École polytechnique fédérale de Lausanne

Introduction to Database Systems
CS-322

Database Project

Fall 2019 — Airbnb

Team 11
Jollès Eric, Mamie Robin, Montial Charline

31st May 2019

EPFL

Contents

EPFL

Chapter 1

ER model and DDL

1.1 Assumptions

The majority of assumptions we made can be seen in our ER model. We made sensible assumptions about listings which will be discussed in the next section. Also, these assumptions have been made after having studied the data.

1.2 Entity Relationship Schema

In the ER-model (figure 1.1), the attribute numbers are given in appendix A. All attribute numbers not given in the ER-model belong to Listing.

First of all, it was pretty clear that the central entity would be the `Listing`. The `Listing` must be owned by exactly one `Host` entity and must be situated in precisely one `Neighbourhood`, which is also an entity.

Since it was not concretely helping nor useful to split the large amount of attributes for the entity `Listing`, they are regrouped.

Moreover, while implementing the required queries, we decided to have the amenities in a separate list. Thus we now have an `Amenity` table. Moreover, to establish the link between a particular listing and its amenities, a table `has_amenity` has been created.

Based on the same model, the entity `Host` is also linked to a new table `Host_verifications` through the table `has_host_verifications`.

The entities `Cancellation_policy`, `Room_type`, `Property_type`, `Country` and `Bed_type` have been added to avoid repetitions through the listings which may refer to the same policy, for example.

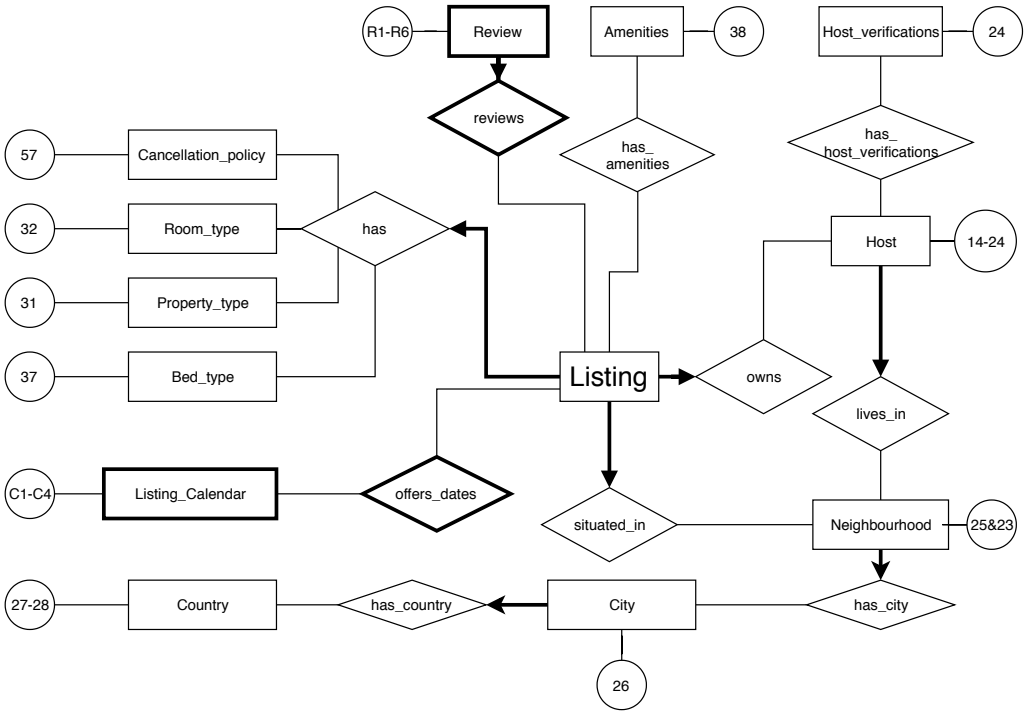

Figure 1.1: The ER model of our project.

A `Host` may not have any listing and can still remain in the database system but must live in exactly one `Neighbourhood`.

A `Review` must refer to exactly one `Listing`, otherwise it has no purpose.

Finally, we linked the neighbourhood, the city and the Neighbourhood together since each neighbourhood has one city which has one country.

1.3 Relational Schema

First, a script in Python has been written – `length_finder.ipynb`, which computes the maximum length of each field. We did not want to reserve a space of 1000 characters if the maximum length of this field was only of, for example, 300 characters.

About the code to create SQL tables, we naturally did our best to enforce all conditions of our ER model and use sensible types. For instance, the `DATE` type has been used for dates, `INTEGER` for ids and `FLOAT` for prices. Also, instead of `CHAR` fields, we put `VARCHAR2` ones for better performance.

As previously stated, we observed that in some fields, some values were repeated in different rows. We thus decided to normalise by creating some linked entities in the database. The fields chosen are:

- host_response_time
- room_type
- property_type
- bed_type
- cancellation_policy
- Neighbourhood
- City
- Country (which contains the country and the country code)
- Amenities
- Host_Verifications

The last two fields, since they are lists, also have an additional table which corresponds to a relationship respectively between Listing and Amenities and between Host and Host_Verifications.

Moreover, for all weak entities which compose the listing as stated in the ER model figure, we added the condition ON DELETE CASCADE. Indeed, all these weak entities must be deleted in case the listing they compose is deleted. These entities are Review and Listing_calendar.

