National University of Singapore

CS2102 Database Systems

Project Report

Project Team 61

Chen Anqi (A0188533W)

Chen Su (A0188119W)

Gu Yichen (A0204735J)

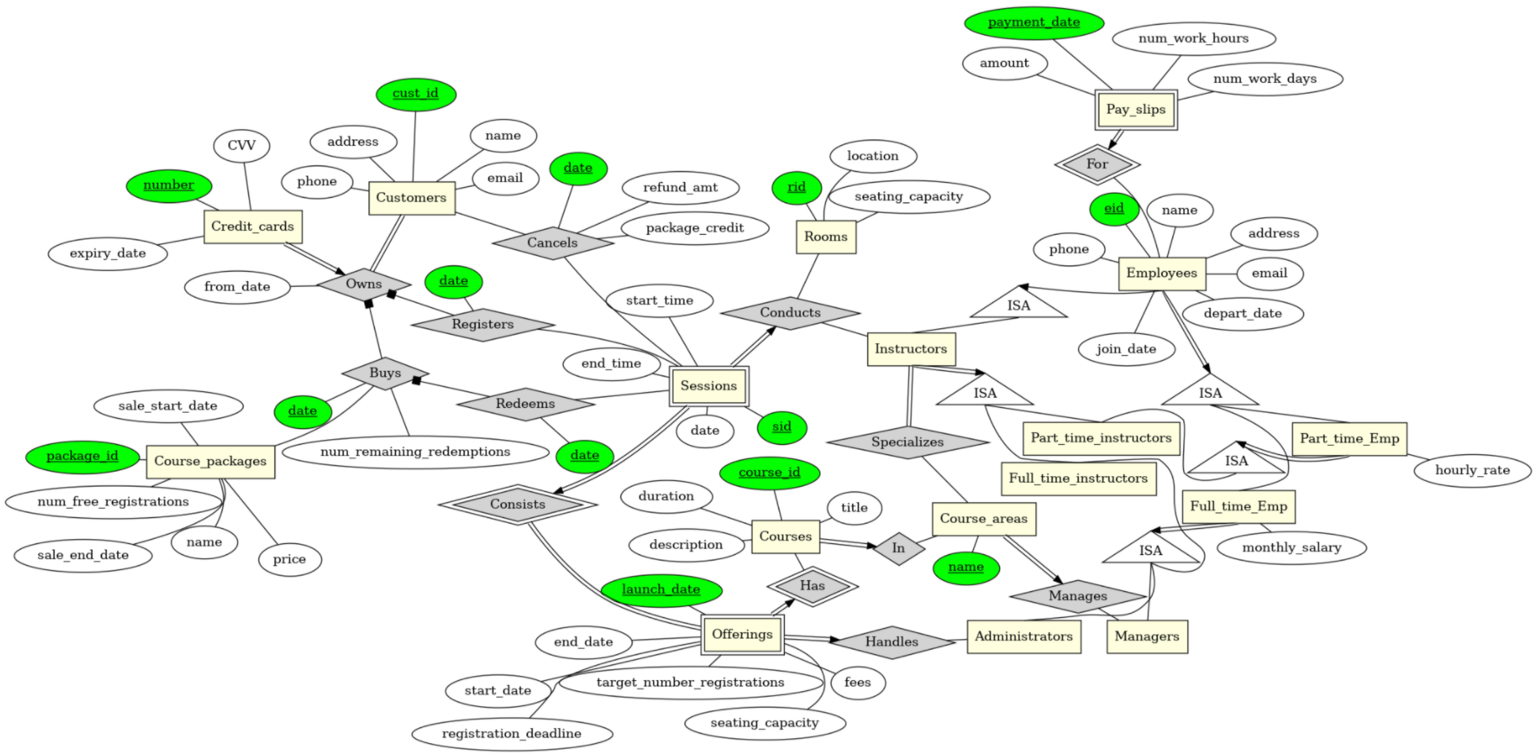
Shao Yufei (A0206427J)

