Lucien Michaël Iseli, 274999
Florian Maxime Charles Ravasi, 245940
Jules Eliott Gottraux, 262413

Milestone 3

Design choices

The goal when creating a database management system is to separate as much as possible the entities so that when using a small portion of the data in the database, which is normally the case, we will only join useful tables and so will have less overhead induced by the useless data. Thus, we separated the hosts, the dates, the countries etc... in their separate tables (see the diagram for an exhaustive list of the entities). This way, when doing a query over the listings, we begin with the Listings table which only contains the ids of the related informations and we *join* with the needed tables. The idea is really to not *join* useless data, the textual informations is completely separated from all other data in an entity Listing_infos for that reason: we will rarely (never in a practical sense) make query regarding the description that the listing has, it is for human and not database.

The idea behind those entity is also to have an intuitive structure. For example for the neighbourhoods, we have a textual description of the neighbourhood of the listing, along with the name of the neighbourhood and the neighbourhood of the host. Thus, the neighbourhood is an entity by itself in realtion with the description of that neighbourhood. That way data that are meaningful to each other are close but are separated enough so that an entity have a meaning but is as close as possible

In order to decrease the duplication in certain tables, we chose to create separate tables for unique "objects" such as the Amenity, City, Room_type or Cancellation_policy tables for example. Therefore, instead of having duplicates stored as string in the Bedroom table, for example, it is stored as identifiers, i.e integers. Hence, the queries are faster because we do not compute equalities on string anymore and we avoid storing strings in all rows of table but rather a string one time and then a reference to this string, reducing the amount of data stored significantly.

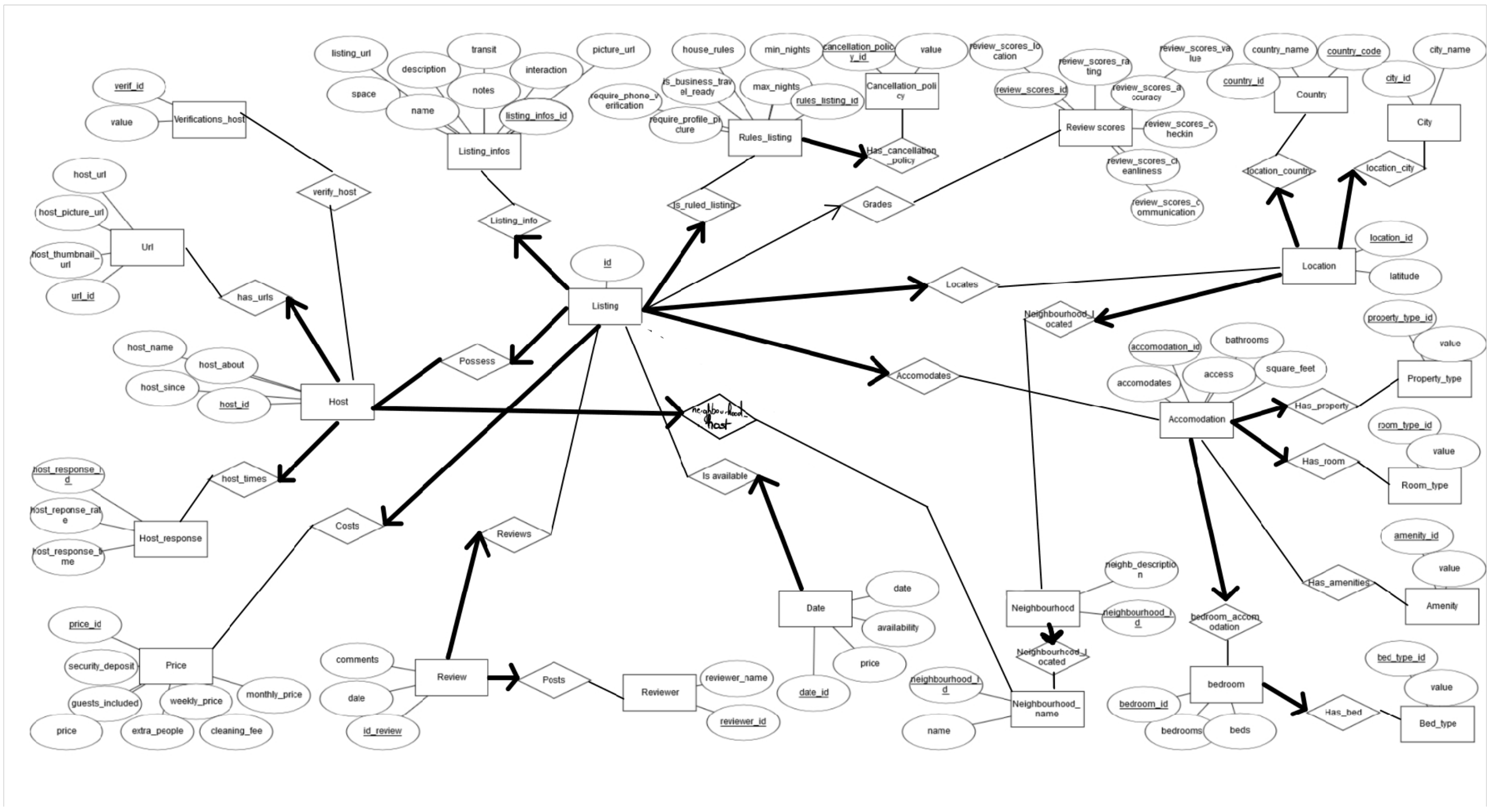
As an example, imagine we are querying about accomodations, but we want to filter them according to the room type and the amenity they possess. We can filter out the accommodations according to the room type id and the amenity ids comparing only integers when using the accomodation table.

Some other less important choice of implementation we made are to store boolean as integer value rather than string or sets has many-to-many relationship rather than strings too. For the amenities for example, at first in the data they are as string of the form {Wifi, Shampoo, ...} but, not even speaking about the duplication, it is a very bad design to store as this in the database. To make a query on accommodation that contain the *Wifi* amenity, we would need to make a *amenity like "%Wifi%"* which is not efficient.

We decided not to map all version of a city name to the real city name to not loose information, thus when making a query for Berlin for example, an equality with "Berlin" would miss a singnificative part of listings that in fact are in Berlin. To lessen this effect when making a query we just lower the two name and check if the string *contains* the lowered city name.

We have to remove around 5 rows of data of the csv, example reason are that one host had a duplicated host_id and some names were on multiple lines.

