Team 002 iCAN:
Tanjie McMeans
Hannah Kim
Mengmeng Fang
Pierre Font

Project Track 1 Report

Please list out changes in the directions of your project if the final project is different from your original proposal (based on your stage 1 proposal submission).

The final implementation of the project is definitely different from our original proposal in Stage 1. We decided to simplify the scope of our game.

We decided to scale back the idea of a "boss rush" game into a functional single-battle style system against one gym leader at a time. Also, the player's team status effects persisting between battles was not implemented, with the logic instead healing teams at the start of each new battle.

Creating a distinct, personality-driven AI turned out to be a huge, complex task so it was streamlined into a single tactical-AI that focuses on type-effectiveness and power of moves rather than using unique strategies or specific Pokemon types.

Discuss what you think your application achieved or failed to achieve regarding its usefulness.

The application was a success at achieving its primary goal, which was to develop a database for a Pokemon-styled game. It has CRUD operations, user authentication, team creation and a very simple turn-based battle simulation. We also implemented advanced features in SQL for user authentication and battle logic such as: stored procedures, transactions and triggers. Additionally, unlike current pokemon website games, our app was not as cluttered with blogs, ads, forums, and other miscellaneous items. We had simple functionalities for each view (user auth, making teams, battling, etc).

Something we wished we were able to achieve during this time was the ability to allow users for more customization. For example, initially, we had planned to have the users customize their user image in the game, but we did not plan for this in our overall database implementation. We did not realize that we would need to save the users custom images and colors in the database until it came to coding our app. However, our overall database design and implementation was very strong.

Discuss if you changed the schema or source of the data for your application

Both the data sources and schema for the app were changed from the original proposal. The project was initially designed to include two Kaggle datasets for Pokemon stats and images. After feedback from the TA, we included two more datasets for Pokemon type matchups and move/pp (power points).

To summarize, our original datasets were:

(“The Complete Pokemon Dataset”, https://www.kaggle.com/datasets/rounakbanik/pokemon/data, pokemon.csv)

(“Pokemon Image Dataset”, https://www.kaggle.com/datasets/kvpratama/pokemon-images-dataset)

The additional datasets we used for stages 3 and 4 were:

(“Pokemon Type Matchup Data”, https://www.kaggle.com/datasets/lunamcbride24/pokemon-type-matchup-data/data, PokeTypeMatchupData.csv)

(“Complete Competitive Pokemon Dataset” https://www.kaggle.com/datasets/n2cholas/competitive-pokemon-dataset?select=pokemon-data.csv, move-data.csv and pokemon-data.csv)

Our UML diagram and database relations changed significantly from their original design in Stage 1.

type_matchups was redesigned to become a table of integer multipliers based on attacking type and defending type. (**attacking_type, defending_type, multiplier**)

user_teams was included to separate concerns of the user data from **user_pokemon_team_members** data. This split helped refine the primary key for user_pokemon_team_members from a three attribute key to one that just includes the **user_team_id** and **user_team_member_id**.

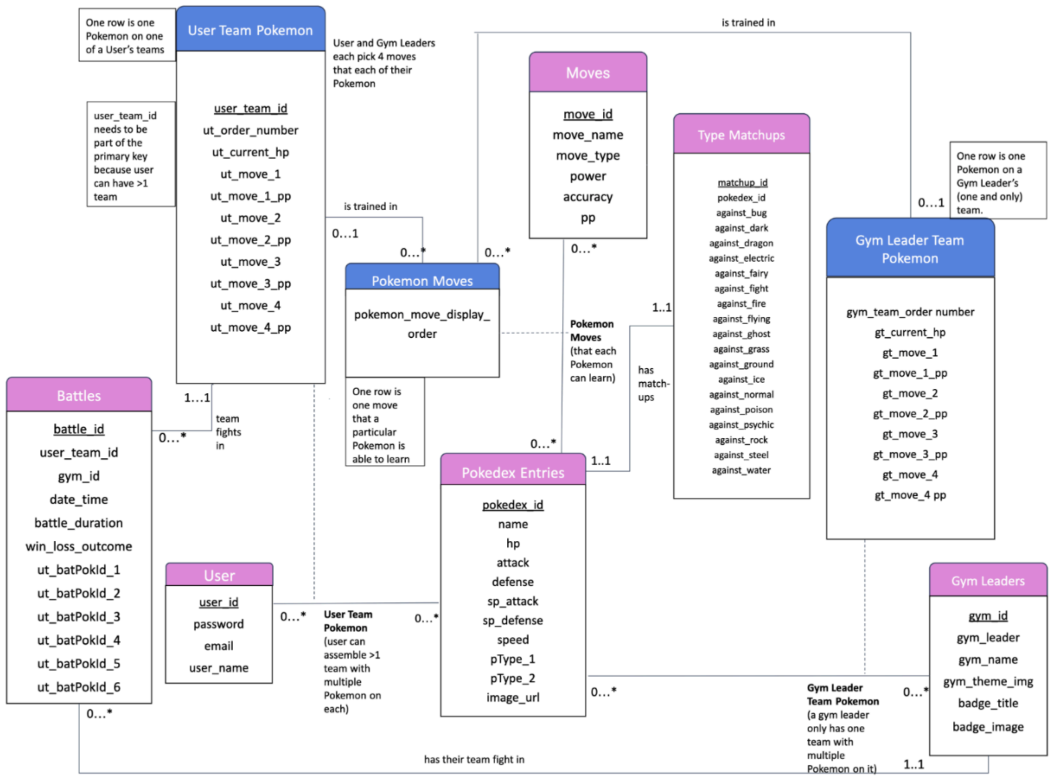
Redundant attributes were removed from the **battles** table as well.

