Sabancı University CS306 Database Systems
Fall 2025-2026

CS306 Database Systems Group Project Phase 2

Sibel Çelen 32629 sibel.celen@sabanciuniv.edu
Computer Science and Engineering & Industrial Engineering Program, Junior Year

Zişan Yeşil 32587 zisan.yesil@sabanciuniv.edu
Computer Science and Engineering & Management Program, Junior Year

Deniz Polat 32476 deniz.polat@sabanciuniv.edu
Mechatronics Engineering & Computer Science and Engineering Program, Junior Year

1- Introduction

The Pilates Studio Management System (PSMS) requires precise control over reservations, payments, and customer packages. Since these operations must follow strict rules to keep the studio's data consistent, the database must enforce many of these rules automatically. For that purpose, the project incorporates triggers and stored procedures, which transform the database into an active component that continuously protects data integrity and simplifies application logic.

Triggers are database-level reactive programs that automatically execute in response to specific events (such as INSERT or UPDATE). In this phase, we implemented three triggers, each aligned with a core business rule in the Pilates studio domain:

Prevent Overbooking a Session – Ensures that the number of reservations for any session never exceeds its capacity.

Automatic Payment Status Assignment – Automatically assigns a valid payment status (e.g., 'paid' or 'pending') based on the amount inserted.

Automatic Package Expiration Assignment – Automatically calculates and sets the expiration date for customer packages (purchase date + 30 days).

Together, these triggers make the database self-regulating and protect critical constraints.

Stored procedures, on the other hand, encapsulate reusable logic that the application can call when performing frequently used operations. They centralize complex workflows, reduce repetition, and offer reliable control over multi-step tasks. In this phase, we developed three stored procedures:

make_reservation – Safely inserts a reservation after checking capacity and returns the created reservation ID.

activate_instructor – Updates an instructor's active status and returns a confirmation message.

calculate_total_payments – Computes the total confirmed payments made by a specific customer.

By using stored procedures, PSMS gains robust automation for tasks such as reservation workflows, instructor activation, and financial reporting.

Overall, Phase II ensures that crucial operational rules are enforced directly within the database engine, resulting in a more reliable, consistent, and maintainable system. The following sections describe each trigger and stored procedure in detail, along with their execution evidence and system impact.

2- Triggers

Trigger 1: Prevent Overbooking of a Session

Purpose and Business Motivation

In the Pilates Studio Management System (PSMS), each Session has a fixed capacity, representing the maximum number of customers allowed to participate. Since reservations occur frequently and possibly concurrently, it is critical that the system prevents inserting more reservations than allowed.

Without an automated rule, overbooking could occur due to application-level bugs or concurrent inserts. To guarantee correctness, we enforced the rule inside the database using a trigger.

This trigger ensures:

- A session never exceeds its capacity.
- Only up to *capacity* number of confirmed reservations may be inserted.
- If a new reservation would exceed this limit, the database blocks the insertion and raises an error.

Thus, the trigger guarantees *data integrity*, regardless of how many different parts of the system insert reservations.

When the Trigger Fires

The trigger fires:

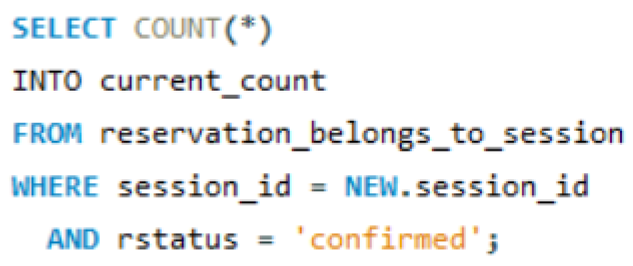
- Before an insert
- On the reservation_belongs_to_session table
- For each row

Meaning: every new reservation attempt is validated *before* MySQL writes it into the table.

Trigger Logic Explained Step-by-Step

Below is the conceptual workflow of the trigger:

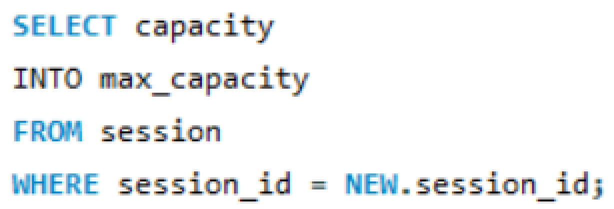
Count existing confirmed reservations for the given session



```
SELECT COUNT(*)
INTO current_count
FROM reservation_belongs_to_session
WHERE session_id = NEW.session_id
  AND rstatus = 'confirmed';
```

1. Only confirmed reservations matter for capacity constraints.

Retrieve the session's maximum capacity

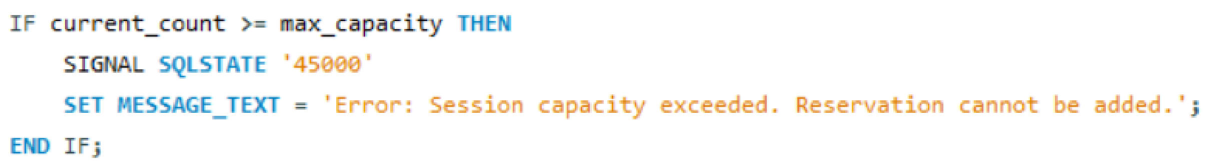


```
SELECT capacity
INTO max_capacity
FROM session
WHERE session_id = NEW.session_id;
```

Compare the current count with the maximum capacity

- If `current_count < capacity` → safe to insert
- If `current_count >= capacity` → block the operation