1. Project Responsibilities

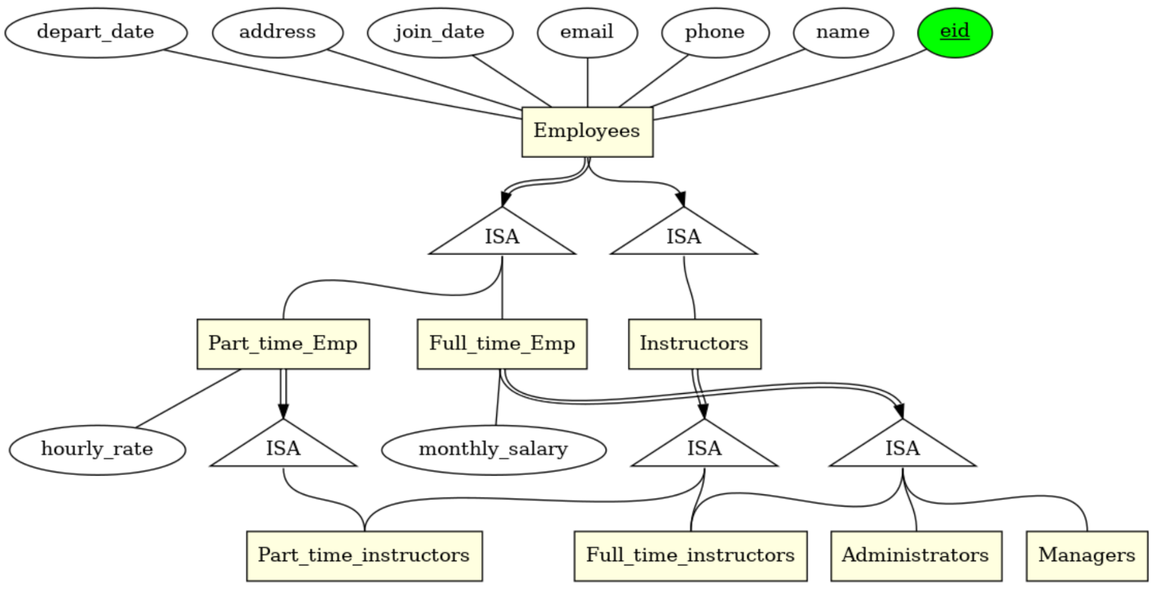
Week	Chen Anqi	Chen Su	Gu Yichen	Shao Yufei
6	Discussed and completed ER model together			
7 & 8	Discussed and wrote schema.sql together			
9	Implemented functionalities 1-6	Implemented functionalities 7-12	Implemented functionalities 13-18	Implemented functionalities 19-24
10	Modified functionalities 1-6 Wrote program to automate generation of `INSERT INTO` statements for populating data	Modified functionalities 7-12 Populated sample data for testing purpose	Modified functionalities 13-18 Populated sample data for testing purpose	Modified functionalities 19-24 Populated sample data for testing purpose
11	Implemented functionalities 25-26 Testing on functionalities 1-6 Wrote triggers to enforce constraints related to `Employees`, `Customers`, `Credit_cards`	Implemented functionalities 27-28 Testing on functionalities 7-12	Implemented functionality 29 Testing on functionalities 13-18 Wrote triggers to enforce constraints related to `Redeems`, `Registers`, `Cancels`, `Buys`	Implemented functionality 30 Testing on functionalities 19-24 Wrote triggers to enforce constraints related to `Redeems`, `Registers`, `Cancels`, `Sessions`
12	Testing on Yichen's code Write report on allocated parts	Testing on Yufei's code Write report on allocated parts	Testing on Anqi's code Write report on allocated parts	Testing on Chen Su's code Write report on allocated parts
	Group debugging & populating data for demo purpose			

2. Entity-Relationship (ER) Data Model

We decide to use the ER model proposed by the teaching team.



The ISA hierarchy for `Employees` is shown more clearly in the figure below.



