CS7052 Database Written Assessment

The intent of this project is to reverse engineer the front-end booking process of www.jet2holidays.com, using SQL and phpMyAdmin, so that a roughly equivalent process can be carried out by myself.

The first part of the project began with group work, originally we explored the jet2Holidays website. The intent was to become familiar with how exactly the booking worked from the user perspective so that an approximation could be carried out by the group, and then the individual, in SQL. Afterwards we began to discuss entities and their relations, as well as discussion of normalisation and ideas for foreign keys using draw.io to demonstrate this. As the lecturers progressed it also led to a progression of thought in what should be in the database. Originally the simple concept of normalisation and foreign keys were all that was discussed, but by the end of week 6 the ideas of order and joining, transactions and basic encryption had been included within the scope of the project.

The project focuses on keeping the data normalised and easily accessible, with intent to render as accurate as reasonable simulacrum of the booking website with reference to Peter Chen's entity-relationship model. His model was developed itself as a sort of unification of three prior data models, the network model, the relational model, and the entity set model.[1] Chen's model adopts a more natural view of the world and how entities and relationships are handled in them, and because of that "the model can achieve a high degree of data independence".[2] Therefore a database should be based around what is more realistic for the designer and user rather than what is easier to design. This project aims to demonstrate some of the challenges that emerge trying to handle data this way, whilst also showing its successes in displaying these relationships and the data within them.

[1]Chen, Peter, "The Entity-Relationship Model – Towards a Unified View of Data," *Special issue: papers from the international conference on very large data bases. Association for Computing Machinery.*, vol. 1, no. 1, pp. 9, September. 1975, doi: 10.1145/320434.
[2] Chen, Peter, "The Entity-Relationship Model – Towards a Unified View of Data," *Special issue: papers from the international conference on very large data bases. Association for Computing Machinery.*, vol. 1, no. 1, pp. 10, September. 1975, doi: 10.1145/320434.

Group Work

After the original work of reading through the website, discussion began on the scope of and what fell outside of it. We decided to leave specific holidays that jet2holidays offered, city breaks and Villa holidays, out as while they kept on the original jet2holidays site, they seemed to be covered by a different part of their website when interacted with. The ideas of car hire, and baggage handling were left outside the scope as they were not directly covered by jet2Holidays themselves and therefore would prove very difficult to implement. Therefore, after some discussion amongst the group and then with the lecturer we omitted them.

The concept of drawing up attributes and entities on draw.io proved particularly useful as it allowed the group to practice some normalisation, based upon Edgar Codd’s 1NF, 2NF and 3NF in particular. The website also allowed us a broad overview of what had been done, and what needed done. Each table that was drawn up with given its primary key, and a name that made obvious what the table was intended to do whilst also following a common naming convention for readability. In our group we chose to use snake case for both attributes and table names. The additional concern was to normalise down the tables, as especially at this early point repetition was quite possible.

The first major tables set up were the hotel, flight, destination, region, route, and guest tables with each obviously linking with one another. It was realised that addresses and flight had to

