# Databases project

# *PokéSallianWorld 2022-2023 – Stage 3*


**List of members (name and email):**

| | |
|---|---|
| Valeria Ezquerra Rodriguez | valeria.ezquerra@students.salle.url.edu |
| Eugènia Pacheco i Ferrando | eugenia.pacheco@students.salle.url.edu |
| Sebastián Felix Gorga | sebastianfelix.gorga@students.salle.url.edu |
| Marc Soto José | m.soto@students.salle.url.edu |

**Date of finalization:** 28/05/2023

# Index

# 1 Introduction

In this last phase of the project we will validate the correct functioning of our database by performing several queries to retrieve relevant information about the game. As we demonstrated on the previous stage, we managed to import all the given data and populate our tables. Now it's time to manage them by taking specific information from particular tables while trying to optimize our queries as much as possible to ensure no unnecessary processing power is being used.

Additionally, we will automatize several procedures by creating *triggers* to guarantee the consistency of our data. That is, whenever some changes of data on a single table affect others, we will first validate they are in the correct format and make sense in the context of the table. We will also use them to detect suspicious activity and document it on a new ***Warnings*** table.

Similarly to previous stages, we have decided to split the work in modules (one for each team member). Once all the queries and triggers of each module are done and have been checked by everyone in the team, we will proceed to do the *cross queries.* These are the queries that require gathering information about tables belonging to different modules.

## 2 Entity-relationship model update

The only change we had to do was related to the ***Cause_Or_Recieve*** table. On our database population we took the different moves provided in the ***Pokemon_Instances.csv*** file (*move1*, *move2*, *move3*, and *move4*) and we created new relationships using this table, but completely ignoring the number of this move (as we thought it wasn't relevant). However, we have now realized that it can be important to know which move is which, as in one of the triggers we have we must access the move that occupies the same slot as a given one. Therefore, we added a new attribute *slot* to this table and now when inserting the column *move1* onto our ***Cause_Or_Recieve*** table, we also insert its *slot* as a 1 (same for the other three moves).

COLOR CODE

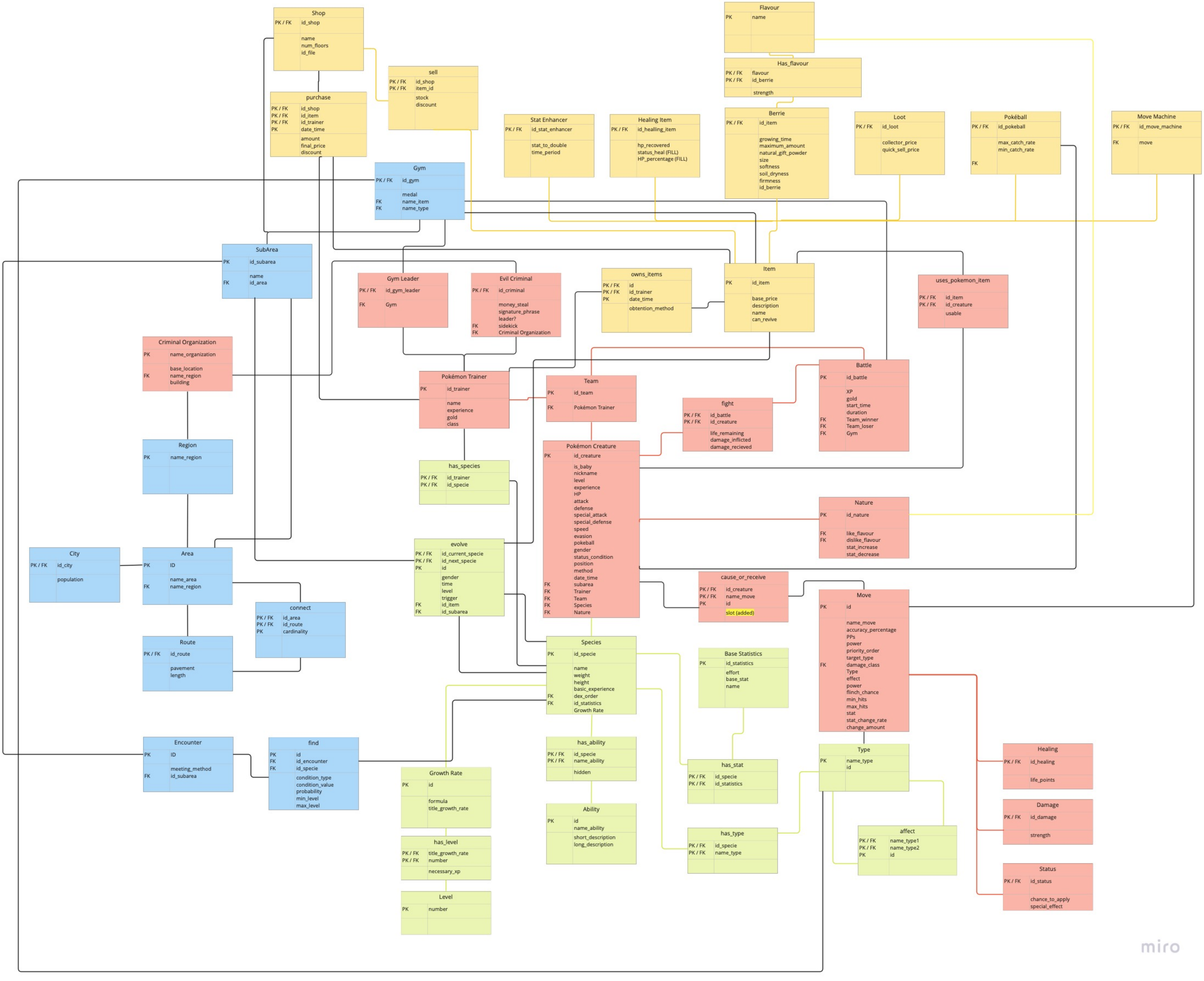
GREEN - 1st block
RED - 2nd block
YELLOW - 3rd block
BLUE - 4th block