Comments (Adapted from the given ER model)

1. **Credit card information:** This design models `Credit_cards` as an entity (rather than as an attribute of `Customers`) using the `Owns` relationship to relate to `Customers`. The reason is that the customer's credit card information may change over time (by the routine `update_credit_card`). The current active credit card is determined by the `from_date` attribute of `Owns`. A registration paid with a credit card is modeled using the `Registers` relationship which is a binary relationship between `Sessions` & `Owns` which keeps track of the credit card used for the payment.
2. **Derived attributes:** Some of the data are derived attribute values (e.g., the start & end date of `Offerings`). Such data is explicitly stored (i.e., captured in the ER model).
3. **Registers/Redeems/Cancels Relationships:** The `Registers`, `Redeems`, and `Cancels` relationships each has a `date` attribute to record the date of the transaction. By having this attribute as part of the key of the relationship, it supports multiple instances of the relationship to exist for the same set of relationship participants.

Five Constraints Not Captured by the Proposed ER Model

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

(2) Each part-time instructor must not teach more than 30 hours for each month.

(3) Each instructor must not be assigned to teach two consecutive course sessions; i.e., there must be at least one hour of break between any two course sessions that the instructor is teaching.

(4) The earliest session can start at 9am and the latest session (for each day) must end by 6pm, and no sessions are conducted between 12pm to 2pm.

(5) Each room can be used to conduct at most one course session at any time.

3. Relational Database Schema

3.1 Non-Trivial Design Decisions

3.1.1 Embedding Key and Total Participation Constraints into Entity

For binary relationships with *key and total participation constraint* on *one* of the entities and no constraints on the other (i.e. `Manages`, `Handles`, `Conducts`, `In`), we do not create separate tables for these relationships. Instead, we **include the relationships in the table of the entity which has the key and total participation constraints**. This constraint is achieved by making the foreign key attributes referencing the other parties **NOT NULL**. For example, `Manages` relationship requires a key and total participation constraint on `Course_areas`, i.e., each course area must be managed by exactly 1 manager. The foreign key `eid` referencing `Managers` is integrated into `Course_areas` and it requires a non-null value. This is to **reduce the number of tables** created, thus **preventing unnecessary complication of the schema**.

3.1.2 Enforcing Unique Values Where Necessary

For tables that only use a *unique identifier* as *primary key*, we impose an additional **UNIQUE** constraint on the rest of the attributes. For example, the primary key of `Course_packages` consists of only the attribute `package_id`. In this case, we will also require `UNIQUE (num_free_registrations, sale_start_date, sale_end_date, name, price)`. This is because `package_id` is automatically generated by the system upon insertion by incrementing 1 from the current largest id. However, it is **not desirable to have two course packages with identical attribute values except `package_id`**. This constraint is imposed on all similar tables with only the auto-incremented ID as primary key.

3.1.3 Using NUMERIC Data Types for Time and Duration

We use **NUMERIC(4, 2)** *(i.e. maximum 4 significant figures with 2 decimal places)* to represent ***time*** and ***duration*** values, whereby the value before the floating point represents hour (in 24-hour format), and the value after the floating point represents minutes (as a fraction of 60 minutes). For example, `14.5` means `14:30`. This is to **facilitate easy computation** (duration-related calculation would be a simple addition / subtraction) while **preserving the flexibility to represent timings that are not on the hour**.

3.1.4 Specifying Appropriate Ranges for NUMERIC Data Types

In addition, we always specify appropriate **significant figures and decimal places** based on common sense for all NUMERIC data types, such as prices, salary, number of work hours, etc.. For example, `monthly_salary` is represented as **NUMERIC(10, 2)** *(i.e. maximum 10 significant figures with 2 decimal places)* because it is a common practice to round monetary values to 2 decimal places, and we feel that the monthly salary of a typical full-time employee would not exceed 99999999.99. The same rationale goes for `hourly_rate`, which is represented as **NUMERIC(6, 2)** *(i.e. maximum 6 significant figures with 2 decimal places)*, because we feel that the hourly salary of a typical part-time employee would not exceed 9999.99. In addition, values such as prices, salary, number of work hours, duration should not fall below zero. Hence, we also impose necessary checks to ensure that these values are **non-negative**.

