**Final Project Report**

1. **Accomplishments**

Our project Pokemon Trainer Hub allows users to create and manage accounts to keep track of Pokemons’ attributes and levels. It also allows users to make their Pokemon info private so that this can be hidden for battles. Additionally, users are able to view the public information of other users to plan ahead of battles. Users could synchronize their accounts in this database with their game accounts in real life.

1. **Usefulness**

Our Trainer info relates the information of different Pokemon trainers to their training pokemons. Users could view what pokemon contain in other users’ account if they set the pokemon to be public. Users could also read Pokemon descriptions in the table pokemon for more detailed information. Users could adopt their own pokemon and design the attributes they like, which other databases cannot satisfy. Our applications are different from similar applications. Instead of a public database, we allow each user to create their own accounts to record the Pokemon information. It is an opportunity to show your own pokemon to other users and record the game experience. In the traditional Pokemon database, there is only general data about the attributes of different Pokemons. However, in the real game, even pokemon with the same type would differ in some attributes. In our application, since every player would have an account, the users allow representing their specialness to other players.

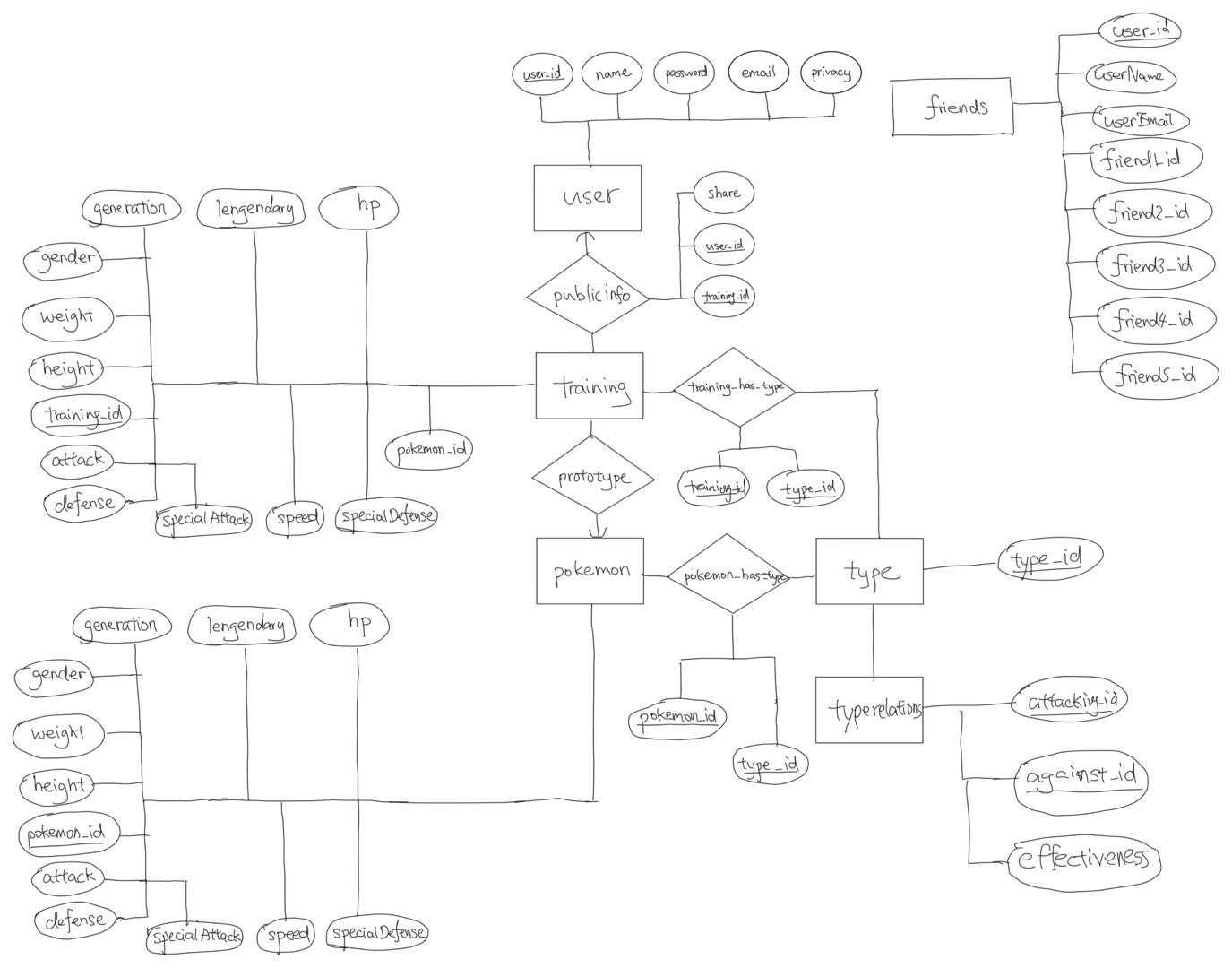
1. **Data Information**

We have data from users, their friends, pokemon and relationships between each entity. The relationship between users and pokemon is characterized by training. The relationship between users is characterized by friends. While additional data is needed to describe the attributes of a pokemon.

Each user is characterized by user\_id. They also have names, password, registered email and friends. The friend relationship is represented in friends table. A pokemon has attack, defense, specialAttack, and lots of other attributes. But the attribute type of a pokemon is a special attribute for a pokemon may have one or more types. We have additional tables for that. Training specifies the training relationship between an adopted pokemon and its prototype. It is similar to the pokemon table despite that a user is authorized to manipulate data in training. A public info table contains user\_id, training\_id, and share\_info. It associates users table with training table.

The type information contains all of the types for pokemons. Each type has a unique typeID. We make several updates to the type table. First, a type table now only has one column, type\_id. The relationship between a pokemon and its types is represented by pokemon\_has\_types table. The same to the training table. In addition, we add another table to illustrate the against, attacking relation between types.