BLACK - relationships between entities of different blocks

Take place
Subarea
ID
name
Has
Happen
Gym
medal
Leads
Gym leader
Shop
name
num_floors
id_file
sell
date_time
stock
discount
gift
Flavour
name
strength
Has
Berries
growing_time
maximum_amount
natural_gift_powder
size
soil_dryness
softness
firmness
id_berrie
Stat Enhancer
stat_to_double
time_period
Healing Item
hp_recovered
status_heal
hp_percentage
Loot
collector_price
quick_sell_price
Pokéball
max_catch_rate
min_catch_rate
Move Machine
Items
ID
name
base_price
description
can_revive
purchase
amount
final_price
discount
date_time
owns
obtention_method
date_time
use
usable
Population
City
Area
cardinality
connect
Route
length
pavement
Has
Has
Region
name
Operate in
Criminal organization
name
base_location
Belongs to
Evil criminal
money_steal
signature_phrase
leader?
Sidekick
Pokémon trainer
ID
name
experience
gold
Owns
found
Team
ID
belong
Pokémon creature
ID
is_baby
nickname
level
experience
effort
HP
attack
defense
Sp. Attack
Sp. Defense
speed
evasion
remaining_health
status_condition
position
method
time
date
loser
winner
Battle
ID
XP
gold
start_time
duration
winner
loser
Fight
life_remaining
damage_inflicted
damage_recieved
Catch
like
dislike
Teach
Has
Nature
ID
stat_increase
stat_decrease
name
Cause/Recieve
slot (added)
Belongs to
Species
ID
name
weight
height
basic_experience
dex_order
has
Encounter
ID
meeting method
Find
id
condition_type
condition_value
probability
min_level
max_level
Evolve
Gender
Time
Level
Trigger
Has
Base Statistics
ID
effort
name
base_stat
Has
hidden
Ability
id
name
short_description
long_description
Move
id
name
accuracy_percentage
PPs
power
priority_order
target_type
damage_class
Has
Type
name
id
Affect
multiplier
id
Healing
life_points
Damage
strength
Status
chance_to_apply
special_effect
Level
number
necessary xp
Has
Growth rate
id
title
formula
has

# 3 Relational model update

As we mentioned in the previous section, we had to add a new attribute *slot* to the ***Cause_Or_Recieve*** table. We defined this one as a simple attribute, without declaring it as a FK or PK.


