**Assessment info**

* No database file needed
* Marking based entirely on report and video

**Written report info**

* Pdf form
* Submitted through canvas
* No cover page or table of contents
* Sections
  + Overview of db design – designer view/ draw io
    - Start and final – allows to explain key design decisions
    - More diagrams can go in appendix
    - Talk through noramlisations and evolution of design
  + List of design assumptions that were made
    - What is in and out of scope?
    - Thisis about package holidays
    - Aircraft baggage etc/ meals out of scope
    - Package holiday: hotel room for x days and flights
    - Assume room and flight are tied, if you cang et one then you can get the other
    - Direct flights only
    - Enumerate the most key assumptions
  + Explanation of some important primary and fk constraints
    - Surrogates?
    - Composite keys?
    - Foreign keys to prouce 1:1, 1:N, N:M etc
    - Types and why they were chosen
    - Not interested in a dictionary of everything
  + Some Important design desicisons **(paired with below – same section)**
    - Micro ones aren’t really necessary
    - Key decisions around prices, availability, pricing etc
  + Some important normalisation decisions **(paired with above)**
    - Good pragmatic decisions
    - Not so interested in specific normal forms, but the pragmatic WHY of decisions
  + Improvements
    - Design improvements (own section maybe)
    - We are only building selected aspects of the system, so there will be reasons why some tables are empty, why some things are not implemented
  + SQL Queries in appendix
    - Demo of important aspects
    - Record all ones from video in report

Text

Description automatically generated

**Notes:**

Display selectable durations

Show pricing differences for different airport/ flight dates

Balanced owed query

Add intermediate ERD to appendix

--- FEEDBACK LECTURE NOTES

Validate data type lengths above data layer is probably smarter

**Database Design Report**

**by Peter Marley (student number 13404067)**

**Introduction and Project Context**

This project is an attempt to reverse engineer the database layer of certain sections of the professional commercial booking system of the [Jet2Holidays](https://www.jet2holidays.com/) website, to create a relational database that mirrors it’s functionality. The Jet2Holiday website is vast and was developed iteratively over many years, and contains much functionality that would not be possible to recreate within the timescale allowed for this project. Thus, the scope of this project will be limited to the data storage requirements necessary to store data for 3 main areas of interest – namely passengers, flights, and hotels – in such a way as to allow a developer to subsequently implement functionality visible on the website.

A high-level overview of the functionality of interest on the website is as follows:

1. Search for a holiday package using departure airport, holiday destination, holiday duration, and number of passengers and their ages.
2. Make a booking for a holiday based on selected departure airport, hotel, room type, board basis, holiday duration, number of passengers. A single passenger is nominated as the booking contact and that passenger requires extra information to be stored, namely address, email and telephone contact information.
3. A room availability model.
4. Variable flight pricing dependent on date and time.
5. Variable accommodation pricing dependent date.
6. Secure storage of payment information, implementing at least the most basic requirements of PCI SSC policies and procedures.
7. Create SQL queries and statements to demonstrate the various CRUD operations underpinning the above use cases.

The database is to serve the data storage needs of all these use cases, and additionally to store all data visible on the in scope sections of the website, which are discussed below.

**Design Assumptions**

Initially we operated under the following design assumptions:

1. If a room is available to choose, a corresponding outbound and return flight is assumed to be available.
2. Financing and recurring monthly payment of the cost of a booking was considered out of scope, but multiple part payments or a single complete payment were considered in scope.
3. Any data required to implement fine grained flight booking were out of scope; namely specific seating on a flight, extra carry-on baggage, hold luggage and it’s pricing and selection, extra leg room seating, suitability of seating for infant passengers, sports equipment transport, and flight meals.
4. Flights are assumed to be direct to destination. Multiple leg flights are considered out of scope.
5. Transfers from airports to hotels were considered out of scope.
6. Car Hire was considered out of scope.
7. Any form of insurance was out of scope.
8. Scope is limited to the Jet2Holidays section of the website, excluding others – such as Agent Finder, Jet2Villas and Jet2CityBreaks, although the latter two have considerable overlap in functionality with Jet2Holidays.
9. We will not be able to implement full Payment Card Industry Security Standard Council (PCI SCC) policies and procedures regarding storage of payment card information. These are very detailed and prescriptive, instead we settled for implementing basic 2-way encryption of card details using AES encryption using a known Secret, adhering to the most basic principles of the Payment Card Industry Data Security Standard (PCI DSS), a subset of PCI SSC policies relating to payment card information storage.
10. A country may use many currencies and languages, but I shall store only the countries primary currency and language data.