1. **Updated ER Diagram**

****

1. **Final Database Schema and Index Design Analysis**

1. Table pokemon:

CREATE TABLE pokemon(

pokemon\_ID INT PRIMARY KEY,

hp INT,

attack INT,

defense INT,

specialAttack INT,

specialDefense INT,

speed INT,

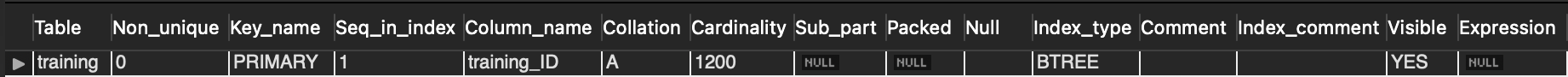
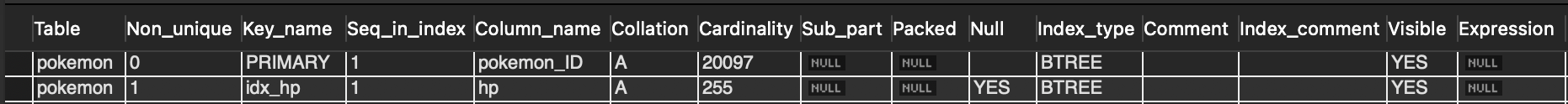
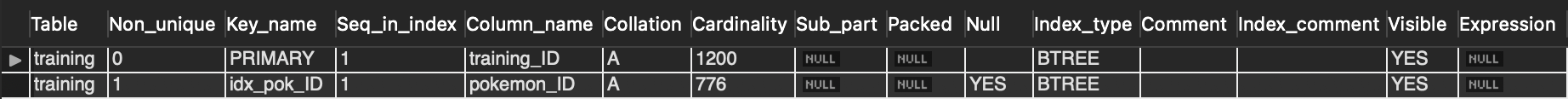
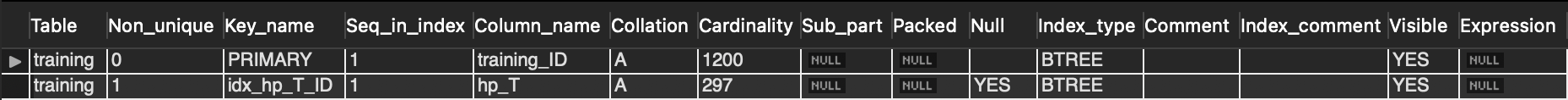
height INT,

weight INT,

gender INT,

generation INT,

legendary BOOLEAN);

1. Before adding index:  
   -> Table scan on <temporary> (actual time=0.000..0.024 rows=414 loops=1)  
   -> Aggregate using temporary table (actual time=2.672..2.725 rows=414 loops=1)  
   -> Nested loop inner join (cost=1435.78 rows=400) (actual time=0.062..2.400 rows=543 loops=1)  
   -> Filter: (training.pokemon\_ID is not null) (cost=126.12 rows=1200) (actual time=0.046..0.571 rows=1200 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.044..0.446 rows=1200 loops=1)  
   -> Filter: (pokemon.hp > training.hp\_T) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=0 loops=1200)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=1200)  
   Current Index:  
     
   
2. Experiment:  
   1. CREATE INDEX idx\_hp on Pokemon.pokemon(hp);  
     
     
   -> Table scan on <temporary> (actual time=0.000..0.020 rows=414 loops=1)  
   -> Aggregate using temporary table (actual time=2.293..2.337 rows=414 loops=1)  
   -> Nested loop inner join (cost=1435.78 rows=400) (actual time=0.066..2.050 rows=543 loops=1)  
   -> Filter: (training.pokemon\_ID is not null) (cost=126.12 rows=1200) (actual time=0.047..0.479 rows=1200 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.046..0.376 rows=1200 loops=1)  
   -> Filter: (pokemon.hp > training.hp\_T) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=0 loops=1200)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=1200)   
   2. DROP INDEX idx\_hp on Pokemon.pokemon; CREATE INDEX idx\_pok\_ID on Pokemon.training(pokemon\_ID);  
   -> Table scan on <temporary> (actual time=0.001..0.022 rows=414 loops=1)  
   -> Aggregate using temporary table (actual time=2.913..2.963 rows=414 loops=1)  
   -> Nested loop inner join (cost=1435.78 rows=400) (actual time=0.063..2.609 rows=543 loops=1)  
   -> Filter: (training.pokemon\_ID is not null) (cost=126.12 rows=1200) (actual time=0.044..0.698 rows=1200 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.043..0.576 rows=1200 loops=1)  
   -> Filter: (pokemon.hp > training.hp\_T) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=0 loops=1200)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=1200)  
   3.DROP INDEX idx\_pok\_ID on Pokemon.training; CREATE INDEX idx\_hp\_T\_ID on Pokemon.training(hp\_T);  
     
   -> Table scan on <temporary> (actual time=0.001..0.020 rows=414 loops=1)  
   -> Aggregate using temporary table (actual time=2.366..2.410 rows=414 loops=1)  
   -> Nested loop inner join (cost=1435.78 rows=400) (actual time=0.067..2.122 rows=543 loops=1)  
   -> Filter: (training.pokemon\_ID is not null) (cost=126.12 rows=1200) (actual time=0.050..0.536 rows=1200 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.048..0.424 rows=1200 loops=1)  
   -> Filter: (pokemon.hp > training.hp\_T) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=0 loops=1200)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=1200)
3. Explanation:  
   Since the advanced query involves one subquery which does the most work, the report of the "EXPLAIN ANALYZE" command does not reflect what the new index the operations take advantage of.  
   Experiment 1:  
   I choose the Attribute hp of Table pokemon as one index in addition to the primary key since the subquery compares the hp of pokemon with hp\_T of training. This index actually slows down this advanced query from 0.228 sec to 0.357 sec on the duration. However, it indeed speeds up some operations during execution. The time for aggregation is reduced to 2.337 from 2.725, the time for inner join is reduced to 2.050 from 2.400, and some other small improvements by pinpointing the Attribute hp more efficiently.  
   Experiment 2:  
   I choose the Attribute pokemon\_ID of Table training as one index in addition to the primary key since the query groups pokemon\_ID of Table temp. This index actually slows down this advanced query from 0.228 sec to 0.424 sec on the duration and almost all operations during execution. One possible reason is that this index forces the server to do some extra useless work.  
   Experiment 3:  
   I choose the Attribute hp\_T of Table training as one index in addition to the primary key since the subquery compares hp\_T of training with hp of pokemon. This index actually speeds up this advanced query from 0.228 sec to 0.218 sec on the duration. However, it indeed slows down some operations during execution. The time for aggregation is reduced to 2.410 from 2.725, the time for inner join is reduced to 2.122 from 2.400, and some other small improvements by pinpointing the Attribute hp\_t more efficiently.