Shop
PK / FK id_shop
name
num_floors
id_file
sell
PK / FK id_shop
PK / FK item_id
stock
discount
purchase
PK / FK id_shop
PK / FK id_item
PK / FK id_trainer
PK date_time
amount
final_price
discount
Flavour
PK name
Has_flavour
PK / FK flavour
PK / FK id_berrie
strength
Stat Enhancer
PK / FK id_stat_enhancer
stat_to_double
time_period
Healing Item
PK / FK id_healling_item
hp_recovered
status_heal (FILL)
HP_percentage (FILL)
Berrie
PK / FK id_item
growing_time
maximum_amount
natural_gift_powder
size
softness
soil_dryness
firmness
id_berrie
Loot
PK / FK id_loot
collector_price
quick_sell_price
Pokéball
PK / FK id_pokeball
max_catch_rate
min_catch_rate
FK
Move Machine
PK / FK id_move_machine
FK move
Gym
PK / FK id_gym
medal
FK name_item
FK name_type
SubArea
PK id_subarea
name
FK id_area
Gym Leader
PK / FK id_gym_leader
FK Gym
Evil Criminal
PK / FK id_criminal
money_steal
signature_phrase
leader?
FK sidekick
FK Criminal Organization
owns_items
PK / FK id
PK / FK id_trainer
PK date_time
obtention_method
Item
PK id_item
base_price
description
name
can_revive
uses_pokemon_item
PK / FK id_item
PK / FK id_creature
usable
Criminal Organization
PK name_organization
base_location
FK name_region
building
Pokémon Trainer
PK id_trainer
name
experience
gold
class
Team
PK id_team
FK Pokémon Trainer
fight
PK / FK id_battle
PK / FK id_creature
life_remaining
damage_inflicted
damage_recieved
Battle
PK id_battle
XP
gold
start_time
duration
FK Team_winner
FK Team_loser
FK Gym
Region
PK name_region
has_species
PK / FK id_trainer
PK / FK id_specie
Pokémon Creature
PK id_creature
is_baby
nickname
level
experience
HP
attack
defense
special_attack
special_defense
speed
evasion
pokeball
gender
status_condition
position
method
date_time
FK subarea
FK Trainer
FK Team
FK Species
FK Nature
Nature
PK id_nature
FK like_flavour
FK dislike_flavour
stat_increase
stat_decrease
City
PK / FK id_city
population
Area
PK ID
name_area
FK name_region
evolve
PK / FK id_current_specie
PK / FK id_next_specie
PK id
gender
time
level
trigger
FK id_item
FK id_subarea
cause_or_receive
PK / FK id_creature
PK / FK name_move
PK id
slot (added)
connect
PK / FK id_area
PK / FK id_route
PK cardinality
Route
PK / FK id_route
pavement
length
Species
PK id_specie
name
weight
height
basic_experience
FK dex_order
FK id_statistics
Growth Rate
Base Statistics
PK id_statistics
effort
base_stat
name
Move
PK id
name_move
accuracy_percentage
PPs
power
priority_order
target_type
FK damage_class
Type
effect
power
flinch_chance
min_hits
max_hits
stat
stat_change_rate
change_amount
Encounter
PK ID
meeting_method
FK id_subarea
find
PK id
FK id_encounter
FK id_specie
condition_type
condition_value
probability
min_level
max_level
has_ability
PK / FK id_specie
PK / FK name_ability
hidden
has_stat
PK / FK id_specie
PK / FK id_statistics
Type
PK name_type
id
Healing
PK / FK id_healing
life_points
Damage
PK / FK id_damage
strength
Growth Rate
PK id
formula
title_growth_rate
Ability
PK id
name_ability
short_description
long_description
has_type
PK / FK id_specie
PK / FK name_type
affect
PK / FK name_type1
PK / FK name_type2
PK id
Status
PK / FK id_status
chance_to_apply
special_effect
has_level
PK / FK title_growth_rate
PK / FK number
necessary_xp
Level
PK number
miro

# 4 Physical model update

When trying to link the ***Criminal_Organization*** and ***Region*** tables in one of the queries we realized that the attribute *region* we had in this first table, hadn't been declared as a foreign key to the second. This was specified on our relational model diagram, but we had forgotten to add it to our physical model. We have fixed this little issue and now the query works as expected.