(There is more discussion of the UML diagram changes in the next question below)

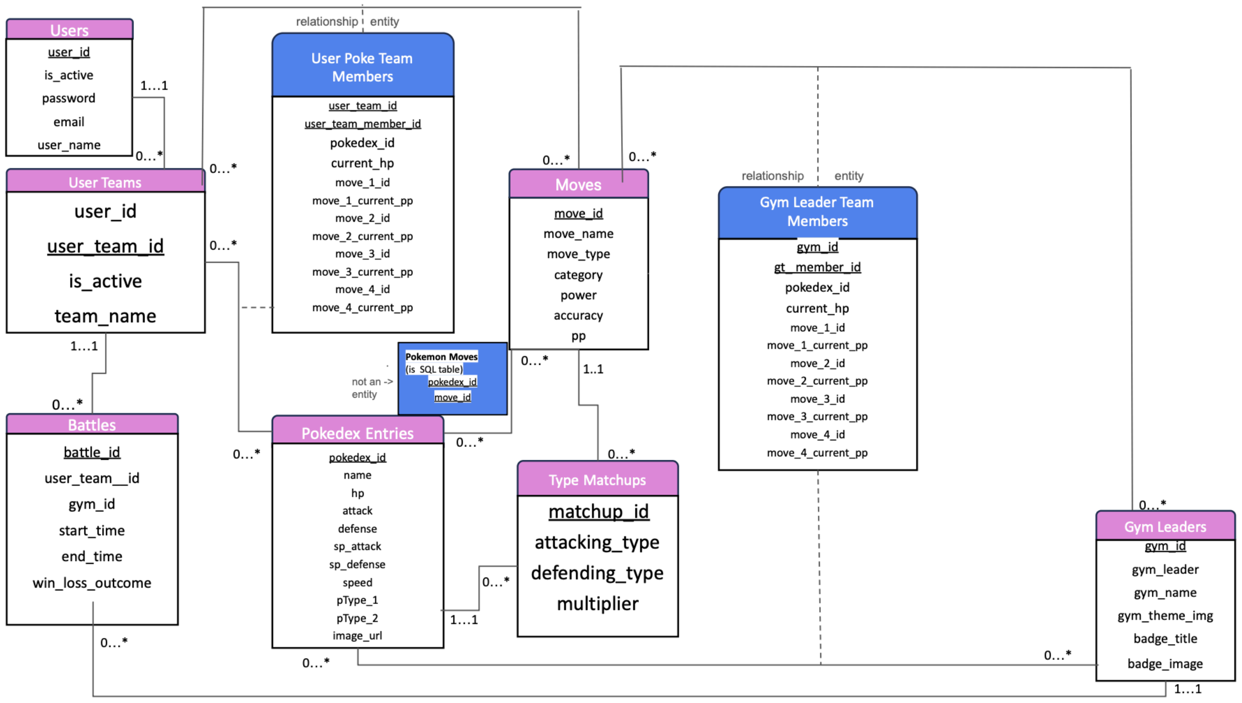
Discuss what you change to your ER diagram and/or your table implementations. What are some differences between the original design and the final design? Why? What do you think is a more suitable design?

