CS306 Project Phase 2
Gym Membership Management System

Duru Nef Özmen – 31090
Zeynep Dağcı – 31061

April 10, 2025

1 Introduction

This project establishes a **Gym Membership Management System** in `MySQL`, comprising a set of interrelated tables for *Membership Plans*, *Members*, *Trainers*, *Classes*, *Payments*, and *Attendance*.

The schema is designed to capture essential operational details—such as membership durations, trainer specialties, class schedules, and payment information—while maintaining referential integrity through carefully defined foreign key relationships. Sample records are inserted for each table to demonstrate practical use cases, such as tracking each member's chosen plan, logging attendance for specific classes, and recording and validating payments. Together, these elements lay a solid foundation for a full-featured system that supports a gym's core needs—from registering new members to processing payments and documenting ongoing class engagement.

In this phase of the project, we introduced **two stored procedures** and **two triggers** to extend the system's functionality and enforce important business rules.

The first procedure, `sp_register_member`, handles new member registrations. It accepts a member's details (name, age, gender, contact information, and selected membership plan), inserts them into the `Member` table, and returns the newly created member ID. The second procedure, `sp_add_payment`, manages payment processing by inserting new records into the `Payment` table. It stores relevant payment data—member ID, amount, date, and payment method—and returns both the generated payment ID and a success message.

Meanwhile, our triggers enforce additional constraints. The first trigger, `trg_prevent_overbooking`, ensures that gym classes do not exceed their capacity by checking the number of existing attendees before allowing new "Attended" entries. Once a class reaches full capacity, no more "Attended" records can be added (though "Missed" entries remain exempt). The second trigger, `trg_verify_payment_amount`, prevents payment errors by verifying that the amount corresponds to the plan's actual cost. If there is a mismatch, the new payment is blocked, ensuring members always pay the correct amount for their chosen plan.

2 Triggers

2.1 Trigger #1: Prevent Overbooking (`trg_prevent_overbooking`)

We designed this trigger to ensure our gym classes never become overcrowded. Each class has a specific capacity, and we wanted to avoid situations where members show up to a fully booked session. To automatically handle this, we set up the trigger to run right before inserting a new attendance record into the database.

Without this safeguard, multiple concurrent inserts could potentially exceed the `Capacity` defined in the `Class` table, leading to overbooked sessions and dissatisfaction among gym members. This trigger stops the violation before the insert occurs, ensuring the transaction rolls back completely—maintaining data integrity without manual intervention.

Property	Value
Timing / Event	BEFORE INSERT ON Attendance
Granularity	Row-level (fires once per row inserted)
Business Rule	Do not allow more 'Attended' seats than class capacity.
Tables Involved	Attendance (new row), Class (capacity lookup)

Table 1: Properties of trg_prevent_overbooking

Step-by-Step Explanation:

1. **Check Attendance Status:** First, we verify if the incoming attendance status is `'Attended'`. If the status is `'Missed'`, capacity checks aren't necessary.
2. **Count Current Attendance:** Next, we count how many members are already marked as `'Attended'` for the class on the specified date to determine current occupancy.
3. **Fetch Class Capacity:** Then, we retrieve the maximum number of attendees allowed for that class from the `Class` table.
4. **Decision and Enforcement:** If the class is full, the trigger raises a clear, descriptive error and prevents the attendance record from being added.

Code Explanation `trg_prevent_overbooking`:

```
CREATE TRIGGER trg_verify_payment_amount
BEFORE INSERT ON Payment -- The trigger runs before any new row is inserted
    into the Payment table
FOR EACH ROW -- It applies to every row being inserted(row-level trigger)
BEGIN
    -- Declare a variable to store the actual cost of the member's plan
    DECLARE plan_cost DECIMAL(10,2);

    SELECT Cost INTO plan_cost -- Retrieve the cost of the membership plan
        that the member is currently enrolled in
    FROM Membership_Plan -- Look up the corresponding Cost from the
        Membership_Plan table
    WHERE Plan_ID = ( -- Find the Plan_ID of the member using their Member_ID
        SELECT Plan_ID FROM Member
        WHERE Member_ID = NEW.Member_ID
    );
    -- Compare the entered payment amount (NEW.Amount) with the actual plan
        cost
    -- If they don't match, block the insertion and raise a custom error
    IF NEW.Amount != plan_cost THEN
        SIGNAL SQLSTATE '45000' -- Raise a controlled exception
        SET MESSAGE_TEXT = 'Payment amount does not match the cost of the
            selected plan.';
    END IF;
END;
```

