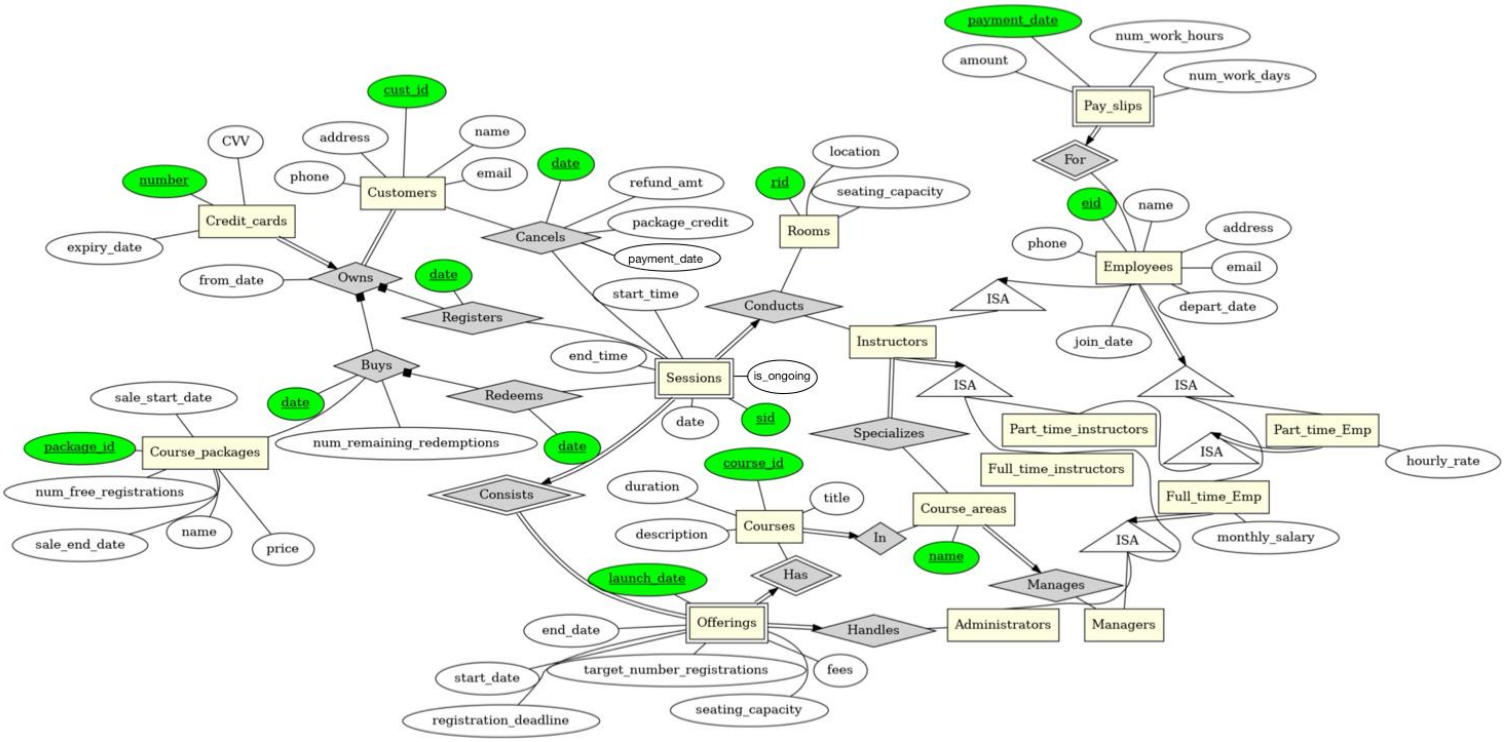
CS2102 Project Report
(Team 38)

Name	Student Number	Responsibilities
He Jialei	A0205103A	- Functions (Lead) - Testing - Test Data - Schema
Tan Xi Zhe	A0200766J	- Functions - Testing - Test Data - Triggers (Lead)
Teh Xue Yong	A0205825H	- Functions - Triggers - Testing - Schema (Lead)
Travis Toh Guan Wei	A0208026N	- Functions - Testing - Test Data (Lead) - Schema

ER Data Model



Justification for non-trivial ER designs

1. CourseOfferings as a weak entity to Courses, and Sessions as a weak entity to CourseOfferings

   CourseOfferings is made a weak entity set with Courses as the owner entity. This is because CourseOfferings lacks a unique identifier, and each CourseOffering can be identified by a composition of its Course, and its launchDate. Likewise, Sessions has session numbers which are only unique within their own offerings. Hence it is a weak entity set dependent on Courses.