1.3.1 SQL code

```
CREATE TABLE country (
    country_id      INTEGER,
    country         VARCHAR2(7),
    country_code    CHAR(2),
    PRIMARY KEY ( country_id )
);

CREATE TABLE city (
    city_id       INTEGER,
    city          VARCHAR2(40),
    country_id    INTEGER,
    PRIMARY KEY ( city_id ),
    FOREIGN KEY ( country_id )
        REFERENCES country ( country_id )
);

CREATE TABLE neighbourhood (
    nid              INTEGER,
    neighbourhood    VARCHAR2(40),
    city_id          INTEGER,
    PRIMARY KEY ( nid ),
    FOREIGN KEY ( city_id )
        REFERENCES city ( city_id )
);

CREATE TABLE bed_type (
    btid        INTEGER,
    bed_type    VARCHAR2(13),
    PRIMARY KEY ( btid )
);
```

```
30
CREATE TABLE cancellation_policy (
    cpid                INTEGER,
    cancellation_policy VARCHAR2(27),
    PRIMARY KEY ( cpid )
35 );

CREATE TABLE host_response_time (
    hrtid               INTEGER,
    host_response_time  VARCHAR2(18),
40  PRIMARY KEY ( hrtid )
);

CREATE TABLE property_type (
    ptid          INTEGER,
45  property_type VARCHAR2(22),
    PRIMARY KEY ( ptid )
);

CREATE TABLE room_type (
50  rtid      INTEGER,
    room_type VARCHAR2(15),
    PRIMARY KEY ( rtid )
);

55 CREATE TABLE host (
    host_id          INTEGER,
    host_name        VARCHAR2(40),
    url              VARCHAR2(43),
    since            DATE,
60  about            VARCHAR2(4000),
    response_time    INTEGER,
    response_rate    INTEGER,
    thumbnail_url    VARCHAR2(120),
    picture_url      VARCHAR2(120),
65  nid              INTEGER,
    verifications    VARCHAR2(170),
    PRIMARY KEY ( host_id ),
    FOREIGN KEY ( response_time )
        REFERENCES host_response_time ( hrtid ),
70  FOREIGN KEY ( nid )
        REFERENCES neighbourhood
);

CREATE TABLE listing (
75  id                                INTEGER,
    listing_url                       VARCHAR2(40),
    name                              VARCHAR2(150),
    summary                           VARCHAR2(1000),
    space                             VARCHAR2(1000),
80  description                       VARCHAR2(1000),
    neighborhood_overview             VARCHAR2(1000),
    notes                             VARCHAR2(1000),
    transit                           VARCHAR2(1000),
    l_access                          VARCHAR2(1000),
85  interaction                       VARCHAR2(1000),
    house_rules                       VARCHAR2(1000),
    picture_url                       VARCHAR2(120),
```

```
    host_id                         INTEGER,
    --neighbourhood_id
    nid                             INTEGER,
    latitude                        FLOAT,
    longitude                       FLOAT,
    --property_type_id
    ptid                            INTEGER,
    --room_type_id
    rtid                            INTEGER,
    accommodates                    INTEGER,
    bathrooms                       FLOAT,
    bedrooms                        INTEGER,
    beds                            INTEGER,
    --bed_type id
    btid                            INTEGER,
    square_feet                     INTEGER,
    price                           FLOAT,
    weekly_price                    FLOAT,
    monthly_price                   FLOAT,
    security_deposit                FLOAT,
    cleaning_fee                    FLOAT,
    guests_included                 INTEGER,
    extra_people                    FLOAT,
    minimum_nights                  INTEGER,
    maximum_nights                  INTEGER,
    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,
    is_business_travel_ready        CHAR(1),
    --cancellation_policy_id
    cpid                            INTEGER,
    require_guest_profile_picture   CHAR(1),
    require_guest_phone_verification CHAR(1),
    PRIMARY KEY ( id ),
    FOREIGN KEY ( host_id )
        REFERENCES host ( host_id ),
    FOREIGN KEY ( ptid )
        REFERENCES property_type ( ptid ),
    FOREIGN KEY ( rtid )
        REFERENCES room_type ( rtid ),
    FOREIGN KEY ( btid )
        REFERENCES bed_type ( btid ),
    FOREIGN KEY ( cpid )
        REFERENCES cancellation_policy ( cpid ),
    FOREIGN KEY ( nid )
        REFERENCES neighbourhood ( nid )
);

CREATE TABLE review (
    rid             INTEGER,
    listing_id      INTEGER NOT NULL,
    reviewer_id     INTEGER,
    reviewer_name   VARCHAR2(60),
    rdate           DATE,
```

```
    comments          VARCHAR2(4000),
    PRIMARY KEY ( rid ),
    FOREIGN KEY ( listing_id )
        REFERENCES listing ( id )
            ON DELETE CASCADE
);

CREATE TABLE listing_calendar (
    listing_id    INTEGER,
    cdate         DATE,
    available     CHAR(1),
    price         FLOAT,
    FOREIGN KEY ( listing_id )
        REFERENCES listing ( id )
            ON DELETE CASCADE
);

CREATE TABLE amenity (
    aid        INTEGER,
    amenity    VARCHAR2(50),
    PRIMARY KEY ( aid )
);

CREATE TABLE host_verification (
    hvid                 INTEGER,
    host_verification    VARCHAR2(30),
    PRIMARY KEY ( hvid )
);

CREATE TABLE has_host_verification (
    listing_id    INTEGER,
    hvid          INTEGER,
    FOREIGN KEY ( listing_id )
        REFERENCES listing ( id )
            ON DELETE CASCADE,
    FOREIGN KEY ( hvid )
        REFERENCES host_verification ( hvid )
            ON DELETE CASCADE
);

CREATE TABLE has_amenity (
    listing_id    INTEGER,
    aid           INTEGER,
    FOREIGN KEY ( listing_id )
        REFERENCES listing ( id )
            ON DELETE CASCADE,
    FOREIGN KEY ( aid )
        REFERENCES amenity ( aid )
            ON DELETE CASCADE
);
```

1.4 General comments

1.4.1 Work allocation between team members

We naturally began to work on the ER model and what we did is that every team member had to present an ER model. The aim of this was to discuss differences we had and take the best out of the three versions. Then, for the tables, Eric wrote the `length_finder` script in Python. About the DDL code, the work has been split between Robin and Charline and the code has been mutually improved. Each of us have also contributed to writing this report.

Chapter 2

First SQL Requests and Interface

2.1 Assumptions and Data Loading

We decided to uniform the city entity, indeed, we stated that all listing which are in the same file belongs to the same city, to avoid different city spelling.

We also changed the type of some fields, indeed we converted the percent values and prices (in $) into `float`s. We also decided to keep all rows in calendar which had null values in prices since they are useful for some queries (Query 10 of second milestone).

Furthermore, to save storage, we decided to only keep substrings for some fields, since they often are descriptions, with redundant information (often contains translations) and not useful for queries. Since not many elements exceed these limits, it is not a big issue. These fields are:

- In `Listing`: `Neighbourhood_overview`
- In `Review`: `comments`

We also considered that all `Listing`s in a file belongs to the file `City`; we made this assumption since there were too many different city names (some with typos) in the same file.

2.2 Query Implementation

For all queries which implied prices, we considered the prices of all listings, available or not.

We reported the running tome for each query. To do so, we have run each queries twenty times and computed the mean. The queries may need a warm-up to give a relevant result. Indeed, the first results may be significantly longer and thus have not been considered.

2.2.1 Query 1

Description of logic

We are looking for the average price of all listings which have 8 bedrooms. We solved this by using the key word `AVG` and adding the condition enforcing that the listing contains 8 bedrooms.

SQL statement

```
SELECT ROUND ( AVG (l.price) , 2)
FROM Listing l
WHERE l.bedrooms = 8
```

Result

The average price is:

```
313.15
```

Running time

The mean of the twenty measures of the running times is of 83.8 ms.

2.2.2 Query 2

Description of logic

The query looks for all listings which propose a TV and computes the average cleaning review of this selection. We only looked for the keyword `TV` in amenities. Even though it can be stated in the small or longer description that there is a television, it is strictly needed to be specified in amenities by definition. This is why we only considered this field. This was our assumption to solve this query. Since `Amenity` is a list which contains all available amenities, we had to see if `TV` was part of the amenities of the listing and to establish the link between the listing and the amenities, `Has_amenity` has been used.

SQL statement

```
SELECT ROUND ( AVG(L.review_scores_cleanliness) , 2)
FROM Listing L,
  Has_amenity H,
  Amenity A
WHERE A.amenity  = 'TV'
AND H.aid          = A.aid
AND H.listing_id = L.id
```

Result

The average cleaning review score is:

9.4

Running time

The mean of the twenty measures of the running times is of 130.2 ms.

2.2.3 Query 3

Description of logic

The query selects all the names of the hosts who have at least one listing between the provided dates. To solve this query, we retrieved the date information in the calendar table and established the link to the host table through the listing one. The dates had to be formatted correctly to be interpreted the way we wanted.

SQL statement

```
SELECT DISTINCT H.host_name
FROM Listing L, Host H
WHERE H.host_id = L.host_id
AND L.id        IN
  ( SELECT DISTINCT listing_id
  FROM Listing_calendar
  WHERE cdate >= '01.03.19'--'2019-03-01'
  AND cdate   <= '01-09-19'--'2019-09-01'
  AND available = 't'
  );
```

Result

1. Antonio
2. Kristjan Y Ana
3. Mar
4. Jaume
5. Jesus

Running time

The mean of the twenty measures of the running times is of 774.35 ms.

2.2.4 Query 4

Description of logic

The query counts the number of listings whose host has the same name as another host – they must be different hosts. The other host must have at least one listing. To solve this, we matched 2 pairs of `Listing` entities with `Host` entities. Once a listing with the correct

condition is found, it finds the name of the host given a listing. This checks if the names of the hosts are equal even though they are not the same person.

SQL statement

```
SELECT COUNT(L.id)
FROM Listing L
WHERE L.host_id IN
  ( SELECT DISTINCT H1.host_id
 FROM Host H1,
    Host H2
  WHERE H1.host_id != H2.host_id
  AND H1.host_name = H2.host_name
  );
```

Result

30393 listings fulfil this condition.

Running time

The mean of the twenty measures of the running times is of 105.85 ms.

2.2.5 Query 5

Description of logic

The query looks for dates of listing whose host is `Viajes Eco`. We decided to solve it by using the `Listing` to establish the link between the `Listing_calender` table and `Host` table. We then find dates of listing whose host is `Viajes Eco` – without forgetting to ensure that the listings proposed are available.

SQL statement

```
SELECT C.cdate
FROM Listing L,
  Host H,
  Listing_calendar C
WHERE L.host_id   = H.host_id
AND H.host_name   = 'Viajes Eco'
AND C.listing_id = L.id
AND C.available   = 't';
```

Result

1. 03.03.19
2. 02.03.19
3. 01.03.19
4. 28.02.19
5. 27.02.19

Running time

The mean of the twenty measures of the running times is of 617 ms.

