NOVEMBER 8, 2021

HOTEL BOOKING SYSTEM

DATABASES CSC7082

NIALL HODGEN
[40040160]

Databases CSC7082
Niall Hodgen [40040160]

Overview

Students were tasked with designing and implementing a database system for a commercial accommodation booking system (Hotels.com). The design and implementation were undertaken individually. However, a substantial amount of group discussion and planning was key to the design decisions which underpinned the final design of the database.

One of the key aims of the task was to try and best mirror selected aspects of the system used by Hotels.com. Sample data was used to demonstrate database interaction in potential real-world scenarios through the effective use of SQL queries and transactions.

A sample of SQL queries used to test the database can be found in the appendixes and explained further in the corresponding video report.

Scope and Limitations

Being based on such a large domain as Hotels.com, it was essential that a clear scope for the project was defined. By default, it was limited to only the UK subdomains of the site. Group consensus was reached on focusing primarily on hotel booking specifically as opposed to the multitude of other property types now available, such as holiday rentals and B&B's. Nevertheless, it was agreed that if the initial hotel database was designed effectively then an extension to other property types would be relatively straightforward to achieve.

Another constraint was that the database only needed to be normalised as far as the third normalised form (3NF). This will be touched upon further in the next section. For example, in the case of `address` it has been normalised out into both `city` and `country`. This could have been taken further by including neighbourhood or any number of other variations. Anywhere where this could have been the case has been clearly indicated in orange on my early ERD diagrams (see fig. 3). In our group, we left it up to individual choice as to which entities we focused on normalising.

Limits as to the scope of the report itself have had to be drawn in view of the word count allowance relative to the size of the database examined. To mitigate this, where similar design decisions have been taken across multiple tables and any linked tables, only one of these groupings will be examined. For example, where there are `room_amenity`, `room_policy`, `room_accessibility` tables, only one of these will be examined in more detail to save unnecessary repetition. A comprehensive view of all table relationships can be found in the ERD examples in this report and in the appendixes.

Our group was split on the value of including Hotels.com's rewards program in our database design. It is evident that it is an integral part of their company's marketing scheme. Nevertheless, I believe it was too big of a task for this project to include it in the main design without detracting from the focal points of the final database design: customer, hotel, room and payment. This is something I will address further in 'Future Improvements.'

Methodology

Chen (1976) and Codd (1971) are the fathers of the relational database systems model. They were primarily concerned with providing a framework for database design that could be applied universally and systematically. In short, their aim was to create database models that negated the input of repeated or redundant data and, consequently, allowed for more efficient storage, updating and searching of data.

The database design and its many iterations are based on the entity relations (ER) model as originally popularised by Chen. He advocated that a database should reflect the real world, a world populated with entities and their relationships to one another. In the database these entities take the shape of `hotel`, `room`, `attraction` and so on. Within these are related attributes (`room_num`) or links to other entities (`room_price`).

Codd's proposals on database normalisation are the filter through which the database has been organised and divided. He proposed three initial, important stages of normalisation, outlined as follows:

The First Normal Form (1NF) is concerned with getting rid of repeating values in tables, either row values or columns. Each attribute must contain only atomic, single values. From the initial ERD draft, 1NF was implicitly applied having done the entity discovery task prior. One of the simplest applications of this was through the creation of many-to-many linking tables such as `room_booking`, `room_amenity` and `hotel_attraction`. This eliminated any potential column duplication.

The Second Normal Form (2NF) states that no non-prime attribute in the table should be functionally dependent on a proper subset of any candidate key. Put simply, Rollins (2009) states this means 'if a column isn't intrinsically related to the entire primary key, then you should break out the primary key into different tables.' This is shown in the division of the hotel table in fig. 2 which is crammed full of different attributes that are not strictly related to the primary key `hotel_id`. In fig. 3 through to fig. 4 you can see the progressive proliferation and dividing of tables in order to apply the principles of 2NF.

The Third Normal Form's (3NF) goal is to remove any field dependencies on the subject of the row if the field is not strictly related. If 3NF is not applied, it can lead to data anomalies when updates to the table occur. An example of how this has been applied to specific tables in the database will be examined later in 'Normalisation.'

Generally speaking, many database designs are robust once 3NF has been applied throughout, hence why our project is limited to such. However, depending on the needs of the system or client it may be appropriate to normalise even further. Decisions to do so need to weigh up the cost of CPU performance (queries that require complex joins) versus storage (large datasets). There is a trade-off either way.

Preliminary Research

Initial plans for the database were mapped out through a series of explorations of the Hotels.com website. This included paying particular attention to the website's sign-up process, including customer information and formatting, to the filter options provided in the hotel search function, and the booking payment process. Anything which was deemed an entity or attribute was then noted and developed further or ignored (see fig. 1). This was essentially a case of reverse engineering, whereby the backend structure was surmised from our exploration of the frontend.

customer	reward	booking	theme/type	search	Hotel
~~customer id~~		hotel	Spa Hotel	location	hotel name
email address		room type	Romantic	neighbourhood	star rating
password		cancellation polic	Luxury	dates	VIP badge
first name		payment option	Boutique	guest number	hotel description
last name		first name	Hot Springs	room type	~~price~~
permissions for contact		last name	Family-friendly	star rating	~~room type~~
membership number		email address	Historic	distance to landmark	images
		mobile number	Casino	guest rating	geolocation (long and la
room		check-in date	Winery/Vineyard	price	~~hotel address~~
room name		checkout date	Golf	amenities	~~customer reviews~~
capacity		check-in time	Beach	cancellation policy	~~customer rating~~
bed type		checkout time	Business	payment terms	~~COVID policies~~
cancellation policy		special request	Adventure	rewards participation	~~amenities~~
room amenities		accesibility reque	LGBTQ Welcoming	accomodation type	~~attractions~~
		card number		theme/type	hotel rooms
attraction		expiry date	**hotel amenities**	accesibility	hotel floors
name		security code	breakfast free	check-in instructions	room availability
geolocation		save card?	pool	check-in requirements	~~check-in time~~
attraction description		currency	wifi free	pets allowed	~~check-out time~~
		total price	spa	internet	~~check-in requirements~~
room amenities		confirmation num	airport transfers	parking facilities	year built
air con			gym	FAQ	
coffee machine		**price**	parking free		
bathtub			bar		
smoking		room price	restaurant		
non-smoking		tax	Internet access		
pet friendly		fees	Smoking areas		
kitchen		city tax?	24/7 reception		
crib		pet fee	meeting facilities		
connecting room		cleaning fee	business facilities		
safe box			electric vehicle charging		
daily housekeeping			casino		
smart TV			childcare		
hair dryer,			terrace		
phone			garden		
rollaway bed			laundry facilities		

