ECM2419 DATABASE THEORY AND DESIGN CA

50/50 Split

Students: 670021141 & 670025221

User requirements specification (Part A)

Data requirements:

The hotel booking database will need to store information about various hotels and their customers. Beyond this, it will need to store information about the individual rooms in hotels, and the booking details of each customer. The overriding purpose of the hotel booking system is to enable customers to purchase bookings for available rooms in these hotels.

Our database will therefore need to store various attributes relating to each hotel. Examples include contact information (name, telephone), location (street, city, postcode), room pricing, discounts and facilities. These attributes consist of information relevant to potential customers, and therefore are a necessary user requirement. For example, a customer may decide to book a room based on the pricing, location or available facilities. Facilities are limited to: Free/Chargeable parking, Free WiFi, Air conditioning, Lift access, and Breakfast/Restaurant services. Also, all hotels in the database will provide a breakfast at their own price, so we'll need to store the breakfast prices for each hotel. Other pricing information we'll need to store includes the price of different room types (Single, Twin, Double and Family), and their corresponding discounts. These discounts depend on a grace period, which will also need to be stored in order to calculate if the discount applies or not. To uniquely identify different records of hotels, we will need to store a unique ID attribute for each hotel.

For rooms, we'll only need to store the room number and type. A customer will need to know what type the room is (e.g. if it's a family room or not) and the room number (so they can locate the room in person after making a booking). Room numbers will be positive integers, and room types will be enumerated with either Single, Twin, Double or Family types. Two hotels can have different rooms with the same room number, so we'll need a separate (unique) ID attribute to identify rooms across different hotels.

We will need to store information about the many customers who make bookings at these hotels. This information is relevant to the hotel owners (and to the booking/invoice), so is a relevant data requirement. Data needing to be stored includes contact details (title, first name, last name), contact details (telephone, email), address (street, city, postcode), and, if paying by card, the payment details (card number, expiration date). As with hotels and rooms, we'll need to store a unique ID attribute to identify different customers in the database. The customer email can be a secondary identification attribute as they are also unique, however can expire or be changed.

Finally, we need to store data about the actual bookings made by customers. This allows the hotel to manage which rooms are occupied at a given time, so they can successfully handle any queries regarding room availability. Information needing to be stored includes customer details, room details, hotel details, duration, guest details, payment methods and any special requirements. We also need to store whether guests will require breakfasts or not. Different bookings will need to be identified from each other, so we'll store a unique booking ID to allow the hotel to make a distinction between bookings in the database.

Transaction requirements:

Below are the detailed interactions a user may need to make with the database in order to use the system, sorted by transaction type. These interactions are outlined in the general case here, but in practise the SQL files will be tailored to a specific case. This being said, our SQL code could easily be modified into well-defined procedures that accept variable inputs.

Data queries

- Find hotels based on whether they have certain facilities, prices of types of rooms, public rating, location, and prices of breakfast.
- Find rooms that are available on certain dates.
- Find the average prices of the rooms in a hotel.
- Find the booking status of rooms on certain dates.
- Find the number of guests in hotels on certain days.
- Find the number of breakfasts ordered in hotels on certain days.
- Find the details of customers that have booked a certain room.

Data insertion

- Insert a new customer record.
- Insert a new hotel record.
- Insert a new room record, associated to an existing hotel.
- Insert a new booking record, associated to an existing customer and room.

Data modification

- Updating hotel information:
    - Add or remove facilities.
    - Change contact details.
    - Change public rating.
    - Change room prices for certain room types.
    - Change the discounts that can be applied to room prices.
    - Change the grace period for booking discounts.
- Removing a booking.

Data manipulation

- Calculate average room prices of different hotels on a certain date.
- Calculate total number of guests in a hotel on a certain date.
- Calculate total breakfasts ordered at a hotel on a certain date.
- Calculate the difference between booking and check-in dates.
- Calculate the price paid by a customer and any refund amounts.
- Calculate whether a discount should be applied and if so calculate cost of rooms with discount applied.

Conceptual model design (Part B)

Entity choice and reasoning:

Customer

We determined that *Customer* should be its own strong entity, with primary key *customerID*, for storing relevant information about hotel customers. This primary key is needed because customers names and addresses are not necessarily unique, so would not be suitable attributes for uniquely identifying each occurrence of the *Customer* entity. We chose to make these *name* and *address* attributes composite, as they are both composed of multiple components, each of which have an independent existence. The *Customer* entity also has an alternate key, its *email* attribute. This is an alternate key because an email address is unique to a given customer, so can be used as a fail-safe key to identify *Customer* entity occurrences.

