# Database Design Report - Booking.com Booking System

## Background

The aim of this project was to replicate (part of) the functionality of the booking.com system. That is, to investigate the system and reverse engineer an appropriate relational database, that would facilitate the completion of a customer's accommodation booking experience at the front end. The database was to be designed in coherence with basic data principles such as Normal form, in order to reduce the need for restructuring of the relations upon appendment of new data and free the data from unwanted CRUD dependencies. Based on the desired functionality of completing an accommodation booking, the database design is required to facilitate a number of crucial functions. Firstly, a user will search for a particular destination, for a defined period and travel party. It is Immediately obvious that this requires several points of data to be recorded in order to return the relevant search results and make a reservation. The various destinations (countries and cities etc) must be stored, accommodations available in that area and their respective relevant information, the ability to reserve an accommodation and complete the booking/payment and lastly the customers information in order to assign, and record, a booking to the relevant customer. Other intricacies of the system (such as a dynamic pricing model) revealed themselves throughout the project and influenced the final database design. Further background to each of these developments has been provided in the relevant areas as this report progressed.

## Entity Discovery

Initially the investigation was carried out by small groups. Individually we carried out our own analysis of the system and then came together to refine our results as a collective. We extracted that which we deemed to be relevant, and within scope, and then developed early models of the entities, respective attributes and any relationships evident. As the project progressed, we shifted from a collaborative group approach to individual development of our designs.

| Destination | Dates | Travel party/ passengers/rooms | Filter options | Accessibility |
|---|---|---|---|---|
| city, | check-in | no of adults | Home apartment? | options>property & Room |
| region | check-out | no of children | Work? | |
| country | | Age | | |
| | | no Rooms | | |
| Accomodation | images rooms/whole accom | Price | Filter options | Room Types |
| name | review rating- out of 10 - customer view? | Curreny | Budget | name |
| Address | preferred plus property Maybe? | amount | property rating | description |
| location | property rating | taxes | activites | cancellation policy |
| longitude | cancellation policy | charges | cancellation policy | food options |
| latitude | description | city tax | Property Type | |
| amenities rooms,bathrooms | board | cancellation charge/fee | | |
| | | | | |
| Landmarks- | Accomodation Facilites | Room facilites | location | Account |
| AKA Hotel surroundings | category/type | name | city | discounts? |
| landmark type | type | category/type | region | |
| distance | name | description | country | |
| Landmark name | description | | long/lat | |
| Landmark category | additional charge boolean? | | | |
| | | | | |
| Payment Table | Reviews | FAQ's | Housse Rules | Guest table |
| Method card/ paypal/ googlepay | guest reviews | Asked | check in time | name |
| | Accomodation ID | Answered | check out time | email |
| Card Payment | reviewer name | Images | prepayment? boolean? | special requests |
| name | reviewer location Review Description | | Curfew | estimated arrival time |
| Card no | Review Date | | Pets | cot bool |
| expiry | review desc | | ... | country/region |
| | Pros | | Fine print | mobile number |
| | Cons | | | paperless - via the app |
| | categories | | | save Details? |
| Promo code | Recent searches | | | Special requests |
| Discount amount | IP address | | | |
| Discount code | Recent search history | | | |
| Eligibility?? | Keywords | | | |
| | | | | |

*Figure 1- Entity discovery*

Once we had narrowed down those entities which were most relevant to the key narrative of the project, we set out to describe these with appropriate attributes and define the relationships between them, whilst normalising the data to reduce data redundancy and simplify the query process.

Each table was given a primary key in order to adhere to the principle of entity integrity and ensure that each entry is uniquely identifiable. I decided against the use of candidate and composite keys as primary keys and opted for surrogate keys instead. There are several advantages to using surrogate keys. Firstly, as they are independent of the source data, they greatly simplify the data integration process. Secondly, as they are system generated uniqueness is enforced. However, they do limit the flexibility and interoperability of the system and, as they do not convey any information about the data or business logic, it can be more difficult to understand and document the schema.

Some obvious relationships were easy to define e.g an accommodation can have many rooms and therefore an accommodation table, linked to a room table via a one-to-many cardinality, was appropriate (accomodation_id as a foreign key in the room table). Similarly a customer can have many bookings and therefore the customer_id has been used as a FK in the booking table. Other relationships proved more complex upon closer inspection, such as the facilities of a given room. It was clear that room facilities should be normalised and each linked to a particular room, however each room could have multiple facilities. Using room_id (the primary key of the room table) as a foreign constraint in the room facility table would achieve this . However, the facilities proved to be recurrent across different rooms (and accommodations). Therefore, we required the ability to also link a facility to several rooms (to reduce data redundancy and duplicate entries) i.e. a many – to – many cardinality. In the final database design this was achieved using two one-to-many relationships and an 'in between' table (there are several instances of this type of relationship throughout the design - see figure 3).

## Assumptions

Several assumptions were made at various stages of the project. Firstly, as a group it was deemed that 3rd party services and additional functionality (e.g flight and vehicle booking) were deemed out of scope, as the focus of the project was to represent a complete accommodation booking.

Apartments and other like accommodation types are assumed to be treated as one room, as a reservation books the entire apartment/property, and it is priced and described accordingly.

Throughout the database there are several points at which an external API would carry out additional functions/calculations. For example, the data base does not store a list of currency conversion rates, this would be carried out at front end. Similarly, the database stores the distance between accommodations and landmarks/POIS (and their coordinates), the actual great circle distance itself would be calculated outside the database with a programming language. User review scores are averaged and displayed alongside the relevant property. As these are updated almost constantly this will be assumed to be calculated in real time at front end. The generation of unique data points such as booking and transaction numbers will be handled by an external API.

A google search reveals that the longest recorded address line is currently 147 characters long, therefore address related fields have been stores as VARCHAR (255). Likewise the maximum image url for an xml sitemap is 2448. Therefore, image url has been described as VARCHAR 2550 in length.

Where financial implications are associated with an attribute (e.g price) I have stored it as a decimal to ensure high level precision during aggregation.