Figure 1: Initial entity discovery notes

Early Designs

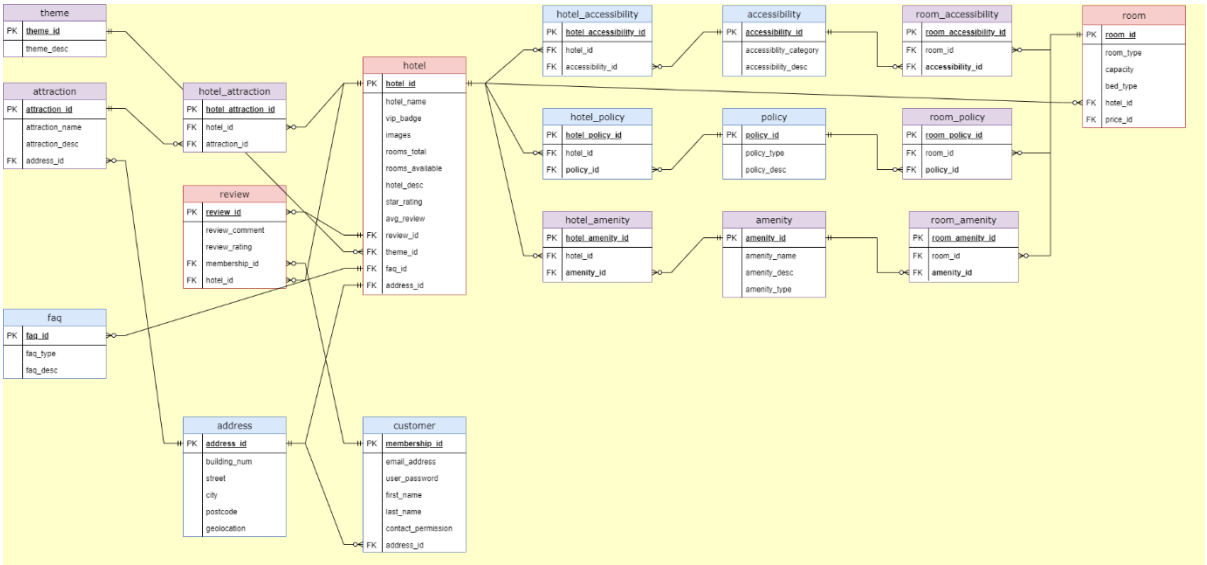


Figure 2: ERD Version 1 (draw.io)

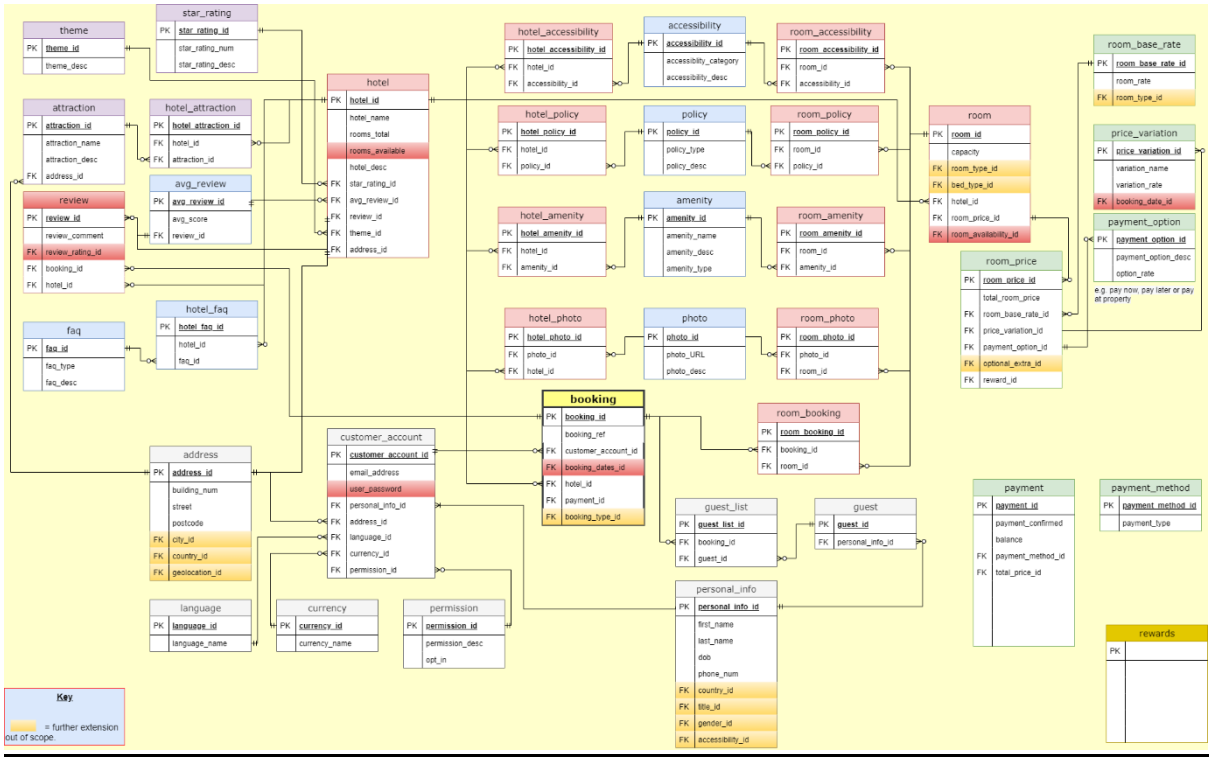


Figure 3: ERD Version 3 (draw.io)