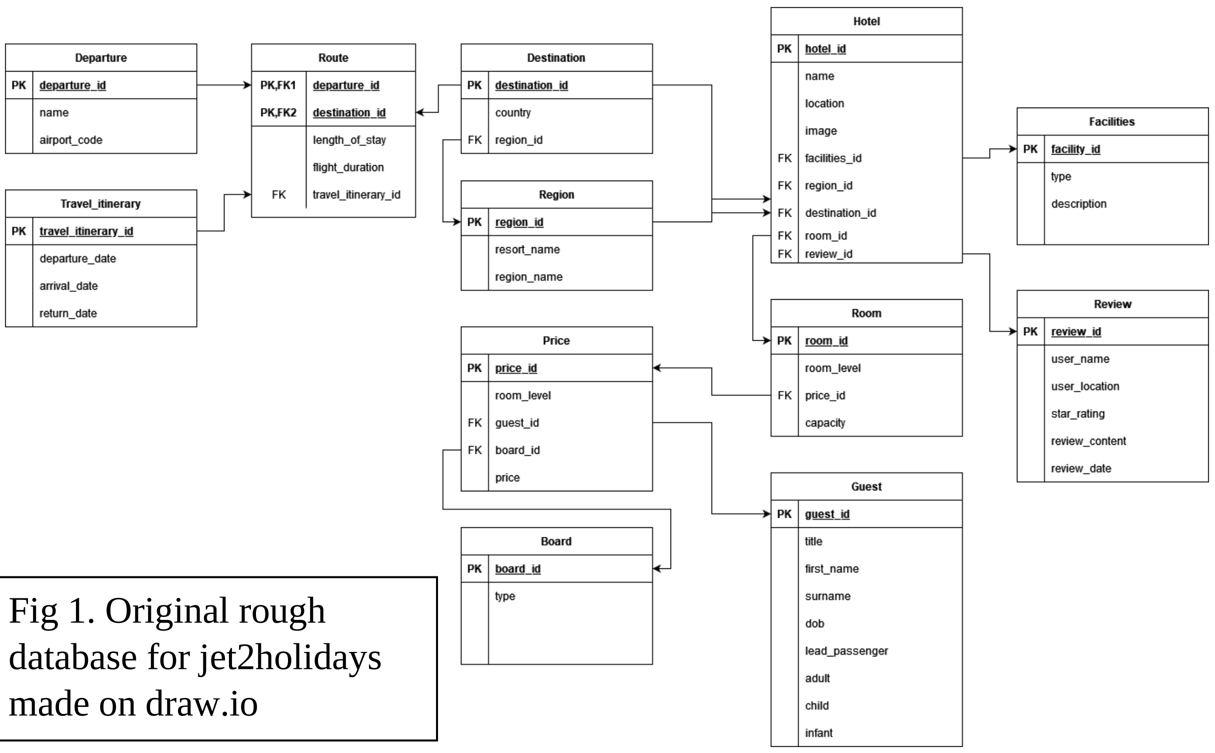


Fig 1. Original rough database for jet2holidays made on draw.io

be kept in separate tables because addresses would be used by other table. Other tables then were brought forward. This formative version is shown below in Fig 1.

As mentioned earlier when setting out the scope for the project, the database only covers the travel and stay aspect of a booking. However, as a rough design there were some serious inadequacies in both naming convention and overall structure of the database that had to be tackled. The most major one was the lack of an overall booking table which formed the heart of the many to many relationships that defined the database, a booking table would also allow us to run transactions later on. The departure table as well was also recognised to be a confusing naming convention, especially as it risked repetition in the flight table. The rough table, however, allowed the group to begin implementing some fundamental principles of database design. Following Codd's ideas, we recognised that we would need to order the tables of the database in such a way that redundancy is minimalised. As "redundant data wastes disk space and creates maintenance problems", it is quite important for overall efficiency that the rules of normalization are respected.[3]

In fig 1. the 1NF, First Normal Form, is already in action. Each table has a primary key, and said primary key exists to uniquely identify the tables and avoid repetition of the records within them. This avoids repetition of groups in separate tables. For example, the room table was split off from the hotel tables, as it was possible that two hotels could have identical room numbers. And what if another room was to be added? Instead, a separate room table could be referenced by the hotel table when it came to assigning it to a guest. The also kept the board table separate from room, as the group was aware that room tier was not necessarily linked to the board the guests selected. It was fully possible for a luxury suite to opt for half-board, for example. Therefore, repetition would be likely as copies of each room but with a different board-type would be in the same table, our decision eliminated this issue.

The 2NF, Second Normal Form, was also followed. New tables were created to ensure that no columns were dependent on candidate keys. This particular rule is the basis of the Foreign Key relationships that define databases. Instead of repeating the relevant information in each table that needs it, it can instead be asked for by the database using these relationships. For

[3] Microsoft Learn, "Description of the Database Normalization Basics." microsoft.com. https://learn.microsoft.com/en-GB/office/troubleshoot/access/database-normalization-description (accessed Nov. 24, 2022).

example, the relationships of the region table in fig 1 allows the region to be called by various tables such as the destination table or in the, later implemented, hotel table.

The 3NF, Third Normal Form, was also actively considered, in fig1. it can be seen used to separate the destination and region tables. But it was used across the whole database. Data inconsistencies in particular would be rife in the database without respecting the rule, references to regions and countries in a non-key fashion could prove problematic to change. As such foreign keys were implemented, ensuring that non-key attributes in the table referred to nothing but the key; extraneous information could then be separated into its own table with a primary key to index it. Such as the separation of the guest table from the room. Both are linked, but both contain information irrelevant to the other.

