Databases Project – Spring 2019

Team No: 6

Names: Maxime Sierro, Michaël Spierer, Guillaume Vray

Contents

Contents 1

Deliverable 1 2

Assumptions 2

Entity Relationship Schema 2

Schema 2

Description 2

Relational Schema 2

ER schema to Relational schema 2

DDL 3

General Comments 3

Deliverable 2 4

Assumptions 4

Data Loading 4

Query Implementation 4

Query a: 4

Description of logic: 4

SQL statement 4

Interface 4

Design logic Description 4

Screenshots 4

General Comments 4

Deliverable 3 5

Assumptions 5

Query Implementation 5

Query a: 5

Description of logic: 5

SQL statement 5

Query Analysis 5

Selected Queries (and why) 5

Query 1 5

Query 2 5

Query 3 5

Interface 6

Design logic Description 6

Screenshots 6

General Comments 6

# 

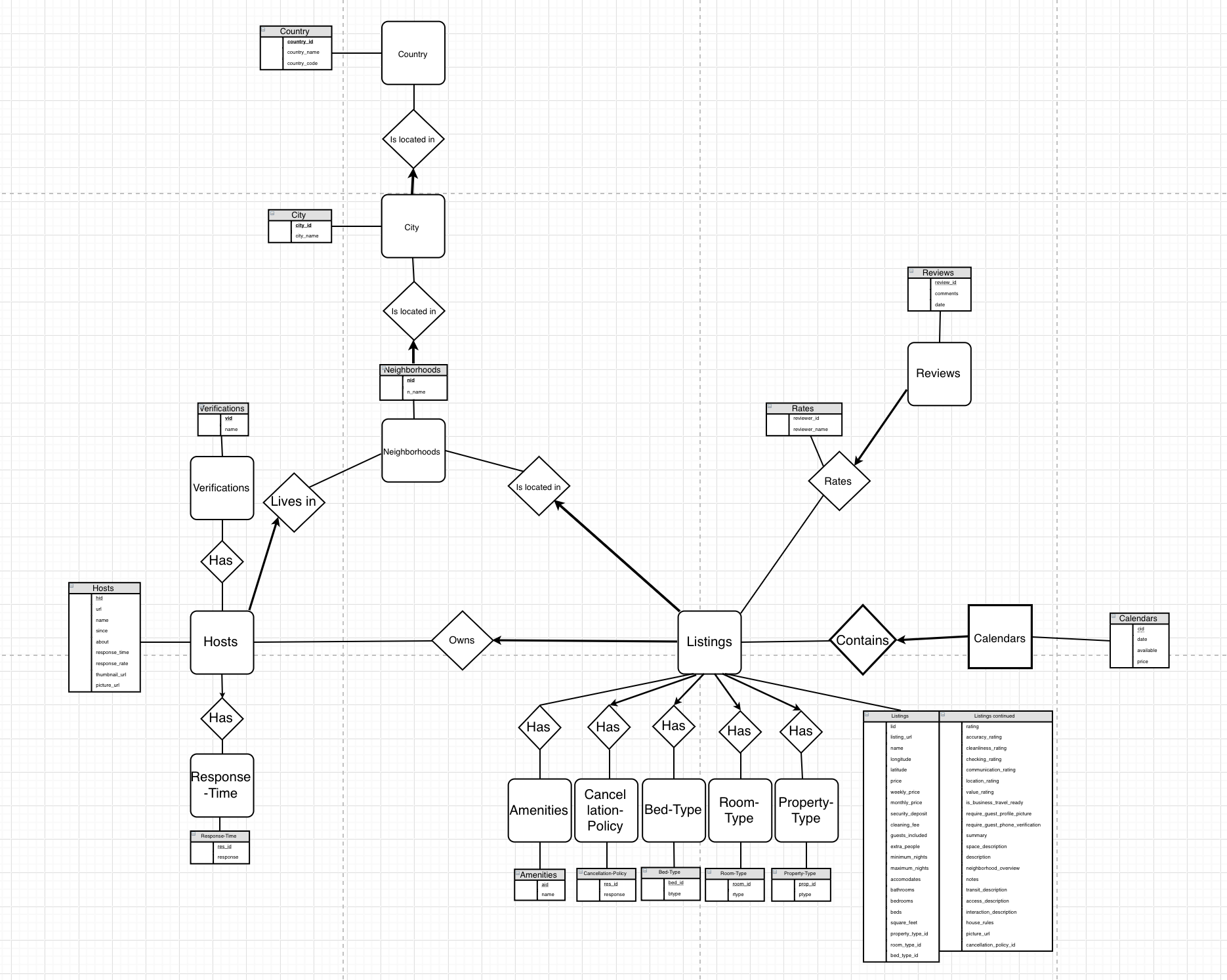
# Deliverable 1

## Assumptions

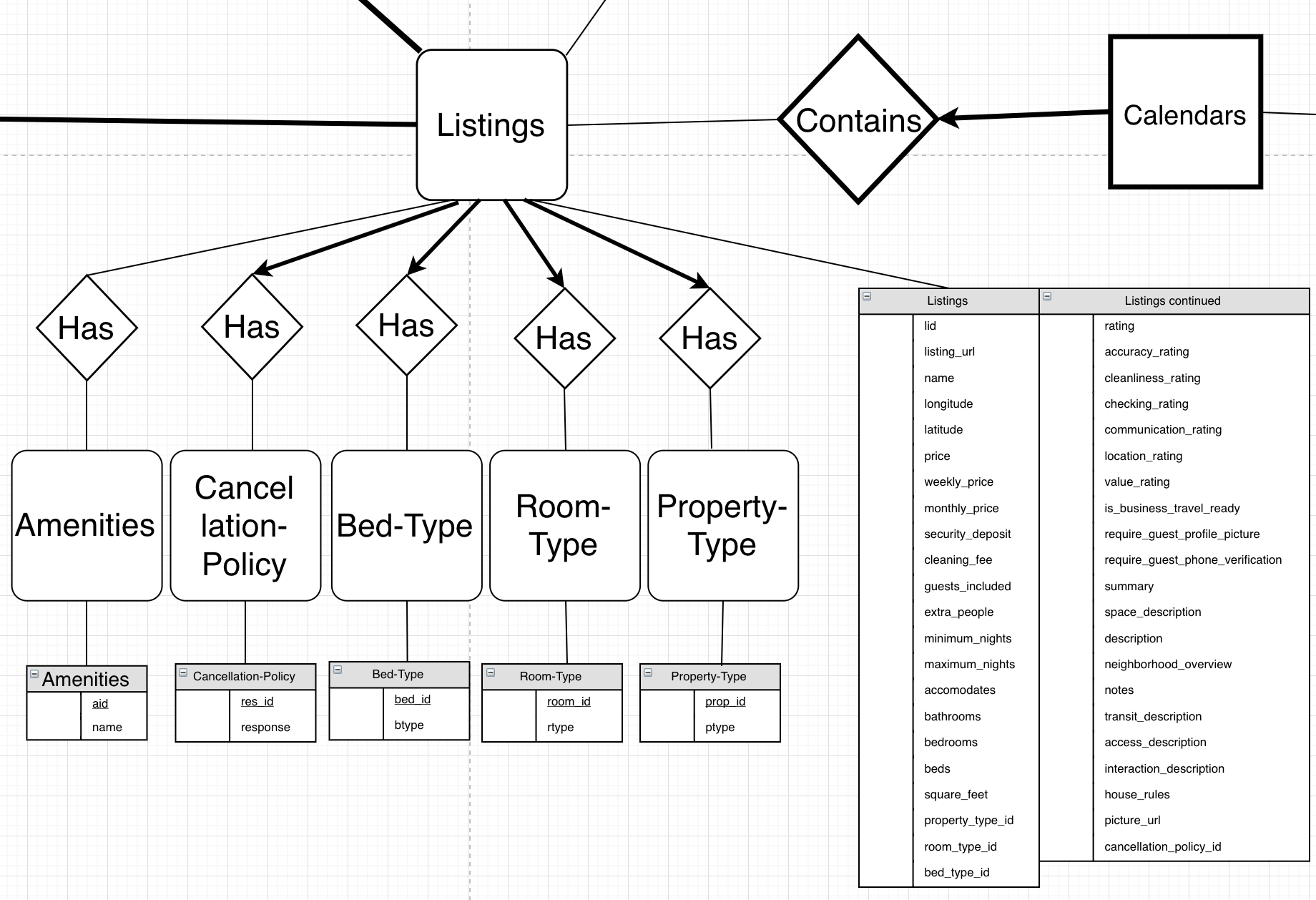
1. Each listing is owned by a unique host
2. The descriptions are not longer than 2048 characters
3. Hosts and listings are both in a unique neighborhood, the host of a listing is not necessarily in the same neighborhood as the listing.
4. No neighborhood in different countries have the same name
5. We can check the availability of a listing at each day of the year

## Entity Relationship Schema

### Schema



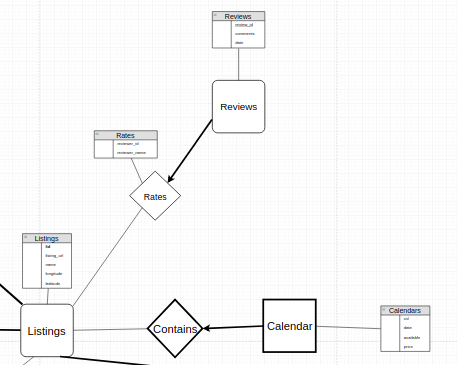
### Description



Let’s start by explaining how we organised the various characteristics listings have, which are all contained in the “listings” csv files.

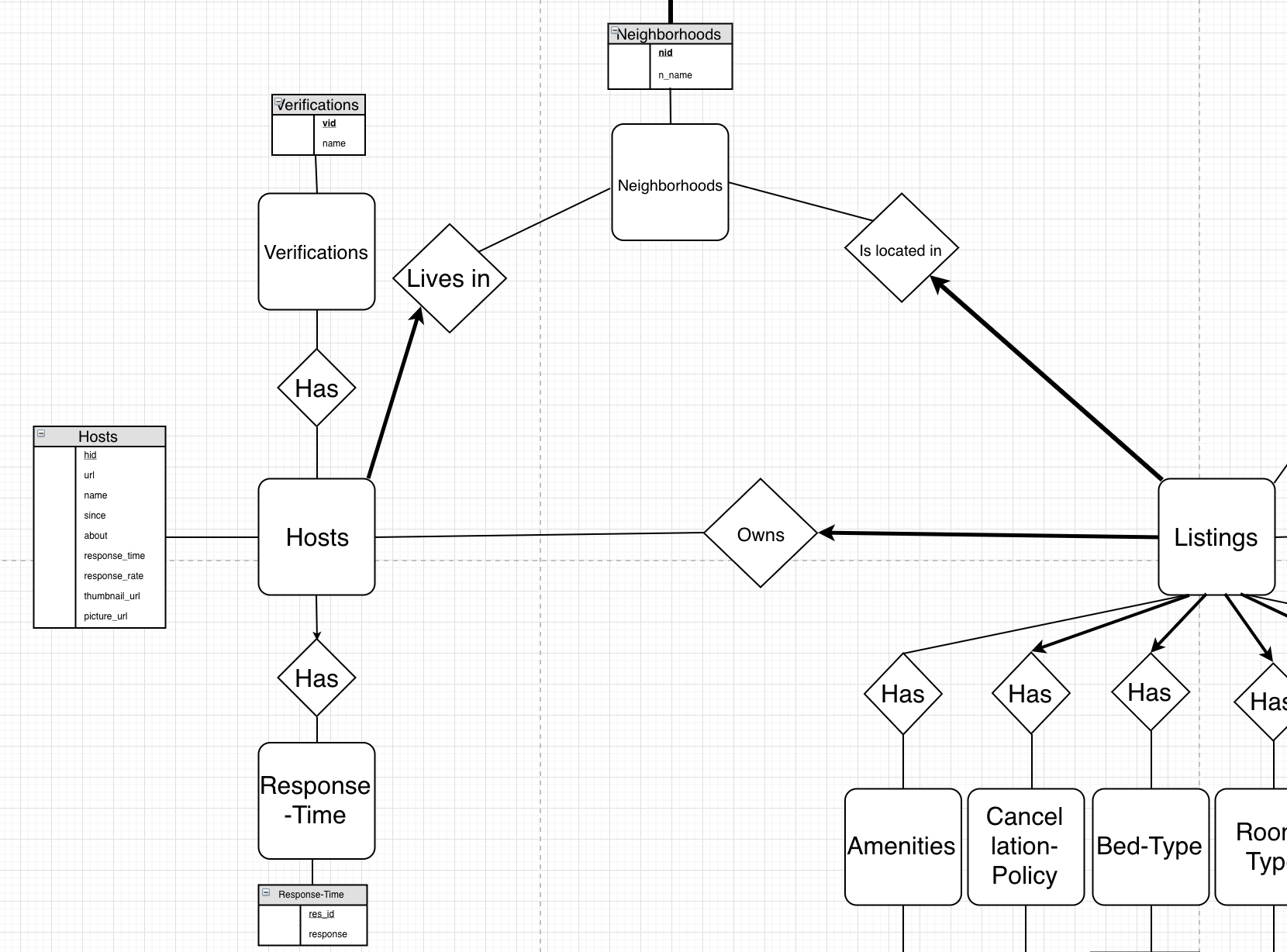
Each amenity is an entity. There are a finite number of different amenities, and they can be found in multiple listings. Since we’re going to be doing queries like “find all apartments which have Wi-Fi”, it makes sense to relate Listings and Amenities in a many-to-many fashion.