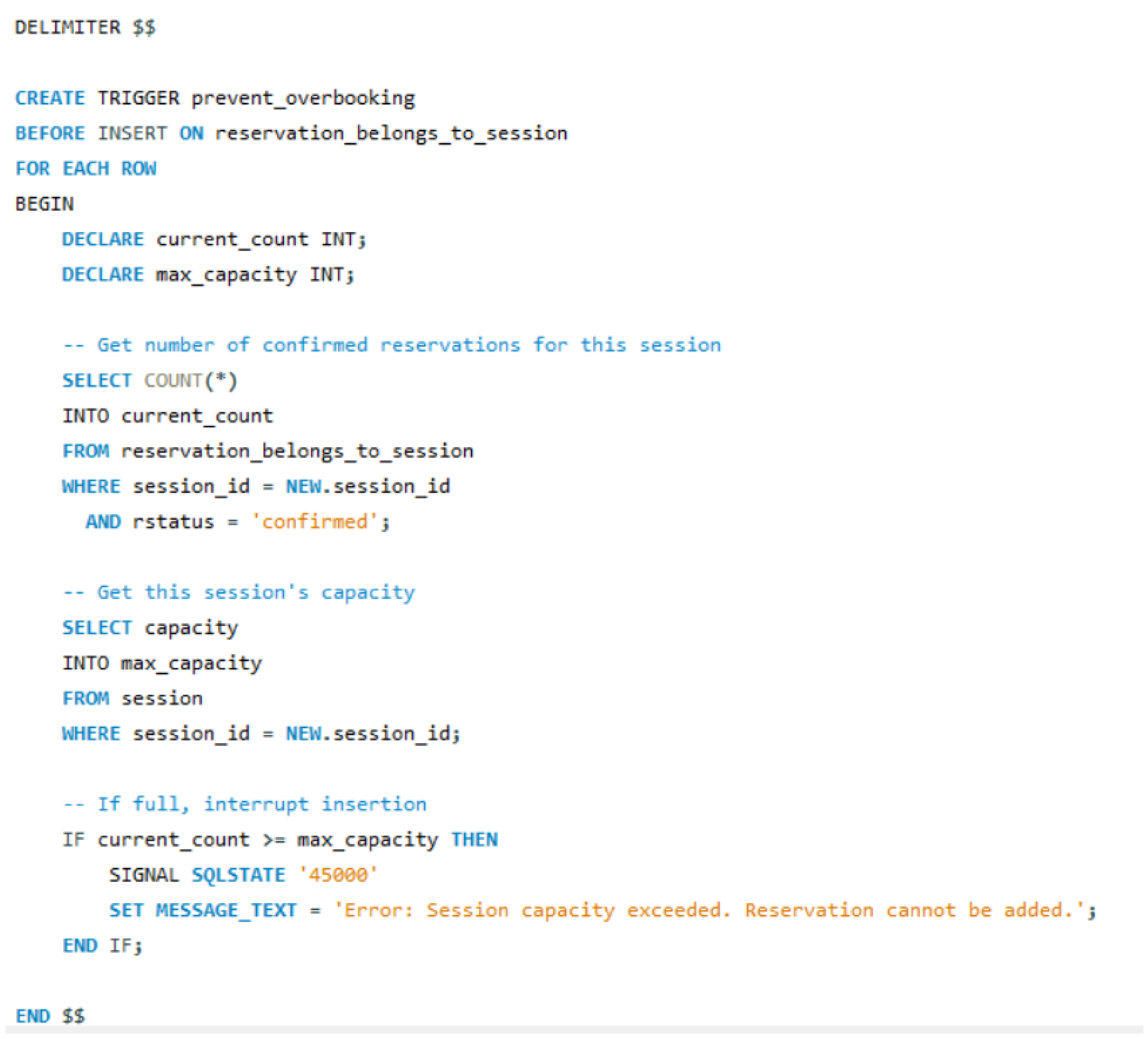
If full, raise a custom SQL error using:



```
IF current_count >= max_capacity THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Error: Session capacity exceeded. Reservation cannot be added.';
END IF;
```

This prevents overbooking even under heavy concurrency or malicious input.

SQL Script of the Trigger

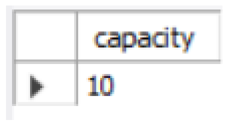


```
DELIMITER $$

CREATE TRIGGER prevent_overbooking
BEFORE INSERT ON reservation_belongs_to_session
FOR EACH ROW
BEGIN
    DECLARE current_count INT;
    DECLARE max_capacity INT;

    -- Get number of confirmed reservations for this session
    SELECT COUNT(*)
    INTO current_count
    FROM reservation_belongs_to_session
    WHERE session_id = NEW.session_id
      AND rstatus = 'confirmed';

    -- Get this session's capacity
    SELECT capacity
    INTO max_capacity
    FROM session
    WHERE session_id = NEW.session_id;

    -- If full, interrupt insertion
    IF current_count >= max_capacity THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Session capacity exceeded. Reservation cannot be added.';
    END IF;

END $$
```

Execution Evidence

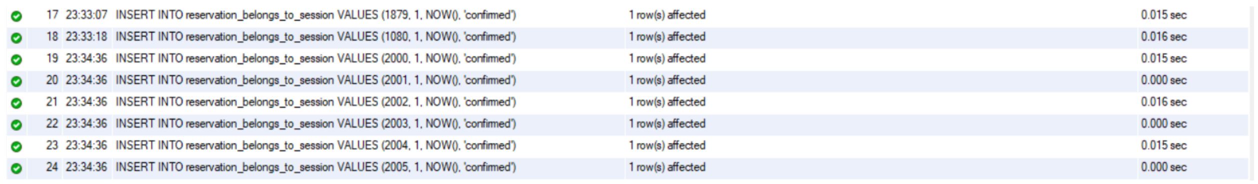
Before the Trigger Execution

The session with `session_id = 1` has a capacity of 10.



	capacity
▶	10

Below, nine confirmed reservations already existed; the tenth insert succeeds.

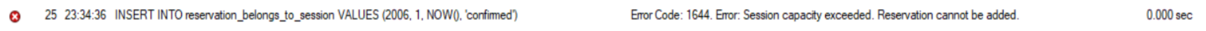


```
17  23:33:07  INSERT INTO reservation_belongs_to_session VALUES (1879, 1, NOW(), 'confirmed')   1 row(s) affected   0.015 sec
18  23:33:18  INSERT INTO reservation_belongs_to_session VALUES (1080, 1, NOW(), 'confirmed')   1 row(s) affected   0.016 sec
19  23:34:36  INSERT INTO reservation_belongs_to_session VALUES (2000, 1, NOW(), 'confirmed')   1 row(s) affected   0.015 sec
20  23:34:36  INSERT INTO reservation_belongs_to_session VALUES (2001, 1, NOW(), 'confirmed')   1 row(s) affected   0.000 sec
21  23:34:36  INSERT INTO reservation_belongs_to_session VALUES (2002, 1, NOW(), 'confirmed')   1 row(s) affected   0.016 sec
22  23:34:36  INSERT INTO reservation_belongs_to_session VALUES (2003, 1, NOW(), 'confirmed')   1 row(s) affected   0.000 sec
23  23:34:36  INSERT INTO reservation_belongs_to_session VALUES (2004, 1, NOW(), 'confirmed')   1 row(s) affected   0.015 sec
24  23:34:36  INSERT INTO reservation_belongs_to_session VALUES (2005, 1, NOW(), 'confirmed')   1 row(s) affected   0.000 sec
```

After the Trigger Execution

Once the 10th reservation is inserted, any additional attempt (11th reservation) results in an error generated by the trigger:

Error Code: 1644.
Error: Session capacity exceeded. Reservation cannot be added.



```
25  23:34:36  INSERT INTO reservation_belongs_to_session VALUES (2006, 1, NOW(), 'confirmed')   Error Code: 1644. Error: Session capacity exceeded. Reservation cannot be added.   0.000 sec
```

This confirms the trigger correctly prevents overbooking and protects database integrity.

