Patrick Ocampo
Student Number: 1003256460 *UTORID: ocampop4*

CSCC43 | Summer 2019 | MyBnB - Database Design Project

Introduction - Description and Assumptions

MyBnB in its very essence is an application that allows its users to rent housing from other users that have available accommodations that suit their needs. Similar to AirBnB, MyBnB seeks to make the process of connecting hosts with its renters simple while also incorporating a quick/efficient system that will deliver satisfaction to the needs of both parties. Its current implementation runs on a console interface, but a GUI can be established in future iterations.

While planning and implementing this system, there were a few conceptual issues that needed to be addressed:

- Efficiency: Users need to have their queries/requests answered/fulfilled in a quick and timely manner. They should not have to wait for a long time due to inefficient algorithms in the back end of the code. In order to ensure this is upheld in the implementation of the code, I have eliminated the use of nesting loops to keep things in polynomial time and I have done my best to ensure that most of the query processing is done with SQL statements and not Java.
- Redundancy: Data shouldn't be repeated multiple times in the database system. To ensure we don't have many duplicate entries of data in the database, I have made an attempt to put things in some normal form that doesn't ruin the integrity of the data.

These only encompass the general implementation issues associated with this project. In addition to this, reasonable assumptions were also necessary in order to preserve the integrity of the data within the system in addition to solving a few other issues that came about in the process of the implementation:

1. A user can only be a host or a renter, but not both → When a user creates an account, they will either be registered as a host or a renter, but they cannot be both. This design choice was made in order to simplify the inner workings of the system and also address concerns such as a host renting a listing in the same time period that they are hosting their own listing.
2. Listings can have the same latitude/longitude → There are many types of housing that an individual can rent from/host for. If we work under the example of two hosts creating a listing for a space in an apartment/condominium (and they were both either below/above each other), then there is no doubt that they will have the same latitude/longitude for their listing.
3. A reviewer can write multiple reviews for a single listing/host → A renter may decide to rent one or more times from a specific host at a specific listing. Each experience may be different from the previous and so it is only natural that they are allowed to write more than one review for the host/listing.

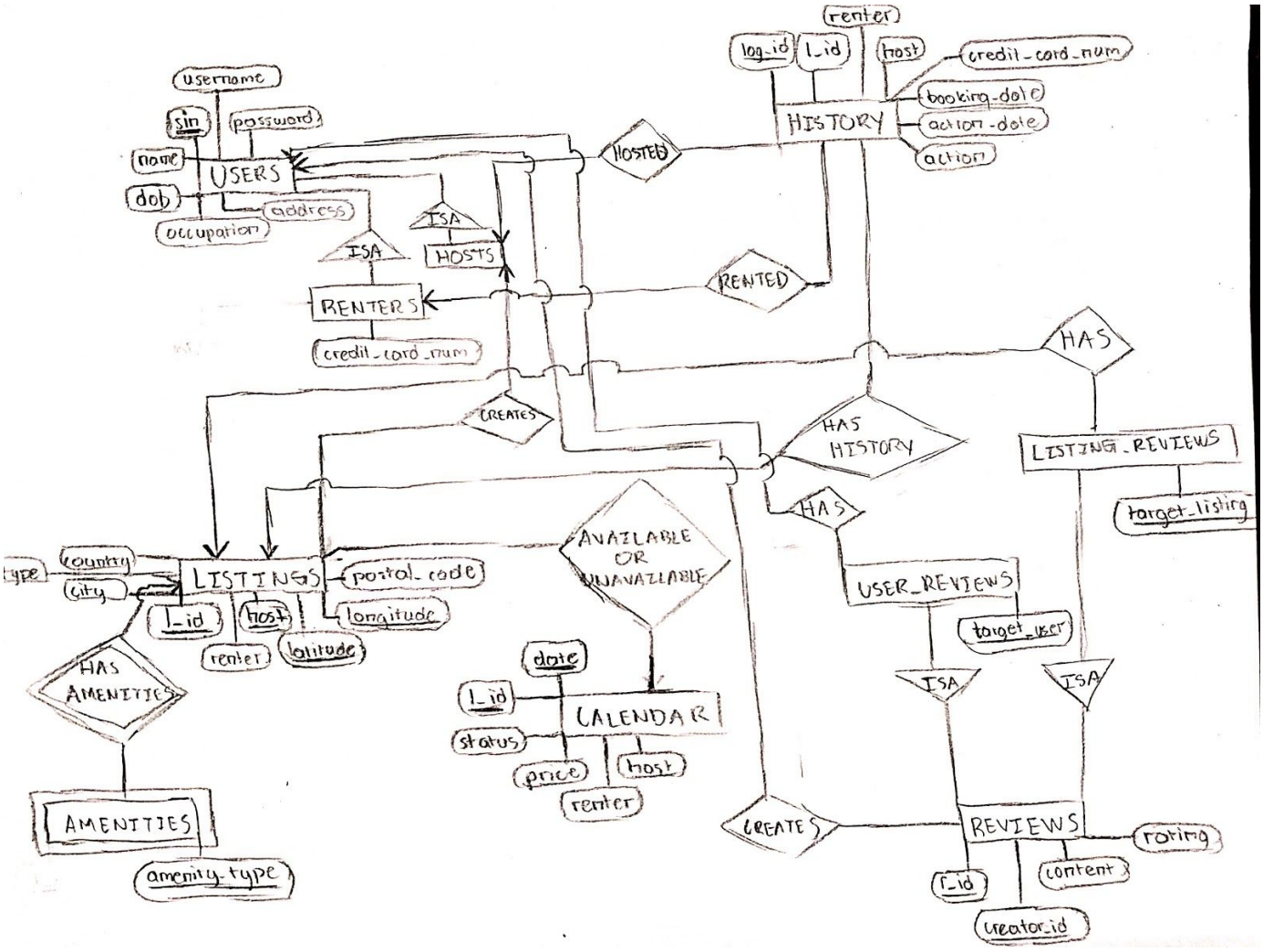
Patrick Ocampo
Student Number: 1003256460 *UTORID: ocampop4*