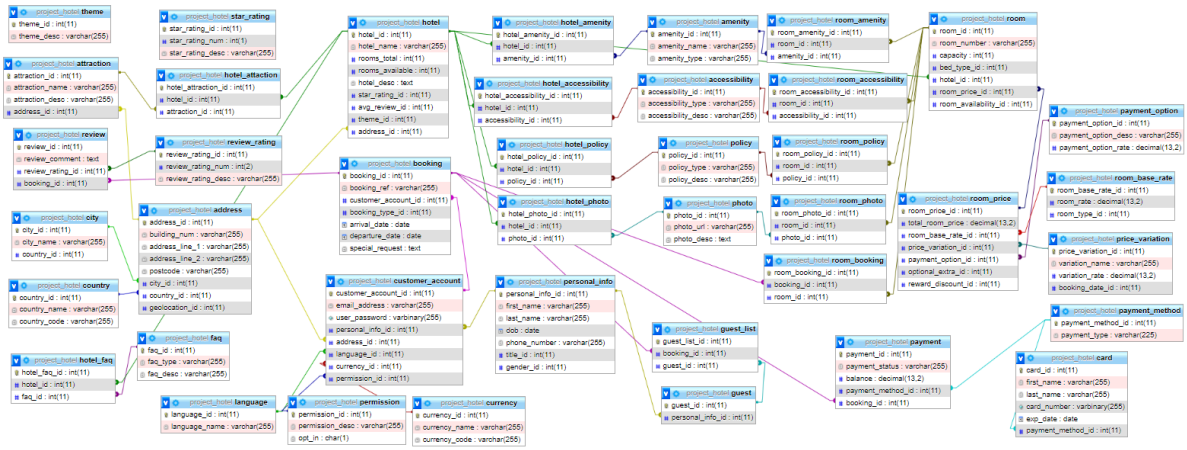


Figure 4: ERD Final Version (phpMyAdmin Designer View)

Key Decisions

Primary Keys

It was agreed across the group at an early stage that surrogate keys would be created for each table, whereby the autoincrement feature in *phpMyAdmin* would assign unique values to each row. The reasoning behind this was straightforward; it simplified the process of ensuring primary keys were unique and not pseudo-unique as is often the case with other candidate keys such as user emails.

Using an integer also simplified the query process later. Similarly, it was also agreed to keep primary keys out of view of the user. For example, where booking details needed to be displayed, instead of displaying the `booking_id` INT primary key, another table column called `booking_ref` with randomised VARCHAR values would be used instead. This is reflective of Hotels.com's own booking information provided to customers.

Table design

One of the key considerations within the group was whether hotel and room amenities should be assigned separate tables or within a single amenity table. Our group argued either option was workable without any great consequence. In the end, I chose to store all amenities in a single `amenity` table, with a many-to-many link table connecting to `room`, and the same to `hotel` (see fig. 5). Additionally, if the `amenity` table needed to be seen in isolation, a separate `amenity_type` column was added to distinguish between hotel and room amenities.

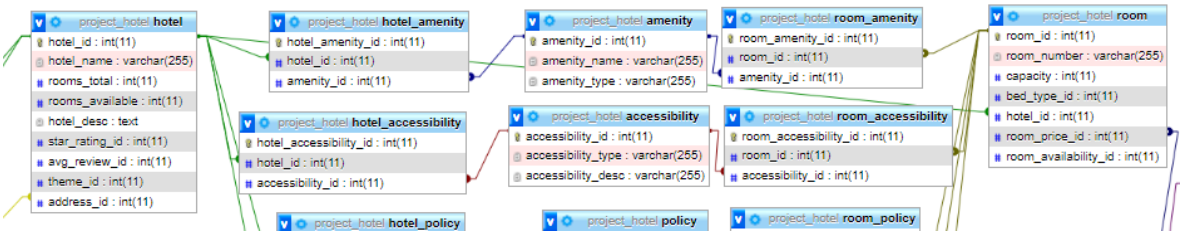


Figure 5: Subsection of ERD displaying linking tables between main hotel and room tables

Some early design considerations related to the need to calculate certain values, such as average review for each hotel based on ratings, total room prices based on combining a number of price factors, and room availability from a hotel's total rooms minus bookings for each day. Specifically in relation to average review score, I had planned to include an `avg_review` table (see fig. 3). The reasoning behind this was that as average reviews for each hotel would change with the addition of each new review linked to that hotel, then a table to store this new review score value in would be the most efficient in terms of processing power and time. In discussion with the group, I decided not to proceed under this line of thought as the consensus was that even a dynamic value such as average review score would be best left to a Java or PHP developer who would be able to better manage the calculations needed on his end.

Ensuring that the database could accurately query room availability on given dates and account for all rooms booked proved tricky to solve. In the end, the group proposed that we approach it as follows: if `room_id` is in `room_booking` table then check for date range it is booked and mark as unavailable on those days. A separate `rooms_available` table (not used in final design due to scope constraints) would recalculate total hotel rooms available on given date by subtracting from `total_rooms` field. Nevertheless, this would be coded by someone working with Java or PHP.

As a workaround, the final database is set up in such a way that by using a complex combination of SQL functions, a query can be carried out that returns all rooms available within a certain date range (see Appendixes C and O).

Data types

A comprehensive view of datatypes used for each field can be viewed in the final ERD version (see fig. 5).

SQL, specifically MySQL, has certain limitations within the datatypes offered. For example, in the `permission` table, ideally the value within this field would have been stored as a boolean for each customer. However, MySQL stores these as TINY INT, translating booleans from integers, where 1 = true and 0 = false. This opens the door to potential input errors, i.e. if any other integer is entered. The decision was made to use CHAR datatype instead with a constraint restricting values to either 'Y' or 'N.' This would be more meaninful when returned in a query and is at the same time easily converted by a developer who is higher up the chain.

The DECIMAL datatype, restricted to 13 digits and 2 decimal places, was used for any fields that concerned price or price factors. The rounding precision reflects the two decimal places adopt by every major world currency. The allowance for thirteen-digit numbers accounts for the fact that the conversion rates for some currencies may translate to larger than normal numbers. For example, at the time of writing, if a customer from Venezuela wants to book a night in a London hotel at £100 per night it will cost him 59,556,444.41 Venezuelan Bolívares. Obviously, this is an extreme example from a country suffering hyperinflation but it makes the case as to why datatype selection is important.

The TEXT data type was used for fields which required longer descriptions (`hotel_desc`) or comments (`review_comment`). This was an accurate reflection of Hotels.com's usage of such data, whereby some reviewers kept their comments short and sweet while others wrote *War and Peace*. In reality there is likely a word or character limit built in but for the purposes of this project an absence of a limit was fine.

The DATE datatype was used to store dates in the booking table's `arrival_date` and `departure_date` fields and the `personal_info` table's `dob` field. This ensured both uniformity of data entry and also allowed easy access to some date-specific methods within SQL queries, which proved invaluable.

The VARBINARY datatype was used for encrypted fields such as `user_password` and `card_number`. This will be explored further in the 'Encryption' subsection.

Finally, check constraints were used to ensure there were no anomalies in certain fields, e.g. `payment_status`, where the datatype selected may not have initially been specific or restrictive enough. This could arguably be normalised into a `payment_status` table and linked via foreign key but further normalisation was deemed to be beyond the scope of the project. It was also an attempt at experimenting with database constraints outside of foreign key constraints, etc.