2.2.6 Query 6

Description of logic

The query finds all hosts that only have a single listing. We decided to solve it by using a nested query. We print all host ids and host names for which the number of listings is exactly equal to one.

SQL statement

```
SELECT host_id, host_name
FROM Host
WHERE host_id IN
  ( SELECT host_id FROM Listing GROUP BY host_id HAVING COUNT(*)=1
  );
```

Result

1. 431839 Xavier
2. 95585 Daniela
3. 48815 Pols
4. 509260 Dalila
5. 66419 Teresa

Running time

The mean of the twenty measures of the running times is of 84.535 ms.

2.2.7 Query 7

Description of logic

It computes the subtraction between the average price of listings with Wifi minus the average price of listings without. We directly subtract the two separate queries using the amenities list as previously done for the 2nd request to solve the query.

SQL statement

```
SELECT ROUND (ABS (
  (SELECT AVG(L.price)
  FROM Listing L
  WHERE L.id IN
    (SELECT H.listing_id
    FROM Has_amenity H,
      Amenity A
```

```
    WHERE A.amenity = 'Wifi'
    AND H.aid       = A.aid
    )
  ) -
  (SELECT AVG(L.price)
  FROM Listing L
  WHERE L.id NOT IN
    (SELECT H.listing_id
    FROM Has_amenity H,
      Amenity A
    WHERE A.amenity = 'Wifi'
    AND H.aid       = A.aid
    )
  )) , 2 )
FROM DUAL;
```

Result

The difference in the average price of listings with and without Wifi is:

```
3.21
```

Running time

The mean of the twenty measures of the running times is of 268.8 ms.

2.2.8 Query 8

Description of logic

It computes the difference between the average price of a listing offering 8 bedrooms in Berlin and the average price of a listing offering 8 beds in Madrid. We solved this by subtracting the two average prices, selecting all listings with 8 beds from Madrid, and then Berlin.

SQL statement

```
SELECT ROUND ( ABS (
  (SELECT AVG(L.price)
  FROM Listing L,
    City C,
    Neighbourhood N
  WHERE L.beds  = 8
  AND L.nid     = N.nid
  AND N.city_id = C.city_id
  AND C.city    = 'Berlin'
  ) -
  (SELECT AVG(L.price)
  FROM Listing L,
    City C,
    Neighbourhood N
  WHERE L.beds  = 8
  AND L.nid     = N.nid
```

```
  AND N.city_id = C.city_id
  AND C.city    = 'Madrid'
  ) ), 2)
FROM DUAL;
```

Result

```
101.59
```

Running time

The mean of the twenty measures of the running times is of 150.8 ms.

2.2.9 Query 9

Description of logic

It selects the 10 hosts who have the highest number of listings in Spain. To solve this, we grouped all listings by their hosts ids. We then ordered them in a decreasing order and took the top 10. We had to perform another manipulation to not only retrieve the host ids, but also the host names.

SQL statement

```
SELECT H.host_id, H.host_name
FROM Host H
WHERE H.host_id IN (SELECT  L.host_id
FROM Listing L, City C1, Country C2, Neighbourhood N
WHERE L.nid = N.nid
  AND N.city_id = C1.city_id
AND C1.country_id = C2.country_id
AND C2.country = 'Spain'
GROUP BY L.host_id
ORDER BY COUNT(*) DESC
FETCH FIRST 10 ROWS ONLY);
```

Result

1. 1391607 Aline
2. 28038703 Luxury Rentals Madrid
3. 32046323 Juan
4. 299462 Stay U-Nique
5. 1408525 Mad4Rent

Running time

The mean of the twenty measures of the running times is of 90.95 ms.

2.2.10 Query 10

Description of logic

It selects the 10 apartments that have the best review score rating in Barcelona. We have simply selected the apartments that are in Barcelona, ordered them according to their rating, and took the top 10.

SQL statement

```
SELECT  L.id, L.name
FROM Listing L,
  City C,
    Neighbourhood N,
    Property_type pt
WHERE L.nid  = N.nid
  AND N.city_id = C.city_id
AND C.city = 'Barcelona'
AND L.ptid = pt.ptid
AND pt.property_type = 'Apartment'
AND L.review_scores_rating IS NOT NULL
ORDER BY L.review_scores_rating DESC
FETCH FIRST 10 ROWS ONLY;
```

Result

1.	71520	Charming apartment with fantastic views!
2.	11997102	Double Room - El Raval, Barcelona
3.	590991	Beautiful Cheap Double NEAR BEACH!!
4.	337755	SEALONA VILA OLIMPICA BEACH
5.	286105	Room at Gran Via Barcelona Spain

Running time

The mean of the twenty measures of the running times is of 94.1 ms.

2.3 Interface

The interface was written using Scala, and mainly the ScalaFX library. It can be run by typing `sbt run` in the interface folder. The communication between the interface and the Oracle database is done using JDBC. It is composed of 4 main panels:

- **Welcome** Indicates the main functionalities of the program.
- **Search** Allows the user to look for any key word in any table he chooses from.
- **Queries** Allows the user to interactively explore the required queries of this project.
- **Insert/Delete** Allows the user to add or remove any data from the database.

The welcome panel also holds a parameter used to fix the maximum number of results displayed per query. It is used to avoid unnecessary waits when loading the data to the

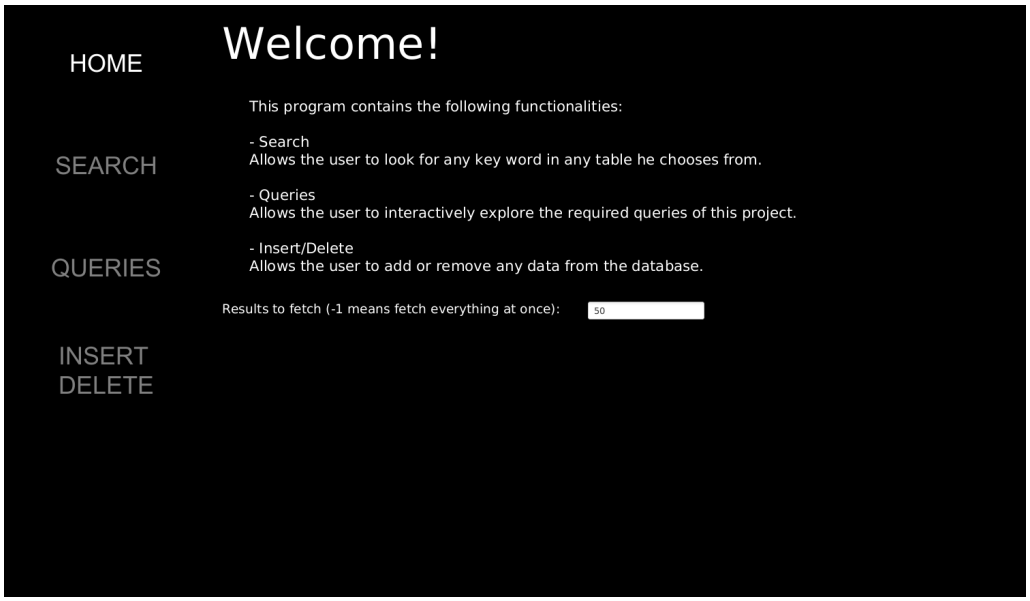

Figure 2.1: The Welcome panel of the interface.

interface, which takes a non zero amount of time.

To set up the table names, the program sends the following query:

```
0 SELECT table_name
  FROM user_tables
```

Then, to retrieve all their attributes, the program sends a query per table. Here is the request sent to retrieve the attributes of Neighbourhood.

```
0 SELECT column_name
  FROM user_tab_columns
  WHERE table_name = 'NEIGHBOURHOOD'
```

2.3.1 Search

The attributes of each table are then used for the search function. Here is the query when searching for `San` in the Neighbourhood table:

```
0 SELECT NID
  FROM NEIGHBOURHOOD
  WHERE NID LIKE '%San%' OR NEIGHBOURHOOD LIKE '%San%'
```

The interface then shows a button per table searched, so that the user can display them on screen – showed in figure 2.2.