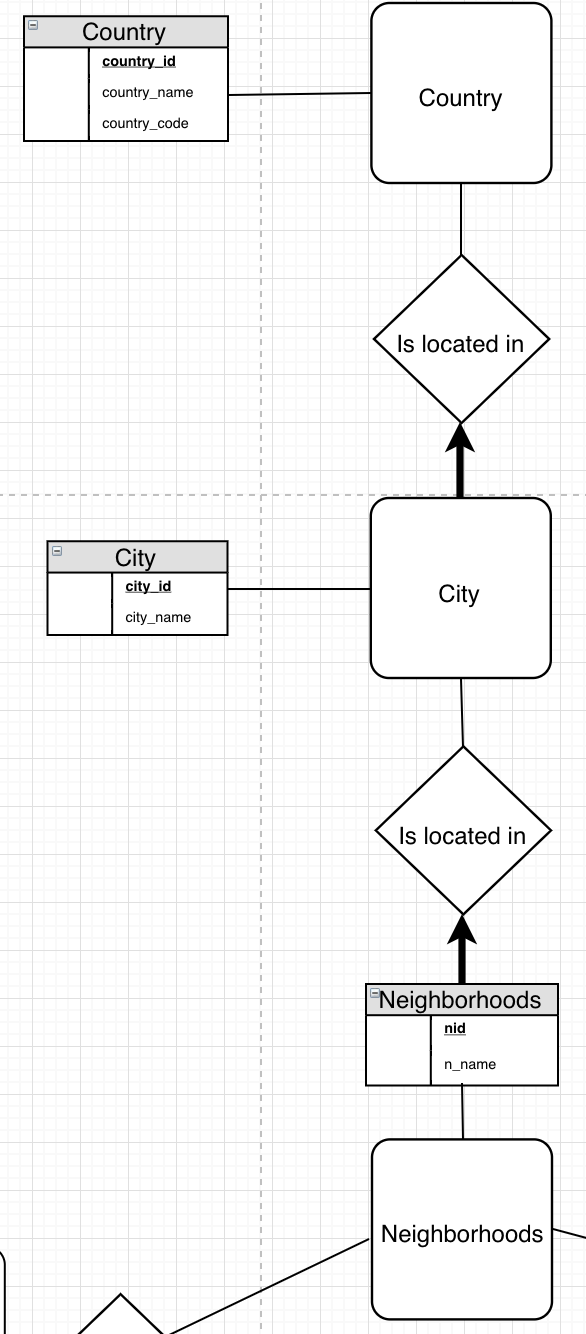
For Cancellation-Policy, Bed-Type, Room-Type and Property-Type the relationships are similar. Each type can be found in multiple listings just like for Amenities, however every listing needs to have one and only one of each of those 4 types.



The Calendar entity is simply a weak entity of listing. This time however it is many-to-one, since each listing can be available for multiple days. It has an id (cid) which needs to be used with lid to be identified. The other attributes are the date, the availability and the price.

The Reviews identity is different from Calendar because it actually has a primary key : review\_id. It’s related to listings in a many-to-one fashion again, since a listing could have multiple reviews. We chose to only include “comments” and “date” in the Reviews entity, and to include the characteristics related to the reviewer in the relationship “Rates” itself. We chose not to create a “Reviewers” entity because it is never needed for the queries of this project ; we don’t need to keep track of different reviewers like we do for hosts. The reviewer’s characteristics (name and id) could also simply be kept as attributes of reviews as well.





The Hosts identity has primary key “hid”, contains attributes related to the host (name, response\_time, picture\_url, etc). Its relationship with Listings, “Owns” is one-to-many since a host can have many listings. Listings has total participation in this relationship since we assume that a listing always has an owner.

The Verifications entity’s relationship with Hosts is analogous to the the relationship Amenities have with Listings ; there are a finite number of verifications and multiple hosts can have multiple verifications. Thus, it’s a many-to-many relationship and the only attribute of Verifications is its primary key “name”.

The host’s response time is not a number but a type that can be “N/A”, which is why Host does not have total participation in the relationship.

The “hierarchy” between Neighborhoods, City and Country is simple and intuitive. A neighborhood has to be located in a city, which in turn has to be located in a country.

There are some participations in relationships that have not been addressed yet. Naturally, all weak entities have total-participation in their relationship. There are also some interesting cases where both options (total participation or not) could be considered. It was often a matter of choosing between designing the schema with more of a theoretical and “open” approach versus a more practical and “closed” approach. We most often chose the former.

A good example would be the relationship between Hosts and Listings. In our project, since hosts are found in listings csv files, they will always have at least a listing. However, we did not considered hosts to be in total-participation in this relationship for our model, since we could imagine theoretical cases where a host deletes his last listing. While this case won’t happen in our project we chose to keep our design open. Similarly, there is no total-participation between Amenities and Listings, even if it’s extremely unlikely for a listing to have zero amenity, or for an amenity to be found in no listing. The main idea is to not force hard rules unless we’re absolutely sure of their necessity.

## 

## Relational Schema

### ER schema to Relational schema

We decided to use the following types for our database representation in Oracle (chosen with the help of Oracle’s Documentation). LONG -> NUMBER(19, 0) , INT -> NUMBER(10,0), DOUBLE -> FLOAT(24), STRING -> VARCHAR2(SIZE), BOOLEAN -> VARCHAR2(1).

**CREATE TABLE** Hosts (

hid **NUMBER**(19, 0),

neighborhood\_id **NUMBER**(19,0) **NOT NULL**,

response\_time\_id **NUMBER**(19,0),

url **VARCHAR2**(2048) **NOT NULL**,

name **VARCHAR2**(255) **NOT NULL**,

since **DATE NOT NULL**,

about **VARCHAR2**(2048),

response\_rate **NUMBER**(10, 0),

thumbnail\_url **VARCHAR2**(2048) **NOT NULL**,

picture\_url **VARCHAR2**(2048) **NOT NULL**,

**PRIMARY KEY** (hid),

**FOREIGN KEY** (neighborhood\_id) **REFERENCES** Neighborhoods(nid) **ON DELETE CASCADE**,

**FOREIGN KEY** (response\_time\_id) **REFERENCES** ResponseTime(resid) **ON DELETE SET NULL**

);

The Hosts table possesses attributes and four relations. One relationship is many to many with verifications, and the other three are one to many. A host lives in only one neighborhood but a neighborhood can have multiple hosts and a host possesses some Listings but a Listing is possessed only by one host. The last relation with Response-Time is also a one to many but the arrow is not in bold because a Host can have no response time.

For the many to many relationship, we created a new table called HostsHasVerifications and its primary key is composed of host\_id and the vid of the Verification.

**CREATE TABLE** HostsHasVerifications (

hid **NUMBER**(19, 0) **NOT NULL**,

vid **NUMBER**(19, 0) **NOT NULL**,

**PRIMARY KEY** (hid,vid),

**FOREIGN KEY** (hid)

**REFERENCES** Hosts(hid) **ON DELETE CASCADE**,

**FOREIGN KEY** (vid)

**REFERENCES** Verifications(vid) **ON DELETE CASCADE**

);

We described the Verifications entity and all the other entities with “predetermined” string (ResponseTime, Amenities, Cancellation-Policy, Bed-Type, Room-Type and Property-Type) like the following. Each one of this entities has unique PK and an attribute for the value. By doing this normalised schema, we eliminate redundancies.

**CREATE TABLE** Verifications (

vid **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

vname **VARCHAR2**(32) **UNIQUE NOT NULL**,

**PRIMARY KEY** (vid)

);

**CREATE TABLE** ResponseTime (

resid **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

response **VARCHAR2**(32) **UNIQUE NOT NULL**,

**PRIMARY KEY** (resid)

);

**CREATE TABLE** CancellationPolicy (

cp\_id **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

cancellation\_policy **VARCHAR2**(32) **UNIQUE NOT NULL**,

**PRIMARY KEY** (cp\_id)

);

**CREATE TABLE** BedType (

bed\_id **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

bed\_type **VARCHAR2**(32) **UNIQUE NOT NULL**,

**PRIMARY KEY** (bed\_id)

);

**CREATE TABLE** RoomType (

room\_id **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

room\_type **VARCHAR2**(32) **UNIQUE NOT NULL**,

**PRIMARY KEY** (room\_id)

);

**CREATE TABLE** PropertyType (

prop\_id **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

property\_type **VARCHAR2**(32) **UNIQUE NOT NULL**,

**PRIMARY KEY** (prop\_id)

);

**CREATE TABLE** Amenities (

aid **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

aname **VARCHAR2**(255) **UNIQUE NOT NULL**,

**PRIMARY KEY** (aid)

);

The one to many relationship between Host and the table Neighborhoods is different because we don’t need a new table to describe the relation. Hosts only have an attribute neighborhood\_id which corresponds to the primary key of the table Neighborhoods : ( the same explanation applies to the relation between Listings and Neighborhood).

A Neighborhoods is decomposed to have a city\_id corresponding to a City from the City table which has a country\_id in the Country Table.

**CREATE TABLE** Neighborhoods(

nid **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

city\_id **NUMBER**(19,0) **NOT NULL**,

neighborhood\_name **VARCHAR2**(32) **NOT NULL**,

**PRIMARY KEY** (nid),

**FOREIGN KEY** (city\_id) **REFERENCES** City(city\_id) **ON DELETE CASCADE**

);

**CREATE TABLE** City(

city\_id **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

country\_id **NUMBER**(19,0) **NOT NULL**,

city\_name **VARCHAR2**(32) **NOT NULL**,

**PRIMARY KEY** (city\_id),

**FOREIGN KEY** (country\_id) **REFERENCES** Country(country\_id) **ON DELETE CASCADE**

);

**CREATE TABLE** Country(

country\_id **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

country\_name **VARCHAR2**(32) **NOT NULL**,

country\_code **VARCHAR2**(32) **NOT NULL**,

**PRIMARY KEY** (country\_id)

);

For the one to many relation between Hosts and Listing where a host possesses some listings, we decided to have a host\_id attribute in the Listing table which references the hid from the Hosts table.

**CREATE TABLE** Listings (

lid **NUMBER**(19, 0),

host\_id **NUMBER**(19,0) **NOT NULL** ,

neighborhood\_id **NUMBER**(19,0) **NOT NULL** ,

cancellation\_policy\_id **NUMBER**(19,0) **NOT NULL** ,

bed\_type\_id **NUMBER**(19,0) **NOT NULL** ,

room\_type\_id **NUMBER**(19,0) **NOT NULL** ,

property\_type\_id **NUMBER**(19,0) **NOT NULL** ,

listing\_name **VARCHAR2**(255),

listing\_url **VARCHAR2**(2084),

latitude **FLOAT** (24),

longitude **FLOAT** (24),

price **FLOAT** (24),

weekly\_price **FLOAT** (24),

monthly\_price **FLOAT** (24),

security\_deposit **FLOAT** (24),

cleaning\_fee **FLOAT** (24),

guests\_included **NUMBER**(10,0),

extra\_people **FLOAT** (24),

minimum\_nights **NUMBER**(10,0),

maximum\_nights **NUMBER**(10,0),

accommodates **NUMBER**(10,0),

bathrooms **FLOAT**(24),

bedrooms **NUMBER**(10,0),

beds **NUMBER**(10,0),

square\_feet **NUMBER**(10,0),

rs\_rating **NUMBER**(10,0),

rs\_accuracy **NUMBER**(10,0),

rs\_cleanliness **NUMBER**(10,0),

rs\_checking **NUMBER**(10,0),

rs\_communication **NUMBER**(10,0),

rs\_rlocation **NUMBER**(10,0),

rs\_rvalue **NUMBER**(10,0),

is\_business\_travel\_ready **VARCHAR2**(1),

require\_guest\_profile\_picture **VARCHAR2**(1),

require\_guest\_phone\_verification **VARCHAR**(1),

summary **CLOB**,

space\_description **CLOB**,

description **CLOB**,

neighborhood\_overview **CLOB**,

notes **CLOB**,

transit\_description **CLOB**,

access\_description **CLOB**,

interaction\_description **CLOB**,

house\_rules **CLOB**,

picture\_url **VARCHAR2**(2084),

**PRIMARY KEY** (lid),

**FOREIGN KEY** (neighborhood\_id)

**REFERENCES** Neighborhoods(nid)**ON DELETE CASCADE**,

**FOREIGN KEY** (cancellation\_policy\_id)

**REFERENCES** CancellationPolicy(cp\_id) **ON DELETE SET NULL**,

**FOREIGN KEY** (bed\_type\_id)

**REFERENCES** BedType(bed\_id) **ON DELETE SET NULL**,

**FOREIGN KEY** (room\_type\_id)

**REFERENCES** RoomType(room\_id) **ON DELETE SET NULL**,

**FOREIGN KEY** (property\_type\_id)

**REFERENCES** PropertyType(prop\_id) **ON DELETE SET NULL**,

**FOREIGN KEY** (host\_id)

**REFERENCES** Hosts(hid) **ON DELETE CASCADE**

);

In the ER model, Listings has many one to many relationships, for example with RoomType or Cancellation\_policy. To describe those relations, we chose to have attributes, for example cancellation\_policy\_id, referencing those relations in the Listings table.

Then we have a many to many relationship between Listings and Amenities. We create a new table called ListingsHasAmenities like we did from the many to many relation between Hosts and Verifications.

**CREATE TABLE** ListingsHasAmenities (

lid **NUMBER**(19, 0) **NOT NULL**,

aid **NUMBER**(19, 0) **NOT NULL**,

**PRIMARY KEY** (lid,aid),

**FOREIGN KEY** (lid)

**REFERENCES** Listings(lid) **ON DELETE CASCADE**,

**FOREIGN KEY** (aid)

**REFERENCES** Amenities(aid) **ON DELETE CASCADE**

);

Then we have the weak one to many relationship between Listings and Calendars. To describe this relationship, the table Calendars has a primary key consisting of cid (calendar id) and lid (listing id) which references the lid from the Listings table.

**CREATE TABLE** Calendars (

cid **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

lid **NUMBER**(19, 0) **NOT NULL**,

cdate **DATE**,

available **VARCHAR2**(1),

price **FLOAT** (24),

**PRIMARY KEY** (lid,cid),

**FOREIGN KEY** (lid)

**REFERENCES** Listings(lid) **ON DELETE CASCADE**

);

Finally there is the last one to many relationship between the Reviews and Listings table which also has some attributes (reviewer\_id and reviewer\_name) and possess a Fk reference to a lid (listing id).

**CREATE TABLE** Reviews (

review\_id **NUMBER**(19, 0) **GENERATED BY DEFAULT AS IDENTITY**,

lid **NUMBER**(19, 0) **NOT NULL**,

rdate **DATE**,

comments **CLOB**,

reviewer\_id **NUMBER**(19,0),

reviewer\_name **VARCHAR2**(32),

**PRIMARY KEY** (review\_id),

**FOREIGN KEY** (lid)

**REFERENCES** Listings(lid) **ON DELETE CASCADE**

);

To decide if an attribute has to be **NOT NULL**, we made some assumptions based on the csv files. If some rows where empty in a column or has the value N/A, we assumed it can be NULL.

At the end, we added some constraints, for example on the price of the Calendars table to make sure the price is positive.