Many to many relationships and many to one relationships were also being implemented even at this early stage as we began to allow certain design decisions to take place. One to One relationships were ignored due to their rarity and rather selective use in databases. We decided that it should be possible for a passenger to book multiple rooms. However, issues

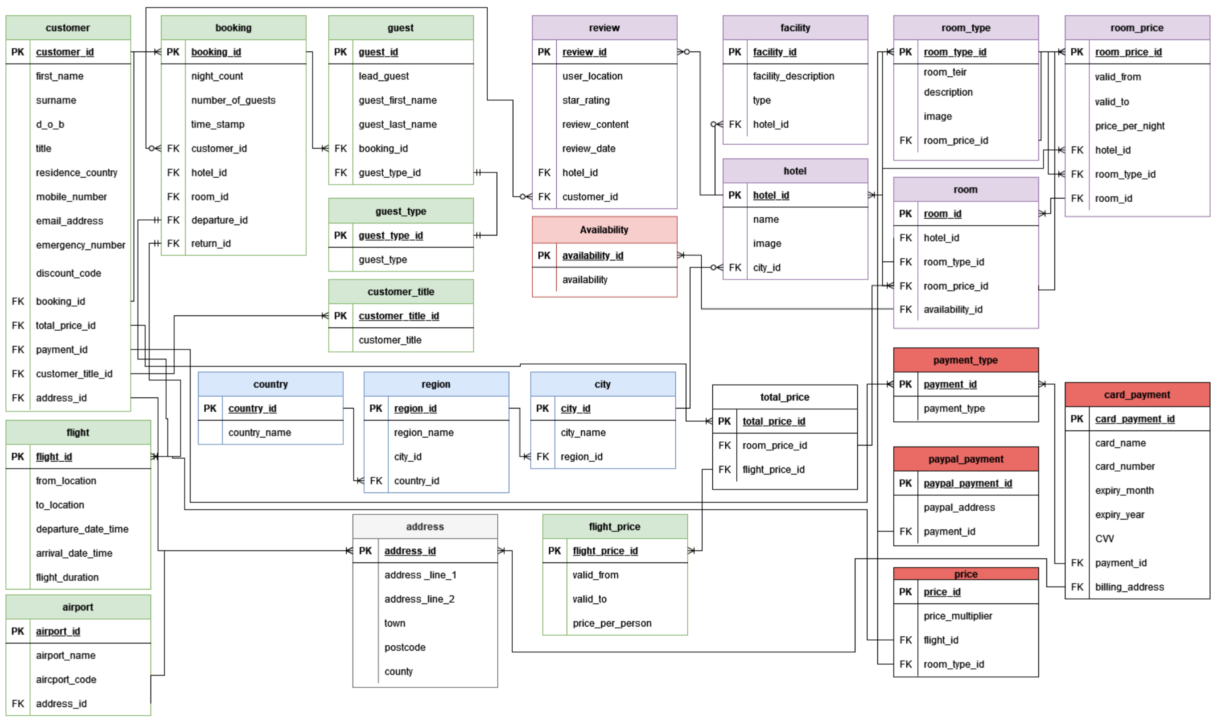


Fig 2. A more mature design on draw.io colour coded done on 18/10/22

remained around readability, so constant improvements were made until the final version.

