1. Introduction

This report will show how a database was developed by reverse engineering a database based on the hotel booking enterprise *Hotels.com*. This database was developed by researching the *Hotels.com* website and creating a list of entities and their attributes through entity discovery. After deciding what should be in and out of scope, an entity relationship diagram was created to encapsulate the relationships between entities and all the entities that were to be used in the database. After multiple drafts, a final version of the entity relationship diagram was used to create the database for this project. After those steps, mock data was inserted to allow for a functioning data set to exemplify three main aspects of the database. These aspects include the hotels and rooms, the customers, and the steps of booking a room. These three main components were discovered during initial research, then developed through the design process. Before examining the design process, there will first be a discussion on the scope of the database and assumptions made.

1.1 Assumptions and Scope

Hotels.com has a diverse range of products and does not exclusively provide hotel booking. The time constraints for this project has restricted the scope and there are a number of fields that could not be included. Any products that were not hotels are out of scope. This includes the following sections found in *Hotels.com's* navigation bar – 'Holiday Lets', 'Things to do', 'Packages & Flights', 'Groups & Meetings', 'Gift Cards', 'List your property', and 'Hotels.com ® Rewards'. All of these sections were omitted from the final design due to the time constraints of this project. The process of booking a hotel is not dependent on any of these sections of the website as well as a customer can book a hotel without interacting with any of the sections mentioned above. Any calculations to show room availability and price surcharging will also be out of scope as these will likely be processed by a developer. The database will provide the relevant data to be used by a developer for both of these dynamic areas. These two areas will be discussed further below. Advanced techniques of data encryption will also be out of scope, with naive encryption techniques being used instead. In a real world enterprise, card payments would be processed by a third-party company to ensure PCI compliance is upheld. UK Sterling is the assumed currency as exchange rates would be handled by a developer. Alongside currency, the default language is assumed to be English for this database. After the majority of the decisions on the scope of the project and the assumptions that were made, the next step could take place. The next step was entity discovery, however, throughout this report questions of what should be in and out of scope, and assumptions were continually examined and evaluated. The construction of the database was dependent on an evolving design phase which will discussed below.

search_filter	location	hotel	hotel_amenity	hotel_accessibility	address_country	amenity	accessibility	hotel_rate	address	country	address_city	city
uniqueId	location_id	hotel_id	hotel_amenity_id	hotel_accessibility	address_country_id	amenity_id	accessibility_id	rate_id	address_id	country_id	address_city_id	city_id
location	location_name	hotel_name	hotel_id	hotel_id	address_id	amenity_name	accessibility_name	rate_id	building_number	country_name	address_id	city_name
date	location_geo	hotel_description	amenity_id		country_id	amenity_description	accessibility_description	rate_start_date	address_line_1	country_code	city_id	
landmarks	country_code	hotel_score						rate_end_date	address_line_2			
star_rating		rstar_rating							postcode			
guest_rating		photo_url							city_id			
price		wifi							country_id			
distance		Covid_19										
facilities		pets										
theme		address_id										
free_cancellation		rate_id										
accommodation_type												

booking	calendar	room	room_booked	room_bed_type	room_type	room_amenity	room_accessibility	room_price	single_room	double_room	family_room	reviews
booking_id	calendar_id	room_no	booking_id	room_bed_type_id	type_id	room_amenity_id	room_accessibility_id	price_id	single_room_id	double_room_id	family_room_id	review_id
booking_date	day	room_name	room_style		quantity_available		room_id	from	single_room_no	double_room_no	family_room_name	review_title
num_of_nights	month	room_rate	hotel_id		date_available		accessibility_id	to	num_single_av	num_double_av	num_family_available	review_body
check_in	year	available_date	booked_dat		price			price_id	num_adult_sing	max_adult_dou	max_adult_family	score
check_out		number_beds						is_available	single_room_p	max_child_dou	room_bed_type_id	hotel_id
payment_id		capacity							room_bed_type	double_room_price		customer_id
hotel_id room_id		free_cancelation								room_bed_type_id		
customer_id		room_reawrds_collect										
		type_id										

rewards	customer	payment	guest	gender	card	gift_card
customer_id	customer_id	payment_id	guest_id	gender_id	card_id	gift_card_id
join	first_name	first_name	first_name	male	cardholder_name	gift_card_number
collect	last_name	last_name	last_name	female	card_long_number	gift_card_amount
redeem	phone_number	billing_house_num	adult_num	other	card_start_date	gift_card_expiry
	email	billing_street	child_num		card_expiry_date	
	address_id	billing_postcode	child_age		card_type	
	password(encrypt)	card_id	special_request			
		gift_card_id	email			
			phone_number			

Figure 1

2. Database Design

To fully encapsulate the booking process, research was conducted on how an individual would book a room in *Hotels.com*. An important aspect of database design "is to characterise fully the data needs of the prospective database users"[1]. This research was carried out by exploring *Hotels.com and creating multiple bookings. During this research a process called entity discovery was employed to organise the data in such a way that it represented the real world. Figure 1* shows an initial interpretation of *Hotels.com's* entities and their corresponding attributes. This became the basis for the initial entity-relationship diagram design which would inform the final schema that was used for the creation of the database. The entity discovery was conducted using an imagined user journey of the website, capturing entities and attributes within a table. *Appendix 1* shows the initial entity-relationship diagram design and is extremely sparse compared to the final design, this was because the initial design was used to capture the three main aspects of the *Hotels.com* booking process. Section 2.1 explains how the initial design was continually tweaked and compounded upon to develop into the final schema for this project.

2.1 Entity Relationship Database Design

An example of a user journey would have the user search for the destination they wish to visit, how long they want to stay at their chosen hotel, and the number of people staying. A search result will be returned with multiple hotels for that destination. Each hotel will show the cheapest room price and after selecting a hotel, the user is shown the different rooms that a user can chose. After choosing a room, the user has the option to sign up for an account, or continue as a site guest. There are extras for each rooms and different payment methods can be used. Once a user has provided payment details they are sent a booking reference, with booking details to their email address. This example was used as the starting for the research and helped in establishing three main aspects of the booking process – customer, hotel, booking.