Foreign Key Relationships

The lynchpins of the entire database are the relationships originating from `room_id` and `booking_id`, found in the room and booking tables respectively. These two primary keys can be used to divide

table relationships into two large subsections. `room_id` connects all aspects of the `room`, `hotel` and `payment` entities. `booking_id` connects all aspects of the `customer_account`, `address` tables and the like. They are then connected through the `room_booking` link table, based on the logic that one booking can have many rooms and, equally, one room can have many bookings. Although there are some other connections that link both halves of the database, for example, `address_id`, `room_id` and `booking_id` have the most influence on the flow of the database and joins required in subsequent queries.

Another example of the power of efficient foreign key relationships is in the `review` table. As evidenced in fig. 2, early plans had two foreign key relationships (`membership_id` and `hotel_id`) but as the database design progressed (see fig. 4) this was stripped down to just one foreign key relationship with `booking_id`. The reasoning behind this was simple. As `booking_id` itself connected to both the `customer_account` and `hotel` tables through its own relationships, it made sense to access that information indirectly through those tables instead of duplicating. Moreover, it is a more accurate depiction of how the review system works on Hotels.com, or any e-commerce site for that matter; it ensures that a review cannot be left unless there is an actual confirmed and connected booking. This guards against fake reviews being generated by the hotel owners, for example, to inflate their review ratings. With the correct joins it is easy to draw out specific hotel reviews and corresponding booking dates, as illustrated below:

```
SELECT CONCAT (review.review_rating_id, ' ',
               '-- ',
               hotel.hotel_name,
               ' -- ',
               review.review_comment,
               ' -- ') AS `Waldorf Astoria NYC - Guest Reviews`,
              booking.arrival_date
FROM hotel
JOIN room ON hotel.hotel_id = room.hotel_id
JOIN room_booking ON room.room_id = room_booking.room_id
JOIN booking ON room_booking.booking_id = booking.booking_id
JOIN review ON booking.booking_id = review.booking_id
JOIN customer_account ON booking.customer_account_id = customer_account.customer_account_id
WHERE hotel.hotel_name LIKE 'Waldorf%'
ORDER BY booking.arrival_date DESC;
```

Waldorf Astoria NYC - Guest Reviews	arrival_date
10 -- Waldorf Astoria New York -- Excellent hotel ...	2022-10-30
10 -- Waldorf Astoria New York -- If it was good e...	2021-11-02
9 -- Waldorf Astoria New York -- Absolutely fabulo...	2021-08-04

Figure 6 - Review search query example

Normalisation

As a group, we agreed it was crucial to normalise the original `customer` table. Initially, this table was used to store personal attributes such as name, permission, etc (see fig. 2). However, it was noted that when making a Hotels.com booking that included guests, guests did not actually need a customer account – all that was required were some simple details like full name and age. This meant if we were to store guests in the original `customer` table, we would need each guest to create an account or leave some of the fields null, e.g. email and password. This would create a whole host of difficulties when it came to querying this table and any linked tables, not least in writing joins and composing aggregate functions which take null values into account (van der Meyden, 1998, p.344). In order to overcome this and more accurately reflect Hotels.com, a separate `personal_info` table was created to both normalise the data and allow this table to link to `customer_account` and `guest` (see fig.3).

Some other easy normalisations were carried out in relation to hotel star rating and review rating. Initially it looked as though the star rating for each of these could be shared in one table but in the end, it made more sense to create separate `star_rating` and `review_rating` tables. The reason for this is that on Hotels.com each rating number had an associated description, e.g., 6 = good. These descriptions were different for hotel ratings and review ratings as the stars have a different meaning altogether. Each table could then have its own separate descriptors.

The application of database normalisation up to 3NF was primarily concentrated on the four focal points of the database: `customer_account`, `hotel`, `room`, `booking`. At first glance, each of these tables appears to be quite dense. However, on closer inspection most of the fields within the tables are foreign keys. Even though there is a lot of information referenced within the tables, they are just that: references. This means that should something need changed or should something try to be mistakenly deleted a foreign key constraint kicks in and ensures that referential integrity remains intact. Further still, it ensures that there is no redundant or duplicated data which can use up memory and cause inefficiencies in processing times when applied to large datasets.

In some cases, not every entity needs normalised. For example, not all of the `booking` table is normalised. This was done consciously with both `arrival_date` and `departure_date`. Both for user simplicity and because we do not necessarily need to normalise for normalisation's sake. However, there are some interesting uses and developments that could arise from a normalised `date` table. These will be examined in 'Future Improvements.'

Like the `average_review` table, the `room_price` table is of particular interest. In the early design stages, it was primarily experimental but as the database developed it became a crucial part of the room subsection. Our group agreed that it was too simplistic to store a hardcoded room price within the `room` table and it would make it less flexible. For example, if a booking's date was to be changed then the price of a room may also change. Having a separate `room_price` table ensured that the principle of 3NF was applied, whereby price was not dependent on each room. As Rollins (2009) clearly summarises, 'pull out columns that don't directly relate to the subject of the row (the primary key), and put them in their own table.'

The various factors that would combine to calculate a total room price were stored in autonomous tables and linked by foreign key to `room_price` (see fig. 4). From here the values could be combined and calculate a `total_room_price`.

In theory, this made perfect sense. In reality, due to the limitations of SQL, this proved difficult. Using precise joins in queries and some SQL wizardry the total room price was able to be calculated for each room (see appendixes J and K). The issue was that this was displayed via a computed column and unable to be stored permanently in an indexed column. The PERSISTED data command only works on calculations that access values from the same table, whereas the data being calculated was from multiple tables. Nevertheless, there is a case to be made for computed columns in this instance. A computed column uses up more processing power but, at the same time, uses less storage space as it is not stored on the disk. Further considerations in this regard will be touched upon in the 'Future Improvements' section of this report.

With regards to the normalisation of data within the `room_price` table, as mentioned, the various factors that would influence price were given their own tables. This decision was made to better

reflect the Hotels.com booking system. Some of the factors included were `payment_option`, referring to ability to pay now or pay later; `room_base_rate`, referring to the assigned base rate of each room based also on a hypothetical foreign key relationship with a `room_type` table, e.g., penthouse suite vs. standard room; `price_variation`, referring to the increase or decrease of room rates due to the time of year or events.

Encryption and Hashing

Basic data encryption and hashing has been used for demonstrative purposes only. In reality, the level of encryption that would be applied to user passwords and card data would be infinitely higher. As a group we agreed that were this a real company database, card payments would be left to the responsibility of a third-party company such as Sage, rather than risk damage to customers and company reputation if security was breached.