2. Table training:

CREATE TABLE training(

pokemon\_ID INT,

training\_ID INT PRIMARY KEY,

hp\_T INT,

attack\_T INT,

defense\_T INT,

specialAttack\_T INT,

specialDefense\_T INT,

speed\_T INT,

height\_T INT,

weight\_T INT);

* a. Before applying index:  
    
    
  -> Limit: 15 row(s) (actual time=0.049..0.052 rows=15 loops=1)  
   -> Sort: <union temporary>.hp\_T, limit input to 15 row(s) per chunk (cost=2.50 rows=0) (actual time=0.049..0.050 rows=15 loops=1)  
   -> Table scan on <union temporary> (actual time=0.003..0.010 rows=130 loops=1)  
   -> Union materialize with deduplication (actual time=31.043..31.047 rows=15 loops=1)  
   -> Inner hash join (pokemon.pokemon\_ID = training.pokemon\_ID) (cost=4803.95 rows=321) (actual time=0.960..15.103 rows=60 loops=1)  
   -> Filter: ((pokemon.attack < 120) and (pokemon.specialAttack < 120)) (cost=13.58 rows=217) (actual time=0.009..13.085 rows=7378 loops=1)  
   -> Table scan on pokemon (cost=13.58 rows=19541) (actual time=0.008..10.770 rows=20001 loops=1)  
   -> Hash  
   -> Filter: ((training.attack\_T > 120) and (training.specialAttack\_T > 120)) (cost=122.00 rows=133) (actual time=0.035..0.878 rows=177 loops=1)  
   -> Table scan on training (cost=122.00 rows=1200) (actual time=0.028..0.756 rows=1200 loops=1)  
   -> Inner hash join (pokemon.pokemon\_ID = training.pokemon\_ID) (cost=4803.95 rows=321) (actual time=0.900..15.788 rows=74 loops=1)  
   -> Filter: ((pokemon.defense < 170) and (pokemon.specialDefense < 170)) (cost=13.58 rows=217) (actual time=0.008..13.480 rows=10414 loops=1)  
   -> Table scan on pokemon (cost=13.58 rows=19541) (actual time=0.006..11.153 rows=20001 loops=1)  
   -> Hash  
   -> Filter: ((training.defense\_T > 170) and (training.specialDefense\_T > 170)) (cost=122.00 rows=133) (actual time=0.028..0.823 rows=132 loops=1)  
   -> Table scan on training (cost=122.00 rows=1200) (actual time=0.014..0.721 rows=1200 loops=1)  
    
  b. Applying index on training.attack\_T:  
    
  -> Limit: 15 row(s) (actual time=0.040..0.043 rows=15 loops=1)  
   -> Sort: <union temporary>.hp\_T, limit input to 15 row(s) per chunk (cost=2.50 rows=0) (actual time=0.040..0.042 rows=15 loops=1)  
   -> Table scan on <union temporary> (actual time=0.000..0.007 rows=130 loops=1)  
   -> Union materialize with deduplication (actual time=1.830..1.834 rows=15 loops=1)  
   -> Nested loop inner join (cost=292.73 rows=17) (actual time=0.085..0.932 rows=60 loops=1)  
   -> Filter: ((training.attack\_T > 120) and (training.specialAttack\_T > 120) and (training.pokemon\_ID is not null)) (cost=126.12 rows=153) (actual time=0.070..0.628 rows=177 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.044..0.506 rows=1200 loops=1)  
   -> Filter: ((pokemon.attack < 120) and (pokemon.specialAttack < 120)) (cost=0.99 rows=0) (actual time=0.002..0.002 rows=0 loops=177)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=177)  
   -> Nested loop inner join (cost=271.61 rows=15) (actual time=0.039..0.775 rows=74 loops=1)  
   -> Filter: ((training.defense\_T > 170) and (training.specialDefense\_T > 170) and (training.pokemon\_ID is not null)) (cost=126.12 rows=133) (actual time=0.035..0.578 rows=132 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.031..0.486 rows=1200 loops=1)  
   -> Filter: ((pokemon.defense < 170) and (pokemon.specialDefense < 170)) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=1 loops=132)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=132)  
    
  c. Applying index on training.specialAttack\_T:  
    
  -> Limit: 15 row(s) (actual time=0.042..0.045 rows=15 loops=1)  
   -> Sort: <union temporary>.hp\_T, limit input to 15 row(s) per chunk (cost=2.50 rows=0) (actual time=0.042..0.044 rows=15 loops=1)  
   -> Table scan on <union temporary> (actual time=0.000..0.008 rows=130 loops=1)  
   -> Union materialize with deduplication (actual time=2.082..2.086 rows=15 loops=1)  
   -> Nested loop inner join (cost=302.18 rows=18) (actual time=0.070..1.084 rows=60 loops=1)  
   -> Filter: ((training.attack\_T > 120) and (training.specialAttack\_T > 120) and (training.pokemon\_ID is not null)) (cost=126.12 rows=161) (actual time=0.053..0.705 rows=177 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.048..0.588 rows=1200 loops=1)  
   -> Filter: ((pokemon.attack < 120) and (pokemon.specialAttack < 120)) (cost=0.99 rows=0) (actual time=0.002..0.002 rows=0 loops=177)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.002..0.002 rows=1 loops=177)  
   -> Nested loop inner join (cost=271.61 rows=15) (actual time=0.047..0.861 rows=74 loops=1)  
   -> Filter: ((training.defense\_T > 170) and (training.specialDefense\_T > 170) and (training.pokemon\_ID is not null)) (cost=126.12 rows=133) (actual time=0.043..0.637 rows=132 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.038..0.533 rows=1200 loops=1)  
   -> Filter: ((pokemon.defense < 170) and (pokemon.specialDefense < 170)) (cost=0.99 rows=0) (actual time=0.001..0.002 rows=1 loops=132)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=132)  
    
  d. Applying index on training.hp\_T:  
    
