**Team member:**

**Kyle Jia, jj3708**

**Junxiao Wang, jw8193**

**Nikhil Diddee, nd2564**

**Milestone 3: Database Setup and SQL**

**1. Database Server**

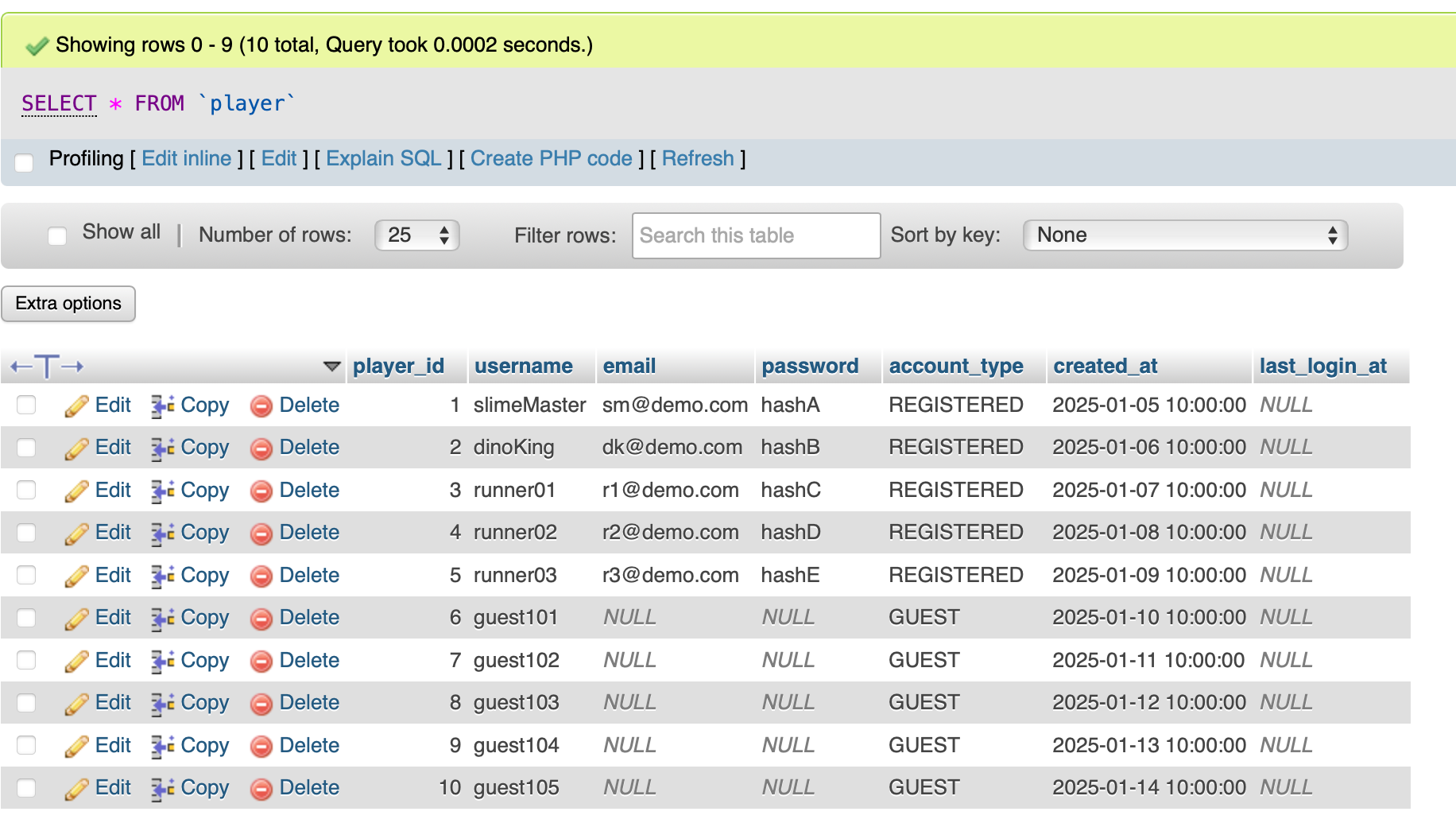
We are hosting our database on phpMyAdmin running locally. The database name is slime\_runner\_db and it contains all 10 tables required for our Slime Runner endless runner game project.

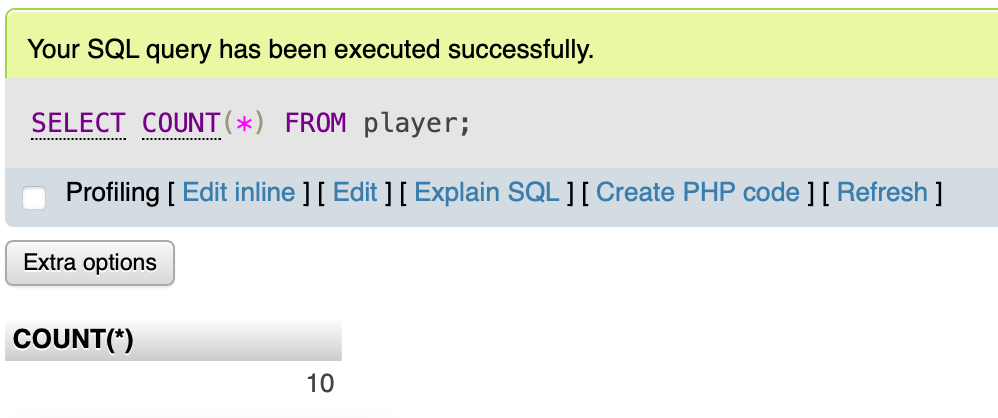
**2. Database Tables and Sample Data**

This section demonstrates that all required tables have been successfully created and populated with sufficient realistic data.

**2.1 Player Table**

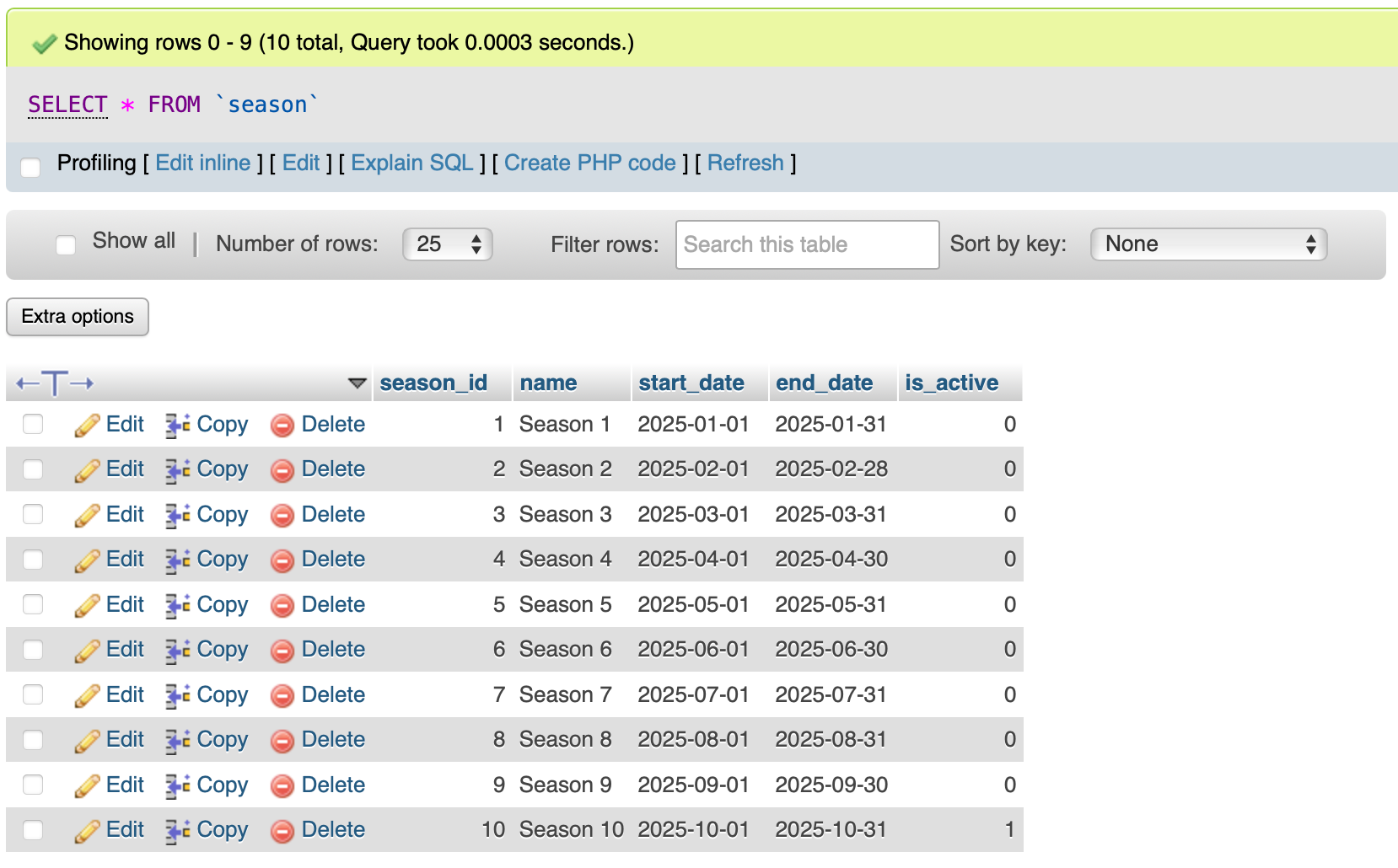
The player table stores information about both registered users and guest players. It contains 10 rows with usernames, emails, passwords, account types, and timestamps. Registered players have email and password while guest accounts have NULL values for these fields.