Entity Relationship Diagram



Creation of Tables

```
CREATE TABLE Reviewers(rev_id INTEGER NOT NULL, rev_name CHAR(50), primary key(rev_id));

CREATE TABLE Reviews(list_id INTEGER NOT NULL, review_id INTEGER NOT NULL, date DATE, rev_id INTEGER NOT NULL,
comments VARCHAR(100), primary key(review_id), foreign key(rev_id) references Reviewers, foreign key(list_id) r
eferences Listings);

CREATE TABLE Listings_infos(list_info_id INTEGER NOT NULL, list_url VARCHAR(30), name VARCHAR(10), summary VARC
HAR(100), space VARCHAR(100), description VARCHAR(100), notes VARCHAR(100), transit VARCHAR(100), interaction V
ARCHAR(100), picture_url VARCHAR(50), primary key(list_info_id));

CREATE TABLE Rules_listing(rules_list_id INTEGER NOT NULL, house_rules VARCHAR(100), min_nights INTEGER, max_ni
ghts INTEGER, is_business_travel_ready INTEGER, canc_policy_id INTEGER NOT NULL, require_profile_picture INTEGE
R, require_phone_verif INTEGER, primary key(rules_list_id), foreign key(canc_policy_id) references Cancellation
_policies);

CREATE TABLE Cancellation_policies(canc_policy_id INTEGER NOT NULL, value CHAR(28) NOT NULL, primary key(canc_p
olicy_id), unique(value));

CREATE TABLE Listings(id INTEGER NOT NULL, list_info_id INTEGER NOT NULL, rules_list_id INTEGER NOT NULL, host_
id INTEGER NOT NULL, price_id INTEGER NOT NULL, location_id INTEGER NOT NULL, acc_id INTEGER NOT NULL, review_s
cores_id INTEGER, primary key(id), foreign key(list_info_id) references Listings_infos, foreign key(rules_list_
id) references Rules_listing, foreign key(host_id) references Hosts, foreign key(price_id) references Prices, f
oreign key(location_id) references Locations, foreign key(acc_id) references Accommodations, foreign key(review
_scores_id) references Review_scores);

CREATE TABLE Host_response(resp_id INTEGER NOT NULL, time VARCHAR(20), rate INTEGER, primary key(resp_id), uniq
ue(time, rate));

CREATE TABLE Host_links(host_links_id INTEGER NOT NULL, host_url VARCHAR(50), host_thumbnail_url VARCHAR(50), h
ost_picture_url VARCHAR(50), primary key(host_links_id));
```

```
CREATE TABLE Prices(price_id INTEGER NOT NULL, price FLOAT, weekly_price FLOAT, monthly_price FLOAT,security_deposit FLOAT,cleaning_fee FLOAT, guests_included INTEGER, extra_people INTEGER, primary key(price_id), unique(price, weekly_price, monthly_price, guests_included, extra_people, cleaning_fee, security_deposit));

CREATE TABLE Review_scores(review_scores_id INTEGER NOT NULL, review_scores_rating INTEGER, review_scores_accuracy INTEGER, review_scores_cleanliness INTEGER, review_scores_checkin INTEGER, review_scores_communication INTEGER, review_scores_location INTEGER, review_scores_value INTEGER, primary key(review_scores_id), unique(review_scores_rating, review_scores_accuracy, review_scores_cleanliness, review_scores_checkin, review_scores_communication, review_scores_location, review_scores_value));

CREATE TABLE Property_types(property_type_id INTEGER NOT NULL, value VARCHAR(6) NOT NULL, primary key(property_type_id), unique(value)) ;

CREATE TABLE Room_types(room_type_id INTEGER NOT NULL, value CHAR(16) NOT NULL, primary key(room_type_id), unique(value));

CREATE TABLE Bedrooms(bedroom_id INTEGER NOT NULL, number_bedroom INTEGER, number_beds INTEGER, bed_type_id INTEGER NOT NULL, primary key(bedroom_id), foreign key(bed_type_id) references Bed_types, unique(number_bedroom, number_beds, bed_type_id));

CREATE TABLE Bed_types(bed_type_id INTEGER NOT NULL, value CHAR(15) NOT NULL, primary key(bed_type_id), unique(value));

CREATE TABLE Has_amenities(acc_id INTEGER NOT NULL, amenity_id INTEGER NOT NULL, primary key(acc_id, amenity_id), foreign key(acc_id) references Accommodations, foreign key(amenity_id) references Amenities);

CREATE TABLE Amenities(amenity_id INTEGER NOT NULL, value VARCHAR(8) NOT NULL, primary key(amenity_id), unique(value));

CREATE TABLE Locations(loc_id INTEGER NOT NULL, descr_id INTEGER NOT NULL, city_id INTEGER NOT NULL, country_id INTEGER NOT NULL, latitude FLOAT, longitude FLOAT, primary key(loc_id), foreign key(country_id) references Countries, foreign key(city_id) references Cities, foreign key(descr_id) references Neighbourhood_descriptions) ;

CREATE TABLE Cities(city_id INTEGER NOT NULL, name VARCHAR(8) NOT NULL, primary key(city_id), unique(name));

CREATE TABLE Countries(country_id INTEGER NOT NULL, country_code CHAR(2) NOT NULL, country_name CHAR(10) NOT NULL, primary key(country_id), unique(country_code, country_name));

CREATE TABLE Hosts(host_id INTEGER NOT NULL, h_name VARCHAR(10), h_since DATE, h_about VARCHAR(50), h_urls_id INTEGER NOT NULL, h_response_id INTEGER NOT NULL, h_neighb_id INTEGER NOT NULL, primary key(host_id), foreign key(h_urls_id) references Host_links, foreign key(h_response_id) references Host_response, foreign key(h_neighb_id) references Neighbourhoods);

CREATE TABLE Host_verifications(host_id INTEGER NOT NULL, verif_id INTEGER NOT NULL, primary key(host_id, verif_id), foreign key(host_id) references Hosts, foreign key(verif_id) references Verifications);

CREATE TABLE Verifications(verif_id INTEGER NOT NULL, verif_value CHAR(25) NOT NULL, primary key(verif_id), unique(verif_value));

CREATE TABLE Neighbourhood_descriptions(descr_id INTEGER NOT NULL, description VARCHAR(100), neighb_id INTEGER NOT NULL, primary key(descr_id), foreign key(neighb_id) references Neighbourhoods);

CREATE TABLE Neighbourhoods(neighb_id INTEGER NOT NULL, name VARCHAR(8) NOT NULL, primary key(neighb_id), unique(name));

CREATE TABLE Dates(date_id INTEGER NOT NULL, list_id INTEGER NOT NULL, date DATE, availability INTEGER, price FLOAT, primary key(date_id), foreign key(list_id) references Listings);

CREATE TABLE Accommodations(acc_id INTEGER NOT NULL, access VARCHAR(100), property_type_id INTEGER NOT NULL, room_type_id INTEGER NOT NULL, accommodates INTEGER, bathrooms INTEGER, bedroom_id INTEGER NOT NULL, square_feet INTEGER, primary key(acc_id), foreign key(property_type_id) references Property_types, foreign key(room_type_id) references Room_types, foreign key(bedroom_id) references Bedrooms, unique(access, property_type_id, room_type_id, accommodates, bathrooms, bedroom_id, square_feet));
```

Parsing and insertion of the data

We parsed the data using terminal commands, with awk, sed, cut, etc... In order to get csv files with information related to a table. Example of command:

```
csvquote All_listings.csv | cut -d, -f1,10,33,34,39,31,32,35,36,37,38 | csvquote -u | awk -F',' '{if(NR!=1) {pr
int NR-2,$0} else {print "acc_id",$0}}' | sed 's/\s/,/' > ../tables_csv/Accommodations_extended
```

We then have many finer-grained csv files containing more specific data, and then we read these files and insert everything in our sqlite database with python.
So the main challenge was extracting the valuable data regarding a specific table from the given csv files. Then reading it and inserting the proper data in sqlite with python while being careful to not break relations between entities.