Customer

When booking a room a user can first sign up for an account and purchase a booking, or they can purchase a booking as a guest. Initially customer was only one entity, however after further research it was later decided that there should be a customer entity that is the parent of a site member customer and a site guest customer. This was realised through the concepts of *generalization* and *specialization*[1] – a site member can post reviews and gain reward points for bookings, whilst a site guest can not. Other personal details were also initially attributes of the customer entity, however during further research a customer account is only made up of a first name, a surname, phone number, email address, and country of residence. Therefore the gender entity table was removed and customer was split into three tables. The rewards table was also removed as this felt unnecessary, the reward points would be stored in the *site_member(appendix figure 2.39)* entity instead as it was assumed that the developer would be responsible for reward points management. The payment process was also broken down using the concepts of generalization and specialization with one main payment table becoming the parent for gift card payment and card payment. A user may only have enough money on a gift card to partially pay for a booking, and a card may have to be used to make up the difference. Both the card payment and the gift card payment entities may not be necessarily stored by *Hotels.com* and a third party provider may process the payments due to the security concerns over storing card information[2]. For this database, naive encryption was used for storing card numbers for demonstration purposes only and in a real

enterprise a more complicated version of encryption would be used(Section 3.1.1). The final entity found was the customer review, which included a title, main body of text and a score. For the purposes of this project the score attribute is an integer value between 0 and 10(Appendix figure 2.31). *Hotels.com* breaks the score down further by asking for a score on certain aspects of a stay, for example cleanliness, quality of service, etc. However, due to the time constraints of this project, the score was left as a single attribute as it does not necessarily affect the booking process. An average score can still be used to calculate each hotel's rating which is more useful when completing a general search of *Hotels.com*.

Hotel

This was the largest section of the entity discovery and the database itself. One hotel contains its name, description, star rating, address, theme, check in information, etc. A hotel will also include numerous rooms of differing types, of which also contain their own attributes. The concept of normalisation was used to ensure that each entity was an entity in its own right. An attribute should be atomic, in that, it should not be divisible. As a room has its own attributes, it is counted as a separate entity. This follows the first normal form. Thus, the room entity now has a relationship with hotel as each hotel has a room(*Appendix figure 2.32*). Each room has one and only one hotel associated with it and this is a one-to-many relationship. The concepts of normalisation and relations will be discussed further below in a separate sections as these justifications become more fleshed out and apparent in the entity-relationship design diagram. Rooms were initially to be divided up into single, double, and family type rooms existing as separate entities. After consideration, however, each room contained a name, description, amount of rooms available in the hotel, the price and hotel id and the maximum amount of child guests allowed. Accessibility was initially tied to rooms, but it was decided to be tied to the hotel and the booking process as accessibility was a check-box selection of choices at the booking stage. As accessibility was an option, it was more logical that it was an entity that was tied to what the hotel could provide in its entirety rather than on a room to room basis(*Figure 2*). Separate tables were created for beds, features and photo urls. This is because a room may have multiple types of beds, different features, and photos of different parts of the room. This again is using normalisation as to ensure the 2nd normal form is adhered to. Rather than having multiple columns for feature1, feature2, feature3 etc… a separate entity is created so that a room can contain multiple features, beds and and photos. The same is the case for the hotel entity in which amenities, landmarks and policies had supportive tables to ensure a hotel could have multiple amenities, landmarks and policies. The hotel entity was also given the relationship to check in details. This is because each hotel will have a policy of check in and out times along with specific instructions for checking in and relevant documents needed. The last component of the hotel section to explain is that of room allocation. Each room has a set amount of rooms and this is shown in the room entity as *amount_of_rooms(Appendix figure 2.32)*. There is then a separate entity which uses the hotel_id and room_id as foreign keys to show how many rooms should be allocated per night. An assumption is made that a developer would use the amount of rooms to generate an allocation depending on business needs. For example, a hotel may limit the amount of rooms during a renovation and certain rooms were unavailable. The room entity is also closely associated with the booking aspect of *Hotels.com*.

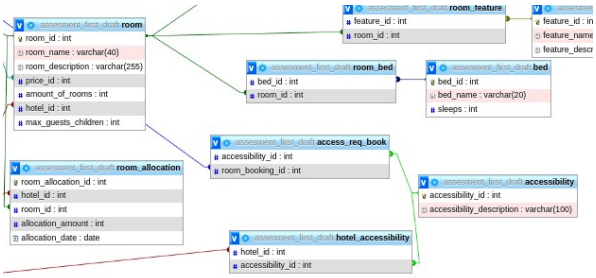


Figure 2

Booking

The booking section of this database was developed to allow one booking to contain multiple room being booked. This includes the price entity as extra cost will be decided at this point. Firstly the booking entity contains a booking reference, payment type, payment and customer ids, check in and out dates and reward points gained from booking. The booking date is a separate date and this is a one to many relationship. Each booking ids can be associated with multiple booking date ids. This is so that a developer can be given how many rooms are booked on a certain day, then compare that with the

allocation table mentioned above. Each day a room is booked is shown in this entity(Figure 3). Another aspect of the booking process is that one booking can contain multiple rooms and this is replicated through the use of a room booking entity. This ended up containing the booking and room ids, any special requests, the guest's name, the cancellation policy, and the terms of the booking. Although it may seem like the guest name being included breaks the rule of unnecessary repetition, the guest staying in the room may not be the person booking the room. *Hotels.com* does auto fill a customer's name if they are signed in to a user account, but this can be updated. Although closely connected to the room entity, the pricing of a room is more apparent during the booking process. A base price is set which is associated with each room in a table names price. An '*extra*'(Appendix 2) table and a '*date_surcharge*' table (Appendix 2) then hold can hold prices to be added to the base price. The extra can be added by the user, for example, breakfast, or late check out. The date surcharge would be populated by the hotelier to add a surcharge for certain days of the week, or longer periods of time such as busy seasons (summer). It is then assumed that a developer will use this information to calculate the final price of the room which is then seen at the front-end.