Impact on the System

This trigger enforces a core studio rule automatically:

- Prevents human error and overbooking
- Ensures fairness (first-come-first-served)
- Makes the database self-protecting even if application code fails
- Supports concurrency control in busy systems

It removes the burden of checking capacity from the application layer and ensures that no invalid state can ever enter the database.

Trigger 2: Automatic Payment Status Assignment

Purpose and Business Motivation

In the Pilates Studio Management System (PSMS), every payment made by a customer is stored in the customer_buying_payment table with fields such as amount, payment method, status, and paid date.

However, manually setting the payment status (`pstatus`) is error-prone:

- A positive payment amount should logically indicate a successful payment.
- A non-positive or inconsistent amount should mark the payment as pending for manual review.

Without automation, the system could store payments with mismatched status values (e.g., a positive amount marked as pending), which would cause reporting inconsistencies and bookkeeping errors.

The purpose of this trigger is to ensure data correctness by automatically determining the payment status based on the **amount** inserted into the table.

When the Trigger Fires

The trigger is defined as:

- BEFORE INSERT
- On the customer_buying_payment table
- For each row

This ensures that every new payment record is validated and corrected *before* entering the database, guaranteeing consistency at the moment of data creation.

Trigger Logic Explained

The logic implemented in this trigger is straightforward but crucial for data integrity:

1. **Check the amount of the payment being inserted**
   - If `amount > 0`, then this is a real, completed payment.
   - If `amount <= 0`, then something is inconsistent (refund, error, or external issue).
2. **Automatically set the status**
   - `NEW.pstatus = 'paid'` when amount is positive
   - `NEW.pstatus = 'pending'` otherwise

This guarantees that all incoming payments have a valid and meaningful status.

SQL Script of the Trigger

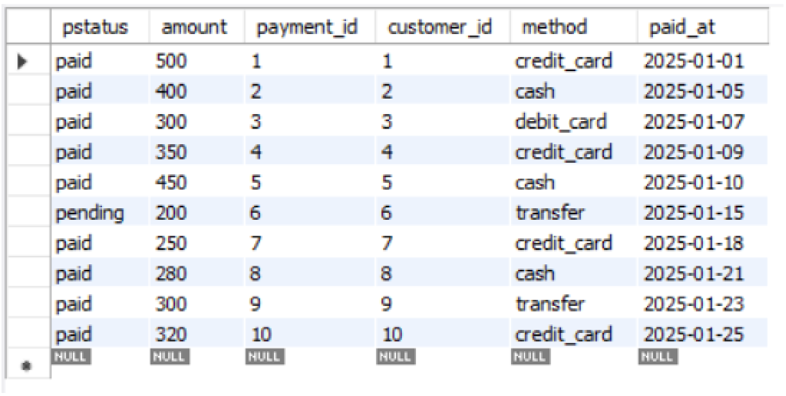
```
USE pilates_management_studio;

DELIMITER $$

CREATE TRIGGER auto_set_payment_status
BEFORE INSERT ON customer_buying_payment
FOR EACH ROW
BEGIN
    -- If amount is positive, auto-set status to 'paid'
    IF NEW.amount > 0 THEN
        SET NEW.pstatus = 'paid';
    ELSE
        SET NEW.pstatus = 'pending';
    END IF;
END $$

DELIMITER ;
```

Execution Evidence

Before Execution (Inserted Row Without Status)

Before the trigger fires, a new payment record is inserted with:

- `amount = -250`
- No status provided, so MySQL initially attempts to insert `NULL`



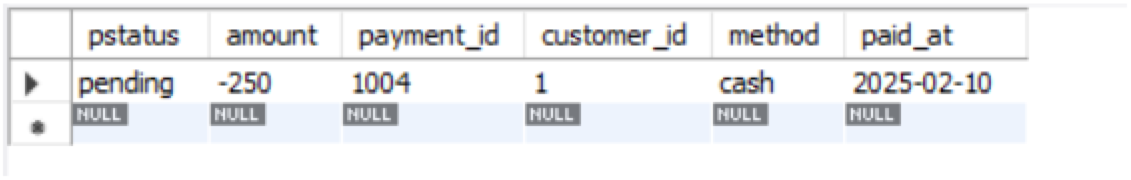
pstatus	amount	payment_id	customer_id	method	paid_at
paid	500	1	1	credit_card	2025-01-01
paid	400	2	2	cash	2025-01-05
paid	300	3	3	debit_card	2025-01-07
paid	350	4	4	credit_card	2025-01-09
paid	450	5	5	cash	2025-01-10
pending	200	6	6	transfer	2025-01-15
paid	250	7	7	credit_card	2025-01-18
paid	280	8	8	cash	2025-01-21
paid	300	9	9	transfer	2025-01-23
paid	320	10	10	credit_card	2025-01-25
NULL	NULL	NULL	NULL	NULL	NULL

After Trigger Execution

Because the amount is negative, the trigger automatically sets:

- `pstatus = 'pending'`

This result can be seen clearly in the updated table:



	pstatus	amount	payment_id	customer_id	method	paid_at
▶	pending	-250	1004	1	cash	2025-02-10
*	NULL	NULL	NULL	NULL	NULL	NULL

Payments with positive amounts were successfully stored with `pstatus = 'paid'` as shown in the full table.

Impact on the System

This trigger contributes to system reliability by:

- Ensuring that **pstatus always matches the payment amount**
- Preventing accidental mismatches by application developers or data-entry users
- Simplifying financial reporting logic
- Guaranteeing that the payment table always maintains semantic consistency

By automating this rule at the database level, the system becomes **safer, cleaner, and more maintainable**.

Trigger 3: Automatic Package Expiration Assignment

Purpose and Business Motivation

In PSMS, every customer may purchase class packages (e.g., 1-month, 10-session packages). Each package must have an **expiration date**, and this date is always determined by:

expiration date = purchase date + 30 days

If this calculation is left to the application layer or manual data entry, several problems may occur:

- Forgetting to set the expiration date

- Incorrect expiration values due to human error
- Inconsistent expiration policies between different parts of the application

To ensure uniformity, correctness, and reliability, the expiration date is computed **automatically by the database** using a trigger.

This guarantees that every package behaves according to the studio's business rule without exception.

When the Trigger Fires

The trigger fires:

- BEFORE INSERT
- On the customer_purchases_Cust_package table
- For each row

Because this is a *weak entity* depending on `Customer`, it is especially important that its dates are automatically controlled in the database.