Queries of deliverable 2

1. What is the average price for a listing with 8 bedrooms?

```
SELECT AVG(P.price)
FROM Prices P, Listings L, Accommodations A, Bedrooms B
WHERE P.price_id = L.price_id AND L.acc_id = A.acc_id AND A.bedroom_id = B.bedroom_id AND B.number_bedroom = 8;
```

Result

AVG(P.price)
313.153846153846

2. What is the average cleaning review score for listings with TV?

```
SELECT AVG(R.review_scores_cleanliness)
FROM Review_scores R
WHERE R.review_scores_id IN(SELECT L.review_scores_id
  FROM Listings L
  WHERE L.acc_id IN(SELECT AC.acc_id
    FROM Accommodations AC, Has_amenities H
    WHERE AC.acc_id = H.acc_id AND H.amenity_id IN(SELECT
      AM.amenity_id
      FROM Amenities AM
      WHERE AM.value = "TV")));
```

Result

AVG(R.review_scores_cleanliness)
8.45286155307528

3. Print all the hosts who have an available property between date 03.2019 and 09.2019.

```
SELECT DISTINCT H.h_name
FROM Hosts H, Listings L, Dates D
WHERE H.host_id = L.host_id AND L.id = D.list_id AND D.availability = 1 AND D.date >= '2019-03' AND D.date <= '
2019-09';
```

Result

h_name
Nieves
Patricia
Rafael
Abdel
Javier

4. Print how many listing items exist that are posted by two different hosts but the

hosts have the same name.

```
CREATE VIEW IdName AS
SELECT L.id, L.host_id, H.h_name
FROM Listings L, Hosts H
WHERE L.host_id = H.host_id;

SELECT COUNT(DISTINCT I1.id)
FROM IdName I1, IdName I2
WHERE I1.host_id > I2.host_id AND I1.h_name = I2.h_name
```

Result

COUNT(DISTINCT I1.id)
26533

5. Print all the dates that 'Viajes Eco' has available accomodations for rent (Assuming that 'Viajes Eco' is a host name)

```
Create View Viajes_ids as
Select id
From Listings L
Inner Join Hosts H on H.host_id = L.host_id
Where H.h_name = 'Viajes Eco';

Select D.date
From Dates D
Inner Join Viajes_ids V on D.list_id = V.id
where D.availability = 1;
```

Result

date
2019-03-03
2019-03-02
2019-03-01
2019-02-28
2019-02-27

6. Find all the hosts (host_ids, host_names) that have only one listing.

```
SELECT DISTINCT H.host_id, H.h_name
FROM Hosts H
WHERE H.host_id IN (SELECT L.host_id
  FROM Listings L
  GROUP BY L.host_id
  HAVING COUNT(*) = 1)
```

Result

host_id	h_name
3073	Ricard
3718	Britta
4108	Jana
5154	"Raúl"
11015	Josaiah

7. What is the difference in the average price of listings with and without Wifi.

```
CREATE VIEW LWifi AS
SELECT L.id
FROM Listings L
WHERE L.acc_id IN(SELECT A.acc_id
  FROM Accommodations A, Has_amenities H
  WHERE A.acc_id = H.acc_id AND H.amenity_id IN(SELECT amenity_id
    FROM Amenities A
    WHERE A.value = "Wifi"));

SELECT
  (SELECT AVG(P.price)
  FROM Prices P, Listings L
  WHERE P.price_id = L.price_id AND L.id IN (SELECT L2.id
                                                FROM LWifi L2))
  -
  (SELECT AVG(P.price)
  FROM Prices P
  WHERE P.price_id IN(SELECT L.id
    FROM Listings L
    WHERE L.id NOT IN(SELECT L2.id
      FROM LWifi L2)))
```

Result

13.9926227916909

8. How much more (or less) costly to rent a room with 8 beds in Berlin compared to Madrid on average?

```
Create View List_ids as
Select L.price_id, L.location_id
From Listings L
Where L.acc_id In (
  Select A.acc_id
  From Accommodations A
  Inner Join Bedrooms B on A.bedroom_id = B.bedroom_id
  Where B.number_beds = 8);
Create View List_ids_Berlin as
Select L.price_id
From List_ids L
Where L.location_id In (
  Select Loc.loc_id
  From Locations Loc
  Where Loc.city_id In (
    Select C.city_id
    From Cities C
    Where LOWER(C.name) LIKE '%berlin%')
);

Create View List_ids_Madrid as
Select L.price_id
From List_ids L
Where L.location_id In (
  Select Loc.loc_id
  From Locations Loc
  Where Loc.city_id In (
    Select C.city_id
    From Cities C
    Where LOWER(C.name) LIKE '%madrid%')
);
```

```
SELECT
  (Select avg(P_b.price)
  From List_ids_Berlin L_b
  Inner Join Prices P_b on P_b.price_id = L_b.price_id)
  -
  (Select avg(P_m.price)
  From List_ids_Madrid L_m
  Inner Join Prices P_m on P_m.price_id = L_m.price_id);
```

Result

-101.592615012107

9. Find the top-10 (in terms of the number of listings) hosts (host_ids, host_names) in Spain.

```
CREATE VIEW LMadrid AS
SELECT LI.id, LI.host_id
FROM Listings LI
WHERE LI.location_id IN(SELECT LO.loc_id
  FROM Locations LO
  WHERE LO.city_id IN(SELECT C.city_id
   FROM Cities C
   WHERE LOWER(C.name) like '%madrid%'));


CREATE VIEW Custom AS
SELECT L.id, H.host_id, H.h_name
FROM Hosts H
INNER JOIN LMadrid L ON H.host_id = L.host_id;

SELECT C.host_id, C.h_name
FROM Custom C
GROUP BY C.host_id
ORDER BY COUNT(C.host_id) DESC
LIMIT 10
```

Result

host_id	h_name
99018982	Apartamentos
32046323	Juan
28038703	"Luxury Rentals Madrid"
3566146	"Home Club"
1408525	Mad4Rent

10. Find the top-10 rated (review_score_rating) apartments (id, name) in Barcelona

```
Create View List_ids_Barcelona as
Select L.id, L.review_scores_id, L.list_info_id
From Listings L
Where L.location_id In (
  Select Loc.loc_id
  From Locations Loc
  Where Loc.city_id In (
    Select C.city_id
    From Cities C
    Where LOWER(C.name) LIKE '%barcelona%')
);
```

```
Create View List_ids as
Select L.id, L.list_info_id
From List_ids_Barcelona L
Inner Join Review_scores R on L.review_scores_id = R.review_scores_id
Order by R.review_scores_rating DESC
LIMIT 10;

Select L1.id, L2.name
From List_ids L1
Inner Join Listings_infos L2 on L1.list_info_id = L2.list_info_id;
```

Result

id	name
71520	"Charming apartment with fantastic views!"
179488	"Room for rent in BCN/ non smoker"
190348	"Apartment with large terrace"
250016	"Excellent - 3 bedrooms
282679	"Charming Penthouse

Queries of deliverable 3

1. Print how many hosts in each city have declared the area of their property in square meters. Sort the output based on the city name in ascending order

```
Select C.name, count(C.name)
From Listings L Inner Join Accommodations A on A.acc_id = L.acc_id
Inner Join Locations Loc on Loc.loc_id = L.location_id
        Inner Join Cities C on C.city_id = Loc.city_id
        Where square_feet is not null
        Group by C.name
        Order by C.name;
```

Result

name	count(C.name)
Barcelona	463
Berlin	402
"Berlin (Kreuzberg)"	1
Berlin-Wedding	1
Chiva	1

61 ms

2. The quality of a neighborhood is defined based on the number of listings and the review score of these listings, one way for computing that is using the median of the review scores, as medians are more robust to outliers. Find the top-5 neighborhoods using median review scores(review_scores_ratingof listings in Madrid. Note: Implement the median operator on your own, and do not use the available built-in operator

```sql
drop view if exists ratings_with_neighb;
Create view ratings_with_neighb as
Select N.neighb_id, N.name, S.review_scores_rating
From Listings L Inner Join Locations Loc on Loc.loc_id = location_id
        Inner Join Neighbourhood_descriptions ND on ND.descr_id = Loc.descr_id
        Inner Join Neighbourhoods N on ND.neighb_id = N.neighb_id
        Inner Join Review_scores S on L.review_scores_id = S.review_scores_id
        Inner Join Cities C on Loc.city_id = C.city_id
Where lower(C.name) like '%madrid%';

Drop view if exists ratings_with_neighb2;
Create view ratings_with_neighb2 as
SELECT neighb_id, name, review_scores_rating, count_el
FROM(SELECT name, neighb_id, review_scores_rating, count(*)
    OVER(PARTITION BY N.neighb_id) as count_el
    From ratings_with_neighb N) N2;

SELECT neighb_id, name, review_scores_rating
FROM(SELECT name, neighb_id, review_scores_rating, count_el, Row_Number()
    OVER(PARTITION BY L.neighb_id ORDER BY L.review_scores_rating DESC) AS Row_Number
    From ratings_with_neighb2 L) L2
Where Row_Number = 1 + (count_el / 2)
Order by review_scores_rating DESC
Limit 5;
```