To be more reliable, here is a concatenation of all our constraints :

**ALTER TABLE** Calendars **ADD CONSTRAINT** min\_price **CHECK** (price >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_guests **CHECK** (guests\_included >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_price **CHECK** (price >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_weekly\_price **CHECK** (weekly\_price >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_monthly\_price **CHECK** (monthly\_price >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_extra\_people **CHECK** (extra\_people >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_cleaning\_fee **CHECK** (cleaning\_fee >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_security\_deposit **CHECK** (security\_deposit >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_minimum\_nights **CHECK** (minimum\_nights >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_maximum\_nights **CHECK** (maximum\_nights >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_bathrooms **CHECK** (bathrooms >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_bedrooms **CHECK** (bedrooms >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_beds **CHECK** (beds >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_square\_feet **CHECK** (square\_feet >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_rating **CHECK** (rs\_rating >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** min\_accuracy **CHECK** (rs\_accuracy >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** cleanliness **CHECK** (rs\_cleanliness >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** checking **CHECK** (rs\_checking >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** communication **CHECK** (rs\_communication >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** rlocation **CHECK** (rs\_rlocation >= 0);

**ALTER TABLE** Listings **ADD CONSTRAINT** rvalue **CHECK** (rs\_rvalue >= 0);

**ALTER TABLE** Hosts **ADD CONSTRAINT** min\_rate **CHECK** (response\_rate >= 0);

**ALTER TABLE** Hosts **ADD CONSTRAINT** max\_rate **CHECK** (response\_rate <= 100);

## General Comments

The work between team members was equally divided and everything went well.

# Deliverable 2

## Assumptions

1. The URL size is never longer than 2083, so we chose VARCHAR2(2084) for our tables.
2. Oracle SQL VARCHAR2 does not accept string longer than 4000, so we’ve chosen CLOB (Character Large OBject) type for long descriptions.
3. Oracle does not accept ‘\n’ and ‘\r’ characters while loading.

## Data Loading

To load the data, we’ve chosen to process csv files with python using pandas library. We’ve concatenated the calendars.csv files to get CALENDAR and the reviews.csv files to get REVIEWS.

Concerning the listing.csv files, we’ve first concatenated them and then created the tabs listed in Deliverable 1.

Below, a list of data process we’ve done:

1. For strings, we’ve removed ‘\n’ and ‘\r’ characters such that the entry is only one line
2. Concerning prices and rates, we removed ‘$’ and ‘%’ characters and cast them as floats
3. To list amenities, verifications, bed types, cities, countries, neighborhoods, response-times, room types, cancellation policies and property types, we went through the tables and listed them in tables
4. Concerning the countries, we observed that cities are stated with different names, we pick a standard name for them (the city name written on the csv files title)

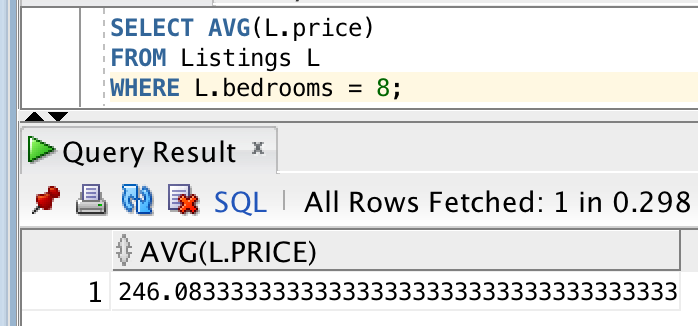
Once the tables are created as csv files and created in Oracle SQL server, we import the csv tabs into the server

## Query Implementation

### Query 1: What is the average price for a listing with 8 bedrooms ?

### 

#### SQL statement



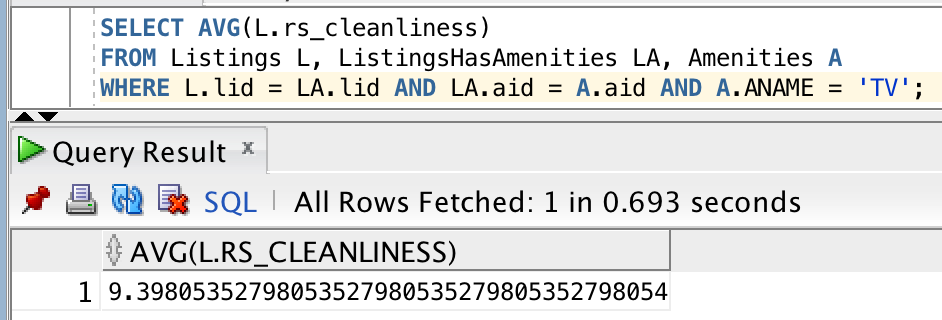
#### Description of logic:

Since our Listings table directly has the number of bedrooms and the price as argument, the SQL code is straightforward : we filter on the bedrooms argument then compute the average price.

### Query 2: What is the average cleaning review score for listings with TV ?

### 

#### SQL statement



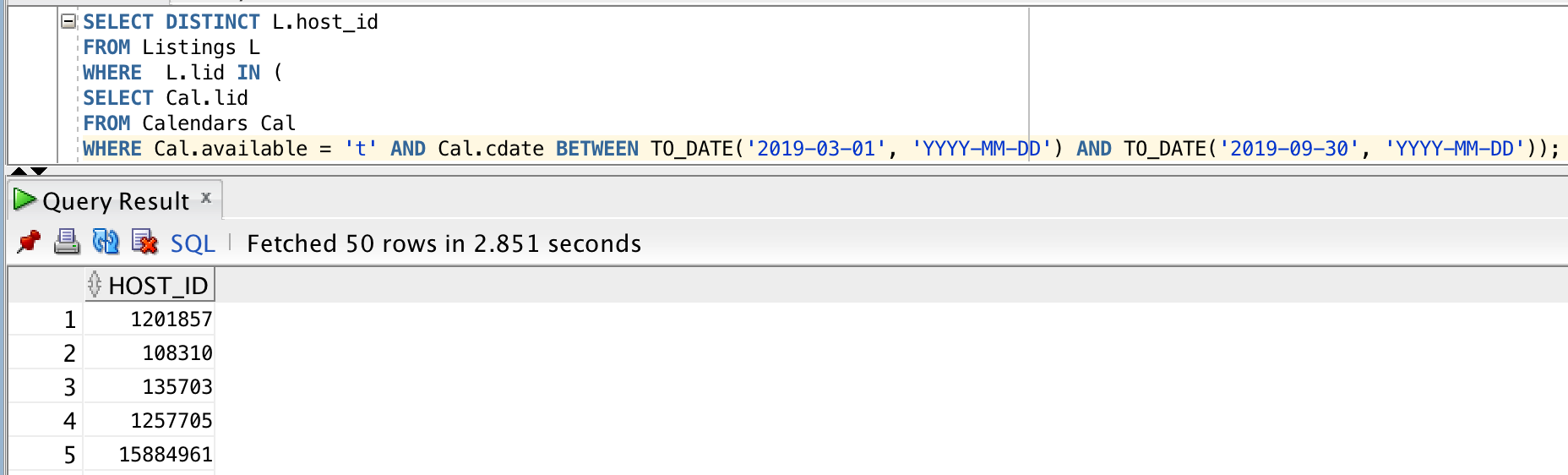
#### Description of logic:

This time, “TV” is not an argument in Listings so the SQL code is a bit more complex. Since there is a many-to-many relationship between Listings and Amenities we need to use the ListingHasAmenities table which “links” the 2 together (each row has a lid and an aid). We only compute the average rs\_cleanliness from listings whose lid is in a row of ListingHasAmenities where the aid is the one of the TV amenity. That’s why we need access to these 3 tables.

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

### 

#### SQL statement



#### Description of logic:

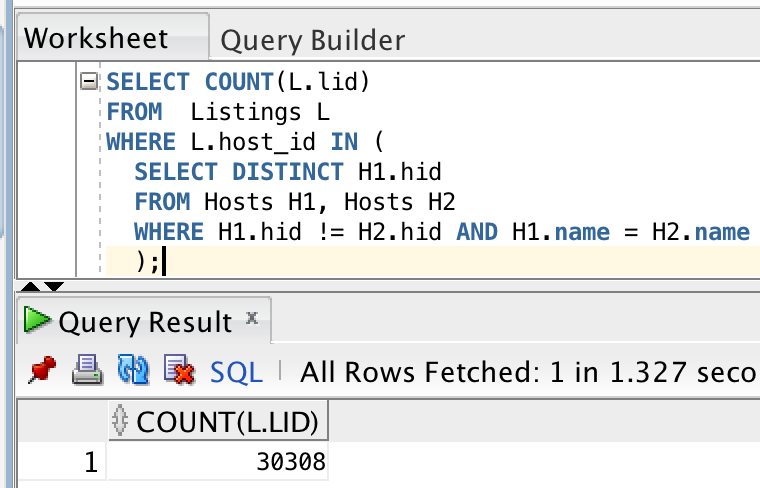
Let’s first explain the “inner select” we do on the Calendar table. We only keep rows which are available and whose cdate attribute is between the limits. To do that, we used the BETWEEN SQL keyword and converted the string values to valid dates. Note : only 30 days in September, if we use 2019-09-31 as the cutoff the code doesn’t work. This gives us a list of Listing IDs that are available for these dates.

For the outer Select on Listings, we only keep rows whose lid is in the aforementioned list. Then, we select the distinct host\_ids, since there will be duplicates otherwise.

### Query 4: Print how many listing items exist that are posted by two different hosts but the hosts have the same name.

### 

#### SQL statement



#### Description of logic:

Again, let’s start with the inner Select. We keep every distinct hid who has a “twin” name of different hid.

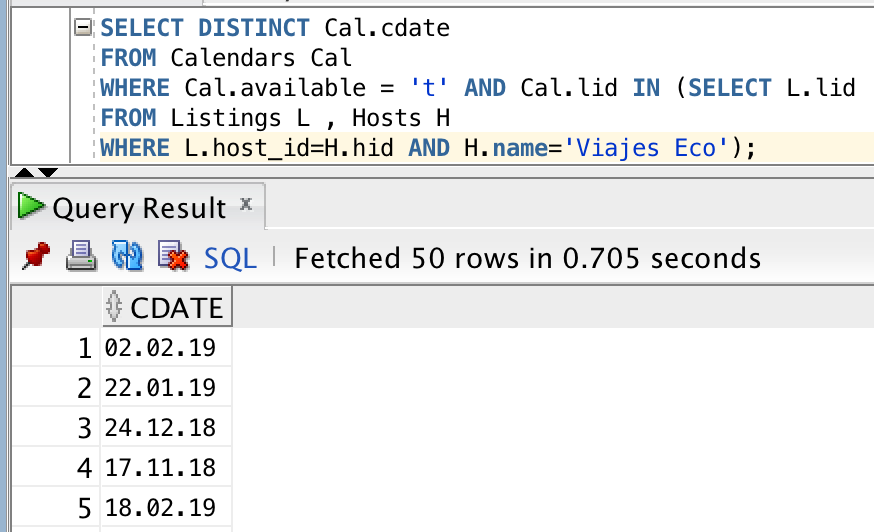
Then, we count how many Listings have one of these hosts thanks to the WHERE IN expression on L.host\_id.

### 

### Query 5: Print all the dates that 'Viajes Eco' has available accommodations for rent.

### 

#### SQL statement



#### Description of logic:

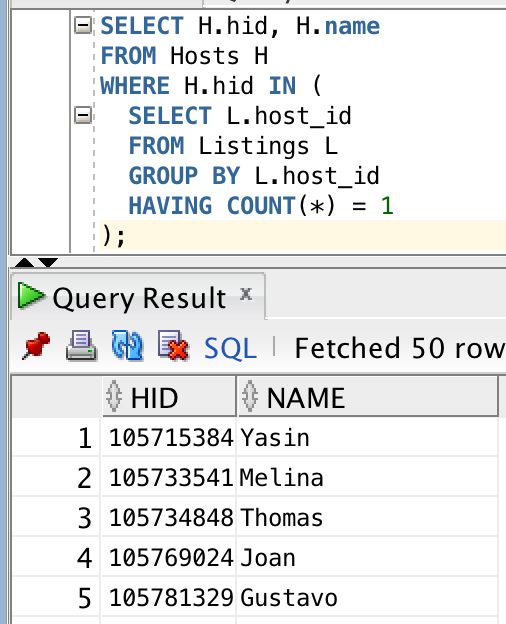
Inner select : we select every listing id from host Viajes Eco. We need to access the Hosts table because the listing only contains the host’s id and not its name.

Outer select : we keep every calendar entity which is available and whose lid is in the aforementioned selection, and removes duplicates with the DISTINCT keyword on cdate, since multiple entities of Calendars have the same date.

### Query 6: Find all the hosts (host\_ids, host\_names) that have only one listing.

### 

#### SQL statement



#### Description of logic:

Inner select : the first thing that happens is the grouping of listings by host\_id. Then, with the HAVING condition on the count, we only keep listings which are the unique listing of their host, and we select their host\_id (no distinct needed ).

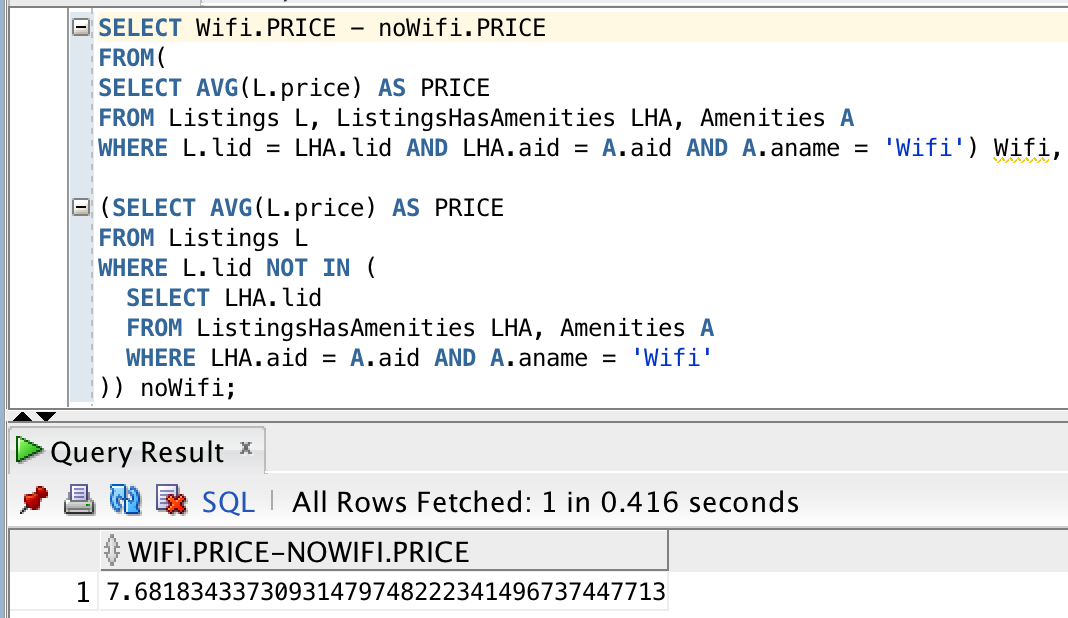
We now have every host\_id which satisfies the condition, but we still need to get the host’s name which is not on Listings : we need to access the Hosts table and that’s the only reason why we’re doing an outer selection.

