

Database - Driven Intelligent Restaurant Management System

by

Li Xinze¹ (2330026083)

Li Jiale¹ (2330026073)

Tian Zhiwen² (2230033036)

Yan Shan² (2230033048)

Group 6 Project (COMP3013 Database Management System)

Bachelor of Science (Honours)

in

¹Computer Science and Technology

²Applied Mathematics

Supervised by

Dr. Zhijian Li

Beijing Normal University - Hong Kong Baptist University

May 10, 2025

Contents

1	Project Description	1
2	Our Dataset	1
2.1	Dataset	1
2.2	Data collection	1
2.3	Data preprocessing approaches	2
3	Database Design	2
3.1	Assumptions	2
3.2	Entity and Relationship Set	2
3.2.1	Entities	2
3.2.2	Relationships	3
3.3	ER diagram	4
3.4	Functional Dependencies	5
3.5	Schemas	5
3.6	Primary Keys	6
3.7	Normal Forms and Analysis	7
3.8	Data insertion	7
4	Features	8
4.1	Triggers	8
4.2	BLOB	9
5	Access Control	9
5.1	User Registration (register.html)	9
5.2	User Login (login.html)	9
5.3	Customer Pages	10
5.3.1	home.php	10
5.3.2	book.php	11
5.3.3	order.php	11
5.3.4	order-detail.php	11
5.3.5	credit.php	11
5.3.6	add-dish-to-order.php	11
5.4	Administrator Page (admin.php)	12
5.5	Receptionist Page (receptionist.php)	12
5.6	Cleaner Page (cleaner.php)	12
5.7	Logout Page (logout.php)	13
6	Implemented Functions	13
6.1	Front-End Functions	13
6.2	Back-End Functions	13
7	SQL Codes and Explanations	14
7.1	User Registration	14
7.2	User Login	14
7.3	List All Table Types	14
7.4	Table Details	14

7.5	Create Order	15
7.6	List Orders	15
7.7	Query Wallet Balance	15
7.8	Update Wallet Balance	15
7.9	Retrieve User Information	16
7.10	Table Management	16
7.11	Table Type Management	16
7.12	Order Management	16
7.13	Find Tables Requiring Cleaning	16
7.14	Update Table Cleaning Status	17
7.15	Add Dish to Order	17
7.16	Trigger: Update Order Total	17

8	Contribution	17
----------	---------------------	-----------

1 Project Description

The purpose of this project is to develop a modern restaurant management system that addresses the common real-life challenges faced by restaurants in handling reservations, table allocation, and service coordination. The main problem being solved is the inefficiency and rigidity of traditional restaurant management systems, which make it difficult to track table availability, manage reservations, handle special customer requests, and keep staff updated in real time. This often leads to operational difficulties such as overbooking, underutilization of dining space, delayed service, and poor customer experience.

The difficulty of the problem lies in the need to coordinate multiple moving parts — reservations, table statuses, staff roles, menu management, and real-time updates — within a single system, all while minimizing human error. Additionally, the system must balance usability for both staff and customers, ensure data consistency, and remain adaptable to the dynamic nature of restaurant operations.

Abstractly, this problem can be described as a multi-user, multi-role resource allocation and workflow coordination challenge, where tables are the resources, reservations are the demands, and the system must efficiently match supply to demand in real time. This involves building a robust database with automation, providing intuitive interfaces, and designing a flexible architecture that supports diverse roles and workflows.

The major goal of this project is to create a dynamic, secure, and user-friendly restaurant management platform that streamlines operations, automates routine tasks, and improves the overall efficiency and experience for both staff and customers. By leveraging modern database techniques, automation through triggers, and role-based access control, the system aims to bridge the gap between outdated manual processes and the demands of contemporary restaurant operations.

2 Our Dataset

2.1 Dataset

The dataset contains synthetic records simulating real-world restaurant data:

- Users (customers, receptionists, cleaners, admin)
- Tables and table types
- Orders with order details
- Menu items
- Comments/feedback

2.2 Data collection

Data was generated to cover common restaurant operations, including multiple users with different roles, reservations, cleaning tasks, and menu selections.

2.3 Data preprocessing approaches

We cleaned and standardized the data, ensuring consistency in table status, balance amounts, order timestamps, and valid relationships between entities.