Result

neighb_id	name	review_scores_rating
197	"Tetuán"	100
61	Estrella	98
29	Castilla	97
91	"Hispanoamérica"	96
142	Moratalaz	96

62 ms

3. Find all the hosts (host_ids, host_names) with the highest number of listings.

```sql
Select H.host_id, H.h_name
From Hosts H Inner Join Listings L on H.host_id = L.host_id
Group by H.host_id
Having count(L.id) >= (
Select max(counter)
From (
        Select count(L.id) as counter
        From Hosts H Inner Join Listings L on H.host_id = L.host_id
        Group by H.host_id
));
```

Result

host_id	h_name
4459553	Eva&Jacques

105 ms

4. Find the 5 most cheapest Apartments (based on average price within the available dates) in Berlin availablefor at least one daybetween01-03-2019 and 30-04-2019 having at least 2 beds, a location review score of at least 8, flexible

cancellation, and listed by a host with a verifiable government id.

```sql
drop view if exists ids_of_listings_in_Berlin;
Create view ids_of_listings_in_Berlin as
Select L.id
From Listings L Inner Join Locations Loc on L.location_id = Loc.loc_id
        Inner Join Cities C on C.city_id = Loc.city_id
Where C.name like '%berlin%';

drop view if exists ids_of_listings_with_at_least_2_beds;
Create view ids_of_listings_with_at_least_2_beds as
Select L.id
From Listings L Inner Join Accommodations A on L.acc_id = A.acc_id
        Inner Join Bedrooms B on B.bedroom_id = A.bedroom_id
Where B.number_beds > 1;

drop view if exists ids_of_listings_with_at_least_8_loc_rev_score;
Create view ids_of_listings_with_at_least_8_loc_rev_score as
Select L.id
From Listings L Inner Join Review_scores Rev on L.review_scores_id = Rev.review_scores_id
Where Rev.review_scores_location > 7;

drop view if exists ids_of_listings_with_flexible_cancellation;
Create view ids_of_listings_with_flexible_cancellation as
Select L.id
From Listings L Inner Join Rules_listing Ru on L.rules_list_id = Ru.rules_list_id
        Inner Join Cancellation_policies CP on Ru.canc_policy_id = CP.canc_policy_id
Where CP.value = 'flexible';

drop view if exists ids_of_hosts_with_verif_gov_id;
Create view ids_of_hosts_with_verif_gov_id as
Select H.host_id
From Hosts H Inner Join Host_verifications HV on H.host_id = HV.host_id
        Inner Join Verifications V on V.verif_id = HV.verif_id
Where V.verif_value = 'government_id';

drop view if exists ids_of_listings_with_verif_gov_host;
Create view ids_of_listings_with_verif_gov_host as
Select L.id
From Listings L Inner Join Hosts H on L.host_id = H.host_id;

drop view if exists intersection_of_view_query_4;
Create view intersection_of_view_query_4 as
Select *
From ids_of_listings_in_Berlin
Intersect
Select *
From ids_of_listings_with_at_least_2_beds
Intersect
Select *
From ids_of_listings_with_at_least_8_loc_rev_score
Intersect
Select *
From ids_of_listings_with_flexible_cancellation
Intersect
Select *
From ids_of_listings_with_verif_gov_host;

Select * from
(
        Select D.list_id, avg(price) as average_price
        From Dates_clustered D
        Where D.date >= '2019-03-01' and D.date <= '2019-04-30' and D.availability = 1
        and D.list_id in intersection_of_view_query_4
        Group by D.list_id
)
Order by average_price
Limit 5;
```

Result

list_id	average_price
16307669	19.6481481481481
1490274	20.0
27494978	20.0
28406345	20.0
24043706	21.0655737704918

1931 ms

5. Each property can accommodate different number of people (1 to 16). Find the top-5 rated (review_score_rating) listings for each distinct category based on number of accommodated guests with at least two of these facilities: Wifi, Internet, TV,and Free street parking.

```
drop view if exists amenities_query_5;
CREATE VIEW amenities_query_5 as
select amenity_id
from Amenities
where value = "Wifi" or value = "TV" or value = "Internet" or value = "Free street parking";

drop view if exists listings_filtered_query_5;
CREATE VIEW listings_filtered_query_5 as
Select L.id, R.review_scores_rating, A.accommodates
from Listings L Inner Join Accommodations A on L.acc_id = A.acc_id
Inner Join Review_scores R on R.review_scores_id = L.review_scores_id
Inner Join Has_amenities H_a on H_a.acc_id = A.acc_id
Inner Join Amenities A on A.amenity_id = H_a.amenity_id
where A.amenity_id in amenities_query_5
group by L.id
Having count(*) > 1
order by R.review_scores_rating desc;

Select accommodates, id, review_scores_rating
from (Select accommodates, id, review_scores_rating, Row_Number()
        OVER(PARTITION BY L.accommodates ORDER BY review_scores_rating DESC) as Row_Number
        FROM listings_filtered_query_5 L) L2 where Row_Number <= 5;
```

Result

accommodates	id	review_scores_rating
1	109369	100
1	179488	100
1	240735	100
1	250121	100
1	287660	100

160 ms

6. What are top three busiest listings per host? The more reviews a listing has, the busier the listing is.

```
DROP VIEW IF EXISTS listing_id_number_reviews;
Create view listing_id_number_reviews as
Select L.id, L.host_id, count(R.review_id) as busy_count
From Reviews R Inner Join Listings L on R.list_id = L.id
Group By L.id;

SELECT host_id, id
FROM(SELECT host_id, id, Row_Number()
    OVER(PARTITION BY L.host_id ORDER BY L.busy_count DESC) AS Row_Number
    From listing_id_number_reviews L) L2 Where Row_Number <= 3;
```

Result

host_id	id
2217	2015
2217	21315310
2217	18773184
3073	6287375
3718	3176

1049 ms

7. What are the three most frequently used amenities at each neighborhood in Berlin for the listings with “Private Room” room type?

```
DROP VIEW IF EXISTS Neighbourhood_In_Berlin_Private_Room;
CREATE VIEW Neighbourhood_In_Berlin_Private_Room AS
SELECT N.name, Am.value
From Listings Li Inner Join Locations Loc on Li.location_id = Loc.loc_id Inner Join
Neighbourhood_descriptions ND on Loc.descr_id = ND.descr_id Inner Join Neighbourhoods N
on ND.neighb_id = N.neighb_id Inner Join Accommodations A on A.acc_id = Li.acc_id Inner Join
Has_amenities H on H.acc_id = A.acc_id Inner JOIN Amenities Am on Am.amenity_id = H.amenity_id
Inner Join Cities C on C.city_id = Loc.city_id Inner Join Room_types R on R.room_type_id = A.room_type_id
WHERE R.value='Private room' and LOWER(C.name) like '%berlin%';

DROP VIEW IF EXISTS Neighbourhood_In_Berlin_Private_Room2;
CREATE VIEW Neighbourhood_In_Berlin_Private_Room2 AS
SELECT N.name, N.value, Count(*) as count
FROM Neighbourhood_In_Berlin_Private_Room N
GROUP BY N.name, N.value;

SELECT N2.value, N2.name
FROM(SELECT value, name, Row_Number()
    OVER(PARTITION BY N.name ORDER BY N.count DESC) AS Row_Number
    From Neighbourhood_In_Berlin_Private_Room2 N) N2 WHERE Row_Number <= 3;
```