The keys are stored in memory, and the results are retrieved once clicking on the button `Show`. After clicking, the following query is sent:

Figure 2.2: The Search panel of the interface.

```
0 SELECT *
FROM NEIGHBOURHOOD
WHERE  NID = '11' OR NID = '18' OR NID = '19' OR NID = '23' OR NID = '28' OR
    NID = '29' OR NID = '43' OR NID = '47' OR NID = '52' OR NID = '57' OR NID =
     '66' OR NID = '166'
```

As a side note, all these queries are built using Scala methods on `List`, assuring efficient string builds.

This spawns the window showed in figure 2.3 for the neighbourhood table. For fields having bigger texts, the line breaks at the first space after 100 characters.

2.3.2 Queries

Figure 2.4 shows the window allowing the user to send the predefined queries. The queries sent are the same as the ones described in sections 2.2 and 3.1. Two boxes at the bottom of the screen allow the user to parametrize queries number 1 and 5.

All queries run the exact same code as presented in this document, except for the date formatting, which is different because the original one did not work in this instance. Views are accustomed by sending first a query to create them, and then the main query in itself is sent.

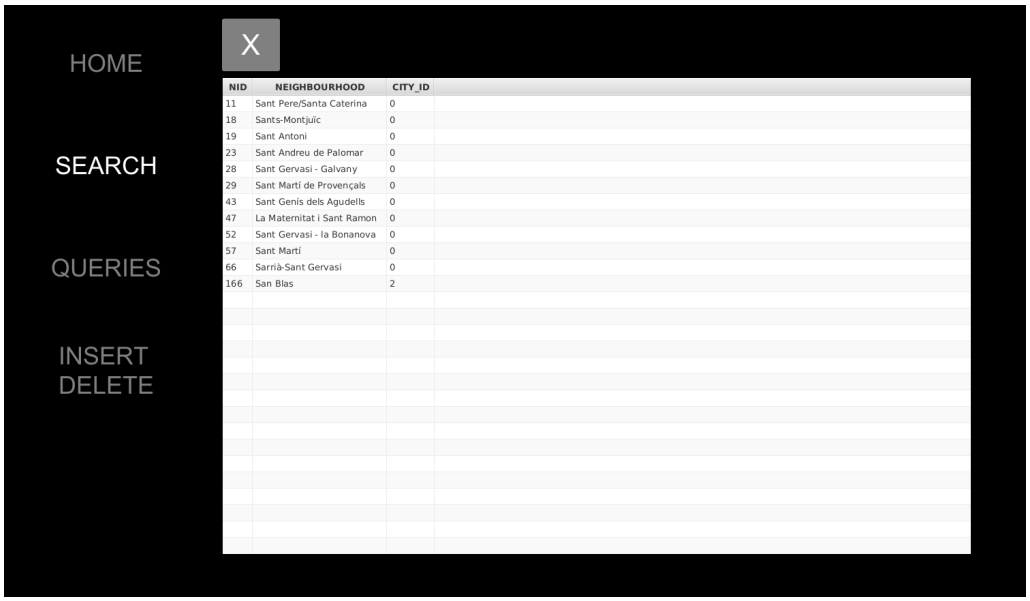

Figure 2.3: The result of the Search displayed.

2.3.3 Insertion and Deletion

The insertion and deletion window spawns a menu on the left, allowing the user to choose which table to modify. Then, on the right, all the fields concerning the chosen table are present, as well as an `Insert` and a `Delete` button – see figure 2.5.

An insertion for the city table would send e.g. the following query:

```
INSERT INTO CITY VALUES ('3', 'Hamburg', '1')
```

And a deletion:

```
DELETE FROM CITY WHERE CITY_ID='3' AND CITY='Hamburg' AND COUNTRY_ID='1'
```

Or, alternatively, if only one field is mentioned:

```
DELETE FROM CITY WHERE CITY_ID='3'
```

2.4 General Comments

2.4.1 Work allocation between team members

Eric worked on the data parsing and its insertion into the database. Charline wrote the SQL requests, with close collaboration with all team members. Robin designed and created the graphical interface and its related SQL queries.

Figure 2.4: The queries panel of the interface, after executing query nb. 1 with 7 bedrooms

2.4.2 Issues

We did not manage to give the `TEXT` type to some attributes as advised in the previous feedback. SqlDeveloper returned the following error when we tried to do it:

```
SQL Error : ORA-00902: invalid datatype
```

For selecting the first 5 rows we chose to use ”FETCH FIRST 5 ROWS ONLY”, we didn’t use ”limit” option of ”order by” since it seems to not exist in Oracle SQL queries.

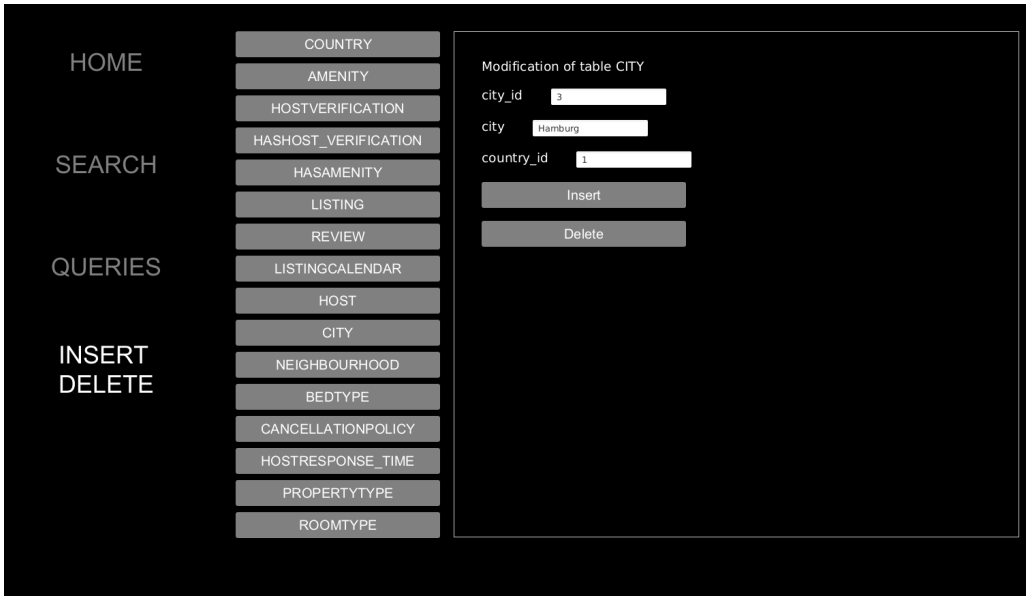

Figure 2.5: The insertion and deletion window

Chapter 3

More SQL Requests and Indexing

3.1 Query Implementation

We reported the running tome for each query. To do so, we have run each queries twenty times and computed the mean. Since the queries may need a warm-up to give a relevant result, the first runs have not been considered.

3.1.1 Query 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.

Description of logic

As a first step we select listings with existing square feet, we group them by their city and their host id, thus we have pairs of cities and host_id, we then count how many pairs per city we have, to finally output the result order by their city name.

SQL statement

```
SELECT C.city, cnt
FROM City C,
  (SELECT cid1 AS cid,
    COUNT(*)   AS cnt
  FROM
    (SELECT N.city_id AS cid1
    FROM Listing L, Neighbourhood N
    WHERE L.square_feet IS NOT NULL
    AND L.nid = N.nid
    GROUP BY N.city_id,
      L.host_id
    )
  GROUP BY cid1
  )
```

```
WHERE cid = C.city_id
ORDER BY C.CITY;
```

Result

	City	# of host
1.	Barcelona	345
2.	Berlin	370
3.	Madrid	245

Running time

The mean of the twenty measures of the running times is of 86 ms.

3.1.2 Query 2

The quality of a neighbourhood 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 neighbourhoods using median review scores (review_scores_rating) of listings in Madrid. Note: Implement the median operator on your own, and do not use the available built-in operator.

Description of logic

We first group listings by neighbourhood and order them (we store their rank) by review_scores_rating in the neigh_listing view. We then retrieve the position of the median for every neighbourhood and retrieve with the view the median value (with median formula).