Within the database, the `user_password` (see appendix A) and `card_number` (see appendix I) fields are of datatype VARBINARY to demonstrate how encryption would be included in a real-world scenario. The `card_number` field was an obvious candidate for encryption, being the gateway to a whole host of potential abuses. The Advanced Encryption Standard (AES) was used to encrypt the card number. It creates a cipher text using a secret key to encrypt and a secret key to decrypt the cipher through the AES_DECRYPT () function. By today's security standards the version of AES used is not very secure and can be easily cracked. In addition, it is worth noting that encryption is best performed through coding on the server side, outside of SQL itself, as often secure data can be passed through as plain text unbeknown to the person trying to encrypt the data. This is a huge vulnerability in a database.

Whereas the encryption method was used to store card numbers, it was decided that the password would be hashed using the salted hash method. Unlike encryption, which is a two-way system, hashing is a one-way system where data of any size is mapped to a fixed length. In simple terms, in the case of passwords, it allows the database to check if the password entered matches the same exact values in the same exact order of the user password stored on the system.

Future Improvements

As mentioned at the beginning, the final design is not a million miles away from being able to integrate other property types beyond hotels. The beginnings of this could be made through changing the `hotel` table to `property` table and having a foreign key within it that linked to a separate `property_type` table that contained values such as hotel, B&B and holiday rental. Other tables such as `amenity`, `policy` and `photo` could be appropriately updated to include new property types. In the end, the core elements of the database would not change.

The final database design could have included a basic rewards system which linked to the `customer_account` table and influenced the `room_price` table (hypothetical foreign key included as `reward_discount_id` in fig. 4). However, this would not have done justice to the complexity of Hotels.com's rewards scheme. It is arguably a separate database in and of itself. If the database were

to be revisited, one of my key considerations would be to include a more complex reward subsection as it is an important aspect of the company's website.

Again, if the database design was revisited, more experimentation would be done with the `price_variation` table and its various link tables. As alluded to earlier, an independent `date` table could be created containing not only a `date_id` field corresponding with a date but also other fields that allowed descriptions or categories such as `holiday_type`, `day_type` (weekend or weekday), etc. It would then be possible to link the `date` table to the `price_variation` and `booking` tables via foreign key. This would mean that the `price_variation_id` would automatically change dependent on the time of year a date fell on, e.g. Christmas or Easter.

As mentioned previously, even though the `room_price` table was successful in the end, the computed column limited the usefulness of the table. Limitations within SQL meant that trying to store the values as indexed values or as variables was overly complex and extremely rigid. In all likelihood, if the database was to be revised then some of the calculations carried out in the queries would be streamlined along with some of the corresponding tables. This kind of data manipulation is better left for more advanced programming languages such as Java. Dialogue with the software developer would be required to determine what kind of data they would need access to and the datatypes most easily worked with.

Review

All the objectives of the database design and implementation were met. The initial entity discovery and design were completed under the influence of Chen's (1976) ER model, mirroring Hotels.com's website as much as possible. The database implementation was carried out under the tenets of Codd's (1971) 1NF through 3NF as thoroughly outlined in the previous pages and evident in the progression of the different ERD iterations.

Many improvements to design could be made, as already suggested. However, in light of the project's scope, the database is already beyond what is expected in terms of robustness and size. The database can handle basic data storage and queries, such as filtering hotels by average review or amenities to more demanding tasks such as searching available rooms on a given date range, along with dynamic room pricing, and performing transactions to insert new bookings and payments. Some experimentation regarding calculations using SQL have thoroughly demonstrated the limitations of it as a programming language and the need for interaction with higher level languages for a database to access its full functionality.

Bibliography

Codd, E. F. 'Further Normalization of the Data Base Relational Model'. (Presented at Courant Computer Science Symposia Series 6, 'Data Base Systems', New York City, May 24–25, 1971.) IBM Research Report RJ909 (August 31, 1971). Republished in Randall J. Rustin (ed.), Data Base Systems: Courant Computer Science Symposia Series 6. Prentice-Hall, 1972.

Peter Pin-Shan Chen. 1976. *The entity-relationship model—toward a unified view of data*. ACM Trans. Database Syst. 1, 1 (March 1976), 9–36. DOI:https://doi.org/10.1145/320434.320440

Rollins, A., 2009. *Database Normalization: First, Second, and Third Normal Forms - Andrew Rollins*. [online] Andrewrollins.com. Available at: <http://www.andrewrollins.com/2009/08/11/database-normalization-first-second-and-third-normal-forms/> [Accessed 7 November 2021].

Ron van der Meyden. 1998. 'Logical approaches to incomplete information: a survey' in Chomicki, J. and Gunter Saake (ed.). *Logics for databases and information systems*. Boston: Kluwer Academic Publishers.

Appendixes

Appendix A

Query 1a. - new customer account creation with password encryption

```
/* Add new rows to personal_info, address and customeR_account tables
using transaction */

START TRANSACTION;



INSERT INTO personal_info (personal_info_id, first_name, last_name, dob, phone_number, title_id, gender_id)
VALUES (NULL, 'Jordan', 'Spieth', '1993-07-27', '+1 214 432 6745', 1, 1);

/* use variable and last_insert_id method function to conveniently store autoincrement id */
SELECT @pi_last_id := LAST_INSERT_ID();


INSERT INTO address (address_id, building_num, address_line_1, address_line_2, postcode, city_id, country_id, geolocation_id)
VALUES (NULL, '5000', 'S Great Trinity Forest Way', NULL, 'TX 75217', 12, 2, NULL);


SELECT @add_last_id := LAST_INSERT_ID();

/* This should only be used as a demo or in our project -
in production systems use external code to secure passwords */

SET @email = 'j.spieth@pgatour.com';
SET @password = 'GreenJacket';

INSERT INTO customer_account (customer_account_id, email_address, user_password)
      VALUES (NULL,  @email,     PASSWORD(@password));


#Define our username and email
SET @email = 'j.spieth@pgatour.com';

#Define our password
SET @plainPassword = 'GreenJacket';

/* Start (very) basic password storage cryptography */

#Create a random code, six chars in length
SELECT @salt := SUBSTRING(SHA1(RAND()), 1, 6);

#Concat our salt and our plain password, then hash them.
SELECT @saltedHash := SHA1(CONCAT(@salt, @plainPassword)) AS salted_hash_value;

#Get the value we should store in the database (concat of the plain text salt and the hash)
SELECT @storedSaltedHash := CONCAT(@salt,@saltedHash) AS password_to_be_stored;

#Store this user in the database

INSERT INTO customer_account (customer_account_id, email_address, user_password, personal_info_id, address_id, language_id, permission_id)
VALUES (NULL, @email, @storedSaltedHash, @pi_last_id, @add_last_id, 1, 1);

/* save variable for use when booking later on */
SELECT @cust_last_id := LAST_INSERT_ID();


COMMIT;
```