**Database Design Process**

Initial entity discovery was undertaken individually. We analysed the Jet2Holidays website to try to deduce the entities and attributes underpinning data visible on in-scope sections of the website. We combined our entities into one list, and as a group pinpointed which of these bits of information were entities, and which were attributes. Furthermore, we attempted some initial pragmatic normalisation of attributes to remove obvious repetition of data, expand composite attributes into their constituent attributes, and decide which were more appropriate as derived/ computed attributes. Ultimately this process was intended to bring all the entities and their attributes up to the standard of the 3rd normal form at a minimum.

**Currency and Language**

I made the decision that all prices in the database where to be stored in Great British Pounds sterling (GBP), as Jet2Holidays is specifically a British package holiday provider. I investigated this by visiting the website via a virtual private network that spoofed my connection’s origin as another country. The prices were always shown as GBP. I also searched for any other top-level domains Jet2Holiday instances may reside in, but could find none. The prices are therefore based in British tender. If the website wished to expand to other markets, it would seem likely to me that either the new market would be served by a separate instance of the Jet2Holday system under a different top-level domain, or the system would use an external service or API to convert the GBP prices recorded in the database into a localised currency based on exchange rates. I acknowledge that this design decision may cause unfavourable re-design decisions in the future if the website would like to expand its service to operate natively under other currencies, as attributes would either need renamed (e.g. total\_cost\_gbp attribute of the booking entity may need renamed total\_cost\_eur), or foreign key attributes linking currency entities may need to be added to all price containing entities (route\_price, room\_type\_price and hotel\_board\_type).

Different currencies are still stored within the currency entity, as this data is visible on the website at the level of the region (e.g., Costa Blanca). I chose to store my currencies at the level of country, specifically in the form of a primary\_currency\_id foreign key in the country entity, which are then accessible via a chain of 1:N relationships from region entity up to the currency entity.

Figure 1:N relationship chain from currency to region

**Evolution of Design**

My initial Entity-Relationship Diagram (figure 2) describes our entity-relationship model, however it is naïve in many aspects within the specified scope of the project – these issues were mostly addressed in the final diagram (figure 3):

Naming Conventions

Naming conventions of entities and their attributes are inconsistent. In the end I opted for lower-case and snake-case (words separated with an underscore), with no pluralisation. Notable examples are:

1. The initial design has an entity named Hotel\_images, but the final design settled on hotel\_image, removing the capitalisation and pluralisation in the name
2. Board Types in the initial design became board\_type in the final design, removing capitalisation and internal whitespace from the entity name.
3. Hotels entity became hotel, removing capitalisation and pluralisation.

Pricing of Rooms, Board Basis and Flights

The initial design does not consider pricing at all (flights, hotel rooms and board basis selected)! Subsequently it does not consider time/date dependent pricing, nor how to tabulate payments made nor balanced owed on a booking. The final design addresses this as discussed in the bullet points below.