SQL statement

```
CREATE OR REPLACE VIEW neigh_listing AS
SELECT L.id,
  L.nid AS nid,
  L.review_scores_rating,
  row_number() over ( partition BY L.nid order by L.review_scores_rating DESC)
    row_number
FROM Listing L
WHERE review_scores_rating IS NOT NULL;


SELECT N.neighbourhood,
  S.median
FROM
  Neighbourhood N,
  (SELECT co.nid,
    (nl1.review_scores_rating + nl2.review_scores_rating)/2 AS median
  FROM neigh_listing nl1,
    neigh_listing nl2,
    (SELECT L.nid           AS nid,
      FLOOR((COUNT(*)+1)/2) AS low,
```

```
      CEIL((COUNT( *)+1)/2) AS high
    FROM Listing L
    WHERE L.review_scores_rating IS NOT NULL
    GROUP BY L.nid
    ) co
  WHERE co.nid        = nl1.nid
  AND co.nid          = nl2.nid
  AND nl1.row_number = co.high
  AND nl2.row_number = co.low
  ) S
WHERE N.nid = S.nid
ORDER BY median DESC
FETCH FIRST 5 ROWS ONLY;
```

Result

	Neighbourhood	Median
1.	Rahnsdorf	100
2.	Hakenfelde	100
3.	Charlottenburg-Nord	100
4.	Wilhelmsruh	100
5.	Konradshöhe	100

Running time

The mean of the twenty measures of the running times is of 120.35 ms.

3.1.3 Query 3

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

Description of logic

For this query, we first created a view which group listings by their host (host_id). Then we keep hosts, who have has the same number of listings as the maximum.

SQL statement

```
CREATE OR REPLACE VIEW host_list_c AS
SELECT L.host_id AS hid , COUNT(*) AS cnt FROM Listing L GROUP BY L.host_id ;


SELECT h.host_id,
  h.host_name
FROM host_list_c hlc,
  Host h
WHERE hlc.cnt =
  (SELECT MAX(hlc2.cnt) FROM host_list_c hlc2
  )
AND h.host_id = hlc.hid ;
```

Result

	Host id	Host
1.	4459553	Eva&Jacques

Running time

The mean of the twenty measures of the running times is of 177.3 ms.

3.1.4 Query 4

Find the 5 most cheapest Apartments (based on average price within the available dates) in Berlin available for at least one day between 01-03-2019 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.

Description of logic

We first find all the listings which were available for at least one day between 01-03-2019 and 30-04-2019 and compute their average price. We then apply all other constraint to listings and find the 5 with the cheapest average price.

SQL statement

```
SELECT L.id,
  Round(lprice,2)
FROM Listing L,
  City C,
  Neighbourhood N,
  Property_type pt,
  Cancellation_policy CP,
  HAS_HOST_VERIFICATION HHV,
  HOST_VERIFICATION HV,
  (SELECT C.listing_id AS lid ,
    AVG(C.price)       AS lprice
  FROM Listing_calendar C
  WHERE C.cdate  >= '01.03.19'--'2019-03-01'
  AND C.cdate    <= '30-04-19'--'2019-09-01'
  AND C.available = 't'
  GROUP BY C.listing_id
  )
WHERE L.id = lid
  -- # Beds >= 2
AND L.beds >= 2
  -- REVIEW_SCORES_RATING >= 8
AND L.REVIEW_SCORES_RATING >= 8.0
  -- City : Berlin
AND C.city     = 'Berlin'
AND N.nid = L.nid
AND N.city_id = C.city_id
AND L.ptid = pt.ptid
AND pt.property_type = 'Apartment'
  -- Cancellation_policy : flexible
AND CP.CANCELLATION_POLICY = 'flexible'
AND CP.cpid                = L.cpid
  -- host_verification : government_id
```

```
AND HV.HOST_VERIFICATION = 'government_id'
AND HHV.hvid             = HV.hvid
AND HHV.listing_id       = L.id
  --search the 5 cheapest
ORDER BY lprice
FETCH FIRST 5 ROWS ONLY ;
```

Result

	Listing id	Price
1.	1490274	20
2.	24043706	21.07
3.	1368460	21.29
4.	7071541	22
5.	6691656	22

Running time

The mean of the twenty measures of the running times is of 673.89 ms.

3.1.5 Query 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.

Description of logic

We first Select the listings which have all the necessary amenities. We group them by the number of accommodates and sort in each group listings by their review_scores_rating. We then only keep the top-5 rated in each group.

SQL statement

```
SELECT *
FROM
  (SELECT L.id,
    L.accommodates,
    L.review_scores_rating,
    row_number() over ( partition BY L.accommodates order by L.
    review_scores_rating DESC) row_number
  FROM Listing L
  WHERE review_scores_rating IS NOT NULL
  AND L.id                   IN
    (SELECT HA.LISTING_ID
    FROM HAS_AMENITY HA
    WHERE HA.aid IN
      (SELECT A.aid
      FROM AMENITY A,
        HAS_AMENITY HA
```

```
        WHERE A.AMENITY = 'Wifi'
        OR A.AMENITY    = 'Internet'
        OR A.AMENITY    = 'TV'
        OR A.AMENITY    = 'Free street parking'
        )
    GROUP BY HA.listing_id
    HAVING COUNT(*)>= 2
    )
  )
WHERE row_number <= 5;
```

Result

	Listing id	# of accommodates	Rating (%)	Rank
1.	10742139	1	100	1
2.	26150481	1	100	2
3.	26146463	1	100	3
4.	26085997	1	100	4
5.	28268567	1	100	5

Running time

The mean of the twenty measures of the running times is of 238 ms.

3.1.6 Query 6

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

Description of logic

We first count the number of reviews per listing. Then we group them by their Host and order them by their number of reviews. We keep for each host the top three busiest listings.

SQL statement

```
SELECT H.host_name, S.Lid
FROM
  Host H,
  (SELECT L.host_id,
    lid,
    cnt,
    row_number() over ( partition BY L.host_id order by cnt DESC) row_number
  FROM Listing L,
    (SELECT listing_id AS lid, COUNT(*) AS cnt FROM Review GROUP BY listing_id
    )
  WHERE review_scores_rating IS NOT NULL
  AND L.id                   = lid
  ) S
WHERE row_number <=3
AND H.Host_id = S.Host_id;
```

Result

	Host	Listing
1.	Ian	2015
2.	Ian	21315310
3.	Ian	18773184
4.	Ricard	6287375
5.	Britta	3176

Running time

The mean of the twenty measures of the running times is of 396.25 ms.

3.1.7 Query 7

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

Description of logic

We first group all the listing by their neighbourhood and their amenities. By counting the number of elements we have the number of a certain type of amenity per neighbourhood. We then can group the result by neighbourhood and take for each neighbourhood the three most frequently used amenities.

SQL statement

```
SELECT N.neighbourhood, A.amenity
FROM
  Neighbourhood N,
  Amenity A,
  (SELECT Selector.nid,
    Selector.aid,
    cnt ,
    row_number() over ( partition BY Selector.nid order by cnt DESC) row_number
  FROM
    (SELECT L.nid,
      HA.aid,
      COUNT(*) AS cnt
    FROM Has_amenity HA,
      Listing L,
      Room_type RT,
      City C,
      Neighbourhood N
    WHERE L.id        = HA.listing_id
    AND L.rtid        = RT.rtid
    AND RT.room_type = 'Private room'
    AND N.nid     = L.nid
    AND N.city_id = C.city_id
    AND C.city        = 'Berlin'
    GROUP BY L.nid,
      HA.aid
    ) Selector
```

```
  ) S
WHERE row_number <=3
AND S.aid = A.aid
ANd S.nid = N.nid;
```

Result

	Neighbourhood	Amenity
1.	Kreuzberg	Wifi
2.	Kreuzberg	Kitchen
3.	Kreuzberg	Heating
4.	Friedrichshain	Wifi
5.	Friedrichshain	Kitchen

Running time

The mean of the twenty measures of the running times is of 141.4 ms.

3.1.8 Query 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 verifying hosts, pick a host one from the most and one from the least verifying hosts.

Description of logic

We first compute the number of amenity of each listing and put the result in a view. Then we find one listing with the min number of amenity and one with the maximum and we make the difference between their review communication scores.