3.2 Five Constraints Not Enforced by Our Relational Schema

In this section, we outline five constraints that are not enforced by our implemented relational schema. Among these, (4) the total participation constraints and (5) `Employees'` ISA hierarchy constraints are further explained in detail. For each constraint, we also include a brief description of how it is enforced by triggers.

(1) Each customer can have at most one active or partially active package.

Enforced by: `buy_package_trigger`

Upon insertion or update on the `Buys` table, this trigger requires that the customer must not have any remaining redemptions in another package (i.e. active package) or there is any redeemed session that could be refunded if it is cancelled (i.e. partially active package). The trigger raises an exception if these checks fail.

(2) Each room can be used to conduct at most one course session at any time.

Enforced by: `insert_session_trigger`

Upon insertion or update on the `Sessions` table, this trigger requires that there is no other session conducted in the same room on the same day that overlaps in time with the current session (i.e. starts or ends in the middle of the current session). The trigger raises an exception if these checks fail.

(3) For each course offered by the company, a customer can register for (1) at most one of its sessions (2) before its registration deadline.

Enforced by: `register_if_valid_trigger, redeem_if_valid_trigger`

The details of these triggers will be covered in *Section 4. Triggers, Part (1)*.

(4) Total Participation Constraints:

a. **Each instructor specializes in a set of *one or more* course areas.**
   Enforced by: `instructor_specializes_total_part_con_trigger`

Upon insertion or update on the `Instructors` table, this trigger requires that the instructor's employee ID must appear in the `Specializes` table (i.e. the instructor must participate in the `Specializes` relationship), and raises an exception if this check is not fulfilled.

b. **Each customer must own *at least one* credit card.**
Enforced by: `customer_owns_total_part_con_trigger`
Upon insertion or update on `Customers` table, this trigger requires that the customer ID must appear in `Owns` table (i.e. the customer must participate in `Owns` relationship), and raises an exception if this check is not fulfilled.

c. **Each credit card must be owned by *at least one* customer.**
Enforced by: `credit_card_owns_total_part_con_trigger`
Upon insertion or update on `Credit_cards` table, this trigger requires that the card number must appear in `Owns` table (i.e. the credit card must participate in `Owns` relationship), and raises an exception if this check is not fulfilled.

d. **Each course offering must consist of *at least one* session.**
Enforced by: `offering_consists_total_part_con_trigger`
Upon insertion or update on `Offerings` table, this trigger requires that there is at least one entry in the `Sessions` table with the same course ID and launch date as this offering (i.e. the session belongs to this course offering), and raises an exception if this check is not fulfilled.

(5) Covering Constraints on `Employees`' ISA Hierarchy (refer to ER model for more clear visualization):

a. **Each employee in the company is either a full-time employee or a part-time employee, but not both.** *(i.e. full-time employees and part-time employees form a **disjoint bi-partition** of all employees, with covering constraint = True and overlap constraint = False)*
Enforced by: `emp_covering_con_trigger`

Upon insertion or update on the `Employees` table, this trigger requires that the new employee ID must appear in one of `Full_time_Emp` and `Part_time_Emp` tables, but not both. The trigger raises an exception if these checks are not fulfilled.

b. **Full-time employees can only be ONE of the following: administrators, managers, full-time instructors.** *(i.e. administrators, managers and full-time instructors form a **disjoint tri-partition** of all full-time employees, with covering constraint = True and overlap constraint = False)*

   Enforced by: `full_time_emp_covering_con_trigger`

   Upon insertion or update on the `Full_time_Emp` table, this trigger requires that the new employee ID must appear in *exactly* one of `Administrators`, `Managers` and `Full_time_instructors` tables, but not simultaneously in more than one table. The trigger raises an exception if these checks are not fulfilled.

c. **Part-time employees can only be part-time instructors.** *(i.e. part-time instructors form a **uni-partition** of all part-time employees, with covering constraint = True and overlap constraint = False)*