3 Database Design

3.1 Assumptions

- Each user has a distinct and exclusive role within the restaurant management system: admin, receptionist, cleaner, or customer. This role-based structure simplifies access control and ensures that users can only perform actions relevant to their job functions.
- Each order corresponds to one table, establishing a clear link between the customer's dining experience and the physical seating arrangement. Additionally, each table is assigned to a specific table type, helping with pricing, capacity management, and customer expectations.
- Customers can place orders and provide comments. Orders are central to the restaurant's operations, while comments serve as valuable feedback for improvement.
- Receptionists manage orders, including taking reservations, updating order statuses, and communicating with customers. Their role ensures a smooth customer experience from booking to meal completion.
- Cleaners are responsible for maintaining the cleanliness of dining areas by updating the cleaning status of tables. This ensures that tables are ready for the next customers in a timely manner.
- Admins manage table types, including creating new table types, updating their properties (e.g., price, description), and monitoring availability. This flexibility allows the restaurant to adapt to changing business needs and customer demands.

3.2 Entity and Relationship Set

3.2.1 Entities

- **User:** Represents all individuals interacting with the system, storing common information such as uID, name, phone, password, user_type, and balance. This entity provides a unified framework for managing user accounts and access rights.
- **Admin:** Inherits from the User entity and adds an AdminID to distinguish administrators. Admins have elevated privileges to manage the system, including table types, staff, and settings.
- **Receptionist:** Inherits from the User entity and is identified by Receptionist_ID. Receptionists manage customer orders, take reservations, and update order statuses.
- **Cleaner:** Inherits from the User entity and is identified by Cleaner_ID. Cleaners focus on maintaining the cleanliness of tables and updating their clean status.

- **Customer:** Inherits from the User entity and is identified by cID. Customers place orders, make reservations, and provide feedback.
- **Table:** Represents each physical table in the restaurant. It has a unique yID, links to a specific table type (ttID), and records its cleanliness status.
- **Table_type:** Defines the different table types in the restaurant. It includes ttID, Name, Introduction, Price, Remain, and Img to provide details about each table type for customers and management.
- **Menu:** Stores details about each dish, including dID as the unique identifier, dish name, and price.
- **Order:** Captures each customer order with oID as the unique identifier, linking to the cID of the customer, the tID of the assigned table, the check-in status, the order price, and the order time.
- **Comment:** A weak entity dependent on the order, ensuring that comments are tied to specific orders.

3.2.2 Relationships

- **User ISA Admin, Receptionist, Cleaner, Customer** *ISA*
This inheritance relationship models role-based access control. Each user is assigned one role, and common attributes (e.g., name, contact) are stored in the User entity, with role-specific attributes defined in sub-entities.
- **Admin Edit Table_type (admin_table_type)** *(1:N)*
An admin can modify multiple table types, and each table type can be edited by an admin. This relationship supports dynamic configuration of table properties (e.g., capacity, pricing).
- **Receptionist Deal Order (receptionist_order)** *(1:N)*
A receptionist can handle multiple orders, but each order is processed by exactly one receptionist, ensuring accountability for order management.
- **Cleaner Clean Table (cleaner_table)** *(M:N)*
Multiple cleaners may be assigned to clean a table, and each cleaner can clean multiple tables, supporting collaborative cleaning schedules.
- **Customer Initiate Order (customer_order)** *(1:N)*
A customer can place multiple orders, but each order is associated with a single customer. This relationship tracks customer dining history and preferences.
- **Table_type Belong Table (table_table_type)** *(1:N)*
A table type (e.g., VIP, standard) can have multiple instances, but each table belongs to exactly one type. This relationship supports pricing and resource allocation.
- **Order Select Menu (orders_dishes)** *(M:N)*
An order can include multiple dishes, and each dish can appear in multiple orders. This relationship is modeled via the `orders_dishes` associative table, which records the quantity and customization of each dish.

- **Order Occupy Table (order_table)** (1:1)
Each order is assigned to a specific table during the dining period, ensuring exclusive use of the table. This relationship is vital for managing table availability.
- **Order Create Comment (order_comment)** (1:1)
Comments are weak entities dependent on orders. Each comment must reference an existing order, ensuring that feedback is tied to specific dining experiences.