SQL statement

```
CREATE OR REPLACE VIEW amenity_list_c AS
SELECT listing_id, COUNT(*) AS cnt FROM Has_amenity GROUP BY listing_id ;


SELECT L1.REVIEW_SCORES_COMMUNICATION - L2.REVIEW_SCORES_COMMUNICATION
FROM Listing L1,
  Listing L2
WHERE L1.id IN
  (SELECT alc1.listing_id
  FROM amenity_list_c alc1
  WHERE alc1.cnt =
    (SELECT MAX(alc2.cnt) FROM amenity_list_c alc2
    )
  FETCH FIRST 1 ROWS ONLY
  )
AND L2.id IN
  (SELECT alc1.listing_id
  FROM amenity_list_c alc1
```

```
  WHERE alc1.cnt =
    (SELECT MIN (alc2.cnt) FROM amenity_list_c alc2
    )
  FETCH FIRST 1 ROWS ONLY
  ) ;
```

Result

	Difference
1.	2

Running time

The mean of the twenty measures of the running times is of 455 ms.

3.1.9 Query 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.

Description of logic

We first compute the number of amenities per listing, then we group the listing by room type, we keep the room type groups which have an average number of accommodates greater than 3. We then group the review of listing which have such room type by their city. And then we have to pick the city with the maximum number of such reviews.

SQL statement

```
SELECT C.city
FROM city C,
  (SELECT city_id,
    COUNT(*) AS cnt
  FROM Listing L ,
  Neighbourhood N,
    review R
  WHERE L.rtid IN
    (SELECT rtid
    FROM Listing L,

      (SELECT HA.listing_id,
        COUNT(*) AS cnt
      FROM Has_amenity HA
      GROUP BY ha.listing_id
      )
    WHERE L.id = listing_id
    GROUP BY rtid
    HAVING AVG(cnt) >= 3
    )
  AND R.listing_id = L.id
  AND N.nid = L.nid
  GROUP BY N.city_id
  ) T
```

```
WHERE C.city_id = T.city_id
ORDER BY cnt DESC
FETCH FIRST 1 ROWS ONLY;
```

Result

	City
1.	Madrid

Running time

The mean of the twenty measures of the running times is of 550 ms.

3.1.10 Query 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 01.06.2017

Description of logic

We first Create a view of listings which are in Madrid. Then we find the neighbourhoods whose all their hosts has joined airbnb before 01.06.2017. Then we count for each such neighbourhood the number (N_part) of listings occupied once in 2019. We also compute the total number (N_tot) of listings per neighbourhood. At the end we make the ratio between the 2 numbers (N_part/N_tot) for each neighbourhood and filter those with this percentage greater than 50% .

SQL statement

```
CREATE OR REPLACE VIEW madrid_listing AS
SELECT L.id                AS listing_id,
  L.nid                    AS nid,
  L.host_id
FROM Listing L,
Neighbourhood N,
  City C
WHERE L.nid = N.nid
AND N.city_id = C.city_id
AND C.city      = 'Madrid';


SELECT N.neighbourhood
FROM
  Neighbourhood N,
  (SELECT L.nid                  AS nid ,
    COUNT(DISTINCT L.listing_id) AS cnt
  FROM Listing_calendar C,
    madrid_listing L
  WHERE extract(YEAR FROM C.cdate) = 2019
  AND L.listing_id                 = C.listing_id
  AND C.available                  = 'f'
  AND L.nid                       IN
```

```
    (SELECT L.nid
    FROM Host H,
      madrid_listing L
    WHERE L.host_id = H.host_id
    GROUP BY L.nid
    HAVING MAX(H.since) <= '01.06.17'
    )
  GROUP BY L.nid
  HAVING COUNT(*) > 0
  ) part,
  (SELECT L.nid                    AS nid ,
    COUNT(DISTINCT L.listing_id) AS cnt
  FROM Listing_calendar C ,
    madrid_listing L
  WHERE L.listing_id= C.listing_id
  GROUP BY L.nid
  HAVING COUNT(*) > 0
  ) total
WHERE part.nid           = total.nid
AND part.cnt / total.cnt > 0.5
AND part.nid = N.nid;
```

Result

	Neighbourhood
1.	Tetuán
2.	Atocha

Running time

The mean of the twenty measures of the running times is of 2530.1 ms.

3.1.11 Query 11

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

Description of logic

We first compute the number (N_part) of listing which were available once in 2018 per country. Then we compute the total number (N_tot) of listing per country. As in the previous query, we make the ratio between the 2 numbers and keep countries which have at least 20% of their listings available in 2018.

SQL statement

```
SELECT C.COUNTRY,
  100 * Round(part.cnt / total.cnt,3)
FROM
  Country C,
  (SELECT city.country_id AS country_id ,
    COUNT(DISTINCT L.id)  AS cnt
  FROM Listing_calendar C,
    Listing L,
```

```
    Neighbourhood N,
    City city
  WHERE extract(YEAR FROM C.cdate) = 2018
  AND L.id                         = C.listing_id
  AND L.nid = N.nid
  AND N.city_id                    = city.city_id
  AND C.available                  = 't'
  GROUP BY city.country_id
  HAVING COUNT(*) > 0
  ) part,
  (SELECT city.country_id AS country_id ,
    COUNT(DISTINCT L.id)  AS cnt
  FROM Listing_calendar C ,
    Listing L,
    Neighbourhood N,
    City city
  WHERE L.id     = C.listing_id
  AND L.nid = N.nid
  AND N.city_id = city.city_id
  GROUP BY city.country_id
  HAVING COUNT(*) > 0
  ) total
WHERE part.country_id     = total.country_id
AND total.country_id = C.country_id
AND part.cnt / total.cnt > 0.2 ;
```

Result

	Country	% of listing
1.	Germany	41.6
2.	Spain	77.5

Running time

The mean of the twenty measures of the running times is of 1921 ms.

3.1.12 Query 12

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

Description of logic

We first made a view of Barcelona Listing. Then we compute the the number of listing with strict cancellation policy with grace period per neighbourhoods. We also count the total number of listing for each neighborhood and finally keep each ratio (part/total) greater than 0.05 .

SQL statement

```
CREATE OR REPLACE VIEW barcelona_listing AS
SELECT L.id AS listing_id ,
```

```
  L.nid      AS nid,
  L.cpid     AS cpid
FROM Listing L,
  City C,
  Neighbourhood N
WHERE N.city_id = C.city_id
AND N.nid = L.nid
AND C.city      = 'Barcelona' ;

SELECT part.nid,
  100 * ROUND(part.cnt / total.cnt, 3)
FROM
  (SELECT L.nid                    AS nid ,
    COUNT(DISTINCT L.listing_id) AS cnt
  FROM Barcelona_listing L,
    CANCELLATION_POLICY CP
  WHERE L.cpid                  = CP.CPID
  AND CP.CANCELLATION_POLICY = 'strict_14_with_grace_period'
  GROUP BY L.nid
  HAVING COUNT(*) > 0
  ) part,
  (SELECT L.nid                    AS nid ,
    COUNT(DISTINCT L.listing_id) AS cnt
  FROM barcelona_listing L
  GROUP BY L.nid
  HAVING COUNT(*) > 0
  ) total
WHERE part.nid           = total.nid
AND part.cnt / total.cnt > 0.05 ;
```

Result

	Nid	% of listing
1.	6	41.6
2.	14	49
3.	23	31.8
4.	27	30.5
5.	50	20

Running time

The mean of the twenty measures of the running times is of 173 ms.

3.2 Query Analysis

We have selected queries 9, 10 and 11 to optimize with indexes simply because the other queries have a running time smaller than 500 ms (except Query 4) Among these, some are even executed in less than 100 ms. It was simply not relevant to try to speed up any other query than these three (except Query 4 but we chose Query 9), even though the 9th query also has a running time shorter than one second.

The initial and improved running times are computed as the mean of twenty measures, as previously done for the other queries in the report. As a reminder, to collect these measures, the queries have been run a few times without considering the result because of the warm-up period needed to have something relevant. Indeed, the first executions tend to be significantly longer.

When creating indexes, if no keyword is specified, Oracle SQL Developer uses by default B-tree indexes. As we have seen in class, these indexes are ideal for range-searches and are also good for equality searches. This turned out to be quite profitable in our case.

3.2.1 Query 9

Initial running time

The mean of the twenty measures of the running times is of 550 ms.

Optimized running time

The mean of the twenty measures of the improved running times is of 321.75 ms, which represents a speed up factor of approximately 1.7.