1. **Hotel Room Pricing:**
   1. A room\_type\_price entity was created that allows a hotel room – which we have priced at the hotel’s room\_type level, as they appear to be on the website – to have individual pricing across date ranges via the room\_type\_price\_valid\_from\_date and room\_type\_price\_valid\_to\_date attributes.
   2. A room booking must consider the price of the board basis selected. All-inclusive, after all, will be priced differently from a self-catering board basis. Board basis’ are defined in the board\_type entity, and the price of the board basis is stored in the hotel\_board\_type entity, which is linked via a foreign key to a board\_types primary key in a 1:N relationship. This allows the board basis to be used across multiple hotels, with each hotel implementing its own pricing for each board basis.
   3. A hotel rooms price therefore is based upon several factors: the hotel to which it belongs, the room type of the room, the board basis selected and the date upon which the room is required. I made the design decision to limit this range to dates and not times, as a hotel room is booked by the day and therefore time is not a relevant consideration.
2. **Flight Pricing:**
   1. A route\_price entity was created that allows a flight to be priced depending on both the route upon which it flies, and the date and time upon which the flight is to depart. The reason I implemented both time and date dependent pricing is due to how certain flights are more desirable and so therefore command a higher price. For example, a flight at 9am on a Monday will be priced differently (due to different demand) than a flight at 1am on a Sunday morning. Including the time in the pricing model allows us to account for this.
3. One issue with our room/ flight pricing model is that I understand that holiday pricing is dynamic in that the price of a room on any given day, not only depends on the date the booking is for, but also the date that the booking is made. Booking a room for tomorrow might incur additional “scarcity” costs, over booking a room further in the future, assuming that all other costs are equal. We have isolated our pricing models into their own tables and so a way of having this dynamic pricing would be to update the respective valid\_from and valid\_to attributes of the route\_price and room\_type\_price entities as a scheduled job, perhaps during maintenance down time. These pricing tables are the source of truth for pricing, and so updating these tables will update the price for all queries using the updated records. The booking\_line\_items entity and subsequently the booking\_cost\_gbp attribute in the booking entity record the price of a booking at the time it was made, so updating the pricing tables would not affect the cost of a previously finalised booking.

Payment Card Information

The initial design does not consider how to store payment card information, nor the PCI SSC/ PCI DSS requirements of doing so. It is worth noting that I recognise that in a real-world system this payment card information would be stored by a specialised business that does implement the full set of PCI SSC/ PCI DSS policies and procedures and would abstract the legal and financial liability of doing so from Jet2Holidays to said payment processing business.

1. My final design settled on encrypting the payment card’s long number (*primary account number* or *PAN* as per PCI SSC policies) and expiry date. These are classified as *Card Holder Data* by PCI SSC policies and may be stored, if they are encrypted in line with their requirements. Our encryption method falls short of the stringent guidelines of PCI SSC, in that they are to be stored using “One-way hash functions based on strong cryptography”, whereas we implemented storage using a 2-way AES encryption method using a known secret.
2. We chose not to store any *Sensitive Authentication Data* such as the cards CVV number. As per PCI SSC policies and procedures these may be used to authenticate the payment card, but must only be stored by certain legitimate authorisation entities such as the card issuer.
3. It does not consider the ability for a user to create an account and save provisional booking information to a short list.
4. Some entities are not normalised as far as the 3rd normal form, for example, star\_rating attribute of the review entity violates .
5. The estimated\_time attribute of the flight entity (which was intended to describe the time duration of a flight) would be better represented as derived attributes. Unfavourable update dependencies have been introduced: for example if a flight arrival time is updated, estimate\_time now requires an update at the same time or data update anomalies may be introduced!
6. The diagram does not consider data types for each attribute.

Geographical Information

Text

Description automatically generated with low confidenceIt does not adequately describe the set of relationships between the geographic entities destination, region, resort, and hotel visible on the website. The region entity (the name we later assigned to the data *Costa Blanca* in *figure 1*) is missing entirely! These were later added in a “chain” of relationships that links all this data together via several 1:M relationships. From the *figure 1* example destination *Spain* contains many regions, region *Costa Blanca* contains many resorts, resort *Benidorm* contains many hotels.

Figure Example of layers of geographical data from the website, and our chosen names for our entities representing them in red.

Diagram

Description automatically generated

Figure Initial Entity Relationship Diagram

Diagram

Description automatically generated

Figure Final Entity Relationship Diagram

**Important Normalisation Decisions**

In this section I will discuss some of the key normalisation decisions taken on the road to producing our database schema. This list of key decisions is far from exhaustive.