4. A host cannot change the availability of a listing once they have made it unavailable → The action of a host making a listing unavailable should be final in order to avoid cases of them constantly switching between it being available and unavailable. This is to mitigate the confusion/uncertainty a renter has when they see the availability for a given listing constantly changing.
5. A host is allowed to remove a listing completely even if renters have bookings set for it → There will be times when a host encounters unforeseen circumstances that require them to cancel any sort of plans they have for renting their spaces out. To simplify the process, renters will be allowed to cancel a listing instantaneously even if renters have already created bookings.
6. The distance between two latitude/longitude points on a map will refer to the radius → To check for listings within a given vicinity, we need to assume that "distance" refers to the radius circle around a given latitude/longitude point. With that being said, we will need to employ a specific method which employs the Haversine formula to calculate this distance.
7. Once a user deletes their account, any record associated with their unique ID will be deleted as well EXCEPT for the reviews they leave for a listing or user → This is to ensure that the data we have in our database makes sense and we don't have references to entities that no longer exist. This includes removing them from our History Table. However, we would still like to keep track of the comments they made for listings/users even when they're already gone.
8. Users cannot create reviews/comments for listings/users they have no association with → I am aware that this was already a requirement from the handout, but I have made sure in my code that if a user wanted to write a review, they will be given a list of users/listings that they had an affiliation with. So if a user entered the command to write a review/listing, they will first see a list of users they have an affiliation with. They will then enter a name that they see and proceed to write a review.

(PLEASE SEE NEXT PAGE FOR THE ER DIAGRAM)

Patrick Ocampo
Student Number: 1003256460 *UTORID: ocampop4*

ER Diagram



Schema

Users(sin, full_name, date_of_birth, occupation, username, password, address)
Renters(Users.sin, full_name, date_of_birth, occupation, username, password, address, credit_card_num)
Hosts(Users.sin, full_name, date_of_birth, occupation, username, password, address)
Listings(listing_id, Hosts.sin, latitude, longitude, Renters.sin, postal_code, city, country, listing_type)
Amenities(amenity_type, Listings.listing_id)
Calendar(calendar_date, Listings.listing_id, status, price, Renters.sin, Hosts.sin)
History(log_id, listing_id, Renters.sin, Hosts.sin, booking_date, action_date, action, Renters.credit_card_number)
Reviews(review_id, Users.sin[creator_id], content, rating)