3.3 ER diagram

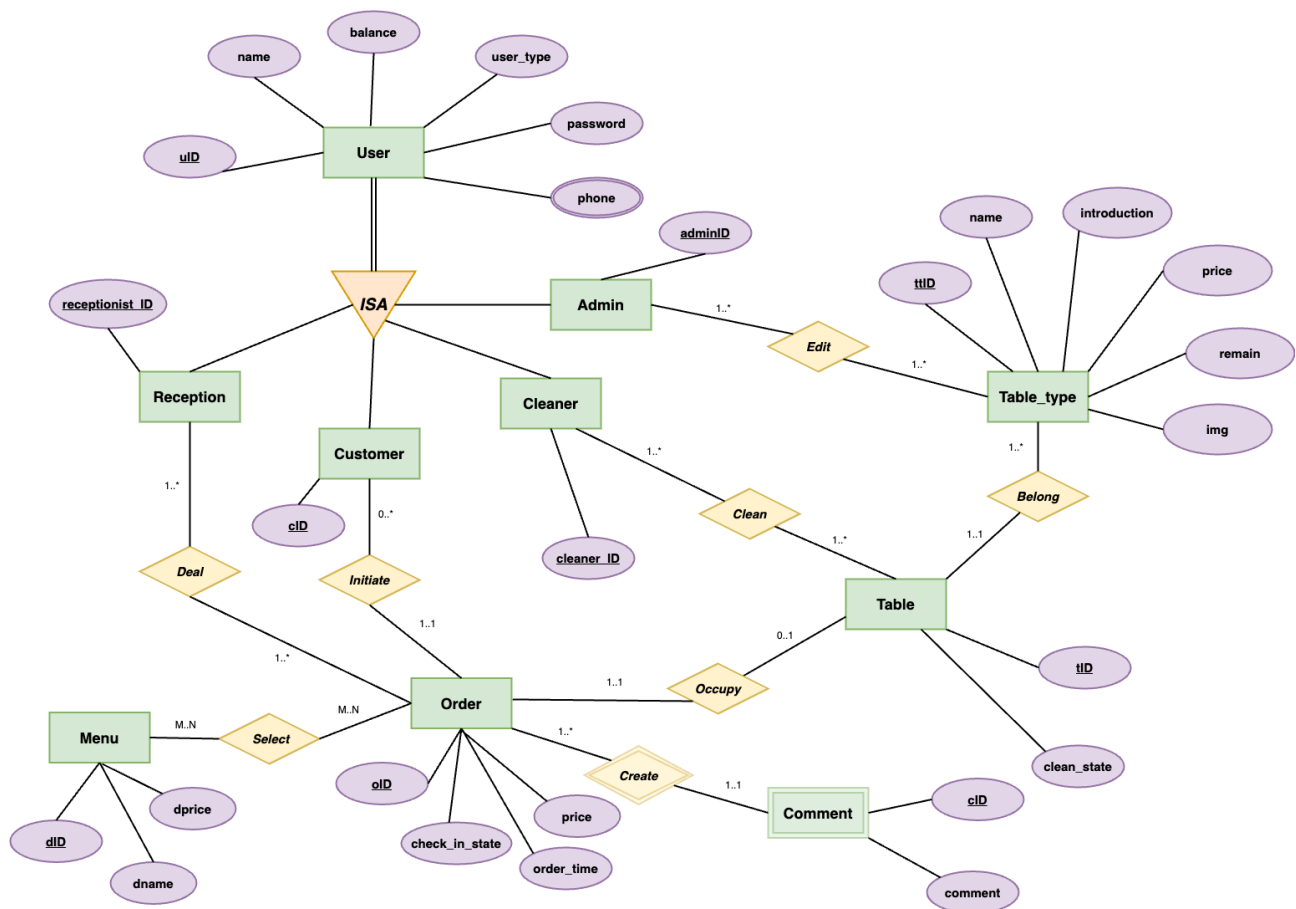


Figure 1: ER diagram

The ER diagram visually represents the entities and relationships in the database. At the center is the user entity, with inheritance relationships branching out to admin, receptionist, cleaner, and customer entities, clearly indicating the role - based structure. The table entity is connected to the table type entity through a one - to - many relationship, signifying that multiple tables can belong to the same table type. The order entity acts as a central hub, connecting to the user (customer), table, and menu entities, which effectively reflects the complex interactions during the dining process. The comment entity is linked to the order entity, emphasizing its dependence on the existence of an order.