   Enforced by: `part_time_emp_covering_con_trigger`

   Upon insertion or update on the `Part_time_Emp` table, this trigger requires that the new employee ID must appear in the `Part_time_instructors` table. The trigger raises an exception if this check is not fulfilled.

d. **Instructors can only be either full-time instructors or part-time instructors, but not both.** *(i.e. full-time instructors and part-time instructors form a **disjoint bi-partition** of all instructors, with covering constraint = True and overlap constraint = False)*

   Enforced by: `instructor_covering_con_trigger`

   Upon insertion or update on the Instructors table, this trigger requires that the new employee ID must appear in one of `Full_time_instructors` and `Part_time_instructors` tables, but not both. The trigger raises an exception if these checks are not fulfilled.

4. Triggers

In this section, we describe the three most interesting triggers we have implemented for our application. For each of these triggers, we will first provide the name of the trigger, then explain the usage of the trigger, and subsequently justify our design of the trigger implementation.

(1) `register_if_valid_trigger`

- #### Explanation of the Usage of the Trigger

The `register_if_valid_trigger` checks if a registration is valid when a customer wants to register a session by making a credit card payment. Only after all relevant conditions are satisfied, the registration can be made and necessary information can be inserted into the `Registers` table. Otherwise, exceptions will be raised by the trigger. The trigger works together with our procedure `register_session` when its input **`paymentMethod` = 0** (a Boolean flag which, when set to 0, indicates that the registration is made through a direct credit card payment).

The trigger also helps during the initial data population using insert statements. It ensures that the records inserted into the `Registers` Table are all valid.

- #### Justification of the Design of the Trigger Implementation

Firstly, all five values of the `NEW` tuple are checked in the following ways upon insertion:

- `NEW.card_number`: The credit card identified by `card_number` must be owned by an existing customer, i.e. this `card_number` exists in `Owns` table.
- `NEW.course_id`, `NEW.launch_date`, `NEW.sid`: These three values form the primary key which uniquely identifies a `Session` record. The trigger checks if there exists a session with the primary key `(NEW.course_id, NEW.launch_date, NEW.sid)` in the `Sessions` table.
- `NEW.date`: The **date of registration cannot pass the registration deadline** of the valid course offering. Also, a valid registration date **cannot pass the credit card's**

expiry date. The date **cannot be earlier than the date when the customer starts to own the card either.** Hence the trigger checks if `Course_offerings.registration_deadline >= NEW.date` and `Credit_cards.expriy_date >= NEW.date` and `Owns.from_date <= NEW.date`.

Secondly, if any of the conditions above is not fulfilled, respective exceptions will be raised and the intended insertion will not be allowed. Otherwise, the trigger then proceeds to check if the customer **has completed another session before or has a coming session** under the same course as the intended registration. Since a customer can register **at most one** session for a course, **the total valid session registration for a course can only be 0 or 1**. However, this checking could not be done by simply checking the `Registers` table, because multiple cancelations for a course session are allowed and the customer could also register by redeem. Therefore, our trigger has to go through all three tables**, `Registers`, `Redeems`,** and **`Cancels`**, to count the **number of sessions of the same course with `NEW.course_id`** that belong to the **same customer that owns the credit card with `NEW.card_number`**. Let the three numbers be `count_registered`, `count_redeemed` and `count_canceled` respectively, then **this customer's total valid session registration of the course = (`count_registered` + `count_redeemed` - `count_canceled`)**. If the resultant value is 1, i.e. the customer has a valid registration for that course already, the trigger raises an exception. Otherwise, the trigger proceeds to the final check.

Finally, the trigger checks **if the session has been fully booked**. Similarly, the trigger counts the number of the same sessions as (`NEW.course_id`, `NEW.launch_date`, `NEW.sid`) by in `Registers`, `Redeems`, and `Cancels` respectively. The number of valid registrations will be (`count_registered + count_redeemed - count_canceled`). Only when the **registration number is smaller than the seating capacity of the session's room**, indicating that the session is not fully booked, the insertion of the record is then allowed.