Where certain attributes are optional (e.g in the passport section of an account) these fields have been set to have a default of null in order to allow those attributes to be added on an ad hoc basis and not prevent the record from being inserted. Conversely, where the data for a particular attribute is crucial the default is not null. Furthermore, where an attribute would likely be distinct e.g booking number / payment number for a customer's reference, these have been enforced as unique.

## Design

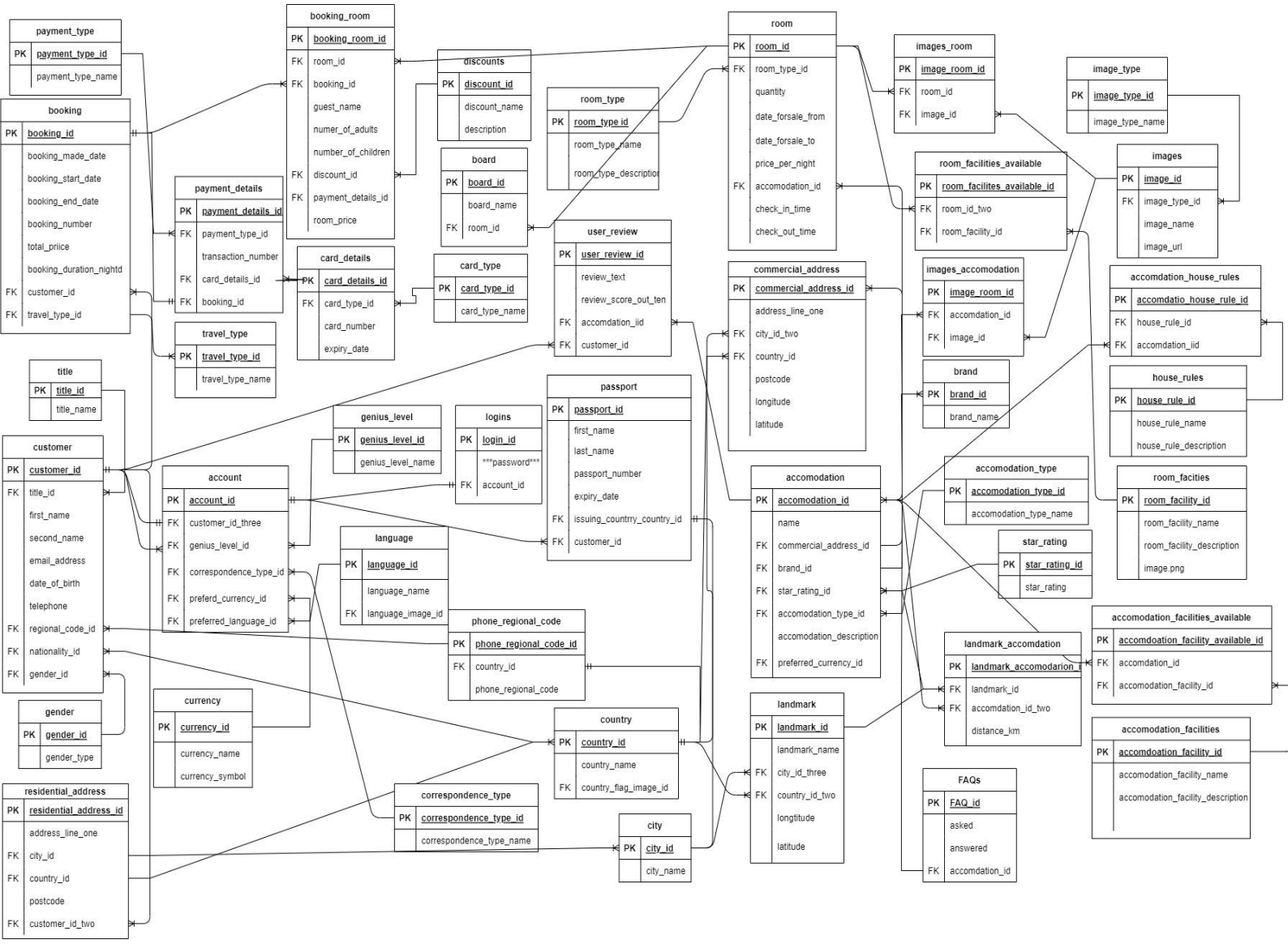


*Figure 2- Initial Database ERD*

Figure 2 shows my initial, individual, ERD diagram. Several issues regarding the readability of the diagram led me to introduce 2 main changes. Firstly, I rearranged the attributes in each table such that all foreign (and primary) keys where at the top of each table (some attributes were also rearranged for further basic readability e.g ensuring price next to currency_id). Secondly, switching to angular links within phpMyAdmin decreased the convolution considerably (see figure 3). Generally, I have kept the naming convention of FKs consistent with their primary key counterpart. However, where that may have impacted the meaningfulness of the attribute, relevant to the context, I have given it a suitable name.

This is perhaps best demonstrated in the account table where the country_id primary key is linked as a foreign constraint to several different attributes and as such, they have been given distinct, more

meaningful, names. Snake case has also been used consistently throughout the database (both in table names and attribute names), this is my personal preference, partially due to the fact that SQL is case insensitive and the distinction between words, in Pascal or Camel case, is therefore easily lost.

My very first design had a single customer table that stored all the key details (which was consistent with the group approach).In subsequent iterations an account table was introduced in such a way that a customer could also have an associated account (as in figure 2). However, upon further inspection it became clear that this configuration was missing a large aspect of the functionality of the booking system.Booking.com allows customers to complete a booking as both registered and unregistered users. Depending on which of these options are exercised, differing data is recorded for the respective customer. For this reason, I have redesigned both the customer and account tables. The customer table is reserved for those who have completed a booking (and thus become a customer, note this includes account holders). The account table is reserved for those who create a user account. To facilitate the relationship between the two, I have linked the tables via a FK. Note, that I have included the account number in the customer table and not vice versa. This was a deliberate design decision as an account can be created without a booking being made and therefore a customer Id being generated (this will most likely reduce null entries).