One aspect of the design that is not added in any of the three areas above is the address entity. The address contains relationships between both hotels, and payment details. This is a simple representation of how an address table should be represented. In an actual enterprise the storing of addresses would more than likely be handled by an API. The make up of addresses are dependent on the country of origin and can become complex when trying to conform to normal forms due to this. Therefore, within this schema a naive representation of an address is constructed using line1, line2, line3, postcode/zip, city, region id, and country id. Region and country were both created as separate entities as there initially going to contain more information than only a name and an id, however due to time constraints both only have the name and id to represent that they could be fleshed out further given more time.

To minimize redundancy within the database schema, the concept of normalisation was utilised during the research of *Hotels.com*. A further discussion of normalisation and exactly how it was used will be discussed below, with examples of the first, second, and third normal forms.

2.1.1 Normalisation

First Normal Form

This is achieved by ensuring that each attribute contains atomic values and only this value only contains a single value. To put simply, there should be “no repeating elements or groups of elements”[3]. Amenities were initially going to be a single value within the hotel table. Amenities then became its own table to store a number of different amenities and these were connected through a hotel entity table to ensure two one-to-many relationships. This ensured that the hotel table did not include columns named ‘amenity1, amenity2,…’, thus conforming to first normal form. An example of this is shown in figure x where the hotel with the blue background does not conform. The green background shows the two one-to-many relationships between hotel_id and *amenity_id*. This is also apparent throughout the schema and can be seen in multiple tables

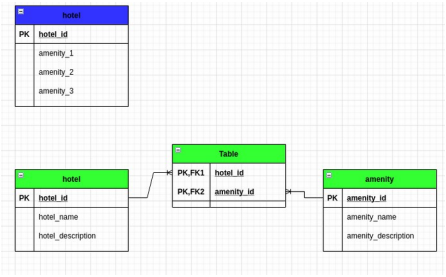


Figure 4

Second Normal Form

A table should only contain information directly related to the primary key or *“no partial dependencies on a concatenated key”[3].* Initially the room for each booking was stored as a single attribute in the booking table, however that meant that if a user wanted to book multiple rooms in one booking then these would be have to be stored twice in the booking table with only the room id changing. To overcome the duplication, a separate table was created named the room_booking table to limit data duplication.

Third Normal Form

By enforcing the rule that there are “no dependencies on non-key attributes”[3], the third normal form will be adhered to. By only including data that directly relates to the primary key. The review table initially held customer name when first created. However, following third normal form rules the customer table became a foreign key of reviews as the customer’s name did not relate to the review itself.

After the rules of normalisation were followed, a final version of the entity relation diagram design was completed and finalised. (Figure 5)

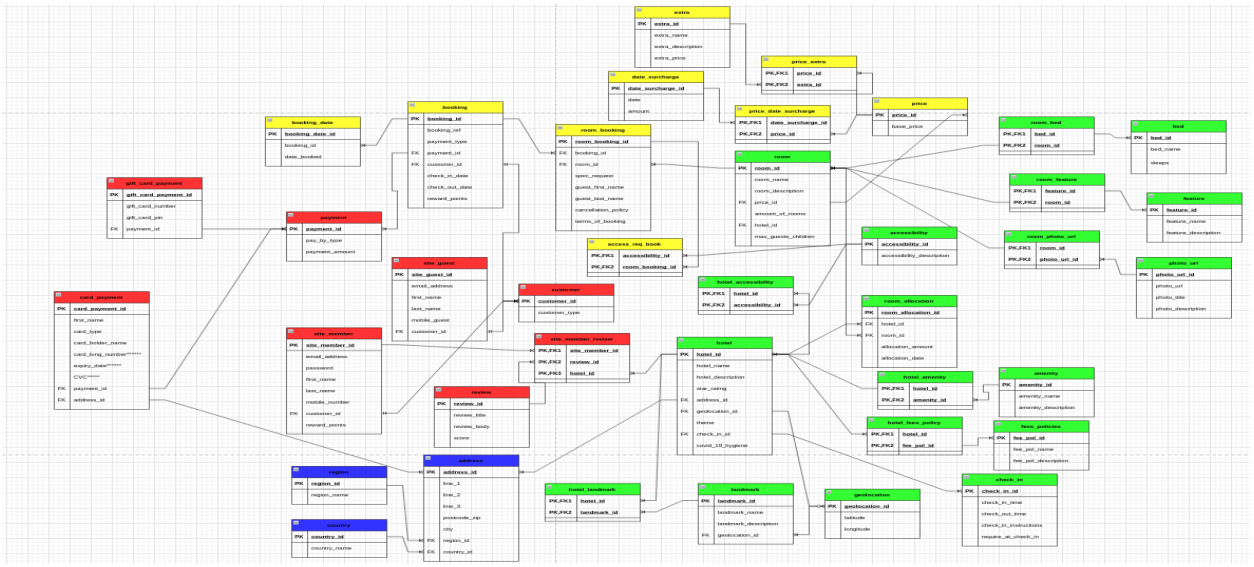


Figure 5

2.2 Keys

Surrogate keys were used as the main primary keys for tables. This was achieved by creating an attribute named x_id where x= table name, with data type of integer and allowing for auto-increment. This ensure uniqueness of primary key as it is an arbitrary number associated with that table's data. By using surrogate keys it is less likely that they turn out to be pseudo unique. For example, a booking reference may be seen as a perfect primary key, however *Hotels.com* may reuse booking references for different hotels. Another reason not to use the reference as a primary key would be to ensure that end users are not given access to internal identifiers.