Outer select : We get the right row in the Hosts table with the WHERE IN on H.hid, which needs to be part of the aforementioned selection.

### Query 7: What is the difference in the average price of listings with and without Wifi ?

### 

#### SQL statement



#### Description of logic:

This query is more complex than previous ones. We will first start with the first half, named Wifi. We select the only rows of the ListingsHasAmenities table which link a listing to the Wifi amenity, and then compute the average price of said listings.

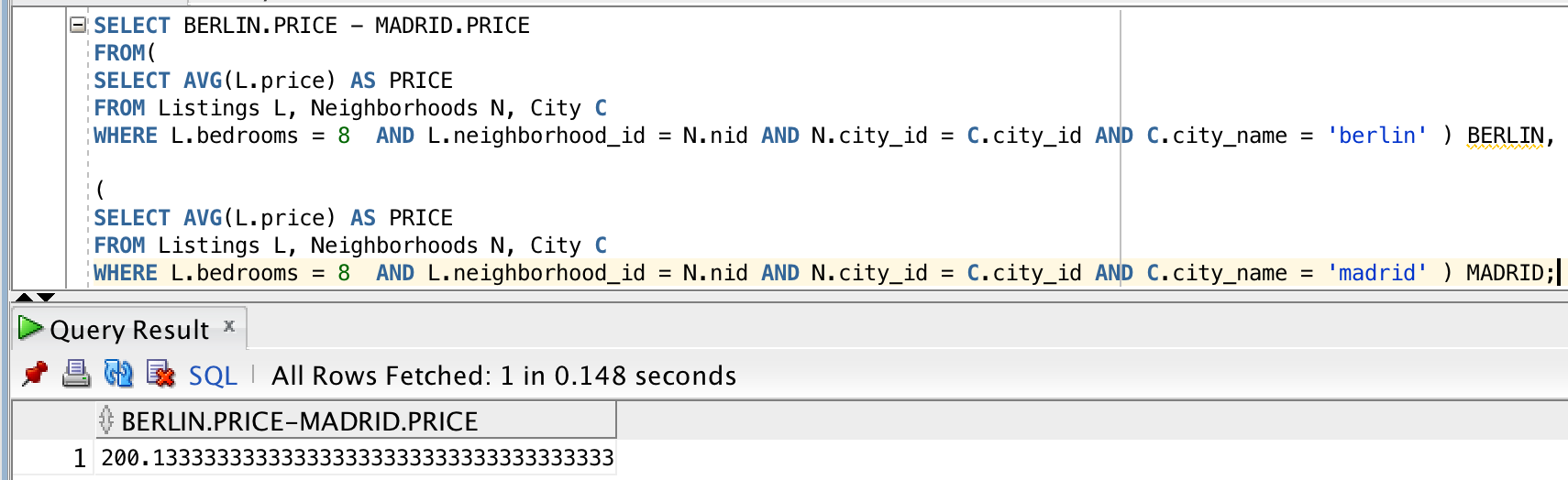
For the second part named noWifi, we use NOT IN on L.lid so that we only get Listings which don’t have that amenity (the inner select works the exact same way as before, those are the same LHA rows), then compute the average price.

Finally, we use those 2 values in a subtraction. As expected the result is positive, which confirms that apartments with Wifi cost more on average.

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

### 

#### SQL statement



#### Description of logic:

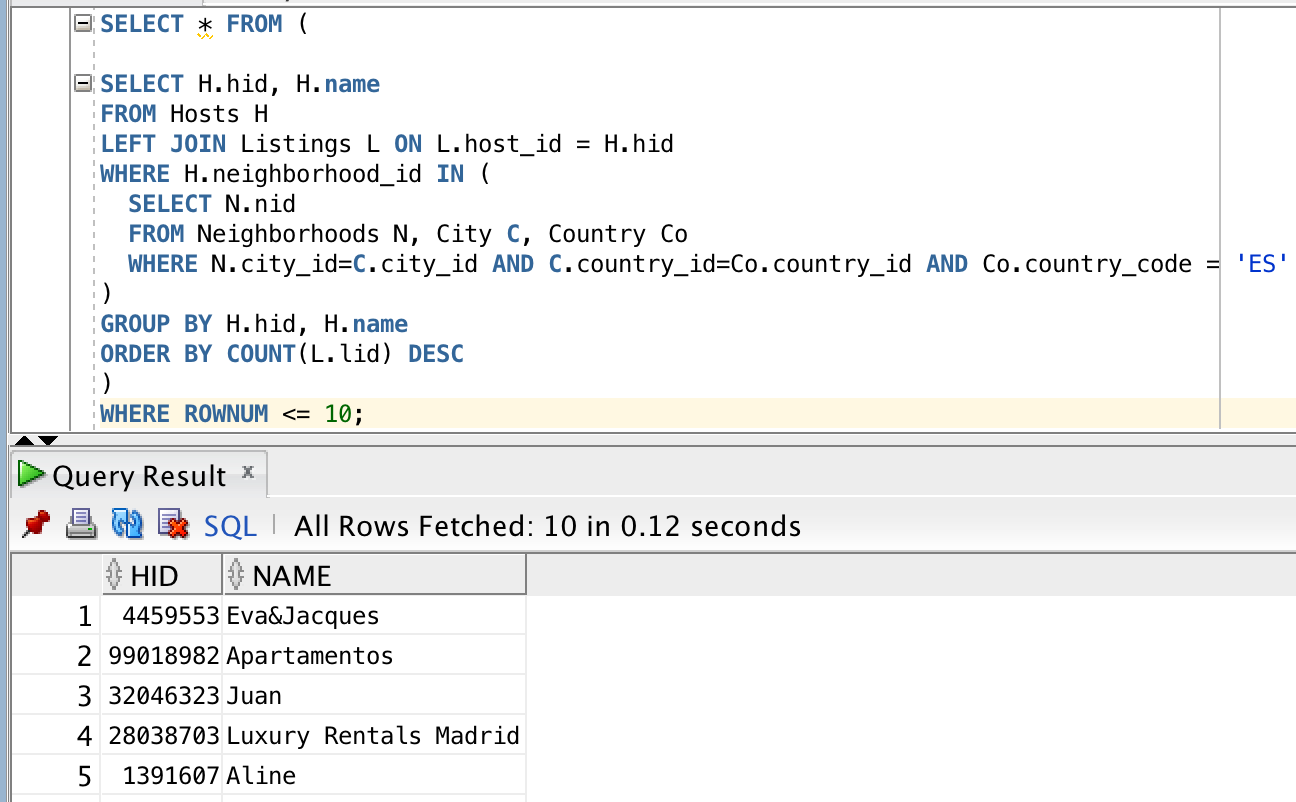
Same strategy here, the 2 inner selections have the number of bedrooms condition as well as the city condition. To access the city name, we need to go from the Listings table to the Neighborhoods table to the City table where the name is stored.

We then use those 2 results in a subtraction where we see that Berlin listings cost a lot more on average.

### Query 9: Find the top-10 (in terms of the number of listings) hosts (host\_ids, host\_names) in Spain.

### 

#### SQL statement



#### Description of logic:

Inner select : we get every Neighborhood ID in Spain. To access Spain, we need to go from Neighborhood to City to Country.

Outer select : thanks to the aforementioned selection, we access every row of the Hosts table with a spanish Neighborhood id. However, we also need information which is in the Listings table, which is why we do a left join between Listings and those specific Hosts, on host\_id. We can now group them by host\_id, then order them in a descending order based on how many listings each host has.

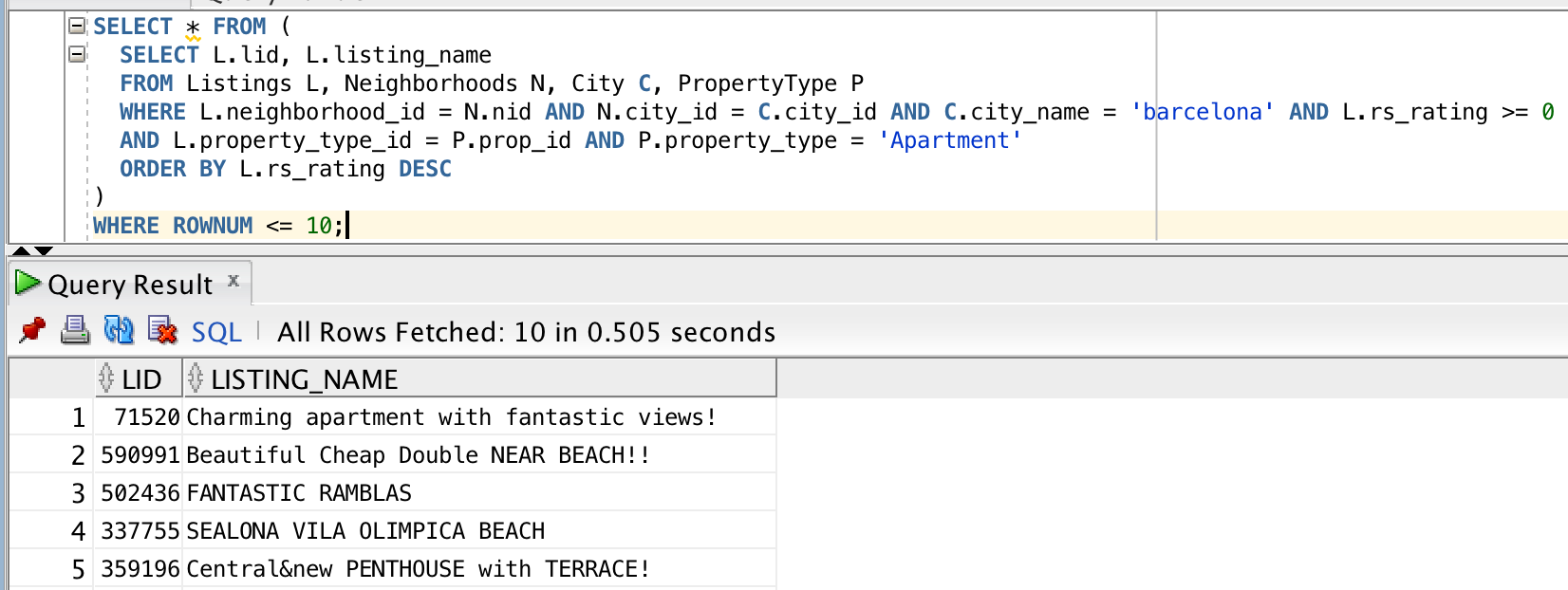
Finally, to only keep the top 10, we use WHERE ROWNUM in an exterior selection, so that the rownum filtering is done at the very end. We tried using more elegant commands like LIMIT but it seems they don’t work with Oracle SQL.

Note that we interpreted the question as literally as possible, we don’t check if the listings are in Spain but only check that the host himself lives there.

### Find the top-10 rated (review\_score\_rating) apartments (id,name) in Barcelona.

### 

#### SQL statement



#### Description of logic:

We need to go from Listings to Neighborhoods to City to access the city name. The condition on having a non-negative rating is here to filter out (null) ratings. We also “link” Listings and PropertyType to only access listings of the “Apartment” type.

We do the same outer rownum condition as in the previous query, to make sure that we don’t do it too early and lose information.

## General Comments

After the feedback from Deliverable 1, we made lots of changes (URL size from 2048 to 2084, refactor of the ER model, refactor of the SQL queries to create the tables and the relations, …) and we directly made changes to this report at Deliverable 1.

# Deliverable 3

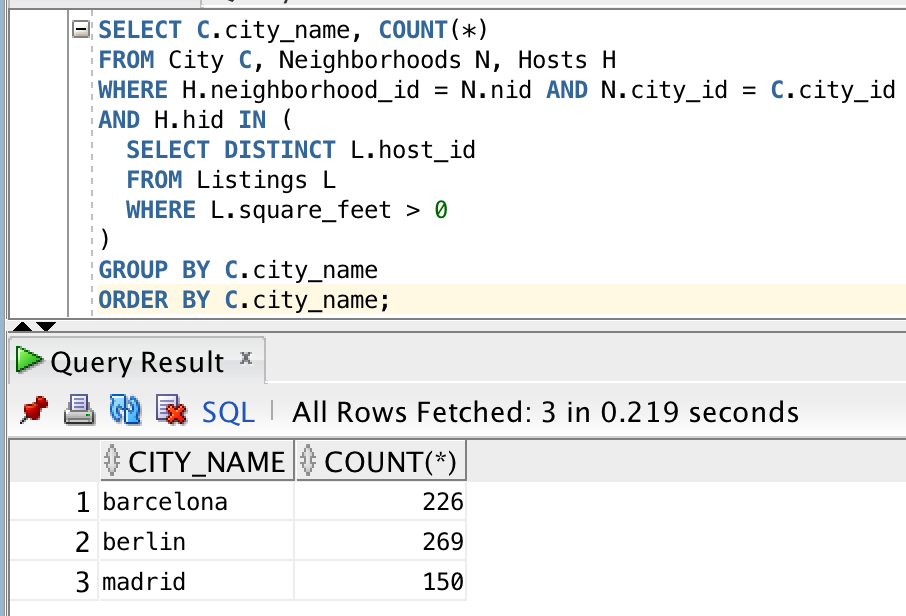
# Assumptions

We did not make more assumptions here since we have only queries to make and an interface to create.