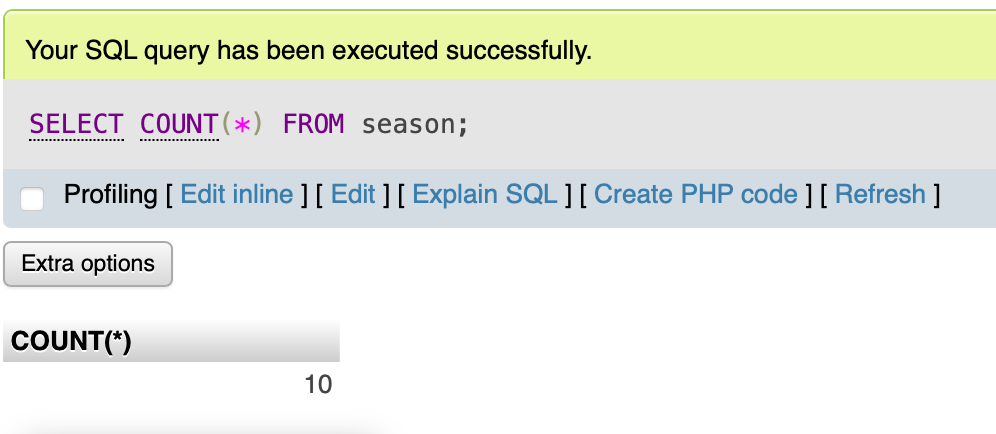
****

****

**2.2 Season Table**

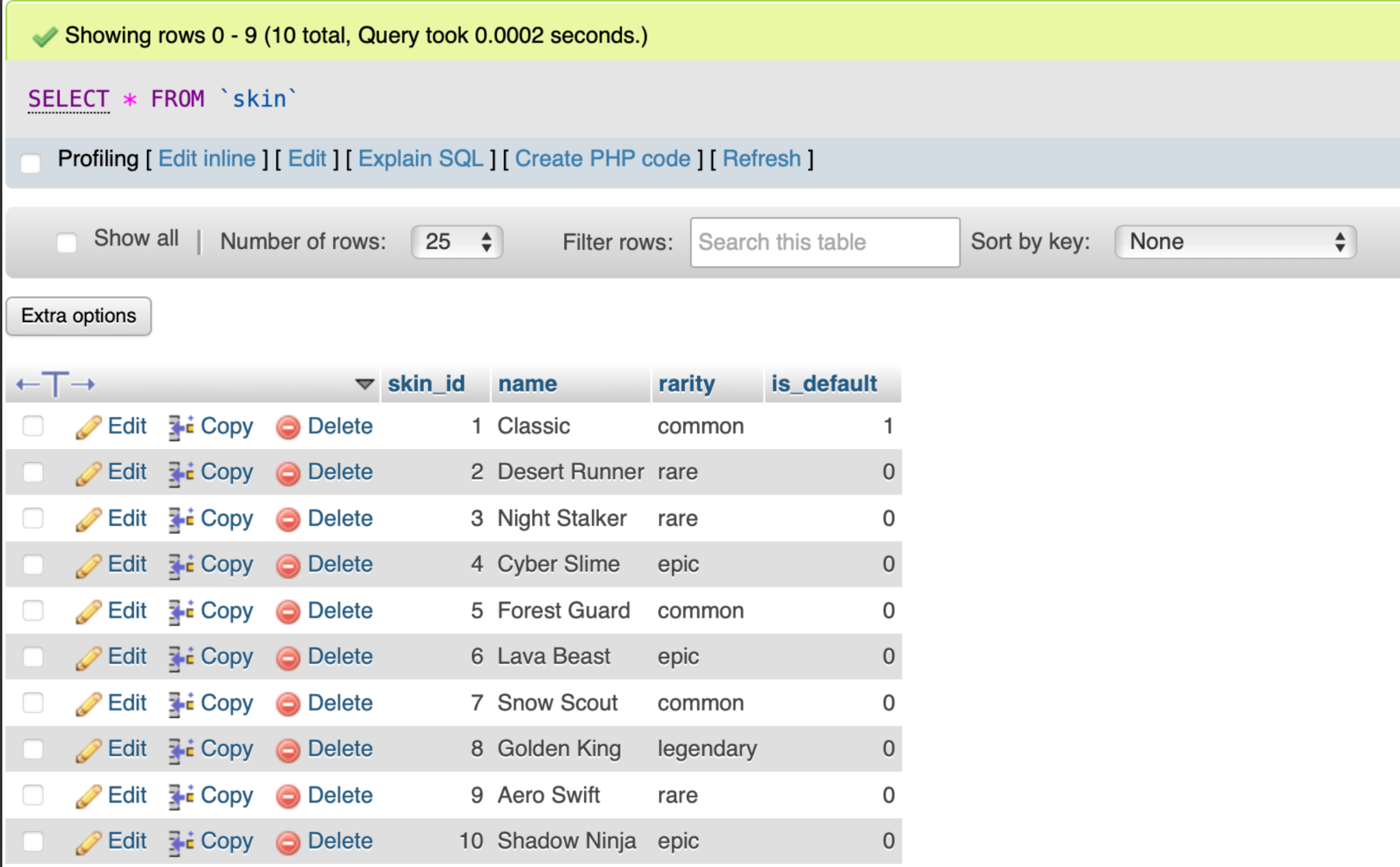
The season table tracks different competition periods spanning January to October 2025. It contains 10 rows with season names, start dates, end dates, and an is\_active flag indicating Season 10 is currently active.

