London Airbnb Reviews

*Note: Sub-titles are not captured in Xplore and should not be used

1st Apoorva Shrinivas Bhapkar
Computer Science
University at Buffalo
Buffalo, New York
abhapkar@buffalo.edu

2nd Jonathan Liao
Computer Science
University at Buffalo
Buffalo, New York
jrliao@buffalo.edu

3rd Sachin Ramesh Sarsambi
Computer Science
University at Buffalo
Buffalo, New York
ssarsamb@buffalo.edu

***Abstract*—Reviews from customers are significant for several reasons. They enable customers to choose whether or not to utilize a product or service in an informed manner. By pointing up areas for development, they also assist businesses in bettering their services. Additionally, they can aid businesses in establishing credibility and trust with potential customers. In the digital era of today, where consumers have more access to information than ever before, customer reviews are particularly significant. Businesses can get customer reviews in a number of ways. One strategy is to get reviews from customers after they utilize a service. Another strategy is to establish a platform where users may provide their experiences and ratings. Businesses must make sure to reply to consumer reviews promptly and professionally once they have been gathered. Customers will see that their opinions are respected and that the company is dedicated to offering high-quality services. The paper aims to create a relational database application that would combine London Airbnb data to allow businesses to obtain analytical details of their services. Customer testimonials are an invaluable resource for businesses.**

I. Academic Integrity Statement

We have read and understood the course academic integrity policy in the syllabus of the class.

II. Introduction

Airbnb is one of the most unique and preferred marketplace for house owners to list, discover and book accommodations. They are different from the traditional hotel networks. Airbnb is so convenient that most travelers tend to include it as part of their trips. Each listing on the Airbnb has multiple features associated with it, such as price, locality, property details and so on. Along with this each listing would have ratings and reviews provided by the customers who most likely utilized the listing property. The ratings and positivity of the reviews play a vital role in defining the price of a place. The customers rate a listing with regards to several features such as location, size, amenities, and most importantly the price. A listing is associated with the rating which is an average of all the customer ratings. Further each listing would have review section which displays the customers experience in few words.

A. Targeted Audience

A real-life scenario could be a review site such as Yelp but used for properties. The users who are intended to use our application would retrieve information about the properties and their reviews. The other types of the users would be property hosts who would enlist their property listings as well. The administrator will be the person responsible for updating the database with new listings and new reviews. While also querying the database to get the correct information displayed in the application.

III. Data

Airbnb collects all the data about hosts, properties, guests and their reviews. All of this data is publicly available on the Inside Airbnb website and even on Kaggle. The source dataset on Kaggle is available in csv format and even had innumerable abnormalities which required cleaning. The dataset size was 2GB and utilizing a tool like excel would crash one's OS. Hence we decided to use Postgres as our database system. However, as the data had some inconsistencies we had to preprocess the data before moving it to the Postgres. In order to preprocess and load data we developed Python scripts. The dataset was divided into 5 individual cvs files: calendar, listings, listings_summary, neighborhoods, reviews and reviews_summary and the data initially was not normalized. The reason for choosing PostgreSQL, unlike excel, is that it provides a myriad of capabilities to deal with such a huge amount of data. Accessing such huge data via csv results in kernel freeze whereas in the database each data is indexed and can be accessed via these indices. Also, as we need to establish relation between multiple data like host and the listing, postgresQL allows us to create foreign key constraints. We enhanced this feature further by using the cascade delete on the foreign keys.

IV. Data Pre-processing

The London Airbnb Reviews data has lot of abnormalities that required cleaning. The files were obtained in compressed zip format and contained multiple csv files. If loaded to Postgres as they were, the data cleaning would've been harder to perform on Postgres. Hence we developed a python script that performed data cleaning by removing NaN values, missing values and so on. Further the relation schemas were not in BCNF, hence we used the same python script to decompose the relation schemas to BCNF. Once the relations were in

BCNF, the data was loaded to Postgres from the script itself. For the purpose of data cleaning, decomposition and loading we used pythons pandas module which is capable of handling large amount of datasets.

