SQL Code for Stage 4

Requirements for SQL part of stage 4:

1. Application Requirements:
    - Develop a complete application with CRUD (Create, Read, Update, Delete) operations as outlined in your proposal.
    - Implement keyword search functionality, enabling user input and interface results display.
2. Advanced Database Feature:
    - Your database should have stored procedure(s), transaction(s), constraints, and trigger(s), accessible via the frontend.
3. Detailed Feature Requirements:
    - Transaction: Functional, with correct isolation level, at least two advanced queries, and application utility.
    - Stored Procedure: Functional, with at least two advanced queries, cursors (optional), control structures, and application utility.
    - Trigger: Functioning triggers, involving an event, condition (IF statement), and action (Update, Insert, Delete), enhancing the application.
    - Constraints: This requirement can be simply satisfied by defining the appropriate primary keys, and foreign keys. But we strongly encourage the team to use other types of constraints, such as attribute-level, tuple-level, and assertions.
    - Implement features in SQL (not with Object Relational Mapping, ORM. More details in the next section), ensuring relevance to the application.

Rubric for Checkpoint 2:

Presentation: Timeliness and completion within scheduled time (+1%).

- CRUD Operations: The application functionality includes reads, insertions, updates, and deletions from the database(6% total).
- Keyword Search: Interface, code, and query execution (5% total).
- Advanced Program: Transaction(s), trigger(s), constraints, and stored procedure(s) are implemented and integrated into the application (10% total), with detailed criteria for transactions, stored procedures, and triggers.
- Extra Credit: GCP hosting and creative component relevance (up to +3% total).

Code for SQL parts of Stage 4

SQL Code for Registering a new user:

Pierre
INSERT INTO users
VALUES(<new_user_id>, 1, <pwd>, <email>, <user_name>);

SQL Code for Letting a User change their password, username, or email:

Pierre
UPDATE users
SET pwd = <new_password>
WHERE user_id = <user_making_change>;

Pierre
UPDATE users
SET is_active = 0
WHERE user_id = <user_deactivating_account>;

SQL Code for letting a User make a new team:

Pierre
INSERT INTO user_teams
VALUES(<user_id>, <new_user_team_id>, 1, <team_name>);

SQL Code for letting a User change an existing team (name of team, Pokemon on it, moves for each Pokemon)

In user_teams, user can only change the team name or make it inactive:

Pierre
UPDATE user_teams
SET team_name = <new_team_name>
WHERE user_team_id = <team_being_renamed>;

Pierre
UPDATE user_teams
SET is_active = 0
WHERE user_team_id = <user_deactivating_team>;

In user_poke_team_members, make updates for Pokemon and for moves

Pierre
UPDATE user_poke_team_members
SET pokedex_id = <new_pokemon_on_team>,
move_1_id = null,
move_1_current_pp = null,
move_2_id = null,
move_2_current_pp = null,
move_3_id = null,
move_3_current_pp = null,
move_4_id = null,
move_4_current_pp = null
WHERE user_team_id = <team_being_changed>
AND user_team_member_id = <id_of_member_being_changed>;

Pierre
UPDATE user_poke_team_members
SET move_1_id = <new_move_id>, move_1_current_pp =
<new_move_pp_from_moves_table>
WHERE user_team_id = <team_being_changed>
AND user_team_member_id = <id_of_member_being_changed>;

SQL Code for More lookups a user can do on the Pokedex

SQL Code for battles

SQL Code for when a user wants to look at records of their battles, badges, teams, etc.

<u>IDEAS FOR TRIGGERS, STORED PROCEDURES, TRANSACTIONS, CONSTRAINTS:</u>

Transaction: for registering new user, to make sure no write-write conflict, etc.

Trigger: check if a pokemon's team has been defeated after a Pokemon's HP is updated during battle (Tanjie)