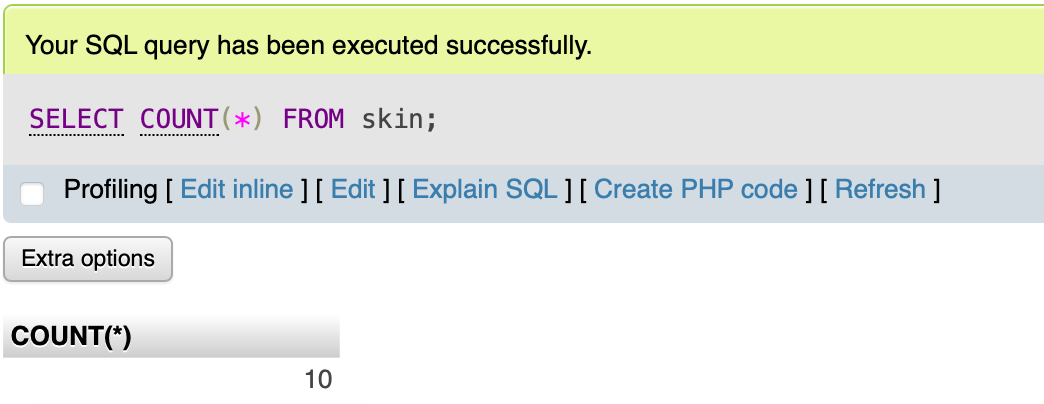
****

****

**2.3 Skin Table**

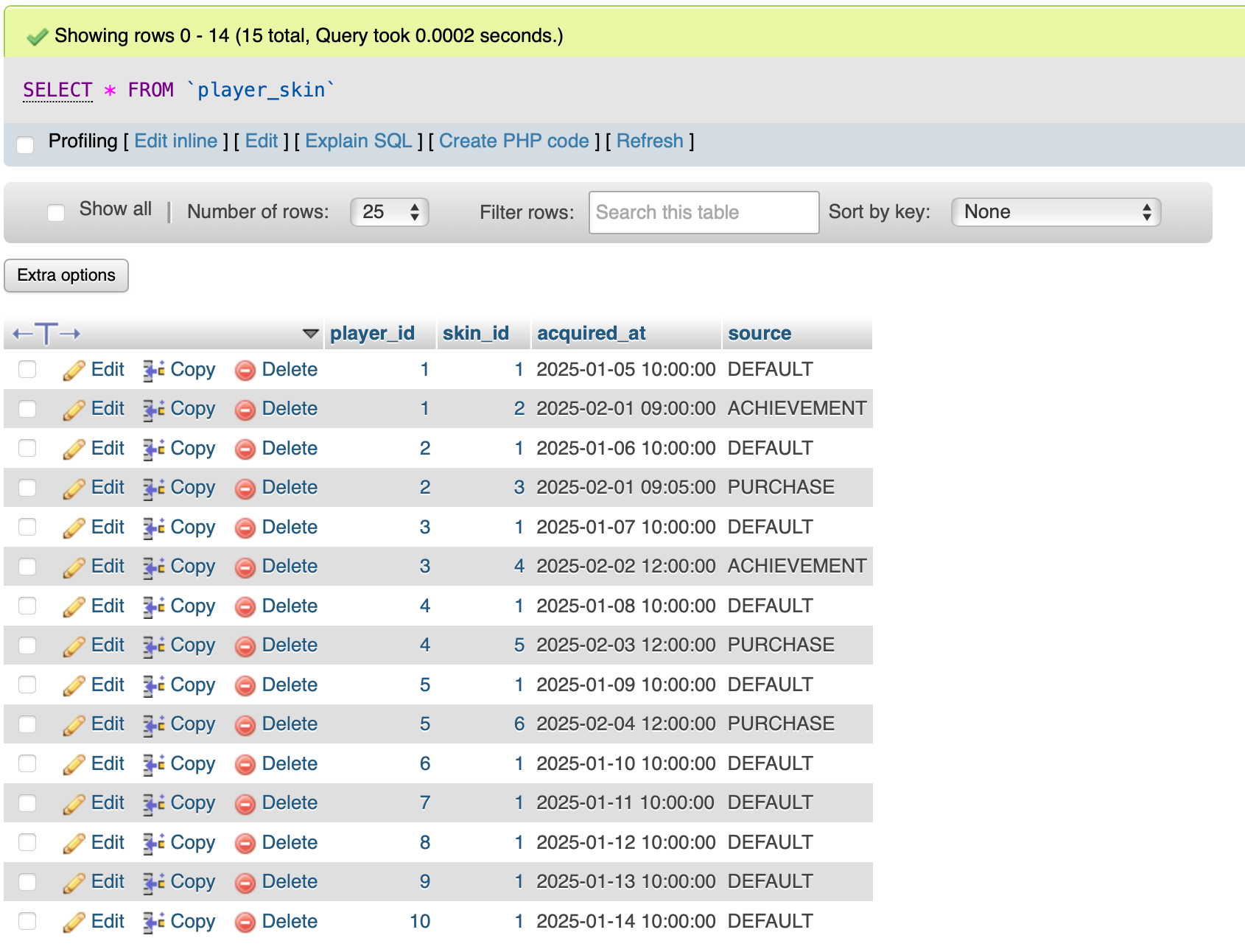
The skin table contains 10 different character skins with varying rarity levels from common to legendary. The Classic skin is marked as the default skin that all players own automatically.



****

**2.4 PlayerSkin Table**

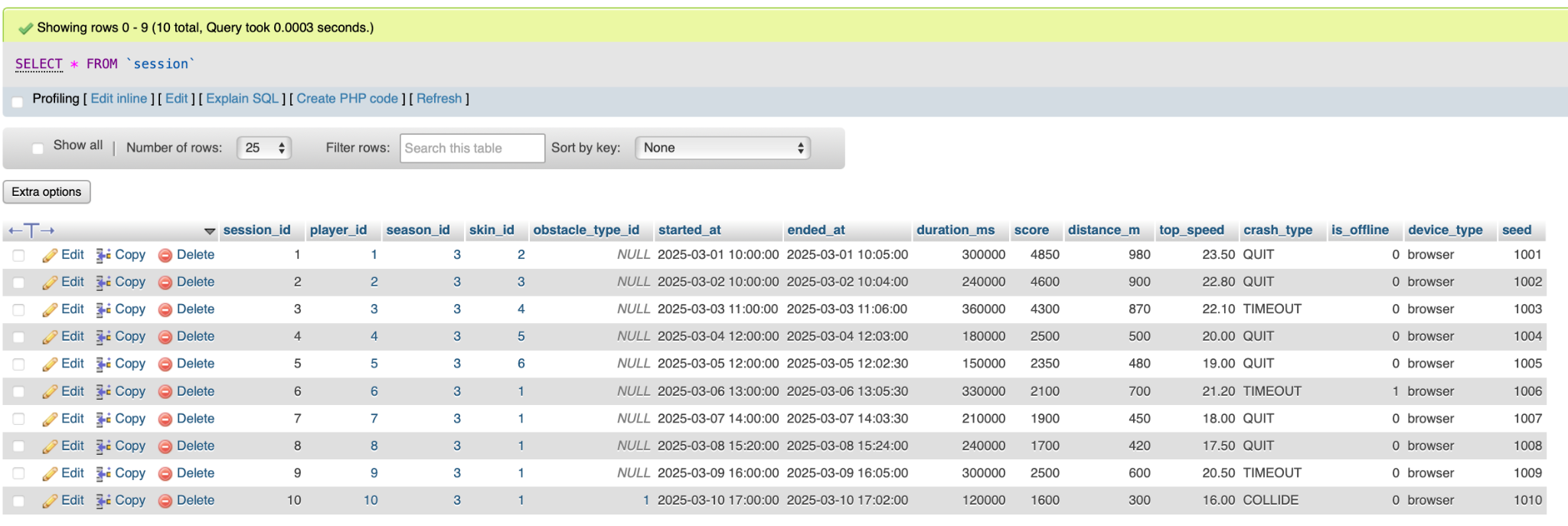
The player\_skin table contains 15 rows tracking which skins each player owns and how they acquired them. All players own the default skin and some have acquired additional skins through achievements or purchases.