3.4 Functional Dependencies

The following functional dependencies represent the relationships between attributes in the database schema, capturing the rules of attribute determination based on keys:

- $uID \rightarrow name, phone, , password, user_type, balance$
For any user (uID), the associated name, phone number, user type (role), and balance can be uniquely determined. This applies to all user types (Admin, Receptionist, Cleaner, Customer).
- $tID \rightarrow ttID, clean_state$
A specific table (tID) uniquely determines the table type ($TtID$) and its cleanliness state ($Clean_state$). Each table has a designated type and cleanliness status.
- $oID \rightarrow cID, tID, receptionist_ID, check_in_state, price, order_time$
For an order (oID), the customer (cID), table (tID), receptionist ($Receptionist_ID$), check-in state, price, and order time are uniquely determined. This describes the details of a specific order.
- $dID \rightarrow dname, dprice$
A dish (dID) uniquely determines its name ($Dname$) and price ($Dprice$). Each dish has a distinct identifier and corresponding attributes.
- $(oID, dID) \rightarrow Quantity$
A combination of order (oID) and dish (dID) uniquely determines the quantity options for that specific dish in the order. This models the many-to-many relationship between orders and dishes.
- $ttID \rightarrow name, introduction, price, remain, img$
A table type ($ttID$) uniquely determines its name, introduction, price, remaining capacity ($Remain$), and image (Img). Each table type (e.g., VIP, Standard) has associated details that are determined by its type.

3.5 Schemas

- user (uID , name, phone, password, user_type, balance)
- admin (adminID, uID)
. [FK: $uID \rightarrow User$]
- Receptionist (receptionist_ID, uID)
. [FK: $uID \rightarrow User$]
- cleaner (cleaner_ID, uID)
. [FK: $uID \rightarrow User$]
- customer (cID, uID)
[FK: $uID \rightarrow User$]
- table (tID , $ttID$, clean_state)
. [FK: $ttID \rightarrow Table_type$]

- table_type (ttID, name, introduction, price, remain, img)
- menu (dID, dname, dprice)
- ord (oID, cID, tID, receptionist_ID, check_in_state, price, order_time)
. [FKs: cID → Customer, tID → Table, receptionist_ID → Receptionist]
- comment (cID, oID, comment)
[Composite PK, FKs: cID → Customer, oID → Order]
- admin_Table_type (adminID, ttID, edit_time)
[Composite PK, FKs: adminID → Admin, ttID → Table_type]
- cleaner_Table (cleanerID, tID, clean_time)
[Composite PK, FKs: cleanerID → Cleaner, tID → Table]
- orders_dishes (oID, dID, quantity)
[Composite PK, FKs: oID → ord, dID → Menu]

3.6 Primary Keys

- user: uID
- admin: adminID
- receptionist: receptionist_ID
- cleaner: cleaner_ID
- customer: cID
- table: tID
- table_type: ttID
- menu: dID
- order: oID
- comment: cID, oID
- admin_table_type: adminID, ttID
- cleaner_Table: cleanerID, tID
- orders_dishes: oID, dID

3.7 Normal Forms and Analysis

All schemas are in **Third Normal Form (3NF)** because:

- Each relation has a primary key.
- All non-prime attributes are fully functionally dependent on the primary key.
- There are no transitive dependencies.

Explanation examples:

- **User:** uID is the primary key; attributes such as name, phone, user_type, and balance depend only on uID.
- **Table:** tID is the primary key; attributes like clean_state and ttID depend only on tID.
- **Order:** oID is the primary key; attributes such as cID, tID, receptionist_ID, check_in_state, price, and order_time depend only on oID.
- **Orders Dishes:** (oID, dID) as the composite key; attributes such as quantity and customization depend on both oID and dID.

No decomposition needed, as all schemas already satisfy 3NF. If decomposition were necessary (e.g., if a non-prime attribute depended on another non-prime attribute), we would apply the standard 3NF decomposition steps:

1. Identifying partial or transitive dependencies.
2. Splitting the relation into smaller relations, preserving dependencies and keys.
3. Ensuring lossless join and dependency preservation.