Explanation

Query 9 used the three following indexes:

```
0 CREATE INDEX listing_nid_idx
  ON Listing (nid);
```

```
0 CREATE INDEX listing_room_type_idx
  ON Listing (rtid)
```

```
0 CREATE INDEX review_listing_id_idx
  ON review (listing_id);
```

We can see in figure B.2 – the improved plan – that the indexes `listing_room_type_idx` and `listing_nid_idx` have been used for a fast full scan. The index `review_listing_id_idx` has been used for a range scan as well as for a fast full scan.

In our case, the index `review_listing_id_idx` is relevant when we want to group the reviews of listings which have the room type we want. Indeed, since we are only interested in the room type groups which have an average number of accommodates greater than three, a range scan using a B-tree index is useful.

For the other two indexes, namely `listing_room_type_idx` and `listing_nid_idx`, instead of a classical table access, they perform a fast full scan which is less costly, especially for the table `has_amenity`. The index `listing_nid_idx` has been used for the equality test `N.nid = L.nid` at line 21 of the query. The index `listing_room_type_idx` has been used when we check if the attribute `room_type` is in the subset of the ones having an average number of accommodates greater than three.

Initial plan

Please refer to figure B.1 in Appendix B.

Improved plan

Please refer to figure B.2 in Appendix B.

3.2.2 Query 10

Initial running time

The mean of the twenty measures of the running times is of 2530.1 ms.

Optimized running time

The mean of the twenty measures of the improved running times is of 803.15 ms, which represents a speed up factor of approximately 3.15.

Explanation

Query 10 used the two following indexes:

```
CREATE INDEX listing_nid_idx
ON Listing (nid);
```

```
CREATE INDEX Calendar_idx
ON Listing_Calendar (extract(YEAR FROM cdate), available);
```

The index `calendar_idx` has been used for a range scan for the first part of the query, when we compute the number of listings occupied once in 2019 for each neighbourhood whose all their hosts has joined airbnb before 01.06.2017. In this case it is indeed useful to index the calendar according to the date and its availability as a B-tree. It could indeed search the desired result binarily. It can be seen in the second part of the improved plan.

The index `listing_nid_idx` has been used when the view `madrid_listing` was needed to see if its neighbourhood ID was also part of the subset of itself where only hosts having joined airbnb before 01.0.6.2017 were considered.

The same index can also be seen in the second part of the plan and is used for a fast full scan as well when the `nid` of the listing had to be compared to the `nid` of the neighbourhood when the total number of listings per neighbourhood being in Madrid was computed.

As we know, this makes sense since B-tree indexes are also quite good for equality tests.

Initial plan

Please refer to figure B.3 and B.4 in Appendix B.

Improved plan

Please refer to figures B.5 and B.6 in Appendix B.

3.2.3 Query 11

Initial running time

The mean of the twenty measures of the running times is of 1921 ms.

Optimized running time

The mean of the twenty measures of the improved running times is of 1044 ms, which represents a speed up factor of approximately 1.8.

Explanation

Query 11 uses the exact same indexes as Query 10, namely these two :

```
CREATE INDEX listing_nid_idx
ON Listing (nid);
```

```
CREATE INDEX calendar_idx
ON Listing_Calendar (extract(YEAR FROM cdate), available);
```

The index `calendar_idx` has been used for a range scan for the first part of the query, when we compute the number of listings which were available once in 2018 per country. In this case it is indeed useful to index the calendar according to the date and its availability as a B-tree. It could indeed search the desired result binarily.

We then see that the index `listing_nid_idx` has been used twice for a fast full scan for the same reasons as in Query 10, namely to make an equality test between the `nid` of the listing and the one of the neighbourhood.

Initial plan

Please refer to figure B.7 in Appendix B.

Improved plan

Please refer to figure B.8 in Appendix B.

3.3 General Comments

3.3.1 Work allocation between team members

Eric wrote the additional SQL requests, with close collaboration with all team members. Charline created the indexes to optimize the queries. Robin finished the interface.

Appendix A

Attributes

A.1 Listings

1. **Id** The unique listing identifier.
2. **listing_url** The URL of the listing.
3. **name** The name of the listing.
4. **summary** A small description of the listing.
5. **space** A small description of the space of the listing.
6. **description** A large description of the listing.
7. **neighborhood_overview** Description of the neighbourhood of the listing.
8. **notes** An extra note about the listing.
9. **transit** Description of the transportation to the listing.
10. **access** Specification of the accessibilities of household stuff, such as kitchen facilities.
11. **interaction** Description of whom/how to interact regarding the listing.
12. **house_rules** House rule specifications.
13. **picture_url** The URL to the picture of the listing.
14. **host_id** The unique host identifier.
15. **host_url** The URL of the host.

16. **host_name** The name of the host.
17. **host_since** The date that the host has started working with Airbnb.
18. **host_about** A small description of the host.
19. **host_response_time** The amount of time within which the host responses.
20. **host_response_rate** The rate at which the host replies the messages.
21. **host_thumbnail_url** The URL to a thumbnail profile photo of the host.
22. **host_picture_url** The URL to a profile photo of the host.
23. **host_neighbourhood** The neighbourhood the host lives in.
24. **host_verifications** The way with which the host can be verified.
25. **neighbourhood** The neighbourhood where the listing is in.
26. **city** The city where the listing is in.
27. **country_code** The code of the country where the listing is in.
28. **country** The country where the listing is in.
29. **latitude** The latitude of the listing.
30. **longitude** The longitude of the listing.
31. **property_type** The type of the property.
32. **room_type** The type of the room.
33. **accommodates** The number of people that the listing can accommodate.
34. **bathrooms** The number of bathrooms that the listing has.
35. **bedrooms** The number of bedrooms that the listing has.
36. **beds** The number of beds that the listing has.
37. **bed_type** The type of the beds.
38. **amenities** The set of amenities that listing features.
39. **square_feet** The area of the listings in square feet.
40. **price** The daily price of the listing. It is the price for the day when the data is collected. For the price for a particular date, please see the *_calendar.csv files.
41. **weekly_price** The weekly price of the listing.

42. **monthly_price** The monthly price of the listing.

43. **security_deposit** The amount of money for security deposit.

44. **cleaning_fee** The fee for cleaning.

45. **guests_included** The number of guests that the daily price covers.

46. **extra_people** The additional price to be paid for every extra guest in addition to the number of guests specified by the guests_included attribute.

47. **minimum_nights** The minimum number of nights to rent.

48. **maximum_nights** The maximum number of nights to rent.

49. **review_scores_rating** The rating score of the listing.

50. **review_scores_accuracy** The accuracy score of the listing.

51. **review_scores_cleanliness** The cleanliness score of the listing.

52. **review_scores_checkin** The checkin score of the listing (to quantify how easy the checkin is).

53. **review_scores_communication** The communication score of the host.

54. **review_scores_location** The location score of the listing.

55. **review_scores_value** The score on the value that the listing provides for the price.

56. **is_business_travel_ready** Whether the listing can be used for business travels.

57. **cancellation_policy** The cancellation policy of the listing.

58. **require_guest_profile_picture** Whether the listing requires a guest profile picture.

59. **require_guest_phone_verification** Whether the listing requires guest phone verification.

A.2 Reviews

R1. listing_id The identifier of the listing that is reviewed.

R2. id The unique review identified.

R3. date The date that the review has been written.

R4. reviewer_id The uniquer reviewer identified

R5. reviewer_name The name of the reviewer

R6. comments The review.

A.3 Calendar

C1. listing_id The identifier of the listing whose availability and price information is given.

C2. date The date on which the listing is available or not.

C3. available Whether the listing is available or not.

C4. price The price of the listing for the particular date.