V. Changes since Milestone 1

In the milestone 1, our relation schema had one transitive dependency in the reviews table: {id}→{reviewer_id} & {reviewer_id}→{reviewer_name}. Hence we decided to split the reviews table into reviewers and reviews table. Below are the new/edited tables:

A. Table reviewer

Reviewer			
Name	**Data type**	**Description**	**Constraints**
reviewer_id	BigInt	Unique id assigned to a reviewer	No Null values
reviewer_name	VARCHAR(255)	Name of the reviewer	No Null values

B. Table review

Review			
Name	**Data type**	**Description**	**Constraints**
Id	BigInt	Unique id assigned to a review	No Null values
listing_id	BigInt	Unique id assigned to a listing	No Null values
date	Date	Date of the review	No Null values
reviewer_id	BigInt	Unique id assigned to a reviewer	No Null values
comments	Text	Comments provided	No Null values

VI. Database and Table Creation

We have created a database named 'London Airbnb' in Postgres SQL. Then we developed a python script to create the tables in the database and insert all the records into the tables. Then we altered all the tables to add constraints such as primary key, foreign key and cascade.

1) Table neighbourhood: We created neighbourhood table using neighbourhood.csv. Then we altered the table to add column 'id' as the primary key.

```
CREATE TABLE public.neighbourhood
(
    id bigint NOT NULL,
    neighbourhood character varying(255)
);
```

Query 1 : Creating neighbourhood table

```
ALTER TABLE public.neighbourhood ADD PRIMARY KEY (id
    );
```

Query 2 : Alter neighbourhood table

2) Table host: We created host table using listings.csv. Then we altered the table to add column 'host_id' as the primary key.

```
CREATE TABLE public.host
(
    host_id bigint NOT NULL,
    host_url character varying(255),
    host_name character varying(255),
    host_since date,
    host_location character varying(255),
    host_about text COLLATE,
    host_response_time character varying(255),
    host_response_rate character varying(5),
    host_acceptance_rate double precision,
    host_is_superhost boolean,
    host_neighbourhood character varying(255),
    host_listings_count integer,
    host_total_listings_count integer,
    host_verifications character varying(255)
)
```

Query 3 : Creating host table

```
ALTER TABLE public.listing ADD PRIMARY KEY (host_id)
    ;
```

Query 4 : Alter host table

3) Table listing: We created listing table using listings.csv. Then we altered the table to add column 'listing_id' as the primary key, 'host_id' and 'neighbourhood_id' as foreign keys. We have also added constraint ON DELETE CASCADE for 'host_id'. ON DELETE CASCADE constraint is used to specify whether we want the corresponding rows in the child relation to be deleted if the rows from the parent relation are deleted.

```
CREATE TABLE public.listing
(
    listing_id bigint NOT NULL,
    name character varying(255),
    description text,
    experiences_offered character varying(255),
    house_rules text,
    host_id bigint,
    street character varying(255),
    city character varying(255),
    state character varying(255),
    zipcode character varying(255),
    country character varying(255),
    latitude double precision,
    longitude double precision,
    property_type character varying(255),
    room_type character varying(255),
    accommodates integer,
    bathrooms integer,
    bedrooms integer,
    beds integer,
    bed_type character varying(255),
    amenities text,
    square_feet double precision,
    guests_included integer,
    minimum_nights integer,
    maximum_nights integer,
    cancellation_policy character varying(255),
    neighbourhood_id bigint
)
```

Query 5 : Creating listing table

```
ALTER TABLE public.listing ADD PRIMARY KEY (
    listing_id);
```

Query 6 : Alter listing table

```
ALTER TABLE public.listing ADD FOREIGN KEY (host_id)
        REFERENCES public.host (host_id)
        ON DELETE CASCADE;
```

Query 7 : Alter listing table

```

ALTER TABLE public.listing ADD FOREIGN KEY (
    neighbourhood_id)
        REFERENCES public.neighbourhood (id);
```

Query 8 : Alter listing table

4) Table calendar: We created calendar table using calendar.csv. Then we altered the table to add column 'listing_id' and 'date' as the primary key, 'listing_id' as foreign keys with ON DELETE CASCADE constraint.

```
CREATE TABLE public.calendar
(
    listing_id bigint NOT NULL,
    date date NOT NULL,
    available boolean,
    price money,
    adjusted_price money,
    minimum_nights integer,
    maximum_nights integer
)
```

Query 9 : Creating calendar table

```
ALTER TABLE public.calendar ADD PRIMARY KEY (
    listing_id, date);
```

Query 10 : Alter calendar table

```

ALTER TABLE public.calendar ADD FOREIGN KEY (
    listing_id)
        REFERENCES public.listing (listing_id)
        ON DELETE CASCADE;
```

