Patrick Duggan - 40125560

Database Design Report – Booking.com

Booking.com, one of the world's leading online travel agencies, has revolutionised the way we search, book, and experience accommodations worldwide. The website's user-friendly interface, extensive property listings, and personalized recommendations have made it a go-to resource for travellers across the globe. Behind the scenes, however, lies a complex database that powers this travel giant.

The purpose of this project was to design and implement a database that could function as a lightweight interpretation of the Booking.com accommodation database by mirroring the main aspects of the user journey. I will explore the intricacies of managing, structuring, and manipulating vast amounts of data to provide users with a seamless and satisfying experience. I have combined what I have learned in this module with regards to the history of databases, ER notation, attributes, cardinality, normalisation, keys, naming conventions, subqueries, joins, encryption, and transactions to build what I believe to be a strong representation of how the Booking.com accommodation database is structured and how it operates, albeit at a more stripped-back level.

When it comes to database design and implantation, it would be wrong to write a report and not mention the groundbreaking work of Dr. Peter Chen and Dr. Edgar F Codd. Chen's pioneering work on Entity-Relationship Modelling (ERM) has had a profound and enduring impact on the field of database design and information modelling. Chen's development of the Entity-Relationship Diagram (ERD) in the 1970s provided a revolutionary visual tool for representing and understanding the structure of complex data systems. The ERD's simple yet expressive notation, which includes entities, attributes, and relationships, made it accessible to a wide audience, from database designers to business analysts. Codd's rules for database normalisation, which he developed while working at IBM, laid the foundation for relational database management systems (RDBMS), making it easier to organize, access, and manipulate data. His work significantly simplified the complex web of data relationships that plagued earlier database models, ensuring data consistency and accuracy. Normalisation is a fundamental concept in database design, playing a pivotal role in ensuring data integrity, reducing redundancy, and improving overall efficiency. Learning about Codd's third normal form (3NF) database schema design approach helped me reduce duplication of data, avoid data anomalies, ensure referential integrity, and simplify data management.

One of the greatest challenges of this project was deciding what needs to be included and what doesn't. Scoping is an issue faced by every developer and database engineer in the world. I enjoyed this challenge as it is a genuine real-world problem that I will be faced with once I graduate. The early entity discovery and entity-relationship design stressed the importance of making sure I had everything I needed scoped in. It was of crucial importance as anything that must be retro-fitted into a database can cause major issues with data already present. The tables and data that were deemed to be in-scope had to be relevant to what I am trying to display with this project – a database that can function to take the user through their journey from browsing to booking and paying, as it is on Booking.com. I believe my final product has everything that is crucial to the user journey within scope with a few extras to show off the workings of the database.

With all of this in mind, I planned the design of an entity-relationship diagram, ensuring that the data captured accurately reflects the real-world relationships and constraints of the Booking.com system. While designing I tried to structure the system in accordance with a series of normal forms to reduce data redundancy and improve integrity.

As a team we discussed what entities and attributes we thought would be relevant for the database. We each did some entity discovery and came up with the following tables (Fig. 1):

	A	B	C	D	E	F	G	H	I	J	K
1	**Destination**	**Dates**	**Travel party**	**Price**	**Filter options**	**Property type**	**Landmarks**	**Facilities**	**Room types**	**User account**	**Images**
2	City	Checkin	No of adults	Currency	Budget	Hotels	Category	Room	Type	Discounts	Accom images
3	Region	Checkout	No of children	Amount	Property rating out of 5 stars by indi	Apartments	Type	Area	Description	Click tracking	Room images
4	Country		Age of children	Taxes	Activities	Entire homes	Name	Hotel	Beds	Country	Main image
5	Continent		No of rooms	Charges	Cancellation policy	Villas	Distance	Accessibility			
6				City tax	Facilities	Hostels	Long/lat				
7				Cancellation charge							
8											
9	**Accommodation**	**Reviews**	**Guest**	**Payment**	**Card payment**	**Promo code**	**Recent searches**	**FAQs**	**House rules**	**Currency**	
10	Longitude	Guest reviews	Name	Method	Name	Discount amount	IP address	Asked	Checkin time	Type	
11	Latitude	Categories	Email		Card no	Discount code	Recent search history	Answered	Checkout time	Symbol	
12	Name	Accommodation ID	Country		Expiry	Eligibilty	Keywords	Images	Prepayment	Shortcode	
13	Location name	Property review	Mob number						Pets		
14	Amenities (bathrooms/kitchen)	Reviewer name	Special requests						Fine print		
15	Images	Reviewer location	Est arrival time						Cancellation policy		
16	Review rating out of 10 by custome	Review description	Cot request						Catering options		
17	Property rating out of 5 stars by indi	Review rating	Paperless								
18	Cancellation policy	Review date	Save details								
19	Board (self/half/full)	Pros description									
20	Address	Cons description									
21	Description										

Fig. 1 – Excel spreadsheet outlining early entity discovery.

Getting hands-on with the Booking.com website was the best way to become accustomed to the nature of the data, how it was stored and how the different tables would link together. Naturally, we did not agree on everything that was put forward. Personally, I felt as though being able to create a booking as a guest was trivial and bypassed one of the most fascinating and transformable parts of the database – creating a user account.

Creating a user account is more interesting than booking as a guest as it requires some thinking behind it. I could see the scope for showing off what we had learnt so far in the course. When a user creates an account, they must set up a username and password which would allow me to create a secure table where these details would be encrypted, and we could write some SQL queries, using salts and hashes to store and check these each time they log in. Customer accounts also store card information – another chance to show off a different method of encryption – the inbuilt AES Encryption tool within PHP MyAdmin.

Discussing entities and talking about initial designs with the group was useful for ensuring I wasn't missing anything but for me, it was time to start designing my entity relationship diagram. When designing this I tried to make sure that all that was scoped-in would be useful in the representation of the user journey as the purpose of the entire project was about showing how our database would reflect that of the real-world application of the Booking.com accommodation database.

When designing the ERD, I began to write down which field would be the unique identifier/primary key for each table. The primary key must be distinctive for each record so that it can be accessed efficiently and used as a foreign key in other tables, allowing records to be linked together using joins. According to Codd, a database relation (e.g., a database table) is said to meet third normal form standards if all the attributes (e.g. database columns) are functionally dependent on solely the primary key. Most of the primary keys in my database are what you call surrogate keys. Surrogate keys do not have any contextual or business meaning, they are created artificially for the purposes of data analysis. Essentially, they are counters which attach themselves in ascending order to each new record that is added to the table. Not using foreign keys as primary keys is a design choice of mine as I prefer to be able to know which table the primary key is for without having to check the ERD and I could be assured of no duplicate records.

Once I had created a rough plan of what entities I thought would be useful to the user journey and which columns were to be primary keys, I began to create the first ERD which can be seen in Fig. 2 in the appendix. Obviously, this was not fully fleshed out yet but it is a good insight into where I began in this project and is useful in showing how I developed the ERD over time to make it more efficient and interwoven.

