DIAS: Data-Intensive Applications and Systems Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Building BC, Station 14
CH-1015 Lausanne
URL: http://dias.epfl.ch/

Databases Project – Spring 2019

Team No: 32

Names: Sophie Ammann, Samuel Chassot and Daniel Filipe Nunes Silva

Contents

Deliverable 1

Assumptions

We use MySQL syntax for this project.

Listings all have a bed_type, property_type, room_type, cancellation_policy, a host and a neighborhood.

A review must be written by a reviewer about a given listing.

A calendar must have a day and a listing, otherwise it should not exist.

A host must be in a neighborhood, which must be itself in a city, which must be itself in a country.

DIAS: Data-Intensive Applications and Systems Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Building BC, Station 14
CH-1015 Lausanne
URL: http://dias.epfl.ch/

Entity Relationship Schema

Schema

See Annex*/deliv_1/deliv_1_ER_Schema.png* (first version), Annex*/deliv_2/deliv_2_ER_Schema.png* (latest version)

Description

Most of the attributes related to a listing are grouped in one single table *Listing*.

The *Amenity*, Bed*_type*, *Property_type*, *Room_Type*, *Cancellation_policy*, *Country*, *City*, *Day* and *Host_verification* have been normalized to reduce redundancy.

Foreign key relationships are hardly connected to preserve integrity even if it implies to drop a lot of data after deletions (*ON DELETE CASCADE*). For example, if we delete a *Country*, we will delete every *City* in this *Country* and therefore delete every *Neighborhood* in the *Cities* and so forth.

Relational Schema

ER schema to Relational schema

The translation of our ER schema to our Relation schema was more or less straightforward. Nevertheless, the type were not always obvious. We also decided to add a unique *id* in our tables.

DDL

See Annex*/deliv_2/deliv_2_create.sql*

General Comments

For this first work, we thought it was important to work the three together to understand the database correctly. We designed the basis of the ER model, and modified it until the three of us were satisfied.

DIAS: Data-Intensive Applications and Systems Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Building BC, Station 14
CH-1015 Lausanne
URL: http://dias.epfl.ch/

Deliverable 2

Assumptions

We updated the ER model with respect to the feedback. See Annex*/deliv_2/deliv_2_ER_Schema.jpg*

We assume that all listings' city corresponds to the filename in which they are.

We put new ids for all listings and hosts to not have duplicates between cities.

We remove reviewers if they have the same id but not same name to keep only one.

We remove comments in reviews that consist of only one character (often a quote or a comma).

After parsing and cleaning all the data, we end up with simple csv files, whose name corresponds to our tables' names and columns are the same as specified in our schema.

Data Loading

Query Implementation

Query 1:

The query finds the average price for a listing with a specified number of bedrooms. We use 8 bedrooms for the example. Since the table Listing has an attribute for the number of bedrooms and one for the price, the query is direct.

Query 2:

The query finds the average cleaning review score for the listings with TV. We suppose that these listings include the amenities holdng. The implementation requires the Listing and Amenity tables, and the relation that maps both of them (Listing_amenity_map). We chose to use the price that is listing instead of computing the average from Calendar because of the time it takes. It gives satisfying results in a lot improved time.

Query 3:

The query finds the hosts that have an available listing between two dates (03.2019 and 09.2019). We suppose that it suffices that the listing is available just one day in this interval. We need the Host table for the name, the Listing, the Day and the Calendar tables for the implementation.

Query 4:

The query finds how many listings exist that are posted by two different hosts but the hosts have the same name. For the implementation, we used D.day to show the dates in MySQL. We had to change this in the interface code with python and write SELECT DAY(d.day), MONTH(d.day), YEAR(d.day) to show the dates correctly.

Query 5:

The query finds the dates that a specified host (we use 'Viajes Eco') has available accommodations for rent. The Day, Calendar (listing-calendar relation), Listing (shows which listings Viajes Eco owns) and Host tables are necessary for the implementation.

Query 6:

The query finds all the pairs (host_ids, host_names) that only have one listing online. We only need Listing and Host table to implement this query. The COUNT syntax is necessary to implement the constraint of "exactly one".

Query 7:

The query computes the difference of price (average) between listings with or without Wifi. We suppose that the listing with Wifi have either 'Wifi' or 'Pocket wifi' in their amenities. For the implementation we need the many-to-many Listing_amenity_map to link the wifi information (Amenity table) and the listings (Listing table).

For the implementation, it is very expensive if we use the price in calendar and compute its average, so we use listing.price because it is satisfying approximation and time goes from 442sec to 3sec

Query 8:

The query computes the difference of price in a room with 8 beds in Berlin compared to Madrid. We need the Listing, Neighbourhood and City tables to implement the query, since the Neighbourhood table links the Listing and the City by our normalization.