Booking contact and contact info

Geographic location information chain (location\_name, country, destination, region, resort, hotel, gps)

Bits of info shared between hotels, that have hotel specific information at the hotel level (hotel\_board\_type + board\_type// room\_type\_facility + room\_type // hotel\_facility + hf\_type + hk\_image + hf bullet)

Flight info (flight has route, airport, route\_price)

Star\_rating plus flag

Although normalisation was approached as a pragmatic on-going process in the design of this database, certain key normalisations were undertaken early on to bring our tables up to the 3rd normal form.

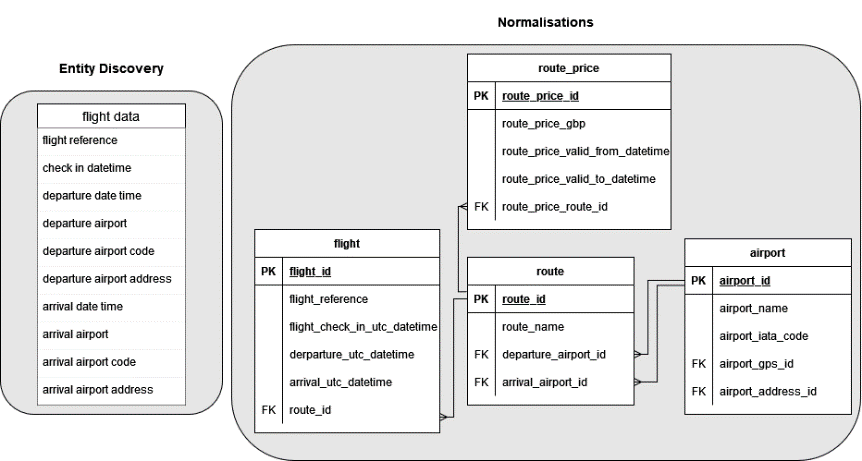
Flight tables

Figure Flight Data Normalisation

One of the most important sets of normalisations arose from considering the data storage requirements for a flight. During initial entity discovery I tried to record all the data visible on the website that was tied to a flight entity (with respect to the design assumptions laid out on the first page).

The table on the left in *figure 4* (flight data)is a list of all this data. To store multiple flights, it would be required to have much repetition of data in the form of repetitive airport information across many flight records. The data that describes the flight itself and the airport it flies to and from were normalised out into their own tables (flight and airport), and surrogate primary keys were added to enforce uniqueness of each record. While it would have been legitimate and possible to store route information as two foreign keys in the flight table (departure and arrival airport ids), early on I identified my preferred method of pricing a flight as to price by route, date and time, this information was further normalised out into route and route\_price tables, and surrogate primary keys added to both.

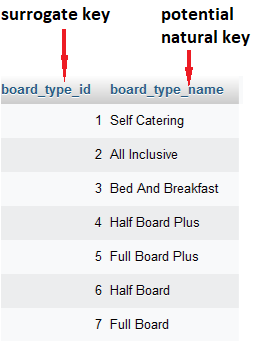
This left our flight tables normalised up to the 3rd normal form, and importantly adhered to the fundamental principles of normalisation in that it:

* freed the collection of relations from undesirable insertion, update and deletion dependencies, for example:
  + an airport may be updated or deleted with a single update/delete SQL statement to the airport table and this change would be propagated to all entities that use the changed record’s primary key as as a foreign key automatically.
  + the price of a route may be amended with a single update to the route\_price table.
  + The route of a flight may be amended at the route level with a single update to the route table, or at the flight level by changing the value of the route\_id foreign key in the route table.
* and made the relational model more informative to users

Asd

Asd

asd

**Primary Keys**

Throughout the project I chose to use exclusively surrogate primary keys. That is, PRIMARY KEY attributes of type INT(11) used to uniquely identify each record in a table by means of AUTO INCREMENT flag.

At several points I considered using natural primary keys. As in the example in *figure 4,* at first the board\_type\_name seemed like a good candidate for a natural primary key, but to ensure that this table’s design would enjoy a long lifespan, I chose a surrogate key.