Query 11 : Alter calendar table

5) Table pricing: We created pricing table using listings.csv. Then we altered the table to add column 'listing_id' as the primary key, 'listing_id' as foreign key with ON DELETE CASCADE constraint.

```
CREATE TABLE IF NOT EXISTS public.pricing
(
    price money,
    weekly_price money,
    monthly_price money,
    security_deposit money,
    cleaning_fee money,
    extra_people money,
    listing_id bigint NOT NULL
)
```

Query 12 : Creating pricing table

```
ALTER TABLE public.pricing ADD PRIMARY KEY (
    listing_id);
```

Query 13 : Alter pricing table

```

ALTER TABLE public.pricing ADD FOREIGN KEY (
    listing_id)
        REFERENCES public.listing (listing_id)
        ON DELETE CASCADE;
```

Query 14 : Alter pricing table

6) Table rating: We created rating table using listings.csv. Then we altered the table to add column 'listing_id' as the primary key, 'listing_id' as foreign key with ON DELETE CASCADE constraint.

```
CREATE TABLE IF NOT EXISTS public.rating
(
    number_of_reviews integer,
    review_scores_rating double precision,
    review_scores_accuracy double precision,
    review_scores_cleanliness double precision,
    review_scores_checkin double precision,
    review_scores_communication double precision,
    review_scores_location double precision,
    review_scores_value double precision,
    listing_id bigint NOT NULL
)
```

Query 15 : Creating rating table

```
ALTER TABLE public.rating ADD PRIMARY KEY (
    listing_id);
```

Query 16 : Alter rating table

```

ALTER TABLE public.rating ADD FOREIGN KEY (
    listing_id)
        REFERENCES public.listing (listing_id)
        ON DELETE CASCADE;
```

Query 17 : Alter rating table

7) Table reviewer: We created reviewer table using reviews.csv. Then we altered the table to add column 'reviewer_id' as the primary key.

```
CREATE TABLE public.reviewer
(
    reviewer_id bigint NOT NULL,
    reviewer_name character varying(255)
);
```

Query 18 : Creating reviewer table

```
ALTER TABLE public.reviewer ADD PRIMARY KEY (
    reviewer_id);
```

Query 19 : Alter reviewer table

8) Table reviews: We created reviews table using reviews.csv. Then we altered the table to add column 'id' as the primary key, 'listing_id' and reviewer_id as foreign keys with ON DELETE CASCADE constraint.

```
CREATE TABLE public.reviews
(
    listing_id bigint,
    id bigint NOT NULL,
    date date,
    reviewer_id bigint,
    comments text
)
```

Query 20 : Creating reviews table

```
ALTER TABLE public.reviews ADD PRIMARY KEY (id);
```

Query 21 : Alter reviews table

```

ALTER TABLE public.reviews ADD FOREIGN KEY (
    listing_id)
        REFERENCES public.listing (listing_id)
        ON DELETE CASCADE;
```

Query 22 : Alter reviews table

```

ALTER TABLE public.reviews ADD FOREIGN KEY (
    reviewer_id)
        REFERENCES public.reviewer (reviewer_id)
        ON DELETE CASCADE;
```