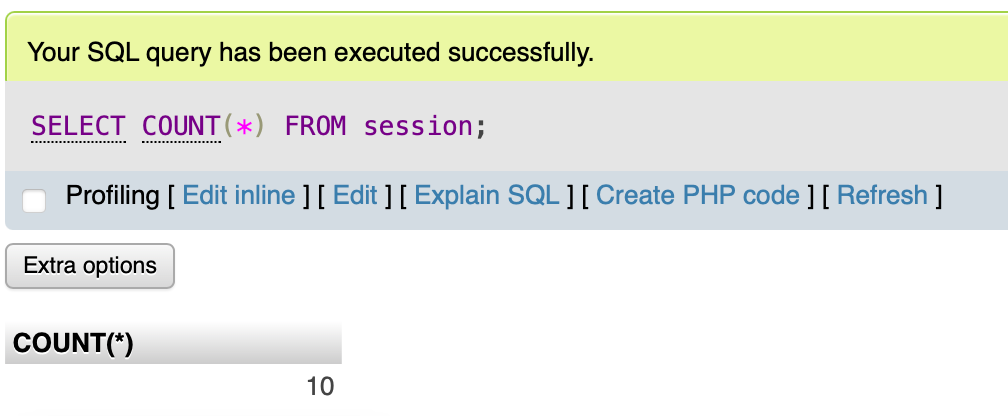
****

****

**2.5 Session Table**

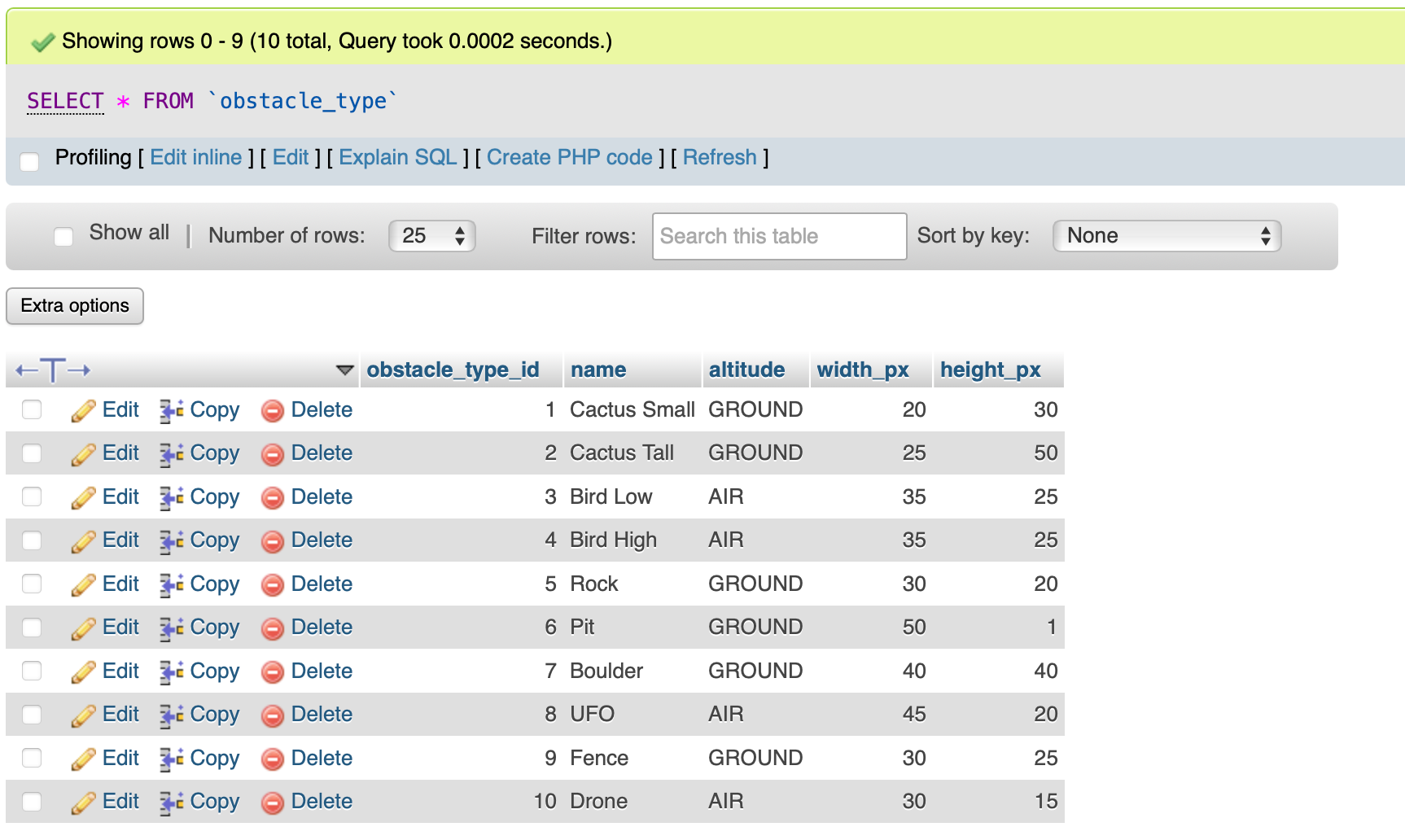
The session table contains 10 gameplay sessions with realistic score and distance data from Season 3. Each session includes player ID, season ID, skin used, timestamps, scores, distances, top speeds, and crash types. The duration\_ms field is automatically calculated.

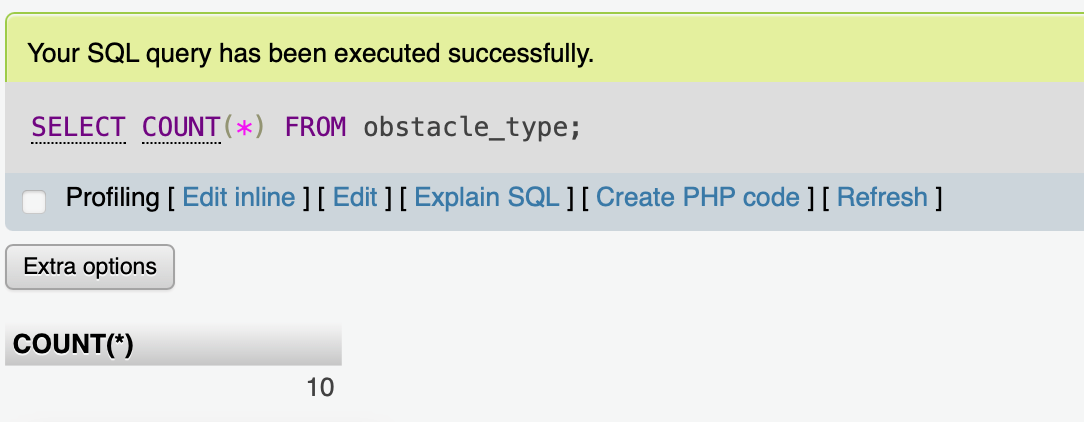
****

****

**2.6 ObstacleType Table**

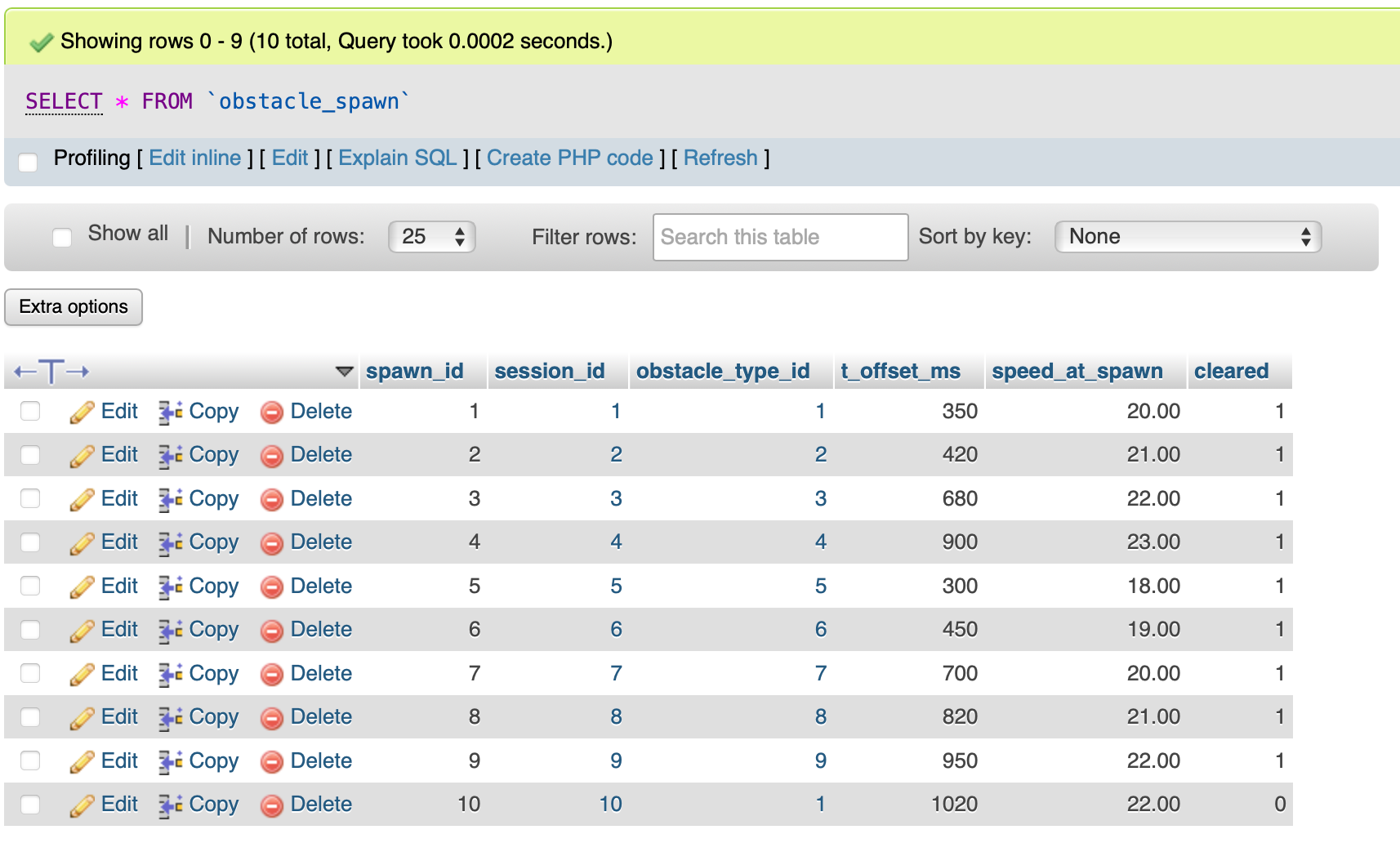
The obstacle\_type table defines 10 different obstacles with varying sizes and altitudes. Some are ground obstacles like cacti and rocks while others are air obstacles like birds and drones. Each has width and height dimensions in pixels.