Patrick Ocampo
Student Number: 1003256460 *UTORID: ocampop4*

UserReviews(review_id, Users.sin[creator_id], Users.sin[target_user_id], content, rating)
ListingReviews(review_id, Users.sin[creator_id], Listings.sin[target_user_id], content, rating)
Cancellations(Users.sin[canceller_sin], count)

creates_listing(Hosts.sin, Listings.listing_id, Listings.latitude, Listings.longitude)
has_amenities(Listings.listing_id, Amenities.amenity_type)
has_dates_available_or_unavailable(Calendar.calendar_date, Listings.listing_id)
hosted(Hosts.sin, History.log_id)
rented(Renters.sin, History.log_id)
has_history(Listings.listing_id, History.log_id)
creates_review(Users.sin, UserReviews.review_id)
user_has_review(Users.sin, UserReviews.review_id)
listing_has_review(Users.sin, ListingReviews.review_id)
cancels(Users.sin)

DDL Statements

I have defined all of my DDL statements in the source code under a function. Below are the specific statements that have been written in the code and a few brief explanations of my intentions:

- Begin by dropping all the tables in the database and include CASCADE to ensure everything is deleted accordingly

```
DROP TABLE IF EXISTS Cancellations CASCADE
DROP TABLE IF EXISTS UserReviews CASCADE
DROP TABLE IF EXISTS ListingReviews CASCADE
DROP TABLE IF EXISTS Reviews CASCADE
DROP TABLE IF EXISTS History CASCADE
DROP TABLE IF EXISTS Calendar CASCADE
DROP TABLE IF EXISTS Amenities CASCADE
DROP TABLE IF EXISTS Listings CASCADE
DROP TABLE IF EXISTS Hosts CASCADE
DROP TABLE IF EXISTS Renters CASCADE
DROP TABLE IF EXISTS Users CASCADE
```

- Create the Users table and ensure the username is unique. Also, we make the user's SIN the primary key

```
CREATE TABLE Users
    (full_name VARCHAR(100)
    date_of_birth DATE NOT NULL
    occupation VARCHAR(200)
```

```
    sin INT NOT null
    username VARCHAR(50) UNIQUE
    password VARCHAR(50)
    address VARCHAR(200)
    PRIMARY KEY ( sin )
    )
```

- Create a trigger to ensure that each user inserted into the database is of the appropriate age and throw an error if the user is not above said age

```
CREATE TRIGGER date_check BEFORE INSERT ON Users
    FOR EACH ROW
    BEGIN
    IF DATEDIFF(CURDATE(), NEW.date_of_birth) < 18 * 365 THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'You must be older than 18 to use
this application!';
    END IF;
    END
st.execute(sql);
```

- Create a table for Renters, which inherits from the Users table and has an extra field for their credit card number

```
CREATE TABLE IF NOT EXISTS Renters
    (credit_card_num VARCHAR(50)
    sin INT NOT null
    PRIMARY KEY ( sin )
    FOREIGN KEY ( sin ) REFERENCES Users( sin ) ON DELETE CASCADE )
```

- Create a table for Hosts, which inherits from the Users table

```
CREATE TABLE IF NOT EXISTS Hosts
    (sin INT NOT null
    PRIMARY KEY ( sin )
    FOREIGN KEY ( sin ) REFERENCES Users( sin ) ON DELETE CASCADE )
```

- Create a table for the Listings that a Host creates. Set the primary key to be an auto incremented ID, the host's sin, the latitude, and longitude. This table will have a reference to the Hosts table

```
CREATE TABLE IF NOT EXISTS Listings
    (sin_host INT NOT NULL
    listing_type VARCHAR(200)
    listing_id INT NOT NULL AUTO_INCREMENT
    latitude DECIMAL(9,6) NOT NULL
    longitude DECIMAL(9,6) NOT NULL
```

```
    postal_code VARCHAR(7)
    city VARCHAR(50)
    country VARCHAR(50)
    PRIMARY KEY ( listing_id, sin_host, latitude, longitude )
    FOREIGN KEY ( sin_host ) REFERENCES Hosts( sin ) ON DELETE CASCADE
    CHECK (latitude >= -90 AND latitude <= 90)
    CHECK (longitude >= -180 AND longitude <= 180)
    )
```

- Create an amenities table that will store all the different types of amenities for a given listing and have it reference the Listing table. The primary key will be the amenity type and the listing id.

```
CREATE TABLE IF NOT EXISTS Amenities
    (amenity_type VARCHAR(100)
    listing_id INT NOT NULL
    PRIMARY KEY ( amenity_type, listing_id )
    FOREIGN KEY ( listing_id ) REFERENCES Listings( listing_id ) ON DELETE CASCADE)
```