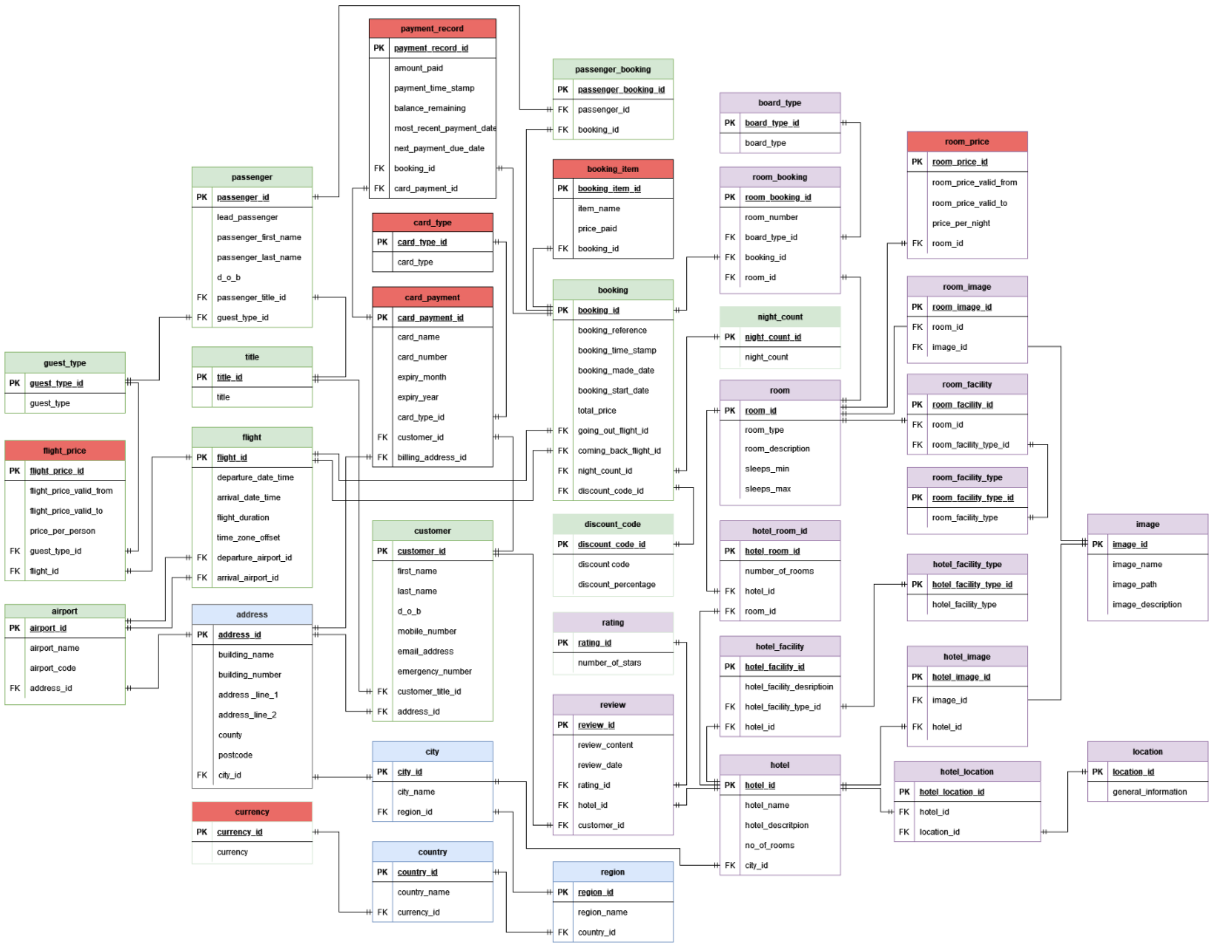


Fig 3. Draw.io diagram 18/11/22, the final version used as a reference point for phpMyAdmin

The design, fig 3. achieved through consultation with the lecturer and constant revisions, allowed the database to implement many of the ideas we had been discussing. Namely the idea of central heart in the booking table was realised as the database is structured around it and ultimately feeds back into it. We managed to overcome the issues of redundancy of having a departure and arrival airports, by instead having both departure and arrival airports foreign keys link to the one airport id. Some compromises were made, as we decided a customer could only book one room, and, despite jet2holiday's allowing PayPal, it was decided that we would only allow card payments. At this juncture, group work mostly ended apart from some discussion around SQL queries and our work became individual.

<u>Individual Work</u>

Individual work began on phpMyAdmin, as at this point draw.io had become a constraint on seeing how the table itself would function. Whilst draw.io could visually show FK, foreign key, relationships it could not show data being accessed from them. Personally, I had some disagreements with some attribute names that I changed whilst in phpMyAdmin, for example the room's choice of 'sleeps', based off jet2holiday's terminology, proved confusing to me so I used 'min' and 'max' occupants which was equivalent. I also chose to use the term 'guest' rather than passenger, as I felt it was a more appropriate term especially as the customer id was to be used in the review table. These changes were just for my own sake and for readability. I agreed with the group that Snake Case was preferable to Camel Case and used it for both attribute and table names. Whilst we had discussed somewhat as a group as to what datatypes should be used it was ultimately left to the individual to decide. Some major decisions made were to ensure that the card number was stored in varbinary, allowing it to be encrypted later. In addition in the customer table, had a 'lead_customer' attribute, which I chose to represent with a Boolean, tinyint(1), as it was a yes or no answer. I also chose to keep phone numbers as a varchar, rather than an int as the group itself had previously discussed. This was due to ints truncating the 0s which form the start of a UK phone number causing serious data inaccuracy.