3.8 Data insertion

During the data entry stage, the database simulates the large-scale data environment of hotel operations. For instance, 50,000 pieces of user data were inserted into the system, including one administrator, 200 cleaners, 200 receptionists, and the rest of the customers. At the same time, data for a total of 50,000 orders over the past three years were also inserted. Each order contains information such as the order time, current status, and price, which is intended to be used by the front desk staff for managing customer reservations and settlements. Additionally, five types of tables were provided for customers to choose from, totaling 400 tables, and the hygiene conditions of each table were marked so that the cleaners would know which tables needed cleaning. We also set 82 dishes on the menu for customers to order.

4 Features

4.1 Triggers

Triggers are designed to automatically update customer balances and table cleanliness statuses when orders are created or completed. The following triggers ensure data consistency and automate critical business processes:

1. Automatically Update Total Order Amount

- **Scenario:** Updates the `total_amount` field in the `order` table whenever items are inserted into or deleted from the `orders_dishes` table.
- **Trigger Type:** `AFTER INSERT` and `AFTER DELETE` on `orders_dishes`.
- **Logic:** The trigger calculates the total amount by multiplying the dish quantity with its price (queried from the `menu` table) and updates the corresponding order record.

2. Log Order Status Changes

- **Scenario:** Records historical changes of order statuses (e.g., from "In Progress" to "Completed") in a dedicated log table.
- **Trigger Type:** `BEFORE UPDATE` on `order`.
- **Logic:** Checks if the status field is modified. If changed, inserts a new entry into `order_status_log` with timestamps and status transitions.

3. Automatic Inventory Deduction

- **Scenario:** Reduces dish stock in the `menu` table when an order is marked as 'Completed';.
- **Trigger Type:** `AFTER UPDATE` on `order`.
- **Logic:** Activates only when the status transitions to "Completed". Deducts the ordered quantities from the corresponding dish inventory.

4. Data Integrity Validation

- **Scenario:** Ensures that every `dish_id` added to `orders_dishes` exists in the `menu` table.
- **Trigger Type:** `BEFORE INSERT` on `orders_dishes`.
- **Logic:** Validates the existence of `dish_id` in the `menu` table. Raises an error if the dish is invalid, preventing invalid data insertion.

Key Implementation Notes

- Use database indexing (e.g., on `dish_id` and `order_id`) to optimize trigger performance.
- Ensure transaction atomicity to avoid partial updates during concurrent operations.
- Regularly monitor trigger execution times to mitigate potential performance bottlenecks.

This design ensures robust data management while adhering to business rules and minimizing manual intervention.

4.2 BLOB

BLOB fields are used in the `Table_type` entity to store table type images, enhancing the visual presentation for customers.

5 Access Control

Each page checks the user's role and login status to control access authority and ensure that only authorized users can perform certain actions.

5.1 User Registration (`register.html`)

The registration form captures user details and stores them in the database. Upon submission:

- User inputs are validated via POST request to `reg-check.php`.
- Passwords are checked for matching values.
- `usrType` determines access privileges:
 - `receptionist`: Full reservation management.
 - `customer`: View/reserve tables.
 - `cleaner`: Access cleaning schedules.
- Session variables (e.g., `$_SESSION['usrType']`) are set post-login.

5.2 User Login (`login.html`)

The login page allows registered users to authenticate themselves and gain access to role-specific dashboards. User credentials are securely transmitted via a POST request to `login-check.php` for server-side verification. Upon successful authentication, user session data is stored to maintain authentication state and role information across pages.

The login form includes the following components:

- **UserID Field:** Required input for the user's unique identifier (autogenerated during registration).
- **Password Field:** Masked input for the user's password (required).
- **Login Button:** Submits the form data to `login-check.php`.

- **Registration Link:** Redirects new users to `register.html` if they need to create an account.

The server-side script `login-check.php` should perform the following actions:

1. Validate UserID and password against database records.
2. Retrieve user role (receptionist, customer, or cleaner) from the database.
3. Start a session and store user information:
 - `$_SESSION['userID']`: Unique identifier for the user.
 - `$_SESSION['usrType']`: User role determining access privileges.
 - `$_SESSION['loggedin']`: Boolean flag to track authentication status.
4. Redirect user to their role-specific dashboard:
 - Receptionists: `receptionist-dashboard.php`.
 - Customers: `customer-dashboard.php`.
 - Cleaners: `cleaner-dashboard.php`.
5. Handle authentication failures with appropriate error messages.