  -> Limit: 15 row(s) (actual time=0.044..0.046 rows=15 loops=1)  
   -> Sort: <union temporary>.hp\_T, limit input to 15 row(s) per chunk (cost=2.50 rows=0) (actual time=0.043..0.044 rows=15 loops=1)  
   -> Table scan on <union temporary> (actual time=0.001..0.008 rows=130 loops=1)  
   -> Union materialize with deduplication (actual time=2.126..2.130 rows=15 loops=1)  
   -> Nested loop inner join (cost=271.61 rows=15) (actual time=0.084..1.167 rows=60 loops=1)  
   -> Filter: ((training.attack\_T > 120) and (training.specialAttack\_T > 120) and (training.pokemon\_ID is not null)) (cost=126.12 rows=133) (actual time=0.064..0.774 rows=177 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.055..0.642 rows=1200 loops=1)  
   -> Filter: ((pokemon.attack < 120) and (pokemon.specialAttack < 120)) (cost=0.99 rows=0) (actual time=0.002..0.002 rows=0 loops=177)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.002..0.002 rows=1 loops=177)  
   -> Nested loop inner join (cost=271.61 rows=15) (actual time=0.053..0.818 rows=74 loops=1)  
   -> Filter: ((training.defense\_T > 170) and (training.specialDefense\_T > 170) and (training.pokemon\_ID is not null)) (cost=126.12 rows=133) (actual time=0.047..0.585 rows=132 loops=1)  
   -> Table scan on training (cost=126.12 rows=1200) (actual time=0.041..0.490 rows=1200 loops=1)  
   -> Filter: ((pokemon.defense < 170) and (pokemon.specialDefense < 170)) (cost=0.99 rows=0) (actual time=0.002..0.002 rows=1 loops=132)  
   -> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=132)  
  e. explanation and analysis report:  
  For experiment 1, I choose to apply indexing on training.attack\_T because my advanced query consisting of condition check of attack\_T with some constants. By looking at the explain analyze result of both before and after index application, we found that when doing the filter process, one with attack\_T indexing is doing much faster than the original query(0.628 < 0.878).  
  For experiment 3, I choose to apply indexing on training.hp\_T because I've included an ordering statement at the end of the query by hp\_T attribute in the final table. Looking at the result of both original and one after application of indexing, we have the evidence to say that there's some improvement in the processing speed during the sorting stage in our query. (0.044 < 0.05) However, compared to the former two indexing strategies, this improvement is so tiny that could even be ignored. This is because that we are doing the sorting after a lot of filtering stages, and our times accessing into attack\_T and specialAttack\_T are huge compared to the time where we accessing hp\_T(number of rows are already limited where all rows left are satisfying the conditions).  
  For experiment 2, I choose to apply indexing on training.specialAttack\_T because similar to the first experiment, there's also condition checking process in my query for the attribute specialAttack. Looking at the result of explain analyze of both before and after the index application, we found that during the filtering process, applying indexing on column specialAttack is also faster than the original one (0.705 < 0.878) but slower than the previous experiment. Probably the reason is that inside the checking stage, attack\_T is checked first so when attack\_T is not satisfying the condition we automatically drop the row and don't care of the specialAttack.

3. Table friend\_table:

CREATE TABLE friend\_table(

user\_id varchar(8),

userName varchar(16),

userEmail varchar(32),

friend1\_id varchar(8),

friend2\_id varchar(8),

friend3\_id varchar(8),

friend4\_id varchar(8),

friend5\_id varchar(8));

1. Before adding index:



-> Sort: <union temporary>.legendary DESC (cost=2.50 rows=0) (actual time=0.074..0.108 rows=343 loops=1)

-> Table scan on <union temporary> (actual time=0.001..0.019 rows=343 loops=1)

-> Union materialize with deduplication (actual time=2.716..2.770 rows=343 loops=1)

-> Nested loop inner join (cost=270.59 rows=7) (actual time=0.071..1.262 rows=113 loops=1)

-> Filter: ((Pokemon.training.specialAttack\_T > Pokemon.training.attack\_T) and (Pokemon.training.hp\_T > 100) and (Pokemon.training.pokemon\_ID is not null)) (cost=125.10 rows=133) (actual time=0.050..0.615 rows=412 loops=1)

-> Table scan on training (cost=125.10 rows=1200) (actual time=0.047..0.473 rows=1200 loops=1)

-> Filter: ((Pokemon.pokemon.legendary = 1) and (Pokemon.training.specialDefense\_T > Pokemon.pokemon.defense)) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=0 loops=412)

-> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=Pokemon.training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=412)

-> Nested loop inner join (cost=367.59 rows=22) (actual time=0.043..1.199 rows=230 loops=1)

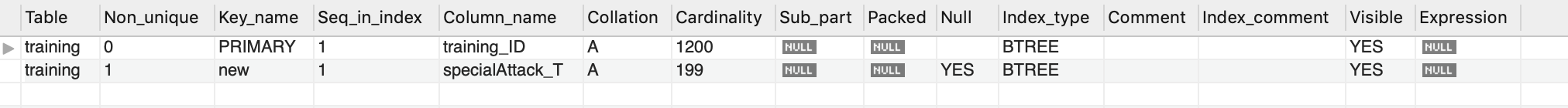
-> Filter: (((Pokemon.training.specialAttack\_T > 150) or (Pokemon.training.specialDefense\_T > 150)) and (Pokemon.training.hp\_T > 100) and (Pokemon.training.pokemon\_ID is not null)) (cost=125.10 rows=222) (actual time=0.038..0.589 rows=445 loops=1)

-> Table scan on training (cost=125.10 rows=1200) (actual time=0.035..0.438 rows=1200 loops=1)

-> Filter: (Pokemon.pokemon.legendary = 0) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=1 loops=445)

-> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=Pokemon.training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=445)

b. Applying index on special attack



-> Sort: <union temporary>.legendary DESC (cost=2.50 rows=0) (actual time=0.069..0.103 rows=343 loops=1)

-> Table scan on <union temporary> (actual time=0.001..0.018 rows=343 loops=1)

-> Union materialize with deduplication (actual time=2.601..2.656 rows=343 loops=1)

-> Nested loop inner join (cost=270.59 rows=7) (actual time=0.072..1.206 rows=113 loops=1)

-> Filter: ((Pokemon.training.specialAttack\_T > Pokemon.training.attack\_T) and (Pokemon.training.hp\_T > 100) and (Pokemon.training.pokemon\_ID is not null)) (cost=125.10 rows=133) (actual time=0.052..0.604 rows=412 loops=1)

-> Table scan on training (cost=125.10 rows=1200) (actual time=0.049..0.486 rows=1200 loops=1)