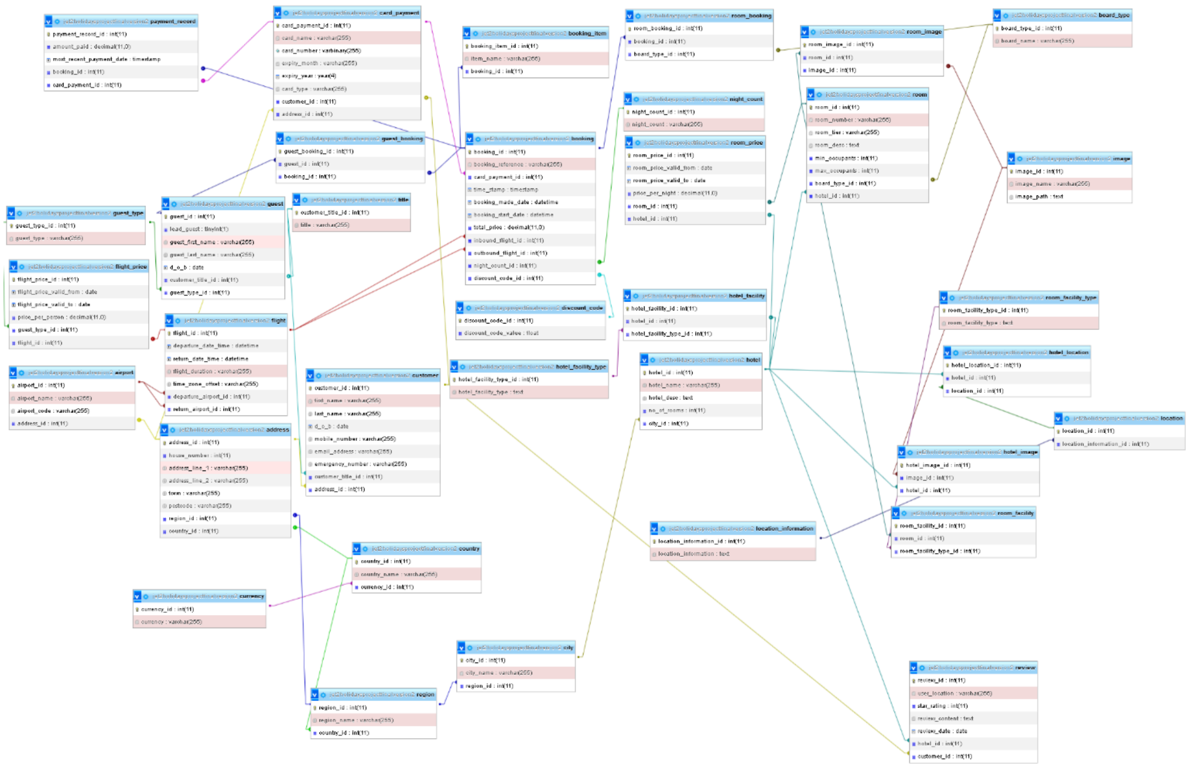
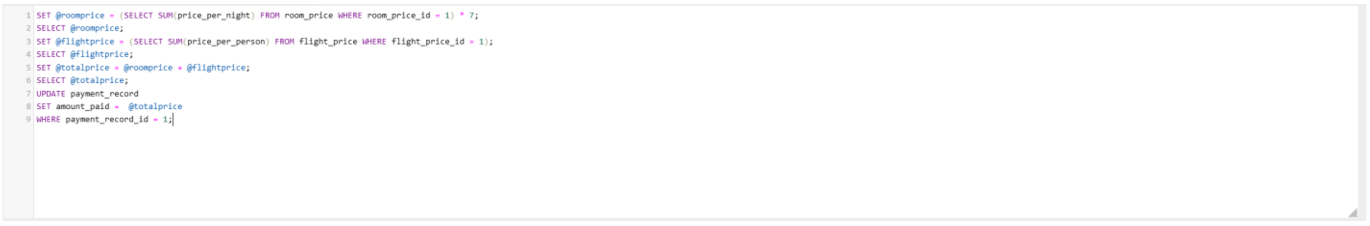
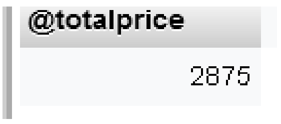


Fig 4. Entity relations shown through Designer view of database created in phpMyAdmin