## Query Implementation

### Query 11:

#### SQL statement



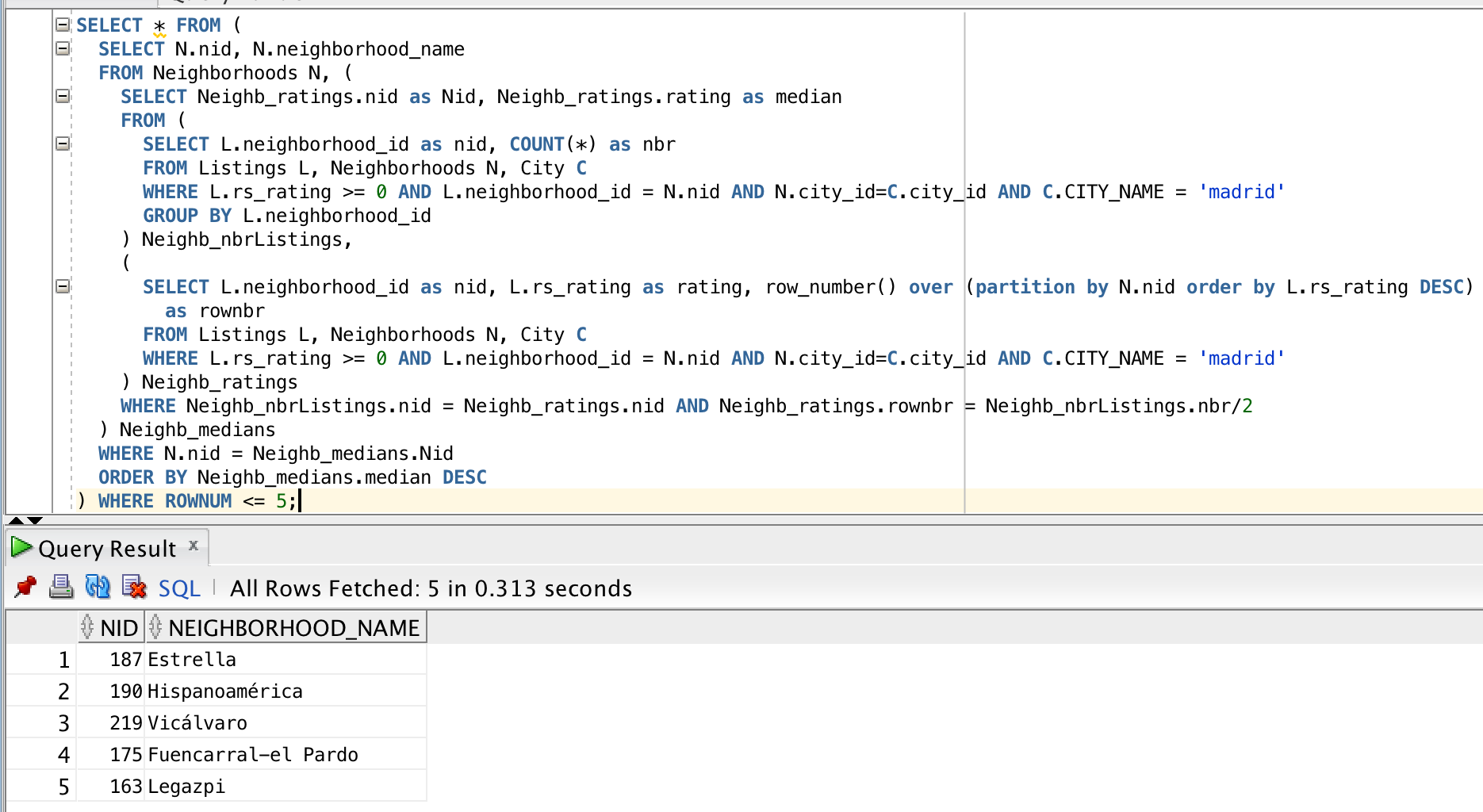
#### Description of logic:

In the inner selection, we get every host\_id which has a listing whose area is declared in square feet (the positive condition also filters out null values).

We then group those hosts by cities and count how many there are per city in the outer select. We can see that Berlin is the city with the most hosts fulfilling this condition.

### Query 12:

#### SQL statement



#### Description of logic:

The way we’re going to compute the median of each neighborhood is by ordering the listings’s review scores and associating them with a row number corresponding to the neighborhood, done with the command :

“ row\_number() over (partition by N.nid [...])”

Then, we access the score of the review score whose row number is half of the total and this is our median for this particular neighborhood. Since we can’t compute the total number of review score at the same time, we’re computing that number in another select.

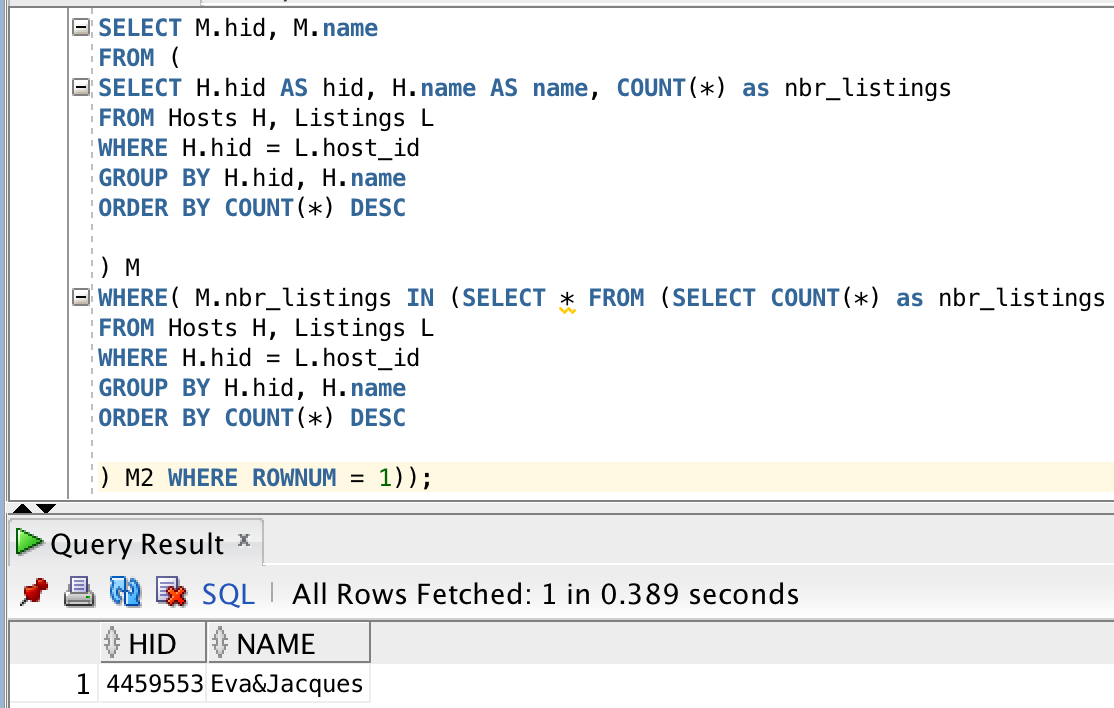
The table “Neighb\_nbrListings” associates each neighborhood with its total number of listings. The table “Neighb\_ratings” ’s rows each contain a score associated with its neighborhood and the row number for this neighborhood. For example, an item with “rownbr” 3 would mean that it’s the 3rd best score of its neighborhood.

In the select which encloses both of those tables, we can compute the median row number for each neighborhood by dividing the number found in Neighb\_nbrListings by 2 to get the right rows of the other table Neighb\_ratings. The table containing this selection is called Neighb\_medians and associates each nid with its median.

We now only need to order them by median in the outer selection, then select the top 5 with the ROWNUMBER command in the outermost select. As stated previously for another query, this command isn’t elegant but it’s the only one we found to work in Oracle SQL (LIMIT does not seem to work for example).

### Query 13:

#### SQL statement



#### Description of logic:

Since multiple hosts could have the same maximum number of listings, we first compute a table where each row associates a host with its number of listings (first half : table M). This is the only table of the main selection.

In the second half of the query, we compute the maximum number a listings hosts can have. This second part (the first row of table M2) is used in a WHERE IN command in the main selection, specifying that we only keep hosts whose associated number of listings is the maximum. We only keep the host id and name in the end.

We can see from the only result that there aren’t multiple hosts tied for the maximum number of listings.

### Query 14:

#### SQL statement



#### 

#### 

#### Description of logic:

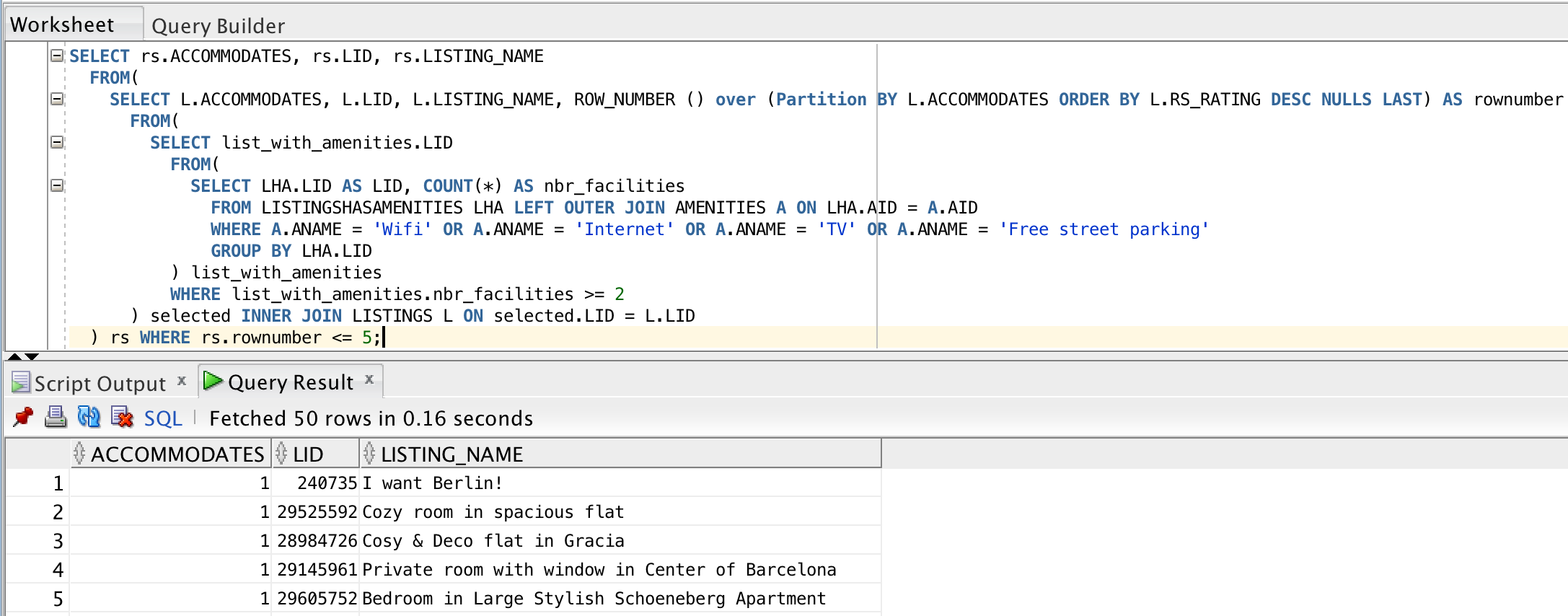
As usual, we’ll start with the innermost selection then work our way to the outermost one. The innermost computes every host id which has a government ID. We will use it in the enclosing selection to only keep listings of those hosts (IN condition).

In this selection, we also need to do multiple joins to access the property type, the city and the cancellation policy of each listing. We can then filter the listings based on the multiple criterias we have with WHERE statements. The only thing left at this point are the criteria specific to the Calendars table as well as computing the average price, which also requires access to Calendars.

This will be done in the next enclosing selection, which joins the Calendars table with the current one, on the listing id. We can now add the conditions on the availability. To compute the average price, we then group the rows by listing id, then order it by descending average price. We then only keep the first 5 rows in the outermost selection.

### Query 15:

#### SQL statement



#### 

#### Description of logic:

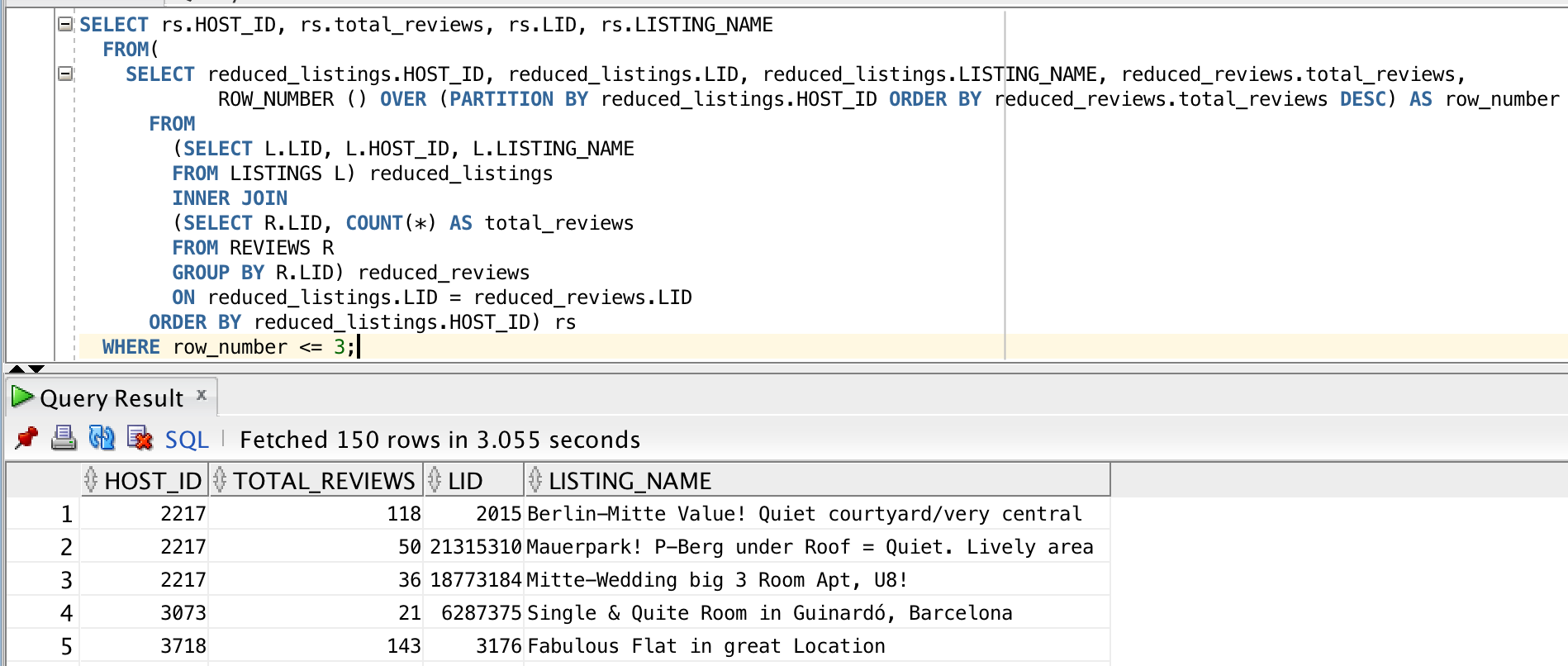
The innermost selection associates listings with the number of valid amenities they have, which needs to be the 4 specified. We then only keep the lid of those with at least 2 in the enclosing selection.