Composite keys were used as well. For the hotel amenity table hotel id and amenity id foreign keys were used to create the composite key. This was done so that one hotel could have numerous amenities, while confirming to the normalisation principles above and allowing for each record to retain its uniqueness. This use of composite keys can be seen with numerous tables(figs) similar to the hotel amenities table. To ensure that the naming conventions remained consistent within the database, foreign key constraints would follow the pattern of 'FK_x_y' where 'FK' stood for foreign key, x=table referenced, and y=foreign key. Any foreign key set up within this database followed these naming conventions.

Foreign keys were also used to reference entities that could not simply be an attribute. For example, check_in_id was a foreign key in the hotel table. This was because the information surrounding check-ins was to vast to kept in the hotel table and conforming to the third normal form, it should be stored in a separate entity. It, along with similar instances was named 'FK_x_y' as described above.

2.3 Data Types

The two main data types used were 'int' and 'varchar', however 'varbinary', 'text', 'decimal', 'date', and 'tinyint' were also used. Ints were used for surrogate keys with auto_incremental enabled. As

explained above this was to provide uniqueness for multiple tables. There were arbitrary limits of 20 given to many of these surrogate keys, however in an enterprise these may not have a defined limit due to the breadth of data *Hotels.com* would process. Ints were also used for reward points, review score, sleeps, maximum guests, room allocation, room amount and hotel stars. Review score had a limit of 10 as this is the maximum a property can receive and hotel stars were limited to 5 for the same reason. Varchar was also used a considerable amount. Most uses were for the name and description of a field such as landmark or hotel, however varchar was also used for other fields as well. Varchar was used to store mobile number, this is because a user may enter a country code to the number, for example '+44', before their mobile number. Varbinary was used for fields that were encrypted. More on encryption will be detailed below, but it was used for storing specific fields in card_payment and site_member. Varbinary allowed for encryption to take place over card numbers and passwords. The text data type was used for fields that needed more than 255 characters, the prime example being the review_body section in the review table. Decimal was used for all fields that were concerned with amounts of money and were set to length 15,2 to limit the number to only two decimal places to allow for exact pricing. Date was used for any dates entered, and is represented in the format of 'yyyy-mm-dd'. Finally tinyints was used for payment_type in the booking table. This was used as there were two options for paying, either online or at the hotel. '0' denoted paying online, while '1' denoted paying at the hotel.

Section 2 explains the database design and a final view of the schema is shown in Figure 5 which is a screenshot of the relational view of the final database. The following section will exhibit how the booking process works using the created database.

3. Business Enterprise

This section will show how a user interacts with the data contain within the database, by showing how data may be created, read , updated and deleted using Structured Query Language(SQL). Throughout this example , two naive encryption methods for storing a user's password and card details will be discussed.

Creating, reading, updating, and deleting (CRUD) data within a database is integral to any user experience. For this report, an example of a user creating an account and booking a room using SQL will be utilized to exemplify CRUD in action.

3.1 Create

Firstly, as shown in fig x a customer is is created for the user in the first statement by inserting a '1' for the customer_type. This shows that the customer is a site member. The next lines of SQL set variables for the customer's email, name, mobile number, customer id, reward points and finally password. The reason a variable is first set is because the user's password is going to be encrypted. A process of salting a hash is then used. This information is then inserted into the site-member table.

3.1.1 Encryption

The process of salting and hashing was used to store passwords in this database. This does has limitations and is a naive form of encryption. In this process salt is produced by creating a random six character string using the *RAND()* function. The *SHA1()* function will then produce a 160-bit hash value for the random 6 character string(@salt). A salted hash will then be stored which is a concatenation of the salt and the user password using the *CONCAT()* function, then hashed using *SHA1()* function

again(concatenation of @salt and @plainPassword). The password that should then be stored is a concatenation of both the salt and the salted hash(concatenation of @salt and @saltedHash). This can then be decrypted by first asking a user for their email address and password used. The salt will be obtained by getting the first six characters from the user's password using the *SUBSTRING()* function. The salted hash that is used will be obtained by gathering the other 32 characters, then once this is ascertained, the login password is concatenated with the salt in use and checked against the salted hash. If there is an error in the SQL at this point, the login password would be different than the original stored password. For card payments a function named *AES()* is used. *AES()* is a built in function in MySQL and can be seen in fig x, it takes a password that is associated with each encryption of data which makes this extremely unsafe. The decryption will only work if the password entered is the same as the original. This is very problematic, as if this were an enterprise database a bad actor could easily have access to card information of multitudes of customers. This is why it is important to reinforce that this is for demonstration purposes only.

3.2 Read

After a user has been created, they decide to search for hotels in Manchester and after viewing all of the hotels they decide on a hotel and view all of the rooms with their features. A room is then booked, but changed as they decide they need to update the special requests section for their guest. This is shown as a transaction in Appendix 3.

3.3 Update

Within the transaction the statement starts by declaring 'START TRANSACTION;', this is to effectively turn off auto commit in MySQL. A commit statement is at the very end of the transaction to then commit the whole statement. Within the transaction, an 'INSERT' statement creates a booking, then a booking date. A variable is set so that the transaction saves the auto generated booking id for the booking date insert statement, this is repeated for the room_booking UPDATE statement. The information is saved using 'SAVEPOINT' and naming it 'SAVEPOINT1', this is to represent the booking being saved in some form of basket. After the INSERT for room_booking, the user has had to update the special requirements, This was completed using the UPDATE statement to change the room_booking using the stored variable for room_booking_id.

3.4 Delete

In this example, the user has realised that their friend cannot make the trip, so they have to cancel the booking this is shown in Appendix 3 and follows the same transaction structure as above. The only difference is that the booking table information has to be the last record to be deleted due to the foreign key constraints.

4.Improvements

In [fig x], there is a notable flaw in the design of this data. To exemplify which rooms are most suitable for only two people, family sized rooms are also returning. Normalising the beds to include how many people could sleep in it may not have been necessary. When booking a hotel room, a customer does not have the question of how many people can sleep in one bed, but rather how many people are

permited in one room. Following this logic, the room itself should hold the maximum amount of guests instead of each bed holding a maximum of people that could sleep in it. Although the design would allow for a developer to create code to represent the room's maximum amount is possible, it would make real world sense to store this in the room. For example with Covid regulations, a room may only be able to contain a maximum amount of people, although the beds may have an amount associated to them, the defining factor will be the maximum allowed in one room. Thus, creating an attribute of '*room_maximum_capacity_id'*, with '*maximum_guests'*, and '*maximum_children'* may be more suitable.

