Project: Kitchen Glitchin'

Your favourite restaurant Salero Bundo has just celebrated its first year in business! However, with a growing customer base, it is struggling to manage its food order data. The owners, Adi and Uyen, are your dearest friends, and you have offered to design and implement a database application for the restaurant. Currently, Salero Bundo records all the transactions in tables. The tables are stored as files in a comma separated value (`.csv`) extension. There are four files for four different sets of data.

- **Menu** table (`menu.csv`) records the menu offered by Salero Bundo.

Item	Price	Cuisine

- **Registration** table (`registration.csv`) records the customers registered as a member of Salero Bundo.

Date	Time	Phone	Firstname	Lastname

- **Staff** table (`staff.csv`) records the staff of Salero Bundo.

Staff	Name	Cuisine

- **Order** table (`order.csv`) records the history of orders at Salero Bundo.

Date	Time	Order	Payment	Card	CardType
Item	TotalPrice	Phone	Firstname	Lastname	Staff

  There are 12 columns on this table. Due to space constraint, we show them on two rows above.

Each item in the **Menu** table can be uniquely identified by its item name (i.e., column **Item**). Items are categorised under exactly one cuisine. The restaurant also wants to record the cuisines that do not have any items yet, as they expect to expand their menu in the future.

When a customer signs up as a member, their data is recorded in the **Registration** table. Members are uniquely identified by their phone number, **Phone**. There is a benefit to being a member as will be explained later. Thanks to successful advertising campaigns, many people have signed up as Salero Bundo members, despite never having placed an order.

Note that the dates are in `YYYY-MM-DD` format and times are in `HH:MM:SS` format. However, they may look different if you open the file in spreadsheet applications (e.g., Excel) due to auto-conversion.

The staff of Salero Bundo is recorded in the **Staff** table. Each staff member is given a unique staff ID (i.e., column **Staff**), and is assigned to cook one or more cuisine styles (i.e., column **Cuisine**). Not all cuisine styles may have a staff assigned to them.

Orders made by customers, regardless of whether the customer is a member or not, are recorded in the **Order** table. If the customer is a member, the **Phone**, **Firstname**, and **Lastname** columns indicate their registered details. Otherwise, there is no corresponding value for these columns.

A single order (i.e., rows having the same value in **Order** column) must consist of one or more items. The same item can be ordered multiple times, and each item is prepared by a staff member who must be assigned to cook the item's cuisine. For instance, order `20240316002` is for four "Bun Cha" (your favourite dish!), two of which are made by the same staff "STAFF-06". In the current table, the data will be duplicated into multiple rows.

The payment method (i.e., **Payment**) only has two possible values: "card" and "cash". If the payment method is "card", then the card number is recorded in **Card**, and the card payment organisation in **CardType**.

Rows in the **Order** table also contain **TotalPrice**, indicating the total amount paid for the order. This amount may be cheaper than the sum of prices recorded in the **Menu** table. That is because there is a discount policy: If the customer is a member and the order contains at least 4 items, then a discount of $2 is given. For example,

- Order `20240301002` has 4 items and the customer is a member. The total price is $\$4 \times 4 - \$2 = \$14$.
- Order `20240305005` has 4 items, but is not discounted as the customer is not a member. The total price is $\$4 \times 2 + \$4 \times 2 = \$16$.

A single order will be paid in full using the same payment method and the same card (for card orders). In other words, there are no two rows with the same order but different payment method or different card or different total amount paid. A single order is also for the same customer. As such, there are no two rows with the same order but different member information. Of course, the same order will also be for the same date and time.

In real life, when a customer becomes a member, the customer is given a member card. They can show the member card when making subsequent orders. This guarantees that if an order is made by a member, the member has been registered before or at the same time as the current order.

Tasks

The transition from Salero Bundo's `.csv` files to a relational database will be done in several steps over the course of the semester. The first step is to give a **minimum viable product** (MVP). This is done by creating a preliminary table structure and showing that it can help with enforcing some simple constraints while maintaining the information content of the original `.csv` files.

We will then improve this MVP by applying the Entity-Relationship model as the second step. In this step, some constraints still cannot be enforced due to the limitation of Entity-Relationship model itself. Lastly, we will enforce all constraints using triggers. In this step, we will also provide stored procedures to simplify some operations.

3. (6 points) **Stored Functions and Triggers**

In this part, we will be using the schema given in `schema-part3.sql`. Please take some time to read the schema; in particular, note that we store the orders in table `Food_Order`, and there is a column `total_price` which records the order's total amount paid.

(a) Through the use of the five integrity constraints: `PRIMARY KEY`, `UNIQUE`, `NOT NULL`, `FOREIGN KEY`, and `CHECK`, this schema is able to enforce certain constraints, such as "Items are uniquely identified by their names" and "Each staff member is given a unique ID". However, certain important constraints of the application cannot be enforced. Some of those constraints are listed below (there may be other uncaptured constraints, but focus on these four):

- Each order should have at least one item.
- For each item prepared by a certain staff, that staff member should be assigned to cook the item's cuisine.
- For each order placed by a member, the order's date and time should not occur before the member's registration date and time.
- For each order, the value of column `total_price` should always be correctly computed as the sum of prices of all items contained in that order, minus the discount if applicable.

Write the necessary triggers to implement the four above constraints. You are advised to carefully think of all scenarios where they may be broken, e.g., what kinds of insertion/update/deletion will violate these constraints, and design your triggers to cover all such scenarios.

In addition, your trigger(s) to enforce the fourth constraint on total price consistency should **not** simply reject any operation that would result in an incorrect total price. Instead, the trigger(s) should update the total price to be the correct value.

IMPORTANT: You must ONLY use the `PUBLIC` schema.

File to be submitted: `triggers.sql`.

Next, write the code you used to test your triggers, indicating as comments the expected outcome of each test. For example, an `INSERT` statement is used to demonstrate that attempting to insert an item prepared by an incorrect staff will be rejected.

File to be submitted: `triggers-tests.sql`.

(b) Furthermore, to simplify the process of creating orders in the application, we would like to provide a procedure that performs the insertion rather than having to issue separate SQL statements by ourselves.

Write a procedure `insert_order_item`. The procedure takes in an order ID (`VARCHAR(256)`), the order date (`DATE`) and time (`TIME`), the payment method (`VARCHAR(10)`), card number and card type (both `VARCHAR(256)`), the member's phone number (`INTEGER`), plus the name of the item being ordered (`VARCHAR(256)`) and the ID of the staff preparing said item (`VARCHAR(256)`). This procedure should create a new order if it does not

exist yet, and add the given item into the order.

File to be submitted: `procedure.sql`.

Finally, write the SQL code to insert the first 100 rows from `order.csv` (excluding the header), using **NOT** plain `INSERT` statements as in Part 1 and Part 2, but calling the `insert_order_item` procedure.

File to be submitted: `procedure-tests.sql`.

IMPORTANT: For both the triggers and stored procedure submissions, marks will be awarded based on (private) test cases. Your mark will not be affected by the number of triggers written, or how complicated the code is.

Submission:

Submit your solution on Canvas.

`Assignments`

▶ `Project #3: Stored Functions and Triggers`

Submit the following four separate files.

- `triggers.sql`
- `triggers-tests.sql`
- `procedure.sql`
- `procedure-tests.sql`

Note that you need to submit all files for each submission even if there is no change to one of the files. Otherwise, the latest submission will not include the missing file.

You also need to ensure that the **file names are correct before submission** (i.e., the file names in your computer). Canvas may perform renaming; that is normal.