Trigger: registering new user, check if there is already an account with that email

```
Mengmeng
1. prevent from creating duplicate team_id
```

```
CREATE TRIGGER check_team_id BEFORE INSERT ON user_teams FOR EACH ROW
BEGIN
   -- Deactivate all other teams of this user before insert
   UPDATE user_teams
   SET is_active = 0
   WHERE user_id = NEW.user_id AND is_active = 1;
END;

2. Do not select one pokemon for more than 3 times in your team, otherwise the
system
-- will randomly pick one for the user
CREATE TRIGGER check_duplicate_pokemon BEFORE INSERT user_poke_team_members FOR
EACH ROW
BEGIN
   @count_poke = SELECT COUNT(pokedex_id) FROM user_poke_team_members
                 GROUP BY user_team_id
                 Having NEW.pokedex_id = pokedex_id;
   IF @count_poke >= 3
       SET NEW.pokedex_id = pokedex_id + FLOOR(1 + RAND() * 55);
   END IF;
END;

3. If the gaming time more than 12 hours, automatically stop the game and set
the win_loss_outcome to 0
CREATE TRIGGER stop_game BEFORE INSERT ON battles FOR EACH ROW
BEGIN
   IF TIMESTAMPDIFF(SECOND, NEW.start_time, NEW.end_time) > 12*60*60 THEN
       SET NEW.win_loss_outcome = 0;
       SET NEW.end_time = DATE_ADD(NEW.start_time, INTERVAL 12 HOUR);
   END IF;
END;

4. If the user has received a badge from the gym, he cannot get it again
CREATE TRIGGER badge_check BEFORE INSERT ON battles FOR EACH ROW
BEGIN
   DECLARE badge_count INT DEFAULT 0;

   SELECT COUNT(*) INTO badge_count
   FROM battles AS b
   JOIN user_teams AS ut ON b.user_team_id = ut.user_team_id
   WHERE ut.user_id = (
       SELECT user_id FROM user_teams WHERE user_team_id = NEW.user_team_id
   )
   AND b.gym_id = NEW.gym_id
   AND b.win_loss_outcome = 1;
```

```
    IF badge_count > 0 THEN
        SET NEW.badge_title = NULL;
    END IF;
END;
```

Stored Procedure?: when user changes a Pokemon on their team, all moves for that team member are set to null until user chooses new moves that the new Pokemon can have (from the pokemon_moves table)

```
Mengmeng
-- calculating pp for each move
CREATE PROCEDURE UpdatePP(
    IN move_pp_in INT,
    OUT move_pp_out INT
)
BEGIN
    IF move_pp_in > 0 THEN
        SET move_pp_out = move_pp_in - 1;
    ELSE
        SET move_pp_out = 0;
    END IF;
END;

--calculating hp per battle run
CREATE PROCEDURE UpdateHP(
    IN in_user_team_id INT,
    IN in_user_team_member_id INT,
    IN damage INT,
    OUT updated_hp INT
)
BEGIN
    DECLARE hp_var INT;
    DECLARE defence_var INT;
    DECLARE speed_var INT;

    -- Get current HP
    SELECT current_hp INTO hp_var
    FROM user_poke_team_members
    WHERE user_team_id = in_user_team_id AND user_team_member_id =
in_user_team_member_id;
```

```
    -- Get defence and speed from joined pokedex_entries
    SELECT pe.defence, pe.speed
    INTO defence_var, speed_var
    FROM user_poke_team_members u
    JOIN pokedex_entries pe USING(pokedex_id)
    WHERE u.user_team_id = in_user_team_id AND u.user_team_member_id =
in_user_team_member_id;

    -- Calculate new HP
    SET updated_hp = GREATEST(hp_var + defence_var * speed_var - damage, 0);
END;
```