Result

value	name
Essentials	Adlershof
Heating	Adlershof
Kitchen	Adlershof
Essentials	"Alt-Hohenschönhausen"
Wifi	"Alt-Hohenschönhausen"

176 ms

8. What is the difference in the average communication review score of the host who has the most diverse way of verifications and of the host who has the least diverse way of verifications. In case of a multiple number of the most or the least diverse verifyinghosts, pick a host one from the most and one from the least verifyinghosts.

```
drop view if exists query_8_host_verifications_filtered;
create view query_8_host_verifications_filtered as
Select H.host_id
from Hosts H, Listings L, Review_scores R
where H.host_id = L.host_id and L.review_scores_id = R.review_scores_id and R.review_scores_communication not n
ull;

drop view if exists query_8_host_least_verifications;
create view query_8_host_least_verifications as
select host_id
from query_8_host_verifications_filtered
where host_id not in (select host_id
        from Host_verifications)
limit 1;

drop view if exists query_8_host_most_verifications;
create view query_8_host_most_verifications as
select host_id
from Host_verifications
where host_id in query_8_host_verifications_filtered
group by host_id
order by COUNT(*) desc
limit 1;

SELECT
        (select AVG(R.review_scores_communication)
        from query_8_host_most_verifications H, Listings L, Review_scores R
        where H.host_id = L.Host_id and L.review_scores_id = R.review_scores_id)
        -
        (select AVG(R.review_scores_communication)
        from query_8_host_least_verifications H, Listings L, Review_scores R
        where H.host_id = L.Host_id and L.review_scores_id = R.review_scores_id);
```

Result

0.0

127 ms

9. What is the city who has the highest number of reviews for the room types whose average number of accommodates are greater than 3

```
drop view if exists ids_of_listings_and_loc_with_room_type_hv_acc_gt_3;
Create view ids_of_listings_and_loc_with_room_type_hv_acc_gt_3 as
Select id, location_id
From Listings L Inner Join Accommodations A on L.acc_id = A.acc_id
where A.room_type_id
in (Select * from
(
Select Rt.room_type_id
From Accommodations A2 Inner Join Room_types RT on A2.room_type_id = RT.room_type_id
Group by Rt.room_type_id
Having avg(A2.accommodates) > 3
));
```

```
Select city_name, rev_count
From
(
Select C.name as city_name, count(R.rev_id) as rev_count
From ids_of_listings_and_loc_with_room_type_hv_acc_gt_3 V Inner Join
Locations L on V.location_id = L.loc_id Inner Join
Cities C on L.city_id = C.city_id Inner Join
Reviews R on V.id = R.list_id
Group by C.name
)
Order By rev_count
Limit 1;
```

Result

city_name	rev_count
"SavignyPlatz (Charlottenburg)"	1

3029 ms

10. Print all the neighborhoods in Madrid which have at least 50 percent of their listings occupied in year 2019 and their host has joined airbnb before 01.06.20171

```
drop view if exists ids_and_loc_id_of_listings_in_Madrid_with_host_2017_06_01;
Create view ids_and_loc_id_of_listings_in_Madrid_with_host_2017_06_01 as
Select L.id, L.location_id
From Listings L Inner Join Locations Loc on L.location_id = Loc.loc_id
        Inner Join Cities C on C.city_id = Loc.city_id Inner Join Hosts H
        on H.host_id = L.host_id
Where lower(C.name) like '%madrid%' and H.h_since < '2017-06-01';

drop view if exists listings_occupied_query_10;
create view listings_occupied_query_10 as
Select L.id
from ids_and_loc_id_of_listings_in_Madrid_with_host_2017_06_01 L inner join Dates D on L.id = D.list_id
where D.date >= "2019-01-01" and D.date <= "2019-12-31" and D.availability = 0;

drop view if exists all_listings_query_10;
create view all_listings_query_10 as
Select L.id
from ids_and_loc_id_of_listings_in_Madrid_with_host_2017_06_01 L inner join Dates D on L.id = D.list_id
where D.date >= "2019-01-01" and D.date <= "2019-12-31";

drop view if exists number_of_neighbourhood_occupied_query_10;
create view number_of_neighbourhood_occupied_query_10 as
Select N.neighb_id, COUNT(L.id) as counter
From ids_and_loc_id_of_listings_in_Madrid_with_host_2017_06_01 L inner join listings_occupied_query_10 L2 on L.
id = L2.id Inner Join
        Locations       Loc on L.location_id = Loc.loc_id Inner Join Neighbourhood_descriptions ND
        on Loc.descr_id = ND.descr_id Inner Join Neighbourhoods N on ND.neighb_id = N.neighb_id
Group by N.neighb_id;

drop view if exists number_of_neighbourhood_query_10;
create view number_of_neighbourhood_query_10 as
Select N.neighb_id, COUNT(L.id) as counter
From ids_and_loc_id_of_listings_in_Madrid_with_host_2017_06_01 L inner join listings_occupied_query_10 L2 on L.
id = L2.id Inner Join
        Locations       Loc on L.location_id = Loc.loc_id Inner Join Neighbourhood_descriptions ND
        on Loc.descr_id = ND.descr_id Inner Join Neighbourhoods N on ND.neighb_id = N.neighb_id
Group by N.neighb_id;

Select N.neighb_id
From number_of_neighbourhood_query_10 N Inner join number_of_neighbourhood_occupied_query_10 N1
on N.neighb_id = N1.neighb_id
Where N1.counter > N.counter / 2;
```

Result

neighb_id

0

1

3

4

8

9

8718 ms

11. Print all the countries that in 2018 had at least 20% of their listings available.

```
drop view if exists query_11_available_listings_2018;
create view query_11_available_listings_2018 as
Select distinct L.id, L.location_id
from Listings L inner join Dates_clustered D on L.id = D.list_id
where D.date >= "2018-01-01" and D.date <= "2018-12-31" and D.availability = 1;

Select C.country_name
from Countries C
Where (Select COUNT(L.id)
from query_11_available_listings_2018 L inner join Locations Loc on L.location_id = Loc.loc_id inner join Count
ries C on Loc.country_id = C.country_id) >= (Select COUNT(L.id)
        from Listings L inner join Locations Loc on L.location_id = Loc.loc_id inner join Countries C on Loc.co
untry_id = C.country_id)/5;
```

Result

country_name

Germany

Spain

2535 ms

12. Print all the neighborhouds in Barcelona where more than 5 percent of their accommodation's cancelation policy is strict with grace period.

```
Drop view if exists ids_of_listings_with_strict_grace;
Create view ids_of_listings_with_strict_grace as
Select id, location_id
From Listings L Inner Join Rules_listing RL on L.rules_list_id = RL.rules_list_id
Inner Join Cancellation_policies CP on CP.canc_policy_id = RL.canc_policy_id
Where CP.value = 'strict_14_with_grace_period';

drop view if exists number_of_listing_per_neighbourhood_barcelona;
create view number_of_listing_per_neighbourhood_barcelona as
Select N.neighb_id, COUNT(L.id) as counter
From Listings L Inner Join Locations    Loc on L.location_id = Loc.loc_id
Inner Join Neighbourhood_descriptions ND on Loc.descr_id = ND.descr_id
Inner Join Neighbourhoods N on ND.neighb_id = N.neighb_id
Inner Join Cities C on Loc.city_id = C.city_id
Where lower(C.name) like '%barcelona%'
Group by N.neighb_id;
```

```
drop view if exists number_of_listing_per_neighbourhood_barcelona_with_grace_period;
create view number_of_listing_per_neighbourhood_barcelona_with_grace_period as
Select N.neighb_id, COUNT(L.id) as counter
From ids_of_listings_with_strict_grace L Inner Join Locations   Loc on L.location_id = Loc.loc_id
Inner Join Neighbourhood_descriptions ND on Loc.descr_id = ND.descr_id
Inner Join Neighbourhoods N on ND.neighb_id = N.neighb_id
Inner Join Cities C on Loc.city_id = C.city_id
Where lower(C.name) like '%barcelona%'
Group by N.neighb_id;

Select T1.neighb_id
From number_of_listing_per_neighbourhood_barcelona T1 Inner Join
number_of_listing_per_neighbourhood_barcelona_with_grace_period T2 on T1.neighb_id = T2.neighb_id
Where 20*T2.counter >= T1.counter;
```