Session management is critical for maintaining security and access control. Each subsequent page must include session validation to ensure only authenticated users with the correct role can access restricted content.

5.3 Customer Pages

5.3.1 home.php

Displays all available table types. The top navigation menu includes links to:

- `book.php`
- `credit.php`
- `order.php`
- `logout.php`

Clicking on a specific table redirects to `book.php`. Clicking the “Orders” button redirects to `order.php`. Clicking the “Wallet” button redirects to `credit.php`.

5.3.2 book.php

Shows details about a specific table type, including:

- Reviews
- Prices
- Available table count

If the user does not have sufficient balance, the purchase button will not be displayed. After clicking the purchase button, the request is submitted via POST to `book-check.php` for validation. Successful bookings update the user's order information in the database.

5.3.3 order.php

Lists all of the user's past and current orders.

5.3.4 order-detail.php

Displays details of a specific order, such as:

- Date
- Price
- Table type
- Table number

A comment box is provided for feedback. Comments are submitted via POST to `comment-check.php` for validation. If a review is created, it is submitted via POST to `submit-review.php` for storage in the database.

5.3.5 credit.php

Displays the user's current wallet balance and provides an input field for adding funds. Top-ups are submitted via POST to `charge-check.php`.

5.3.6 add-dish-to-order.php

This page lets logged-in customers add dishes to active (occupying) orders.

Key logic:

- Check `order_id` (GET) and `$_SESSION['userID']`; redirect to login if needed.
- Retrieve order details (status, total, balance) from `Ord`, `Customer_Order`, `User`.
- Allow dish addition only if order status is `occupying`.
- Load menu items from `Menu`; list current order items from `Orders_Dishes`.

On submission:

- Get dish ID and quantity; enforce minimum of 1.
- Check if total exceeds balance; show alert if insufficient.
- Insert or update dish using `ON DUPLICATE KEY UPDATE`.
- Redirect to refresh the page and show updates.

Page layout:

- Header: system name, links (`home.php`, `credit.php`, `order.php`, `logout.php`).
- Main: dish selection form, current order summary.
- Footer: team credits.

5.4 Administrator Page (admin.php)

Administrators can edit user and table type information. All actions are handled by `admin-actions.php`.

5.5 Receptionist Page (receptionist.php)

Receptionists manage check-ins and check-outs. By entering the user's ID, the associated table is retrieved. Check-in and check-out actions are triggered by clicking buttons and submitted via POST to `recept-check.php`.

5.6 Cleaner Page (cleaner.php)

Cleaners can:

- View tables that need cleaning.
- Mark tables as cleaned.

A list of tables awaiting cleaning is displayed. Clicking the “Clean” button removes the table from the list.

5.7 Logout Page (logout.php)

Clears session data and logs the user out.

6 Implemented Functions

This section summarizes all front-end and back-end functions developed in the system.

6.1 Front-End Functions

- **register.html** — User registration form.
- **login.html** — User login form.
- **home.php** — Displays available table types with navigation links.
- **book.php** — Shows table type details; enables booking if balance is sufficient.
- **order.php** — Displays past and current orders.
- **order-detail.php** — Shows detailed order information; allows comments and reviews.
- **credit.php** — Displays wallet balance; provides top-up form.
- **add-dish-to-order.php** — Provides a form to select and add dishes to active orders.
- **admin.php** — Admin panel for managing users and table types.
- **receptionist.php** — Interface for managing check-in and check-out operations.
- **cleaner.php** — Lists tables to clean; provides one-click clean update.
- **logout.php** — Logs out the user and clears session data.

6.2 Back-End Functions

- **reg-check.php** — Validates registration form, stores user data in the database.
- **login-check.php** — Validates login credentials, starts user session.
- **book-check.php** — Validates and processes table reservations.
- **charge-check.php** — Processes wallet top-ups.
- **comment-check.php** — Handles user comments on orders.
- **submit-review.php** — Stores user reviews in the database.

- **admin-actions.php** — Handles admin operations like fetching and updating users, tables, and table types.
- **recept-check.php** — Manages receptionist operations for check-in and check-out.
- **add-dish-to-order.php** (PHP section) — Processes dish additions, validates balance, updates the database.