Table schemas/schemata from Stage 3 (for reference):

```
CREATE TABLE users(
user_id INT PRIMARY KEY,
is_active BOOLEAN NOT NULL,
pwd VARCHAR(255) NOT NULL,
email VARCHAR(100) NOT NULL UNIQUE,
user_name VARCHAR(30) NOT NULL
);
```

```
CREATE TABLE user_teams(
user_id INT NOT NULL,
user_team_id INT PRIMARY KEY,
is_active BOOLEAN NOT NULL,
team_name VARCHAR(30) NOT NULL,
FOREIGN KEY(user_id) REFERENCES users(user_id) ON DELETE CASCADE
);
```

```
CREATE TABLE user_poke_team_members(
user_team_id INT,
user_team_member_id INT NOT NULL CHECK(user_team_member_id >= 1 AND
user_team_member_id <= 6),
pokedex_id INT,
current_hp REAL,
```

```
move_1_id INT,
move_1_current_pp INT,
move_2_id INT,
move_2_current_pp INT,
move_3_id INT,
move_3_current_pp INT,
move_4_id INT,
move_4_current_pp INT,
PRIMARY KEY(user_team_id, user_team_member_id),
FOREIGN KEY(user_team_id) REFERENCES user_teams(user_team_id) ON DELETE
CASCADE,
FOREIGN KEY(pokedex_id) REFERENCES pokedex_entries(pokedex_id) ON DELETE
CASCADE
);
```

```
CREATE TABLE pokemon_moves(
pokedex_id INT,
move_id INT,
PRIMARY KEY (pokedex_id, move_id),
FOREIGN KEY (pokedex_id) REFERENCES pokedex_entries(pokedex_id) ON DELETE
CASCADE,
FOREIGN KEY (move_id) REFERENCES moves(move_id) ON DELETE CASCADE
);
```

```
CREATE TABLE moves(
move_id INT PRIMARY KEY,
move_name VARCHAR(30),
move_type VARCHAR(30),
category VARCHAR(30),
move_power INT,
accuracy FLOAT,
pp INT
);
```

```
CREATE TABLE type_matchups(
matchup_id INT,
attacking_type VARCHAR(30),
defending_type VARCHAR(30),
multiplier REAL,
```

```
primary key(attacking_type, defending_type)
);
```

```
CREATE TABLE pokedex_entries(
pokedex_id INT PRIMARY KEY,
name VARCHAR(30),
hp INT,
attack INT,
defense INT,
sp_attack INT,
sp_defense INT,
speed INT,
pType_1 VARCHAR(30),
pType_2 VARCHAR(30),
image_url VARCHAR(255)
);
```

```
CREATE TABLE gym_leader_team_members(
gym_id INT,
gym_team_member_id INT,
pokedex_id INT,
current_hp REAL,
move_1_id INT,
move_1_current_pp INT,
move_2_id INT,
move_2_current_pp INT,
move_3_id INT,
move_3_current_pp INT,
move_4_id INT,
move_4_current_pp INT,
PRIMARY KEY (gym_id, gym_team_member_id),
FOREIGN KEY(gym_id) REFERENCES gym_leaders(gym_id) ON DELETE CASCADE,
FOREIGN KEY(pokedex_id) REFERENCES pokedex_entries(pokedex_id) ON DELETE
CASCADE);
```

```
CREATE TABLE gym_leaders(
gym_id INT PRIMARY KEY,
gym_leader VARCHAR(30),
```

```
gym_name VARCHAR(30),
gym_theme_img VARCHAR(255),
badge_title VARCHAR(30),
badge_image VARCHAR(255)
);
```

```
CREATE TABLE battles(
battle_id INT PRIMARY KEY,
user_team_id INT,
gym_id INT,
date_time DATETIME,
end_time DATETIME,
win_loss_outcome BOOLEAN,
FOREIGN KEY(user_team_id) REFERENCES user_teams(user_team_id) ON DELETE
CASCADE,
FOREIGN KEY(gym_id) REFERENCES gym_leaders(gym_id) ON DELETE CASCADE);
```

BATTLE FLOW RAW SQL (TANJIE)

TRIGGERS

Trigger: check if a gym leader has been defeated during battle

```
CREATE TRIGGER trg_check_gym_defeat
AFTER UPDATE ON gym_leader_team_members
FOR EACH ROW
BEGIN
    DECLARE remaining_pokemon INT;
    DECLARE latest_battle_id INT;
    IF NEW.current_hp <= 0 AND OLD.current_hp > 0 THEN
        SELECT COUNT(*)
        INTO remaining_pokemon
        FROM gym_leader_team_members
        WHERE gym_id = NEW.gym_id AND current_hp > 0;
        IF remaining_pokemon = 0 THEN
            SELECT battle_id
            INTO latest_battle_id
            FROM battles
            WHERE gym_id = NEW.gym_id AND win_loss_outcome IS NULL
            ORDER BY battle_id DESC
            LIMIT 1;
            UPDATE battles
            SET win_loss_outcome = 0, end_time = NOW()
            WHERE battle_id = latest_battle_id;
```

```
        END IF;
    END IF;
END;
```

Trigger: Check if a user has been defeated in battle

```
CREATE TRIGGER trg_check_user_defeat
AFTER UPDATE ON user_poke_team_members
FOR EACH ROW
BEGIN
    DECLARE remaining_pokemon INT;
    DECLARE latest_battle_id INT;
    IF NEW.current_hp <= 0 AND OLD.current_hp > 0 THEN
        SELECT COUNT(*)
        INTO remaining_pokemon
        FROM user_poke_team_members
        WHERE user_team_id = NEW.user_team_id AND current_hp > 0;
        IF remaining_pokemon = 0 THEN
            SELECT battle_id
            INTO latest_battle_id
            FROM battles
            WHERE user_team_id = NEW.user_team_id AND win_loss_outcome IS NULL
            ORDER BY battle_id DESC
            LIMIT 1;
            UPDATE battles
            SET win_loss_outcome = 0, end_time = NOW()
            WHERE battle_id = latest_battle_id;
        END IF;
    END IF;
END;
```