Result

neighb_id
24
25
27
37
42

75 ms

Optimization of Queries using Indexes

Query 4

This query took 1931 ms and is rather complex because of the views, however the query the final executed query is:

```
Select * from
(
        Select D.list_id, avg(price) as average_price
        From Dates D
        Where D.date >= '2019-03-01' and D.date <= '2019-04-30' and D.availability = 1
        and D.list_id in intersection_of_view_query_4
        Group by D.list_id
)
Order by average_price
Limit 5;
```

So, there is a range search on the dates. Knowing that the Dates table is enormous (more that 15 million dates), this range has a really small selectivity. We thought it could be interesting to index this query.
The query had the following query plan:

```
QUERY PLAN
|--CO-ROUTINE 2
|  |--SCAN TABLE Dates AS D
|  |--LIST SUBQUERY 1
|  |  |--CO-ROUTINE 5
|  |  |  `--COMPOUND QUERY
|  |  |     |--LEFT-MOST SUBQUERY
|  |  |     |  |--SCAN TABLE Listings AS L
|  |  |     |  |--SEARCH TABLE Locations AS Loc USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |  `--SEARCH TABLE Cities AS C USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |--INTERSECT USING TEMP B-TREE
|  |  |     |  |--SCAN TABLE Listings AS L
|  |  |     |  |--SEARCH TABLE Accommodations AS A USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |  `--SEARCH TABLE Bedrooms AS B USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |--INTERSECT USING TEMP B-TREE
|  |  |     |  |--SCAN TABLE Listings AS L
|  |  |     |  `--SEARCH TABLE Review_scores AS Rev USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |--INTERSECT USING TEMP B-TREE
|  |  |     |  |--SEARCH TABLE Cancellation_policies AS CP USING COVERING INDEX sqlite_autoindex_Cancellation_policies_1 (value=?)
|  |  |     |  |--SCAN TABLE Listings AS L
|  |  |     |  `--SEARCH TABLE Rules_listing AS Ru USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     `--INTERSECT USING TEMP B-TREE
|  |  |        |--SCAN TABLE Listings AS L
|  |  |        `--SEARCH TABLE Hosts AS H USING INTEGER PRIMARY KEY (rowid=?)
|  |  `--SCAN SUBQUERY 5
|  `--USE TEMP B-TREE FOR GROUP BY
|--SCAN SUBQUERY 2
`--USE TEMP B-TREE FOR ORDER BY
```

And indeed, sql starts by looking at the Dates first and then everything is based on this. Thus trying to reduce the size of the number of Dates sql reads may cut down drastically the runtime. Moreover, sql does a full scan of the table which is really the worst scenario when you have a big table.
Because of the range search we, of course, tried using a clustered index on `date`.
In sqlite, we unfortunately cannot create a clustered index because sqlite uses the ROWID and doesn't support sorting the table differently. However, we can create a Table without ROWID which essentially creates a clustered index on the primary key of the table. *More info at https://sqlite.org/withoutrowid.html*
Thus we changed the Dates table to Dates_clustered and it now has the following create statement:

```
CREATE TABLE Dates_clustered(date_id INTEGER NOT NULL, list_id INTEGER NOT NULL, date DATE, availability INTEGE
R, price FLOAT, primary key(date, date_id), foreign key(list_id) references Listings) without ROWID;
```

The table doesn't have a ROWID and we add the `date` attribute to the primary key such that the Dates are first sorted on `date`, extending the primary key doesn't change the uniqueness of the key.
Now, the same query but with Dates_clustered instead of Dates (which is essentially the same table, we deleted the Dates table) gives us the following query plan:

```
QUERY PLAN
|--CO-ROUTINE 2
|  |--SEARCH TABLE Dates_clustered AS D USING PRIMARY KEY (date>? AND date<?)
|  |--LIST SUBQUERY 1
|  |  |--CO-ROUTINE 5
|  |  |  `--COMPOUND QUERY
|  |  |     |--LEFT-MOST SUBQUERY
|  |  |     |  |--SCAN TABLE Listings AS L
|  |  |     |  |--SEARCH TABLE Locations AS Loc USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |  `--SEARCH TABLE Cities AS C USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |--INTERSECT USING TEMP B-TREE
|  |  |     |  |--SCAN TABLE Accommodations AS A USING INDEX sqlite_autoindex_Accommodations_2
|  |  |     |  |--SEARCH TABLE Bedrooms AS B USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |  `--SEARCH TABLE Listings AS L USING AUTOMATIC COVERING INDEX (acc_id=?)
|  |  |     |--INTERSECT USING TEMP B-TREE
|  |  |     |  |--SCAN TABLE Listings AS L
|  |  |     |  `--SEARCH TABLE Review_scores AS Rev USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     |--INTERSECT USING TEMP B-TREE
|  |  |     |  |--SEARCH TABLE Cancellation_policies AS CP USING COVERING INDEX sqlite_autoindex_Cancellation_policies_1 (value=?)
|  |  |     |  |--SCAN TABLE Listings AS L
|  |  |     |  `--SEARCH TABLE Rules_listing AS Ru USING INTEGER PRIMARY KEY (rowid=?)
|  |  |     `--INTERSECT USING TEMP B-TREE
|  |  |        |--SCAN TABLE Listings AS L
|  |  |        `--SEARCH TABLE Hosts AS H USING INTEGER PRIMARY KEY (rowid=?)
|  |  `--SCAN SUBQUERY 5
|  `--USE TEMP B-TREE FOR GROUP BY
|--SCAN SUBQUERY 2
`--USE TEMP B-TREE FOR ORDER BY
```

As we can see, sql now searches the Dates instead of scanning it, using the primary key `date` which is exactly what we wanted.
The execution time is now 428 ms!
We went from 1931 ms to 428 ms, which is a ~4.5 speedup.

Query 11

This query took 2535 ms and has the following code:

```
drop view if exists query_11_available_listings_2018;
create view query_11_available_listings_2018 as
Select distinct L.id, L.location_id
from Listings L inner join Dates_clustered D on L.id = D.list_id
where D.date >= "2018-01-01" and D.date <= "2018-12-31" and D.availability = 1;

Select C.country_name
from Countries C
Where (Select COUNT(L.id)
from query_11_available_listings_2018 L inner join Locations Loc on L.location_id = Loc.loc_id inner join Count
ries C on Loc.country_id = C.country_id) >= (Select COUNT(L.id)
         from Listings L inner join Locations Loc on L.location_id = Loc.loc_id inner join Countries C on Loc.co
untry_id = C.country_id)/5;
```

As we can see there once again is filter based on the range of the dates. It's a bigger ranger, all of the year 2018, but in our data most of the dates are in 2019 so it still is a big filter. Here is its query plan:

```
QUERY PLAN
|--SCAN TABLE Countries AS C
|--SCALAR SUBQUERY 1
|  |--MATERIALIZE 1
|  |  |--SCAN TABLE Listings AS L
|  |  `--SEARCH TABLE Dates AS D USING AUTOMATIC PARTIAL COVERING INDEX (availability=? AND list_id=?)
|  |--SCAN TABLE Locations AS Loc
|  |--SEARCH TABLE Countries AS C USING INTEGER PRIMARY KEY (rowid=?)
|  `--SEARCH SUBQUERY 1 AS L USING AUTOMATIC COVERING INDEX (location_id=?)
`--SCALAR SUBQUERY 2
   |--SCAN TABLE Listings AS L
   |--SEARCH TABLE Locations AS Loc USING INTEGER PRIMARY KEY (rowid=?)
   `--SEARCH TABLE Countries AS C USING INTEGER PRIMARY KEY (rowid=?)
```