The remaining and most difficult concern was trying to accommodate jet2holiday's pricing. I was decided what would be run was to run two SUM queries to gather than various prices from around the database and storing them as variables before updating the 'payment_record' table under 'amount_paid' for the particular customer selected using said variables to fill them. The 'amount_paid' was changed to decimal to allow it to express fractional costs. This result could then also be transferred over to the booking table as the total price, or manually filled out. This proved preferable to alternatives like manually filling out the table.



```sql
SET @roomprice = (SELECT SUM(price_per_night) FROM room_price WHERE room_price_id = 1) * 7;
SELECT @roomprice;
SET @flightprice = (SELECT SUM(price_per_person) FROM flight_price WHERE flight_price_id = 1);
SELECT @flightprice;
SET @totalprice = @roomprice + @flightprice;
SELECT @totalprice;
UPDATE payment_record
SET amount_paid =  @totalprice
WHERE payment_record_id = 1;
```



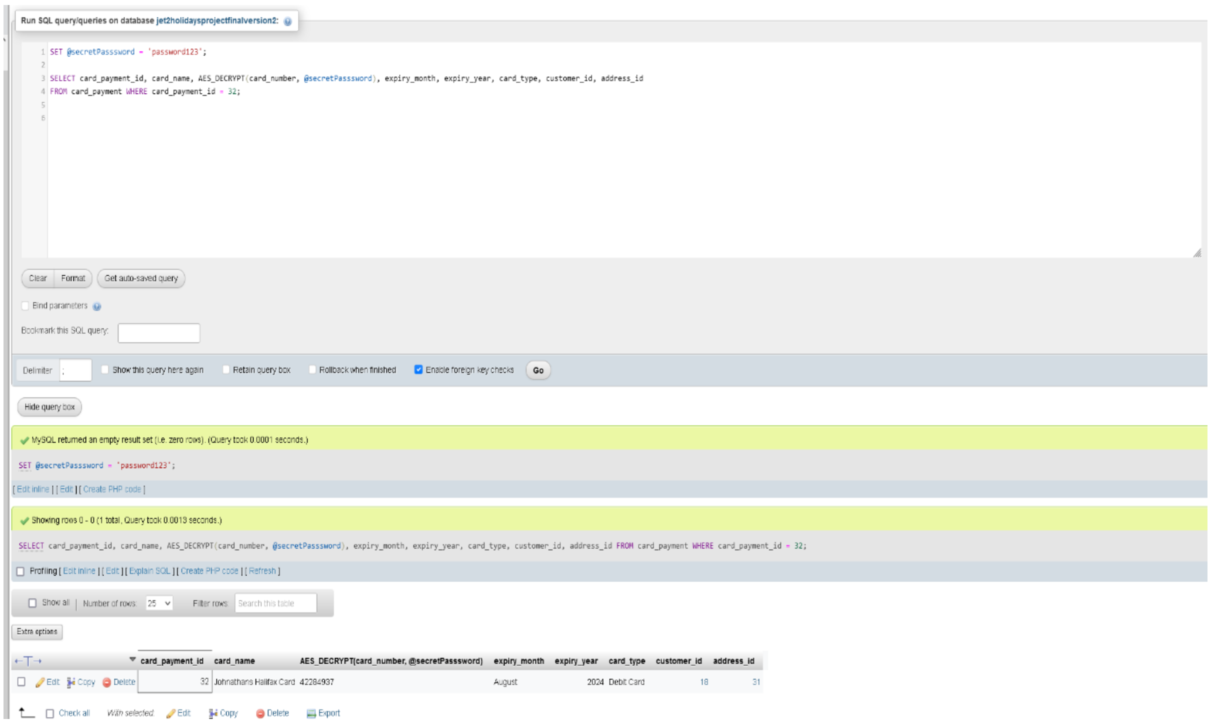
```
@totalprice
       2875
```

Fig 5. the query to set total price and insert it into the table, and proof of it working.

Encryption

The only record attribute in the database that we as a group recognised was worth storing encrypted was the card number, in the card payment table it was originally thought that a salt hash would be used to encrypt the card number. Obviously, considering the limited scope of SQL running natively off a machine, the method of encryption used is far too insecure to be actually used for true cybersecurity purposes. However, it would do for the purpose of the assignment to illustrate encryption could be done. Salt Hash would overcome this as it could not be decrypted but it abandoned, as the database would need the value at a later point. Instead, I opted to use AES_ENCRYPT.