Trigger: **Check if a gym leader has been defeated in battle:**

```
| trg_check_gym_defeat | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION | CREATE DEFINER=`tanjie`@`%` TRIGGER `trg_check_gym_defeat` AFTER UPDATE ON `gym_leader_team_members` FOR EACH ROW BEGIN

    IF new.current_hp <= 0 AND old.current_hp > 0 THEN


        SET @remaining_pokemon = (
            SELECT COUNT(*)
            FROM gym_leader_team_members
            WHERE gym_id = new.gym_id AND current_hp > 0
        );


        IF @remaining_pokemon = 0 THEN
            UPDATE battles
            SET win_loss_outcome = TRUE, end_time = NOW()

            WHERE gym_id = new.gym_id AND win_loss_outcome IS NULL
            ORDER BY battle_id DESC
            LIMIT 1;
        END IF;
    END IF;
END | utf8mb4              | utf8mb4_0900_ai_ci    | utf8mb4_0900_ai_ci | 2025-08-06 02:42:49.95 |
```

QUERIES

QUERY: # Find the user_team_id of the user's active team before battle

```sql
SELECT
 user_team_id
FROM
 user_teams
WHERE
 user_id = 1
 AND is_active = TRUE;
```

QUERY: # Find the pokemon on the active user_team during battle

```sql
SELECT
 utm.user_team_member_id,
 p.name,
 utm.current_hp,
 p.hp AS max_hp,
 p.image_url
FROM
 user_poke_team_members utm
JOIN
 pokedex_entries p
ON
 utm.pokedex_id = p.pokedex_id
```

```
WHERE
 utm.user_team_id = %s
ORDER BY
 utm.user_team_member_id;
```

STORED PROCEDURE: get the initial state of the battle start

```
CREATE PROCEDURE get_battle_state(
  IN userTeamId INT,
  IN gymId INT
)
BEGIN
  # Create the battle record
  INSERT INTO battles (user_team_id, gym_id, start_time)
  VALUES (userTeamId, gymId, NOW());

  # Heal the user's team to full HP
  UPDATE user_poke_team_members utm
  JOIN pokedex_entries p ON utm.pokedex_id = p.pokedex_id
  SET utm.current_hp = p.hp
  WHERE utm.user_team_id = userTeamId;

  # Heal the gym leader's team to full HP
  UPDATE gym_leader_team_members gtm
  JOIN pokedex_entries p ON gtm.pokedex_id = p.pokedex_id
  SET gtm.current_hp = p.hp
  WHERE gtm.gym_id = gymId;


  # Get the initial battle state for USER
  SELECT
      "USER" AS party_type,
    utm.user_team_id as team_id,
    utm.user_team_member_id as member_id,
```

```
    p.name AS pokemon_name,
    utm.current_hp,
    p.hp AS max_hp,
    p.image_url,
    m1.move_name AS move_1_name,
    utm.move_1_current_pp,
    m1.pp AS move_1_max_pp,
    m2.move_name AS move_2_name,
    utm.move_2_current_pp,
    m2.pp AS move_2_max_pp,
    m3.move_name AS move_3_name,
    utm.move_3_current_pp,
    m3.pp AS move_3_max_pp,
    m4.move_name AS move_4_name,
    utm.move_4_current_pp,
    m4.pp AS move_4_max_pp
FROM
    user_poke_team_members utm
JOIN
    pokedex_entries p ON utm.pokedex_id = p.pokedex_id
LEFT JOIN
    moves m1 ON utm.move_1_id = m1.move_id
LEFT JOIN
    moves m2 ON utm.move_2_id = m2.move_id
LEFT JOIN
    moves m3 ON utm.move_3_id = m3.move_id
LEFT JOIN
    moves m4 ON utm.move_4_id = m4.move_id
WHERE
    utm.user_team_id = userTeamId AND utm.user_team_member_id = 1

UNION
 # Get the initial battle state for GYM LEADER
SELECT
    "GYM" AS party_type,
    gtm.gym_id as team_id,
```

```
    gtm.gym_team_member_id as member_id,
    p.name AS pokemon_name,
    gtm.current_hp,
    p.hp AS max_hp,
    p.image_url,
    m1.move_name AS move_1_name,
    gtm.move_1_current_pp,
    m1.pp AS move_1_max_pp,
    m2.move_name AS move_2_name,
    gtm.move_2_current_pp,
    m2.pp AS move_2_max_pp,
    m3.move_name AS move_3_name,
    gtm.move_3_current_pp,
    m3.pp AS move_3_max_pp,
    m4.move_name AS move_4_name,
    gtm.move_4_current_pp,
    m4.pp AS move_4_max_pp
FROM
    gym_leader_team_members gtm
JOIN
    pokedex_entries p ON gtm.pokedex_id = p.pokedex_id
LEFT JOIN
    moves m1 ON gtm.move_1_id = m1.move_id
LEFT JOIN
    moves m2 ON gtm.move_2_id = m2.move_id
LEFT JOIN
    moves m3 ON gtm.move_3_id = m3.move_id
LEFT JOIN
    moves m4 ON gtm.move_4_id = m4.move_id
WHERE
    gtm.gym_id = gymId AND gtm.gym_team_member_id = 1;
END;
```