2. Hierarchy of Employees

   To accurately capture how each employee must either be a Full-time employee or a Part-time employee, we added the covering constraint on employees. Also, each Full-time employee must be a Manager, Administrator, or Full-time Instructor so we added the covering constraint to this Full-time-emp. Similarly, Instructors must either be a Full-time Instructor or Part-time Instructor so we added the covering constraint to Instructors.
   To ensure that Full/Part-time Instructors are both Instructors and Full/Part-time employees respectively, we referenced both superclasses.

3. Detaching credit card information from Customers and Aggregation of Owns to Buys and Registers

   We modelled Credit_card as an entity using the Owns relationship to relate to Customers as the customer might change credit cards over time. This change allows us to keep track of which card the customer used to pay for their Registers or Buys by aggregating the Owns relationship.

4. Aggregation of Buys relationship to Redeems

   To keep track of the number of remaining redemptions and the purchase date of the customer's course package, we used a buys table. Afterwards, we aggregated the Buys relationship to Redeems to keep_track of which (partially) active package the redemption is from.

5. Total and key participation of Courses and CourseAreas

   To capture "Each manager manages zero or more course areas, and each course area is managed by exactly one manager. Each course offering is managed by the manager of that course area.", we decided to model a Manages relationship between Managers and CourseAreas. As CourseOfferings is a weak entity with Courses as its owner entity, it would be sufficient to just show the Managers entity managing the CourseAreas, as this would imply that the manager manages the course offerings offered by the course. Furthermore, there is a key and total participation constraint on Courses in the In relationship, and also a key and total participation constraint on CourseAreas in the Manages relationship, which ensures that every Course (and in turn its course offerings) would be managed by a manager that manages its course area.

6. Cancels relationship between Customers and Sessions

   We chose to have a relationship between Cancels and Sessions because when a Customer updates their Credit_card, it might be due to reasons such as their card being replaced. As such, we would refund the cancellation to their active card. As for redemptions, a Customer would not be able to purchase a new package if there is a partially active package, and thus, we would be able to refund the package_credit back to the correct Buys entry.

7. Ternary relationship between Sessions, Rooms, and Instructors

   To capture the constraint "each session is conducted by an instructor on a specific weekday (Monday to Friday) at a specific hour in some lecture room within the company", we model the relationship between Sessions, Rooms, and Instructors as a ternary relationship. This allows us to ensure that for each Conduct, all three of the mentioned entities must be present.

Constraints not captured by ER model

1. Each course offering has a start date and an end date that is determined by the dates of its earliest and latest sessions, respectively.

2. There must be at least one hour of break between any two course sessions that any instructor is teaching.

3. An instructor who is assigned to teach a course session must be specialized in that course area.

4. Each customer can have at most one active or partially active package.

5. For each registered course, a customer pays for the course fees by either making a credit card payment or redeeming a session from his/her active course package.

Relational Database Schema

Introduction:

In this section, we will be justifying the non-trivial decisions we made when designing our Relational Database Schema. Under certain pointers, we will be briefly mentioning the triggers we implemented as these triggers are meant to enforce certain constraints on our schema.

Justification for non-trivial Relational Database Schema designs:

1. Owns as part of Credit_cards table

   If we implemented Owns using a separate table, we would not be able to capture the total participation and key constraint on w.r.t to Owns. Thus, we combine Owns and Credit_cards into one table. We added the cust_id attribute to Credit_cards and the "NOT NULL" constraint on this attribute to ensure Key and Total Participation on Credit_cards. Our primary key is only Credit_cards as we do not think that multiple customers should be allowed to own the same Credit_card.

2. Inclusion of attribute is_ongoing as a valid bit in Sessions

   To ensure that the session numbers for the sessions are numbered consecutively starting from 1, we added a valid_bit "is_ongoing" to denote if a session has been removed. However, this design consideration cannot be fully captured by the schema, and would require the support of a trigger. A more detailed justification of this point will be recorded under the first point of the "Difficulties Encountered and Lessons Learnt" section.

3. Inclusion of attribute payment_date into Cancels

   To ensure that the right card is refunded, we added payment date into the entity Cancels. This addition allows us to keep track of whether a user is refunded through the difference in dates, and the amount that is refunded. As such, we imposed table constraints to ensure that a customer must only be refunded either money, or credit.

4. Usage of TIMESTAMP instead of DATE in derived attributes requiring date within Credit_cards, Cancels and Buys

   For Credit_cards, the customers might update their credit_cards multiple times a day. As sql tables are unordered, we would require a more precise attribute to keep track of the active card. Thus, we would use the TIMESTAMP attribute to determine the active card instead.