Booking

We designed a (conceptual) strong entity, *Booking*, to store information relevant to room bookings in different hotels. It has primary key *bookingID,* because in the database there can exist two hotels with the same name. This way, we are able to uniquely identify the bookings of different customers, and the hotels they belong to. Other attributes relevant to bookings include guest information, payment options, any special instructions, and breakfast details. Being a strong entity, *Booking* is integral to our system design.

Room

Room is another strong entity, used to store information about rooms in different hotels. Corresponding hotel names and room numbers aren’t necessarily unique, because room numbers may be the same across different hotels (e.g. Hotel A and Hotel B both contain a Room 1), and two hotels can have the same name. Therefore, a more appropriate *roomID* primary key was chosen. Here we also made the design choice not to include a separate entity for unavailable rooms or maintenance, as the *AppearsIn* relationship handles this by storing *checkInDate* and *checkOutDate* attributes that determine whether a room is available or not. Another choice was to only store a *roomNumber*, rather than storing an extra attribute for floor numbers as well. This is because the floor number of a room can be derived from the *roomNumber* attribute, so doesn't need to be stored as its own attribute. The idea behind deriving these values rather than storing them as attributes is to help us control data redundancy.

Hotel

Our final strong entity, *Hotel*, is used to store information relevant to different hotels in the hotel management system. Two hotels can have the same name, so to avoid using a non-static primary key, we provide a more conventional *hotelID* attribute for the primary key. We made the *address* attribute composite, as it is built from three independent component attributes: *street*, *city* and *postCode*. We then provide an abundance of other attributes to describe features of a given *Hotel* occurrence, including room prices, discount prices, and various facilities. This is because each of these attributes are unique to a given hotel, so we made the design choice to include them all in one large entity. Alternatively, we could have provided weak entities *Facilities*, *Discounts* and *Prices*, providing a quaternary 1-to-1 *Has* relationship between these three entities and the *Hotel* entity. However, in practise we found this to be far more cumbersome and ultimately unnecessary (redundant joins), so decided instead to implement a single *Hotel* entity containing all attributes relevant to hotels in the system. For the design of this entity, we made the assumption that each hotel offered on the hotel booking website only has the capacity to provide the following facilities: Free/Chargeable parking, Free WiFi, Air conditioning, Lift access, and Breakfast/Restaurant services. This is because these were the facilities we extracted from the specification at the user requirements stage, and we both had agreed it was a reasonable assumption to make. Each of these facilities have a corresponding attribute in the *Hotel* entity, used to store boolean values denoting whether a given *Hotel* occurrence provides the facility or not. Another reason for making this assumption was to allow for users of the website to filter/sort search results by facility (assuming this is handled by a front-end application), allowing the client to provide a better customer service. However, in retrospect, this may prove problematic if the system needs to handle a hotel with new types of facility (e.g. swimming pool) in the future.

Relationship choice and reasoning:

Makes

There is a one-to-many relationship *Makes* between *Customer* and *Booking* entities. It has this multiplicity constraint because for a user of the website to become a customer, they need to have made one or more bookings, whilst each booking that gets made is specific to a single customer (the *leadGuest*). This also makes mandatory participation appropriate for both *Customer* and *Booking*, as neither entities can exist without an occurrence of the other. The relationship has a *bookingDate* attribute that stores the date a customer makes a booking. This attribute will be useful for generating invoices later.

AppearsIn

AppearsIn is another one-to-many relationship, this time between *Room* and *Reservation* entities. It has this cardinality because whilst a booking can only reserve one room, multiple bookings are able to reserve the same room (at different times). This 'at different times' constraint is enforced with the *checkInDate* and *checkOutDate* attributes of the relationship, which are used to keep track of how long rooms are booked for and to ensure rooms can't be booked twice in the same period. Here we made the design choice for each booking to reserve a single room (rather than a single booking reserving multiple rooms), as we thought it would provide a simpler implementation later down the line. Mandatory participation was appropriate for *Booking*, as a booking exists to reserve a room (so relies on the existence of a room), but participation is optional for *Room*, as not all rooms need to be booked.

Contains

There is a one-to-many relationship between *Hotel* and *Room* entities named *Contains*. It is one-to-many because a hotel can contain many rooms, but each room is only contained by one hotel. For obvious reasons, participation is mandatory for both entities.