STORED PROCEDURES (ALSO A TRANSACTION)

Stored Procedure: process a battle turn

```
CREATE PROCEDURE process_battle_turn(
```

```
  IN attacker_party_type VARCHAR (4), # 'USER' or 'GYM'
  IN attacker_team_id INT, # user_team_id or gym_id
  IN attacker_member_id INT, # 1-6 of the attacker's party
  IN defender_party_type VARCHAR(4), # 'USER' or 'GYM'
  IN defender_team_id INT, # user_team_id or gym_id
  IN defender_member_id INT, # 1-6 of the defender's party
  IN move_slot_used INT, # which move was used (1-4)
  OUT outcome_msg VARCHAR(255) # An output parameter to send a result message back to
Flask
)
proc: BEGIN
 # Get the move_id and the current_pp of the move that the attacker used
 DECLARE a_move_id INT;
 DECLARE a_current_pp INT;

 # Temp vars to hold all four move slots
 DECLARE a_move1_id INT; DECLARE a_move1_pp INT;
 DECLARE a_move2_id INT; DECLARE a_move2_pp INT;
 DECLARE a_move3_id INT; DECLARE a_move3_pp INT;
 DECLARE a_move4_id INT; DECLARE a_move4_pp INT;

 # Attacker and defender stats vars
 DECLARE a_name VARCHAR(30);
 DECLARE a_attack INT;
 DECLARE a_sp_attack INT;
 DECLARE d_name VARCHAR(30);
 DECLARE d_current_hp REAL;
 DECLARE d_defense INT;
 DECLARE d_sp_defense INT;
 DECLARE d_type1 VARCHAR(30);
 DECLARE d_type2 VARCHAR(30);

 # Move details vars
 DECLARE m_name VARCHAR(30);
 DECLARE m_power INT;
 DECLARE m_category VARCHAR(30);
```

```
 DECLARE m_type VARCHAR(30);

 # Declare type multiplier var and set it to a REAL default value
 DECLARE type_multiplier REAL DEFAULT 1.0;

 # Declare final damage vars
 DECLARE final_damage INT;
 DECLARE effective_attack INT;
 DECLARE effective_defense INT;
 DECLARE new_hp REAL;

 # Explicitly set the isolation level for this transaction
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

 START TRANSACTION;
  IF attacker_party_type = "USER" THEN
   SELECT
     move_1_id, move_1_current_pp,
     move_2_id, move_2_current_pp,
     move_3_id, move_3_current_pp,
     move_4_id, move_4_current_pp
   INTO
     a_move1_id, a_move1_pp,
     a_move2_id, a_move2_pp,
     a_move3_id, a_move3_pp,
     a_move4_id, a_move4_pp
   FROM user_poke_team_members
   WHERE user_team_id = attacker_team_id AND user_team_member_id = attacker_member_id;

 ELSEIF attacker_party_type = "GYM" THEN
   SELECT
     move_1_id, move_1_current_pp,
     move_2_id, move_2_current_pp,
     move_3_id, move_3_current_pp,
     move_4_id, move_4_current_pp
   INTO
```

```
    a_move1_id, a_move1_pp,
    a_move2_id, a_move2_pp,
    a_move3_id, a_move3_pp,
    a_move4_id, a_move4_pp
  FROM gym_leader_team_members
  WHERE gym_id = attacker_team_id AND gym_team_member_id = attacker_member_id;
END IF;

# Choose the correct move based on the slot used
  IF move_slot_used = 1 THEN
    SET a_move_id = a_move1_id;
    SET a_current_pp = a_move1_pp;
ELSEIF move_slot_used = 2 THEN
    SET a_move_id = a_move2_id;
    SET a_current_pp = a_move2_pp;
ELSEIF move_slot_used = 3 THEN
    SET a_move_id = a_move3_id;
    SET a_current_pp = a_move3_pp;
ELSEIF move_slot_used = 4 THEN
    SET a_move_id = a_move4_id;
    SET a_current_pp = a_move4_pp;
END IF;

# Check if move is valid and has pp
IF a_move_id IS NULL OR a_current_pp <= 0 THEN
  SET outcome_msg = "This move cannot be used!";
  ROLLBACK;
  LEAVE proc; # Exit the procedure block
END IF;

# Get the attacker's data
IF attacker_party_type = "USER" THEN
  SELECT p.name, p.attack, p.sp_attack INTO a_name, a_attack, a_sp_attack
  FROM user_poke_team_members utm
  JOIN pokedex_entries p ON utm.pokedex_id = p.pokedex_id
  WHERE utm.user_team_id = attacker_team_id AND utm.user_team_member_id =
```

```
attacker_member_id;
 ELSEIF attacker_party_type = "GYM" THEN
   SELECT p.name, p.attack, p.sp_attack INTO a_name, a_attack, a_sp_attack
   FROM gym_leader_team_members gtm
   JOIN pokedex_entries p ON gtm.pokedex_id = p.pokedex_id
   WHERE gtm.gym_id = attacker_team_id AND gtm.gym_team_member_id =
attacker_member_id;
 END IF;

 # Get the defender's data
 IF defender_party_type = "USER" THEN