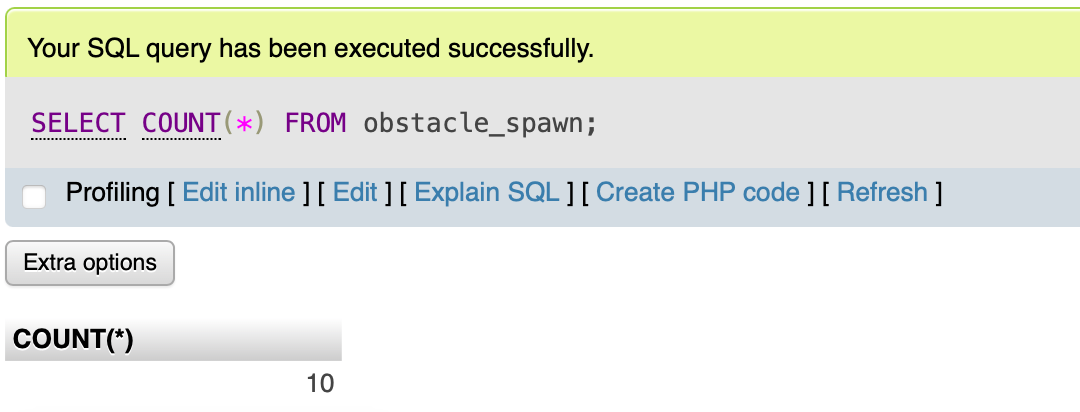
****

****

**2.7 ObstacleSpawn Table**

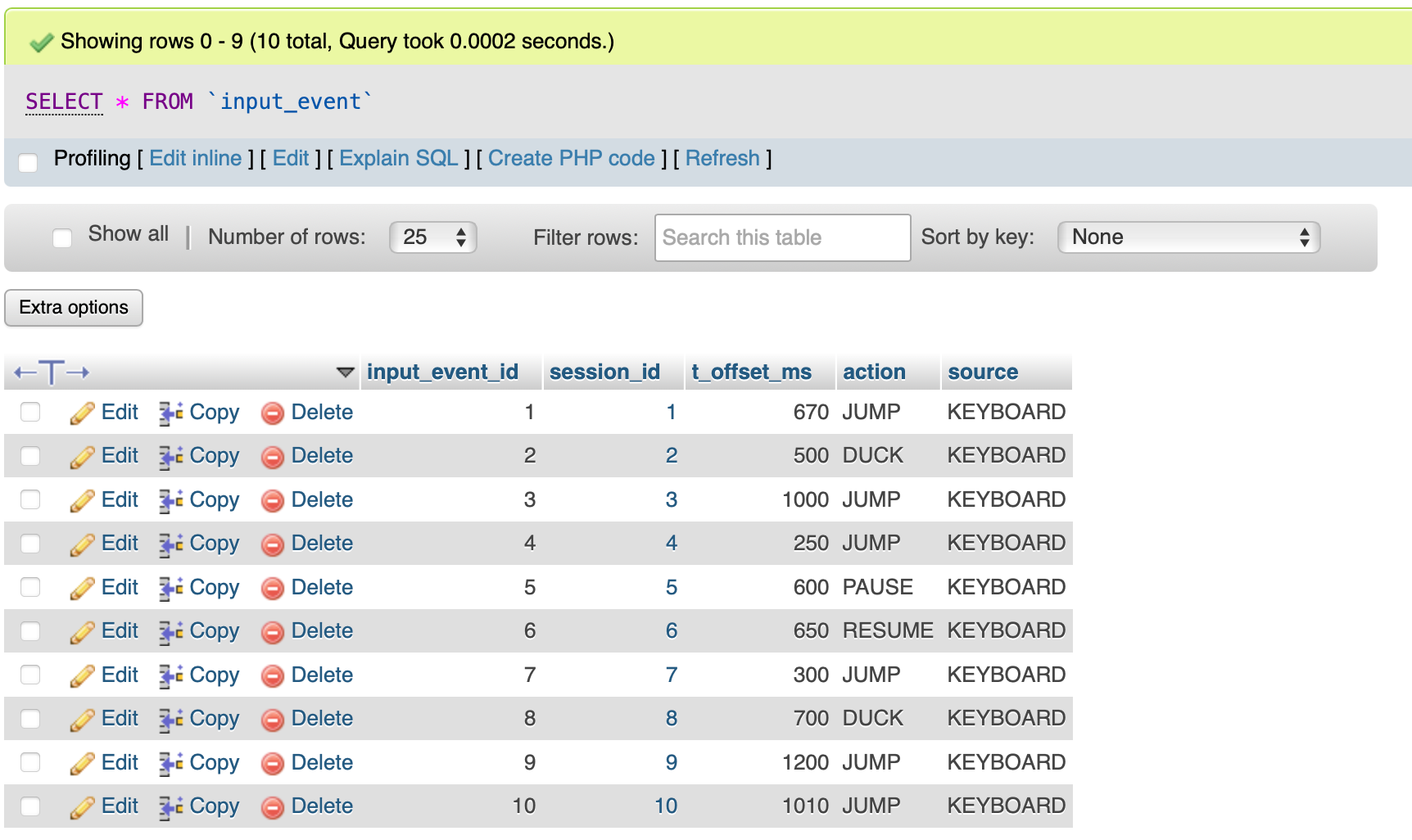
The obstacle\_spawn table contains 10 rows recording when obstacles appeared during gameplay sessions. Each entry includes timing offsets, spawn speeds, and whether the obstacle was cleared. Most obstacles were cleared except one that caused a collision.

****

****

**2.8 InputEvent Table**

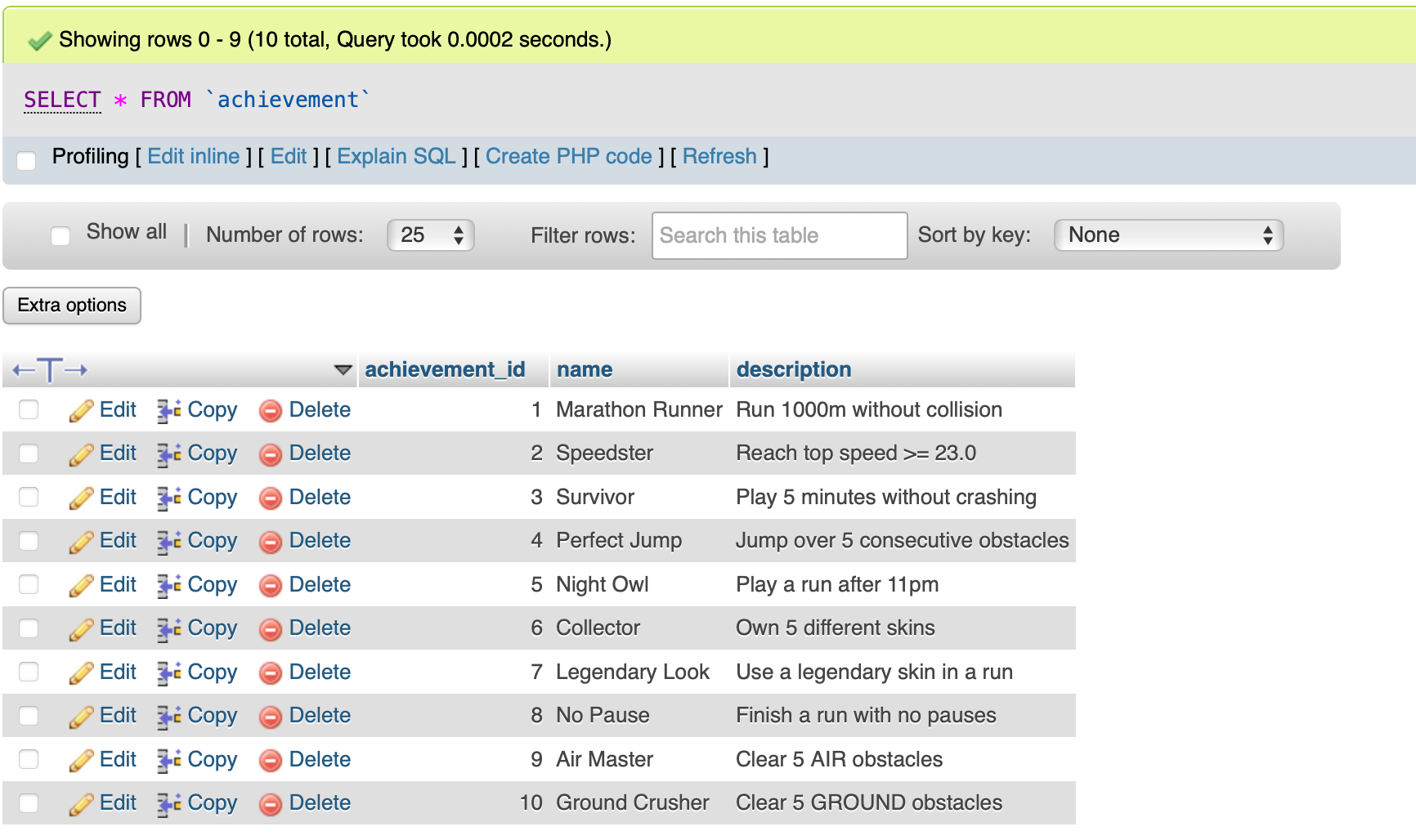
The input\_event table captures 10 player actions with precise timing information. Actions include jumps, ducks, pauses, and resumes with their time offsets and input sources like keyboard or touch.