Assumptions:

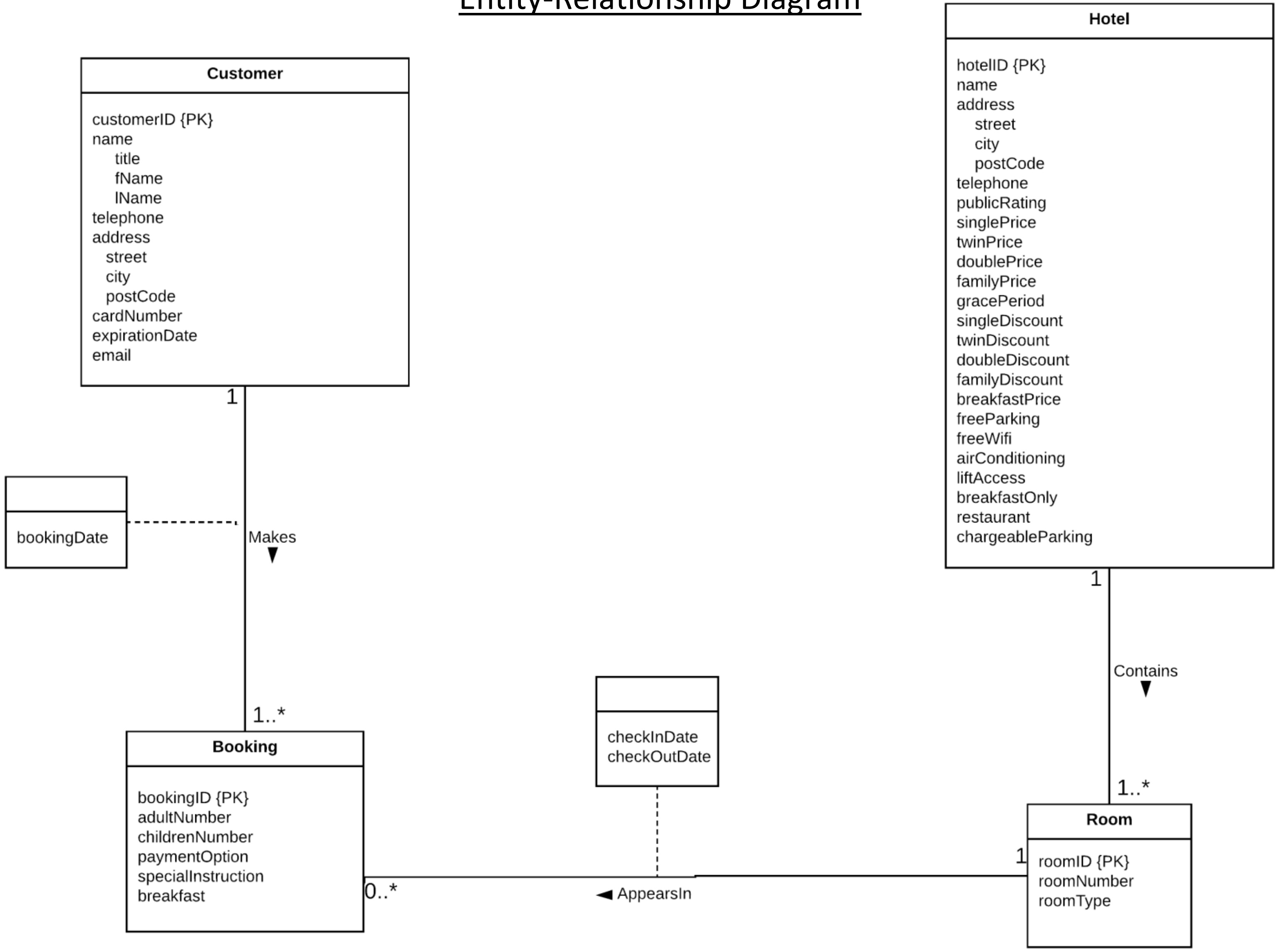
A few preliminary assumptions we made:

- Every hotel has at least one room.
- Each room number is at most three digits (i.e. a hotel can have up to nine floors).
- Discounts from grace periods will only apply to room prices, not breakfast prices.
- There is no age discount for breakfast prices. Adults and children will pay the same breakfast price.
- A room is booked for the entire duration of a booking. For example, if a room is booked from 26/12/18 to 30/12/18, it is occupied for all of these days (including the 26th and 30th). This is a reasonable assumption because after a customer has checked out (i.e. on the check out date) the hotel would need time to clean and maintain the room before the next customer arrives.
- For all guests in a booking, they either all have breakfast every day of their stay, or none have any breakfast at all for the duration of their stay. For example, it is not possible for a customer to book for three days, and have breakfast on two of those days but no breakfast on the third day.
- Formatting of data is handled by the client, and database design is unaffected. For example, database design conventions such as camel-case notation were used over more human-readable formats, because we worked on the assumption that all formatting is handled by a front-end application program.