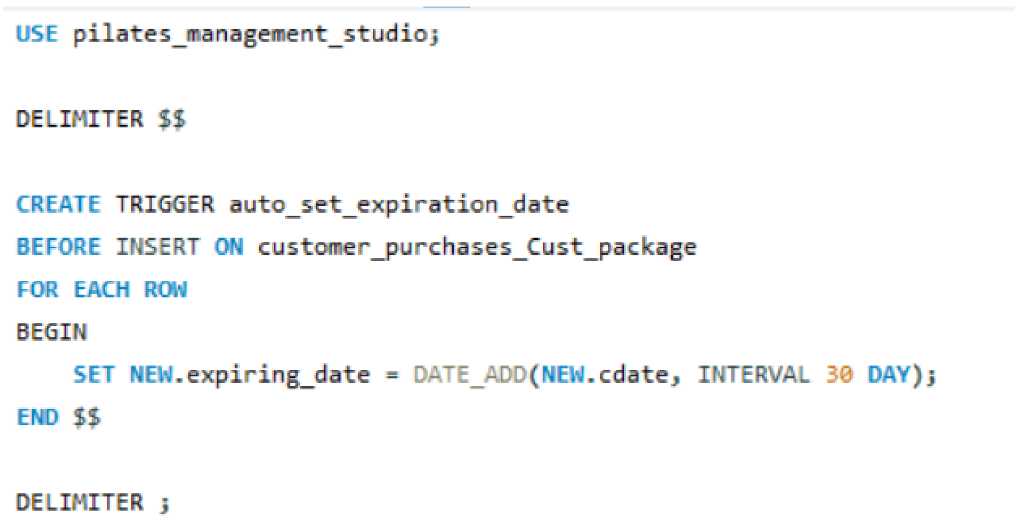
Trigger Logic Explained

The logic of the trigger is simple and deterministic:

1. Take the variable `NEW.cdate` (the package purchase date).
2. Add 30 days to it using MySQL's `DATE_ADD` function.
3. Automatically assign this value to `NEW.expiring_date`.

This ensures that every new package purchased by a customer automatically receives an accurate expiration date.

SQL Script of the Trigger



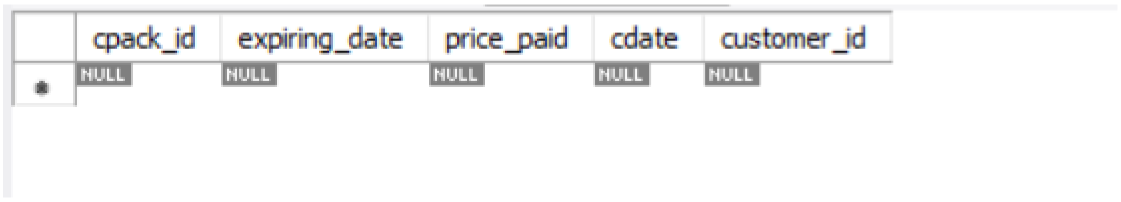
```
USE pilates_management_studio;

DELIMITER $$

CREATE TRIGGER auto_set_expiration_date
BEFORE INSERT ON customer_purchases_Cust_package
FOR EACH ROW
BEGIN
    SET NEW.expiring_date = DATE_ADD(NEW.cdate, INTERVAL 30 DAY);
END $$

DELIMITER ;
```

Execution Evidence

Before the Trigger Execution

A new package purchase is entered with:

- `cdate = 2025-02-01`
- No expiration date supplied (NULL)



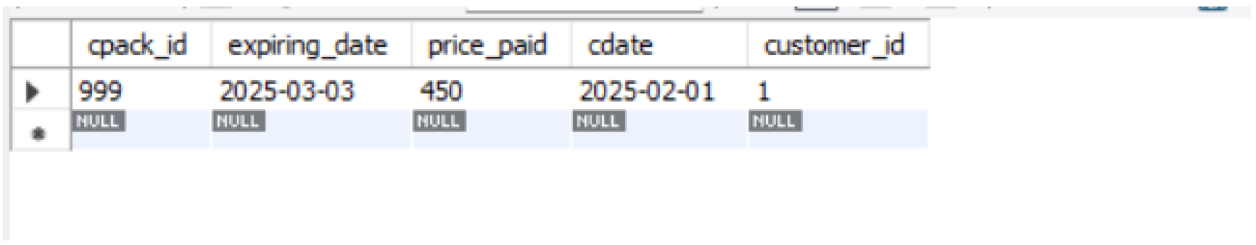
	cpack_id	expiring_date	price_paid	cdate	customer_id
*	NULL	NULL	NULL	NULL	NULL

After the Trigger Execution

The trigger automatically computes:

- `expiring_date = 2025-02-01 + 30 days = 2025-03-03`

The final row stored in the database:



cpack_id	expiring_date	price_paid	cdate	customer_id
999	2025-03-03	450	2025-02-01	1
NULL	NULL	NULL	NULL	NULL

The expiration date is correct, consistent, and automatically generated.

Impact on the System

This trigger strengthens the system by:

- Enforcing consistent business rules for expiration
- Eliminating manual calculations or user mistakes
- Making the weak entity (`Cust_package`) fully self-maintained
- Ensuring all future analyses (e.g., expired packages, active packages) rely on correct data

The automation guarantees that no invalid or inconsistent package expiration can ever enter the database.

3- Procedures

Stored Procedure 1: `make_reservation`

Purpose and Business Motivation

Reservations are a core part of the Pilates Studio Management System (PSMS). Customers book sessions daily, and the system must ensure:

- Reservations are inserted correctly
- Session capacities are respected
- Overbooking is prevented
- Reservation times reflect the exact moment of booking

Although application-side validation helps, the safest and most reliable way to enforce consistent reservation logic is at the **database level**, through a stored procedure.

The procedure `make_reservation` automates and secures this entire operation by:

1. Checking whether the session still has available capacity
2. Automatically inserting the reservation when allowed
3. Returning the newly created reservation ID
4. Raising a clear SQL error when the session is full

This ensures consistency even under concurrent usage or unexpected application behavior.

Inputs

`p_customer_id:`	INT	Customer who is making the reservation
`p_session_id:`	INT	The session the customer wants to book

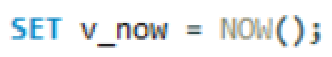
Outputs

- **If successful:**
  Returns the auto-generated `reservation_id`.
- **If capacity is full:**
  Raises an error:

  Error: Session is full

Procedure Logic Explained

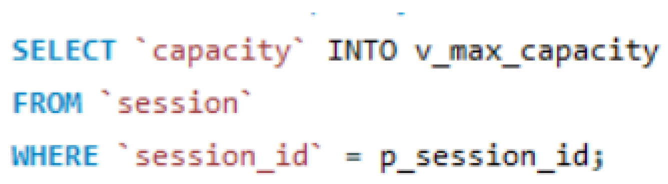
1. Store the current timestamp



```
SET v_now = NOW();
```

Used as the reservation time.