Query 23 : Alter reviews table

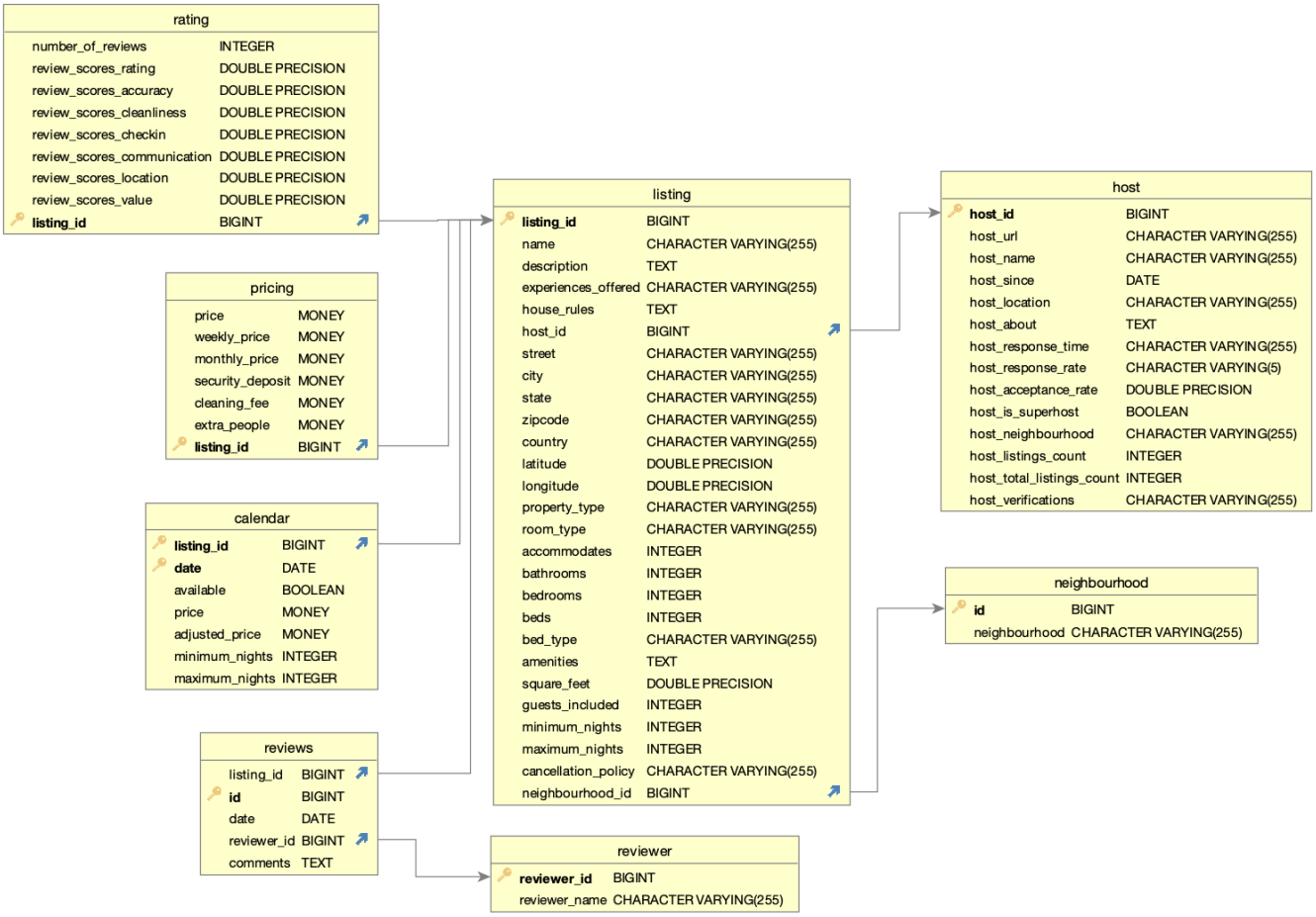


Fig. 1. ER Diagram for London AirBnb Reviews Data.

VII. Issues Encountered

After loading the data from python scripts using pandas, some of the column datatype were inaccurate. Hence we had to modify the table columns using ALTER statements to correct the datatype. We had to change: column for date storing from string to date, column for storing price from string to money and column for storing coordinates to double precision. Further there were no constraints and CASCADE introduced on any of the schemas and we introduced these using ALTER statements. Below are some of the examples to correct the datatype in the tables.

```
1
2 ALTER TABLE pricing ALTER COLUMN weekly_price TYPE
    money using weekly_price::text::money;
3 ALTER TABLE reviews ALTER COLUMN reviewer_name TYPE
    varchar(255) using reviewer_name::text::varchar;
4 ALTER TABLE reviews ALTER COLUMN date TYPE Date
    using TO_DATE(date, 'YYYY-MM-DD');
5 ALTER TABLE host ALTER COLUMN host_is_superhost TYPE
     boolean using host_is_superhost::text::boolean;
6 ALTER TABLE rating ALTER COLUMN number_of_reviews
    TYPE integer using number_of_reviews::bigint::
    integer;
7 ALTER TABLE listing ALTER COLUMN latitude TYPE
    double precision USING latitude::int::double
    precision
```

VIII. Functional Dependencies

A functional dependency is a constraint between two sets of attributes in a relation from a database i.e, a functional dependency is a constraint between two attributes in a relation. A functional dependency is denoted as A $\rightarrow$ B, where A is a set of attributes that can determine the value of B. Functional dependencies are necessary when it comes to normalising a database. The process of normalization involves organizing a relational database according to normal forms to ensure reduced data redundancy and improved data integrity. Normalization involves dividing the database into smaller tables. A relationship between the tables is created using foreign keys. A schema is said to be in BCNF if for every non-trivial FD A $\rightarrow$ B, A is a super key. Initially when we came across the datasets, the relations were in 1NF form as there was evidently myriad of redundancies and anomalies. We had to break the listing table into: listing, hosts, pricing and neighbourhood. Further we split reviews table into: reviews, reviewers and rating. As from the schema it is evident that there are no transitive dependencies involved in the relation schemas and for all the functional dependencies the determinant is the super-key. Below are the FD's for each relation.