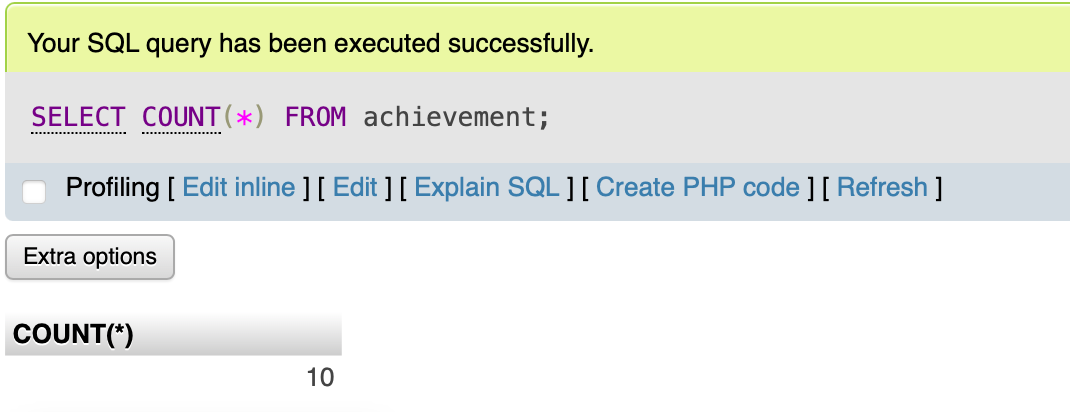
****

****

**2.9 Achievement Table**

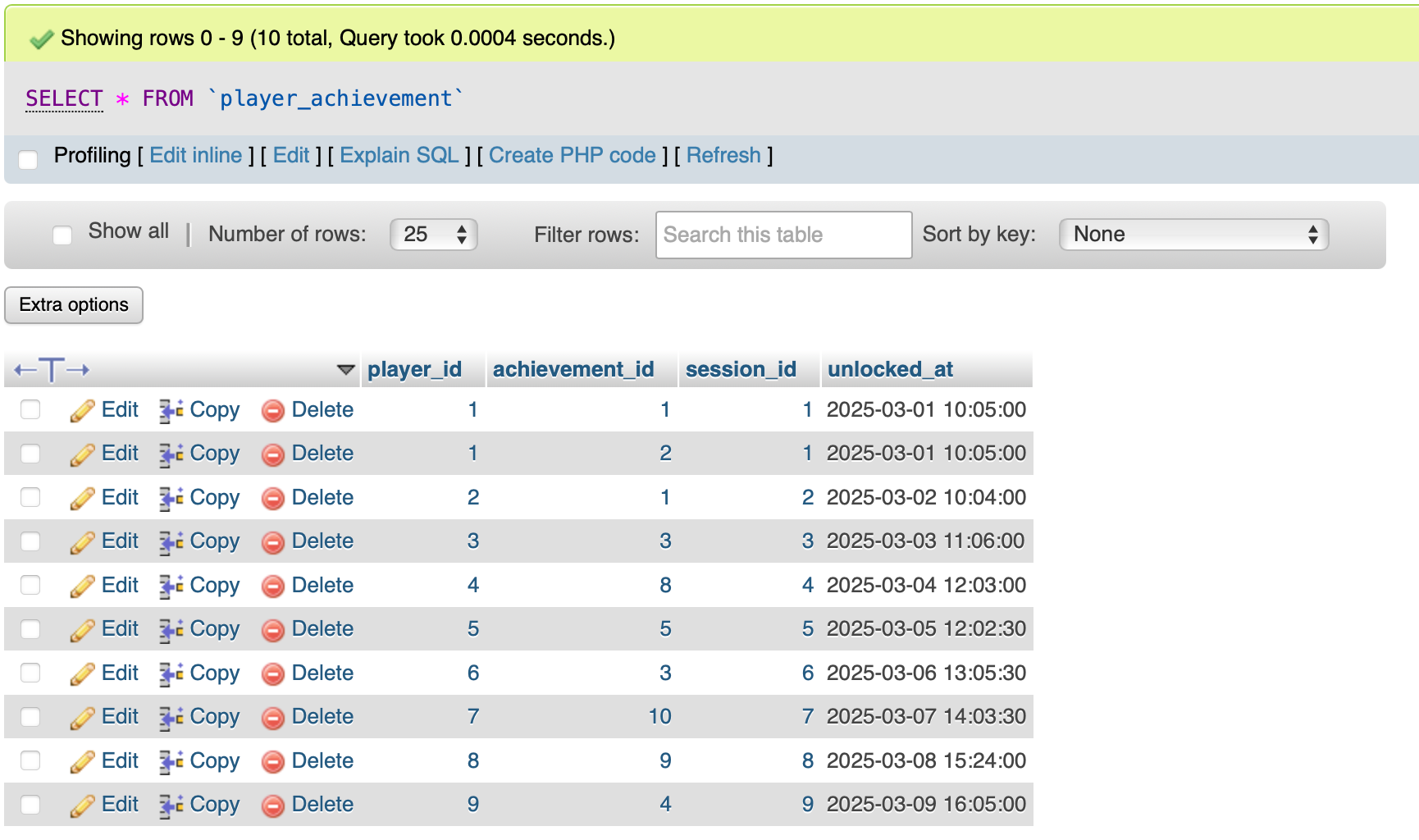
The achievement table defines 10 unlockable achievements with varying difficulty levels. These include distance milestones, speed challenges, collection goals, and gameplay requirements.

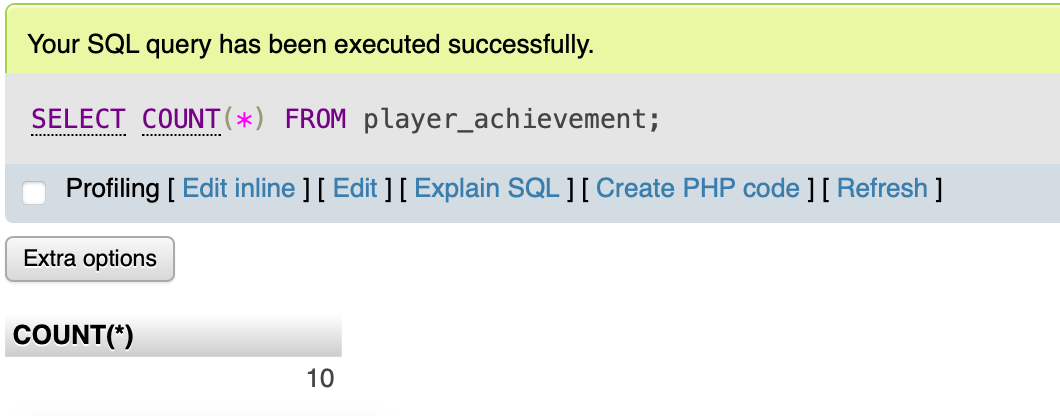
****

****

**2.10 PlayerAchievement Table**

The player\_achievement table contains 10 rows showing which achievements players have unlocked. Each entry links to the specific player, achievement, session where it was earned, and timestamp.

****

****

**3. SQL Commands Used in Our Application**

This section documents all SQL commands that our application uses to interact with the database.

**3.1 Non-Advanced SQL Commands**

**Create Commands**

All 10 tables were created using CREATE TABLE statements with appropriate data types, primary keys, foreign keys, and constraints. The complete statements are in COMMANDS.sql.

**Insert Commands**

We populated each table with realistic sample data. Examples include:

INSERT INTO player (username, email, password, account\_type, created\_at) VALUES ('slimeMaster','sm@demo.com','hashA','REGISTERED','2025-01-05 10:00:00');

INSERT INTO season (name, start\_date, end\_date, is\_active) VALUES ('Season 1','2025-01-01','2025-01-31',0);

INSERT INTO skin (name, rarity, is\_default) VALUES ('Classic', 'common', 1);

All INSERT statements are included in COMMANDS.sql.

**Select Commands**

Our application uses various SELECT queries:

Get all registered players:  
SELECT \* FROM player WHERE account\_type = 'REGISTERED';

Get leaderboard for a specific season:  
SELECT p.username, s.score, s.distance\_m, s.top\_speed FROM session s JOIN player p ON s.player\_id = p.player\_id WHERE s.season\_id = 3 ORDER BY s.score DESC LIMIT 10;

Get achievements for a player:  
SELECT a.name, a.description, pa.unlocked\_at FROM player\_achievement pa JOIN achievement a ON pa.achievement\_id = a.achievement\_id WHERE pa.player\_id = 1;

Get skins owned by a player:  
SELECT s.name, s.rarity, ps.acquired\_at, ps.source FROM player\_skin ps JOIN skin s ON ps.skin\_id = s.skin\_id WHERE ps.player\_id = 1;

Get active season:  
SELECT \* FROM season WHERE is\_active = 1;

Get session history:  
SELECT session\_id, score, distance\_m, started\_at, crash\_type FROM session WHERE player\_id = 1 ORDER BY started\_at DESC;

**Update Commands**

Update player last login:  
UPDATE player SET last\_login\_at = NOW() WHERE player\_id = 1;

Update session score:  
UPDATE session

SET ended\_at = NOW(),

score = 5000,

distance\_m = 1000,

top\_speed = 25.5,

crash\_type = 'COLLIDE',

obstacle\_type\_id = 1 -- keep crash info consistent

WHERE session\_id = 1;

Activate new season:  
UPDATE season SET is\_active = 0 WHERE is\_active = 1;  
UPDATE season SET is\_active = 1 WHERE season\_id = 10;

**Delete Commands**

Delete guest player:  
DELETE FROM player WHERE player\_id = 10 AND account\_type = 'GUEST';

Delete old sessions:  
DELETE FROM session s

WHERE s.season\_id < 3

AND NOT EXISTS (

SELECT 1

FROM player\_achievement pa

WHERE pa.session\_id = s.session\_id

);

**3.2 Advanced PL/SQL Commands**

**Advanced Feature 1: STORED PROCEDURE**

Purpose: Validates that crash data in a session is consistent with the obstacle spawn records.