-> Filter: ((Pokemon.pokemon.legendary = 1) and (Pokemon.training.specialDefense\_T > Pokemon.pokemon.defense)) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=0 loops=412)

-> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=Pokemon.training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=412)

-> Nested loop inner join (cost=367.59 rows=22) (actual time=0.046..1.141 rows=230 loops=1)

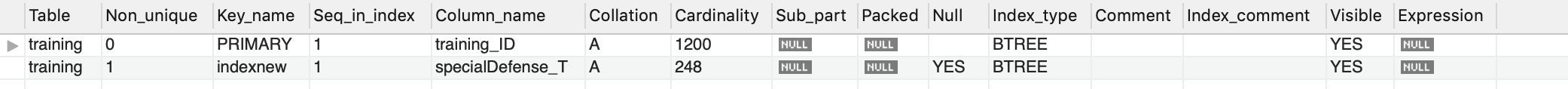
-> Filter: (((Pokemon.training.specialAttack\_T > 150) or (Pokemon.training.specialDefense\_T > 150)) and (Pokemon.training.hp\_T > 100) and (Pokemon.training.pokemon\_ID is not null)) (cost=125.10 rows=222) (actual time=0.043..0.601 rows=445 loops=1)

-> Table scan on training (cost=125.10 rows=1200) (actual time=0.041..0.442 rows=1200 loops=1)

-> Filter: (Pokemon.pokemon.legendary = 0) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=1 loops=445)

-> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=Pokemon.training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=445)

c. Applying index on special defense



-> Sort: <union temporary>.legendary DESC (cost=2.50 rows=0) (actual time=0.069..0.103 rows=343 loops=1)

-> Table scan on <union temporary> (actual time=0.001..0.018 rows=343 loops=1)

-> Union materialize with deduplication (actual time=2.601..2.656 rows=343 loops=1)

-> Nested loop inner join (cost=270.59 rows=7) (actual time=0.072..1.206 rows=113 loops=1)

-> Filter: ((Pokemon.training.specialAttack\_T > Pokemon.training.attack\_T) and (Pokemon.training.hp\_T > 100) and (Pokemon.training.pokemon\_ID is not null)) (cost=125.10 rows=133) (actual time=0.052..0.604 rows=412 loops=1)

-> Table scan on training (cost=125.10 rows=1200) (actual time=0.049..0.486 rows=1200 loops=1)

-> Filter: ((Pokemon.pokemon.legendary = 1) and (Pokemon.training.specialDefense\_T > Pokemon.pokemon.defense)) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=0 loops=412)

-> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=Pokemon.training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=412)

-> Nested loop inner join (cost=367.59 rows=22) (actual time=0.046..1.141 rows=230 loops=1)

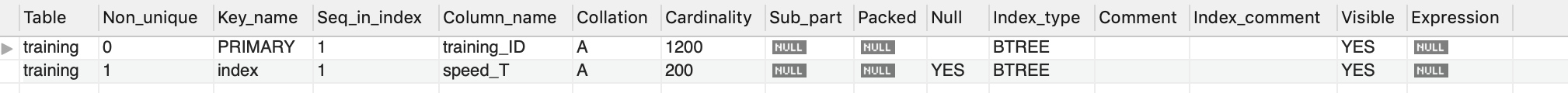
-> Filter: (((Pokemon.training.specialAttack\_T > 150) or (Pokemon.training.specialDefense\_T > 150)) and (Pokemon.training.hp\_T > 100) and (Pokemon.training.pokemon\_ID is not null)) (cost=125.10 rows=222) (actual time=0.043..0.601 rows=445 loops=1)

-> Table scan on training (cost=125.10 rows=1200) (actual time=0.041..0.442 rows=1200 loops=1)

-> Filter: (Pokemon.pokemon.legendary = 0) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=1 loops=445)

-> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=Pokemon.training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=445)

d. Applying index on speed



-> Sort: <union temporary>.legendary DESC (cost=2.50 rows=0) (actual time=0.085..0.126 rows=343 loops=1)

-> Table scan on <union temporary> (actual time=0.000..0.017 rows=343 loops=1)

-> Union materialize with deduplication (actual time=2.732..2.794 rows=343 loops=1)

-> Nested loop inner join (cost=270.59 rows=7) (actual time=0.077..1.343 rows=113 loops=1)

-> Filter: ((Pokemon.training.specialAttack\_T > Pokemon.training.attack\_T) and (Pokemon.training.hp\_T > 100) and (Pokemon.training.pokemon\_ID is not null)) (cost=125.10 rows=133) (actual time=0.055..0.638 rows=412 loops=1)

-> Table scan on training (cost=125.10 rows=1200) (actual time=0.051..0.506 rows=1200 loops=1)

-> Filter: ((Pokemon.pokemon.legendary = 1) and (Pokemon.training.specialDefense\_T > Pokemon.pokemon.defense)) (cost=0.99 rows=0) (actual time=0.002..0.002 rows=0 loops=412)

-> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=Pokemon.training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=412)

-> Nested loop inner join (cost=367.59 rows=22) (actual time=0.039..1.124 rows=230 loops=1)

-> Filter: (((Pokemon.training.specialAttack\_T > 150) or (Pokemon.training.specialDefense\_T > 150)) and (Pokemon.training.hp\_T > 100) and (Pokemon.training.pokemon\_ID is not null)) (cost=125.10 rows=222) (actual time=0.035..0.574 rows=445 loops=1)

-> Table scan on training (cost=125.10 rows=1200) (actual time=0.033..0.415 rows=1200 loops=1)

-> Filter: (Pokemon.pokemon.legendary = 0) (cost=0.99 rows=0) (actual time=0.001..0.001 rows=1 loops=445)

-> Single-row index lookup on pokemon using PRIMARY (pokemon\_ID=Pokemon.training.pokemon\_ID) (cost=0.99 rows=1) (actual time=0.001..0.001 rows=1 loops=445)

e. explanation and analysis report:

Case 1: I use a special attack for this case because in my SQL operation, I contain some comparisons between special attack and other values. After I apply this type of indexing, the sorting time is the same, but the filtering time also decreases from 0.565 to 0.552. Also, the nested loop inner join time decreases from 1.156 to 1.095. Overall, this is a good way to set an index. It improves the performance in both the filtering process and joining loop process.