   For the table Buys and Cancels, we decided to change to TIMESTAMP as it would provide us the flexibility of knowing the sequence of purchases and cancellations for our bookkeeping purposes.

5. The logic flow of cancelling a registration

   When a registration is cancelled, we would insert the corresponding row into Cancels, and remove the registration from the Registers table. If the registration is paid for using a redemption, the corresponding entry in redeems will be removed as well.

   To add on, we implemented Cancels as a separate table to capture the many-to-many relationship between the Customers and Sessions. This allowed for each customer to cancel multiple sessions, and each session to be cancelled by multiple customers.

6. Allow for eid and Offerings identifier (course_id, launch_date) to be deferrable

   In routine 10 add_course_offering, we would perform first an iterative depth-first search to find a set of valid assignments of instructors to each session, before inserting the course offering. In our routine, we will first set these attributes as deferred as a session cannot exist without a course offering. Only at the end of the search, the course offering is inserted and the deferred attributes are updated.

7. Ternary relationship Conducts as part of Sessions table

   If we implemented Owns using a separate table , we would not be able to capture the total participation and key constraint on Sessions w.r.t to Conducts. Thus, we combine Conducts and Sessions into one table. We added the attributes eid and rid to Sessions and the “NOT NULL" constraint on these attributes to ensure Key and Total Participation on Sessions. Our primary key in Sessions do not contain the primary keys of Rooms and Instructors, as the session identifier (course_id, launch_date, sid) should be unique; adding eid and rid into the primary key will allow the table to have duplicate entries for a session as long as they occur in different rooms and/or have a different instructor conducting it.

8. Double reference for the eid attribute of Part-time Instructors and Full-time Instructors

   To capture the aspect of subclass entities Part-time Instructors and Full-time Instructors belonging to 2 superclass entities, we designed the attribute eid to reference both Instructors, and Part_time_emp/Full_time_emp respectively. Modelling these subclasses as such ensured the consistency between these tables without requiring any additional table/column constraints.

9. Manages as part of Course_areas table

   If we implemented Owns using a separate table, we would not be able to capture the total participation and key constraint on Course_areas w.r.t to Manages. Thus, we combine Manages and Course_areas into one table. We added the eid attribute to Course_areas and the “NOT NULL” constraint on this attribute to ensure Key and Total Participation on Course_areas. Our primary key in Course_areas remains as

name as each Course_area is managed by exactly one Manager.

10. Usage of serialised integers in identifiers that are supposed to be generated by the system

    We utilised SERIALs in our schema to generate the integer identifiers of Courses, Course_packages, Customers, Employee, and Rooms. This design encompasses consecutive numbers, identifiers starting from one, together with ignoring of deleted values.

11. The lack of an explicit ON UPDATE Foreign Key Constraint for attributes referencing serialised integers

    As the primary keys of the REFERENCED tables are serialised, they should not be amended to prevent conflict. As such, we will use the default NO ACTION constraint to prevent any unintended changes to the primary keys of the REFERENCED tables. The REFERENCING tables affected by this are Part_time_emp, Full_time_emp, Instructors, Part_time_instructors, Full_time_instructors, Administrators, Pay_slips, Course_areas, Courses, Offerings, Sessions, Cancels, Credit_cards, Buys, Registers, Specializes, and Redeems.

12. Table on Column Constraints

    To capture as many of the application's constraints as possible without the use of triggers, we implemented a series of named constraint checks in our schema. For most FLOAT and INTEGER values, we added constraints to ensure that values inserted are either non-negative or positive depending on the attribute. We also performed constraints across multiple attributes in tables when dates are involved to ensure that constraints such as the deadline must be at least 10 days before the start date would be captured. For tables such as Cancels and Pay_slips that can possibly contain a null value depending on another column, we performed checks to ensure that unintended values would not be inserted into the tables.

13. Registers a separate table

    To capture the many-to-many relationship between the aggregated entity Owns and Sessions, we implemented Registers as a separate table. This allowed for each customer to register for multiple sessions, and each session to host multiple customers.

14. Redeems as a separate table

    To capture the many-to-many relationship between the aggregated entity Buys and Sessions, we implemented Redeems as a separate table. This allowed for each customer (who can only own one active or partially active package) to redeem multiple sessions, and each session to host multiple customers who paid by redemption.

15. For as part of Pay_slips table

    If we implemented For using a separate table, we would not be able to capture the total participation and key constraint on Pay_slips w.r.t to For. Thus, we combine For and Pay_slips into one table. We added the eid attribute to Pay_slips and the “NOT NULL” constraint on this attribute to ensure Key and Total Participation on Pay_slips . Our primary key in Pay_slips contains both eid and payment_date as each employee can be paid more than once, and more than one employee can be paid on each payment_date.