Testing:

To test this trigger, we set the capacity of class 301 to 3. We successfully added three attendance records marked `'Attended'`. We then inserted two entries marked `'Missed'`, which succeeded as these do not count toward capacity limits. Finally, we attempted adding a fourth `'Attended'` record, which triggered an error: `"Cannot add attendance: Class is already full."`

Test Script for Attendance Trigger

```sql
-- Clean up any previous test data
DELETE FROM Attendance WHERE Class_ID = 301 AND Date = '2025-04-10';

-- Set class capacity to 3 for testing
UPDATE Class SET Capacity = 3 WHERE Class_ID = 301;

INSERT INTO Attendance (Member_ID, Class_ID, Date, Status)
VALUES (101, 301, '2025-04-10', 'Attended'); -- SUCCESS

INSERT INTO Attendance (Member_ID, Class_ID, Date, Status)
VALUES (102, 301, '2025-04-10', 'Attended'); -- SUCCESS

INSERT INTO Attendance (Member_ID, Class_ID, Date, Status)
VALUES (103, 301, '2025-04-10', 'Attended'); -- SUCCESS

INSERT INTO Attendance (Member_ID, Class_ID, Date, Status)
VALUES (105, 301, '2025-04-10', 'Missed'); -- SUCCESS

INSERT INTO Attendance (Member_ID, Class_ID, Date, Status)
VALUES (106, 301, '2025-04-10', 'Missed'); -- SUCCESS

INSERT INTO Attendance (Member_ID, Class_ID, Date, Status)
VALUES (104, 301, '2025-04-10', 'Attended'); -- FAILED: triggers an error

-- Final SELECT to verify inserted records
SELECT * FROM Attendance
WHERE Class_ID = 301 AND Date = '2025-04-10';
```

before_trigger1

Attendance_ID	Member_ID	Class_ID	Date	Status
64	101	301	2025-04-10	Attended
65	102	301	2025-04-10	Attended
66	103	301	2025-04-10	Attended
67	104	301	2025-04-10	Attended
68	105	301	2025-04-10	Missed
69	106	301	2025-04-10	Missed

(a) Before Trigger Execution

after_trigger1

Attendance_ID	Member_ID	Class_ID	Date	Status
121	101	301	2025-04-10	Attended
122	102	301	2025-04-10	Attended
123	103	301	2025-04-10	Attended
124	105	301	2025-04-10	Missed
125	106	301	2025-04-10	Missed

(b) After Trigger Execution

Figure 1: Attendance Table View Before and After Trigger

Explanation of Figures: Before activating the trg_prevent_overbooking trigger, it was possible to insert 4 attendees to a class capacity 3, as shown in the image (a). After the trigger was implemented (image (b)), the system correctly prevents the 4th attendee from being added once the class capacity is reached.

2.2 Trigger #2: Verify Payment Amount (`trg_verify_payment_amount`)

We developed this trigger to ensure that members always pay exactly the correct amount based on the membership plan they are enrolled in. This protects the gym's billing integrity by eliminating overpayments, underpayments, or mismatched records.

Without this trigger, a staff member could accidentally (or intentionally) insert a payment amount that doesn't correspond to the actual plan cost stored in the system. This inconsistency could create issues during audits or financial reporting. By enforcing a match between payment amounts and official plan costs, the system guarantees accuracy.

Step-by-Step Functional Description:

1. **Look Up Plan ID:** First, the system identifies the membership plan of the member attempting to make a payment, using their `Member_ID` from the new payment row.
2. **Retrieve Plan Cost:** Next, it retrieves the official plan cost from the `Membership_Plan` table based on the member's `Plan_ID`.
3. **Compare Payment Amount:** The trigger compares the inserted payment amount (`NEW.Amount`) against the plan's official cost.
4. **Decision and Enforcement:** If the two amounts do not match exactly, the system raises an error and prevents the payment from being recorded. If the amount is correct, the insertion is accepted.