Case 2: I use special defense for this case because the special defense is also an important attribute comparing with other targets. However, although the sorting time also does not change. The nested loop inner join time increases from 1.156 to 1.266. The filtering time improves a little from 0.565 to 0.558. But if we consider this as a whole, the performance is worse than the performance before the indexing. So we should not apply this.

Case 3: Speed is a value not influence in my selection of pokemon, because I only choose pokemon that has a high special attack or high special defense. However, speed is contracted with pokemon's attributions, so I want to use speed to set the index. When I use speed to sort, I found that the duration of the sort is increasing from 0.034 to 0.041. The nested loop time is decreasing from 1.156 to 1.085. The union time increases from 0.054 to 0.062. Although it has a time improvement in the nested loop, the overall performance improvement is not significant.

4. Table typerelation:

CREATE TABLE typerelation(

attacking\_id varchar(30),

against\_id varchar(30),

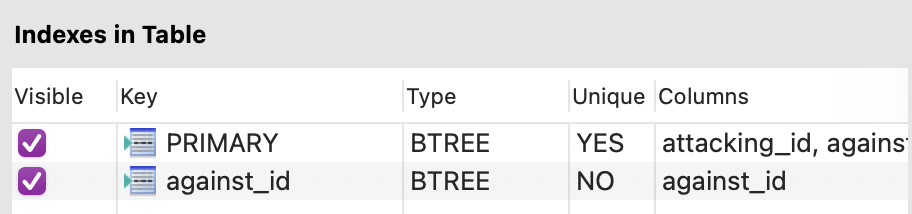
effectiveness decimal(2,1),

PRIMARY KEY(attacking\_id, against\_id),

FOREIGN KEY(attacking\_id) REFERENCES type(id),

FOREIGN KEY(against\_id) REFERENCES type(id));

* + Before adding Index



-> Limit: 15 row(s) (actual time=2.968..2.971 rows=15 loops=1)

-> Sort: <temporary>.s DESC, <temporary>.w, limit input to 15 row(s) per chunk (actual time=2.967..2.968 rows=15 loops=1)

-> Stream results (actual time=2.851..2.944 rows=35 loops=1)

-> Nested loop left join (actual time=2.849..2.904 rows=35 loops=1)

-> Table scan on strength (actual time=0.001..0.011 rows=35 loops=1)

-> Materialize (actual time=0.759..0.771 rows=35 loops=1)

-> Group aggregate: sum(typerelation.effectiveness) (actual time=0.083..0.689 rows=35 loops=1)

-> Index scan on typerelation using PRIMARY (cost=124.00 rows=1225) (actual time=0.058..0.385 rows=1225 loops=1)

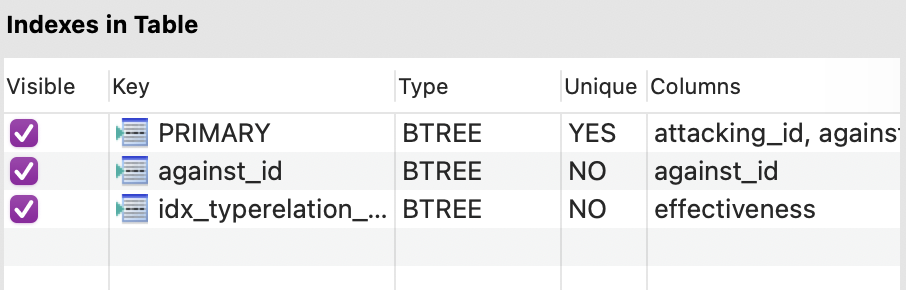
-> Index lookup on weakness using <auto\_key0> (against\_id=strength.attacking\_id) (actual time=0.001..0.001 rows=1 loops=35)

-> Materialize (actual time=0.060..0.061 rows=1 loops=35)

-> Group aggregate: sum(typerelation.effectiveness) (actual time=0.174..2.058 rows=35 loops=1)

-> Index scan on typerelation using against\_id (cost=124.00 rows=1225) (actual time=0.160..1.771 rows=1225 loops=1)

* + Applying index on effectiveness



-> Limit: 15 row(s) (actual time=3.123..3.126 rows=15 loops=1)

-> Sort: <temporary>.s DESC, <temporary>.w, limit input to 15 row(s) per chunk (actual time=3.123..3.124 rows=15 loops=1)

-> Stream results (actual time=3.047..3.102 rows=35 loops=1)

-> Nested loop left join (actual time=3.045..3.090 rows=35 loops=1)

-> Table scan on strength (actual time=0.001..0.005 rows=35 loops=1)

-> Materialize (actual time=0.781..0.787 rows=35 loops=1)

-> Group aggregate: sum(typerelation.effectiveness) (actual time=0.062..0.765 rows=35 loops=1)

-> Index scan on typerelation using PRIMARY (cost=124.00 rows=1225) (actual time=0.040..0.421 rows=1225 loops=1)

-> Index lookup on weakness using <auto\_key0> (against\_id=strength.attacking\_id) (actual time=0.001..0.001 rows=1 loops=35)

* + Explanation and analysis report:
    - Since the table is fairly simple and two indexes are already given to the composite primary key, the only option to add a new index is to use effectiveness. This trial turns out to be less effective when it comes to aggregation options. The time taken to aggregate all results is 0.2 seconds longer than the previous one. Therefore, adding index does not help to reduce query time in this case.

5. Table user:

CREATE TABLE user (

user\_Id VARCHAR(8) PRIMARY KEY,

userName VARCHAR(16),

userPassword VARCHAR(32),

userEmail VARCHAR(64),

privacy boolean);

6. Table publicInfo:

CREATE TABLE publicInfo(

user\_ID VARCHAR(8),

trainingID VARCHAR(8),

share boolean,

FOREIGN KEY(user\_ID) REFERENCES User(userID),

FOREIGN KEY(trainingID) REFERENCES Training(trainingID));

7. Table friend\_table:

CREATE TABLE friend\_table(

user\_id VARCHAR(8),

userName VARCHAR(16),

userEmail VARCHAR(32),

friend1\_id VARCHAR(8),

friend2\_id VARCHAR(8),

friend3\_id VARCHAR(8),

friend4\_id VARCHAR(8),

friend5\_id VARCHAR(8));

Most tables were created during Stage 2 and 3. We only created a new table friend\_table afterward to satisfy the requirements of Stage 2 (at least five tables).