The card number was stored in varbinary, as I discovered that int could not be successfully hashed. The plain password is set and then AES encrypt is used to combine it with the password to store a value. The values are then inserted into the table, and the password can then be used to decrypt it later as shown below. There are issues however with the use of the AES_ENCRYPT chiefly that were someone to get access to the instance of phpMyAdmin the server runs, then they would likely be able to get access to the password allowing them to decrypt the card numbers. In an actual implementation in a real-world scenario, external code would be used to secure passwords and specifically it is necessary that the storage of payment information be complaint with PCI, which dictates how card information is stored in to reduce fraud.[4]



Run SQL query/queries on database jet2holidaysprojectfinalversion2:

```
SET @secretPasssword = 'password123';

SELECT card_payment_id, card_name, AES_DECRYPT(card_number, @secretPasssword), expiry_month, expiry_year, card_type, customer_id, address_id
FROM card_payment WHERE card_payment_id = 32;
```

Delimiter ; — Show this query here again: ☐ — Retain query box: ☐ — Rollback when finished: ☐ — Enable foreign key checks: ✓

```
MySQL returned an empty result set (i.e. zero rows). (Query took 0.0001 seconds.)
SET @secretPasssword = 'password123';
```

```
Showing rows 0 - 0 (1 total, Query took 0.0013 seconds.)
SELECT card_payment_id, card_name, AES_DECRYPT(card_number, @secretPasssword), expiry_month, expiry_year, card_type, customer_id, address_id FROM card_payment WHERE card_payment_id = 32;
```

Number of rows: 25

card_payment_id	card_name	AES_DECRYPT(card_number, @secretPasssword)	expiry_month	expiry_year	card_type	customer_id	address_id
32	Johnathans Halifax Card	42284937	August	2024	Debit Card	18	31

Clear
Format
Get auto-saved query
Bind parameters
Bookmark this SQL query:
Go
Hide query box
[Edit inline] [Edit] [Create PHP code]
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]
Show all
Filter rows: Search this table
Extra options
Edit Copy Delete
Check all With selected: Edit Copy Delete Export

Fig 6. AES_decrypt command and proof of it working

Improvements

Whilst there were definite strong examples of following the ideas of normalisation and trying to keep the database compliant to them, there were several areas I would have liked to improve but could not. When it comes to the total calculation of cost, I was unable to get the room price to be affected by the 'night_count' and instead had to manually memorised, or find the value with an SQL query, so I could use the 'night_count' value to calculate the cost of a trip. In addition, it would have been useful to have information regarding other payment options such as PayPal although they were deliberately chosen to be excluded. Furthermore,

[4] IT Governance, "PCI DSS | What It Is and How to Comply." itgovernance. https://learn.microsoft.com/en-GB/office/troubleshoot/access/database-normalization-description (accessed Nov. 24, 2022).

it would have been preferable to have some way to have run a command that would have automatically applied a discount to the ‘total_price’ reflecting the discount they had rather than having to run the query manually.

Conclusion

In terms of trying to replicate the jet2holidays.com booking system, the project has been somewhat successful. The tables are structured with reference to Chen’s ideas and normalised based on Codd’, avoiding repetition and non-specific attributes. There was also demonstration of ideas of encryption as well as the database being able to be used to run arithmetic queries. There are definitely flaws, not just structural but also from a knowledge point of view however as it stands the database accomplishes what it was designed for. It allows a customer to book a flight to a resort in another country, then stores the information of the room they stayed in, how many nights they stayed and when they were returning whilst recording payment details which was what the database had originally set out to accomplish.

Appendix of SQL queries

Transaction to insert address:

```
SET @house_number = '12';

SET @address_line_1 = 'Beltway Avenue';

SET @address_line_2 = 'Dusky Road';

SET @town = 'Strabane';

SET @postcode = "BT45 XN2";

SET @region_id = 6;

Set @country_id = 2;

INSERT into address (address_id, house_number, address_line_1,
address_line_2, town, postcode, region_id, country_id)

VALUES (null, @house_number, @address_line_1, @address_line_2,
@town, @postcode, @region_id, @country_id);

SELECT * FROM `address`
```

Transaction to create Johnathan Long in Customer table:

```
SET @first_name = 'Johnathan';

SET @last_name = 'Long’;

SET @d_o_b = '1991-10-14';

SET @mobile_number = '07745289942';

SET @email_address = "johnnielong@gmail.com";

SET @emergency_number = '07840665231';

SET @customer_title_id = 1;

Set @address_id = 31;

INSERT into customer(customer_id, first_name, last_name,
d_o_b, mobile_number, email_address, emergency_number,
customer_title_id, address_id)
```

```
VALUES (null, @first_name, @last_name, @d_o_b, @mobile_number,
@email_address, @emergency_number, @customer_title_id,
@address_id);

SELECT * FROM `customer`
```