Property	Value
Timing / Event	BEFORE INSERT ON Payment
Granularity	Row-level (fires once per row inserted)
Business Rule	A member may only pay exactly the cost of their current plan.
Tables Involved	Payment (new row), Member (plan lookup), Membership_Plan (price lookup)

Table 2: Properties of `trg_verify_payment_amount`

Testing:

We tested the trigger with a member (Member_ID = 101) enrolled in a plan that costs 1400.00. First, we inserted a correct payment amount, which succeeded. Then, we attempted to insert an incorrect payment amount (1000.00), which the trigger correctly blocked with an error message.

Test Script for Payment Trigger

```
-- Attempt a correct payment: should succeed
INSERT INTO Payment (Member_ID, Amount, Date, Payment_Method)
VALUES (101, 1400.00, '2025-03-01', 'Credit Card'); -- SUCCESS

-- Attempt an incorrect payment: should fail
INSERT INTO Payment (Member_ID, Amount, Date, Payment_Method)
VALUES (101, 1000.00, '2025-03-01', 'Credit Card'); -- FAILED: triggers an
    error

-- Final SELECT to verify recorded payments
SELECT * FROM Payment
WHERE Member_ID = 101;
```

before_trigger2

Payment_ID	Member_ID	Amount	Date	Payment_Method
401	101	1400.00	2025-03-01	Credit Card
402	102	500.00	2025-02-15	PayPal
403	103	2700.00	2025-01-20	Debit Card
404	104	5200.00	2025-03-05	Credit Card
405	105	9700.00	2025-03-07	Bank Transfer
406	106	1800.00	2025-02-28	PayPal
407	107	4000.00	2025-01-15	Credit Card
408	108	6800.00	2025-02-10	Debit Card
409	109	1600.00	2025-03-12	Bank Transfer
410	110	8500.00	2025-03-18	PayPal
411	101	1400.00	2025-03-01	Credit Card
412	101	1000.00	2025-03-01	Credit Card

(a) Before Trigger Execution

after_trigger2

Payment_ID	Member_ID	Amount	Date	Payment_Method
401	101	1400.00	2025-03-01	Credit Card
402	102	500.00	2025-02-15	PayPal
403	103	2700.00	2025-01-20	Debit Card
404	104	5200.00	2025-03-05	Credit Card
405	105	9700.00	2025-03-07	Bank Transfer
406	106	1800.00	2025-02-28	PayPal
407	107	4000.00	2025-01-15	Credit Card
408	108	6800.00	2025-02-10	Debit Card
409	109	1600.00	2025-03-12	Bank Transfer
410	110	8500.00	2025-03-18	PayPal
411	101	1400.00	2025-03-01	Credit Card

(b) After Trigger Execution

Figure 2: Payment Table View Before and After Trigger

Explanation of Figures:

Before the trigger was active, it was possible to insert a payment for any arbitrary amount, even if it didn't match the member's actual plan cost (as shown in Figure (a)). After the trigger was implemented, incorrect payments are automatically rejected, ensuring only valid payments are recorded. In Figure (b), we see that the invalid payment attempt was blocked and only the correct one appears in the Payment table.

2.3 Why These Triggers Matter:

These triggers provide two crucial protections for our system:

- **Capacity Control:** Ensures classes do not become overcrowded, preserving member satisfaction and operational efficiency.
- **Payment Accuracy:** Guarantees correct financial transactions, supporting transparent and error-free billing processes.

3 Procedures

3.1 `sp_register_member`

We developed this stored procedure to simplify how new members are registered in the gym management system. Rather than issuing multiple SQL statements, users or applications can simply call this procedure to insert a new row into the `Member` table. By returning the auto-generated ID, it also allows the front-end interfaces to reference the newly created record without needing an additional query. This approach ensures consistent data insertion, reduces coding overhead, and sets the stage for future enhancements, such as validations or logging.

Step-by-Step Explanation

1. **Collect Member Details:** The procedure receives `p_name`, `p_age`, `p_gender`, `p_contact_info`, and `p_plan_id`.
2. **Execute INSERT:** These details are inserted into the `Member` table (matching columns to parameters).
3. **Fetch the Member ID:** After inserting, the procedure uses `LAST_INSERT_ID()` to return the new `Member_ID`.
4. **End:** The procedure finishes and hands back the generated `Member_ID` to the caller.