One of the earliest design choices I made was to remove tables that I deemed unnecessary, such as the Filter Options table (which would instead be shown through SQL queries), the Language table (as I believed that could be a front-end/browser problem as it had no bearing on the actual data just how the webpage was presented), the Promo Code table (discounts could be more efficiently displayed in the booking_line_item table as either credits or debits depending on how you look at it – I decided they would be credits as we could show the original price of the room on the receipt), the Travelling Party table (this could be better shown on the Room Booking table and guests could be individually recorded in the booking_guest joiner table to allow for multiple to be added) and the Location table (I decided that location was unnecessary and created the possibility of more data redundancy so I instead linked accommodation to the address table as we could still perform searches for locations based off of their addresses using the linked tables such as city and country).

I took a lot of time to think about which tables would need to be linked together. After simulating the user journey from start to finish I got a good sense of where tables are naturally linked together. This includes Booking with Customer, Payment Details, and Room. Room with Accommodation, Room Type, Booking, Currency, Images, Room facilities. Accommodation with Room, Address, Reviews, House Rules, Accom Facilities, Accessibility, Landmarks, Property Type, FAQs, Images etc. It was important to simulate the user journey as I could see firsthand what Booking.com was already linking together, what attributes were being stored in the main table itself and which attributes would have to be their own separate entity.

I will outline some of the further design choices I made between my original and my final ERD. I eliminated any details stored about PayPal or Google Pay other than the account email as this is a frontend problem where the user can be redirected to the relevant external site (out of scope). I realised that holding a customer's card CVC number would hold a huge potential risk to both us and to the customer, so I removed it.

Once I had made these design choices, my ERD began to feel less cluttered and more focussed. The next step was to improve normalisation by removing data redundancy wherever possible. This came through setting up many-to-many relationships for columns that could have multiple entries for each record. This meant that we could link multiple records from one table to multiple records of another. For example, an accommodation can offer many facilities, but if there is a facilities column in the accommodation table then only one facility can be linked to each hotel. Similarly, if the PK accom_id is used in the facilities table then that facility can only belong to that one hotel. To solve this problem of many-to-many relationships, I created multiple of what are called 'intermediate' or 'linking' tables. These tables allow us to store multiple instances of an accommodation linking to different facilities and vice versa. These tables were the basis for a lot of the normalisation and 3NF within the database.

After making the above design choices, I had an ERD that I was content enough with that I could start building out the database. See the improved ERD below:

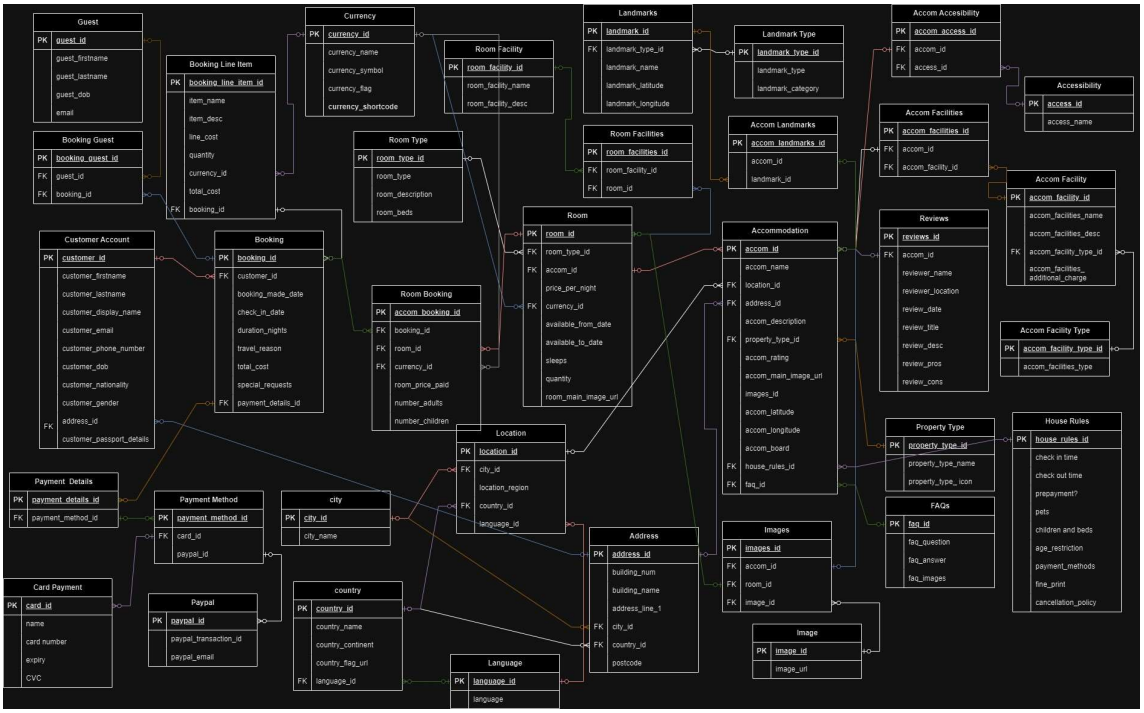

Fig. 3 – Improved ERD design

As can be seen in the diagram, the database design had evolved greatly since the entity discovery and initial ERD stages. One of the most important changes was the introduction of foreign keys into almost every table. This helped my database to become much more interconnected and meant it would be much easier to link records together in different tables, putting the “relational” in “relational database”. With the introduction of foreign keys came the introduction of foreign key constraints. Foreign key constraints enforce referential integrity, which essentially says that if column value A refers to column value B, then column value B must exist.

In Fig. 3 you can see the primary key and foreign key constraint decisions that I made in my design. A good example is the relationship between the Booking, the Room Booking and the Room tables. I used both the primary keys of the Booking and Room tables, booking_id and room_id respectively, as foreign key constraints within the Room Booking table, which acts as the linking table. This meant that the Room and Booking tables could be linked together to form a many-to-many relationship. This mirrors the real-world booking system as on Booking.com you can book many rooms at once under one booking and many rooms can be booked out by different bookings at the same time. By linking the tables together like this it meant that the booking of each room could be individually stored and accessed using a surrogate key (room_booking_id) while linking to both a booking (using the foreign key constraint on booking_id) and a room (using a foreign key constraint on room_id).

Once I began building the database out, I realised that this intermediate table required some additional columns. The other columns I included in the Room Booking table were necessary so that we could access certain information about the record separately to that of the entire booking. These include the price paid for the room (important to have this separate to the full booking and not linked to the room price as this can change if the accommodation changes its price), the quantity of this type of room booked (important for availability searches and to create a total cost price for the booking of said rooms), the total cost of the booking of these specific rooms, and the number of adults and children to each room.

Another important design choice I made was how to link the accommodation table to the many tables related to it. I decided that the most efficient way to do this was to use it was a foreign key in the other tables such as house rules, reviews, and FAQs. This meant that each record in each of these tables would link directly to a specific accommodation. As there will only be one set of house rules, one set of reviews, and one set of FAQs for each accommodation but each of these tables will contain multiple fields, it made more sense to have them as their own standalone tables that we could access for each accommodation using joins. This reduced the size of the accommodation table greatly and improved the normalisation of the database.

