE '%relax%'
LIMIT 5
```

Query 4: Pre-Optimisation

```
WITH tab1 AS (SELECT amenities, name, id,
price, listing_url, room_type, host_name,
picture_url, host_url
    FROM Lsting JOIN Hosts ON Hosts.id =
Lsting.host_id
    WHERE amenities LIKE '%${t}%' AND
description LIKE '%${movie}%' AND price <=
'${price}' AND host_is_superhost = '${sh}' AND
accommodates >= '${ppl}'),
    tab2 AS (SELECT listing_id
    FROM reviews
    WHERE comments LIKE '%${t}%'),
    tab3 AS (SELECT tab1.id, tab1.name,
tab1.price, tab1.listing_url, tab1.host_name,
tab1.room_type, tab1.picture_url, tab1.host_url
    FROM tab1, tab2
    WHERE tab1.id = tab2.listing_id),
    tab4 AS (SELECT amenities, name, id, price,
listing_url, room_type
    FROM Lsting JOIN Hosts ON Hosts.id =
Lsting.host_id
    WHERE amenities LIKE '%${r}%' AND
description LIKE '%${movie}%' AND price <=
'${price}' AND host_is_superhost = '${sh}' AND
accommodates >= '${ppl}'),
    tab5 AS (SELECT listing_id
    FROM reviews
    WHERE comments LIKE '%${r}%'),
    tab6 AS (SELECT tab4.id, tab4.name,
tab4.price, tab4.listing_url
    FROM tab4, tab5
    WHERE tab4.id = tab5.listing_id),
    tab7 AS (SELECT amenities, name, id, price,
listing_url, room_type
    FROM Lsting JOIN Hosts ON Hosts.id =
Lsting.host_id
    WHERE amenities LIKE '%${w}%' AND
description LIKE '%${movie}%' AND price <=
'${price}' AND host_is_superhost = '${sh}' AND
accommodates >= '${ppl}'),
    tab8 AS (SELECT listing_id
    FROM reviews
```

Query 4: Post-Optimisation

```
WITH tab1 AS (SELECT id, name, comments,
amenities,
   room_type, price, listing_url, picture_url,
host_name, host_url
   FROM Lsting JOIN reviews ON Lsting.id =
reviews.listing_id
   JOIN Hosts ON Hosts.id = Lsting.host_id
   WHERE Lsting.price <= 500 AND
Lsting.host_is_superhost = 0
   AND Lsting.accommodates >= 1 AND
description LIKE '%beautiful%'),
   tab2 AS (SELECT id, name, comments,
amenities, room_type, price,
   listing_url, picture_url, host_url, host_name
   FROM tab1
   WHERE amenities LIKE '%Wifi%'),
   tab3 AS (SELECT id, name, comments,
amenities, room_type, price,
   listing_url, picture_url, host_url, host_name
   FROM tab2 WHERE amenities LIKE '%TV%'),
   tab4 AS (SELECT id, name, comments,
amenities, room_type, price,
   listing_url, picture_url, host_url, host_name
   FROM tab3
   WHERE amenities LIKE '%""%')
   SELECT DISTINCT id, name, room_type, price,
   listing_url, picture_url, host_url, host_name
   FROM tab3
   WHERE amenities LIKE '%%' AND (comments
LIKE '%  %'
   OR comments LIKE '%  %' OR comments LIKE
'% Wifi %' OR comments LIKE '% TV %')
   LIMIT 10;
```

```
    WHERE comments LIKE '%${w}%'),
    tab9 AS (SELECT tab7.id, tab7.name,
tab7.price, tab7.listing_url
    FROM tab7, tab8
    WHERE tab7.id = tab8.listing_id),
    tab10 AS (SELECT amenities, name, id, price,
listing_url, room_type
    FROM Lsting JOIN Hosts ON Hosts.id =
Lsting.host_id
    WHERE amenities LIKE '%${k}%' AND
description LIKE '%${movie}%' AND price <=
'${price}' AND host_is_superhost = '${sh}' AND
accommodates >= '${ppl}'),
    tab11 AS (SELECT listing_id
    FROM reviews
    WHERE comments LIKE '%${k}%'),
    tab12 AS (SELECT tab10.id, tab10.name,
tab10.price, tab10.listing_url
    FROM tab10, tab11
    WHERE tab10.id = tab11.listing_id)
    SELECT DISTINCT tab3.id, tab3.name,
tab3.price, tab3.listing_url, tab3.room_type,
tab3.host_name, tab3.picture_url, tab3.host_url
    FROM tab3, tab6, tab9, tab12 WHERE tab3.id
= tab6.id AND tab3.id = tab9.id AND tab3.id =
tab12.id
    LIMIT 10;
```

Query 5: Pre-Optimisation

```
WITH tab1 AS (SELECT DISTINCT
AVG(rating) as avg, host_url, host_name, id,
host_id
 FROM Lsting JOIN Hosts on Lsting.host_id =
Hosts.id
 GROUP BY host_id),
 tab2 AS (SELECT *
FROM Lsting OIN Neighbourhood on
Lsting.neighbourhood_cleaned =
Neighbourhood.neighbourhood_cleansed
WHERE neighbourhood = 'Bronx' AND
host_total_listings_count >= 3
 AND host_is_superhost = 1 AND
number_of_reviews >= 10)
 SELECT  DISTINCT tab1.host_id, tab1.avg,
tab1.host_url, tab1.host_name
 FROM tab1 JOIN tab2 ON tab1.host_id =
```

Query 5: Post-Optimisation

```
 SELECT DISTINCT AVG(rating) as avg,
host_url, host_name
  FROM Lsting
  JOIN Neighbourhood on
Lsting.neighbourhood_cleaned =
Neighbourhood.neighbourhood_cleansed
  JOIN Hosts on Lsting.host_id = Hosts.id
  WHERE neighbourhood = '${borough}' AND
host_total_listings_count >= 3
  AND host_is_superhost = 1 AND
number_of_reviews >= 10
  GROUP BY host_id
  ORDER BY avg DESC
  LIMIT 5;
```

```
tab2.host_id
  ORDER BY tab1.avg DESC
  LIMIT 5;
```