Appendix B

Execution plans

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	27790
VIEW		1	27790
Filter Predicates			
from$_subquery$_009.rowlimit_$$_rownumber<=1			
WINDOW (SORT PUSHED RANK)		3	27790
Filter Predicates			
ROW_NUMBER() OVER (ORDER BY INTERNAL_FUNCTION(CNT) DESC)<=1			
MERGE JOIN		3	27789
TABLE ACCESS (BY INDEX ROWID)	CITY	3	2
INDEX (FULL SCAN)	SYS_C0043172	3	1
SORT (JOIN)		3	27787
Access Predicates			
C.CITY_ID=T.CITY_ID			
Filter Predicates			
C.CITY_ID=T.CITY_ID			
VIEW		3	27786
HASH (GROUP BY)		3	27786
HASH JOIN		630394	27771
Access Predicates			
R.LISTING_ID=L.ID			
HASH JOIN		20855	11478
Access Predicates			
N.NID=L.NID			
TABLE ACCESS (FULL)	NEIGHBOURHOOD	224	3
HASH JOIN (RIGHT SEMI)		20855	11475
Access Predicates			
L.RTID=RTID			
VIEW	VW_NSO_1	1	5972
FILTER			
Filter Predicates			
SUM(CNT)/COUNT(CNT)>=3			
HASH (GROUP BY)		1	5972
HASH JOIN		41932	5970
Access Predicates			
L.ID=LISTING_ID			
TABLE ACCESS (FULL)	LISTING	42094	5503
VIEW		41932	466
HASH (GROUP BY)		41932	466
TABLE ACCESS (FULL)	HAS_AMENITY	830882	446
TABLE ACCESS (FULL)	LISTING	42094	5503
TABLE ACCESS (FULL)	REVIEW	1272377	16289

Figure B.1: The initial plan of the 9th query.

Figure B.2: The improved plan of the 9th query.

Figure B.3: The initial plan of the 10th query - Part 1.

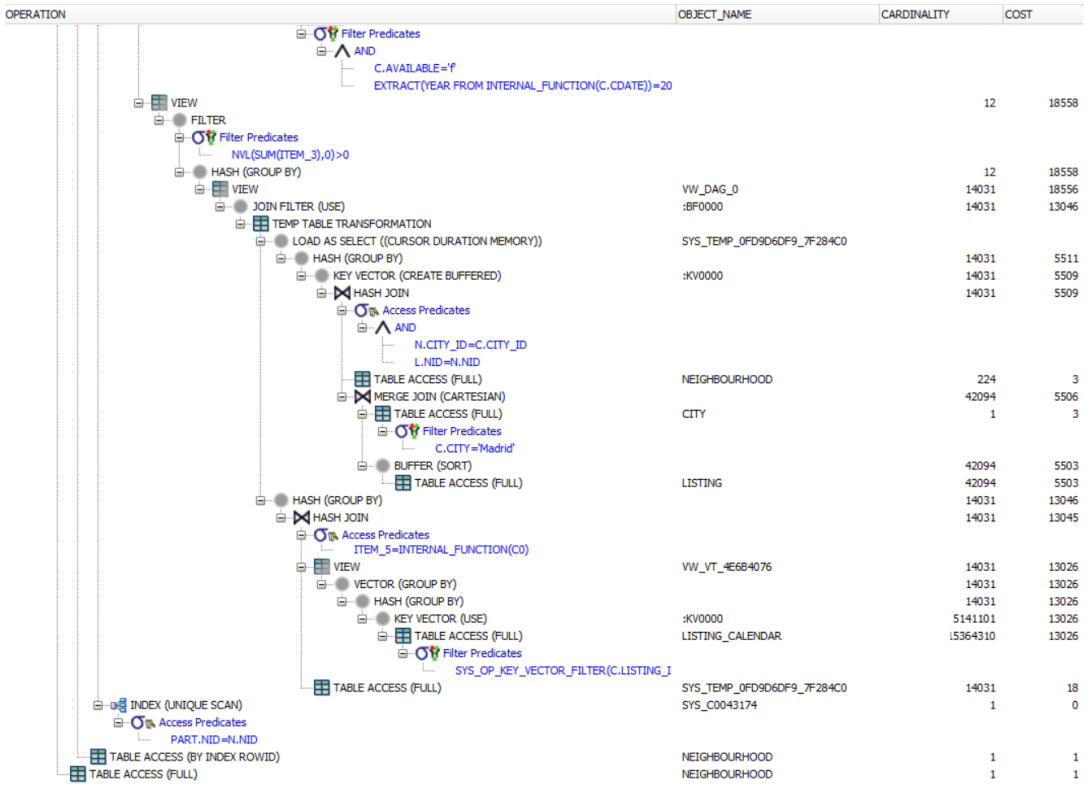

Figure B.4: The initial plan of the 10th query - Part 2.

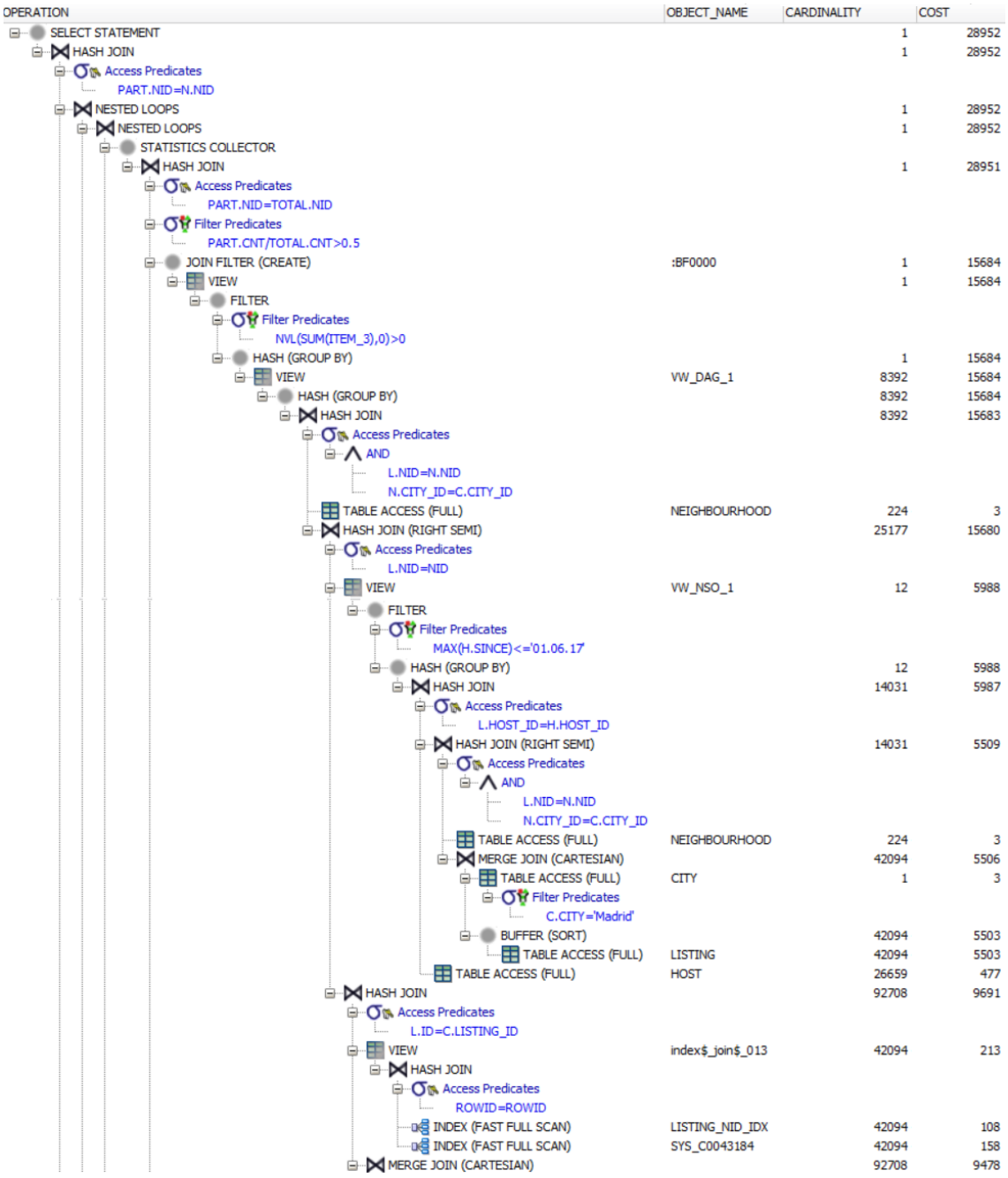

Figure B.5: The improved plan of the 10th query - Part 1.

Figure B.6: The improved plan of the 10th query - Part 2.

Figure B.7: The initial plan of the 11th query.

Figure B.8: The improved plan of the 11th query.