We decided to use the default index of the primary key for most tables since we found that almost all of our queries involve some primary keys of tables.

1. **Data Source and Methods of Collecting**

We found a dataset containing all parameters of pokemons on Github. However, it only has rows less than one hundred. In order to solve this weakness, we decided to use Python Jupyter Notebook with Package pymysql to randomly generate data for twenty thousand rows to insert into the mysql server on GCP.

1. **Application Design and Features**

We design this application by involving databases for users to train their pokemons and record their decisions. Users can firstly adopt pokemons from template pokemon database with default values. Then they could decide to increase or decrease some values of specific attributes. We also involve some restrictions to help users win others by requiring their trainings to have values of attributes that are higher than the average of corresponding attributes. Moreover, we introduce a technology to allow users to update these values without automatically refreshing the webpage so that users will not waste time to wait for the responses from our server.

1. **Advanced Database Program**

**Code of the stored procedure:**

DELIMITER //

CREATE PROCEDURE myupdate (IN userID VARCHAR(32))

BEGIN

Declare done int default 0;

-- "capitalized: training value" "lower-cased: original pokemon value"

Declare ntraining\_ID VARCHAR(8);

Declare npokemonID VARCHAR(8);

Declare ntypeID VARCHAR(32);

Declare nhp INT;

Declare nattack INT;

Declare nattack\_T INT;

Declare ndefense INT;

Declare ndefense\_T INT;

Declare nspecialattack INT;

Declare nspecialattack\_T INT;

Declare nspecialdefense INT;

Declare nspecialdefense\_T INT;

Declare nspeed INT;

Declare nheight INT;

Declare nweight INT;

Declare ngender INT;

Declare ngeneration INT;

Declare nlegendary boolean;

Declare nhp\_avg INT;

Declare changedCur CURSOR for (select training\_ID, hp\_T, attack\_T, specialAttack\_T, attack, specialAttack, defense\_T, specialDefense\_T, defense, specialDefense

from Pokemon.training join Pokemon.pokemon on Pokemon.training.pokemon\_ID = Pokemon.pokemon.pokemon\_ID

where (attack\_T > 100 and specialAttack\_T > 100) and (attack < 100 and specialAttack < 100)

UNION

select training\_ID, hp\_T, attack\_T, specialAttack\_T, attack, specialAttack, defense\_T, specialDefense\_T, defense, specialDefense

from Pokemon.training join Pokemon.pokemon on Pokemon.training.pokemon\_ID = Pokemon.pokemon.pokemon\_ID

where (defense\_T > 140 and specialDefense\_T > 140) and (defense < 140 and specialDefense < 140) );

Declare attributeCur CURSOR for (SELECT training\_ID, hp\_T, attack\_T, specialAttack\_T, defense\_T, specialDefense\_T,speed,legendary

FROM Pokemon.pokemon join Pokemon.training

on Pokemon.pokemon.pokemon\_ID = Pokemon.training.pokemon\_ID

Where legendary = 1 and specialAttack\_T > attack\_T and specialDefense\_T > defense and hp\_T > 100

union

SELECT training\_ID, hp\_T, attack\_T, specialAttack\_T, defense\_T, specialDefense\_T,speed,legendary

FROM Pokemon.pokemon join (

SELECT pokemon\_ID, training\_ID, hp\_T, attack\_T, specialAttack\_T, defense\_T, specialDefense\_T

FROM Pokemon.training

WHERE (specialAttack\_T > 150 or specialDefense\_T > 150) and hp\_T > 100

) as temp on Pokemon.pokemon.pokemon\_ID = temp.pokemon\_ID

Where legendary = 0);

Declare hp\_TCur CURSOR for (SELECT pokemon\_ID, avg(hp) AS average\_hp

FROM

(SELECT hp, hp\_T, training.pokemon\_ID

FROM Pokemon.pokemon JOIN training ON pokemon.pokemon\_ID = training.pokemon\_ID

WHERE hp > hp\_T

) AS temp

GROUP BY pokemon\_ID);

Declare Continue handler for NOT FOUND set done = 1;

-- yaohuiw2

Open changedCur;

Repeat

Fetch changedCur into ntraining\_ID, nhp, nattack\_T, nspecialattack\_T, nattack, nspecialattack, ndefense\_T, nspecialdefense\_T, ndefense, nspecialdefense;

IF (nattack\_T > 100 and nspecialattack\_T > 100) and (nattack < 100 and nspecialattack < 100) THEN

Set nhp = 1, ndefense\_T = 1, nspecialdefense\_T = 1, ndefense = 1, nspecialdefense = 1;

END IF;

IF (ndefense\_T > 140 and nspecialdefense\_T > 140) and (ndefense < 140 and nspecialdefense < 140) THEN

Set nhp = 1, nattack\_T = 1, nspecialattack\_T = 1, nattack = 1, nspecialattack = 1;

END IF;

update Pokemon.training

set hp\_T = nhp, attack\_T = nattack, specialAttack\_T = nspecialattack\_T, defense\_T = ndefense, specialDefense\_T = nspecialdefense\_T

where training\_ID = ntraining\_ID;

UNTIL done

END Repeat;

close changedCur;

-- fyc

SET done = 0;

Open attributeCur;

Repeat

Fetch attributeCur into ntraining\_ID, nhp, nattack, nspecialattack\_T, ndefense, nspecialdefense\_T, nspeed, nlegendary;

SET @AverageSpecialAttack = (select distinct temp.avgst

from Pokemon.training join

(SELECT avg(specialAttack\_T) as avgst

FROM Pokemon.pokemon join Pokemon.training

on Pokemon.pokemon.pokemon\_ID = Pokemon.training.pokemon\_ID) as temp);

SET @AverageSpecialDefense = (select distinct temp.avgsd

from Pokemon.training join

(SELECT avg(specialDefense\_T) as avgsd

FROM Pokemon.pokemon join Pokemon.training

on Pokemon.pokemon.pokemon\_ID = Pokemon.training.pokemon\_ID) as temp);

IF @AverageSpecialAttack >= nspecialattack\_T AND @AverageSpecialDefense >= nspecialdefense\_T THEN

Set nhp = 1, nattack = 1, ndefense = 1, nspeed = 1;

END IF;

UPDATE Pokemon.training

set hp\_T = nhp, attack\_T = nattack, defense\_T = ndefense, speed\_T = nspeed

where training\_ID = ntraining\_ID;