Overall, `register_if_valid_trigger` serves as an effective safeguard to ensure that every time there is a registration request with a credit card payment, only the valid registration will be processed.

- Similar Design in `redeem_if_valid_trigger`

A similar design is used for the trigger before insertion on the `Redeems` table. Since another method of registration is by redemption from a customer's active package, a similar trigger `redeem_if_valid_trigger` is implemented to check all relevant conditions for a valid registration. The trigger works together with the procedure `register_session` when its input **`paymentMethod` = 1** (indicating that the registration is made through redeeming one session from an active package).

There are some differences between the two triggers. Firstly, for redemption, the trigger checks if the customer **owns an active package** instead of a valid credit card. Secondly, after a session has been successfully redeemed, **`num_remaining_redemption` in the `Buys` table needs to be decremented by 1.** This is done by another trigger called `update_buy_redeem_trigger`, which is similar to `update_buy_cancel_trigger` (introduced below).

(2) `update_buy_cancel_trigger`

- Explanation of the Usage of the Trigger

The `update_buy_cancel_trigger` is executed after a cancelation of a registered session. The canceled session could have been registered with a credit card payment or redeemed with an active package. This trigger ensures that the relevant records in `Buys` tables are also updated by **incrementing `num_remaining_redemption` of the package by 1 in the `Buys` table** *if applicable (i.e. if the session was redeemed using package credits and satisfies refund requirements)*. The trigger works closely together with our procedure `cancel_registration` which cancels a valid session registration of a customer.

The trigger also helps during our initial data population using insert statements. Together with `update_buy_redeem_trigger` which decrements `num_remaining_redemption` of the used package by 1 every time a new record has been inserted into the `Redeems` table, the system can then correctly compute and store a final `num_remaining_redemption`.

- Justification of the Design of the Trigger Implementation

The validity of the cancellation is checked in routine `cancel_registration`, and therefore the `update_buy_cancel_trigger` assumes that all the new tuples to the `Cancels` table are valid cancellations .

Firstly, the trigger checks **`package_credit`,** a Boolean flag of the `Cancels` table indicating the **registration method** of the canceled session. The flag value is 1 when the canceled session is **redeemed with an active package**, or is 0 when the canceled session is registered with a credit card payment.

Secondly, only if the canceled session has **`package_credit` = 1**, the trigger proceeds to the next check. Given the specification that "an extra course session to the customer's course package can be credited if the cancellation is made **at least 7 days before** the canceled session's date; otherwise, there is no refund for a late cancellation", the trigger checks whether the cancellation occurs at least 7 days before the session starts. In other words, if `New.date <= Sessions.date - 7` , `num_remaining_redemptions` of the customer's package will increment by 1, otherwise, no update is performed on Buys table. The trigger can easily identify the active or partially active package to be updated through `SELECT` the relevant information from the `Buys` table where `Buys.card_number` exists in `Owns` table with the same `cust_id` as `NEW.cust_id`, followed by `ORDER BY date DESC LIMIT 1.` It is because a customer can have at most one active or partially active package at any time, and since the trigger has checked that the session can be refunded, the package used for redemption of the cancelled session is at least a partially active package, thus is the **latest package** bought by the customer.

Overall, `update_buy_cancel_trigger` handles the side effects on `Buys` table after a cancellation is made, ensuring the consistency of data in the system.

(3) `delete_sessions_trigger & delete_sessions_update_trigger`

- Explanation of the Usage of the Trigger

The two triggers act as a pair and both will be activated upon `DELETE` on the `Sessions` table. The `delete_sessions_trigger` will be executed **before** the deletion, whereas the `delete_sessions_update_trigger` will be executed **after** the deletion. Together, these two triggers can check the legality of the deletion, that is to enforce there are **no registered customers in the session** and **the session has not started**. At the same time, the two triggers ensure that the **derived information (`start_date`, `end_date` and `seating_capacity`) inside the `Offerings` table gets updated correspondingly** if the cancellation of the given session is successful. In this way, the two triggers preserve the integrity and coherence of the data in our schema.