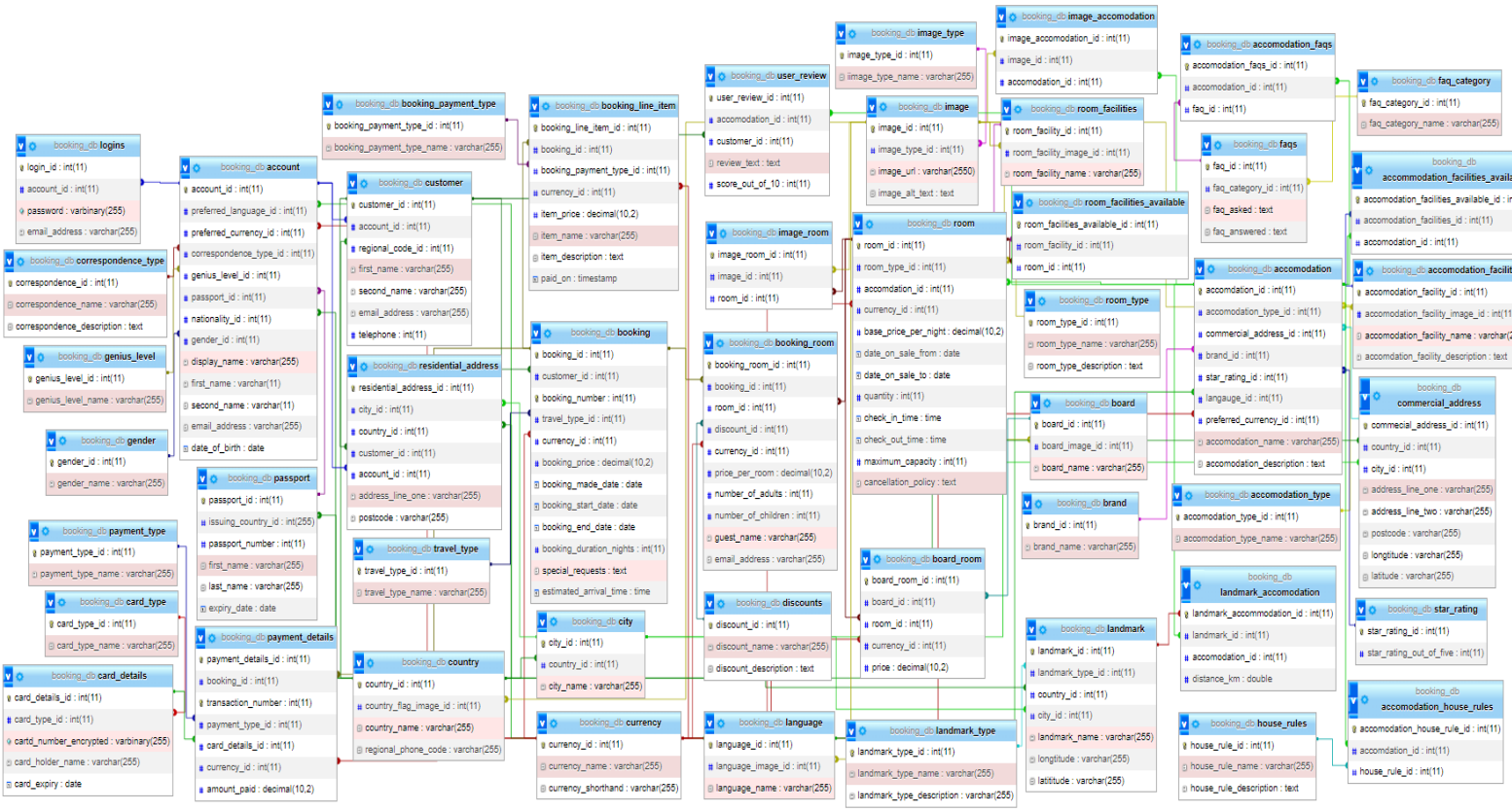


*Figure 3- Final Database design*

At first glance the duplication of certain attributes (such as first name and second name etc) in these tables may appear to be data redundancy, however they are required to account for both scenarios mentioned above. To reduce any potential data duplication the default behaviour could be to take these data points, for the customer table, from the account details where the customer is a registered account holder. However, booking.com does actually ask for some of these details to be re-entered at the final booking stage (first name, second name, phone number and email address) regardless of whether you are an account holder or not. Presumably, this is to allow final dictation in contact details to which the booking confirmation will be sent. The default behaviour mentioned

would be a developer decision, but the ability to store details in both scenarios has been implemented.

The account table stores multiple additional data points that incorporates several features of a registered account. One of these features is a preferred language. I have included a normalised language table which is linked to the preference of the account holder via a FK. This is particularly useful, for example, in the scenario where properties descriptions are in a different language (note preferred language is also an attribute in the accommodation table). At this point an external api would be needed to carry out a live translation into the account holders preferred language. An account can also be linked to a passport, I have linked passports to a particular account to facilitate multiple passport holders (as opposed to a single passport linked to an account via FK). Booking.com does not seem to currently offer the option to store multiple passports for an account, however this configuration allows for the potential feature to be added in the future without a schema adjustment whilst also encapsulating the current functionality.

I have chosen to include an independent, normalised discount table for the genius discounts offered to registered account holders as these are highly repetitive and enumerable. Furthermore, genius discounts are available at a room level and therefore it seemed more appropriate to have these types of discounts normalised and linked to the room table via a FK. More bespoke discounts that are property dependent (for example bluecard for essential workers, or corporate discounts etc) could be solely added as line items in the booking line-item table (all discounts, including genius, will still be present for invoicing/audit purposes). This also allows discounts to be traceable to the relevant room in the event that multiple rooms, for different guests, are booked (which allows them to accurately calculate their individual bill - note this is the same principal that I have used regarding a separate board table).

In many instances the rooms of an accommodation have several different board (meal) options. As such I have designed a many to many relationship between the board and room tables, which is facilitated by an in between table called board_room. I have chosen to price the board items at the board_room level in order to reduce data redundancy, as the prices for the same board appear to be accommodation/room dependent (if this was priced at board level the same meals would need multiple entries at different prices).