In the '*room_booking'*(fig x) table, both '*cancellation_policy'* and '*terms_of_booking'* could be normalised to create a '*terms_and_cancellation*' table. Rather than having to populate each time a booking is taking place, a hotel would have more than likely have only one cancellation policy and one set of terms of booking per room. There may not even be different policies and terms for each room, thus it creates replication of data within the database. This does not comply with normalisation which is a fundamental part of good relational database design. If this was created again, a separate table named '*cancellation_terms_booking*' would be created to store relevant information and it would be a foreign key in the hotel table.

The booking table contains one relationship to the booking date table, however the relationship should be between the '*room_booking table*', and the '*booking_date'* table. This is because a booking can contain multiple rooms, on multiple nights, however the '*room_booking'* table is one specific room. A booking may contain two different rooms of a hotel booked for different nights. Therefore, the '*booking_date'* table should have a foreign key relationship with '*room_booking*'. This was only realised after all of the mock data was created and quires were being ran, however this would be changed if it were to be made again.

The final improvements would be to include a more robust form of encryption for user data and card payment details, and a better representation of the address entity. The encryption may not even be handled by *Hotels.com* and may be provided by a third party enterprise such as SumUp or Stripe. This would remove the need to handle and store information that could be vulnerable to attacks. An API for address storage may also make more sense due to the complexity of addresses. As mentioned above, different countries contain different ways of constructing an address. As *Hotels.com* provides hotel bookings internationally, it is complicated to construct an address correctly. An API call may enable the correct storage of addresses.

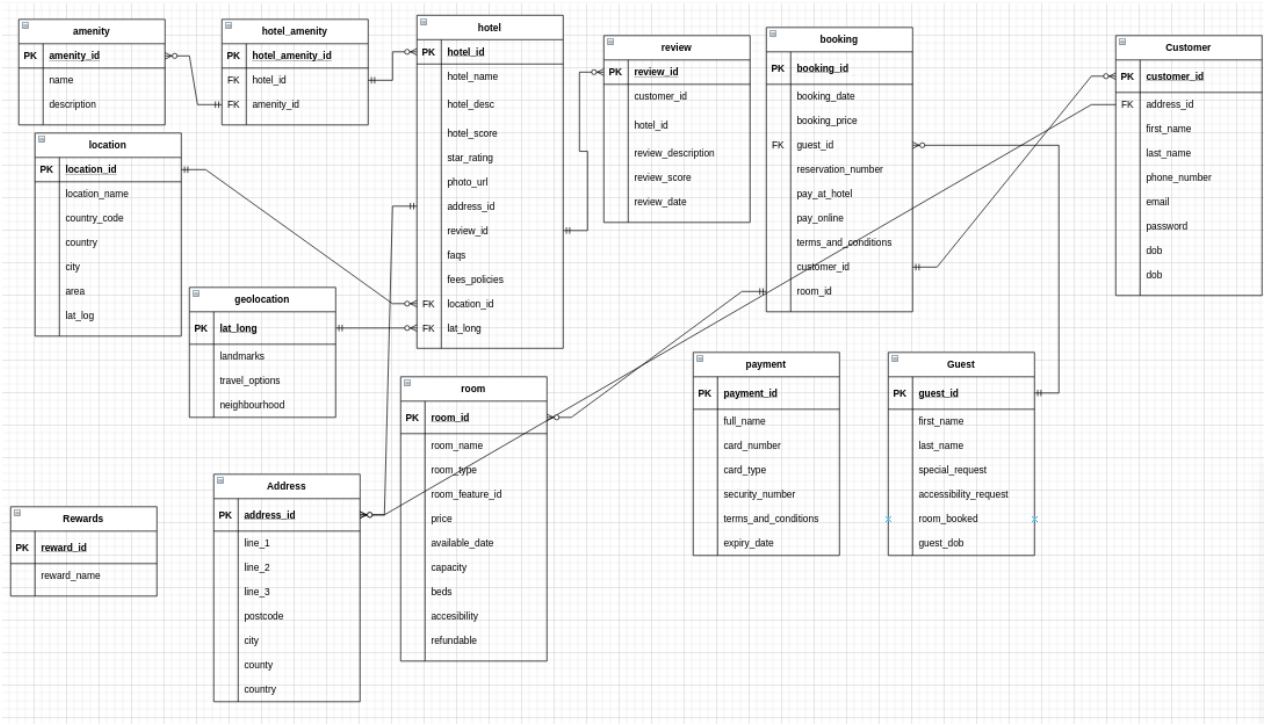
5.Conclusion

The aim of this report was to show how a relation database management system was developed to mirror the booking process of *Hotels.com*. Through this report an examination of key design decisions was shown with an explanation and justification for those design choices. This was laid out by explaining how the concept of normalisation aiding in entity discovery and in the initial entity-relationship design diagram. With further research of *Hotels.com*, a final version of the diagram was established and this was used as the final schema for this project. Data types were chosen for each attribute, and then f=keys were established. This was aided through the concept of normalisation again. Dummy data was then inserted into the using INSERT statements in SQL, an example of one of these mass INSERT statements can be seen in fig x. After the data was inserted, queries were ran in SQL to exhibit how the database processed a sample booking. This was conducted by creating records, reading records, updating records,

then deleting a record. A transaction was also ran to show how a database may work in real time. The above exemplifies how a database for *Hotels.com* may be structured. Although there are a number of alterations that could be made within the schema, this is a sound representation of how a booking system may be managed. By justifying specific components of database design using theoretical knowledge, this database is a fair representation of a real world enterprise's own database.