   SELECT p.name, utm.current_hp, p.defense, p.sp_defense, p.pType_1, p.pType_2
   INTO d_name, d_current_hp, d_defense, d_sp_defense, d_type1, d_type2
   FROM user_poke_team_members utm
   JOIN pokedex_entries p ON utm.pokedex_id = p.pokedex_id
   WHERE utm.user_team_id = defender_team_id AND utm.user_team_member_id =
defender_member_id;
 ELSEIF defender_party_type = "GYM" THEN
   SELECT p.name, gtm.current_hp, p.defense, p.sp_defense, p.pType_1, p.pType_2
   INTO d_name, d_current_hp, d_defense, d_sp_defense, d_type1, d_type2
   FROM gym_leader_team_members gtm
   JOIN pokedex_entries p ON gtm.pokedex_id = p.pokedex_id
   WHERE gtm.gym_id = defender_team_id AND gtm.gym_team_member_id =
defender_member_id;
END IF;

 # Get the move's data
 SELECT move_name, move_power, category, move_type
 INTO m_name, m_power, m_category, m_type
 FROM moves
 WHERE move_id = a_move_id;

 # Calculate the type effectiveness multilpier for defender
 SELECT type_multiplier * multiplier INTO type_multiplier
 FROM type_matchups
 WHERE attacking_type = m_type AND defending_type = d_type1;
```

```
 # If defender has a second type apply its multiplier
 IF d_type2 IS NOT NULL THEN
   SELECT type_multiplier * multiplier INTO type_multiplier
   FROM type_matchups
   WHERE attacking_type = m_type AND defending_type = d_type2;
 END IF;

# Determine which category of attack and defense stats to use
 IF m_category = "Physical" THEN
   SET effective_attack = a_attack;
   SET effective_defense = d_defense;
 ELSEIF m_category = "Special" THEN # Assume "special" type attack
   SET effective_attack = a_sp_attack;
   SET effective_defense = d_sp_defense;
 END IF;

 # Calculate the final damage using a simplified calculation and round down to the
nearest INT
 SET final_damage = FLOOR((((50 * 2 / 5 + 2) * m_power * effective_attack /
effective_defense) / 50 + 2) * type_multiplier);
 SET new_hp = d_current_hp - final_damage;

 # If the damage causes a negative hp value set it to zero and set the correct outcome
message
 IF new_hp <= 0 THEN
   SET new_hp = 0;
   SET outcome_msg = CONCAT(d_name, " fainted!");
 ELSE
    # Otherwise use the normal damage message
   SET outcome_msg = CONCAT(a_name, " used ", m_name, " and did ", final_damage, "
damage!");
 END IF;

 # Apply damage to the defender and UPDATE the current Pokemon's hp
 IF defender_party_type = "USER" THEN
```

```
  UPDATE user_poke_team_members
  SET current_hp = new_hp
  WHERE user_team_id = defender_team_id AND user_team_member_id = defender_member_id;
 ELSEIF defender_party_type = "GYM" THEN
  UPDATE gym_leader_team_members
  SET current_hp = new_hp
  WHERE gym_id = defender_team_id AND gym_team_member_id = defender_member_id;
 END IF;

 # Decrease the attacker's move pp
 IF attacker_party_type = "USER" THEN
  UPDATE user_poke_team_members
  SET
      move_1_current_pp = CASE WHEN move_slot_used = 1 THEN move_1_current_pp - 1
ELSE move_1_current_pp END,
      move_2_current_pp = CASE WHEN move_slot_used = 2 THEN move_2_current_pp - 1
ELSE move_2_current_pp END,
      move_3_current_pp = CASE WHEN move_slot_used = 3 THEN move_3_current_pp - 1
ELSE move_3_current_pp END,
      move_4_current_pp = CASE WHEN move_slot_used = 4 THEN move_4_current_pp - 1
ELSE move_4_current_pp END
  WHERE user_team_id = attacker_team_id AND user_team_member_id = attacker_member_id;
 ELSEIF attacker_party_type = "GYM" THEN
  UPDATE gym_leader_team_members
  SET
      move_1_current_pp = CASE WHEN move_slot_used = 1 THEN move_1_current_pp - 1
ELSE move_1_current_pp END,
      move_2_current_pp = CASE WHEN move_slot_used = 2 THEN move_2_current_pp - 1
ELSE move_2_current_pp END,
      move_3_current_pp = CASE WHEN move_slot_used = 3 THEN move_3_current_pp - 1
ELSE move_3_current_pp END,
      move_4_current_pp = CASE WHEN move_slot_used = 4 THEN move_4_current_pp - 1
ELSE move_4_current_pp END
  WHERE gym_id = attacker_team_id AND gym_team_member_id = attacker_member_id;
 END IF;
 # Commit if all steps succeed and make changes permenent
```

```
 COMMIT;
END
```