A small change regarding the country table between my initial and final design, is that the regional phone code has been added as an attribute, rather than being normalised as its own table. This is because each country only has one regional code and as it is a singular data point normalisation is unnecessary, however a country may have several languages or currencies and as such these have remained as their own normalised entities.

Investigation of the booking.com pricing, at various stages revealed a dynamic pricing model. Live prices are subject to change depending on several factors, at the accommodations (or potentially booking.com's), discretion. For example, allocation (the percentage of rooms remaining of the original allocation) impacts the live price, as does the availability (dates of a reservation). However, this variability in live price does not impact the amount paid for a booking for a particular customer at a different time. As such the database has several price related attributes across multiple tables to store the relevant pricing, ensure independent data recording and also the ability to retain individual pricing and calculate totals (as booking.com allows for multiple rooms to be booked under one booking). At the room level price has been described by the attribute base_price_per_night. This is the live/current price displayed at front end that a user would see. This price can be altered based on the allocation of rooms or other bespoke factors without impacting the other price attributes of previous records. Furthermore, the room table has been structured in such a way that pricing based on dates can also be incorporated. This is facilitated by the attributes of date_on_sale_from and date_on_sale_to which enables rooms to be re-entered with an adjusted price for a different period.

Rather than record each individual room of an accommodation as a separate entry in the room table, I have opted for a quantity attribute. This does remove the ability to assign a room number (or particular room) to a customer at the booking stage, however assignment would generally be handled internally by the accommodation, booking.com merely has a number of rooms to sell. Having a quantity attribute greatly reduces the number of data entries and computational overhead. It also allows for simpler aggregation and calculations as the number of rooms available for that period (of that type) is stated explicitly (as opposed to convoluted SUM and COUNT queries) which for example can then be compared to the total number of rooms, of the same criteria, already booked.

As previously mentioned booking.com facilitates multiple room reservations under one booking. Furthermore it allows for several data points to be specified for each room (e.g. email address and guest name), however the duration of the stay is not customisable at room level and therefore is an attribute of the booking table. The booking_room table has been designed and implemented to enable this level of granularity. The table is linked to both the room table and the booking table via the respective FKs (booking id and room id), this allows for multiple rooms to be linked to a single booking.

The live price is extracted from the room table, used to calculate the total cost for the room over the course of the booking period, and stored in the booking_room.price_per_room attribute. This table provides a snapshot of the price for a specific room for the total duration of a booking. The relevant entries in this table then constitute the total base price (without additional charges – which are recorded in the booking line-item table) for a particular booking. Within the booking table then, there is a booking_price. This attribute derives its value from the sum of the price per room for all rooms booked, as well as any additional charges as invoiced in the booking_line_item table (which has been implemented for this purpose).

Booking.com shows the maximum number of people that can inhabit a room (and then the type of bed for that room )and as such I have included a max capacity attribute in the room table. This also facilitates easier querying and extraction of like (and relevant) rooms. For example, there are several different types of double rooms (e.g deluxe double, standard double, double suite, twin room, 1 bed apartment etc) the high degree of variation in the names of these rooms makes for a difficult search based on room names. Having a max capacity however means that returning the relevant results based on a customer's search for 2 people, for example, becomes infinitely simpler.

Further consideration of the real world functionality and application of the city table lead me to realise that cities are little use without their broader context (i.e their respective country). As a result another change from the initial design was to add the country id to the city table. This is to facilitate the ability to extract all cities of a given country (or the country of any given city) stored in the database. With the previous configuration there was a considerable dependency built into the relationship (particularly in the case of deletion).Although it was possible to retrieve a city and its country, by joining the city table onto one of the address tables, there is the fringe case where a city may be present in the database but does not currently have an address associated with it (or possibly these records have been deleted). In this case there would be no way to confirm the associated country which would impact both the search functionality and future inserts.

The title table has also been removed entirely, as title does not appear to be stored anywhere for a customer/account holder.

Upon closer inspection Booking.com has both website/service specific FAQs and accommodation specific FAQs. As a result, FAQs have been redesigned and normalised to account for FAQs at both levels. I have included a normalised FAQ type table and linked this via FK to the FAQ table to make the different types of FAQs differentiable. Furthermore, I have implemented an accommodation

specific FAQ table to reduce data redundancy in the scenario where an accommodation specific FAQ can be generalised to several properties (e.g properties that have the same COVID policy).

Commercial addresses and residential addresses have been treated as two different data sets. Several factors influenced this decision. First and foremost, the scenario of having multiple customers residing at one address. Should the address table be shared between both customers and accommodation, this would prevent the customer Id from being a valid (appropriate) attribute within the address table. The alternative is to link the address to the customer, via the accommodation id as a FK in the customer table. The primary issue with this approach, is that should one resident need to change their address, this will affect the other resident's address. This could be avoided by controlling data entry and trying to enforce duplication of addresses (i.e an address entry for every customer) however this still leaves room for error. Conversely, using the customer id as a FK within the address table ensures this principle of duplication is adhered to. Secondly, booking.com displays the accommodation distance from landmarks, as well as exact geolocations on maps, and as such gps coordinates for each property must be stored. There is clearly no such need for residential addresses and as such to store this data would be redundant. Hence the two address types should be treated independently. The account id has also been linked to the residential address table as account holders are required to enter their address details during registration, this is then stored and accessed at booking time. Unregistered customers however must enter their address at the booking stage and as such both customer id and account id must be linked to an address. This configuration will lead to null entries (for at least account id where an unregistered customer books) however it seems to be the most effective implementation to include the desired functionality .

As mentioned previously booking.com shows the distance between an accommodation and various landmarks/points of interest. Therefore, there must be a list of landmarks and their coordinates stored. I have designed a landmark table which has a normalised landmark type table linked to it via FK. This allows me to differentiate between various types of points of interest e.g. tourist attractions, stadiums etc and incorporate the filter functionality on booking.com. Furthermore, distance could be calculated in real time, and not stored, at front end using the GPS coordinates of the two entities. However, as these are fixed points they likely would not require recalculation after the initial calculation (as opposed to something like the average user review score) therefore I have included a distance attribute in the landmark table to store the distances after initial calculation. The landmarks and accommodations are linked via a many-to-many cardinality (see landmark_accomodation table).