1) Table listing: All the attributes are defined by listing_id, hence the number of FD is limited to one. In the above FD, the determinant listing_id is the superkey. Hence, the relation is in BCNF as there are no transitive dependencies.

{listing_id} → {listing_id, name, description, experiences_offered, house_rules, host_id, street, city, state, zipcode, country, latitude, longitude, property_type, room_type, accommodates, bathrooms, bedrooms, beds, bed_type, amenities, square_feet, guests_included, minimum_nights, maximum_nights, cancellation_policy, neighbourhood_id}

2) Table host: All the attributes are defined by host_id, hence the number of FD is limited to one. In the above FD, the determinant host_id is the superkey. Hence, the relation is in BCNF as there are no transitive dependencies.

{host_id} → {host_id, host_url, host_name, host_since, host_location, host_about, host_response_time, host_response_rate, host_acceptance_rate, host_is_superhost, host_neighbourhood, host_listings_count, host_total_listings_count, host_verifications}

3) Table reviewer: All the attributes are defined by reviewer_id, hence the number of FD is limited to one. In the above FD, the determinant reviewer_id is the superkey. Hence, the relation is in BCNF as there are no transitive dependencies.

{reviewer_id} → {reviewer_id, reviewer_name}

4) Table reviews: All the attributes are defined by id, hence the number of FD is limited to one. In the above FD, the determinant id is the superkey. Hence, the relation is in BCNF as there are no transitive dependencies.

{id} → {id, listing_id, date, reviewer_id, comments}

5) Table pricing: All the attributes are defined by listing_id, hence the number of FD is limited to one. In the above FD, the determinant listing_id is the superkey. Hence, the relation is in BCNF as there are no transitive dependencies.

{listing_id} → {price, weekly_price, monthly_price, security_deposit, cleaning_fee, extra_people, listing_id}

6) Table calendar: All the attributes are defined by the combination of listing_id and date, hence the number of FD is limited to one. In the above FD, the determinant combination of listing_id and date is the superkey. Hence, the relation is in BCNF as there are no transitive dependencies.

{listing_id , date} → {listing_id, date, available, price, adjusted_price, minimum_nights, maximum_nights}

7) Table rating: All the attributes are defined by listing_id, hence the number of FD is limited to one. In the above FD, the determinant listing_id is the superkey. Hence, the relation is in BCNF as there are no transitive dependencies.

{listing_id} → {number_of_reviews, review_scores_rating, review_scores_accuracy, review_scores_cleanliness, review_scores_checkin, review_scores_communication, review_scores_location, review_scores_value, listing_id}

8) Table neighbourhood: All the attributes are defined by id, hence the number of FD is limited to one. In the above FD, the determinant id is the superkey. Hence, the relation is in BCNF as there are no transitive dependencies.

{id} → {id, neighborhood}

IX. Query Execution Analysis

We performed indexing on some tables to improve query execution time. Below are some queries that were affected and the indexes that were created to improve the query execution time.

A. Query 1

In the fig. 2 and fig. 3, the query execution analysis on calendar table before and after indexing can be seen.

Fig. 2. Below is the query execution analysis on calendar table:

```
30 explain analyze select * from calendar where date='2019-12-12' and available='true'
31 and CAST(price as DECIMAL(5,2))<=CAST('100' as DECIMAL(5,2));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on calendar (cost=0.00..2834.00 rows=48 width=37) (actual time=0.441..16.367 rows=9...
2	Filter: (available AND (date = '2019-12-12'::date) AND ((price)::numeric(5,2) <= 100.00::numeric(5,2...
3	Rows Removed by Filter: 99540
4	Planning Time: 0.237 ms
5	Execution Time: 16.391 ms

Fig. 3. Below is the query execution analysis on calendar table after indexing:

```
29 create index calendar_idx on calendar(date);
30 explain analyze select * from calendar where date='2019-12-12' and available='true'
31 and CAST(price as DECIMAL(5,2))<=CAST('100' as DECIMAL(5,2));
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on calendar (cost=6.35..577.58 rows=48 width=37) (actual time=1.373..4.640 rows=95 loops=1)
2	Recheck Cond: (date = '2019-12-12'::date)
3	Filter: (available AND ((price)::numeric(5,2) <= 100.00::numeric(5,2)))
4	Rows Removed by Filter: 180
5	Heap Blocks: exact=245
6	-> Bitmap Index Scan on calendar_idx (cost=0.00..6.34 rows=273 width=0) (actual time=0.919..0.919 rows=275 loops=1)
7	Index Cond: (date = '2019-12-12'::date)
8	Planning Time: 1.340 ms
9	Execution Time: 4.675 ms

B. Query 2

In the fig. 4 and fig. 5, the query execution analysis on rating table before and after indexing can be seen.