Procedure Code

```
-- Remove the procedure if it already exists
DROP PROCEDURE IF EXISTS sp_register_member;

DELIMITER $$

CREATE PROCEDURE sp_register_member (
    IN p_name VARCHAR(100), -- input: member's name
    IN p_age INT, -- input: member's age
    IN p_gender VARCHAR(10), -- input: member's gender
    IN p_contact_info VARCHAR(255), -- input: member's contact information
    IN p_plan_id INT -- input: the ID of the selected membership plan
)
BEGIN
    -- Insert the new member's details into the Member table
    INSERT INTO Member (
        Name, Age, Gender, Contact_Info, Plan_ID
    ) VALUES (
        p_name, p_age, p_gender, p_contact_info, p_plan_id
    );
```

```
    -- Return the auto-generated Member_ID of the new record
    SELECT LAST_INSERT_ID() AS new_member_id;
END$$
DELIMITER ;
```

Usage Example

```
-- Insert a new member named "Mert Can"
CALL sp_register_member(
    'Mert Can', -- name
    26, -- age
    'Male', -- gender
    'mert.can@sabanciuniv.edu',-- contact info
    2 -- plan id
);

-- Verify the new record
SELECT * FROM Member ORDER BY Member_ID DESC LIMIT 5;
```

Why We Need This Procedure

- **Consistency:** Ensures every new member is added with uniform data fields.
- **Convenience:** One stored procedure call replaces multiple SQL statements.
- **Future-Proofing:** Additional checks (e.g., plan validity or logging) can be added here without changing other parts of the app.

3.2 2. sp_add_payment

The `sp_add_payment` procedure centralizes the process of recording new payments in the database, making it both consistent and easy to maintain. When invoked, it gathers key data—such as the member's ID, the payment amount, the date of the transaction, and the payment method—and issues an `INSERT` statement into the `Payment` table. Immediately following this insertion, it retrieves the newly generated `Payment_ID` using the `LAST_INSERT_ID()` function and returns it alongside a confirmation message. By encapsulating these steps, `sp_add_payment` ensures that every payment follows the same structure and validations, thus preventing data discrepancies. In the broader application, this procedure provides a straightforward interface for payment handling, allowing front-end or service-layer components to log payments without manually writing SQL. Over time, developers can extend `sp_add_payment` with additional checks—for example, verifying member eligibility or logging transaction histories—making the gym's financial tracking more robust and reliable.

Step-by-Step Explanation

1. **Gather Payment Details:** The procedure receives `p_member_id`, `p_amount`, `p_date`, and `p_method`.
2. **Insert into `Payment`:** It uses these parameters to create a new record in the `Payment` table.
3. **Retrieve Payment ID:** With `LAST_INSERT_ID()`, the newly created `Payment_ID` is returned, alongside a success message.
4. **End:** The procedure terminates and provides the caller with both the `payment_id` and message.

Procedure Code

```
-- Remove the procedure if it already exists
DROP PROCEDURE IF EXISTS sp_add_payment;

DELIMITER $$

CREATE PROCEDURE sp_add_payment (
    IN p_member_id INT, -- input: ID of the member making the payment
    IN p_amount DECIMAL(10,2), -- input: amount being paid
    IN p_date DATE, -- input: date of the payment
    IN p_method VARCHAR(50) -- input: method of payment (e.g., Credit Card,
        PayPal)
)
BEGIN
    -- Insert the new payment details into the Payment table
    INSERT INTO Payment (
        Member_ID, Amount, Date, Payment_Method
    ) VALUES (
        p_member_id, p_amount, p_date, p_method
    );

    -- Return the auto-generated Payment_ID and a success message
    SELECT LAST_INSERT_ID() AS payment_id, 'Payment recorded successfully' AS
        message;
END$$
DELIMITER ;
```

Usage Example

```
-- Insert a new payment record
CALL sp_add_payment(
    103, -- Member_ID
    2700.00, -- Amount
    '2025-04-09', -- Date
    'Credit Card' -- Payment Method
);

-- Confirm the new payment
SELECT * FROM Payment ORDER BY Payment_ID DESC LIMIT 5;
```

Why We Need This Procedure

- **Standardization:** Ensures every payment record includes required fields.
- **Immediate Feedback:** Returns the `payment_id` plus a success message.
- **Extendability:** Future enhancements (e.g., verifying if the amount matches the plan's cost) can be added directly into this procedure.

Together, these procedures:

- Provide consistent methods for adding data (members and payments) into the system.
- Reduce duplicated SQL code across the application.
- Facilitate better maintenance and enable more powerful features (like validations, triggers, or logging) to be added in one place.