Cancellation policies appear to be customisable at a room level on booking.com and as such I have included a cancellation policy attribute in the room table, note I have not normalised cancellations as although some policies are recurrent (e.g no cancellation) across properties/rooms, there appears to be a large degree of variation and property dependencies. A more generalised cancellation policy is included in the house rules section and can be included in the accommodation house rules table. I have also included check in and check out times at a room level as some accommodations may offer flexible check-in/ check out times depending on the room. This configuration allows for more control and flexibility. The house- rules will then contain the general check-in /checkout times for a property. Again a default behaviour of taking these details from the generic property check-in times could easily be implemented.

Images of both an accommodation and its rooms are displayed on page. Furthermore, there are several other types of images that are recurrent throughout the webpage. For example, icons of accommodation and room facilities, country flags (when selecting a language) and board icons etc. Therefore, a normalised image table has been created to store all images related to the various entities of the database. The attributes of this table include the image url and an alternative identifier which describes the actual image for easy identification. It has been further normalised to include an FK to an image_type table, these image types have been used when creating other tables to facilitate many to many relationships, or linked directly via a FK. As previously mentioned, there

are several many to many relationships within this database design so that each entity can be linked to multiple other instances of the other. In the case of the images for accommodation, an image may need to be linked to several different accommodations (say in the event of a chain of hotels – each distinct hotel of the brand could use the same logo image) and several images in turn linked to a particular accommodation (an image of reception, the building front, various facilities etc).This also applies to the image_room table which facilitates the same relationship, as different types of rooms appear to reuse stock images and the image itself has a link to the particular room (bottom left hand corner of image).

Each user review has been linked to the relevant accommodation and customer id (which can then be joined onto the relevant booking ) for audit and validation purposes, the actual logic and check of this would be completed in an external api outside the database, but this configuration facilitates the functionality of validating a customers stay before adding their review.

**Protection and Encryption of Data**

There are a few instances of sensitive data being stored within the database. Firstly, regarding the recording of payments, there is a (normalised) payment details table that is linked to several others via FK. Each payment in the table is linked to its respective booking via a FK, namely the primary key of the booking table (booking_id). This configuration (as opposed to linking a booking to a payment via payment id) allows precise retention of relevant payment details for a customer and any payment(s) they have made (as opposed to just 1 singular payment). This could also support the introduction of split payments without a schema change (e.g where multiple rooms have been booked for different guests under one booking). I have normalised the card details for each payment (and further normalised the card type) and decided that only the card number need be encrypted as the other details, such as expiry, are inconsequential without it. The card holders name would also theoretically be extractable through the link to the booking and then the customer table, and as such has not been encrypted either. The card number has been encrypted using SQLs native Advanced Encryption Standard (AES). This is a FIPS approved cryptographic (symmetric block cipher) algorithm which can encrypt and decrypt data. This differs from the password encryption, which is an example of one-way encryption, as the card details will be needed to be accessed to process payments and, as such, we must be able to convert the card number back to plain text. The simplistic algebraic structure of AES and the fact that every block is always encrypted in the same manor, renders it liable to side-channel attacks and inappropriate for highly sensitive data. Therefore, in a real live database, card details would typically be handled by a third party such as stripe or at least encrypted outside the database using a language such as PHP. As payment is a crucial part of the booking process, for the sake of completeness, the storage and encryption of these details has been included in the scope of the project . The AES_ENCRYPT() function takes a string and, using a predefined secret password key, returns a binary string, hence the card number data type has been set as a VARBINARY (255). The secret key is then called in conjunction with the AES_DECRYPT() function to decrypt the card number into its original plain text.

As there is an account functionality for Booking.com login details must be stored in order to validate a particular account holders' access. I have chosen not to store the actual login attempt details, as such a degree of audit and compliance is perhaps out of scope (and massively memory intensive) and it is unclear if this is even implemented in booking.com. Instead, only the registered login details at signup are stored in the normalised logins table. This table contains only the critical information required for a user's login, namely the email address they used at sign up (note this is independent of the preferred email address stored at account level), alongside their (encrypted) password. Each set of details is then linked via FK (account_id) to the respective account. There are several encryption techniques native to SQL such as MD5 (message-digest algorithm), SHA-1 (Secure hash algorithm) which are relatively safe against preimage attacks, however as these algorithms encrypt an input in the same way, to a fixed-length output, each time, they are easily broken via brute force

or dictionary attack. A simple google search will reveal several websites such as hashtoolKit.com that can be used to decrypt these using rainbow tables. There is also a native function called PASSWORD() which returns a hashed string, however despite the name, this function is not suitable for password storing and has actually been deprecated, due to the misleading nature of the function name. In a real world database encryption of such sensitive information would be done externally. One such long standing reputable solution is bcrypt. bcrypt is a password hashing function, which first salts (adds additional random data) and then hashes the concatenated input, which is deliberately slow. Since honest systems tend to use off the shelf hardware, which are also available to the attacker, the goal is to make password hashing times slower for both the attacker and for the database owner, to the highest possible threshold without exceeding available resources. I have replicated a simplified version of this natively within SQL by generating a random string of length 6 (the salt), encrypting that using SHA-1, concatenating the salt onto the user password and then hashing the result.