Query 9:

The query finds the top-10 host (host_ids, host_names) in terms of number of listings per host in Spain.

Query 10:

The query finds the top-10 listings (review_score_rating) in terms of review_score_rating apartments in Barcelona. For the implementation, we used the TOP syntax which seems to not work.

SQL statements

Annex/*deliv_2/deliv_2_queries.sql*

Indexes:

For the query fetch times and possible indexes :

Annex/*deliv_2/deliv_2_indexes.sql*

Results:

See Annex*/deliv_2/deliv_2_query_results.sql*

Interface

Design logic Description

We used a local MySQL server, Python 3.7 with Tkinter and mysql-connector to link the two.

See README.md for configuration and how to run it.

When launching the application, the database is automatically created and populated (It takes ~20min.).

You can see the status of the connection and of the database on the top of the application. If any problem occurs, you can the delete DB and connect DB buttons to repeat the operations again. If the problem persists, close the application, go to the MySQL shell and execute *DROP DATABASE Airbnb;*.

The application is separated in Tabs, for now, you can search listings, hosts and neighborhoods using the available fields, each of them is initially set to the less restricted input or execute the predefined queries.

After searching or executing, a new window shows up to display the executed statement with the parameterized values as well as the results of this query.

Some operations may take some time (executing the queries and populating the tables) and these are blocking so we have to wait until it finishes to continue.

Screenshots

interface*/deliv_2/*.png*

General Comments

We worked the three together to make the suggested changes after Milestone 1. Then, Sophie worked on the queries as well as on redesigning the new ER schema, Samuel worked on parsing and cleaning the data and Daniel worked on the interface.

For the next Milestone, we will work harder on the predefined queries before going further.

DIAS: Data-Intensive Applications and Systems Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Building BC, Station 14
CH-1015 Lausanne
URL: http://dias.epfl.ch/

Deliverable 3

Assumptions

We updated the ER model with respect to the feedback. See Annex*/deliv_2/deliv_2_ER_Schema.jpg*

We assume that all listings' city corresponds to the filename in which they are.

We put new ids for all listings and hosts to not have duplicates between cities.

We remove reviewers if they have the same id but not same name to keep only one.

We remove comments in reviews that consist of only one character (often a quote or a comma).

After parsing and cleaning all the data, we end up with simple csv files, whose name corresponds to our tables' names and columns are the same as specified in our schema.

Query Implementation

See Annex*/deliv_3/deliv_3_querie.sql*

Query 1:
Description of logic:

Join all tables Host, Listing, Neighbourhood and City, filter by host_id, keeping only host_id that have at least one listing with square_feet NOT NULL. Group this by city_name and order by city_name.

Query 2:
Description of logic:

We create a temporary table (WITH statement) containing listing with review_scores_rating NOT NULL and we add the row number to each row. Then we group by this by neighbourhood and compute the half of the sum of all row numbers for each neighbourhood in another temporary table. Then we select from the first one, for each neighbourhood, the row with the number that is in the second one. We then keep only the 5 with the biggest scores.

Query 3:
Description of logic:

Select the host_id, host_name and the number of listings per hosts ordered by number of listings descending and keep the first.

Query 4:

Description of logic:

We join Listing, Calendar, Neighbourhood, City, Day, Cancellation_policy, Host, Host_verification, Host_verification_map and filter that by the given conditions. We then select listing_id and the average of the price per day (contained in Calendar table) group by listing_id, order by this average descending and keep the five first.

Query 5:

Description of logic:

We create a temporary table that contains each listing with the number of amenities it contains from the given list. Then we select from this table, for each number of accommodates, the 5 listings with the biggest number of amenities.

Query 6:

Description of logic:

We compute a subtable containing listing_id, host_id and the number of reviews per listing. Then we select the host_id, the listing_id concatenanted and the number of reviews concatenated from the previous table where the listing_id is in the three first in when sorted by number of reviews descending.

Query 7:
Description of logic:

We create a subtable containing the number of listings that meet the conditions per amenity and neighbourhood. From this table, we select the neihbourhood and the three amenities that have the biggest number of listings per neighbourhood.

Query 8:
Description of logic:

We create a temporary table containing hosts with their number of possible verifications. Then we select the average review_scores_communication for the first host by sorting the first table descending then for the first one sorting ascending and we compute the difference.

Query 9:
Description of logic:

We select city from City and a subquery that selects the number of reviews of reviews from another subquery that selects the room type with its average number of accommodates and number of reviews, where the average number of accommodates is greater or equal to 3. We keep the one with the highest number of reviews.

Query 10:
Description of logic:

We select listings with their number of days occupied and that satisfy the conditions and with number of occupied days not zero. Then from that we select the neighborhood with the number of listings from previous query and then select the neighbourhood where ratio between this number and the total number of listings is greater or equal 0.5.