Dynamic calculation:

Rather than hard-coding a separate *Invoice* entity, we went with the approach of dynamically calculating and generating the invoice upon request via queries. We chose to do this when analysing the user requirements of the invoice found in the specification, realising that all attributes of a potential *Invoice* entity could be derived from attributes in other entities. Therefore, storing an invoice in its own table would not only cause redundant data to be stored, but also potentially have significant drawbacks in terms of efficiency (assuming we are designing a scalable system). Beyond this, it keeps our design straight-forward and database flexible. However, the disadvantage of this design choice is that data queries regarding invoices can be very complicated. We make the assumption that a skilled developer would be able to query our database in whichever ways possible, and therefore would be able to generate invoices for an application program to handle and present.

Entity-Relationship Diagram



Logical model design (Part C)

Relational model

The ER diagram was transformed into the following relational data model:

Customer (customerID, title, fName, lName, telephone, street, city, postCode, cardNumber, expirationDate, email)

Primary key: customerID

Hotel (hotelID, name, street, city, postCode, telephone, publicRating, singlePrice, twinPrice, doublePrice, familyPrice, singleDiscount, twinDiscount doubleDiscount, familyDiscount, gracePeriod, breakfastPrice, freeParking, freeWifi, airConditioning, liftAccess, breakfastOnly, restaurant, chargeableParking)

Primary key: hotelID

Room (roomID, hotelID, roomNumber, roomType)

Primary key: roomID

Foreign key: hotelID references Hotel(hotelID)

Booking (bookingID, customerID, roomID, bookingDate, checkInDate, checkOutDate, adultNumber, childrenNumber, paymentOption, specialInstruction, breakfast)

Primary key: bookingID

Foreign key: customerID references Customer(customerID)

Foreign key: roomID references Room(roomID)

Physical model design (Part D)

Choice of RDBMS:

To implement our physical design of the hotel booking database, we chose to use *MySQL* as our relational database management system (RDBMS).

Here we had other options such as *Oracle*, *Microsoft SQL Server*, *IBM Db2*, *PostgreSQL*, and more. However, we chose *MySQL* because it is lightweight and open-source, unlike commercial RDBMS such as *Oracle*. It is therefore a very popular RDBMS that is widely-used across both academic and professional disciplines, supporting the back-ends of many large-scale systems used by organisations such as YouTube, Facebook and NASA. As a result, *MySQL* is very well-documented and has an active community of developers behind it, allowing us to easily implement our physical design (with DDL/DML). Commercial RDBMS (such as *Oracle*) can be very expensive and better-suited for large-scale business applications, so were an inappropriate choice for the purposes of this coursework.

Using an RDBMS allows us to control data redundancy, improve data integrity and ensure data consistency when implementing our physical database design. However, in practical applications (such as a hotel booking database), relying on a DBMS comes with several disadvantages. Namely, DBMS are general-purpose, complex pieces of software that can be expensive, large (occupying lots of disk space) and sometimes inefficient (not optimised for custom systems). Also, by providing a single point of failure, they introduce centralisation. This can ultimately increase system vulnerability; If the DBMS breaks so does everything else. However, the advantages of a DBMS tend to outweigh its disadvantages, as was the case for this assignment.

Data design:

For each of our tables, we use the *INTEGER NOT NULL AUTO_INCREMENT* constraint when defining the primary key attribute, ensuring that each entity occurrence has a unique value and can be identified. Almost all of our defined attributes feature type constraints followed by the *NOT NULL* constraint. These constraints enforce data integrity when new columns are inserted to a given table. However, we made exceptions to this in the *Customer* and *Booking* tables, with the *lName*, *cardNumber*, *expirationDate*, *adultNumber* and *childrenNumber* attributes. Giving these the *NULL* constraint allows us to implement room maintenance, by considering the maintenance company as a type of customer and making a booking for the room with a dummy customer. This ensures a room can only be booked by (real) customers if it isn't undergoing maintenance. This implementation choice doesn't interfere with any other queries, and therefore kept our physical database design simple.