**Improvements**

There are several design improvements that could be implemented in this system. Firstly, the review table could be normalised further to include sub scores for each facility/aspect of the accommodation as well as pros and cons. The scoring system could also be normalised to ensure only valid entries from 1 to 10 are recorded. Sustainability has been purposefully omitted, however it could be normalised and implemented at the accommodation level via FK, as could accessibility. Accessibility, however, would likely be needed at an accommodation and room level, and require a many to many relationship for both (as accommodations/rooms could be linked to several accessibility policies/facilities and vice versa). Booking.com shows several spoken languages at the accommodation not just a preference. As such this could have been recorded in a many to many relationship as opposed to a singular language being linked by FK. Currently the house rules table has a name, a description and contains all house rules for all properties. The current configuration is perhaps liable to inefficiencies (for example varying check-in times would each need their own entry). A potential improvement would be to have generic descriptions for each house rule and then include a 'house-rule specifics' attribute in the accommodation_house_rules table (this would reduce the check in- check out entries example to a single entry in the house rules, with the specific times being specified in the table aforementioned). Another potential improvement would be to have an additional attribute in the booking table that would keep track of the outstanding balance. This would be easily implementable with the current credit/debit attribute indicating if an amount is to be added or deducted from a bill. Currently the booking table simply states the total at the time of booking confirmation and the line-item table can then be used for invoicing purposes, which could ultimately be used to derive an outstanding balance calculation in real-time. Similarly, current status of booking could be implemented as an attribute of the booking table.

There are a few additional functionalities of a registered account that have been omitted such as preferences like card details. This would be relatively easy to add into the current system. Linking an account to an entry in the card details table via FK (account_id) would facilitate the relationship but restrict each registered card to only one account (there does not seem to be any such restriction on booking.com). For that reason, the reverse implementation (card details id as a FK in account) would be more suitable, describing the required relationship as before, whilst also incorporating the ability to link a card to multiple accounts. Profile pictures are also stored for account holders. Again, this could be easily added to the current database by adding a 'profile_picture' entry to the image_type table and then storing all those images in the image table. Profile pictures would likely be unique (and any repetitions infrequent) and therefore not require the ability to assign one image to several accounts, a simple FK to link the two would suffice in this case.

Booking.com does seem to retain search history and then provide suggestions accordingly (particularly with previously selected filters). Initial searches could be stored, linked to a

customer/account and then used to tailor future filter results to provide a more personalised experience.

**Conclusion**

In conclusion, this database design successfully facilitates the completion of a user booking as described in the introduction. Whilst some additional features have been omitted, the majority of impactful filter options and features have been implemented and justified. The database is consistent with Chen's and Codd's principles, accurately describing a real-life scenario whilst being free of significant dependencies. With the adjustments outlined in the improvements section, I believe this would be a complete representation of the full accommodation booking process.

## Appendix

### Queries used in video

**1) show ability to filter language and currency**

```
Select * FROM language;
Select * FROM currency;
```

**2)create account with log in credentials and address**

```
START TRANSACTION;

SET @DOB='1995-04-03';
SET @first_nm ='Andrew';
SET @second_nm='Morrison';

SET @nationality = 142;
SET @gender = 1;
SET @email_add = 'ajmorrison@hotmail.com';
SET @genius = 1;
SET @currency = 1;
SET @language = 1;
SET @display = 'AndrewJM';
SET @correspondence=1;

SET @addressline = '3 Thornbush road';
SET @postcode='BT5 64N';
SET @city= 8;
SET @country=142;
SET @enteredPassword = 'samplePassword5';

INSERT INTO account
(account_id,preferred_language_id,preferred_currency_id,correspondence_type_id,genius_level_id,passport_id,nationality_id,gender_id,display_name,first_name,second_name,email_address,date_of_birth) VALUES
(NULL,@language,@currency,@correspondence,@genius,NULL,@nationality,@gender,@display,@first_nm,@second_nm,@email_add,@DOB);

SET @account_id = LAST_INSERT_ID();

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

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

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

INSERT INTO logins (login_id,account_id,password,email_address) VALUES
(NULL,@account_id,@saltedPassword,@email_add);

INSERT INTO
residential_address(residential_address_id,city_id,country_id,customer_id,account_id,address_line_one,postcode) VALUES (NULL,@city,@country,NULL,@account_id,@addressline,@postcode);

COMMIT;
```

**3) Confirm updated tables and check login**

```
Select* FROM account;
Select* FROM logins;
```

```
Select* FROM residential_address;

#Get the password attempt entered by the user as LOGIN

SET @loginPassword = 'samplePassword5';
SET @loginEmail = 'ajmorrison@hotmail.com' COLLATE utf8mb4_general_ci;

#Get the salt which is stored in clear text

SELECT @saltInUse := SUBSTRING(password, 1, 6) FROM logins WHERE email_address = @loginEmail;

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

SELECT @storedSaltedHashInUse := SUBSTRING(password, 7, 40)FROM logins WHERE email_address =
@loginEmail;
#Concat our salt in user and our login password attempt.
SELECT @saltedHash := SHA1(CONCAT(@saltInUse,@loginPassword)) AS salted_hash_value_login;
```

**4)show all rooms and filter search**

```
select* from room;

#search for all available rooms during given dates

SELECT * FROM city INNER JOIN commercial_address ON city.city_id = commercial_address.city_id INNER JOIN
accomodation ON commercial_address.commercial_address_id=accomodation.commercial_address_id INNER
JOIN room ON accomodation.accomodation_id=room.accomodation_id WHERE city_name IN ('LONDON') AND
date_on_sale_from <= '2023-11-06' AND date_on_sale_to >= '2023-11-09' AND maximum_capacity =2 ORDER BY
base_price_per_night ASC;

# narrow down to  4 star minimum

SELECT * FROM accomodation LEFT JOIN star_rating ON accomodation.star_rating_id=star_rating.star_rating_id
WHERE accomodation_id in (1,2,5,6,7) AND star_rating in ('four star','five star');

# narrow down to hilton brand and compare accomodation facilities
#only hotels

SELECT * FROM accomodation LEFT JOIN accomodation_type ON accomodation.accomodation_type_id=
accomodation_type.accomodation_type_id WHERE accomodation_id in (1,2,6,7) AND accomodation_type_name
Like 'hotels';

#only hilton (2 valid hotels now)

 SELECT * FROM accomodation LEFT JOIN brand ON accomodation.brand_id= brand.brand_id WHERE
accomodation_id IN (1,6,7) AND brand.brand_name LIKE 'Hilton%';
```