C. Query 3

In the fig. 6 and fig. 7, the query execution analysis on reviews table before and after indexing can be seen.

Fig. 4. Below is the query execution analysis on rating table:

```
47 explain analyze select * from rating where number_of_reviews>600;
48
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on rating (cost=0.00..1981.34 rows=5 width=68) (actual time=0.428..12.450 rows=5 loops…
2	Filter: (number_of_reviews > 600)
3	Rows Removed by Filter: 85062
4	Planning Time: 0.405 ms
5	Execution Time: 12.489 ms

Fig. 5. Below is the query execution analysis on rating table after indexing:

```
46 create index rating_idx on rating(number_of_reviews);
47 explain analyze select * from rating where number_of_reviews>600;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on rating (cost=4.40..55.39 rows=14 width=68) (actual time=0.008..0.012 rows=5 loops=1)
2	Recheck Cond: (number_of_reviews > 600)
3	Heap Blocks: exact=4
4	-> Bitmap Index Scan on rating_idx (cost=0.00..4.40 rows=14 width=0) (actual time=0.004..0.004 rows=5 loops=1)
5	Index Cond: (number_of_reviews > 600)
6	Planning Time: 1.422 ms
7	Execution Time: 0.039 ms

Fig. 6. Below is the query execution analysis on reviews table:

```
31 explain analyze select * from reviews where reviewer_id=13237496;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on reviews (cost=0.00..6267.98 rows=47 width=361) (actual time=32.643..70.070 rows=40 loops…
2	Filter: (reviewer_id = 13237496)
3	Rows Removed by Filter: 99958
4	Planning Time: 0.211 ms
5	Execution Time: 70.112 ms

Fig. 7. Below is the query execution analysis on reviews table after indexing:

```
30 create index reviews_idx on reviews(reviewer_id);
31 explain analyze select * from reviews where reviewer_id=13237496;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on reviews (cost=4.66..179.60 rows=47 width=361) (actual time=0.516..0.938 rows=40 loops=1)
2	Recheck Cond: (reviewer_id = 13237496)
3	Heap Blocks: exact=19
4	-> Bitmap Index Scan on reviews_idx (cost=0.00..4.65 rows=47 width=0) (actual time=0.476..0.476 rows=40 loops=1)
5	Index Cond: (reviewer_id = 13237496)
6	Planning Time: 0.660 ms
7	Execution Time: 1.270 ms

X. SQL Operations

A. Insert Query

For insert operation we insert a new neighbourhood entry into neighbourhood table. The insert operations can be seen in Fig.8 and Fig. 9.

B. Delete Query

For delete operation we checked the records for reviewer_id=548673 in reviewer and reviews table. After we deleted this reviewer from reviewer table, all records for this reviewer_id were deleted from reviewer and reviews table. This is because we enabled CASCADE for reviewer_id. The delete operations can be seen in Fig. 10, Fig. 11, Fig. 12, Fig. 13 and Fig. 14.

Fig. 8. Insert query for neighbourhood table

```
30 Insert into neighbourhood values(151,'Inglewood');
```

Data Output | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 67 msec.
```

Fig. 9. Neighbourhood table after inserting a row:

```
29 select * from neighbourhood where id=151;
```

Data Output | Messages | Notifications

	id [PK] bigint	neighbourhood character varying (255)
1	151	Inglewood

Fig. 10. Before Delete Query:

```
29 select * from reviewer where reviewer_id=548673;
30
```

Data Output | Messages | Notifications

	reviewer_id [PK] bigint	reviewer_name character varying (255)
1	548673	Nicky

Fig. 11. Before Delete Query:

```
32 select * from reviews where reviewer_id=548673;
```

Data Output | Messages | Notifications

	listing_id bigint	id [PK] bigint	date date	reviewer_id bigint	comments text
1	280234	2817366	2012-11-05	548673	Our stay at Helen's hous

Fig. 12. Delete operation:

```
34 delete from reviewer where reviewer_id=548673;
```

Data Output | Messages | Notifications

```
DELETE 1

Query returned successfully in 135 msec.
```

C. Update Query

For update operation we checked the records for listing_id=38151 in pricing table and then updated the weekly price. The update operations can be seen in Fig. 15, Fig. 16 and Fig. 17.

Fig. 13. After Delete Query:

```
29  select * from reviewer where reviewer_id=548673;
```

Data Output | Messages | Notifications

reviewer_id [PK] bigint	reviewer_name character varying (255)

Fig. 14. After Delete Query:

```
32  select * from reviews where reviewer_id=548673;
```

Data Output | Messages | Notifications

listing_id bigint	id [PK] bigint	date date	reviewer_id bigint	comments text

Fig. 15. Before Update Query:

```
44  select * from pricing where listing_id=38151;
```

Data Output | Messages | Notifications

	price money	weekly_price money	monthly_price money	security_deposit money	cleaning_fee money	extra_people money	listing_id [PK] bigint
1	$65.00	$0.00	[null]	[null]	[null]	$0.00	38151

Fig. 16. Before Update Query:

```
52  UPDATE pricing SET weekly_price='300'::numeric::money
53  WHERE listing_id=38151;
54
```

Data Output | Messages | Notifications

```
UPDATE 1

Query returned successfully in 82 msec.
```

Fig. 17. After Update Query:

```
44  select * from pricing where listing_id=38151;
45
```

Data Output | Messages | Notifications

	price money	weekly_price money	monthly_price money	security_deposit money	cleaning_fee money	extra_people money	listing_id [PK] bigint
1	$65.00	$300.00	[null]	[null]	[null]	$0.00	38151

D. Select Query : Join

This query displays selected columns from listing and host tables using join. This can be seen in Fig. 18.

E. Select Query

This query displays all the listings that are available on 2019-12-12 and price is less than 100 USD. This can be seen in Fig. 19.

Fig. 18. Select query for listing and host:

```
29  select l.listing_id, h.host_id, h.host_name from listing l
30  join host h on l.host_id=h.host_id order by host_id;
31
```

Data Output | Messages | Notifications

	listing_id bigint	host_id bigint	host_name character varying (255)
1	32722203	2697	Bertrand
2	32191740	4775	Sebastian
3	32722192	4775	Sebastian
4	32190981	4775	Sebastian
5	35913421	4775	Sebastian
6	12140520	4775	Sebastian
7	15267243	4775	Sebastian
8	12032227	4775	Sebastian
9	28805389	5653	Atilla Ata
10	37828755	6774	Roly & Peter

Total rows: 1000 of 85067 | Query complete 00:00:00.914

Fig. 19. Select query for Price:

```
26  select * from calendar where date='2019-12-12' and available='true' and
27  CAST(price as DECIMAL(5,2))<=CAST('100' as DECIMAL(5,2));
28
29
```

Data Output | Messages | Notifications

	listing_id [PK] bigint	date [PK] date	available boolean	price money	adjusted_price money	minimum_nights integer	maximum_nights integer
1	92399	2019-12-12	true	$75.00	$75.00	1	365
2	38151	2019-12-12	true	$65.00	$65.00	1	730
3	38605	2019-12-12	true	$58.00	$58.00	2	14
4	93263	2019-12-12	true	$95.00	$95.00	5	360
5	13913	2019-12-12	true	$50.00	$50.00	1	29
6	38950	2019-12-12	true	$50.00	$50.00	1	25
7	38995	2019-12-12	true	$40.00	$40.00	1	4
8	39387	2019-12-12	true	$49.00	$49.00	5	10
9	96008	2019-12-12	true	$80.00	$80.00	2	365
10	41509	2019-12-12	true	$42.00	$42.00	1	50

F. Select Query : Group By

This query displays all the listing who has more that 500 reviews. This can be seen in Fig. 20.

Fig. 20. Select query using group by:

```
46  select listing_id,count(listing_id) from reviews group by listing_id having count(listing_id)>500;
47
```

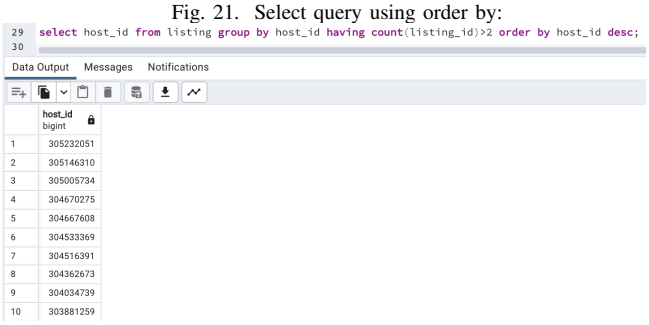
Data Output | Messages | Notifications

	listing_id bigint	count bigint
1	392220	511
2	604185	616
3	502190	524
4	36660	517
5	361662	546
6	521359	507
7	375006	501

G. Select Query : Order By

This query displays all the host_id in descending order who have more that two listings. This can be seen in Fig. 21.

Fig. 21. Select query using order by:



```
select host_id from listing group by host_id having count(listing_id)>2 order by host_id desc;
```

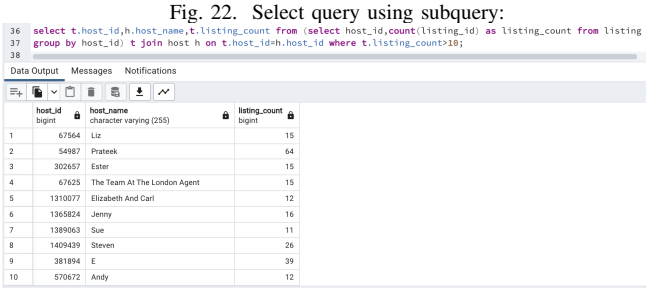
Data Output | Messages | Notifications

	host_id bigint
1	305232051
2	305146310
3	305005734
4	304670275
5	304667608
6	304533369
7	304516391
8	304362673
9	304034739
10	303881259

H. Select Query : Sub-Query

This query displays all the hosts and the total number of listing that they have. This query displays those hosts that have listing count greater than 10. This can be seen in Fig. 22.

Fig. 22. Select query using subquery:



```
select t.host_id,h.host_name,t.listing_count from (select host_id,count(listing_id) as listing_count from listing
group by host_id) t join host h on t.host_id=h.host_id where t.listing_count>10;
```

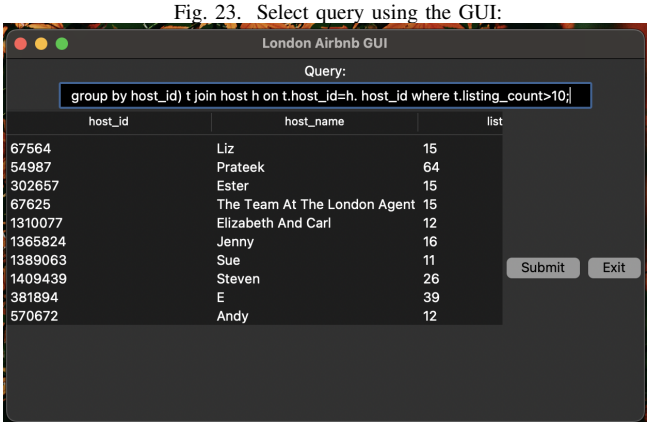
Data Output | Messages | Notifications

	host_id bigint	host_name character varying (255)	listing_count bigint
1	67564	Liz	15
2	54987	Prateek	64
3	302657	Ester	15
4	67625	The Team At The London Agent	15
5	1310077	Elizabeth And Carl	12
6	1365824	Jenny	16
7	1389063	Sue	11
8	1409439	Steven	26
9	381894	E	39
10	570672	Andy	12

XI. GUI Application

We used pythons *tkinter* library to develop a GUI application to run and display queries.

Fig. 23. Select query using the GUI:



Future Work

The database was created using existing collected datasets. As a future project we can add a data scraping option which would look for London Airbnb Reviews data on insideairbnb site and fetch the data, preprocess and load it to our database. This would require using an ETL service like AWS Glue.

Important Links

https://buffalo.box.com/s/mrzrkbs7byljrgya18nwhmfvovb4j068

Contribution

Fig. 24. Contribution from each team member

Sr. No.	Team Member Name	Contribution (%)
1	Apoorva Bhapkar	33%
2	Jonathan Liao	33%
3	Sachin Ramesh Sarsambi	33%

References

[1] https://www.kaggle.com/datasets/labdmitriy/airbnb
[2] http://insideairbnb.com/get-the-data/
[3] https://medium.com/analytics-vidhya/exploratory-data-analysis-on-airbnb-properties-in-london-39eb80da6d15
[4] https://www.postgresql.org/download/
[5] https://www.postgresql.org/docs/current/datatype.html