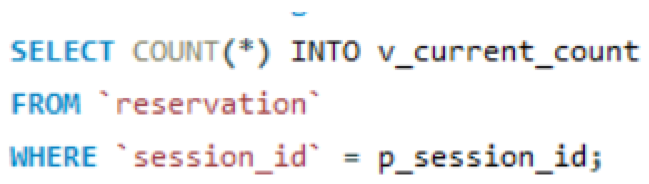
2. Retrieve the capacity of the session



```
SELECT `capacity` INTO v_max_capacity
FROM `session`
WHERE `session_id` = p_session_id;
```

This ensures the procedure uses the correct capacity from the database.

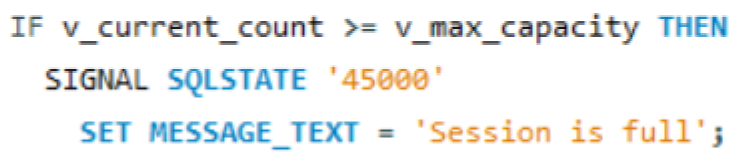
3. Count existing reservations for that session



```
SELECT COUNT(*) INTO v_current_count
FROM `reservation`
WHERE `session_id` = p_session_id;
```

This determines how many customers have already booked the session.

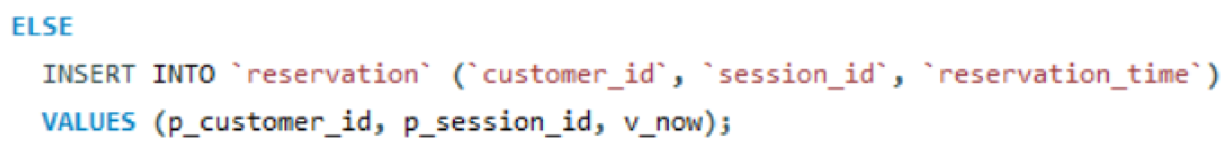
4. If session is full, raise an error



```
IF v_current_count >= v_max_capacity THEN
  SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Session is full';
```

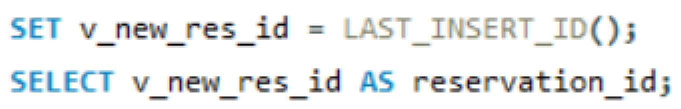
This safely prevents adding invalid data.

5. Otherwise, insert the reservation



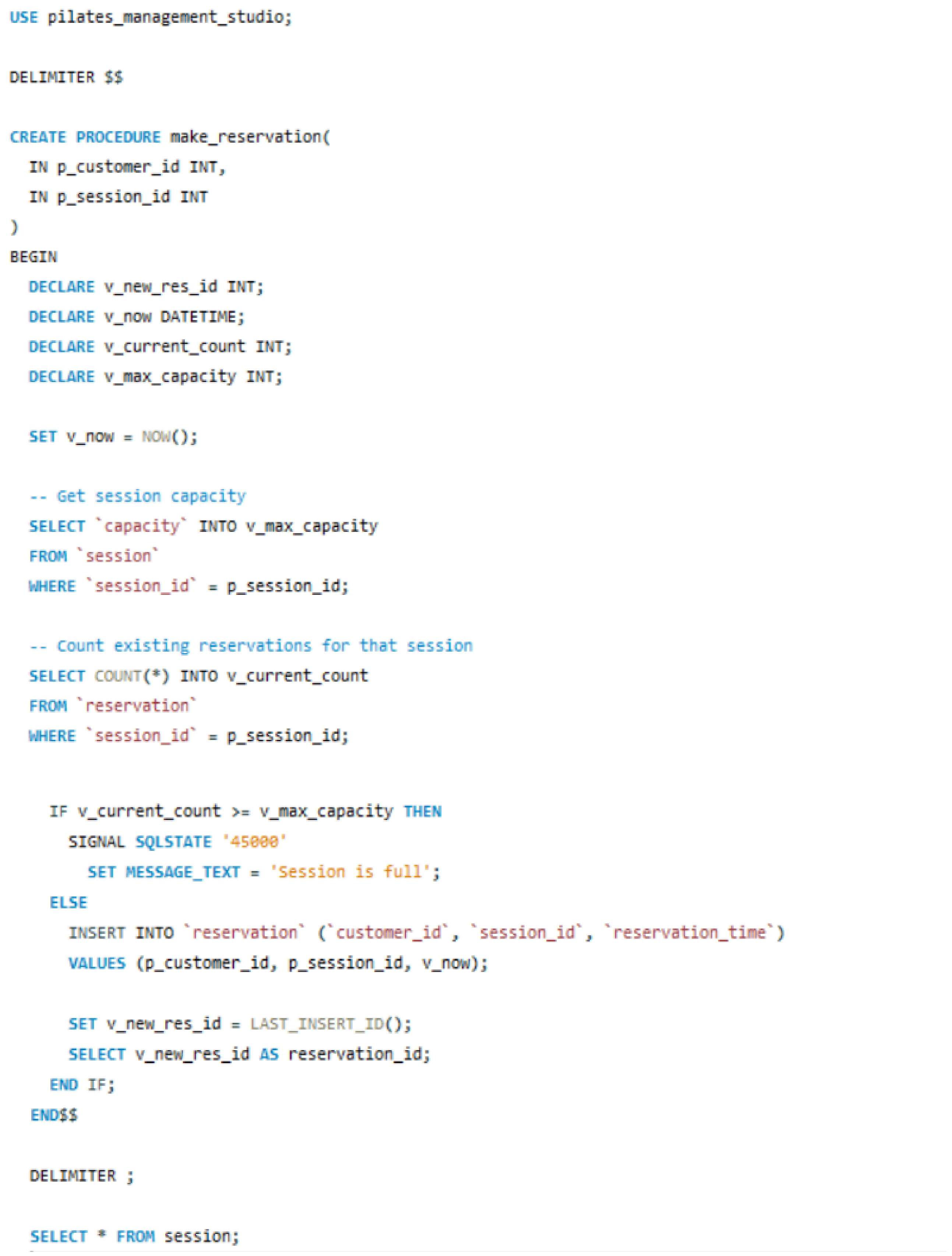
```
ELSE
  INSERT INTO `reservation` (`customer_id`, `session_id`, `reservation_time`)
  VALUES (p_customer_id, p_session_id, v_now);
```

And retrieve the new ID:



```
SET v_new_res_id = LAST_INSERT_ID();
SELECT v_new_res_id AS reservation_id;
```