7 SQL Codes and Explanations

7.1 User Registration

```
INSERT INTO User (uID, name, phone, password, user_type, balance)
VALUES (NULL, 'username', '1234567890', 'hashed_password', '
    customer', 0.0);

INSERT INTO Customer (cID, uID)
VALUES (NULL, LAST_INSERT_ID());
```

Explanation: Inserts a new user into the User table and links to the Customer table.

7.2 User Login

```
SELECT *
FROM User
WHERE uID = '$userID' AND password = '$password';
```

Explanation: Checks username and password for authentication.

7.3 List All Table Types

```
SELECT *
FROM Table_type
WHERE remain > 0;
```

Explanation: Retrieves all available table types with remaining seats.

7.4 Table Details

```
SELECT *
FROM Table_type
WHERE ttID = '$ttID';
```

```
SELECT comment FROM Comment WHERE oID IN (
    SELECT oID FROM Ord WHERE tID IN (
        SELECT tID FROM `Table` WHERE ttID = '$ttID'
    )
);
```

Explanation: Retrieves table details and associated user comments.

7.5 Create Order

```
INSERT INTO Ord (oID, cID, tID, receptionist_ID, check_in_status,
    price, order_time)
VALUES (NULL, '$cID', '$tID', '$receptionistID', 'occupying', 0.0,
    NOW());
```

Explanation: Creates a new order entry in the system.

7.6 List Orders

```
SELECT *
FROM Ord
WHERE cID = '$cID';
```

Explanation: Retrieves all orders for a specific user.

7.7 Query Wallet Balance

```
SELECT balance
FROM User
WHERE uID = '$userID';
```

Explanation: Retrieves the current balance of the user.

7.8 Update Wallet Balance

```
UPDATE User SET balance = balance + $amount WHERE uID = '$userID';
```

Explanation: Updates the user's wallet balance after recharge.

7.9 Retrieve User Information

```
SELECT *  
FROM User  
WHERE uID = '$userID';
```

Explanation: Fetches full user details using their ID.

7.10 Table Management

```
SELECT *  
FROM `Table`  
WHERE tID = '$tID';  
  
UPDATE `Table` SET clean_status = '$status' WHERE tID = '$tID';
```

Explanation: Retrieves or updates table cleaning status.

7.11 Table Type Management

```
SELECT *  
FROM Table_type  
WHERE ttID = '$ttID';  
  
UPDATE Table_type SET introduction = '$intro', price = $price,  
    remain = $remain  
WHERE ttID = '$ttID';
```

Explanation: Retrieves or modifies table type information.

7.12 Order Management

```
SELECT *  
FROM Ord  
WHERE oID = '$oID';  
  
UPDATE Ord SET check_in_status = '$status', price = $price WHERE  
    oID = '$oID';
```

Explanation: Retrieves or updates details of a specific order.

7.13 Find Tables Requiring Cleaning

```
SELECT *  
FROM `Table`  
WHERE clean_status = 'dirty';
```

Explanation: Finds all tables marked as needing cleaning.

7.14 Update Table Cleaning Status

```
UPDATE `Table` SET clean_status = 'clean' WHERE tid = '$tid';
```

Explanation: Marks a table as cleaned.

7.15 Add Dish to Order

```
INSERT INTO Orders_Dishes (oID, dID, quantity)  
VALUES ('$oID', '$dID', $quantity)  
ON DUPLICATE KEY UPDATE quantity = quantity + $quantity;
```

Explanation: Adds a dish to an existing order or updates its quantity.

7.16 Trigger: Update Order Total

```
CREATE TRIGGER update_order_total  
AFTER INSERT ON Orders_Dishes  
FOR EACH ROW  
BEGIN  
    UPDATE Ord SET price = price + (  
        SELECT dprice * NEW.quantity FROM Menu WHERE dID = NEW.dID  
    ) WHERE oID = NEW.oID;  
END;
```

Explanation: Automatically updates the total price when new dishes are added.

8 Contribution

Table 1: Team Member Contributions

Name	Student ID	Contribution
Li Xinze	2330026083	
Li Jiale	2330026073	
Tian Zhiwen	2230033036	ER diagram design, Trigger implementation, Report writing
Yan Shan	2230033048	Dataset insert