Appendix B

Query 1b. – demonstrating password decryption

```
#Get the password attempt entered by the user as LOGIN
SET @loginPassword = 'GreenJacket';
SET @loginUsername = 'j.spieth@pgatour.com';


/* Start (very) basic password checking */

#Get the salt which is stored in clear text
SELECT @saltInUse := SUBSTRING(user_password, 1, 6)
FROM customer_account WHERE email_address = @loginUsername;

#Get the hash of the salted password entered by the user at SIGN UP
SELECT @storedSaltedHashInUse := SUBSTRING(user_password, 7, 40)
FROM customer_account WHERE email_address = @loginUsername;

#Concat our salt in user and our login password attempt, then hash them.
SELECT @saltedHash := SHA1(CONCAT(@saltInUse, @loginPassword))
AS salted_hash_value_login;
```

Appendix C

Query 2. – searching for hotel by room availability and city

```
SELECT @ArrivalDate AS DATETIME;


SELECT @DepartureDate AS DATETIME;


SELECT @City;


SET @ArrivalDate = '2021-12-20';


SET @DepartureDate = '2021-12-23';


SET @City = 'New York';


SELECT hotel.hotel_name,
       room.room_number,
       hotel.hotel_desc,
       star_rating.star_rating_desc
FROM room
JOIN hotel ON room.hotel_id = hotel.hotel_id
JOIN star_rating ON hotel.star_rating_id = star_rating.star_rating_id
WHERE room_id NOT IN
    (SELECT room_id
     FROM booking b
     JOIN room_booking rb ON b.booking_id = rb.booking_id
     WHERE (arrival_date <= @ArrivalDate
            AND departure_date >= @ArrivalDate)
       OR (arrival_date < @DepartureDate
           AND departure_date >= @DepartureDate)
       OR (@ArrivalDate <= arrival_date
           AND @DepartureDate >= arrival_date) )
  AND hotel.address_id IN
    (SELECT address_id
     FROM address a
     JOIN city c ON a.city_id = c.city_id
     WHERE (c.city_name = @City) );
```

Appendix D

Query 3. – filter New York hotels by average review

```
SELECT hotel.hotel_name,
       ROUND(AVG(review_rating.review_rating_num), 0) AS 'Guest Average'
FROM review_rating
JOIN review ON review_rating.review_rating_id = review.review_rating_id
JOIN booking ON review.booking_id = booking.booking_id
JOIN room_booking ON booking.booking_id = room_booking.booking_id
JOIN room ON room_booking.room_id = room.room_id
JOIN hotel ON room.hotel_id = hotel.hotel_id
JOIN star_rating ON hotel.star_rating_id = star_rating.star_rating_id
WHERE hotel.address_id IN
    (SELECT address_id
     FROM address a
     JOIN city c ON a.city_id = c.city_id
     WHERE (c.city_name = 'New York') )
GROUP BY hotel.hotel_name
ORDER BY `Guest Average` DESC;
```

Appendix E

Query 4a. – view individual hotel review comments and ratings

```
SELECT CONCAT (review.review_rating_id, ' ',
               '-- ',
               hotel.hotel_name,
               ' -- ',
               review.review_comment,
               ' -- ') AS `Waldorf Astoria NYC - Guest Reviews`,
              booking.arrival_date
FROM hotel
JOIN room ON hotel.hotel_id = room.hotel_id
JOIN room_booking ON room.room_id = room_booking.room_id
JOIN booking ON room_booking.booking_id = booking.booking_id
JOIN review ON booking.booking_id = review.booking_id
JOIN customer_account ON booking.customer_account_id = customer_account.customer_account_id
WHERE hotel.hotel_name LIKE 'Waldorf%'
ORDER BY booking.arrival_date DESC;
```

Appendix F

Query 4b. – view hotel and room amenities

```
/* Hotel amenities standalone */

SELECT hotel.hotel_name,
       amenity.amenity_name
FROM hotel_amenity
JOIN hotel ON hotel_amenity.hotel_id = hotel.hotel_id
JOIN amenity ON hotel_amenity.amenity_id = amenity.amenity_id
WHERE hotel.hotel_name = 'Waldorf Astoria New York'
  AND amenity.amenity_type = 'Hotel';

/* Room-specific amenities standalone */

SELECT room.room_number,
       room.room_number,
       amenity.amenity_name
FROM room_amenity
JOIN amenity ON amenity.amenity_id = room_amenity.amenity_id
JOIN room ON room_amenity.room_id = room.room_id
WHERE room.room_id IN
    (SELECT room.room_id
     FROM room
     JOIN hotel ON room.hotel_id = hotel.hotel_id
     WHERE hotel.hotel_name = 'Waldorf Astoria New York' );
```

Appendix G

Query 4c. – view attractions nearby select hotel

```
SELECT CONCAT (attraction.attraction_name,
               ' --- ',
               attraction.attraction_desc) AS `Things to do...`
FROM attraction
JOIN address ON attraction.address_id = address.address_id
JOIN city ON address.city_id = city.city_id
WHERE city.city_id IN
    (SELECT address.city_id
     FROM address
     JOIN hotel ON address.address_id = hotel.address_id
     WHERE hotel.hotel_name = 'Waldorf Astoria New York');
```

Appendix H

Query 4d. – view attractions nearby select hotel

```sql
SELECT room.room_number,
       accessibility.accessibility_type,
       accessibility.accessibility_desc
FROM room_accessibility
JOIN room ON room_accessibility.room_id = room.room_id
JOIN accessibility ON room_accessibility.accessibility_id = accessibility.accessibility_id
WHERE room.room_number = '2b';
```

Appendix I

Query 5. – book hotel room

```sql
START TRANSACTION;

/* Provisional booking made */
INSERT INTO booking (booking_id, booking_ref, customer_account_id, booking_type_id, arrival_date, departure_date) /
VALUES(NULL, '34562-094', (SELECT MAX(customer_account_id) FROM customer_account), 1,'2021-12-20', '2021-12-23');


SELECT @booking_last_id := LAST_INSERT_ID();

/* Price calculated and payment taken */
SELECT @total_price;

/* Add total price to variable calculated from previous query */
SET @total_price = 400.00;

/*Customer pays by card */
SET @first_name = 'Jordan';


SET @last_name = 'Spieth';


SET @card_number= '1234 5678 1224 5678';


SET @exp_date= '2023-09-08';

/* A very simple secret password */
SET @secretPasssword = 'EasyPassword';

/* Start (very) basic data encryption */

SET @card_number = AES_ENCRYPT(@card_number, @secretPasssword);

/* Insert the record with the encrypted data */

INSERT INTO card (card_id, first_name, last_name, card_number, exp_date)
VALUES (NULL, @first_name, @last_name, @card_number, @exp_date);
```