UNTIL done

END Repeat;

close attributeCur;

-- yangd4

SET done = 0;

Open hp\_TCur;

Repeat

Fetch hp\_TCur into npokemonID, nhp\_avg;

SET @user\_hp = (select avg(hp\_T) From training t where t.training\_id in (SELECT training\_id from public\_info where public\_info.user\_id = userID));

IF @user\_hp >= nhp\_avg THEN

Set @user\_hp = @user\_hp - 1;

ELSE

Set @user\_hp = @user\_hp + 1;

END IF;

Update Pokemon.pokemon

SET hp = @user\_hp

where pokemon\_id = npokemonID;

UNTIL done

END Repeat;

close hp\_TCur;

Select training\_id, pokemon\_id, hp\_T, attack\_T, defense\_T, specialAttack\_T, specialDefense\_T, speed\_T, height\_T, weight\_T

From Pokemon.user NATURAL JOIN Pokemon.public\_info NATURAL JOIN Pokemon.training

WHERE user\_id = userID

Order by training\_id ASC;

END;

**Code of the Trigger:**

use Pokemon;

DELIMITER //

CREATE TRIGGER pokemon\_trigger

Before INSERT ON training

FOR EACH ROW

BEGIN

SET @average\_hp\_T = (SELECT avg(hp\_T) FROM training);

SET @average\_attack\_T = (SELECT avg(attack\_T) FROM training);

SET @average\_defense\_T = (SELECT avg(defense\_T) FROM training);

SET @average\_specialAttack\_T = (SELECT avg(specialAttack\_T) FROM training);

SET @average\_specialDefense\_T = (SELECT avg(specialDefense\_T) FROM training);

SET @average\_speed\_T = (SELECT avg(speed\_T) FROM training);

SET @average\_height\_T = (SELECT avg(height\_T) FROM training);

SET @average\_weight\_T = (SELECT avg(weight\_T) FROM training);

IF new.hp\_T < 0 THEN

SET new.hp\_T = @average\_hp\_T;

END IF;

IF new.attack\_T < 0 THEN

SET new.attack\_T = @average\_attack\_T;

END IF;

IF new.defense\_T < 0 THEN

SET new.defense\_T = @average\_defense\_T;

END IF;

IF new.specialAttack\_T < 0 THEN

SET new.specialAttack\_T = @average\_specialAttack\_T;

END IF;

IF new.specialDefense\_T < 0 THEN

SET new.specialDefense\_T = @average\_specialDefense\_T;

END IF;

IF new.speed\_T < 0 THEN

SET new.speed\_T = @average\_speed\_T;

END IF;

IF new.height\_T < 0 THEN

SET new.height\_T = @average\_height\_T;

END IF;

IF new.weight\_T < 0 THEN

SET new.weight\_T = @average\_weight\_T;

END IF;

IF new.pokemon\_id not in (SELECT pokemon\_id FROM pokemon) THEN

INSERT INTO pokemon

VALUES (new.pokemon\_id,new.hp\_T,new.attack\_T,new.defense\_T,new.specialAttack\_T,new.specialDefense\_T,

new.speed\_T,new.height\_T,new.weight\_T,0,1,0);

END IF;

END;

We choose to use a stored procedure and trigger to implement our program with several reasons.

First, this progrm will not meet problems related to concurrent read and write to one certain object. Each user is only allowed to query his or her own trainings, and do some updates.

Second, we use procedure to help users check whether they will win the most of other users based on our database. Additionally, it will give users the feedback by reducing a subset of values of attributes for their tranings.

In addition to a stored procedure, a trigger can help us check whether the user gives us some extreme values such as negative numbers and help us correct these values by replacing them with averages values of corresponding attributes.

1. **Dataflow**
2. User login/register: Create user / Read data from user table
3. Enter home page: load data from pokemon table
   1. At this stage, users can navigate through the website using a navigation bar.
   2. Or click the ADOPT button: user adopt a pokemon; create a new training with a private record of public info; specify the relationship between training and types.
4. Navigate to MY HUB: load data from the training table where user\_id = current user.
   1. Click any table cell: update value of corresponding training attribute
   2. Click toggle: update share\_info attribute
5. Navigate to Training: load data from the public\_info table with data from the training table.
6. Navigate to Statistics table: load data that is associated with advanced queries. This page read data from multiple sources.
7. **Challenge**

We encounter problems when deploying the application to google cloud. We use AJAX to communicate frontend with backend. However, due to potential security issues, we cannot make a successful AJAX call to the server. After searching for relevant information, we figure out that the problem occurs because we set the Allow-Cross-Origin-Header incorrectly. Because most of us are not familiar with the async calls between frontend and backend. solving this issue costs us almost 6 hours.

Another challenge we met is to connect local machines to the Google Cloud SQL. Since all of our group members are in China now, we met some problems regarding connections with Google servers. We firstly decided to use private keys to solve the problem that we needed to trust our dynamic IPv4 addresses whenever we connected with mysql server on GCP. However, it did not solve the connection issue for us. Using a VPN and then connecting with the cloud server was viable. In order to let our local web app connect the Mysql server on GCP, we used the ssh command “ssh -L <mapped local port>:<GCP mysql server IP>:<GCP mysql server port> <GCP username>@<GCP server IP> -N -i <local private key address>” to mapped this specific local port as the port on mysql server of GCP. While keeping this process running on one terminal, we can use 127.0.0.1 and this mapped local port to connect the mysql server on GCP.

1. **Ongoing Modification**

We would like to add images of pokemons in the next update. This would help us to create a more vivid website.

1. **Teamwork Assignment**

bingjig2: dealt with most of the front-end development and created table Type relation; developed backend CRUD and advanced query related to table type-relation. Taught group members how to use Quasar(front-end development stage) and Insomnia(app for testing back-end).

yangd4: created table Pokemon; developed backend CRUD and advanced query related to table type-relation. Contributed to the development of all backend codes(debugging and ssh connection issues); Contributed a lot to the setup of procedure in the final demo.

yf9: created table friends; developed backend CRUD and advanced query related to this table. Applied the trigger in the database on the final stage. Applied part of the procedure related to one of the queries inside procedure.

yaohuiw2: created table training; developed backend CRUD and advanced query to table training. Contributed to ideas behind the final applied trigger and procedure. Communication between group members and workload divisions; Final demo presentations