Figure Natural Key Consideration in board\_type table.

I did so as I wanted to isolate the data being stored from its method of unique identification, so that it would remain trivial to edit the board\_type\_name without also violating foreign key constraints across all tables which potentially used board\_type\_name as a foreign key. This consideration was also apparent in several other entities containing only a single piece of data (for example language, email, card\_vendor, telephone\_type).

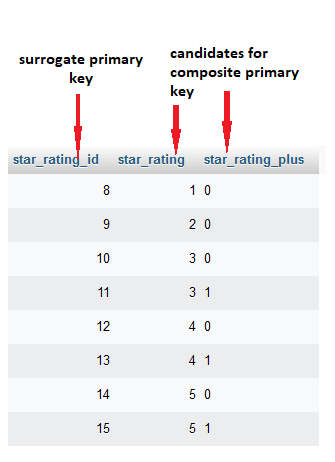
At several points I also considered using composite primary keys, one example is the the star\_rating entity (*figure 5*). On the website I noted that the star rating of a website was a combination of integer values (1-5 inclusive), with an optional “plus” attribute that may or may not be present for a given hotel. The plus was a Boolean value, and so I implemented it in our database as a non-null bit(1) type, allowing only true (1) / false (0) values to be present. Each star rating was therefore defined by a composite of its star\_rating integer and the boolean/ non-null bit(1) star\_rating\_plus value. Initially I intended to use star\_rating/ star\_rating\_plus as a composite primary key, but as in the considerations around board\_type above, I wanted to isolate the data itself from its unique method of identification, so the proposition of using this composite primary key was discarded for the use of a surrogate primary key. Although the star rating value was unlikely to be changed in the future, it seems prudent to consider the fact that perhaps in the future further subtypes or rating beyond the integer and bit value combinations may be added, and I did not wish to rely on the composite key for unique identification after these potential changes in the future.

Figure Composite Primary Key Consideration in star\_rating table

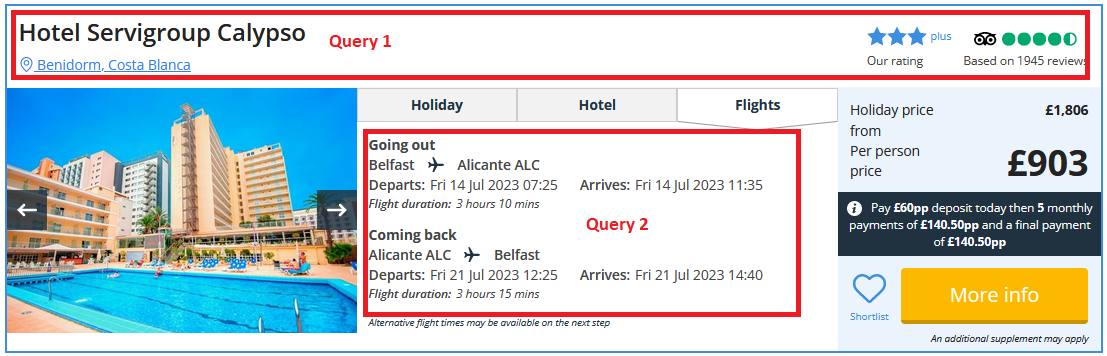
**Foreign Key Constraints**

**APPENDIX 1 : QUERIES**

Note Test data does not necessarily match website data

**Use Case - 1:**

**Generate the data visible on a hotel “card” on the website after using main search form**



**Use Case 1 – Queries 1:**

SET @HotelName = 'Spanish Hotel 1';

SELECT

  hotel.hotel\_name AS HotelName,

    region.region\_name AS RegionName,

    resort.resort\_name AS ResortName,

    star\_rating.star\_rating AS OurRating,

    star\_rating.star\_rating\_plus AS OurRatingPlusFlag,

    CAST(SUM(review\_rating.review\_rating) / COUNT(review.review\_id) AS DEC(2,1)) AS ReviewScore,