Another interesting table, or set of tables, I want to mention are the images tables (see Fig.4 in Appendix). This was a challenging aspect of the database to design as one accommodation or one room can have hundreds of images and these images can be reused across chain hotels. I solved this issue during the database build by splitting it out into two linking tables (accom_images and room_images) and a standalone table just for the images. This means that each accommodation/room can access many images and each image can be accessed by many different accommodations/rooms. By creating a single table for images with simply the primary key of image_id and the image URL, this meant that images can be used by both an accommodation or a room if necessary. This works well in a real-world application as many accommodations/rooms may use a brand logo or a stock image or a coming soon image without having to be manually input and linked each time. This is exactly what companies do on Booking.com so I thought it was important to replicate this in the database.

After reviewing Booking.com I realised that the main image for each Accommodation or Room had a main image that was displayed larger than the rest. This gave an interesting dilemma of whether to store that image within the newly created images table and access them through the room_images/accom_images table and apply some frontend programming to make them display differently or to store them separately from the other images within the Accommodation/Room tables themselves. I chose the latter as I believe it would be easier for the frontend team if these main images are already defined and ready to be slotted into the main image slot. This should allow for less processing power and time of having to go and manually find these images, also how would you be able to define that this certain image is the main image for future users of the database?

Another important database design decision that came later in the process was the Booking Line-Item table (see Fig.6 in Appendix). This table took in booking_id as a foreign key and allowed us to compute the total cost of all items on the bill together as well as input debits to the bill when the customer had paid the deposit, used discounts, or paid the bill off in part or in full. The creation of this table allowed me to then write some queries which would serve as a receipt or bill to the customer for the entire cost of the booking, not just the cost of each room and to keep a track of how much outstanding balance there was left. This table is a much more efficient manner of calculating and keeping a track of all the costs to the booking. If we had split all of the different costs into different tables, then this would require long queries with numerous joins and just a whole lot more work than keeping them all centralised and linked to the bookings through the booking_id.

As previously mentioned, having a secure way to store the customers sensitive information was of upmost importance when designing this database. My initial designs did not have this in place as I wasn't sure as to how this would look in practice. When I built the database, I realised it would be possible to store their sensitive data (that is their email, username, and password) in its own separate table, the Security table. This was linked to the customer table using the Primary Key security_id as a foreign key in the Customer Account table. This meant that their sensitive data wouldn't be stored alongside the rest of their details, and we could perhaps add an extra layer of

security to this table within the database (too advanced for this project). This would tie-in nicely with the user journey as this Security table would be the first table that the customer would update with data as within the Booking.com website you must create a username and password before you can create the rest of your account.

To make sure that this data was secure, I would need to encrypt the user's password using basic password storage cryptography. This involves writing queries to create salts and hashes based off the user's password and a random number generator. It was important that this data would be accessible when writing the queries so that we could make changes without making changes to the table directly. The encrypted password would then be stored where it could be checked again upon the user logging in.

Below is the final ERD design of my database (larger version available in Appendix – Fig. 7). All the above design choices lead to a refinement over time as well as seeing possible shortcomings/improvements once I began working with data within the database.

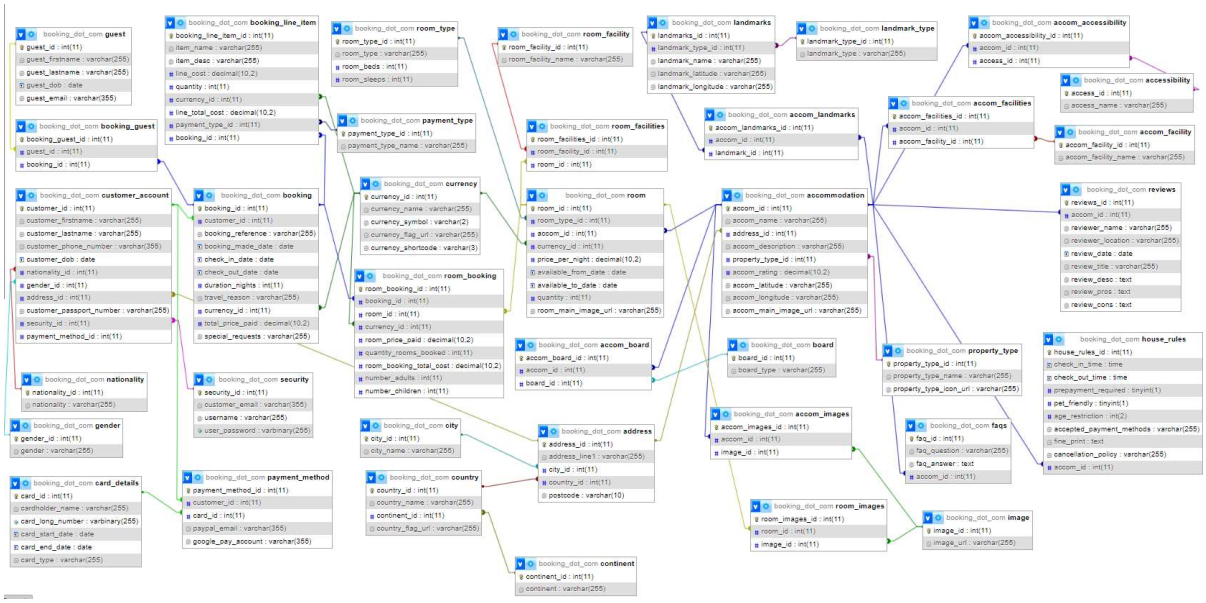

Fig. 5 – Final ERD/database schema

As can be seen in the diagram above, the final ERD or Database Schema has come a long way since the start of the project. Normalisation has been the key focus as the database has developed. I achieved First Normal Form (1NF) throughout the database by: making sure a single cell cannot hold more than one value, satisfying the rule of atomicity; there are primary keys in every table for identification; there are no duplicated rows or columns (which could lead to data redundancy and unwanted complexities within the database); and each column must only have one value for each row in the table (this was achieved by creating new tables for any attributes that may contain more than one value, for example, room/accommodation facilities and then linking the many-to-many relationship via an intermediate table). As 1NF only eliminates repeating groups, not data redundancy, it was important to make sure the database adhered to the Second Normal Form (2NF) as well. This was achieved by making sure no tables had any partial dependencies. That is, all non-key attributes had to be fully dependent on the primary key. Once the tables in the database were in 2NF, we had eliminated repeating groups and data redundancy, but we had not eliminated transitive partial dependency, 2NF does not put any restriction on the non-prime to non-prime attribute dependency. That is addressed in third normal form. A database relation (e.g. a database table) is said to meet 3NF standards if all the attributes (e.g. database columns) are functionally dependent

on solely the primary key. Codd defined this as a relation in 2NF where all non-prime attributes depend only on the candidate keys and do not have a transitive dependency on another key.