Hannah:

Stored procedure:

```
DELIMITER // CREATE PROCEDURE set_badge_level()
BEGIN
DECLARE exit_loop INT default 0;
DECLARE var_user_id INT;
DECLARE var_badge_count INT;
DECLARE var_badge_level VARCHAR(20);
DECLARE usercur CURSOR FOR ( SELECT DISTINCT user_id FROM users );

DECLARE CONTINUE HANDLER FOR NOT FOUND SET exit_loop = 1;
OPEN usercur;

cloop: LOOP
FETCH usercur INTO var_user_id;
IF exit_loop THEN
LEAVE cloop;
END IF;

SELECT COUNT(DISTINCT B.gym_id)
INTO var_badge_count
FROM battles B JOIN user_teams UT ON B.user_team_id = UT.user_team_id
WHERE UT.user_id = var_user_id AND B.win_loss_outcome = 1;

IF var_badge_count = 0 THEN
SET var_badge_level = 'Novice';
ELSEIF var_badge_count < 4 THEN
SET var_badge_level = 'Intermediate';
ELSE
SET var_badge_level = 'Advanced';
END IF;

UPDATE users
SET badge_level = var_badge_level
```

WHERE user_id = var_user_id;

END LOOP; CLOSE usercur;
END;
//
DELIMITER ;

2. Transaction and more for user login/signup:

```
def login():
    """
    The user has submitted a form, by pressing a submit/login/signup button on
        login.html.
    We will sign the user up for the game, or log them in.
    """

    # User has submitted a form on login.html
    if request.method == 'POST':
        # Setup and get information
        form_type = request.form.get('form_type')
        username = request.form.get('user_id')
        password = request.form.get('pwd')
        email = request.form.get('email')

        print(form_type, username, password, email)

        # Setup to connect to GCP
        db_conn = getconn()

        try:
            # Open "context manager" for sql_cursor (auto-ends)
            with db_conn.cursor() as sql_cursor:

                # Set isolation level - we do not want write-write conflicts
                # There should only be unique usernames
                sql_cursor.execute("SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;")

                # User initiated SIGN UP
                if form_type == 'signup':
                    # Check if user already exists
                    check_user_query = "SELECT * FROM users WHERE user_name = %s"
```

```
                    sql_cursor.execute(check_user_query, (username,))
                    existing_username = sql_cursor.fetchone()

                    if existing_username:
                        return "Username already exists."

                    # Insert new user using the STORED PROCEDURE AddNewUser
                    new_user_query = "INSERT INTO users (user_name, pwd, email,
is_active) VALUES (%s, %s, %s, %s)"
                    sql_cursor.execute(new_user_query, (username, password, email, 1))
                    user_id = sql_cursor.lastrowid
                    db_conn.commit()

                    session['username'] = username
                    session['user_id'] = user_id
                    session['email'] = email

                    print("Signup successful! Please log in.")
                    return redirect(url_for('home.load_homepage'))


                # User initiated LOGIN
                elif form_type == 'login':
                    # Check if username already exists
                    check_user_query = "SELECT * FROM users WHERE user_name LIKE %s AND
pwd = %s"
                    sql_cursor.execute(check_user_query, (f"%{username}%", password))
                    existing_user = sql_cursor.fetchone()
                    print(f"existing_user results = {existing_user}")
                    print(f"User_id is {existing_user['user_id']}")


                    # Get user_id
                    get_user_id = "SELECT user_id FROM users WHERE user_name LIKE %s
AND pwd = %s"
                    sql_cursor.execute(get_user_id, (f"%{username}%", password))
                    user_id = sql_cursor.fetchone()

                    if existing_user:
                        session['username'] = username
                        session['user_id'] = existing_user['user_id']
                        session['email'] = email
```

```
                    return redirect(url_for('home.load_homepage'))
                else:
                    return "Incorrect username or password."

            else:
                return "Unknown form type was submitted."

        # Close connection to GCP
    finally:
        db_conn.close()
return render_template('login.html')
```