    COUNT(DISTINCT review.review\_id) AS NumberOfReviews

FROM hotel

INNER JOIN resort ON resort.resort\_id = hotel.hotel\_resort\_id

INNER JOIN region ON region.region\_id = resort.resort\_id

INNER JOIN star\_rating ON star\_rating.star\_rating\_id = hotel.hotel\_star\_rating\_id

INNER JOIN room\_type ON room\_type.room\_type\_hotel\_id = hotel.hotel\_id

INNER JOIN room\_booking ON room\_booking.room\_booking\_room\_type\_id = room\_type.room\_type\_id

INNER JOIN booking ON booking.booking\_id = room\_booking.room\_booking\_booking\_id

INNER JOIN booking\_contact ON booking\_contact.booking\_contact\_id = booking.booking\_contact\_id

INNER JOIN review ON review.reviewer\_id = booking\_contact.booking\_contact\_id

INNER JOIN review\_rating ON review\_rating.review\_rating\_id = review.review\_rating\_id

WHERE hotel.hotel\_name = @HotelName

GROUP BY hotel.hotel\_id;

**Use Case 1 – Results 1:**

Graphical user interface, application, website

Description automatically generated

**Use Case 1 – Queries 2:**

-- parameters gathered from jet2Holidays main search feature form

SET @OutboundDate = '2023-01-01';

SET @Duration = 2;

SET @HomeAirportId = 5; -- Belfast Internation Airport ID

-- parameter from website card shown above, changes for each card

SET @HotelId = 11; -- 'Spanish Hotel 1' Id

-- get route ids & compute return date

SET @AwayAirportId = (SELECT hotel.hotel\_serving\_airport\_id FROM hotel WHERE hotel.hotel\_id = @HotelId);

SET @OutboundRouteId = (SELECT route.route\_id FROM route WHERE route.departure\_airport\_id = @HomeAirportId AND route.arrival\_airport\_id = @AwayAirportId);

SET @ReturnRouteId = (SELECT route.route\_id FROM route WHERE route.departure\_airport\_id = @AwayAirportId AND route.arrival\_airport\_id = @HomeAirportId);

SET @ReturnDate = DATE\_ADD(@OutboundDate, INTERVAL @Duration DAY);

-- outbound flight info query

SELECT

  dep\_airport.airport\_name AS DepartureAirport,

    dep\_airport.airport\_iata\_code AS DepartureAirportCode,

    arr\_airport.airport\_name AS ArrivalAirport,

    arr\_airport.airport\_iata\_code AS ArrivalAirportCode,

    flight.departure\_utc\_datetime AS DepartureTime,

    flight.arrival\_utc\_datetime AS ArrivalTime,

    TIMEDIFF(flight.arrival\_utc\_datetime, flight.departure\_utc\_datetime) AS Duration

FROM flight

INNER JOIN route ON route.route\_id = flight.route\_id

INNER JOIN airport AS dep\_airport ON dep\_airport.airport\_id = route.departure\_airport\_id

INNER JOIN airport AS arr\_airport ON arr\_airport.airport\_id = route.arrival\_airport\_id

WHERE flight.route\_id = @OutboundRouteId LIMIT 1;

-- return flight info query

SELECT

  dep\_airport.airport\_name AS DepartureAirport,

    dep\_airport.airport\_iata\_code AS DepartureAirportCode,

    arr\_airport.airport\_name AS ArrivalAirport,

    arr\_airport.airport\_iata\_code AS ArrivalAirportCode,

    flight.departure\_utc\_datetime AS DepartureTime,

    flight.arrival\_utc\_datetime AS ArrivalTime,

    TIMEDIFF(flight.arrival\_utc\_datetime, flight.departure\_utc\_datetime) AS Duration

FROM flight

INNER JOIN route ON route.route\_id = flight.route\_id

INNER JOIN airport AS dep\_airport ON dep\_airport.airport\_id = route.departure\_airport\_id

INNER JOIN airport AS arr\_airport ON arr\_airport.airport\_id = route.arrival\_airport\_id