In the enclosing selection, we do a join on this table with listing on lid. We use partition by the number of accommodates, ordering by their review score to associate each listing with a corresponding row number. This row number indicates what the listing’s rank is for its “number of accommodates” category.

Finally, in the outermost selection, we only keep rows with low row numbers (<= 5) to only keep the best listings for each category. The order is still the same, meaning the first ones are the best rated.

### Query 16:

#### SQL statement



#### 

#### Description of logic:

Our first inner selection, called reduced\_listings, keeps every row of Listings but filters out useless columns.

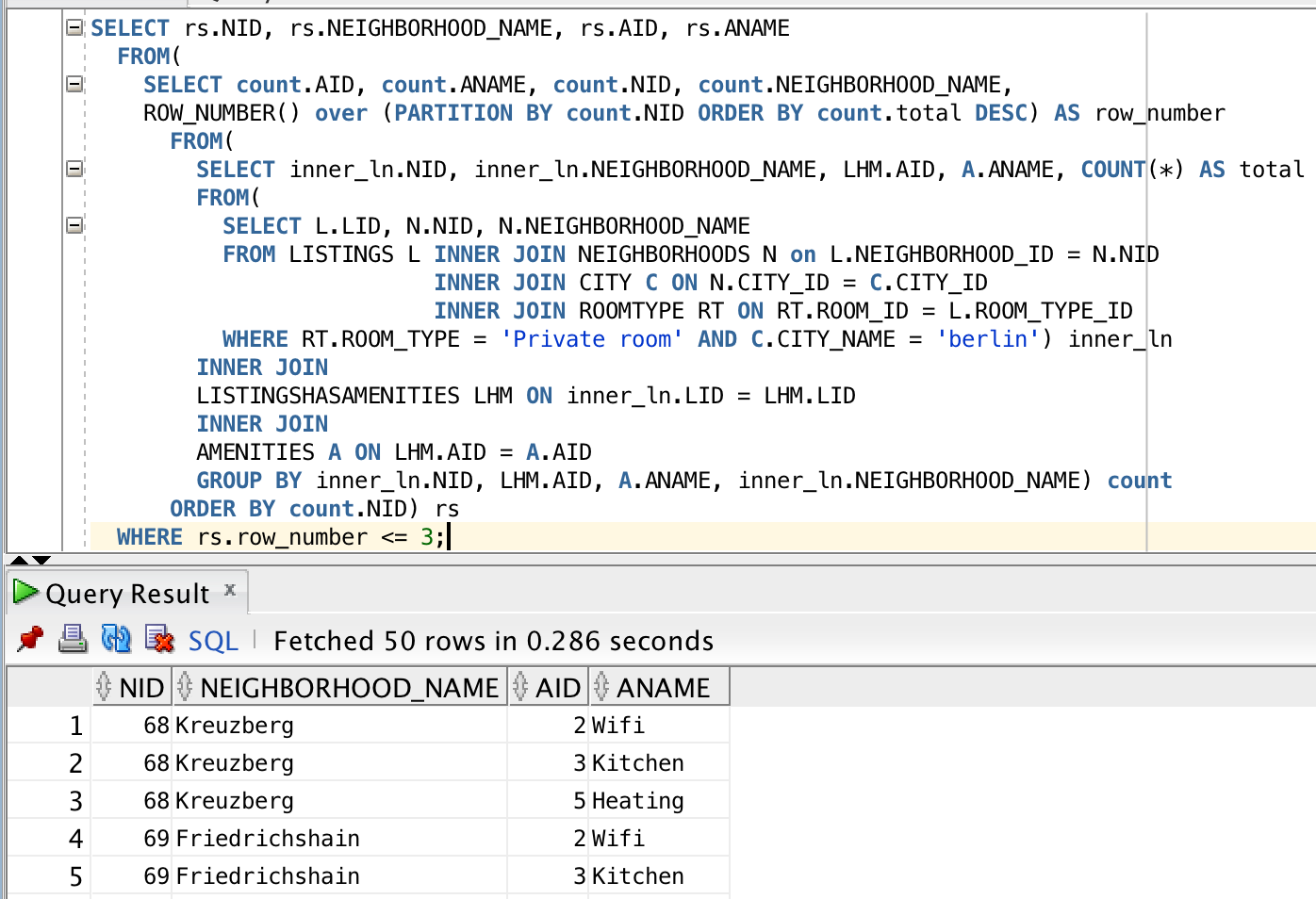
The second, called reduced\_reviews, associates a Listing id with its number of reviews. We join those 2 tables on lid.

The next enclosing selection only accesses this joined table. It adds a row number to each row, partitioned by host\_id and ordered by descending total reviews. Thus this row number is the rank of a listing for its host based on popularity. We order everything by host\_id.

Finally, the outermost selection only keeps rows whose rank is 3 or lower.

### Query 17:

#### SQL statement



#### Description of logic:

Let’s start with the “inner\_ln” table. It uses joins to access each listing’s neighborhood, city and room type, then filters based on the two specified conditions to only keep valid listings.

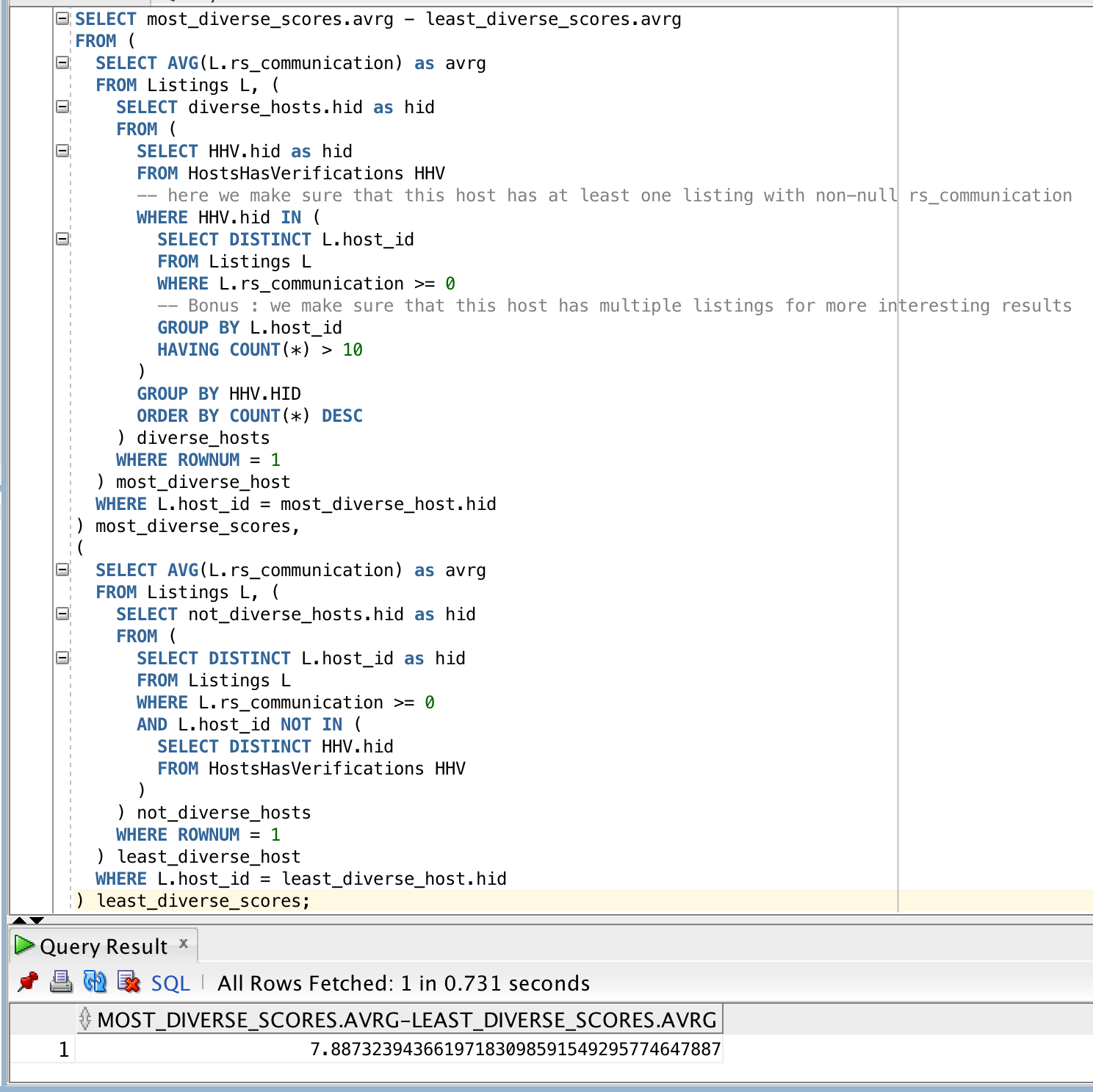
In the enclosing selection, joins are used to associate the amenities with their listings. We then group them by neighborhood to compute the total of each amenity per neighborhood.

Then, in the next enclosing selection, we create row numbers which indicate the rank of an amenity for a certain neighborhood in terms of popularity.

As usual, we then do one last enclosing selection which only keeps small row numbers (rank, <= 3 in this case to keep the top 3 of each neighborhood).

### Query 18:

#### SQL statement



#### Description of logic:

This query is very long but can be divided in two halves : the first one finds a host who has the most verifications, while the second finds a host who has none. We then compute the average rs\_communication of listings with the corresponding host\_id then compute the difference.

For the first half, the general strategy is to use the HostsHasVerifications table to find out which hosts have the most of those verifications. Since each row of this table is a relation, grouping those rows by HID then ordering them based on COUNT(\*) will give us a list with the most diverse hosts on top.

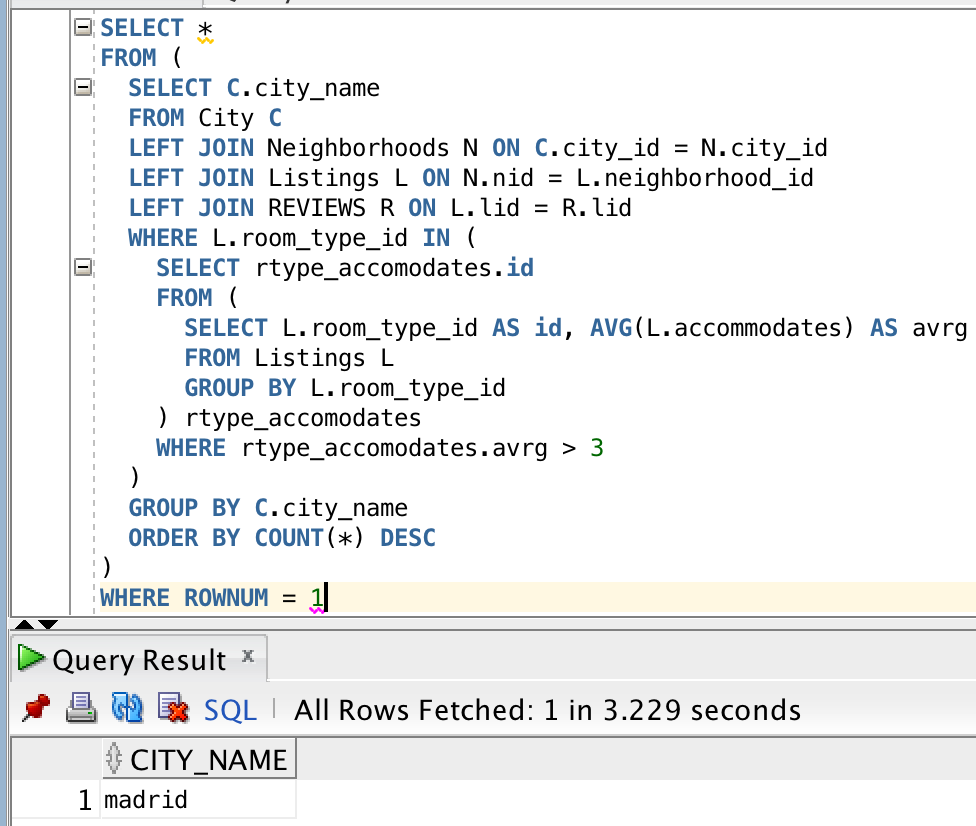
We’re also adding conditions on the innermost selection to make sure that we select hosts which have at least 10 listings and that those have non-null communication scores. The 10 listings condition wasn’t specified but we added it to get more interesting results : since we’re only taking one such host, we might as well choose one with more associated data. We then take the first host of this ordering in the enclosing selection.

For the second half, the strategy is a bit different because we actually want a host who has no verification. This means that its id should not be in the HostsHasVerifications table. We again use a condition on rs\_communication to make sure we don’t get null values.

The result is an insanely big difference considering the maximum theoretical difference would be 10. The result is of course positive, meaning the “least diverse host” indeed has a worse communication rating which is not surprising at all.

### Query 19:

#### SQL statement



#### Description of logic:

Once again let’s start with the innermost selection. This table called rtype\_accomodates associates a room\_type\_id with the average number of accommodates those listings have.