Here is a comparison between our UML diagram for Stage 2 and our (final) UML diagram that we submitted for Stage 3 and are using for Stage 4:

UML Diagram for Stage 2 (Next Page):



UML Diagram for Stages 3 and 4:



One big change we made between the two UML diagrams was in the type matchups table. As our TA pointed out in his feedback for stage 2, it is hard to look up how the type of a move (that is an attack from one Pokemon) and the type(s) of a second Pokemon (that is defending against the attack) contribute to the multiplier that determines the damage caused by that move. In the new type_matchups table, a tuple includes the attacking_type of the attacking move from the moves_table (because it is the type of the attacking move, not the type(s) of the attacking Pokemon, that influences the damage done), the defending_type of the Pokemon being attacked, and the corresponding multiplier given to the “power” of the attacking move from the moves table. Since a defending Pokemon can have up to two types, there can be up to two tuples in type_matchups that determine the outcome of a particular attack, and if there are two corresponding multipliers for these tuples, they are multiplied times each other before being multiplied times the power of the attack. The final schema for type_matchps is actually slightly different than shown in the final UML diagram:

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

matchup_id is no longer the primary key, and should be removed as a column (we would have done this if we had more time). The primary key consists of the attacking_type (which corresponds to move_type on the moves table) and defending_type (which can correspond to either pType_1 or pType_2 on the pokedex_entries table). These are not foreign keys, though, and the relationship between type_matchups and the moves and pokedex_entries is more complex than many-to-one to each of those two other tables (despite what is shown in the UML diagram) or as a relationship table in a many-many relationship between those two other tables because there are two different types in the pokedex_entries table, among other reasons.

Discuss what functionalities you added or removed. Why?

We added: better battle mechanics, including type effectiveness modifiers, move and pp (power points).

We removed: boss rush mode to simplify the game, battle persistence and game AI personalities which both couldn't be implemented in the time we had to complete the project but we would have liked to implement it. We also would have allowed the user to see a more detailed breakdown of past battles their teams have fought and search through them if we had more time.

Explain how you think your advanced database programs complement your application.

Stored procedures: the two main procedures in the battles themselves are **get_battle_state** and **process_battle_turn**. Both store a lot of logic about how battles work in the game including handling setup for a battle, creating a new battle record in the battles table, resetting team health and providing the calculations for turn-based damage for both the user and AI player. The get_battle_state stored procedure has the advanced SQL features of a join of multiple tables and a set union.