The data type used for each column within the database was something that required constant thought and change. It is important that each field has the right data type and size so that real-world data can be captured without causing an error. A good example of this is the email field within the Security and Guest tables. If you look up what the longest possible email length is, it is 320 characters. While this may feel unnecessary, it is important that our database allows for such things if it is an acceptable email address. This is important in the Booking.com website as someone may have that length of an email address and not be able to set up an account which in turn loses the company potential bookings and more importantly, money.

For columns where there could be numbers and letters used, I went for the standard 250-character varchar. This allows for any combination or length (up to 250 characters) of numbers, letters or special characters and is an easy standard to implement across the database where no bespoke solution was required. For any primary or foreign keys, I used int of length 11 as no id should ever be longer than 11 characters. For any fields where there may be a price, I used a decimal of length 10 with 2 decimal places as this is satisfactory when referring to monetary values. Columns such as review description/pros/cons and House Rules fine print I set the data type as text as we want the user to be able to input as much data as they feel necessary. It is of value to Booking.com that users can create descriptive reviews of accommodations as this will help other users within the system when deciding which accommodation would be best suited to them.

Now that I had finalised the database design, which tables to include, which primary keys, foreign keys and relationships had to exist between tables as well as which data type each column should be, it was time to fill out the table with some data. When deciding what data was to be input into the table, I wanted to make sure that I didn't include any irrelevant data, that is any data that would not impact the user journey from start to finish that I had simulated on Booking.com. This included coming up with a fictional user who wished to make a booking on the Booking.com website. I decided on a couple who wanted to book a hotel in Belfast for 3 nights in December 2023. The couple wanted a hotel with; a 7.5 or above rating, parking at the hotel, the board to be bed and breakfast, and they had a budget of £320. To ensure I had relevant data to play around with, I made some similar bookings (that would affect availability) prior to their booking. I made several different accommodations in Belfast, including multiple hotels. I created dynamic pricing dates around their booking due to limited availability of the room they desired. I created several property types so that we could show the filtering for hotels. I created multiple accommodation facilities and linked different ones to different hotels in Belfast to again show off that the filtering query would work as desired. I created a full range of board options for accommodations and linked them differently to several hotels. I created prices per night that would fall within their budget and outside of their budget.

Improvements:

While brainstorming, doing entity discovery, designing ERDs, building the database, creating PKs, FKs and FK constraints, populating data, and performing queries during this project, I have always tried to make sure that all of my decisions made sense in a real-world application such as that of Booking.com and that I was adhering to the principles of database optimisation laid by Chen and Codd all those years ago. However, within databases, there is almost always room for improvement.

With limited time and resources, I had to create a simplistic version of the Booking.com accommodation database, this meant that certain design decisions had to be made along the way to ensure that I could build an effective database that allowed for a complete user journey from start to finish without getting bogged down in filling out unnecessary tables and data that would be out of scope for what I was trying to achieve.

One of the main improvements that could obviously be carried out with this database is the amount of data I have stored in each table. As the data is only relevant to the user journey I chose, there isn't much, if any, data for things outside of the searches I created. For example, if you wanted any accommodation outside of Belfast that wasn't a hotel, you would find nothing. If you wanted to search via proximity to landmarks within Belfast (my fictitious user did not), you would find nothing as the landmark's tables have no data whatsoever. The list goes on and on but again, this was unnecessary to show that my database works and that queries can be run and can update tables without having to manually insert. With more time (and money), this database could be absolutely filled with data and could allow for incredibly precise searches containing refinement of almost every possible attribute of an accommodation. A site like Booking.com is run by thousands of people and has been constantly updated and refined over the 27 years it has been operational. Therefore, part of our task was to define a small scope that would allow us to demonstrate the user journey like that of a real-life one on Booking.com.

On the Booking.com website whenever you perform a search based on location, dates, and number of adults/children, the price of the relevant accommodations appears as a total price of the room cost per night times the number of nights desired. Within my database, this search only presents the price per night of each accommodation, this is because I was only storing price per night in the room tables and could not find a way to present this within the result table. Perhaps this may be a frontend algorithm, but it would've been nice to be able to see this information without having to manually perform the multiplication myself and display it in a separate results table.

The hotel rating attribute within Booking.com is an amalgamation of the user's overall review score. This would be done by creating a separate table for rating and performing calculations to sum up all of the user's overall review scores and dynamically updating and storing this within the hotel rating table. This is something I would include in a further adaptation of the database as it is an important part of the real website. This would mean that when users left a review/rating of the hotel, it would automatically update the accommodations overall rating without having to manually calculate and update it yourself. Storing the accommodation rating as a decimal column in the accommodation table is rather primitive and reduces efficiency within the database.

Another improvement I could make is to store travel reason as a Boolean or tiny int as opposed to using VARCHAR. This would mean that we could do some data analysis with greater ease as to what kind of reasons guests are staying at a hotel. In the real-world this could let accommodations know what sort of travellers they need to appeal to so that they could increase bookings/desirability. I also should have created a separate table for passport details as there should be first name, last name, issuing country, and expiry date as well as the passport number. This would need to be separated into its own table for normalisation and to not take up too much room in the customer account table. While this information is not necessary to make a booking on Booking.com it is necessary to have a complete customer account.

A key aspect of the Booking.com system that is overlooked by my database is the use of maps to search for accommodations and landmarks. In a real-world user case, a user may search for hotels in a city and then decide which one they wish to book based on its location and proximity to places

within the city. For example, if I had the correct latitude and longitude for accommodations and for landmarks, I could then pass this information through an external mapping service and be able to display on a map where the accommodation is in relation to the rest of the location and could even show its distance from landmarks or points of interest. In the current state, the user would have to take the address of the accommodation themselves and pass this through a mapping application so that they could assess the location and distance to landmarks/points of interest themselves.

Another improvement that could be made is to the security of sensitive information within the database. Currently, I am salting and hashing the users' passwords to store within the database and while this is fine for this project, in the real-world, companies go to much greater lengths to ensure that their customers data is safe and secure. If I was to develop this database, I would make use of bcrypt to encrypt, store and check the users' passwords. Bcrypt stands out among other hashing algorithms because it uses a cost factor. With it, you can determine the number of password iterations and hashing rounds to be performed, increasing the amount of time, effort, and computational resources needed to calculate the final hash value. The cost factor makes bcrypt a slow algorithm that takes significantly more time to produce a hash key, turning it into a safe password-storing tool. This cost factor is what elevates it above my current salting and hashing method.

For this project, I held card details and used the inbuilt encryption service which is not reliable at all against bad actors but was useful for demonstration of the encryption process. Holding this sort of data in the database would make the data extremely vulnerable as it is very easily extrapolated and decrypted into its original form. Customer card details are usually outsourced to fintech or card processing companies as they have the most secure infrastructure for sensitive data such as this. Removing this sort of risk for the company is a very important design choice when building a database and if I were to keep developing this database, I would not store any card details within the system.