Appendices

Appendix 1 – Initial entity discovery diagram



Appendix 2 – Data tables

accessibility

Column	Type	Null	Default	Comments
accessibility_id *(Primary)*	int	No		
accessibility_description	varchar(100)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	accessibility_id	15	A	No	

Appendix Figure 2.1

access_req_book

Column	Type	Null	Default	Comments
accesibility_id	int	No		
room_booking_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_access_req_book_room_booking_id	BTREE	No	No	room_booking_id	3	A	No	
FK_access_req_book_accesibility_id	BTREE	No	No	accesibility_id	6	A	No	

Appendix Figure 2.2

address

Column	Type	Null	Default	Comments
address_id *(Primary)*	int	No		
line_1	varchar(60)	No		
line_2	varchar(60)	No		
line_3	varchar(60)	No		
postcode_zip	varchar(60)	No		
city	varchar(85)	No		
region_id	int	No		
country_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	address_id	31	A	No	
FK_address_country_id	BTREE	No	No	country_id	9	A	No	
FK_address_region_id	BTREE	No	No	region_id	15	A	No	

Appendix Figure 2.3

amenity

Column	Type	Null	Default	Comments
amenity_id *(Primary)*	int	No		
amenity_name	varchar(40)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	amenity_id	12	A	No	

Appendix Figure 2.4

bed

Column	Type	Null	Default	Comments
bed_id *(Primary)*	int	No		
bed_name	varchar(20)	No		
sleeps	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	bed_id	8	A	No	

Appendix Figure 2.5

booking

Column	Type	Null	Default	Comments
booking_id *(Primary)*	int	No		
booking_ref	varchar(20)	No		
payment_type	tinyint(1)	No		
payment_id	int	No		
customer_id	int	No		
check_in_date	date	No		
check_out_date	date	No		
reward_points	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	booking_id	7	A	No	
booking_ref	BTREE	Yes	No	booking_ref (16)	7	A	No	
FK_booking_payment_id	BTREE	No	No	payment_id	7	A	No	
FK_booking_customer_id	BTREE	No	No	customer_id	5	A	No	

Appendix Figure 2.6

booking_date

Column	Type	Null	Default	Comments
booking_date_id *(Primary)*	int	No		
booking_id	int	No		
date_booked	date	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	booking_date_id	6	A	No	
FK_booking_date_booking_id	BTREE	No	No	booking_id	4	A	No	

Appendix Figure 2.7

card_payment

Column	Type	Null	Default	Comments
card_payment_id *(Primary)*	int	No		
cardholder_name	varchar(50)	No		
card_type_id	int	No		
card_long_number	varbinary(255)	No		
expiry_date	varbinary(255)	No		
CVC	varbinary(255)	No		
payment_id	int	No		
address_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	card_payment_id	2	A	No	
FK_card_payment_payment_id	BTREE	No	No	payment_id	2	A	No	
FK_card_payment_address_id	BTREE	No	No	address_id	2	A	No	
FK_card_payment_card_type_id	BTREE	No	No	card_type_id	2	A	No	

Appendix Figure 2.8

card_type

Column	Type	Null	Default	Comments
card_type_id *(Primary)*	int	No		
card_type_name	varchar(100)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	card_type_id	6	A	No	

Appendix Figure 2.9

check_in

Column	Type	Null	Default	Comments
check_in_id *(Primary)*	int	No		
check_in_time	time	No		
check_out_time	time	No		
check_in_instructions	text	No		
req_at_check_in	varchar(255)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	check_in_id	10	A	No	

Appendix Figure 2.10

country

Column	Type	Null	Default	Comments
country_id *(Primary)*	int	No		
country_name	varchar(85)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	country_id	10	A	No	

Appendix Figure 2.11

customer

Column	Type	Null	Default	Comments
customer_id *(Primary)*	int	No		
customer_type	tinyint(1)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	customer_id	19	A	No	

Appendix Figure 2.12

date_surcharge

Column	Type	Null	Default	Comments
date_surcharge_id *(Primary)*	int	No		
date	date	No		
amount	decimal(15,2)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	date_surcharge_id	26	A	No	

Appendix Figure 2.13

extra

Column	Type	Null	Default	Comments
extra_id *(Primary)*	int	No		
extra_name	varchar(30)	No		
extra_description	varchar(200)	No		
extra_price	decimal(15,2)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	extra_id	4	A	No	

Appendix Figure 2.14

feature

Column	Type	Null	Default	Comments
feature_id *(Primary)*	int	No		
feature_name	varchar(30)	No		
feature_description	varchar(255)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	feature_id	10	A	No	

Appendix Figure 2.15

fees_policies

Column	Type	Null	Default	Comments
fee_pol_id *(Primary)*	int	No		
fee_pol_name	varchar(30)	No		
fee_pol_description	text	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	fee_pol_id	7	A	No	

Appendix Figure 2.16

geolocation

Column	Type	Null	Default	Comments
geolocation_id *(Primary)*	int	No		
latitude	decimal(15,10)	No		
longitude	decimal(15,10)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	geolocation_id	41	A	No	

Appendix Figure 2.17

gift_card_payment

Column	Type	Null	Default	Comments
gift_card_payment_id *(Primary)*	int	No		
gift_card_number	varbinary(255)	No		
gift_card_pin	varbinary(255)	No		
payment_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	gift_card_payment_id	4	A	No	
FK_gift_card_payment_payment_id	BTREE	No	No	payment_id	4	A	No	

Appendix Figure 2.18

hotel

Column	Type	Null	Default	Comments
hotel_id *(Primary)*	int	No		
hotel_name	varchar(40)	No		
hotel_description	varchar(255)	No		
star_rating	int	No		
address_id	int	No		
geolocation_id	int	No		
theme	varchar(50)	No		
check_in_id	int	No		
covid_19_hygiene	text	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	hotel_id	7	A	No	
FK_hotel_address_id	BTREE	No	No	address_id	7	A	No	
FK_hotel_check_in_id	BTREE	No	No	check_in_id	6	A	No	
FK_hotel_geolocation_id	BTREE	No	No	geolocation_id	7	A	No	