Transaction to create Johnathan Long in Guest table:

```
SET @guest_first_name = 'Johnathan';

SET @guest_last_name = 'Long';

SET @d_o_b = '1991-10-14';

SET @customer_title_id = 1;

SET @guest_type_id = 2;

INSERT into guest (guest_id, lead_guest, guest_first_name,
guest_last_name, d_o_b, customer_title_id, guest_type_id)

VALUES (null, 1, @guest_first_name, @guest_last_name, @d_o_b,
@customer_title_id, @guest_type_id);

SELECT * FROM `guest`
```

Inner join to get information on flight table and flight price:

```
SELECT *

FROM flight

INNER JOIN flight_price

ON flight.flight_id = flight_price.flight_id
```

Transaction and AES encryption in card table

```
SET @card_name = 'Johnathans Halifax Card';

SET @card_number = 42284937;

SET @expiry_month = 'August';

SET @expiry_year = 2024;
SET @card_type = "Debit Card";

SET @customer_id = 18;

Set @address_id = 31;

SET @secretPasssword = 'password123';

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

INSERT into card_payment (card_payment_id, card_name,
card_number, expiry_month, expiry_year, card_type,
customer_id, address_id)

VALUES (null, @card_name, @cardlongnumber, @expiry_month,
@expiry_year, @card_type, @customer_id, @address_id);

SELECT * FROM `card_payment`
```

AES decryption:

```
SET @secretPasssword = 'password123';

SET @secretPasssword = 'password123';

SELECT card_payment_id, card_name, AES_DECRYPT(card_number,
@secretPasssword), expiry_month, expiry_year, card_type,
customer_id, address_id

FROM card_payment WHERE card_payment_id = 32;
```

Transaction to insert into booking log:

```
SET @booking_reference = 'LON1483921';

SET @card_payment_id = 1;

SET @time_stamp = '2022-06-06 00:00:00';

SET @booking_made_date = '2022-06-02 00:00:00';

SET @booking_start_date = '2022-06-06 00:00:00';

SET @total_price = 0;
SET @inbound_flight_id = 3;

SET @outbound_flight_id = 3;

SET @night_count_id = 1;

SET @discount_count_id = 1;

INSERT into booking (booking_id, booking_reference,
card_payment_id, time_stamp, booking_made_date,
booking_start_date, total_price, inbound_flight_id,
outbound_flight_id, night_count_id, discount_code_id)

VALUES (null, @booking_reference, @card_payment_id,
@time_stamp, @booking_made_date, @booking_start_date,
@total_price, @inbound_flight_id, @outbound_flight_id,
@night_count_id, @discount_count_id);

SELECT * FROM booking
```

Transaction to create Johnathan Long's payment record:

```
SET @amount_paid = 0;

SET @most_recent_payment_date = '2022-09-11';

SET @booking_id = 14;

SET @card_payment_id = 31;

INSERT into payment_record (payment_record_id, amount_paid,
most_recent_payment_date, booking_id, card_payment_id)

VALUES (null, @amount_paid, @most_recent_payment_date,
@booking_id, @card_payment_id);

SELECT * FROM payment_record
```

Calculating total price

```
SET @flightprice = (SELECT SUM(price_per_person) FROM
flight_price WHERE flight_price_id = 3);

SELECT @flightprice;

SET @roomprice = (SELECT SUM(price_per_night) FROM room_price
WHERE room_price_id = 6) * 7;

SELECT @roomprice;

SET @totalprice = @roomprice + @flightprice;

SELECT @totalprice;

UPDATE payment_record

SET amount_paid =  @totalprice

WHERE payment_record_id = 18;
```

<u>Bibliography</u>

Chen, Peter, “The Entity-Relationship Model – Towards a Unified View of Data,” *Special issue: papers from the international conference on very large data bases. Association for Computing Machinery.*, vol. 1, no. 1, pp. 10-13, September. 1975, doi: 10.1145/320434.

Microsoft Learn, "Description of the Database Normalization Basics." microsoft.com. https://learn.microsoft.com/en-GB/office/troubleshoot/access/database-normalization-description (accessed Nov. 24, 2022).

IT Governance, "PCI DSS | What It Is and How to Comply." itgovernance. https://learn.microsoft.com/en-GB/office/troubleshoot/access/database-normalization-description (accessed Nov. 24, 2022).