Data validation is another area of my database that could be greatly improved on. At the moment, I am only applying data validation to a few select columns within the database to ensure they are of the correct format and length. Examples of columns that could be improved: Ensuring any URLs for images are the correct format (e.g. contains a working domain name and .com for example); user/guest emails could be validated in a similar way, ensuring they contain @ and a website domain; phone numbers are left as a VARCHAR of size 255, this could be changed so that the customers address is linked to a correct phone number format and this validation be applied when they are adding their number; card details are stored as VARCHAR and should definitely be changed to the correct length and format for card number and card type should be a pre-populated list of accepted card types; postcodes in the address table are a VARCHAR of size 10 and again could be validated depending on which country the user selects; we could also validate users passwords so that the passwords they input are secure to begin with, this could mean including capital letters, numbers and special characters, this would make our salted hashes even stronger and harder to decrypt. There are more examples like this, and I believe this could be done at the database level as well as the frontend to ensure that our data integrity and usability is at its highest level.

One of the early design decisions I made was to remove the option for someone to make a booking through the database without first creating a customer account. While this helped to reduce the scope of the user journey, in the Booking.com website, it is possible to create a booking without creating an account. This would certainly be an improvement I would make going forward as I've tried to mirror the real-world user journey, but this was a deviation from that. To do this I would create a guest customer table and populate it with all the fields necessary to complete a booking on

the website without making an account. This would then need to be able to be linked in with the booking table so that we could show whether the booking was made by a customer with an account or by a guest customer. I would also include some functionality where the guests' details would be temporarily stored, and they would be presented with saving their details to an account to speed up future bookings. This is useful in the real Booking.com sites as customers that have accounts have less details to fill in next time they book and so this really speeds up the booking process. When things are quicker and easier, people are more likely to do them than things that take lots of time, its human nature. The company would see increased revenue as a result of guests becoming customers with accounts.

Another area of the database that could be improved is the payment section. It should be clearer as to when and how customers pay. Perhaps this can be done using links to the card details table and creating a transaction table where transactions are stored, and reference numbers are created. This could have been surrounded with a transaction with rollbacks/save points if the customers payment didn't go through so that the booking wasn't made despite payment being unsuccessful. However, at the moment, I am manually inputting customer payments into the Booking Line Item table which is time consuming and not very efficient.

Conclusion

The objective of this project was to design and implement a database system for a commercial accommodation booking system (Booking.com). The database system was to mirror the operation of selected aspects of the accommodation booking system. We were expected to individually create the database tables, populate them with some sample data and write SQL queries to demonstrate the database.

I believe I have used all the principles we have learned in this course, and used the theory created by the forefathers of databases, Chen and Codd, to design and implement a database system that mirrors that of Booking.com's accommodation database. Having run through the user journey from start to finish countless times, I recorded all of the necessary data I would need to gather to create a database that could operate at a fully functional level to allow me to create a user, create a full customer account, store and check card details, search for accommodation based off of filters similar to that of Booking.com, check the availability of a room before booking, book a room, create a bill for the booking, and change prices dynamically, mirroring that of what accommodations do on the Booking.com website when there is low availability or there is an event on in town.

My database has been made with a keen focus on normalisation as it is structured in accordance with a series of normal forms to reduce data redundancy and improve data integrity. The tables and the relationships between them have been researched, designed, and improved upon throughout. The sample data has been carefully chosen and tweaked to best showcase the usefulness of the database in replicating the user journey throughout the Booking.com website and the SQL queries (see Appendix) have been impeccably written and executed to display the interdependencies between the tables and how the user journey can be replicated through a series of queries, involving retrieving data from multiple tables at once, reporting the results to the user and updating multiple entries/tables at a time without having to manually do so.