Appendix Figure 2.19

hotel_accessibility

Column	Type	Null	Default	Comments
hotel_id	int	No		
accessibility_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_hotel_accessibility_hotel_id	BTREE	No	No	hotel_id	1	A	No	
FK_hotel_accessibility_accessibility_id	BTREE	No	No	accessibility_id	7	A	No	

Appendix Figure 2.20

hotel_amenity

Column	Type	Null	Default	Comments
hotel_id	int	No		
amenity_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_hotel_amenity_hotel_id	BTREE	No	No	hotel_id	1	A	No	
FK_hotel_amenity_amenity_id	BTREE	No	No	amenity_id	5	A	No	

Appendix Figure 2.21

landmark

Column	Type	Null	Default	Comments
landmark_id *(Primary)*	int	No		
landmark_name	varchar(30)	No		
landmark_description	varchar(255)	No		
geolocation_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	landmark_id	11	A	No	
FK_landmark_geocation_id	BTREE	No	No	geolocation_id	1	A	No	

Appendix Figure 2.22

hotel_fee_policy

Column	Type	Null	Default	Comments
hotel_id	int	No		
fee_pol_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_hotel_fees_policies_fee_pol_id	BTREE	No	No	fee_pol_id	7	A	No	
FK_hotel_fees_policies_hotel_id	BTREE	No	No	hotel_id	1	A	No	

Appendix Figure 2.23

hotel_landmark

Column	Type	Null	Default	Comments
hotel_id	int	No		
landmark_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_hotel_landmark_hotel_id	BTREE	No	No	hotel_id	1	A	No	
FK_hotel_landmark_landmark_id	BTREE	No	No	landmark_id	7	A	No	

Appendix Figure 2.24

price

Column	Type	Null	Default	Comments
price_id *(Primary)*	int	No		
base_price	decimal(15,2)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	price_id	18	A	No	

Appendix Figure 2.27

payment

Column	Type	Null	Default	Comments
payment_id *(Primary)*	int	No		
pay_by_type	tinyint	No		
payment_amount	decimal(15,2)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	payment_id	14	A	No	

Appendix Figure 2.25

photo_url

Column	Type	Null	Default	Comments
photo_url_id *(Primary)*	int	No		
photo_url	varchar(255)	No		
photo_title	varchar(20)	No		
photo_description	varchar(255)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	photo_url_id	40	A	No	

Appendix Figure 2.26

price_date_surcnarge

Column	Type	Null	Default	Comments
date_surcharge_id	int	No		
price_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_price_date_surcharge_price_id	BTREE	No	No	price_id	2	A	No	
FK_price_date_surcharge_date_surcharge_id	BTREE	No	No	date_surcharge_id	2	A	No	

Appendix Figure 2.28

price_extra

Column	Type	Null	Default	Comments
extra_id	int	No		
price_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_price_extra_price_id	BTREE	No	No	price_id	1	A	No	
FK_price_extra_extra_id	BTREE	No	No	extra_id	2	A	No	

Appendix Figure 2.29

region

Column	Type	Null	Default	Comments
region_id *(Primary)*	int	No		
region_name	varchar(85)	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	region_id	37	A	No	

Appendix Figure 2.30

review

Column	Type	Null	Default	Comments
review_id *(Primary)*	int	No		
review_title	varchar(255)	No		
review_body	text	No		
score	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	review_id	34	A	No	

Appendix Figure 2.31

room

Column	Type	Null	Default	Comments
room_id *(Primary)*	int	No		
room_name	varchar(40)	No		
room_description	varchar(255)	No		
price_id	int	No		
amount_of_rooms	int	No		
hotel_id	int	No		
max_guests_children	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	room_id	6	A	No	
FK_room_price_id	BTREE	No	No	price_id	6	A	No	
FK_room_hotel_id	BTREE	No	No	hotel_id	2	A	No	

Appendix Figure 2.32

room_allocation

Column	Type	Null	Default	Comments
room_allocation_id *(Primary)*	int	No		
hotel_id	int	No		
room_id	int	No		
allocation_amount	int	No		
allocation_date	date	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	room_allocation_id	6	A	No	
FK_room_allocation_hotel_id	BTREE	No	No	hotel_id	2	A	No	
FK_room_allocation_room_id	BTREE	No	No	room_id	3	A	No	

Appendix Figure 2.33

room_bed

Column	Type	Null	Default	Comments
bed_id	int	No		
room_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_room_bed_bed_id	BTREE	No	No	bed_id	4	A	No	
FK_room_bed_room_id	BTREE	No	No	room_id	6	A	No	

Appendix Figure 2.34

room_booking

Column	Type	Null	Default	Comments
room_booking_id *(Primary)*	int	No		
booking_id	int	No		
room_id	int	No		
spec_request	varchar(255)	No		
guest_first_name	varchar(40)	No		
guest_last_name	varchar(40)	No		
cancellation_policy	text	No		
terms_of_booking	text	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	room_booking_id	7	A	No	
FK_room_booking_booking_id	BTREE	No	No	booking_id	5	A	No	
FK_room_booking_room_id	BTREE	No	No	room_id	4	A	No	

Appendix Figure 2.35

room_feature

Column	Type	Null	Default	Comments
feature_id	int	No		
room_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_room_feature_feature_id	BTREE	No	No	feature_id	6	A	No	
FK_room_feature_room_id	BTREE	No	No	room_id	6	A	No	

Appendix Figure 2.36

room_photo_url

Column	Type	Null	Default	Comments
room_id	int	No		
photo_url_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_room_photo_url_photo_url_id	BTREE	No	No	photo_url_id	8	A	No	
FK_room_photo_url_room_id	BTREE	No	No	room_id	6	A	No	