Stored Procedure Code

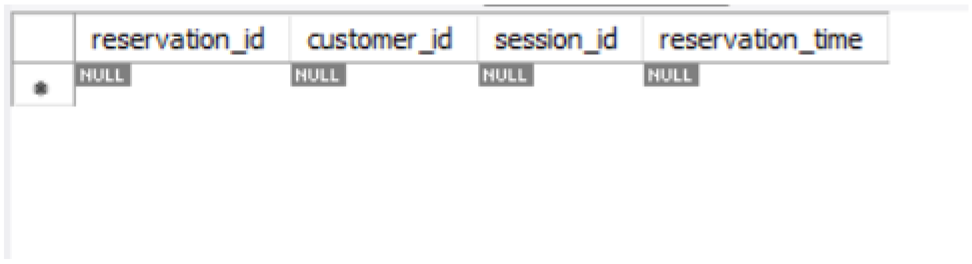


```sql
USE pilates_management_studio;

DELIMITER $$

CREATE PROCEDURE make_reservation(
  IN p_customer_id INT,
  IN p_session_id INT
)
BEGIN
  DECLARE v_new_res_id INT;
  DECLARE v_now DATETIME;
  DECLARE v_current_count INT;
  DECLARE v_max_capacity INT;

  SET v_now = NOW();

  -- Get session capacity
  SELECT `capacity` INTO v_max_capacity
  FROM `session`
  WHERE `session_id` = p_session_id;

  -- Count existing reservations for that session
  SELECT COUNT(*) INTO v_current_count
  FROM `reservation`
  WHERE `session_id` = p_session_id;


    IF v_current_count >= v_max_capacity THEN
      SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Session is full';
    ELSE
      INSERT INTO `reservation` (`customer_id`, `session_id`, `reservation_time`)
      VALUES (p_customer_id, p_session_id, v_now);

      SET v_new_res_id = LAST_INSERT_ID();
      SELECT v_new_res_id AS reservation_id;
    END IF;
  END$$

  DELIMITER ;

  SELECT * FROM session;
```

Execution Evidence

1. Before calling the procedure

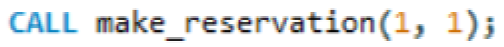
The table is initially **empty**, showing that no reservations have yet been made:



	reservation_id	customer_id	session_id	reservation_time
●	NULL	NULL	NULL	NULL

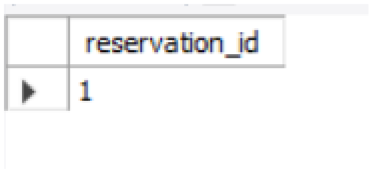
2. Procedure call output

Calling:



```
CALL make_reservation(1, 1);
```

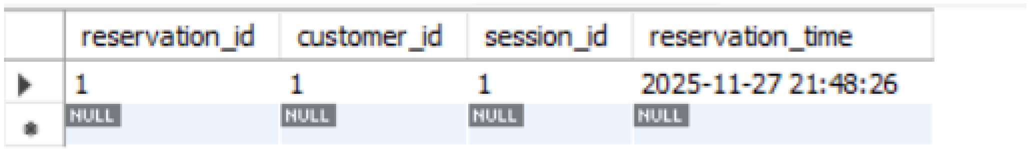
The system returns the new reservation ID:



	reservation_id
▶	1

3. After executing the procedure

The reservation table now contains the newly inserted row:



	reservation_id	customer_id	session_id	reservation_time
▶	1	1	1	2025-11-27 21:48:26
●	NULL	NULL	NULL	NULL

This confirms that the stored procedure correctly inserted the reservation with accurate timestamp and session reference.

Impact on the System

This stored procedure ensures:

- Consistent reservation handling across the entire system
- Server-side enforcement of session capacity
- Accurate timestamping
- Safer concurrent reservation operations
- Better maintainability by centralizing reservation logic in one place

It provides a clean interface for the application and protects database integrity.

Stored Procedure 2: `activate_instructor`

Purpose and Business Motivation

In PSMS, instructors can be either **active** (currently teaching sessions) or **inactive** (on break, left the studio, unavailable, etc.). Managing this status is essential for:

- Assigning sessions only to active instructors
- Preventing scheduling errors
- Updating instructor availability efficiently
- Providing clear administrative control

Instead of manually editing instructor rows, an error-prone and inefficient process, the system uses a stored procedure to update instructor status safely and consistently.

The procedure **activate_instructor** centralizes this logic and provides a clean textual confirmation message after updating the record.

Inputs

`p_instructor_id:`	INT	The instructor whose status is being updated

`p_new_status:`	BOOLEAN	1 = active, `0` = inactive

Outputs

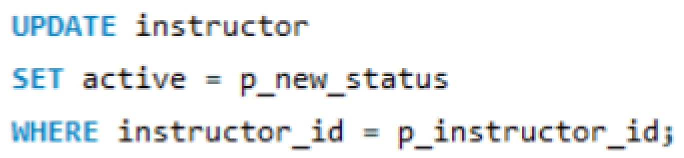
- A textual confirmation message, e.g.:

  “Instructor 5 status updated to 1”

This message allows the front-end or admin tools to display success feedback.

Procedure Logic Explained Step-by-Step

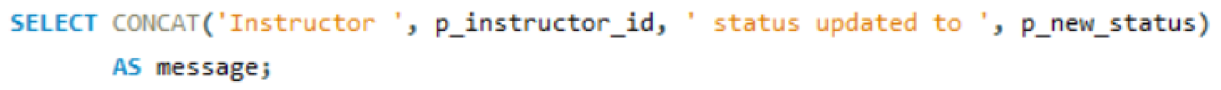
1. Update the instructor’s active field



```sql
UPDATE instructor
SET active = p_new_status
WHERE instructor_id = p_instructor_id;
```

This directly updates the instructor’s availability.

2. Return a confirmation message



```sql
SELECT CONCAT('Instructor ', p_instructor_id, ' status updated to ', p_new_status)
       AS message;
```

This is helpful for logging, UI notifications, or debugging.