Hannah: Trigger:

```
DELIMITER //
CREATE PROCEDURE update_previous_user_win_percents()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE varUserId INT;
    DECLARE varTotalBattles INT DEFAULT 0;
    DECLARE varTotalWins INT DEFAULT 0;
    DECLARE varWinPercentage INT DEFAULT 0;
    DECLARE idcur CURSOR FOR SELECT user_id FROM users;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN idcur;
      cloop: LOOP
      FETCH idcur INTO varUserId;
      IF done THEN
          LEAVE cloop;
      END IF;


      SELECT COUNT(*), SUM(B.win_loss_outcome)
      INTO varTotalBattles, varTotalWins
      FROM battles B
      JOIN user_teams UT ON B.user_team_id = UT.user_team_id
      WHERE UT.user_id = varUserId
      GROUP BY UT.user_id;

      IF varTotalBattles IS NULL OR varTotalBattles = 0 THEN
          SET varWinPercentage = 0;
      ELSE
          SET varWinPercentage = ROUND((varTotalWins / varTotalBattles) * 100);
      END IF;


      UPDATE users
      SET win_percentage = varWinPercentage
      WHERE user_id = varUserId;
      END LOOP;

    CLOSE idcur;
END;
//
DELIMITER ;
```

Hannah Stored Procedure:

Get the number of badges earned by a user -- JOIN

```
get_num_badges = """
   SELECT COUNT(DISTINCT B.gym_id) AS badge_nums
   FROM user_teams UT NATURAL JOIN battles B
   WHERE UT.user_id = %s AND B.win_loss_outcome = 1
"""
sql_cursor.execute(get_num_badges, (user_id, ))
num_badges_result = sql_cursor.fetchone()
badges_earned = num_badges_result['badge_nums']


# Get the percentage win/loss rate -- JOIN, GROUP BY
get_win_rate = """
   SELECT ((COUNT(B2.battle_id) / total_battles.num_battles)*100) AS win_percentage
   FROM (
      SELECT UT.user_id, COUNT(B.battle_id) AS num_battles
      FROM user_teams UT NATURAL JOIN battles B
      WHERE UT.user_id = %s
      GROUP BY UT.user_id
   ) AS total_battles JOIN user_teams UT2 ON UT2.user_id = total_battles.user_id
   JOIN battles B2 ON B2.user_team_id = UT2.user_team_id
   WHERE total_battles.user_id = %s AND B2.win_loss_outcome = 1

"""
sql_cursor.execute(get_win_rate, (user_id, user_id))
win_rate_result = sql_cursor.fetchone()
win_loss_rate = win_rate_result['win_percentage']
print(f"win loss rate is = {win_rate_result}")
```

Avg battle time

```
# Get the average battle time
get_avg_battle_time = """
   SELECT AVG(TIMESTAMPDIFF(MINUTE, B.start_time, B.end_time)) AS avg_time
   FROM user_teams UT NATURAL JOIN battles B
   WHERE UT.user_id = %s
"""
sql_cursor.execute(get_avg_battle_time, (user_id, ))
avg_battle_time_result = sql_cursor.fetchone()
avg_battle_time = avg_battle_time_result['avg_time']
print(f"Avg battle time result is = {avg_battle_time_result}")
```

Hannah Trigger update win percentage:

```
trigger:
CREATE TRIGGER  update_win_percentage
AFTER INSERT ON battles
FOR EACH ROW
BEGIN
   DECLARE var_user_id INT;
   DECLARE var_total_battles INT DEFAULT 0;
   DECLARE var_total_wins INT DEFAULT 0;
   DECLARE var_win_percentage INT DEFAULT 0;

   SELECT user_id INTO var_user_id
   FROM user_teams
   WHERE user_team_id = NEW.user_team_id;

   SELECT COUNT(), SUM(B.win_loss_outcome)
   INTO var_total_battles, var_total_wins
   FROM battles B
   JOIN user_teams UT ON B.user_team_id = UT.user_team_id
   WHERE UT.user_id = var_user_id
   GROUP BY UT.user_id;

   IF var_total_battles IS NULL OR var_total_battles = 0 THEN
      SET var_win_percentage = 0;
   ELSE
      SET var_win_percentage = ROUND((var_total_wins /
var_total_battles) 100);
   END IF;
```

```
    UPDATE users
    SET win_percentage = var_win_percentage
    WHERE user_id = var_user_id;
END
```