Appendix Figure 2.37

site_guest

Column	Type	Null	Default	Comments
site_guest_id *(Primary)*	int	No		
email_address	varchar(255)	No		
first_name	varchar(50)	No		
last_name	varchar(50)	No		
mobile_number	varchar(10)	No		
customer_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	site_guest_id	6	A	No	
FK_site_guest_customer_id	BTREE	No	No	customer_id	6	A	No	

Appendix Figure 2.38

site_member

Column	Type	Null	Default	Comments
site_member_id *(Primary)*	int	No		
email_address	varbinary(255)	No		
password	varbinary(255)	No		
first_name	varchar(70)	No		
last_name	varchar(70)	No		
mobile_number	varchar(10)	No		
customer_id	int	No		
reward_points	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	site_member_id	2	A	No	
FK_site_member_customer_id	BTREE	No	No	customer_id	2	A	No	

Appendix Figure 2.39

site_member_review

Column	Type	Null	Default	Comments
site_member_id	int	No		
review_id	int	No		
hotel_id	int	No		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
FK_site_member_review_site_member_id	BTREE	No	No	site_member_id	3	A	No	
FK_site_member_review_review_id	BTREE	No	No	review_id	7	A	No	
FK_site_member_review_hotel_id	BTREE	No	No	hotel_id	4	A	No	

Appendix Figure 2.40

Appendix 3. SQL Statements

Appendix 3.1 – Simple insert statement (CREATE)

```sql
INSERT INTO `region` (`region_id`, `region_name`) VALUES (NULL, 'Britol');

INSERT INTO `region` (`region_id`, `region_name`) VALUES (NULL, 'Buckinghamshire');

INSERT INTO `region` (`region_id`, `region_name`) VALUES (NULL, 'Cambridgeshire');

INSERT INTO `region` (`region_id`, `region_name`) VALUES (NULL, 'Chesire');

INSERT INTO `region` (`region_id`, `region_name`) VALUES (NULL, 'City of London');

INSERT INTO `region` (`region_id`, `region_name`) VALUES (NULL, 'Cornwall');

INSERT INTO `region` (`region_id`, `region_name`) VALUES (NULL, 'Cumbria');

INSERT INTO `region` (`region_id`, `region_name`) VALUES (NULL, 'Derbyshire');
```

Appendix 3.2- Encryption

```sql
/*Creates a customer entity*/

INSERT INTO `customer` (`customer_id`, `customer_type`) VALUES (NULL, '1');

SET @last_in_customer_id = LAST_INSERT_ID();

#Define our email, name, mobile, customerId, rewardPoints
SET @email = 'hotelEnthusiast@hotmail.com';
SET @firstName ='Hotel';
SET @lastName ='Enjoyer';
SET @mobile ='+44 4567895';
SET @customerID =@last_in_customer_id;
SET @rewardPoints ='0';


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



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

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

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

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

#Store this user in the database
INSERT INTO site_member (site_member_id, email_address, password, first_name, last_name, mobile_number, customer_id, reward_points)
                  VALUES (NULL, @email, @storedSaltedHash, @firstName, @lastName, @mobile, @customerID, @rewardPoints);


#Get the password attempt entered by the user as LOGIN
SET @loginPassword = 'easyToGuess';
SET @loginEmail = 'hotelLover@hotmail.com';


/* Start (very) basic password checking */

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

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

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

/*READ*/
```

Appendix 3.3 READ

```sql
/*READ*/

SELECT * FROM hotel WHERE address_id IN(SELECT address_id FROM address WHERE city IN('Manchester'));

SELECT * FROM room  WHERE hotel_id IN (SELECT hotel_id FROM hotel WHERE hotel_name IN('Copthorne Hotel Manchester'));

SELECT * FROM room_feature INNER JOIN room on room_feature.room_id=room.room_id INNER JOIN feature ON room_feature.feature_id=feature.feature_id WHERE room.room_id IN(SELECT room.room_id FROM room WHERE hotel_id IN (SELECT hotel_id FROM hotel WHERE hotel_name
IN('Copthorne Hotel Manchester')));
```

Appendix 3.4 - Transaction

```sql
/*Update/Transaction*/

START TRANSACTION;

INSERT INTO `booking` (`booking_ref`, `payment_type`, `payment_id`, `customer_id`, `check_in_date`, `check_out_date`, `reward_points`) VALUES ('BOOKREF130', '0', '1', '21', '2021-11-05', '2021-11-06', '1');

SET @last_id_in_booking = LAST_INSERT_ID();


INSERT INTO `booking_date` (`booking_date_id`, `booking_id`, `date_booked`) VALUES (NULL, @last_id_in_booking, '2021-11-05');

SAVEPOINT SAVEPOINT1;

INSERT INTO `room_booking` (`room_booking_id`, `booking_id`, `room_id`, `spec_request`, `guest_first_name`, `guest_last_name`, `cancellation_policy`, `terms_of_booking`) VALUES (NULL, @last_id_in_booking, '2', 'N/A', 'My', 'Friend', 'this is the cancellation policy',
'These are the terms of booking');

SET @last_id_in_room_booking = LAST_INSERT_ID();

#ROLLBACK TO SAVEPOINT1;

RELEASE SAVEPOINT SAVEPOINT1;

UPDATE `room_booking` SET `spec_request` = 'Need extra pillow' WHERE `room_booking`.`room_booking_id` = @last_id_in_room_booking;

COMMIT;
```

Appendix 3.5 - DELETE

```sql
/*DELETE*/

START TRANSACTION;

DELETE FROM `booking_date` WHERE `booking_date`.`booking_date_id` = 9;


DELETE FROM `room_booking` WHERE `room_booking`.`room_booking_id` = 10;

DELETE FROM `booking` WHERE `booking`.`booking_id` = 17;

COMMIT;
```

Appendix 4 – Literature References

[1] - Database System Concepts. 6th Edition, by Silberschatz, Korth and Sudarshan

[2] - http://www.andrewrollins.com/2009/08/11/database-normalization-first-second-and-third-normal-forms/

[3] - http://phlonx.com/resources/nf3/