Query 5. continued

```
/* A very simple secret password */

SET @secretPasssword = 'EasyPassword';

/* Start (very) basic data decryption */

SELECT card_id,
       first_name,
       last_name,
       AES_DECRYPT(card_number, @secretPasssword),
       exp_date
FROM card
WHERE card_ID = LAST_INSERT_ID('card');

/* Update payment table */

INSERT INTO payment (payment_id, payment_status, balance, payment_method_id, booking_id)
VALUES (NULL, 'PAID', NULL, 1, @booking_last_id);

/* Rollback to savepoint and try payment again if payment fails */

#SAVEPOINT payment_stage

/* After payment confirmation, booking is finalised */

INSERT INTO room_booking (room_booking_id, booking_id, room_id)
VALUES(NULL, @booking_last_id, 7);


SELECT *
FROM booking
JOIN room_booking ON booking.booking_id = room_booking.booking_id;


SELECT *
FROM payment;


SELECT *
FROM card;

#ROLLBACK TO payment_stage

COMMIT;
```

Appendix J

Query 6a. – display room pricing factors

```
SELECT room.room_price_id, room_base_rate.room_rate, price_variation.variation_rate, payment_option.payment_option_rate,

/* A calculation which uses combination of various price factors to give total room price*/
ROUND (room_base_rate.room_rate*price_variation.variation_rate)*payment_option.payment_option_rate AS total_price, 2

/* Table joins necessary to access required fields for above calculation*/
FROM room
JOIN room_price
  ON room.room_price_id=room_price.room_price_id
JOIN room_base_rate
  ON room_base_rate.room_base_rate_id=room_price.room_base_rate_id
JOIN price_variation
  ON price_variation.price_variation_id=room_price.price_variation_id
JOIN payment_option
  ON payment_option.payment_option_id=room_price.payment_option_id
ORDER BY `total_price` ASC;
```

Appendix K

Query 6b. – display individual booking details

```
SELECT @last_booking_id;


SET @last_booking_id =
  (SELECT MAX(booking.booking_id)
   FROM booking);


SELECT personal_info.first_name,
       personal_info.last_name,
       booking.booking_ref,
       booking.arrival_date,
       booking.departure_date,
       CONCAT (TIMESTAMPDIFF(DAY, booking.arrival_date, booking.departure_date),
               ' nights') AS stay_length,
              hotel.hotel_name,
              star_rating.star_rating_desc
FROM personal_info
INNER JOIN customer_account ON personal_info.personal_info_id = customer_account.personal_info_id
INNER JOIN booking ON customer_account.customer_account_id = booking.customer_account_id
INNER JOIN room_booking ON booking.booking_id= room_booking.booking_id
INNER JOIN room ON room_booking.room_id= room.room_id
INNER JOIN hotel ON room.hotel_id = hotel.hotel_id
INNER JOIN star_rating ON hotel.star_rating_id = star_rating.star_rating_id
WHERE room_booking.booking_id = @last_booking_id;

/* Show payment info for booking */

SELECT @last_booking_id;
```

Query 6b. continued

```
SET @last_booking_id :=
  (SELECT MAX(booking.booking_id)
   FROM booking);


SELECT @total_stay;


SELECT @total_stay := TIMESTAMPDIFF(DAY, booking.arrival_date, booking.departure_date)
FROM booking
WHERE booking.booking_id = @last_booking_id ;


SELECT room.room_number,
       @total_stay AS 'Stay Duration (Nights)',

       /* A calculation which uses combination of various price factors to give total room price*/
       ROUND(room_base_rate.room_rate*price_variation.variation_rate)*payment_option.payment_option_rate AS total_price_per_night,

       /* Total stay price */
       ROUND (room_base_rate.room_rate*price_variation.variation_rate)*payment_option.payment_option_rate * @total_stay AS total_stay_price,

FROM booking
LEFT JOIN payment ON booking.booking_id = payment.booking_id
JOIN room_booking ON booking.booking_id = room_booking.booking_id
JOIN room ON room_booking.room_id = room.room_id
JOIN room_price ON room.room_price_id=room_price.room_price_id
JOIN room_base_rate ON room_base_rate.room_base_rate_id=room_price.room_base_rate_id
JOIN price_variation ON price_variation.price_variation_id=room_price.price_variation_id
JOIN payment_option ON payment_option.payment_option_id=room_price.payment_option_id
WHERE booking.booking_id =
    (SELECT MAX(booking_id)
     FROM booking);
```

Appendix L

Query 7. – view upcoming bookings

```sql
/* Receptionist at Hotel Central Times Square wants to see all upcoming bookings in order of nearest arrival date */

SELECT booking_ref AS 'Booking #',
      CONCAT(first_name, ' ', last_name) AS 'Booking Name',
      country_name AS 'Country',
      phone_number AS Tel, arrival_date AS 'Check-in',
      departure_date AS 'Check-out',
      CONCAT (TIMESTAMPDIFF(DAY, booking.arrival_date, booking.departure_date), ' nights') AS 'Stay Length',
      payment_status,

      #Counts and displays number of rooms for each booking
      COUNT(DISTINCT room_booking.room_id) AS '# of Rooms',

      #Counts and displays number of guests for each booking
      COUNT(DISTINCT guest_list.guest_id) AS 'Guests', special_request AS 'Special Requests'

FROM country
  JOIN address ON country.country_id = address.country_id
  JOIN customer_account ON address.address_id = customer_account.address_id
  JOIN personal_info ON customer_account.personal_info_id = personal_info.personal_info_id
  JOIN booking ON customer_account.customer_account_id = booking.customer_account_id
  JOIN room_booking ON booking.booking_id=room_booking.booking_id
  JOIN room ON room_booking.room_id=room.room_id
  LEFT JOIN hotel ON room.hotel_id=hotel.hotel_id
  LEFT JOIN payment ON booking.booking_id = payment.booking_id
  LEFT JOIN guest_list ON booking.booking_id = guest_list.booking_id
  LEFT JOIN guest ON guest_list.guest_id = guest.guest_id
WHERE hotel.hotel_name = 'Hotel Central Times Square'
  AND booking.arrival_date >= CURRENT_DATE
  GROUP BY room_booking.booking_id
    ORDER BY arrival_date;
```