**5)Compare accommodation facilities and reviews**

```
#compare accom facilities

SELECT
accomodation_name,accomodation_description,accomodation_facility_name,accomodation_facility_description,
accomodation_facility_image_id FROM accomodation LEFT JOIN accommodation_facilities_available on
accomodation.accomodation_id=accommodation_facilities_available.accomodation_id LEFT JOIN
accomodation_facilities on
accommodation_facilities_available.accomodation_facilities_id=accomodation_facilities.accomodation_facility_i
d WHERE accomodation.accomodation_id IN (1,6);

#check accoomdation photos

SELECT accomodation_name,image.image_id,image_url,image_alt_text FROM accomodation LEFT JOIN
image_accomodation on accomodation.accomodation_id=image_accomodation.accomodation_id LEFT JOIN
```

*image on image_accomodation.image_id = image.image_id WHERE accomodation.accomodation_id IN (1,6);*

*#compare reviews for accoms*

*SELECT first_name,second_name,review_text,score_out_of_10,accomodation_name FROM customer Right Join user_review on customer.customer_id=user_review.customer_id INNER JOIN accomodation on user_review.accomodation_id=accomodation.accomodation_id WHERE accomodation.accomodation_id IN (1,6);*

**6)filter landmarks check house rules and FAQS for wembley.**

*#Filter distance to london eye*

*SELECT landmark_name,accomodation_name,distance_km,landmark_type_name FROM accomodation INNER JOIN `landmark_accomodation` ON accomodation.accomodation_id= landmark_accomodation.accomodation_id LEFT JOIN landmark on landmark_accomodation.landmark_id=landmark.landmark_id LEFT JOIN landmark_type ON landmark.landmark_type_id= landmark_type.landmark_type_id WHERE accomodation.accomodation_id in (1,6) AND landmark_name IN ('Tower of LONDON','LONDON EYE');*

*#Nearby attractions*

*SELECT landmark_name,accomodation_name,distance_km,landmark_type_name FROM accomodation INNER JOIN `landmark_accomodation` ON accomodation.accomodation_id= landmark_accomodation.accomodation_id LEFT JOIN landmark on landmark_accomodation.landmark_id=landmark.landmark_id LEFT JOIN landmark_type ON landmark.landmark_type_id= landmark_type.landmark_type_id WHERE accomodation.accomodation_id in (1,6);*

*#FAQS*

*SELECT accomodation_name,faq_asked,faq_answered FROM accomodation LEFT JOIN accomodation_faqs on accomodation.accomodation_id=accomodation_faqs.accomodation_id LEFT JOIN faqs ON accomodation_faqs.faq_id=faqs.faq_id WHERE accomodation.accomodation_id =1;*

*#house rules*

*SELECT house_rule_name,house_rule_description,accomodation_name FROM house_rules INNER JOIN accomodation_house_rules ON house_rules.house_rule_id=accomodation_house_rules.house_rule_id LEFT JOIN accomodation on accomodation_house_rules.accomdation_id=accomodation.accomodation_id WHERE accomodation.accomodation_id = 1*

**7)compare rooms in hilton wembley**

*SELECT accomodation.accomodation_name,room_id,base_price_per_night,date_on_sale_from,date_on_sale_to,room_type_name,cancellation_policy FROM city INNER JOIN commercial_address ON city.city_id = commercial_address.city_id INNER JOIN accomodation ON commercial_address.commercial_address_id=accomodation.commercial_address_id INNER JOIN room ON accomodation.accomodation_id=room.accomodation_id INNER JOIN room_type on room.room_type_id=room_type.room_type_id WHERE city_name IN ('LONDON') AND date_on_sale_from <= '2023-11-06' AND date_on_sale_to >= '2023-11-09' AND maximum_capacity =2 AND accomodation.accomodation_id =1 ORDER BY base_price_per_night ASC;*

*#room_ids available during this period are 1 and 4 select images for rooms*

*Select room_type_name,room.room_id,image_url,image_alt_text FROM room_type INNER JOIN room on room_type.room_type_id=room.room_type_id INNER JOIN image_room on room.room_id= image_room.room_id LEFT JOIN image ON image_room.image_id = image.image_id WHERE accomodation_id = 1 AND room.room_id in (1,4);*

*#compare room facilities*

*select image_url,image_alt_text,room_facility_name,accomodation_id,room.room_id from image Right Join room_facilities on image.image_id=room_facilities.room_facility_image_id Right JOIN room_facilities_available*

*on room_facilities.room_facility_id = room_facilities_available.room_facility_id LEFT JOIN room on room_facilities_available.room_id=room.room_id WHERE room.room_id in (1,4) ORDER BY room.room_id;*

*#Check breakfast options*

*SELECT room.room_id,board_room.currency_id,price,board_name,board_image_id FROM room INNER JOIN board_room ON room.room_id = board_room.room_id LEFT JOIN board on board_room.board_id=board.board_id WHERE room.room_id in (1,4);*

**8)check allocation and update price**

*#Show Original allocation of double rooms in this date range*

*SELECT city_name,accomodation_name,room_type_name,date_on_sale_from,date_on_sale_to,quantity preferred_currency_id,base_price_per_night FROM city INNER JOIN commercial_address ON city.city_id = commercial_address.city_id INNER JOIN accomodation ON commercial_address.commercial_address_id=accomodation.commercial_address_id INNER JOIN room ON accomodation.accomodation_id=room.accomodation_id INNER JOIN room_type ON room.room_type_id=room_type.room_type_id WHERE city_name IN ('LONDON') AND date_on_sale_from <= '2023-11-01' AND date_on_sale_to >= 2023-11-03 AND maximum_capacity =2 AND accomodation.accomodation_id = 1 ORDER BY base_price_per_night ASC;*

*#check for bookings in booking table*

*SELECT COUNT(room.room_id) AS total_rooms_already_booked, booking_room.booking_id,booking_start_date,booking_end_date,booking_duration_nights,room.room_id FROM booking INNER JOIN booking_room on booking.booking_id = booking_room.booking_id LEFT JOIN room on booking_room.room_id=room.room_id GROUP BY room.room_id HAVING booking_start_date <= '2023-11-06' AND booking_end_date >= '2023-11-09';*