Appendix

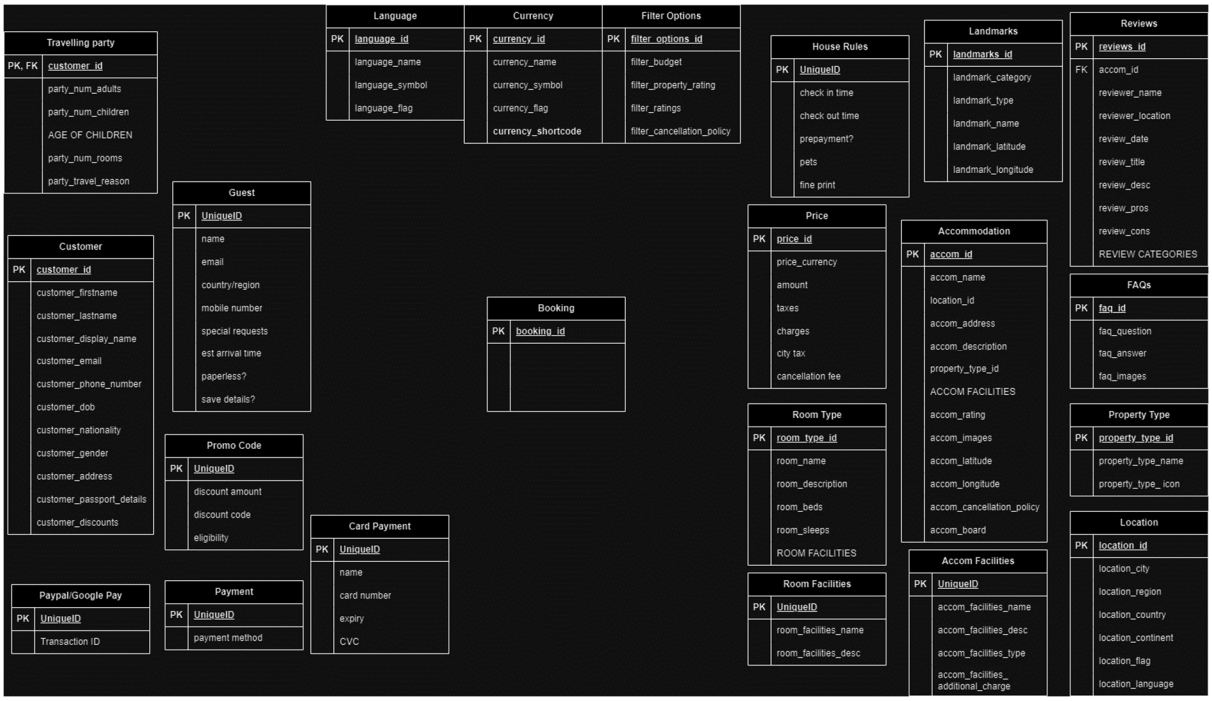

Fig. 2 – First attempt at creating an ERD

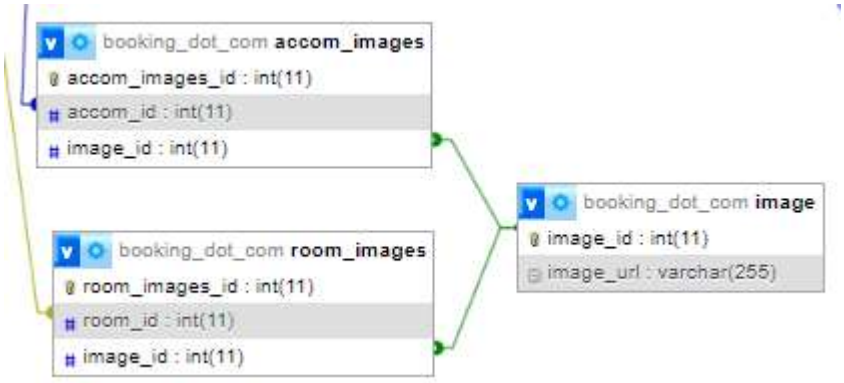

Fig. 4 – Image tables

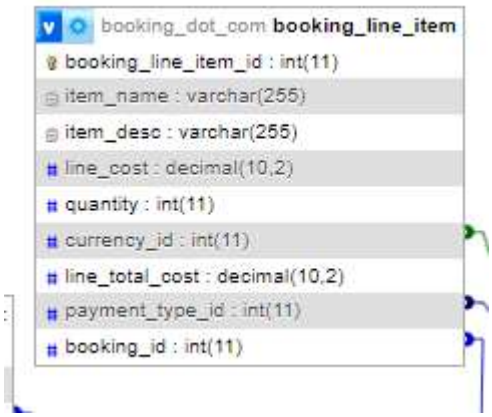

Fig. 6 – The Booking Line Item table

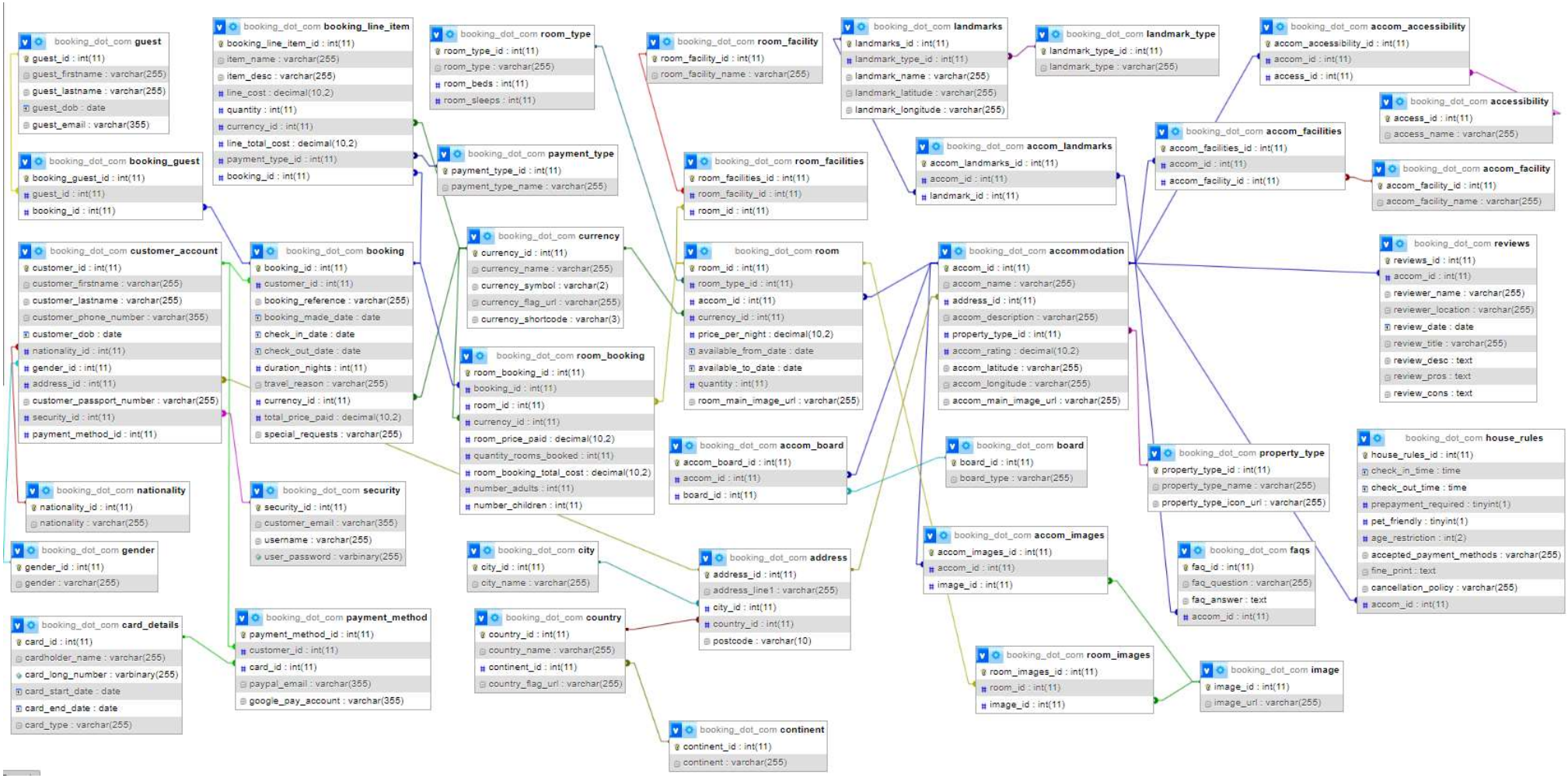

Fig. 7 – Final ERD/Database Schema

Patrick Duggan - 40125560

The following are the queries I will demonstrate in my video:

1) Customer account creation

```
#Showing current customer accounts

SELECT *

FROM customer_account INNER JOIN security

ON customer_account.security_id = security.security_id;


#Customer is asked to create a username and password before creating a full customer account. Usually, they will need to verify their email address by clicking a link that gets sent through to the selected email address - however, that is out of scope for this project.


#The user selects a username and provides their email address

SET @username = 'jbloggs';

SET @email = 'jbloggs@qub.ac.uk';

SET @security_id = 3;


#User sets their password (not very secure)

SET @plainPassword = 'password';


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


#Create a random code, six chars in length

SELECT @salt := SUBSTRING(SHA1(RAND()), 1, 6);


#SHA-1 is a hash function which takes an input (randomly generated code in this case) and produces a 160-bit hash value known as a message digest – typically rendered as 40 hexadecimal digits - we then take the first 6 digits - this is the salt


#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 security (security_id, username, customer_email, user_password)

                    VALUES (@security_id,    @username, @email,    @storedSaltedHash);


#Now we show what full customer accounts have been created - new customer wont appear yet as not fully set up account

SELECT *
```

```
FROM customer_account INNER JOIN security

ON customer_account.security_id = security.security_id;


#Using a right join to show that the customers inital account has been set up on the security table

SELECT *

FROM customer_account RIGHT JOIN security

ON customer_account.security_id = security.security_id;


#Customer fills in the rest of the account details

INSERT INTO address (address_id, address_line1, city_id, country_id, postcode)

                                    VALUES (14, '43 Jimbob Road', 2, 1, 'DU123BL');

INSERT INTO customer_account (customer_id, customer_firstname, customer_lastname, customer_phone_number, customer_dob,
nationality_id, gender_id, address_id, customer_passport_number, security_id, payment_method_id)

                                                VALUES (NULL, 'Joe', 'Bloggs', '07976412375', '09/09/1981', 2, 1, 14, '58372901', 3, 2);


#showing that the full customer account has been created

SELECT *

FROM customer_account INNER JOIN security

ON customer_account.security_id = security.security_id;
```