- Justification of the Design of the Trigger Implementation

When some record is deleted from the `Session` table, there are two parts in the `Offerings` table that need to be considered.

First is the `seating_capacity` of course offerings. The `seating_capacity` of an offering is the **sum of the seating capacities of its sessions** and will decrease if the session is removed. The new `seating_capacity` after the deletion is equal to the old `seating_capacity` minus the seating capacities of the deleted session. Therefore, the `seating_capacity` of the course offerings must be calculated **before the record is actually deleted** from the `Sessions` table. Otherwise, it is impossible to find the old `seating_capacity`. So this update is in `delete_sessions_trigger`, which is a `BEFORE` trigger.

Secondly, the `start_date` and `end_date` of the course offering will also change if the deleted session **happens to be the earliest session or the last session** in the course offering. If we want to make this update in the `delete_sessions_trigger`, we have to check if the date of the deleted session is the earliest session or last session first, and then update the `start_date` and `end_date` correspondingly if necessary. But if we do the update **after** the record is deleted, we can just **apply MAX and MIN function to the dates of all existing sessions in that course offering** to find the `start_date` and `end_date` which is more intuitive and concise, improving implementational convenience and reducing logical complexity. Therefore we implement this update in `delete_sessions_update_trigger`, which is an `AFTER` trigger.

Apart from the two updates of derived attributes in `Offerings` table mentioned above, other validity checking is all handled by the `delete_sessions_trigger` **before** the deletion action occurs. If any constraint as per the project specification is violated, the deletion will not proceed at all.

5. Difficulties Encountered and Lessons Learnt

5.1 Identify Constraints and Design Database

The requirements are written in paragraphs and the descriptions are quite general. Therefore, it took us quite some time to digest the requirements and break down the specifications into smaller subpoints. Moreover, not all constraints can be captured by the schema alone. Therefore, we spent several meetings discussing what was the best way to capture each constraint. In the end, we decided to capture as many constraints as possible in our schema, since it is the easiest way to check the requirements. For constraints that cannot be captured by schema, for example, values from another table are involved in checking the constraint, we will try our best to capture these constraints using triggers. Lastly, for constraints that can be captured by neither schema nor triggers, we will perform the checking in routines (functions and procedures). This happens typically when values need to be computed on the fly.

5.2 Data Population

A reasonable amount of pre-existing valid data is needed to test our routines. However, it is quite cumbersome to manually organize, edit and debug SQL `INSERT` statements while brainstorming new data. As such, we created a few entries of sample data for each table with Excel worksheets, and translated the entire workbook into `INSERT` statements using a Python program. Nevertheless, this manual data population process was still very time-consuming and error-prone as we needed to constantly cross-reference other tables to make sure that no constraint was violated.

5.3 Testing

We learnt how to test the routines systematically. When we transited from implementation to the testing stage, we realized that the sample data we created before we started testing was not enough. Those existing entries were not able to cover all situations and edge cases we would like to test on. Therefore, we needed to create even more sample data to fit the purpose of comprehensive testing. What we did was to first list down all the cases we would like to test. If the current sample data could not fulfill the purpose of a particular test, we would populate more data depending on the test.

We also realized that it could be difficult for the person who implemented the function to come up with boundary test cases that he/she might have potentially missed out. Therefore, we decided to cross-test other group member’s functions. We divided ourselves into pairs, and every member was responsible for testing the routines implemented by his/her partner. In this way, we were able to cover more corner cases and improve the correctness of our routines efficiently.

5.4 Teamwork

Teamwork is extremely important in this project. Every week, we conducted one or two meetings of varying duration to update our progress, plan tasks for the next week, as well as engage in team tasks or group debugging together.

During the meetings, we realized that there were many overlaps among different routines, or several routines required the same checking conditions. Therefore, our efficiency was improved by checking on each other’s progress constantly. The correctness of the routines was also improved by using the same code snippet for the checking of similar constraints to reduce the probability of encountering errors.

Furthermore, the subteam cross-testing was also helpful in improving efficiency and accuracy as mentioned in *Section 5.3*.