WHERE flight.route\_id = @ReturnRouteId LIMIT 1;

**Use Case 1 – Results 2:**





**Use Case - 2:**

**Demonstrate Time & Date Dependent Flight Pricing**

**Use Case 2 – Queries 1** *(ran once with each datetime variable below)***:**

SET @DateTimeToCheck = '2023-01-01 03:00:00';

-- SET @DateTimeToCheck = '2023-01-01 14:00:00';

-- SET @DateTimeToCheck = '2023-01-01 21:00:00';

-- SET @DateTimeToCheck = '2023-01-02 12:20:00';

SET @RouteId = 10; -- 'Spain to Belfast'

SELECT

  route\_price.route\_price\_gbp AS FlightPrice,

    @DateTimeToCheck AS FlightDateTime,

    route\_price.route\_price\_valid\_from\_datetime AS ValidFrom,

    route\_price.route\_price\_valid\_to\_datetime AS ValidTo FROM route\_price

INNER JOIN route ON route.route\_id = route\_price.route\_price\_route\_id

WHERE route\_price.route\_price\_route\_id = @RouteId

AND @DateTimeToCheck BETWEEN route\_price.route\_price\_valid\_from\_datetime AND route\_price.route\_price\_valid\_to\_datetime

AND route.route\_id = @RouteId;

**Use Case 2 – Results 1**

**Text

Description automatically generated with low confidence**

**Graphical user interface

Description automatically generated with medium confidence**

**Graphical user interface

Description automatically generated with medium confidence**

**A picture containing graphical user interface

Description automatically generated**

**Use Case - 3:**

**Hotel Reviews**

Graphical user interface, text, application, email

Description automatically generated

**Use Case 3 – Queries 1:**

SET @HotelName = 'Spanish Hotel 1';

SELECT

    hotel.hotel\_name AS HotelName,

    town\_city.town\_city\_name AS ReviewerTown,

    CountryName.location\_name AS ReviewerCountry,

    passenger.passenger\_first\_name AS ReviewerFirstName,

    passenger.passenger\_last\_name AS ReviewerLastName,

    DATE(review.review\_timestamp) AS ReviewDate,

    review\_rating.review\_rating AS ReviewRating,

    review.review\_content AS Review

FROM review

INNER JOIN booking\_contact ON booking\_contact.booking\_contact\_id = review.reviewer\_id

INNER JOIN passenger ON passenger.passenger\_id = booking\_contact.booking\_contact\_passenger\_id

INNER JOIN booking ON booking.booking\_contact\_id = booking\_contact.booking\_contact\_id

INNER JOIN review\_rating ON review\_rating.review\_rating\_id = review.review\_rating\_id

INNER JOIN address ON address.address\_id = booking\_contact.booking\_contact\_address\_id

INNER JOIN town\_city ON town\_city.town\_city\_id = address.town\_city\_id

INNER JOIN country ON country.country\_id = town\_city.town\_city\_country\_id

INNER JOIN location\_name AS CountryName ON CountryName.location\_name\_id = country.country\_location\_name\_id

INNER JOIN room\_booking ON room\_booking.room\_booking\_booking\_id = booking.booking\_id

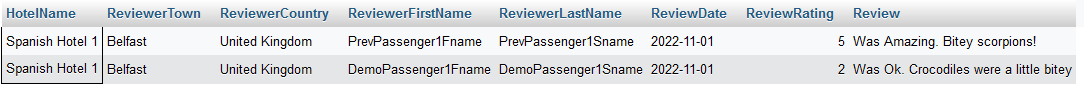
INNER JOIN room\_type ON room\_type.room\_type\_id = room\_booking.room\_booking\_room\_type\_id

INNER JOIN hotel ON hotel.hotel\_id = room\_type.room\_type\_hotel\_id

WHERE hotel.hotel\_name = @HotelName

GROUP BY review.review\_id;

**Use Case 3 – Results 1:**



**APPENDIX 2 : FINAL DRAW IO DIAGRAM**

**Put it here**