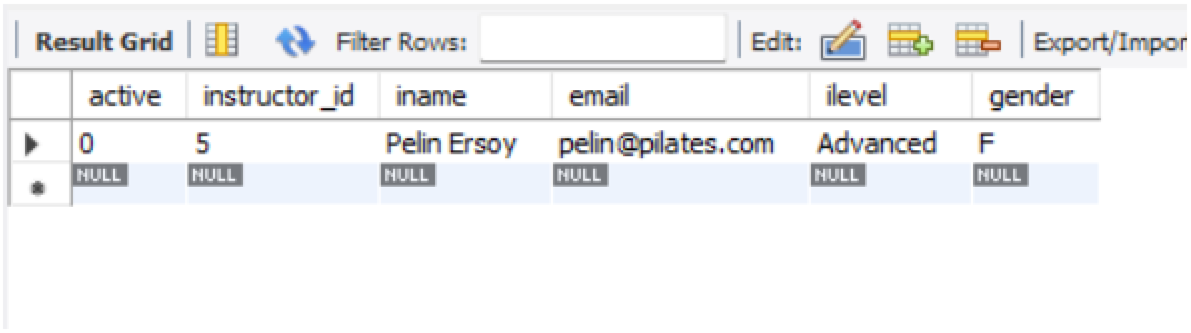
Stored Procedure Code

```
DROP PROCEDURE IF EXISTS activate_instructor;

DELIMITER $$

CREATE PROCEDURE activate_instructor (
    IN p_instructor_id INT,
    IN p_new_status BOOLEAN
)
BEGIN
    UPDATE instructor
    SET active = p_new_status
    WHERE instructor_id = p_instructor_id;

    SELECT CONCAT('Instructor ', p_instructor_id, ' status updated to ', p_new_status)
           AS message;
END$$

DELIMITER ;

SELECT * FROM instructor WHERE instructor_id = 5;

CALL activate_instructor(5, 1);

SELECT * FROM instructor WHERE instructor_id = 5;
```

Execution Evidence

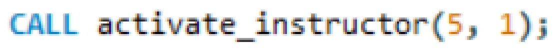
Before Calling the Procedure

Instructor with ID **5** is initially **inactive (active = 0)**:



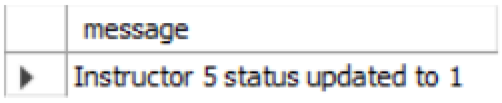
active	instructor_id	iname	email	ilevel	gender
0	5	Pelin Ersoy	pelin@pilates.com	Advanced	F
NULL	NULL	NULL	NULL	NULL	NULL

Procedure Call



```
CALL activate_instructor(5, 1);
```

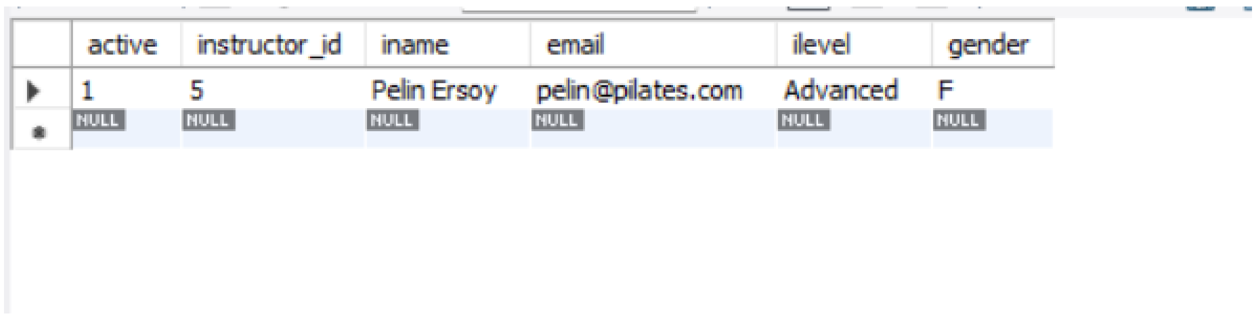
The system responds with:



	message
▶	Instructor 5 status updated to 1

After Execution (Status Updated)

The instructor's status is now updated to **active (1)**:



	active	instructor_id	iname	email	ilevel	gender
▶	1	5	Pelin Ersoy	pelin@pilates.com	Advanced	F
*	NULL	NULL	NULL	NULL	NULL	NULL

Impact on the System

This stored procedure improves system manageability by:

- Providing secure and controlled status updates
- Removing the need for manual SQL edits
- Ensuring admin operations are consistent and traceable
- Integrating seamlessly with the scheduling subsystem
  (only active instructors can teach sessions)

It creates a more maintainable and less error-prone workflow for administrative users.

Stored Procedure 3: `calculate_total_payments`

Purpose and Business Motivation

A common requirement in PSMS is to generate customer-level financial summaries, for example:

- How much a customer has paid to the studio in total
- Generating monthly revenue reports
- Displaying a customer's payment history inside the application
- Supporting analytics such as "top spending customers"

Because payment data may include both valid positive amounts and negative/pending entries (refunds, failed transactions, or reversed operations), the system must only consider successful, paid payments.

The stored procedure `calculate_total_payments` provides a clean, reusable, database-level method for computing the total confirmed revenue from any given customer.

Inputs

Parameter	Type	Meaning
`p_customer_id`	INT	The customer whose payment total is requested

Outputs

The procedure returns a single-row result set:

customer	total_payments
customer_id	total of all "paid" amounts

Procedure Logic Explained Step-by-Step

1. Initialize a working variable

```
DECLARE v_total_amount INT DEFAULT 0;
```

This variable will hold the final computed sum.

2. Calculate the sum of all “paid” payments

```
SELECT IFNULL(SUM(amount), 0)
INTO v_total_amount
FROM customer_buying_payment
WHERE customer_id = p_customer_id
  AND pstatus = 'paid';
```

Key points:

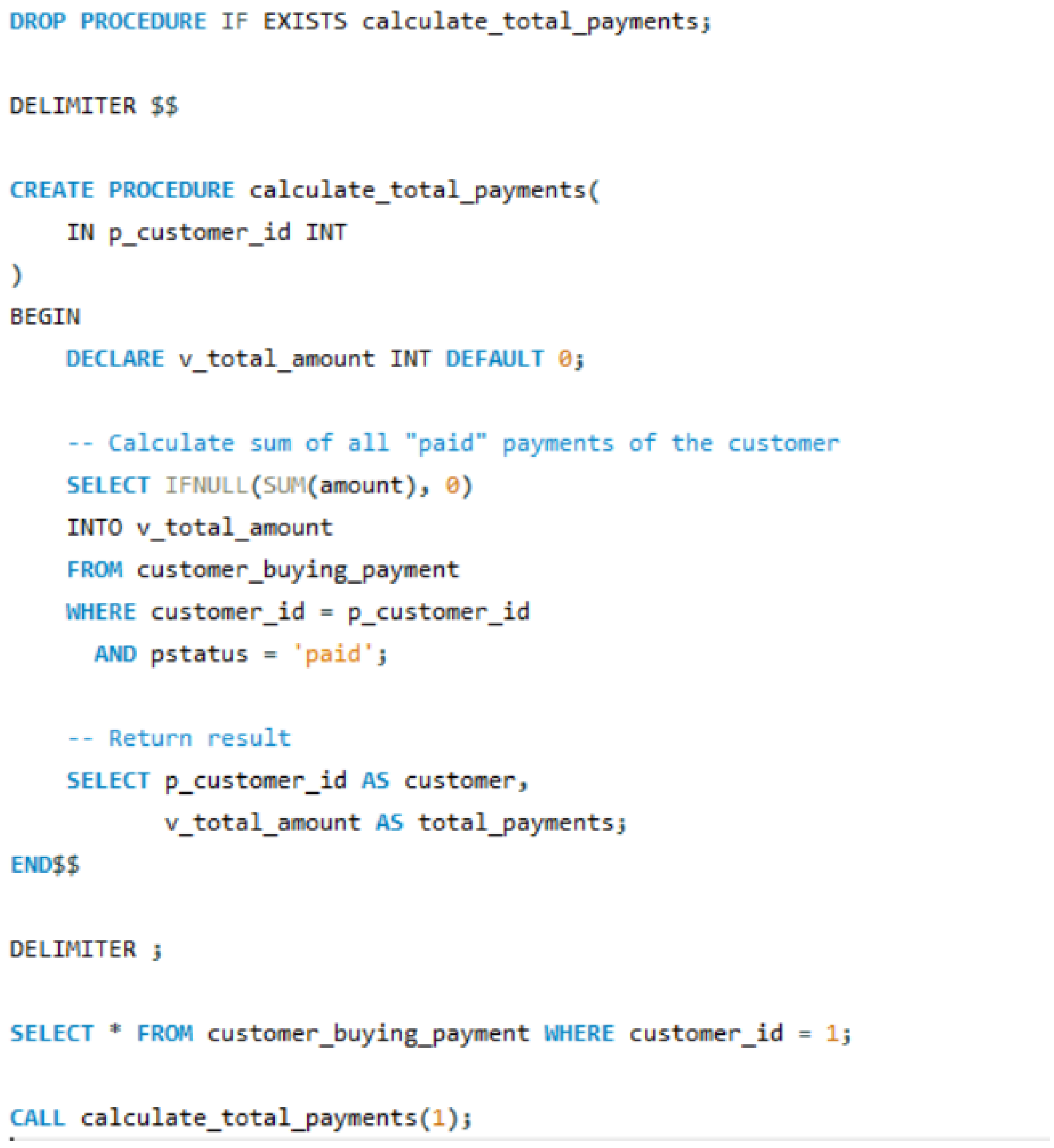
- Only **paid** payments count toward total revenue
- Pending, negative, or failed payments are ignored
- `IFNULL` ensures the result is **0** if the customer has no paid transactions
- This step ensures financial accuracy and consistency regardless of the number of records

3. Return the final result

```
SELECT p_customer_id AS customer,
       v_total_amount AS total_payments;
```

This allows the admin interface or reporting system to display the result directly.

Stored Procedure Code

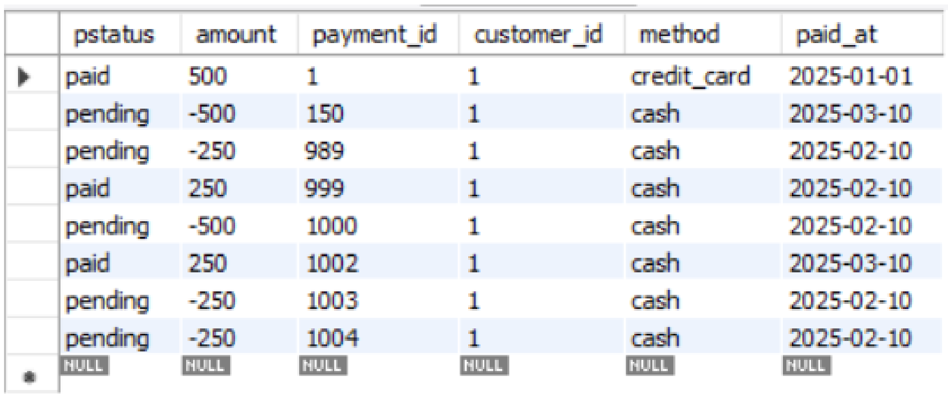


```sql
DROP PROCEDURE IF EXISTS calculate_total_payments;

DELIMITER $$

CREATE PROCEDURE calculate_total_payments(
    IN p_customer_id INT
)
BEGIN
    DECLARE v_total_amount INT DEFAULT 0;

    -- Calculate sum of all "paid" payments of the customer
    SELECT IFNULL(SUM(amount), 0)
    INTO v_total_amount
    FROM customer_buying_payment
    WHERE customer_id = p_customer_id
      AND pstatus = 'paid';

    -- Return result
    SELECT p_customer_id AS customer,
           v_total_amount AS total_payments;
END$$

DELIMITER ;

SELECT * FROM customer_buying_payment WHERE customer_id = 1;

CALL calculate_total_payments(1);
```

Execution Evidence

Before Calling the Procedure

Payments belonging to `customer_id = 1` appear as follows:



pstatus	amount	payment_id	customer_id	method	paid_at
paid	500	1	1	credit_card	2025-01-01
pending	-500	150	1	cash	2025-03-10
pending	-250	989	1	cash	2025-02-10
paid	250	999	1	cash	2025-02-10
pending	-500	1000	1	cash	2025-02-10
paid	250	1002	1	cash	2025-03-10
pending	-250	1003	1	cash	2025-02-10
pending	-250	1004	1	cash	2025-02-10
NULL	NULL	NULL	NULL	NULL	NULL

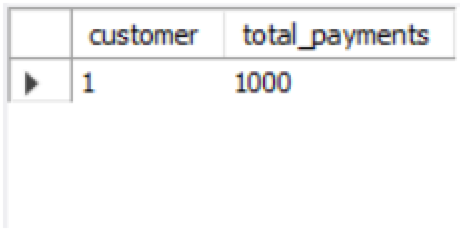
- There are several *paid* entries (positive amounts)
- Several negative or “pending” entries also exist (ignored by the procedure)

Procedure Call

```
CALL calculate_total_payments(1);
```

After Execution (Correct Total Computed)

The procedure successfully sums only the **paid** amounts and returns:



customer	total_payments
1	1000

In this example, customer 1 has **total confirmed payments = 1000**.

Impact on the System

This stored procedure adds several important benefits:

- Ensures **consistent financial calculations** across all parts of the application
- Encapsulates logic so front-end developers do not need to manually write sums
- Prevents mistakes such as:

    - Including pending payments
    - Including negative/invalid entries
    - Double-counting revenues
- Makes reporting and analytics significantly simpler
- Allows clean integration into dashboards or automated email statements

With this procedure, PSMS gains a reliable, single source of truth for customer-level payment totals.