Constraints not enforced by the Schema:

1. Total participation constraint on Customers in the Owns relationship.

2. Total participation constraint on Instructors in the Specializes relationship.

3. No two sessions for the same course offering can be conducted on the same day and at the same time.

4. Every Employee must be a full-time employee or a part-time employee. Every full-time employee must be an administrator, a manager or a full-time instructor, every part-time employee must be a part-time instructor.

5. For each course offered by the company, a customer can register for at most one of its sessions before its registration deadline.

Triggers

Introduction:

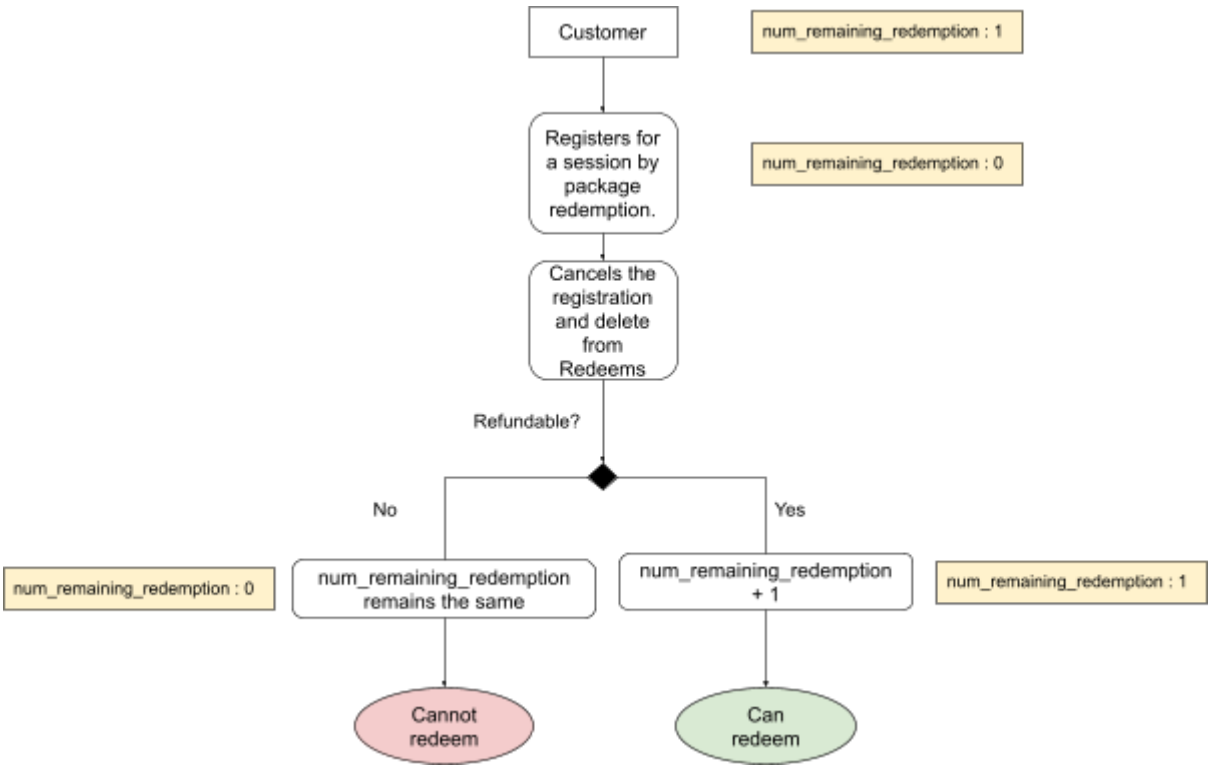
To ensure that the explicit and implied constraints in the Application are fully captured, we extensively employed the use of triggers; many of the functions we implemented do not have data validation or checks as our triggers will perform the check when the function attempts to insert/delete/update the tables. We implemented a total of 24 sets of triggers for the application; some triggers perform more than 1 check, and certain sets of triggers have more than 1 trigger in it.

Three most interesting triggers:

Trigger 1

Name : refund_redemption_trigger
Table : Cancels
Usage :
If the cancelled registration was paid by redemption from the course package, the corresponding redemption record in the Redeems table will be deleted. If the cancellation is valid for a credit back to the customer's active course package, the value of num_remaining_redemption in the corresponding record in Buys table will be incremented by one.
Design Justification :
When a registration paid by course package redemption is cancelled, we should increment the number of remaining provided the redemption can be credited back. After that, we can register for another session by redeeming the refunded package redemption.