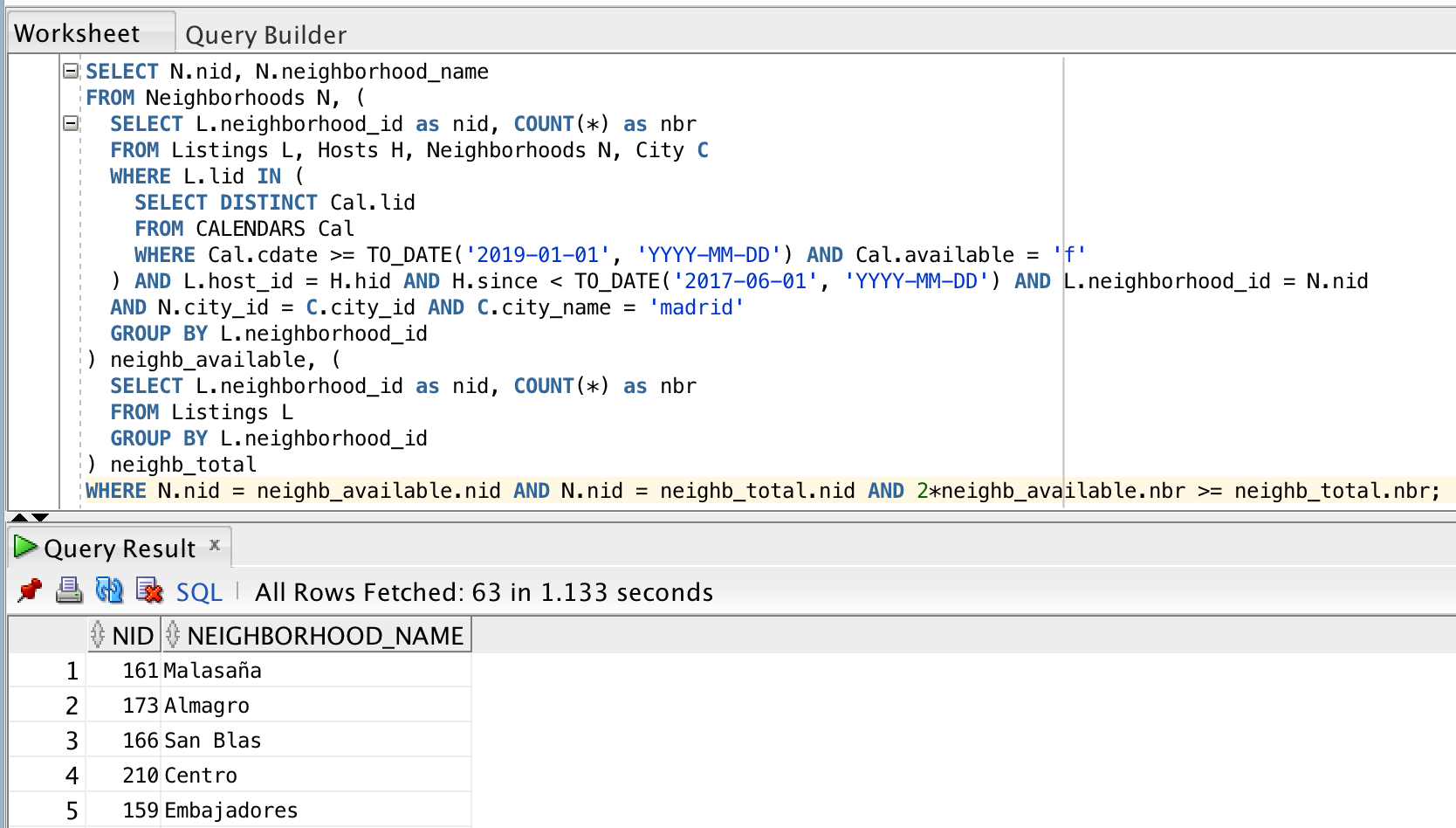
In the enclosing selection, we now only keep the room\_type\_id’s whose average of accommodates is greater than 3.

In the enclosing selection, we associate each review with its city thanks to multiple joins, and only keep reviews whose associated listing has a room\_type\_id contained in our previous selection. We then group reviews by city and order by that count to get an ordering of cities with the highest number of valid reviews.

Then we simply use rownum = 1 in a last enclosing selection to get the number 1 city, which is Madrid.

### Query 20:

#### SQL statement



#### Description of logic:

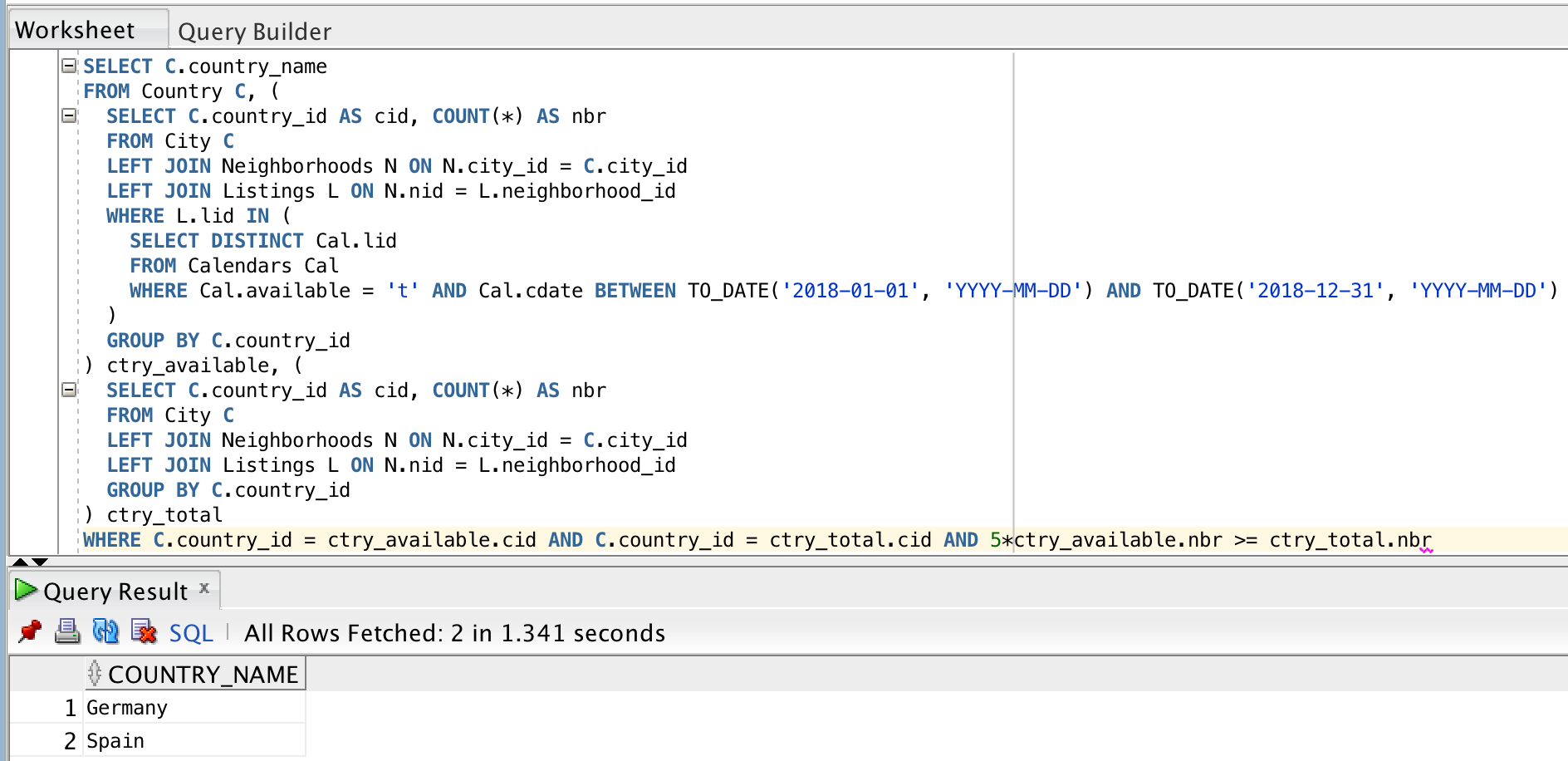
Our general strategy to compute if the 50% cap is passed or not is to compute the total number of listings per neighborhood in the table called neighb\_total on one hand, and to compute the number of listings that satisfy the wanted conditions on the other, in the table called neighb\_available.

We assumed that “occupied in 2019” meant being unavailable for at least one day, or one row of the Calendars table. The way we compute the total number of listings per neighborhood is straightforward. The 3 conditions on the listings (being in Madrid, having a host who joined airbnb before the date and being occupied for a day in 2019) are all done on the other table neighb\_available.

In the enclosing selection, we then get corresponding rows of those 2 tables with their nid, and write a condition where the number of valid listings multiplied by 2 can’t be smaller than the total, which is equivalent to the number of valid listings being equal or superior to 50%.

### Query 21:

#### SQL statement



#### Description of logic:

We once again compute the total of listings on one side and the number of valid ones on the other, in a different table.

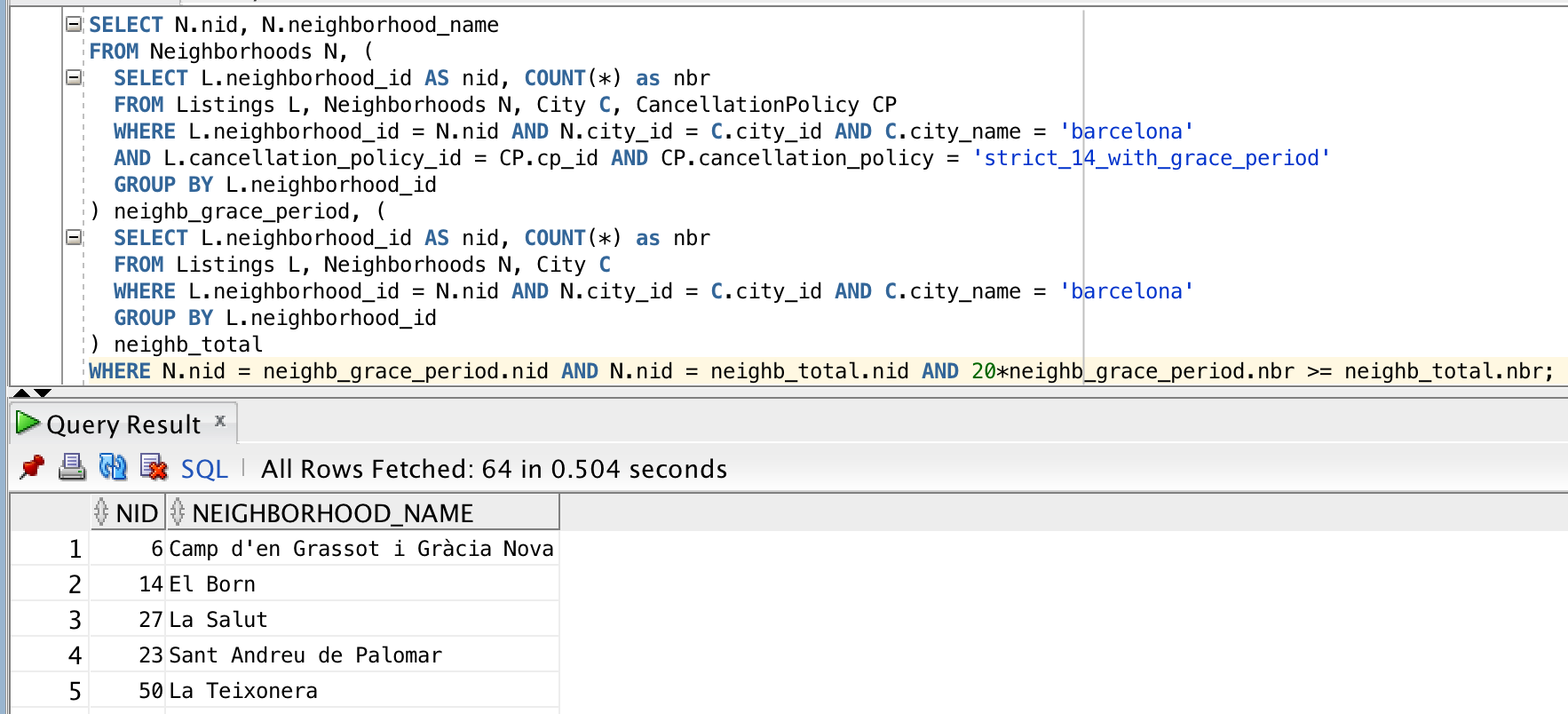
Since we want to associate the listings to the countries we do a bunch of joins (from Listings to Neighborhoods to City which has the country\_id) then we group on country\_id. The condition on the availability is done as usual with a “WHERE L.lid IN” statement. We assumed that “available in 2018” meant being available for at least a day, or one row of Calendars.

In the enclosing selection, we do the exact same strategy as we did for the previous query. 20% is this time translated to a multiplier of 5.

The result (every country printed) isn’t surprising, no country in the world should have more than 80% of its listings be completely unavailable for an entire year (unless we think of amusing scenarios for countries like Vatican City or North Korea).

### Query 22:

#### SQL statement



#### Description of logic:

Once again we use the same strategy of having 2 tables at the same level, one for the total and one for the valid listings exactly like our 2 last queries. The first table called neighb\_grace\_period counts how many listings per neighborhood verify the condition, while neighb\_total computes the total number.

The 5% condition translates to a multiplier of 20 this time. It is interesting to note that every single neighborhood in Barcelona is printed in that case (64). This isn’t too surprising since there are only 6 different cancellation policies : it makes sense that this one would be present more than 5% of the time in every neighborhood.

If we change the cap to 10% (which translates to a multiplier of 10), 63 out of 64 neighborhoods get printed. We can see that the 5% cap doesn’t seem to be well suited to this particular query.

#### 

## Query Analysis

### Selected Queries (and why)

We chose 3 different queries which each require a different indexing. We added indexes in that order.

Query 22 : it contains 2 similar tables (neighb\_grace\_period and neighb\_total) which are created by accessing the neighborhood\_id column of Listings for WHERE conditions as well as for GROUP BY commands. Thus, by creating an index on Listings where the expression is the neighborhood\_id, we should see major improvements.

Query 13 : This one is pretty straightforward, we’re again constructing 2 tables by associating rows of Listings based on a column which is not the primary key (lid) : thus we should see major improvements. We’ll use the same strategy except that instead of neighborhood\_id, it’s host\_id this time.

Query 18 : Arguably one of the more complex queries, it makes use of the index created for query 13 as well as a new one on Listings’s column rs\_communication, because we only want to keep Listings which have non-null communication scores. Once again, since this is done in 2 different tables this guarantees major improvements making it a strong candidate.

General note on running times : since there seems to be some sort of caching when running the same query multiple times, the runtimes are those obtained when doing a “fresh” query.

#### Query 22

Initial Running time: 507ms

Optimized Running time: 109ms

There’s a full table access on Listings in both selections with a cost of about 10’000 for each. Since we need to associate/group rows of Listings based on their neighborhood\_id, if we don’t have access to indexes for this column we have no choice but to check every single row, unordered, for their host\_id.

After the creation of the index, the query plan changes as expected : while the cardinality stays the same for the 2 scans (42’094, our number of Listings), the cost goes down to 25 (basically negligible) and it shows that our new index was used in both places. It’s not written “Table access full” anymore as expected.

Initial plan : 20’419 total cost

Improved plan : 1’378 total cost

#### 

#### Query 13

Initial Running time: 576

Optimized Running time: 102ms

It’s interesting to see how similar the improvement is to the previous query. Even though the queries are different, their overall “architecture” is pretty similar and the added index is used in a similar way even if it’s not the same column. The runtimes and cost changes are clearly comparable as well. Once again we’re not doing a full table access 2 times after the change (each of cost ~10’000) and the new indexing is used in both parts (tables M and M2).