2) Password checking

```
#Get the password attempt entered by the user at LOGIN

SET @loginPassword = 'password';

SET @loginUsername = 'jbloggs'COLLATE utf8mb4_general_ci;


/* Start (very) basic password checking */

#Get the salt which is stored in clear text

SELECT @saltInUse := SUBSTRING(user_password, 1, 6) FROM security WHERE username = @loginUsername;


#Get the hash of the salted password entered by the user at SIGN UP

SELECT @storedSaltedHashInUse := SUBSTRING(user_password, 7, 40) FROM security WHERE username = @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;


#Compare with what the users password is stored as in the security table

SELECT user_password FROM security

WHERE username = @loginUsername;
```

3) Storing and checking card details

```
#Get the cardholder details to be stored

SET @cardholdername = 'Mr Joe Bloggs';

SET @cardlongnumber = '1234 1234 5678 5678';

SET @cardstartdate= '2021-10-12';

SET @cardenddate= '2025-4-12';

SET @cardType = 'Mastercard';


#Secret password - tailored for each card

SET @secretPassword = 'thisIsTooEasyToGuess';


#Start data encryption

SET @encryptedcardlongnumber = AES_ENCRYPT(@cardlongnumber,@secretPassword);


#Insert the record with the encrypted data

INSERT INTO card_details (card_id, cardholder_name, card_long_number, card_start_date, card_end_date, card_type)

VALUES (NULL, @cardholdername, @encryptedcardlongnumber, @cardstartdate, @cardenddate, @cardType);


SELECT * FROM card_details;
```

4) Decrypting card details

```
#Decrypt using password for this customer

SET @secretPassword = 'thisIsTooEasyToGuess';


#Data decryption

SELECT card_id, cardholder_name, AES_DECRYPT(card_long_number, @secretPassword), card_start_date, card_end_date, card_type

FROM card_details WHERE card_ID = 14;


#CHANGE DECRYPTION PASSWORD SO IT NO LONGER MATCHES ENCRYPTION PASSWORD TO PROVE IT WORKS (added a full stop) - GIVES NULL

SET @wrongsecretPasssword = 'wrong password';


#Data decryption

SELECT card_id, cardholder_name, AES_DECRYPT(card_long_number, @wrongsecretPasssword), card_start_date, card_end_date, card_type

FROM card_details WHERE card_ID = 14;
```

5) Searching for a room

```
#Creating variables we can plug our search parameters in to so that we dont have to hard code each time

SET @cityid  = 1; #Belfast

SET @roomtype = 1; #1 double bed

SET @fromdate = '2023-12-08';

SET @todate = '2023-12-11';

SET @adults = 2;

SET @accomrating = 7.5;

SET @propertytype = 2; #Hotel

SET @accomfacility = 3; #Parking

SET @accomboard = 4; # Bed and breakfast

SET @budget = 320.00;

SET @nights_stay = 3;

SET @daily_budget = @budget / @nights_stay;


#Show all accomodations before filtering

SELECT * FROM accommodation;


#The fields I have chosen to represent from the room search mirrors that of booking.com

SELECT accom_name, room_type, accom_rating, accom_main_image_url, city_name, price_per_night FROM room INNER JOIN
accommodation

ON room.accom_id = accommodation.accom_id

INNER JOIN room_type

ON room.room_type_id = room_type.room_type_id

INNER JOIN address

ON accommodation.address_id = address.address_id

INNER JOIN city

ON address.city_id = city.city_id

INNER JOIN accom_facilities

ON accommodation.accom_id = accom_facilities.accom_id

INNER JOIN accom_board

ON accommodation.accom_id = accom_board.accom_id


#Here we add our filters

WHERE room.room_type_id = @roomtype

AND city.city_id = @cityid

AND (available_from_date <= @fromdate AND available_to_date >= @todate)

AND accom_rating >= @accomrating
```

```
AND property_type_id = @propertytype

AND accom_facility_id = @accomfacility

AND board_id = @accomboard


#Here we have it whittled down to 2 rooms based off our search criteria - now to see which fits our budget

#AND price_per_night <= @daily_budget;


#We have now found the perfect room and hotel for us to stay at given our requirements
```

6) Availability

#We are checking what quantity of a specific room type in a specific hotel are available by detracting the quantity of those rooms booked (from the room_booking table) from the quantity available in the room table for the dates we are looking to book

```
SET @roomid = 6;

SET @datefrom = '2023-12-08';

SET @dateto = '2023-12-12';


SELECT @rooms_available := (quantity - SUM(quantity_rooms_booked)) FROM room INNER JOIN room_booking

ON room.room_id = room_booking.room_id

INNER JOIN booking

ON room_booking.booking_id = booking.booking_id

WHERE room.room_id = @roomid AND ((check_in_date <= @datefrom < check_out_date) OR (@datefrom <= check_in_date < @dateto));


#Fact checking this calculation by pulling up the room and room booking table for this specific type of room within the chosen hotel

SELECT * FROM room

WHERE room_id = 6;


SELECT * FROM room_booking

WHERE room_id = 6;
```

7) Booking a room

#Setting variables so that entering new bookings is much easier

```
SET @booking_id = 4;

SET @customer_id = 11;

SET @booking_ref = '123ABCXYZ';

SET @booking_made_date = '2023-11-02';

SET @check_in_date = '2023-12-08';

SET @check_out_date = '2023-12-11';

SET @duration_nights = 3;
```

```
SET @travel_reason = 'Leisure';

SET @currency_id = 1;

SET @total_price_paid = 0;

SET @special_requests = 'None';

SET @room_id = 6;

SET @room_price_paid = 100;

SET @quantity_rooms_booked = 1;

SET @number_adults = 2;

SET @number_children = 0;


#Showing current bookings

SELECT * FROM booking INNER JOIN room_booking

ON booking.booking_id = room_booking.booking_id;


SET @room_booking_total_cost = (@room_price_paid * @duration_nights) * @quantity_rooms_booked;


#Using a transaction as we dont want the booking table to be updated if the room booking table fails to update for some reason

START TRANSACTION;


#Inserting into booking table using our variables

INSERT INTO booking (booking_id, customer_id, booking_reference, booking_made_date, check_in_date, check_out_date,
duration_nights, travel_reason, currency_id, total_price_paid, special_requests)

                                VALUES (@booking_id, @customer_id, @booking_ref, @booking_made_date, @check_in_date,
@check_out_date, @duration_nights, @travel_reason, @currency_id, @total_price_paid, @special_requests);


#Inserting into room booking table using our variables

INSERT INTO room_booking (room_booking_id, booking_id, room_id, currency_id, room_price_paid, quantity_rooms_booked,
room_booking_total_cost, number_adults, number_children)

                            VALUES (NULL, @booking_id, @room_id, @currency_id, @room_price_paid, @quantity_rooms_booked,
@room_booking_total_cost, @number_adults, @number_children);


COMMIT;


#showing the record has been added to both the booking and room booking table

SELECT * FROM booking INNER JOIN room_booking

ON booking.booking_id = room_booking.booking_id;
```