Query 11:
Description of logic:

We select the countries with the number of listings available for each days between the two given dates. Then we select countries where the ratio between the number of available listings and the total number of listings is at least 0.2 at least one day.

Query 12:
Description of logic:

We select the neighbourhood with the number of listings with strict_14_with_grace_period policy. Then we select the total number of listings per neighbourhood. From these two tables we select neighbourhood where the ratio between the number with strict_14_with_grace_period policy and the total number is a least 0.05.

Results:

See Annex*/deliv_3/deliv_3_query_results.sql*

Query Analysis

Selected Queries (and why)

For the choice strategy, we first thought that the queries with the longest time execution had the best optimization potential. We ran a few times each queries and found that query 4), 10) and 11) had the worst fetch times (approximatively 4.5sec, 75sec and 32sec, respectively).

Adding indexes improved the fetch time of query 4) and 10), but query 11) remained just as long. Hence we went through the other queries and their query plans and chose query 1). The improvement by adding one index was significant, even though the query had a low fetch time without adding indexes (1.37sec).

Moreover, it appears that MySQL generates automatically indexes in the table creations. Therefore, columns that are declared as a PRIMARY KEY, UNIQUE or FOREIGN KEY are already indexed by B+Trees. That explains why most of our queries are already very fast. See Annex*/generated_indexes_MySQL.txt*. This has as consequence that lots of queries have bottle neck on big tables like Listing or Host but cannot really be improved because indexes are already in place.

To compute the running times, we ran each queries a few times to find the average times. We noticed that the running times also depend on what computer we're using. To see the running times and the added indexes for queries 4,10 and 11, see Annex*/deliv_3/deliv_3_indexes.sql*

For the plans linked to the three queries, see Annex*/deliv_3/deliv_3_initial_plan_*.png* and Annex*/deliv_3/deliv_3_improved_plan_*.png*

Query 1:

In the initial plan, the first join is very costly. The full index scan on host table is as bad as a table scan, with the 23k rows.
We put an index on Listing on its square_feet column. Therefore the index is used in the first join in the improved query plan. We can directly retrieve in the subquery the listings that do not have NULL for their square feet. The order of the joins differ since we can start by the inner query. The cost is lowered, less rows are retrieved with the new plan, and we do not need to run a full index scan any more.

Query 4:

In the initial plan, we have joins which use full table scans. In terms of performance, this is the worst we can do. The three concerned joins use the City, Cancellation_policy and Host_verification tables.
We put an index on the table Host_verification on its description column. Since it is a text, it will be faster to join on the condition [host_verification_description = 'government_id']. We chose to not put indexes on the

other tables, because the performance does not change. The number of retrieved rows and the size of the table is to small to have an impact.
With the index added, the running time is divided by more than 2. The index is used to retrieve the row satisfying the host_verification_description.

Query 10:

This last query plan uses in the listing_occupied_days materialized table a full table scan of the Host table. The retrieval of 27k rows is required. To help improve, we inserted an index on Host table on the host_since column. Therefore only an index range scan is used for the same join, reducing to 11k rows.

Interface

Design logic Description

We added new features to the interface.

Now, when searching a listing, we can choose a city but no review score rating anymore.

The user is able to insert new listings in the database by inserting its name, summary, square feet, price, business travel ready, property type, room type, bed type and cancellation policy. Sql queries are runned in background to fill the option menus. Hosts can be inserted or found by their city, neighborhood and name. When checking the host, this one will be created if (s)he does not exist yet, the same applies for neighborhoods and the inserted ids will appear next to the check buttons. The insert button is disabled until the host and neighborhood are checked.

The new queries have been added to the *Predefined Queries* tab. The indexes can be activated and deactivated manually under the settings tab.

There is a *Delete* tab where the user can delete any table entry by entering its id.

The results window has been improved. We can now display and scroll larger queries.

Screenshots

interface/*deliv_3/*.png*

Evolution since first deliverable

For the first deliverable, we conceived an ER model which was not appropriated for the database. We then improved it by normalizing most of the tables to avoid redundancy. For example, we had a table House_properties(square_feet, room_type, amenities, …) which was a one-to-one relation with the listing. They were many strings and after inspection we realized that properties such as amenities, room_type, etc. deserved their own table : they only had a few possible values in the dataset. We created tables for each of them (Amenity, Bed_type, Cancellation_policy, …) and linked them to the listings. We have the foreign keys in the Listing table that are integers, hence it reduces considerably the size of a listing's tuple and simplifies the retrieval of those properties.

Concerning the queries, we fixed the deliverable 2 queries and added some indexes when needed.

General Comments

For this last deliverable, we split the work between queries and interface. Daniel improved the interface. Samuel and Sophie wrote the queries with their optimization. The report was written and updated by the three of us.