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
```

```sql
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
```

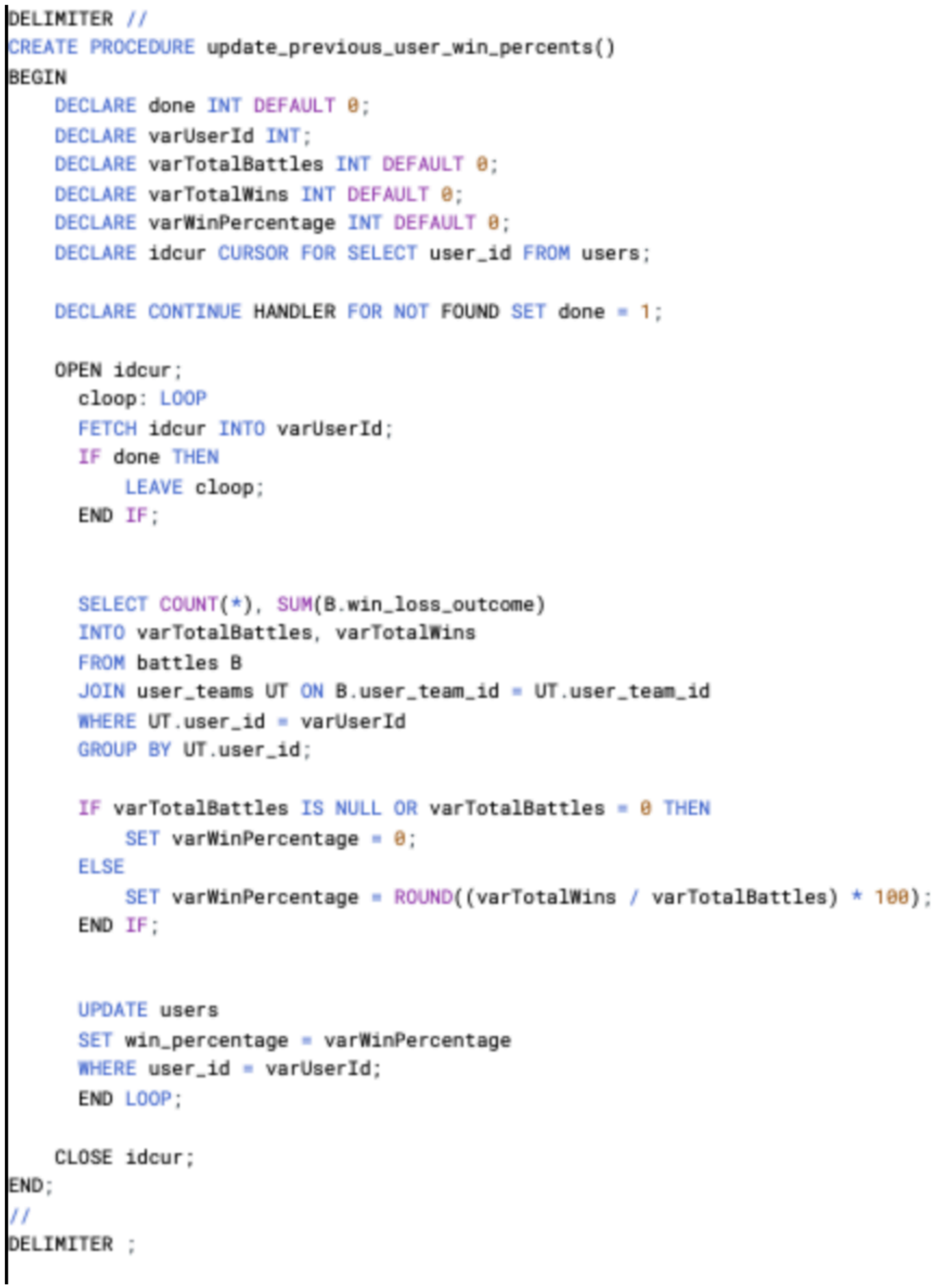
```
ELSE move_2_current_pp END,
       move_3_current_pp = CASE WHEN move_slot_used = 3 THEN move_3_current_pp - 1
ELSE move_3_current_pp END,
       move_4_current_pp = CASE WHEN move_slot_used = 4 THEN move_4_current_pp - 1
ELSE move_4_current_pp END
   WHERE user_team_id = attacker_team_id AND user_team_member_id = 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
```

There also is stored procedure for determining what “level” badge a user has won based on the gym leaders they have defeated:

```
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;
```

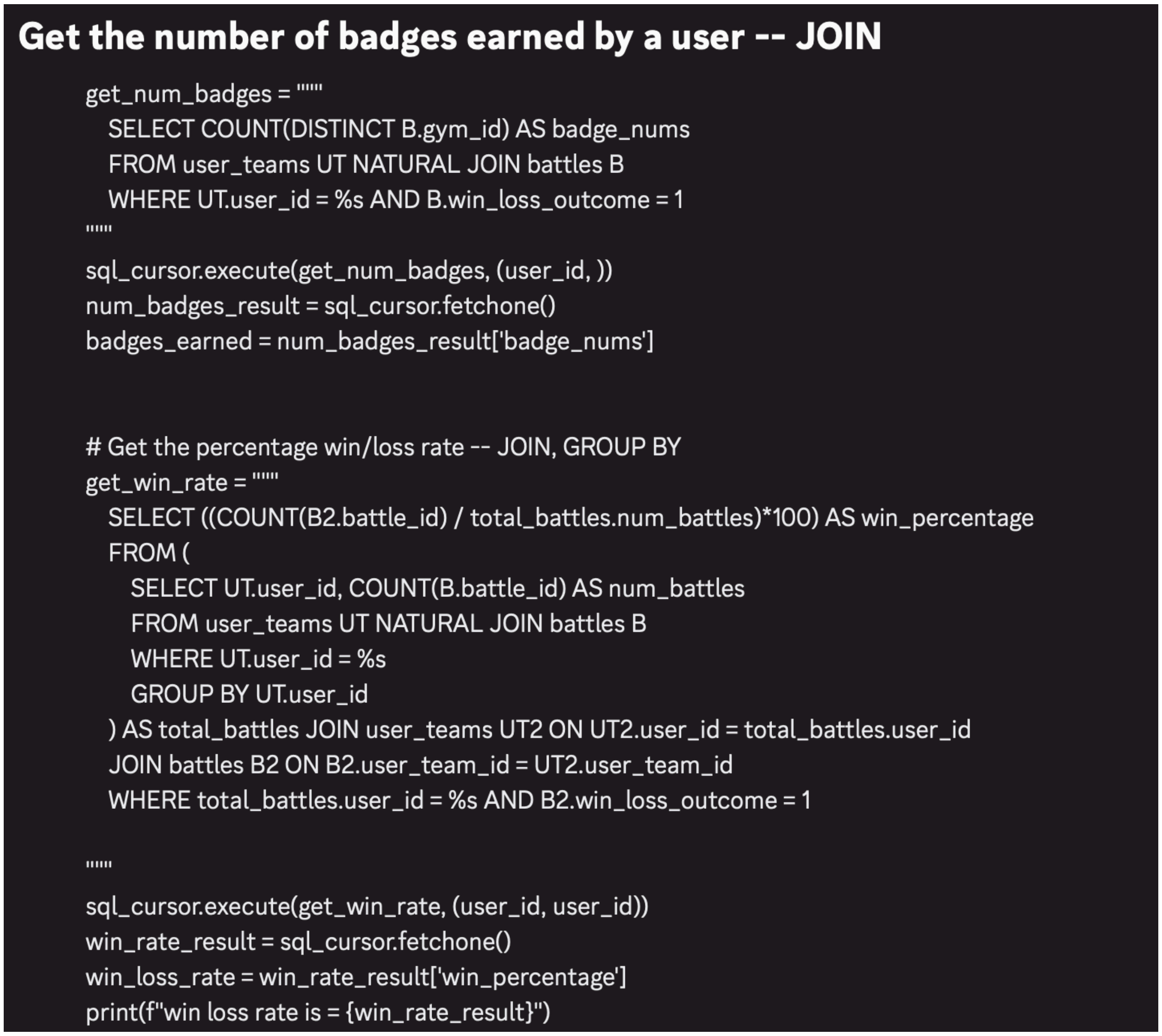
```
cloop: LOOP
FETCH usercur INTO var_user_id;
IF exit_loop THEN
LEAVE cloop;
END IF;

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 ;
```

Here is a stored procedure for updating the win/loss percentage for a particular user in their profile, which features aggregation via GROUP BY:



```sql
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 ;
```

Here is a stored procedure for counting the number of badges won by a user (a user earns a gym's "badge" when they beat that gym's gym leader in battle).


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}")
```

Transactions: the transaction in process_battle_turn (above) is important for reliable turn-based battles. Its default isolation level of repeatable read makes sure that the transaction for finding and storing variables for damage application each turn operates on a consistent basis. Once read at the start of a turn, Pokemon stats can't be changed by another concurrent action. This helps improve game integrity and provide a stable battle state for the game.

Triggers: We had a trigger, called trg_check_gym_defeat, to update the battle win loss outcome after there was an update on the gym leaders' team member table. This complemented our application because this way, we were able to easily identify if we had won because we could keep track of which gym team leaders' pokemons were fainting (no health) and when we had successfully won the battle (after all pokemon had no more health, or hp). An additional trigger we had was called trg_check_user_defeat that checked to see if a user had been defeated in battle. The logic for this trigger was similar to that of the trg_check_gym_defeat trigger, where we keep track of the number of pokemon on the user's team that have fainted. The last trigger we had recalculated and updated a user's win percentage after each battle. This is helpful because it allows the user to view their stats on their profile page, without having to do these calculations themselves. It also gives them an idea of their progress throughout the game.