8) Creating a bill

```
#displaying what entries already exist in the booking line item table

SELECT * FROM booking_line_item INNER JOIN payment_type

ON booking_line_item.payment_type_id = payment_type.payment_type_id;


#this can be changed depending on which booking we are looking to add

SET @booking_id = 4;


#retrieving all of the data related to the booking we are going to add to the booking line item table

SELECT * FROM booking 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 room_type

ON room.room_type_id = room_type.room_type_id

INNER JOIN accommodation

ON room.accom_id = accommodation.accom_id

WHERE booking.booking_id = @booking_id;


#setting the room booking data to be input into the booking line item table

SET @item_name = '1 double bedroom at the Hilton Hotel Belfast - 3 nights';

SET @item_desc = '';

SET @line_cost = (SELECT room_price_paid  FROM room_booking WHERE booking_id = @booking_id);

SET @quantity = (SELECT quantity_rooms_booked FROM room_booking WHERE booking_id = @booking_id);

SET @currency_id = 1;

SET @line_total_cost = (SELECT room_booking_total_cost FROM room_booking WHERE booking_id = @booking_id);

SET @payment_type = 1;


#inserting the data into the booking line item table

INSERT INTO booking_line_item (`booking_line_item_id`, `item_name`, `item_desc`, `line_cost`, `quantity`, `currency_id`, `line_total_cost`,
`payment_type_id`, `booking_id`)

                                                VALUES (NULL, @item_name, @item_desc, @line_cost, @quantity,
@currency_id, @line_total_cost, @payment_type, @booking_id);


#displaying that this booking has been successfuly added to the booking line item table

SELECT * FROM booking_line_item INNER JOIN payment_type

ON booking_line_item.payment_type_id = payment_type.payment_type_id;
```

#adding more line items - extras onto booking

```
INSERT INTO booking_line_item (`booking_line_item_id`, `item_name`, `item_desc`, `line_cost`, `quantity`, `currency_id`, `line_total_cost`, `payment_type_id`, `booking_id`)

                                                    VALUES (NULL, 'breakfast', '3 days of breakfast', 10, 3, @currency_id, 30, @payment_type, @booking_id);


INSERT INTO booking_line_item (`booking_line_item_id`, `item_name`, `item_desc`, `line_cost`, `quantity`, `currency_id`, `line_total_cost`, `payment_type_id`, `booking_id`)

                                                    VALUES (NULL, 'parking', '3 days of parking', 12, 3, @currency_id, 36, @payment_type, @booking_id);
```

#using a transaction here as we are updating the total cost table using a query - we want to ensure that the update doesnt fail due to outside factors effecting our dataset while we run it (e.g. the line item table being updated by someone else as we are trying to calculate the total cost of the relevant line items) - ensures integrity of result - if the customer was given an incorrect total cost and paid it off, they would be angry when we go back to them and explain the total cost was incorrect and they have to pay more - ensures atomicity, consistency, isolation and durability of our data

```
START TRANSACTION;
```

#creating a total cost variable so that we can show the total cost of the credits for this booking

```
SET @total_cost = (SELECT SUM(line_total_cost) FROM booking_line_item

WHERE booking_id = @booking_id AND payment_type_id = @payment_type);
```

#inputting the total cost we have just calculated into the total price paid column in the booking table for the relevant booking

```
UPDATE booking SET total_price_paid = @total_cost WHERE booking.booking_id = @booking_id;


COMMIT;
```

#showing that this has successfully updated the booking table

```
SELECT * FROM booking

WHERE booking_id = @booking_id;
```

#paying off all or part of the bill

```
INSERT INTO booking_line_item (`booking_line_item_id`, `item_name`, `item_desc`, `line_cost`, `quantity`, `currency_id`, `line_total_cost`, `payment_type_id`, `booking_id`)

                                                    VALUES (NULL, 'deposit', '', 100, 1, @currency_id, 100, 2, @booking_id);


INSERT INTO booking_line_item (`booking_line_item_id`, `item_name`, `item_desc`, `line_cost`, `quantity`, `currency_id`, `line_total_cost`, `payment_type_id`, `booking_id`)

                                                    VALUES (NULL, 'discount', '', 50, 1, @currency_id, 50, 2, @booking_id);
```

#showing the statement so far

```
SELECT * FROM booking_line_item INNER JOIN payment_type

ON booking_line_item.payment_type_id = payment_type.payment_type_id

WHERE booking_id = @booking_id;


SET @total_paid = (SELECT SUM(line_total_cost) FROM booking_line_item

WHERE booking_id = @booking_id AND payment_type_id = 2);


SET @outstanding_balance = @total_cost - @total_paid;

SELECT @total_cost, @total_paid,@outstanding_balance;
```

9) Dynamic Pricing

```
SET @room_type_id = 1;

SET @accom_id = 4;

SET @currency_id = 1;

SET @quantity_limited = 3;

SET @quantity_normal = 24;

SET @room_main_image_url = 'imageurl';

SET @check_in_date = '2023-12-08';

SET @check_out_date = '2023-12-11';


#showing the price and available dates before making dynamic

SELECT room_id, room_type_id, accom_id, price_per_night, available_from_date, available_to_date FROM room

WHERE room_type_id = @room_type_id AND accom_id = @accom_id;


#updating the current, normal availability dates for this room to end before the low availability period starts

UPDATE room SET available_to_date = '2023-12-07'

WHERE room_id = 6;


#inserting a new row for the low availability period with an increased price per night of £150

INSERT INTO room (room_id, room_type_id, accom_id, currency_id, price_per_night, available_from_date, available_to_date, quantity,
room_main_image_url)

                    VALUES (NULL, @room_type_id, @accom_id, @currency_id, 150, '2023-12-08', '2023-12-11', @quantity_limited,
@room_main_image_url);


#inserting a new period of normal availability for this room after the uprated period

INSERT INTO room (room_id, room_type_id, accom_id, currency_id, price_per_night, available_from_date, available_to_date, quantity,
room_main_image_url)

                    VALUES (NULL, @room_type_id, @accom_id, @currency_id, 100, '2023-12-12', '2024-12-31', @quantity_normal,
@room_main_image_url);
```

Patrick Duggan - 40125560

#showing that the price paid for rooms already booked during this period have not changed

```
SELECT * FROM room_booking;
```

#showing the new dynamic pricing range

```
SELECT room_id, room_type_id, accom_id, price_per_night, available_from_date, available_to_date FROM room

WHERE room_type_id = @room_type_id AND accom_id = @accom_id;
```

#showing the uprated price for booking within the low availability period

```
SELECT room_id, room_type_id, accom_id, price_per_night, available_from_date, available_to_date FROM room

WHERE room_type_id = @room_type_id AND accom_id = @accom_id AND ((@check_in_date BETWEEN available_from_date AND
available_to_date) OR (@check_out_date BETWEEN available_from_date AND available_to_date));
```