Sql uses some automatic index on availability and list_id, trying to optimize the Dates read but the query is still pretty slow and we thought the Dates probably was the biggest bottleneck here. So this automatic index probably isn't the best.
We followed the same reasoning as for query 4, we wanted to sort the Dates such that it only checks the Dates that are in 2018, while never looking at any 2019 date. We once again need a clustered index. Since we already have a clustered index on dates we used the same one as for query 4 to optimize this one. Which gives us the following query plan:

```
QUERY PLAN
|--SCAN TABLE Countries AS C
|--SCALAR SUBQUERY 1
|  |--MATERIALIZE 1
|  |  |--SEARCH TABLE Dates_clustered AS D USING PRIMARY KEY (date>? AND date<?)
|  |  |--SEARCH TABLE Listings AS L USING INTEGER PRIMARY KEY (rowid=?)
|  |  `--USE TEMP B-TREE FOR DISTINCT
|  |--SCAN SUBQUERY 1 AS L
|  |--SEARCH TABLE Locations AS Loc USING INTEGER PRIMARY KEY (rowid=?)
|  `--SEARCH TABLE Countries AS C USING INTEGER PRIMARY KEY (rowid=?)
`--SCALAR SUBQUERY 2
   |--SCAN TABLE Listings AS L
   |--SEARCH TABLE Locations AS Loc USING INTEGER PRIMARY KEY (rowid=?)
   `--SEARCH TABLE Countries AS C USING INTEGER PRIMARY KEY (rowid=?)
```

Sql now uses our index on date, great! But what about the runtime?
It previously took 2535 ms to execute the query and now it only takes 423 ms, almost a x6 speedup!

Query 1

This query took 61 ms to execute, so it clearly isn't the longest running query but we thought it could be interesting to optimize a smaller one. This is its code:

```
Select C.name, count(C.name)
From Listings L Inner Join Accommodations A on A.acc_id = L.acc_id
Inner Join Locations Loc on Loc.loc_id = L.location_id
        Inner Join Cities C on C.city_id = Loc.city_id
        Where square_feet is not null
        Group by C.name
        Order by C.name;
```

Here is the query plan:

```
QUERY PLAN
|--SCAN TABLE Listings AS L
|--SEARCH TABLE Accommodations AS A USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE Locations AS Loc USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE Cities AS C USING INTEGER PRIMARY KEY (rowid=?)
`--USE TEMP B-TREE FOR GROUP BY
```

Due to inner join, sql searches with rowid
We noticed that the output contains very few results -> there are very few people who declared the area of their property in square feet. So maybe we can use this information and create an index on square_feet to access Accommodations with no square_feet more quickly. We tried using a unclustered index but it didn't improve run time, we then wanted to try a clustered index on square_feet. Because of sqlite we have to put the clustered indexed attribute in the primary key but right now square_feet is nullable and thus cannot be a primary key. We still wanted to try this idea, so we changed the Accommodations table such that square_feet is nullable and we assume that `square_feet =0 => square_feet=NULL`. We now can create a clustered index on square_feet. Here is the new declaration of the table:

```
CREATE TABLE Accommodations(acc_id INTEGER NOT NULL, access VARCHAR(100), property_type_id INTEGER NOT NULL, ro
om_type_id INTEGER NOT NULL, accommodates INTEGER, bathrooms INTEGER, bedroom_id INTEGER NOT NULL, square_feet
INTEGER NOT NULL, primary key(square_feet, acc_id), foreign key(property_type_id) references Property_types, fo
reign key(room_type_id) references Room_types, foreign key(bedroom_id) references Bedrooms, unique(access, prop
erty_type_id, room_type_id, accommodates, bathrooms, bedroom_id, square_feet)) without ROWID;
```

In short, it no longer has ROWID, square_feet is part of the primary key and is now `NOT NULL`. But how is the query plan now?

```
QUERY PLAN
|--SEARCH TABLE Accommodations AS A USING PRIMARY KEY (square_feet>?)
|--SEARCH TABLE Listings AS L USING AUTOMATIC COVERING INDEX (acc_id=?)
|--SEARCH TABLE Locations AS Loc USING INTEGER PRIMARY KEY (rowid=?)
|--SEARCH TABLE Cities AS C USING INTEGER PRIMARY KEY (rowid=?)
`--USE TEMP B-TREE FOR GROUP BY
```

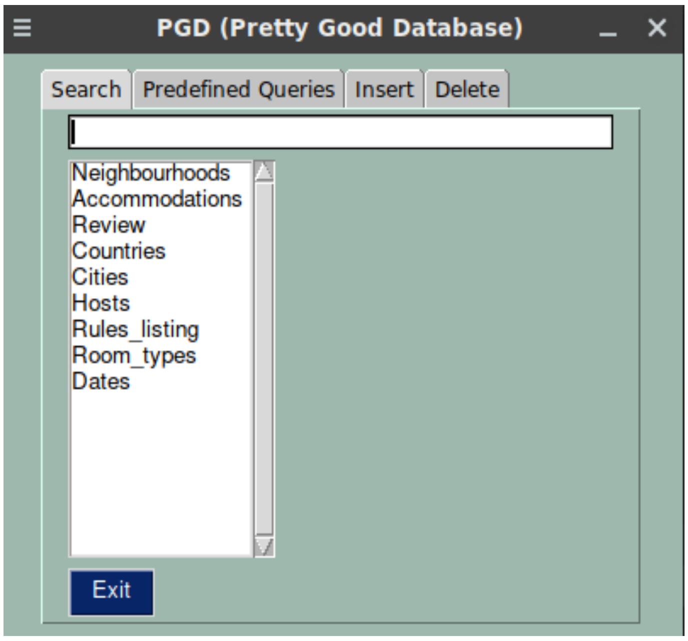
Great, sql uses it to find square_feet > 0, i.e. square_feet that weren't NULL. But what about the run time? We went from 61 ms to 32 ms -> ~2x speedup. This isn't as good but this was to be expected because the run time is smaller thus it's less consistent relatively and the index has less of an impact because the tables are smaller.

Graphical Interface

Concerning the graphical interface, it is divided into 4 tabs : Search, Predefined Queries, Insertion and Deletion. The user can navigate through the tabs in order to interact with the tables in the database, i.e either modify them or just look something up.

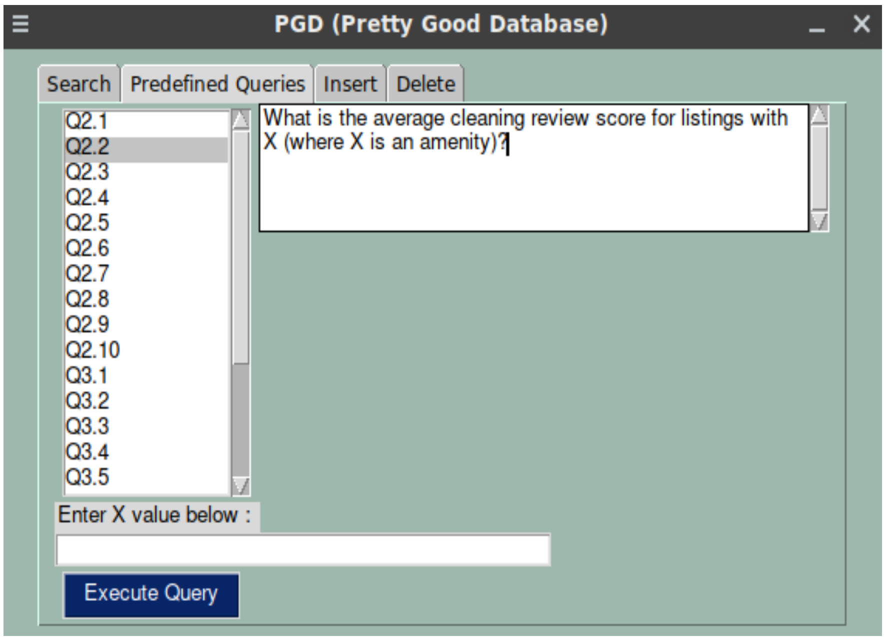
Search

In the search tab the user can choose a table and enter a value. It will search among every columns (element of the schema) in the table and print all the lines that have at least one element that matches the searched pattern.