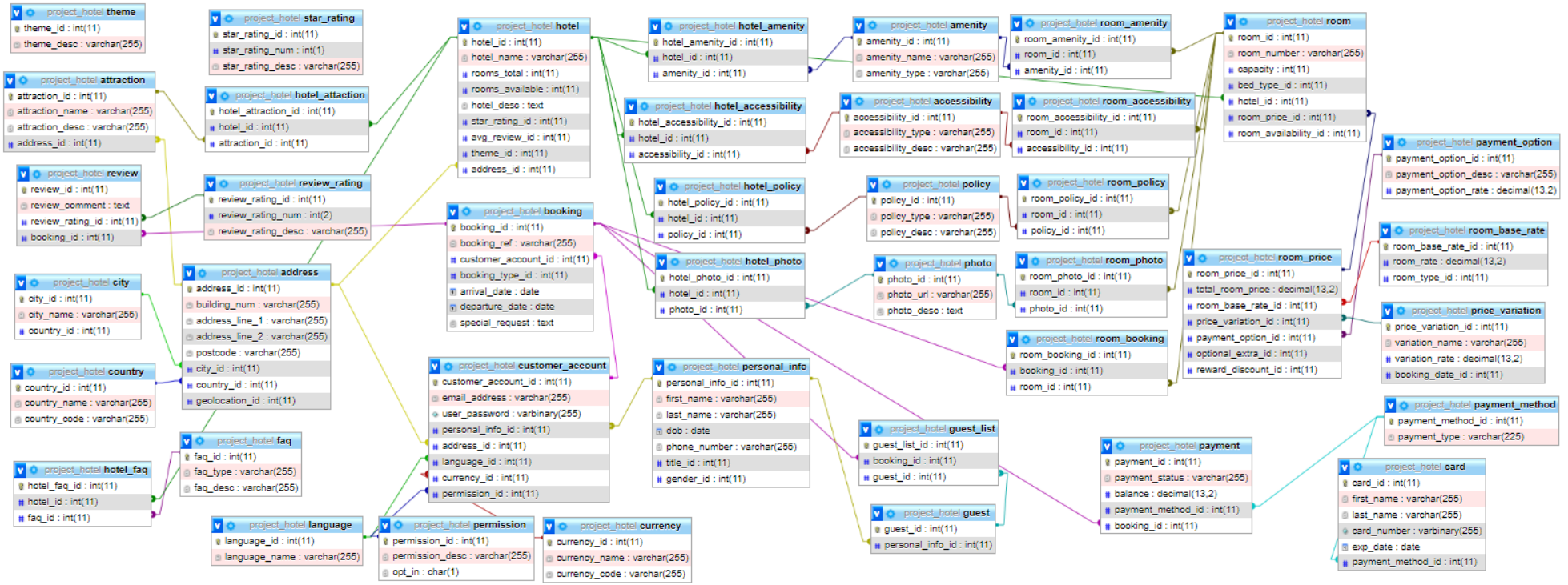
Appendix M

Query 8. – change booking

```
/* Show Mrs Sorenstam's original bookings for comparison */
SELECT * FROM booking

/* Save booking_ref required as variable to decrease chance of input error */
SELECT @booking_ref_num := '52533-028';

START TRANSACTION;

/* Room availability would be checked before date change allowed */
/* ROLLBACK function potentially suitable to avoid double bookings */

UPDATE booking
SET arrival_date = '2022-06-15',
  departure_date = '2022-06-22'
    special_request = 'Late check-out if possible and dinner reservation on 21st June'

 /* If date change meant different price_variation_id then an updated price would be re-calculated */
 /* Out of scope but ideally a date table would be linked to price_variation table to apply this automatically */

WHERE booking.booking_ref = @booking_ref_num

COMMIT;

/* Show changed booking details */
SELECT * FROM booking
```

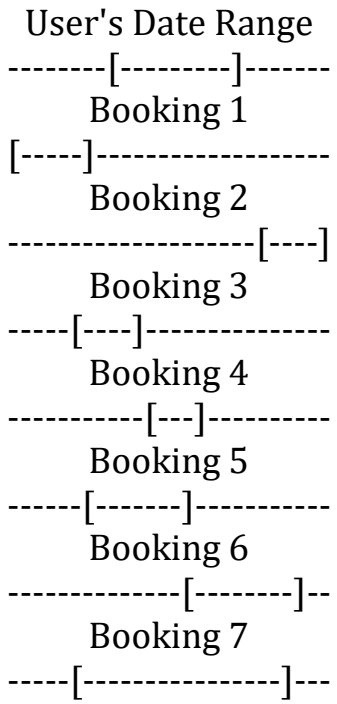
Appendix N

Enlarged version of final ERD diagram from phpMyAdmin designer view



Appendix O

A diagrammatic illustration of the logic used in SQL query 2 (appendix C)



Databases CSC7082
Niall Hodgen [40040160]

Appendix P

A selection of schema notes generated by phpMyAdmin

Column	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
address_id	int(11)		No		auto_increment			
building_num	varchar(255)		No				VARCHAR as not always strict integer, e.g. 5b	
address_line_1	varchar(255)		No					
address_line_2	varchar(255)		Yes	NULL				
postcode	varchar(255)		No					
city_id	int(11)		No			-> city.city_id ON UPDATE RESTRICT ON DELETE RESTRICT		
country_id	int(11)		No			-> country.country_id ON UPDATE RESTRICT ON DELETE RESTRICT		
geolocation_id	int(11)		Yes	NULL				

address table

Column	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
star_rating_id	int(11)		No		auto_increment			
star_rating_num	int(1)		No				kept to 1 in length as it should be a hotel rating of 1-5	
star_rating_desc	varchar(255)		No					

star_rating table

Column	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
customer_account_id	int(11)		No		auto_increment			
email_address	varchar(255)		No					
user_password	varbinary(255)		No					
personal_info_id	int(11)		Yes	NULL		-> personal_info.personal_info_id ON UPDATE RESTRICT ON DELETE RESTRICT		
address_id	int(11)		Yes	NULL		-> address.address_id ON UPDATE RESTRICT ON DELETE RESTRICT		
language_id	int(11)		Yes	NULL		-> language.language_id ON UPDATE RESTRICT ON DELETE RESTRICT		
currency_id	int(11)		Yes	NULL		-> currency.currency_id ON UPDATE RESTRICT ON DELETE RESTRICT		
permission_id	int(11)		Yes	NULL		-> permission.permission_id ON UPDATE RESTRICT ON DELETE RESTRICT		

customer_account table