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;
```

Each team member should describe one technical challenge that the team encountered. This should be sufficiently detailed such that another future team could use this as helpful advice if they were to start a similar project or where to maintain your project.

Tanjie: The biggest technical challenge for me was implementing the entire turn-based battle logic and encapsulating that within two stored procedures. Being able to implement so much logic purely in SQL at the data level definitely showed me the power of SQL as a language for communicating with databases. The second technical challenge was figuring out how to spin up a GCP cloud server and creating a database instance. Figuring out how to connect the cloud database to a Flask backend was also pretty challenging as well.

Mengmeng: The biggest technical challenge for me was connecting to Google Cloud from Visual Studio Code while testing game pages. Since I am based in China, internet restrictions and instability, especially when using a VPN, caused significant difficulties. I was able to create and test interface pages stored locally, but I couldn’t successfully test the backend, which required a connection to the database via GCP. Another technical challenge was setting up the Flask environment. In my personal project, I used PHP for the backend, which does not require any terminal commands. However, when using Flask and Python, I always have to set up the environment before running the page.

Furthermore, thinking about SQL queries directly for databases is different from real game design. In actual game design, we might not need to use advanced queries all the time. In most cases, simple queries are sufficient to carry out tasks. Therefore, incorporating advanced queries while still aligning with the game design is another challenge we tackled in this project.

Hannah: The biggest technical challenge for me was trying to think of advanced queries to use with a trigger and stored procedure. Because we had multiple views, it was simple to use non-advanced queries where the most we needed was a join. Because we only cared about showing a view or data on an HTML page for only one user, we did not need to GROUP BY for many queries. We simply grabbed the user_id and username from the login authentication page, stored that information in a flask session, and used that in the WHERE clause.

Pierre: The biggest technical challenge for me was trying to think of how to translate the advanced SQL queries that we had practiced in class into something that could be integrated into a larger stored procedure, transaction, or trigger, and how that could be integrated to work with the code in our backend or frontend. I also found it difficult to learn how to work with pushing and pulling code up to github, because although this is routine for more experienced coders, it was new for me.

Are there other things that changed comparing the final application with the original proposal?

Looking back at our earlier drafts, something we noticed was the change in our User-Interface plans. Initially, we wanted our gym leaders to have customized images for their gym leader and gym leader theme. Additionally, we wanted to create customized badges for each gym leader. However, due to time constraints, we decided to use the same image for all gym leaders, gym leader themes, and badges. Although this was not what we planned, our UI still looked neat with all images and color schemes in sync with each other.

Another thing that changed was the user's ability to customize their image. Initially, we wanted the users to be able to customize how their game character looked, however, as we thought of our database design, we did not consider how this would fit into the database. For example, we did not consider that the user's would customize their image but we would then need to also save their settings in the database. We only realized this as we were crafting our app in its final stages and realized we would need to create more columns in the database. Because this would require us to reanalyze our relationships and entities, we decided not to make any large changes at this time. Instead, we added a gender-neutral image for the users. The image is a lovely llama.

Describe future work that you think, other than the interface, that the application can improve on

Full boss rush styled game implementation would be a significant step and included a persistent challenge for players. Also, iterating on the simple AI logic and creating diverse AI personalities with strategy and tactics would be a great addition to the game. We had a lot of fun creating a game as a web app, but we would be interested in exploring the development of this application in collaboration with game engines, which are known to be more powerful. Currently, the game logic depends heavily on javascript, with a lot of back and forth between the front end and back end. Lastly, improving on battle mechanics would enhance the game further. If we are able to improve the communication between the front end and backend without having to hard code multiple javascript functions to handle these click interactions, we believe the UI and game flow could be a lot smoother.

Describe the final division of labor and how well you managed teamwork.

Tanjie: As Tech Lead, my role was to implement the gym leader and battle AI systems, while also providing technical mentorship, leading code reviews and resolving blockers. This role involved writing some of the more complex battle logic that serves as an important part to the battle mechanics, including the stored procedures, as well as figuring out how to install the architecture for the GCP cloud database server and implement a Flask backend API.

Mengmeng: I contributed as much as I could at each stage of this project. Even though I had problems connecting to GCP via Visual Studio Code, I designed several game pages in HTML as well as the corresponding backend code. I collaborated with my teammates to get those pages working. I also helped design advanced database components, such as triggers and stored procedures, for our project.

Everyone: Collaborated on database design, schema, DDL and DML operations, cleaning and storing outside data sources into the database, writing SQL queries, indexing and query optimization.

Hannah: I helped with creating views and a backend for user authentication, the homepage, the profile page, and the user teams and moves page. Along with this, I helped with connecting everyone’s HTML designs, sections, and functions with the rest of the application.

Pierre: I worked on the SQL queries and advanced SQL features, backend code, uploading new data to our database as needed and modifying existing data, and wherever else my help was needed.

To get some points back in the earlier stages, you can add a section with two subsections for Stage 2 and 3 respectively. In each subsection, you should

mention the state of your design/implementation before the feedback, the feedback itself and the change you have made as per the feedback. If there was no feedback for a stage (or if you have not addressed anything), you can leave the corresponding subsections empty.

Here is listed our TA feedback from Stage 2 and Stage 3:

Stage 2:

Question 1. "In the user table, email should determine name, right? You won't probably allow two names with same email."

Here is the SQL command we have written to create the users table (we are keeping the names of all our tables plural now):

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

Here email is designated as unique. I am not sure if we can leave the table this way or if we need to have email as part of the primary key of users (along with user_id, or whether because email is a candidate key for users the table needs to be split up to users(user_id (PK), is_active, pwd, user_name) and a separate table emails(user_id (PK), email) (or should user_id be gotten rid of altogether and email be used as the primary key of users)?

2. **"Instead of connecting type matchup with pokemon id, you can have a matrix that you can filter by pokemon type 1 and 2. Refer to this: https://pokemondb.net/type."**

We've made a lot of changes to type matchups. Our type_matchups table creation command in SQL is currently this:

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

The attacking type refers to the "type" in the moves section (ie, it is the type of the attacking move and not necessarily the ptype_1 or ptype_2 of the attacking Pokemon). The defending type refers to either the ptype_1 or the ptype_2 of the defending pokemon (whichever type matchup you are looking up). Calculating the type effect of an attack on a Pokemon involves looking up two multipliers from the pokemon_moves table: one multiplier for the match between the attacking move type and the defending Pokemon's ptype_1 and another lookup for the match between the attacking move type and the defending Pokemon's ptype_2, if any (this can be null so you may want to do a right/left join when querying for it). These two multipliers are then multiplied by each other to give the total_defensive_multiplier as affected by types. Here is a sample query:

```
SELECT TM1.attacking_type, (TM1.multiplier * TM2.multiplier) AS
total_defensive_multiplier
FROM Pokedex_Entries p
JOIN Type_Matchups TM1 ON p.PType_1 = TM1.defending_type
LEFT JOIN Type_Matchups TM2 ON p.PType_2 = TM2.defending_type
        AND TM1.attacking_type = TM2.attacking_type
WHERE p.name = “Pikachu”;
```

This query may not be worded in the best way, and we may change it to involve a set operator instead, but I hope it gives you an idea of how the type matchups work. (Please see also the discussion of UML diagram changes in the report above.)

3. **I think all Pokemon cannot be trained on all moves. How do you ensure that?**

The pokemon moves table, which reflects the many-many relationship between the pokedex entries entity and the moves entity, will have pokedex_id and move_id in it and allow someone to query what moves a particular pokemon is able to learn.

Also, we have decided not to implement what we told you on Campuswire about preventing users from picking the moves they want from all learnable moves for their Pokemon. The Moves table has all moves. The Pokemon Moves table shows, for given pokedex_id, all the moves that Pokemon is able to learn. The user can choose from the Pokemon Moves table (or the front end equivalent of it) which four moves they want each of their Pokemon to have in battle when they assemble their team.

Please see our updated UML diagram above in the report for more details.

Stage 3:

1. Query shows impact of two indexes (instead of three). Also, the set operation in it was easily achievable with an OR, so better to make the query simpler in your final application.
Answer (Mengmeng):

In our game design, the main focus is on the battle experience, where players enjoy watching two Pokémon fight against each other. This is the part we dedicated most of our effort to. As a result, we intentionally simplified earlier stages of the game, such as team building and Pokémon selection.

To make the selection process straightforward, we allow players to search for specific Pokémon by using keywords from their names. This lets them view the stats and features of the Pokémon directly, helping them build a team more efficiently without needing complex queries. Therefore, while we experimented with set operations and indexes during development, the final implementation prioritized simplicity and user experience.

2. For query 3, A better explanation on why cost does not change for different indexes would be helpful.

3. In query 3, what you are trying to find and what you are reporting mismatches a little bit as the same pokemon is reported for different moves.

4. Query 4 does not seem very relevant to your application.
Answer (Mengmeng): In our game, players do not have direct access to the gym leader’s team when they first challenge the gym. However, since the gym team remains unchanged, players can learn from their initial failures and adjust their strategy accordingly. After losing a battle, the player can create a new team with different Pokémon and challenge the gym again.

While building their team, players can view the stats and features of available Pokémon, which allows them to make more informed decisions based on their previous experience with the gym. For this reason, Query 4 was not implemented in the actual game, as the game design already provides an indirect way for players to adapt and strategize without needing to calculate or display average HP comparisons.