- Example Illustration:



Trigger 2

Name : emp_del_triggerX, $1 \leq X \leq 8$
Tables : Employees, Part_time_emp, Full_time_emp, Instructors, Part_time_instructors, Full_time_instructors, Administrators, Managers
Usage :
Avoid users from manually deleting employees from the employee-related tables.
Design Justification :
To allow for bookkeeping of the entities Sessions and Offerings, we have to keep the employee in the record after we have removed the employee. For example, doing so will enable us to find out who is the instructor of a session in the past even if the instructor has left the company. Therefore, we enforce this trigger to make sure that users can only remove employees with the given remove_employee() procedure, which will update the employees' departure date without removing them from the record. Furthermore, this trigger prevents the covering constraint from being violated as it would be impossible to remove an entry from Managers, or Administrators, or Part_time_emp, or Full_time_emp or Part_time instructors, or Full_time_instructors without "removing" the entry from Employees.

Trigger 3

Name : payslip_validation_trigger
Tables : Pay_slips
Usage :
Check the amount of salary before inserting into Pay_slips.
Design Justification :

Before inserting into Pay_slips, the trigger will check if the amount of salary is correct given the working record of that employee. Based on the employee's status(full time or part time), the amount of salary can be calculated differently.

- Full time employee :
    - $Salary\ of\ the\ month = monthly\ salary \times \frac{number\ of\ work\ days\ for\ the\ month}{number\ of\ days\ in\ the\ month}$
- Part time employee :
    - $Salary\ of\ the\ month = hourly\ rate \times number\ of\ work\ hours\ for\ the\ month$

With the equation given above, the trigger can check if the amount of salary inserted into Pay_slips is correct. It will reject the insertion if the amount does not align with the work time recorded.

Difficulties Encountered and Lessons Learnt

1. **Difficulty in bookkeeping when deleting sessions**

   When deciding the best way to implement a sequence on Sessions that is dependent on (course_id, launch_date), we had to consider what happens if a session is deleted, and the issue of registers and cancels referencing the Sessions table. Hence, we identified 3 possible solutions to this issue:

   In the beginning, we attempted to simply delete the session and move all registrations to Cancels. This solution would be simple to implement with a trigger, although it came with numerous downsides — we do not know whether we should refund the customers if it is within 7days, and it would also be hard to determine whether we should skip a sid as it belongs to a deleted session. Thus, we came out with the second idea.

   For our second idea, we considered using a separate table to contain the number of sessions for each offering; the table would contain the offering identifier and the total number of sessions. However, this raised the issue of redundancy, and it might also have synchronisation issues when there are concurrent connections.

   Thus, we finally settled on the idea of having a valid bit that is by default ‘true’ to identify if a session is ongoing. We added a trigger that would prevent deletion from the Sessions table, and instead set the valid bit to ‘false’ instead. The benefit of this implementation is that the Cancels table could still refer to the corresponding session which improves the bookkeeping. Additionally, we would not face the issue of newly added sessions having the same sid as a deleted session without the use of a separate table.

2. **Unable to ensure Cancels table only contains cancelled sessions**

   When we attempted to implement a trigger that checks if an insertion into the Cancels table is due to the cancellation of a registration, we had to ensure that a corresponding entry in the Registers exists before the insertion, and is deleted after the insertion. As such, we learnt how to implement triggers in sets to perform checks on the same table before and after an insertion. Before the insertion, we would check if the entry corresponding to the NEW Cancels exists, and after the insertion (deferred), we would check whether the entry is removed from the Registers table. This allowed us to prevent bogus entries from being inserted into Cancels.

3. **Difficulties in implementing a complete search function to find a set of valid instructor assignments**

   Routine 10 searches for a set of valid instructor assignments using depth first search. In each step, we find the available instructors with find_instructors routine. This is

essentially the branches from the current node. A session is then matched with an instructor and inserted in where triggers are responsible for checking if the session is valid (E.G. if they are consecutive, if the room is available for the session). If there are no available instructors, we then revert the insertion and try the next closest permutation.

Usually, DFS is implemented in a recursive manner. However, we have to defer our triggers that do the checkings in the intermediate steps. We realised that triggers are fired whenever the recursive function returns. Hence, we had to change it to the iterative version of DFS with an array of integers to keep track of the search's progress.

Through this routine, we better realise the limitations of a declarative language like SQL. In this case, the assignment of one instructor to a session affects another assignment. Hence, there is no easy definition of the goal state but we do know how to get to that state. This does not bode very well with SQL where defining how is often difficult.