*#update price*

*UPDATE `room` SET `base_price_per_night` = '185' WHERE `room`.`room_id` = 4;*

**9) run booking transaction**

*START TRANSACTION;*

*SET @first_nm='Andrew';*
*SET @second_nm='Morrison';*
*SET @account_id = 22;*
*SET @phone_code = 142;*
*SET @email_add = 'ajmorrison@hotmail.com';*
*SET @telephone = 77747573;*
*SET @booking_no= 221914;*
*SET @s_requests = 'Wake up call at 06:00';*
*SET @travel = 2;*
*SET @currency=1;*
*SET @booking_made='2023-11-04';*
*SET @booking_start='2023-11-06';*
*SET @booking_end='2023-11-09';*
*SET @booking_duration = 3;*
*SET @arrival_time = '15:00:00';*
*SET @roomid1=4;*
*SET @roomid2=1;*
*SET @adults=2;*
*SET @children=0;*

*INSERT INTO customer (customer_id,account_id,regional_code_id,first_name,second_name,email_address,telephone) VALUES (NULL,NULL,@phone_code,@first_nm,@second_nm,@email_add,@telephone);*

```
SET @cust_id = LAST_INSERT_ID();


INSERT INTO booking
(booking_id,customer_id,booking_number,travel_type_id,currency_id,booking_price,booking_made_date,bookin
g_start_date,booking_end_date,booking_duration_nights,special_requests,estimated_arrival_time) VALUES
(NULL,@cust_id,@booking_no,@travel,@currency,0.00,@booking_made,@booking_start,@booking_end,@book
ing_duration,@s_requests,@arrival_time);

SET @booking_id = LAST_INSERT_ID();

SELECT base_price_per_night into @base_price1 FROM room where room_id=@roomid1;
SET @room_price1 = @booking_duration *@base_price1;

SELECT base_price_per_night into @base_price2 FROM room where room_id=@roomid2;
SET @room_price2 = @booking_duration *@base_price2;

INSERT INTO booking_room
(booking_room_id,booking_id,room_id,discount_id,currency_id,price_per_room,number_of_adults,number_of_c
hildren,guest_name,email_address) VALUES
(NULL,@booking_id,@roomid1,NULL,@currency,@room_price1,@adults,@children,'Andrew
Morrison','ajmorrison@hotmail.com');

INSERT INTO booking_room
(booking_room_id,booking_id,room_id,discount_id,currency_id,price_per_room,number_of_adults,number_of_c
hildren,guest_name,email_address) VALUES
(NULL,@booking_id,@roomid2,NULL,@currency,@room_price2,@adults,@children,'Erin
Mehan','Emeehan@hotmail.com');

INSERT INTO booking_line_item
(booking_line_item_id,booking_id,booking_payment_type_id,currency_id,item_price,item_name,item_descriptio
n,paid_on) VALUES
(NULL,@booking_id, 2,1, @room_price1, 'room charge', 'Deluxe double room for 3 nights', '2023-11-04
09:13:34'),
(NULL,@booking_id, 2,1, @room_price2, 'room charge', 'Standard double room for 3 nights', '2023-11-04
09:13:34'),
(NULL,@booking_id, 2,1, 10.00,'Full English Breakfast', 'Deluxe double - add breakfast', '2023-11-04 09:13:34');

SELECT SUM(item_price) INTO @total_price FROM  booking_payment_type INNER JOIN booking_line_item ON
booking_payment_type.booking_payment_type_id=booking_line_item.booking_payment_type_id WHERE
booking_id =@booking_id;


SELECT @total_price;
#SELECT @total_credit;
#SELECT @total_debit;

UPDATE `booking` SET `booking_price`= @total_price WHERE `booking`.`booking_id`= @booking_id;

SET @cardholdername = 'Andrew Morrison';
SET @cardlongnumber = '12345677895434534';
SET @cardenddate= '2023-10-08';

SELECT card_type_id into @card_type FROM card_type WHERE card_type_name LIKE 'VISA%';


#A very simple secret password

SET @secretPasssword = 'thisIsTooEasyToGuess';

#Start (very) basic data encryption
```

```
SET @cardlongnumber = AES_ENCRYPT(@cardlongnumber,@secretPasssword);



#Insert the record with the encrypted data

INSERT INTO card_details (card_details_id,card_type_id,card_number_encrypted,card_holder_name,card_expiry)
VALUES (NULL, @card_Type,@cardlongnumber,@cardholdername,@cardenddate);


SET @card_details = LAST_INSERT_ID();
SET @trans_no=23243522;
SELECT payment_type_id into @payment_type FROM payment_type WHERE payment_type_name LIKE '%Card%';

 INSERT INTO
payment_details(payment_details_id,booking_id,transaction_number,payment_type_id,card_details_id,currency
_id,amount_paid) VALUES (NULL,@booking_id,@trans_no,@payment_type,@card_details,'1',@total_price);
#booking confirmation

 SELECT
booking_number,guest_name,room_type_name,number_of_adults,number_of_children,booking_price,booking.c
urrency_id,booking_made_date,booking_start_date,booking_duration_nights FROM `booking` RIGHT JOIN
booking_room on booking.booking_id=booking_room.booking_id LEFT JOIN room on room.room_id
=booking_room.room_id LEFT JOIN room_type on room.room_type_id=room_type.room_type_id WHERE
booking.booking_id = @booking_id;

 COMMIT;
```

**10)verify updates and demonstrate description ability for payment processing**

```
SELECT * FROM booking;
SELECT * FROM booking_room;
SELECT* FROM booking_line_item;
SELECT * FROM customer;
SELECT * FROM payment_details;
SELECT * FROM card_details;
SET @secretPassword = 'thisIsTooEasyToGuess';

SELECT
card_details_id,card_holder_name,card_expiry,card_type_id,AES_DECRYPT(card_number_encrypted,@secretPass
word) FROM card_details WHERE card_details_id =27;
```