- Create a table for the Calendar dates that will store the dates that a given listing is available/unavailable (depending on what the host defines it as). The primary key will be the listing id and the calendar date. Of course, this is to ensure that a listing does not have 2 separate availability dates

```
CREATE TABLE IF NOT EXISTS Calendar
    (calendar_date DATE NOT NULL
    price REAL NOT NULL
    listing_id INT NOT NULL
    status BOOLEAN
    sin_renter INT DEFAULT NULL
    sin_host INT
    PRIMARY KEY ( listing_id, calendar_date )
    FOREIGN KEY ( listing_id ) REFERENCES Listings( listing_id ) ON DELETE CASCADE)
```

- Create a table that keeps track of all the bookings/cancellations that take place in the system. It will make references to the renters table, hosts table, and listings table

```
CREATE TABLE IF NOT EXISTS History
    (log_id INT NOT NULL AUTO_INCREMENT
    listing_id INT NOT NULL
    sin_renter INT NOT NULL
    sin_host INT NOT NULL
    booking_date DATE
    action_date DATE
    action VARCHAR(15)
```

```
    payment_info VARCHAR(50)
    PRIMARY KEY ( log_id )
    FOREIGN KEY ( sin_renter ) REFERENCES Renters( sin ) ON DELETE CASCADE
    FOREIGN KEY ( sin_host ) REFERENCES  Hosts ( sin ) ON DELETE CASCADE
    FOREIGN KEY ( listing_id ) REFERENCES Listings( listing_id ) ON DELETE CASCADE)
```

- Create a table that keeps track of all the reviews related to users

```
CREATE TABLE IF NOT EXISTS UserReviews
    (creator_id INT NOT NULL
    target_user_id INT NOT NULL
    content VARCHAR(1000)
    rating INT NOT NULL
    review_id INT NOT NULL AUTO_INCREMENT
    PRIMARY KEY ( review_id )
    FOREIGN KEY ( creator_id ) REFERENCES Users( sin )
    FOREIGN KEY ( target_user_id ) REFERENCES Users( sin ) ON DELETE CASCADE
    CHECK (rating > 0 AND rating < 6)
    CHECK (creator_id <> target_user_id)
    )
```

- Create a table that keeps track of all the reviews related to a listing

```
CREATE TABLE IF NOT EXISTS ListingReviews
    (creator_id INT NOT NULL
    target_listing_id INT
    content VARCHAR(1000)
    rating INT NOT NULL
    review_id INT NOT NULL AUTO_INCREMENT
    PRIMARY KEY ( review_id )
    FOREIGN KEY ( creator_id ) REFERENCES Users( sin )
    FOREIGN KEY ( target_listing_id ) REFERENCES Listings( listing_id ) ON DELETE
CASCADE
    CHECK (rating > 0 AND rating < 6)
    CHECK (creator_id <> target_listing_id)
    )
```

- Create a table that keeps track and count of all the cancellations a renter/host makes

```
CREATE TABLE IF NOT EXISTS Cancellations
    (canceller_sin INT NOT NULL
    count INT NOT NULL DEFAULT 0
    PRIMARY KEY ( canceller_sin )
    FOREIGN KEY ( canceller_sin ) REFERENCES Users( sin ) ON DELETE CASCADE)
```

Patrick Ocampo
Student Number: 1003256460 *UTORID: ocampop4*

Limitations and Next Steps/Future Work

Currently, there are several limitations/poor design choices that come with the current implementation of this system.

1. Availability dates are stored as individual entries in the database → Initially, I had planned to store start/end dates as ranges for availability. As I thought about this, I realized that if a user wanted to book a subrange between the 2 dates, I'd have to divide the ranges into 3 subranges. I would continuously do that with each request until I reach a point where there are single entries for each date. With my current implementation, I assume the worse case from the start - more space would be occupied in the system. As an improvement, I could work with storing date ranges in the database and consider consolidating the singular entries listed as available.
2. The current user interface's implementation is on the console → Given that this application runs off the console, the user has to go through a few more extra steps to complete their tasks. For example: if a user wanted to search for listings with multiple filters, they'd have to type in each filter individually. To improve on this in future iterations, I could create a GUI to support the functionality of this system and have an overall improvement on the user experience.
3. There is a lot of duplicate code → Going through with this project's implementation, I found myself duplicating lines of code and not making it modular for ease of change in the future. To improve on this, I'd separate a lot of the logic/functionalities into separate classes/methods and ensure I'm following SOLID design.
4. Not all inputs are validated properly → If a user enters invalid inputs in some of the fields, then there is a chance that the program will crash. For future iterations, I should be handling inputs more properly, i.e., have proper error handling for invalid inputs.