It is interesting to note that 748 of that total cost comes from the full table access of the Hosts table, of cardinality 26632.

Initial plan : 21’450 total cost

Improved plan : 1’104 total cost

#### Query 18

Initial Running time: 378ms

Optimized Running time: 110ms

It’s very important to note that our initial executions were done after the creation of the indexing on Listings’s host\_id column, which was made for query 13 but is also used in this one. Thus, the initial runtime/cost would have been even higher than what is noted here without it.

Like in previous cases, we no longer make a complete table access 2 times which tremendously reduces the cost, our new index is used in 2 places. Even if the >= 0 condition on the communication rating may seem minor (to avoid null values), it requires checking every row if there is no way to order the rows based on this column and adds a ton of cost to the total. The indexing on Listings’s host\_id and this new one work together to bring the cost down to a satisfying level.

Initial plan : 20’614 total cost

Improved plan : 552 total cost

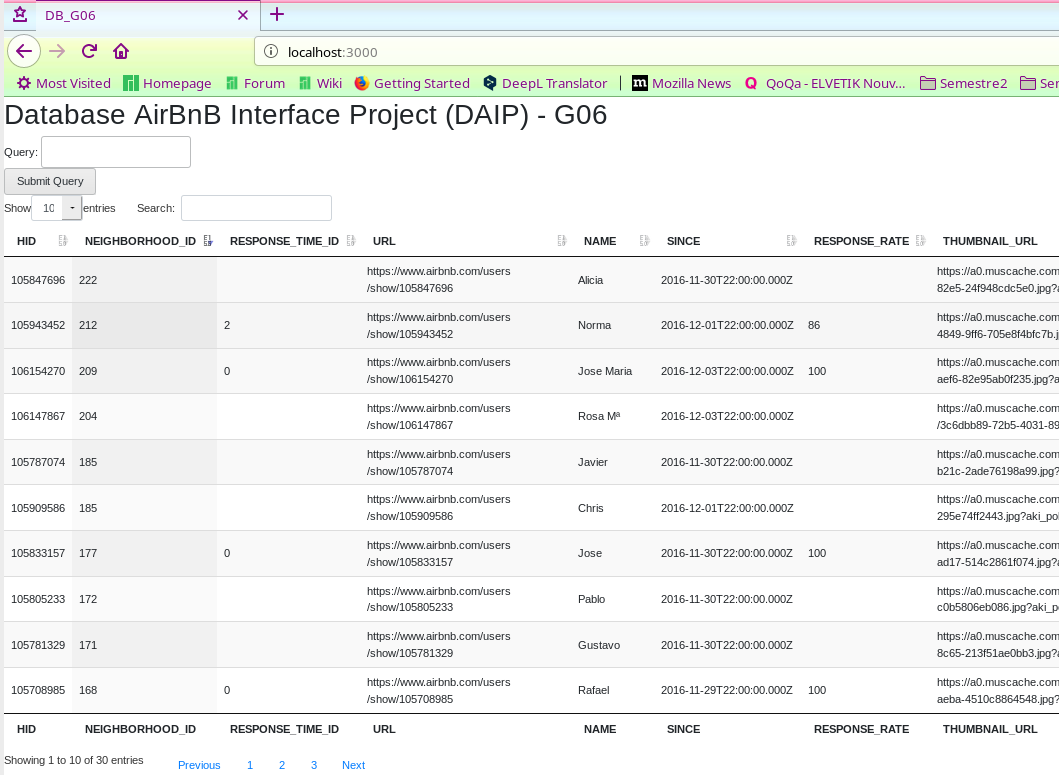
# 

# Interface

### 

### First unsuccessful attempts

We first gravitated towards a website for our interface, thinking it would be well suited to our goals and that we could use tools like Bootstrap to efficiently implement things. This was a bit of a gamble since neither of us were particularly experienced in web-based solutions. Our first choice was using NodeJS since there were many tutorials available. We were successful with our first version, which displayed results and had pagination. Unfortunately we were quickly stuck in part because we weren’t sure how to implement and interface and have it interact with our RESTapi. A big problem was also that some of the tools we were using weren’t suited for OracleSQL, which we didn’t know when we first decided to go this way. As a result, we chose to go for a “simpler” solution using html/css, javascript and php but once again we found ourselves stuck again for various reasons. After all this time spent on trying out web-based solutions, we had to accept to start again from scratch and use something we were more familiar with which is Python.

Here is a screenshot of our first version which was using NodeJS and DataTables :

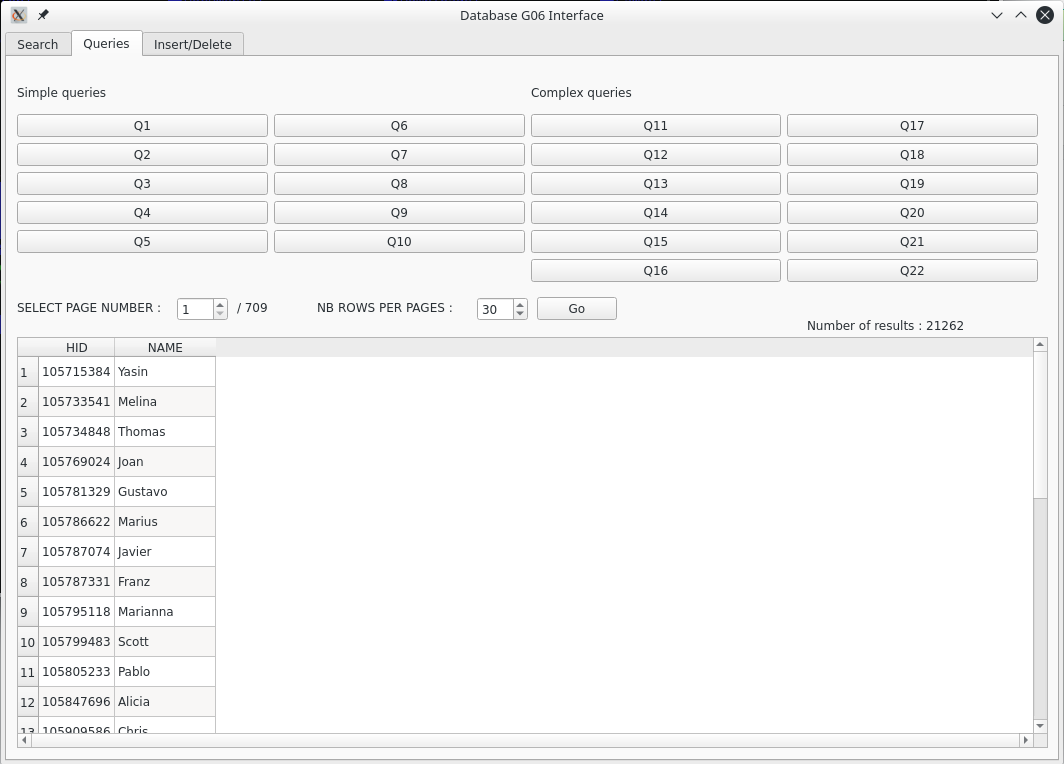
### Design logic Description

### We chose to design the interface with python 3 and more precisely, with pyqt5. Our interface has three tabs which are respectively for the search, the predefined queries of deliverable 2 and 3 and for the insert/delete feature.

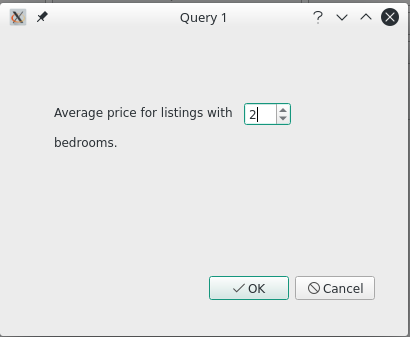
### 

### Screenshots

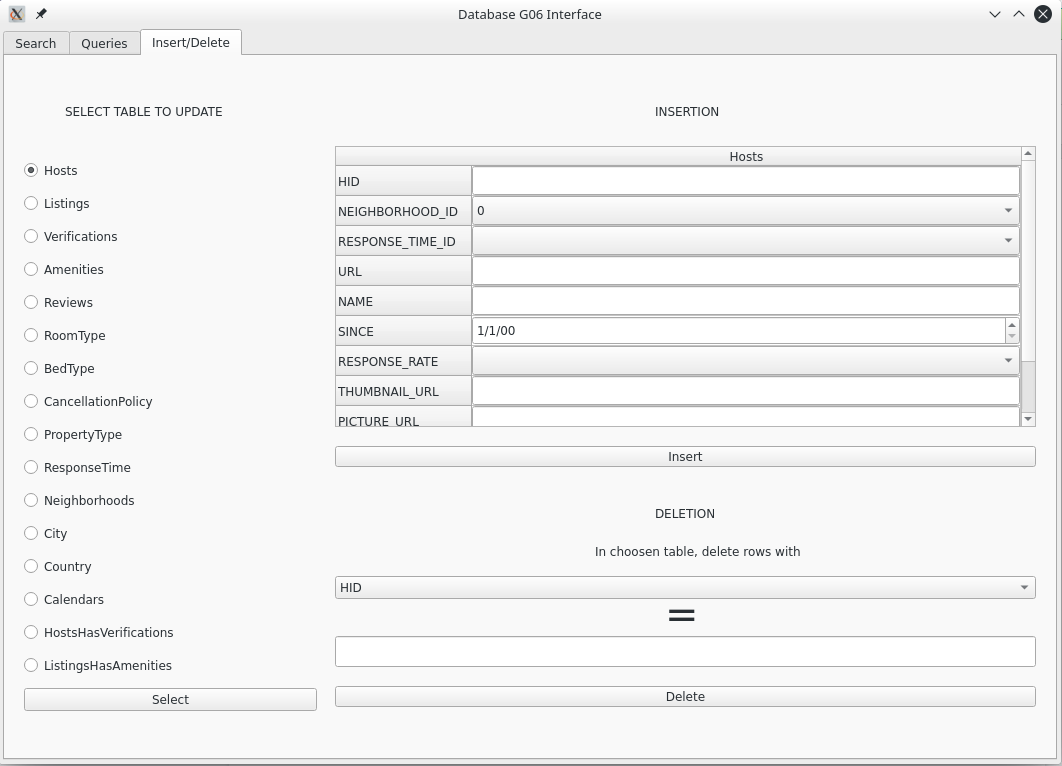
### In the queries tab, we have chosen to have all 22 queries in the form of buttons than when are pressed, open a pop-up with a description of what the specific query does. With such an approach, we were able to implement the parameterization of queries quite easily. In our project, queries 1,2 and 6 can be configured. In the display of the results, we have implemented pagination with a selection of pages and the number of elements per page.



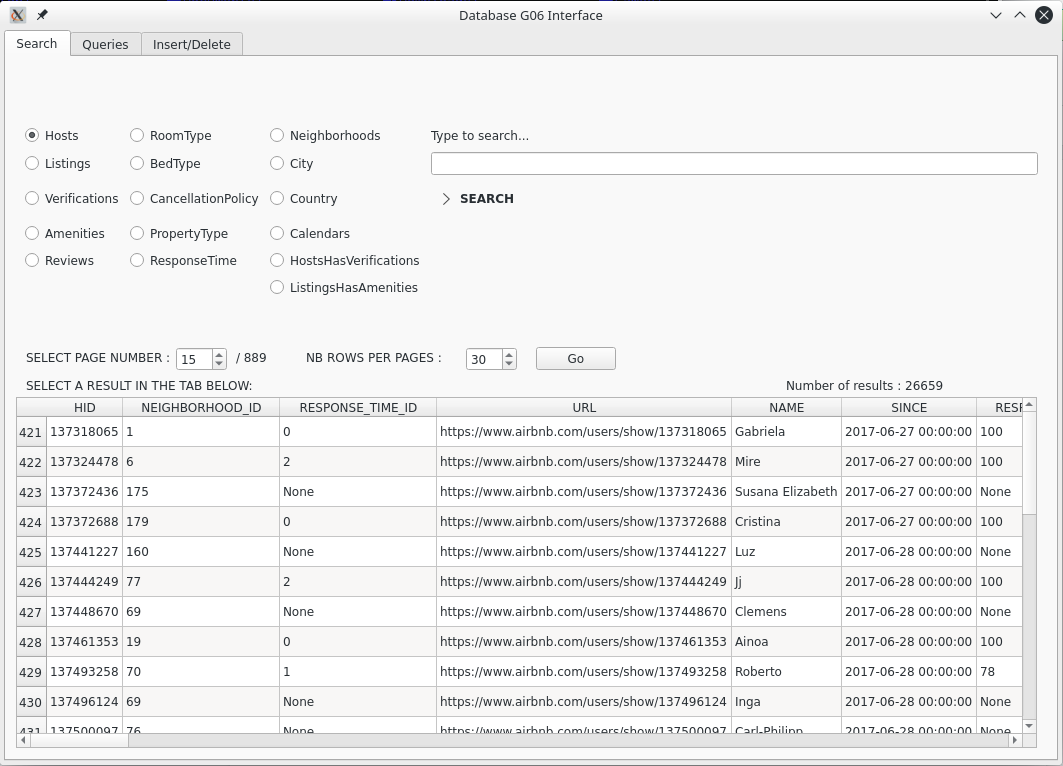
Here is an example of how to set up query 1 where you can choose the number of bedrooms.



In this tab, you can choose the table to modify on the left (and validate by pressing Select) in order to be able to load characteristics of the table to make a correct insertion or deletion. If the operation was successful, a confirmation message appears and if not, an error message pops up.

Additional comments: The delete action cans delete in cascade when it is allowed in the database (See tables definitions in Part I). For the insert Part, we designed the interface to be as much user friendly as possible. When we need to enter a data for a foreign key or when we expect a range of numbers, we use combo boxes such that the user can’t choose invalid data. We did the same trick with date entries. Also we avoid the user to enter text when we expect numbers. Now, if an ID for a table already exists, we display an error message.

For the Search tab, you have to choose a table in which you want to search and it will search for all the rows that contain what has been entered in the field, regardless of the column.



# 

# General Comments

The interface was stressful part since we had to start again from scratch for this deliverable when we realised that a web-oriented solution was not realistically feasible for us.

If the choice had to be made again, we would probably have chosen something other than OracleSQL. Indeed, we encountered difficulties during the interface implementation because of incompatible technologies and it seemed much easier with something like MySQL.

Once again, we made sure to work together while designing central things like the queries and indexes, while we did some other things more independently like the implementation of the interface.

Overall we’re satisfied both with the learning experience and with the final product we created.