Parameters:

p\_session\_id (INT): The session ID to validate

Returns: None (raises an error if validation fails)

Business Logic: This procedure enforces complex validation rules for session crashes. If a session crashed due to collision (COLLIDE), there must be exactly one uncleared obstacle spawn and the obstacle\_type\_id must match between the session and the uncleared spawn. For non-collision crashes (QUIT or TIMEOUT), all obstacle spawns must be marked as cleared. This ensures data consistency between the session and obstacle\_spawn tables.

-- USE dino\_runner;

DELIMITER //

CREATE PROCEDURE sp\_validate\_session\_crash(IN p\_session\_id INT)

BEGIN

DECLARE v\_crash VARCHAR(10); DECLARE v\_obst INT; DECLARE v\_uncleared INT; DECLARE v\_match INT;

SELECT crash\_type, obstacle\_type\_id INTO v\_crash, v\_obst FROM session WHERE session\_id = p\_session\_id;

IF v\_crash IS NULL THEN SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Session not found'; END IF;

IF v\_crash = 'COLLIDE' THEN

SELECT SUM(CASE WHEN cleared = 0 THEN 1 ELSE 0 END) INTO v\_uncleared FROM obstacle\_spawn WHERE session\_id = p\_session\_id;

IF v\_uncleared <> 1 THEN SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'COLLIDE must have exactly one uncleared spawn'; END IF;

SELECT COUNT(\*) INTO v\_match FROM obstacle\_spawn WHERE session\_id = p\_session\_id AND cleared = 0 AND obstacle\_type\_id = v\_obst;

IF v\_match <> 1 THEN SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Crash obstacle\_type\_id mismatch'; END IF;

ELSE

SELECT SUM(CASE WHEN cleared = 0 THEN 1 ELSE 0 END) INTO v\_uncleared FROM obstacle\_spawn WHERE session\_id = p\_session\_id;

IF v\_uncleared <> 0 THEN SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Non-collide session cannot contain uncleared spawns'; END IF;

END IF;

END//

DELIMITER ;

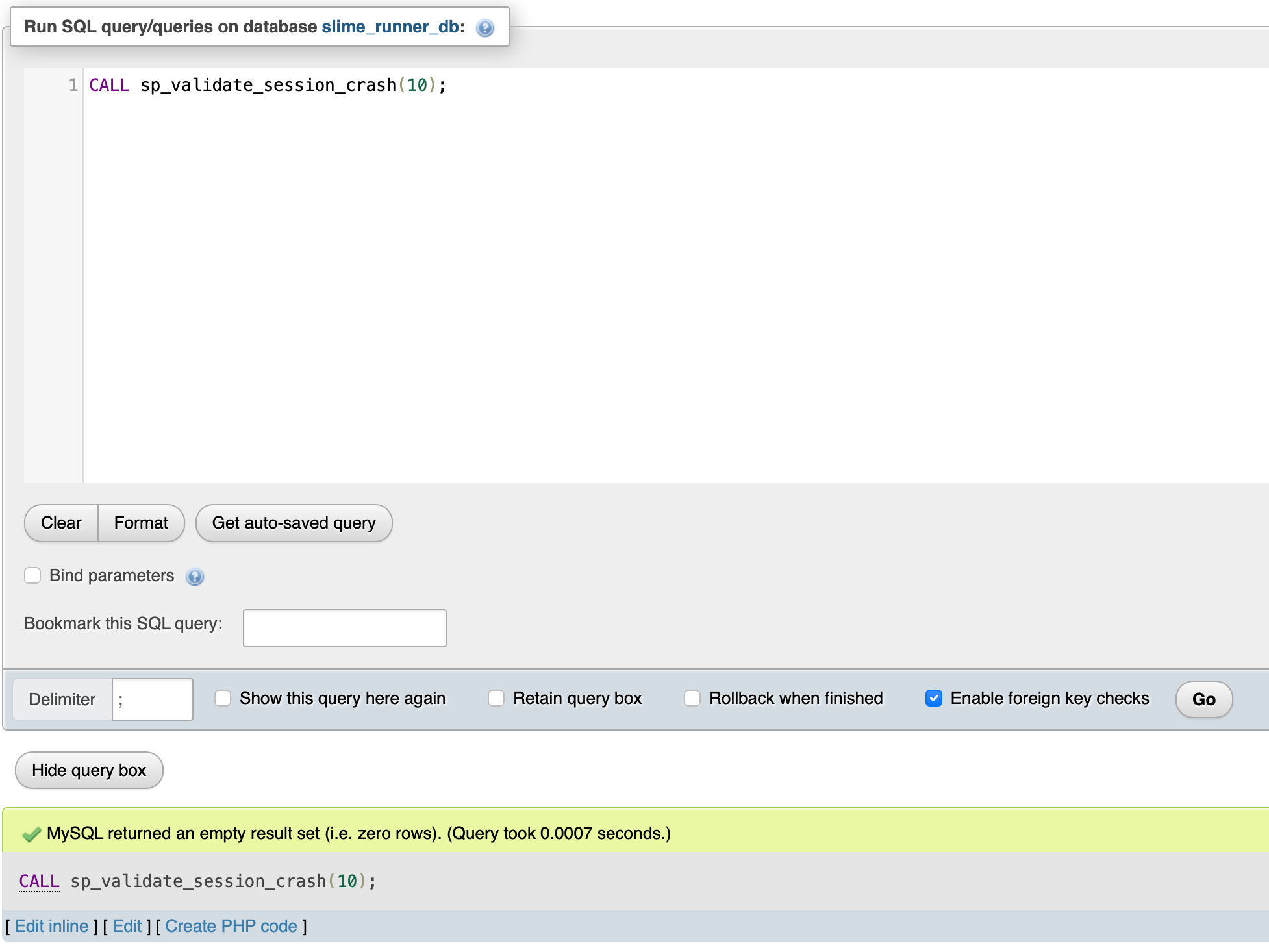
Procedure Testing:

Test 1: CALL sp\_validate\_session\_crash(10);

Result: Success because session 10 has crash\_type COLLIDE with exactly one uncleared obstacle spawn and matching obstacle\_type\_id

Test 2: CALL sp\_validate\_session\_crash(1);

Result: Success because session 1 has crash\_type QUIT with all obstacle spawns cleared



**Advanced Feature 2: FUNCTION**

**fn\_player\_owns\_skin Function**

Purpose: Checks whether a player owns a specific skin at a given point in time.

Parameters:  
p\_player\_id (INT): The player ID to check  
p\_skin\_id (INT): The skin ID to verify  
p\_at (DATETIME): The timestamp to check ownership

Returns: BOOLEAN (TRUE if player owns the skin, FALSE otherwise)

The function first checks if the skin is the default skin and returns TRUE since all players own it. Otherwise it queries the player\_skin table to verify if the player acquired that skin before the specified time.