Predefined Queries with Parameters

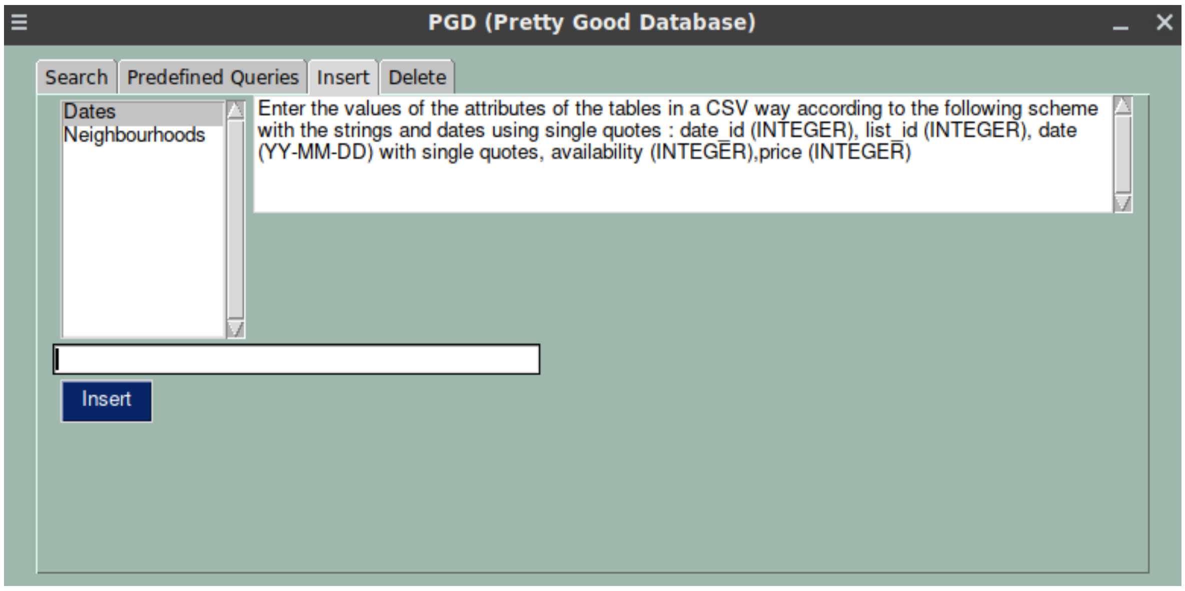
Here the user can choose among multiple predefined queries and tweak some of them. For instance, for the Q2.1, one can select the number of bedrooms for the average that the query computes.



What is the average cleaning review score for listings with X (where X is an amenity)?

Insertion

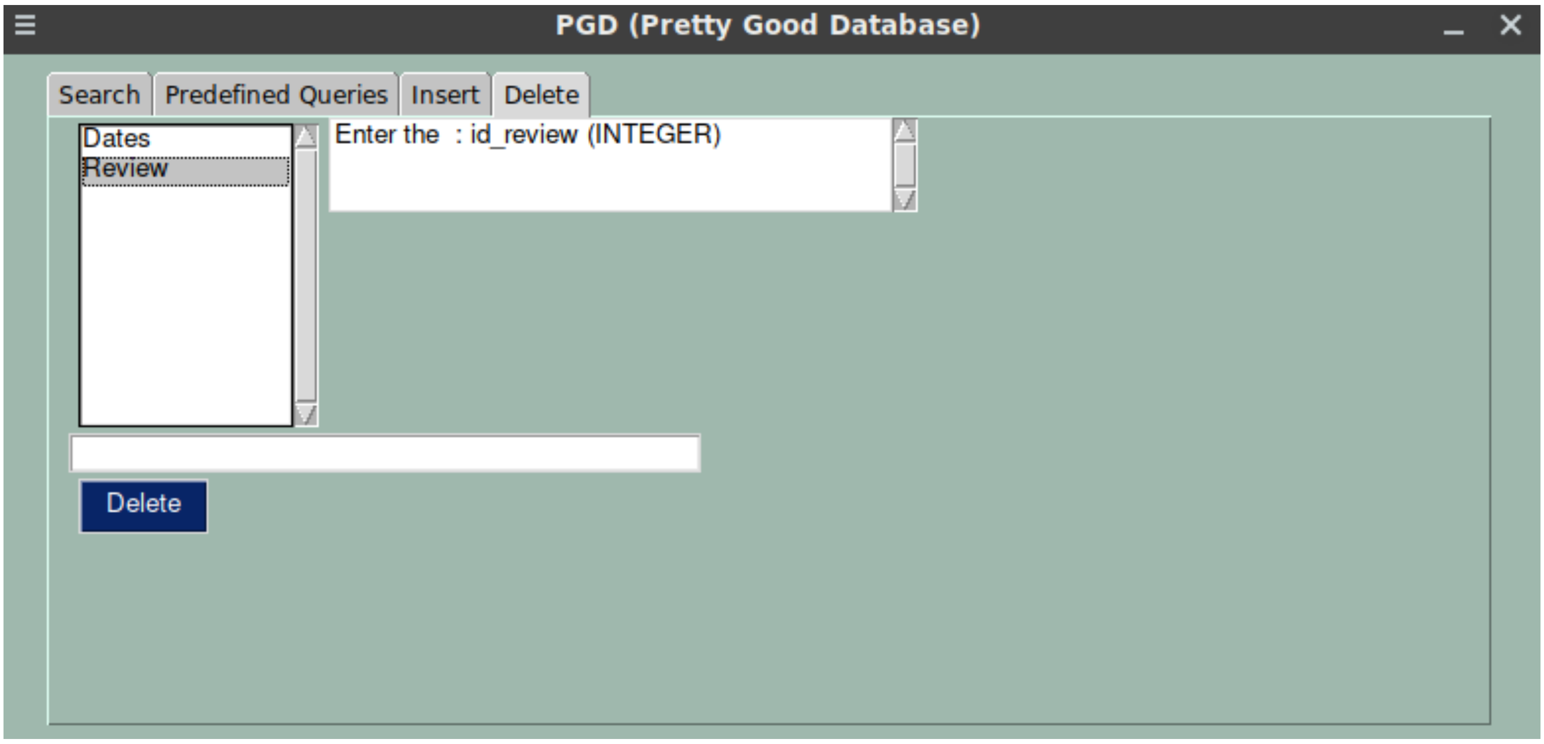
In the insertion tab, one can select a table and insert an element following the syntax requirements on the right.



Enter the values of the attributes of the tables in a CSV way according to the following scheme with the strings and dates using single quotes : date_id (INTEGER), list_id (INTEGER), date (YY-MM-DD) with single quotes, availability (INTEGER),price (INTEGER)

Deletion

In the deletion tab, one can select a table and delete an element according to its primary key (id).



Enter the : id_review (INTEGER)