Code:  
DELIMITER //  
CREATE FUNCTION fn\_player\_owns\_skin(p\_player\_id INT, p\_skin\_id INT, p\_at DATETIME)  
RETURNS BOOLEAN  
DETERMINISTIC  
BEGIN  
DECLARE has\_default BOOLEAN;  
DECLARE owned\_count INT;

SELECT is\_default INTO has\_default FROM skin WHERE skin\_id = p\_skin\_id;  
IF has\_default = 1 THEN  
RETURN TRUE;  
END IF;

SELECT COUNT(\*) INTO owned\_count FROM player\_skin WHERE player\_id = p\_player\_id AND skin\_id = p\_skin\_id AND acquired\_at <= p\_at;

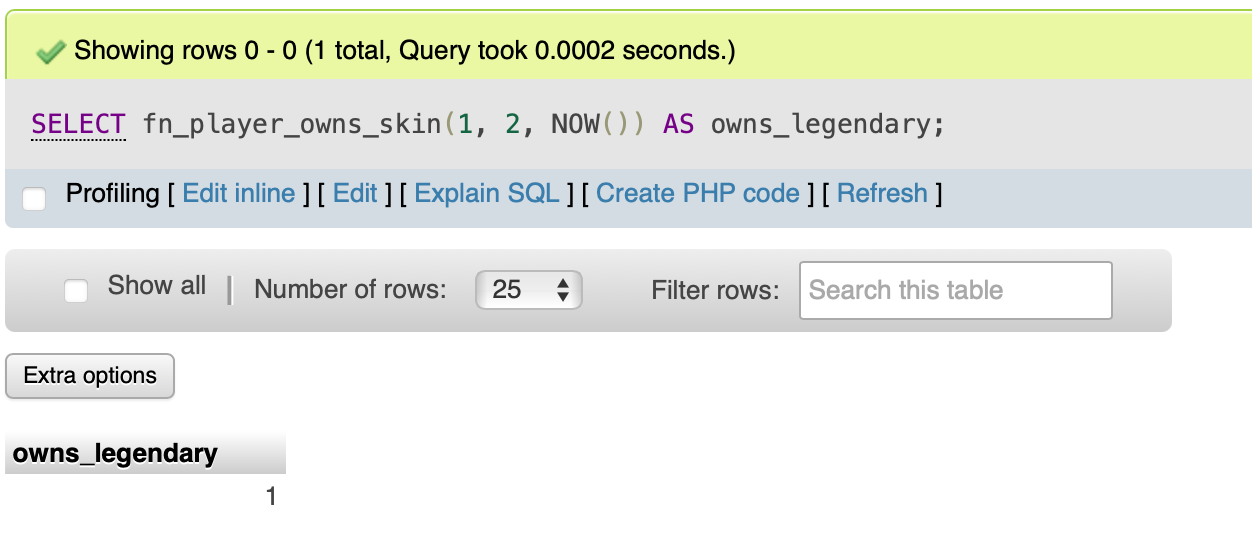
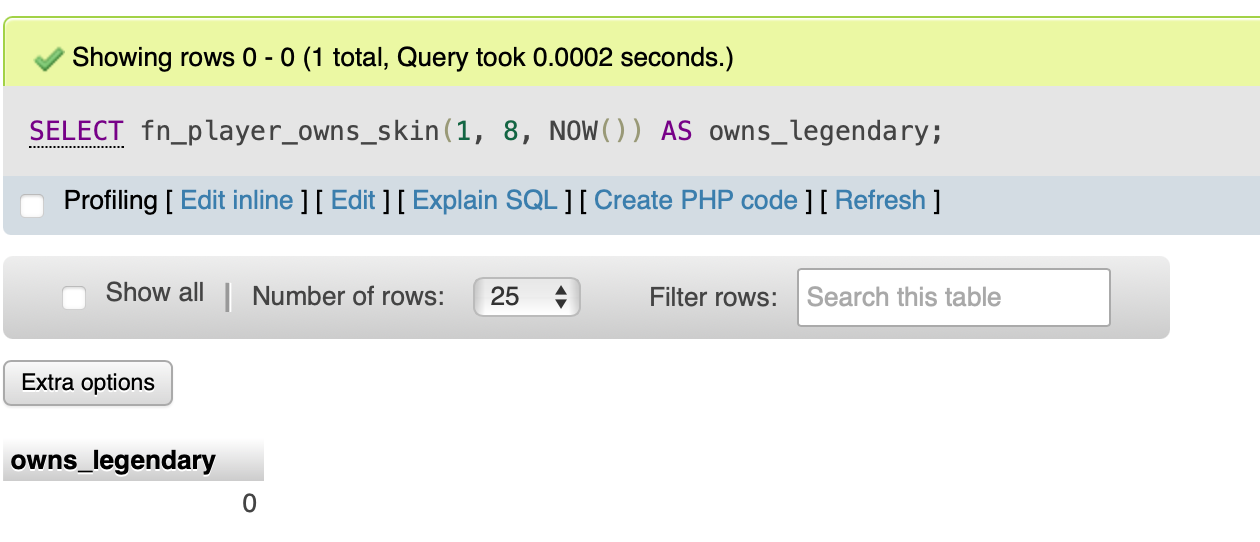
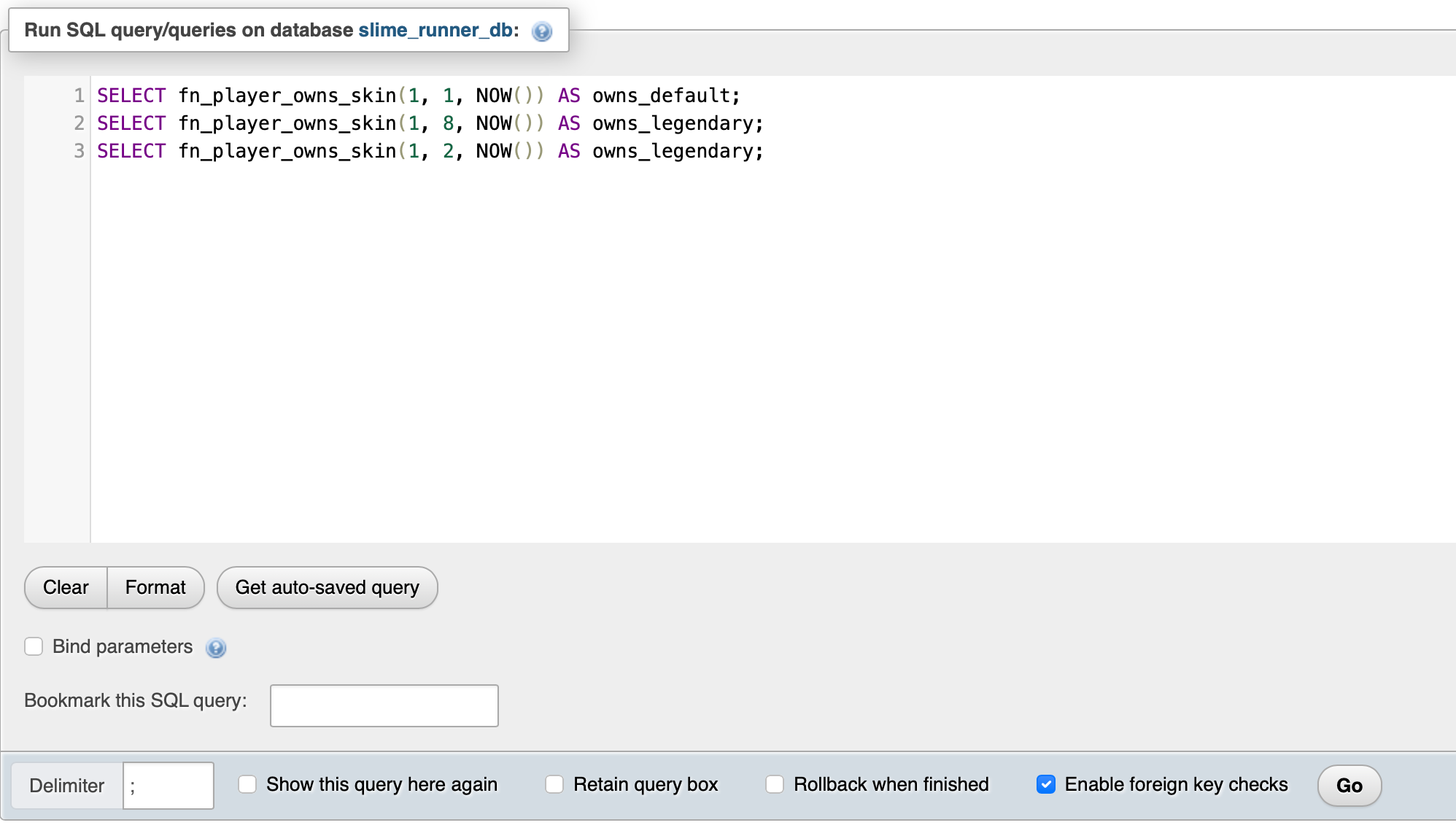
RETURN owned\_count > 0;  
END//  
DELIMITER ;

**Function Testing:**

Test 1: SELECT fn\_player\_owns\_skin(1, 1, NOW()) AS owns\_default;  
Result: 1 (TRUE) because player 1 owns the default skin

Test 2: SELECT fn\_player\_owns\_skin(1, 8, NOW()) AS owns\_legendary;  
Result: 0 (FALSE) because player 1 does not own the legendary skin

Test 3: SELECT fn\_player\_owns\_skin(1, 2, NOW()) AS owns\_rare;  
Result: 1 (TRUE) because player 1 acquired skin 2 on 2025-02-01

****

All tests passed successfully confirming the function works correctly.

**4. Conclusion**

We have successfully set up our MySQL database with all 10 required tables and populated them with realistic data. All tables contain at least 10 rows of practical data demonstrating our game functionality. We have implemented comprehensive SQL commands including CREATE, INSERT, SELECT, UPDATE, and DELETE operations. We implemented two different advanced SQL features: a stored procedure and a function (we also use CHECK constraints for integrity).