# 5 Pokémon are unique warriors

## 5.1 Query 1

### 5.1.1 Solution

```
SELECT DISTINCT s.name, bs.base_stat AS defense, bs2.base_stat AS
special_defense, s.weight
FROM species AS s
JOIN has_stat AS hs1 ON s.id = hs1.id_species
JOIN base_statistics AS bs ON hs1.id_statistics = bs.id
JOIN has_stat AS hs2 ON s.id = hs2.id_species
JOIN base_statistics AS bs2 ON hs2.id_statistics = bs2.id
JOIN has_ability AS ha ON s.id = ha.id_species
JOIN ability a ON ha.id_ability = a.id
WHERE bs.name = 'defense' AND bs2.name = 'special defense' AND a.name = 'sturdy'
ORDER BY s.weight DESC
LIMIT 10;
```

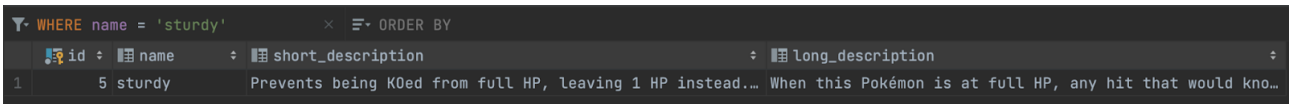
| | name | defense | special_defense | weight |
|---|---|---|---|---|
| 1 | steelix | 200 | 65 | 4000 |
| 2 | aggron | 180 | 60 | 3600 |
| 3 | probopass | 145 | 150 | 3400 |
| 4 | golem | 130 | 65 | 3000 |
| 5 | regirock | 200 | 100 | 2300 |
| 6 | onix | 160 | 45 | 2100 |
| 7 | magnezone | 115 | 90 | 1800 |
| 8 | bastiodon | 168 | 138 | 1495 |
| 9 | forretress | 140 | 60 | 1258 |
| 10 | donphan | 120 | 60 | 1200 |

### 5.1.2 Explanation

As we want to output the *name*, the *defense* and *special defense* values together with the *weight*, we must select all of them. In order to get the desired data, we *JOIN* species with ***has_stat*** and ***has_stat*** with ***base_statistics*** twice since each species has 4 base_statistics but we just want 2 of them (*defense* and *special_defense*). After this, the *WHERE* clause is used to indicate that we want only those *defense* and *special_defense* values as long as the ability name is sturdy. Finally, we order the result by descendent weight and we limit to 10 as the statement says.

### 5.1.3 Query validation

In order to check this query, we get the id of the 'sturdy' ability and we filter by this ID in the ***has_ability*** table. This way, we obtain all the species IDs that have that particular ability. Then, by doing another query, we get all the species names ordered by descendent weight.




```
SELECT ha.id_species, s.name
FROM species AS s
```

```
JOIN has_ability AS ha ON ha.id_species = s.id
WHERE id_ability = 5
ORDER BY s.weight DESC;
```

WHERE id_ability = 5 × ORDER BY

| | id_species | id_ability | hidden |
|---|---|---|---|
| 1 | 74 | 5 | false |
| 2 | 75 | 5 | false |
| 3 | 76 | 5 | false |
| 4 | 81 | 5 | false |
| 5 | 82 | 5 | false |
| 6 | 95 | 5 | false |
| 7 | 185 | 5 | false |
| 8 | 204 | 5 | false |
| 9 | 205 | 5 | false |
| 10 | 208 | 5 | false |
| 11 | 213 | 5 | false |
| 12 | 227 | 5 | false |
| 13 | 232 | 5 | false |
| 14 | 299 | 5 | false |
| 15 | 304 | 5 | false |
| 16 | 305 | 5 | false |
| 17 | 306 | 5 | false |
| 18 | 369 | 5 | true |
| 19 | 377 | 5 | true |
| 20 | 410 | 5 | false |
| 21 | 411 | 5 | false |
| 22 | 438 | 5 | false |
| 23 | 462 | 5 | false |
| 24 | 476 | 5 | false |

| | id_species | name |
|---|---|---|
| 1 | 208 | steelix |
| 2 | 306 | aggron |
| 3 | 476 | probopass |
| 4 | 76 | golem |
| 5 | 377 | regirock |
| 6 | 95 | onix |
| 7 | 462 | magnezone |
| 8 | 411 | bastiodon |
| 9 | 205 | forretress |
| 10 | 232 | donphan |
| 11 | 305 | lairon |
| 12 | 75 | graveler |
| 13 | 299 | nosepass |
| 14 | 304 | aron |
| 15 | 82 | magneton |
| 16 | 410 | shieldon |
| 17 | 227 | skarmory |
| 18 | 185 | sudowoodo |
| 19 | 369 | relicanth |
| 20 | 213 | shuckle |
| 21 | 74 | geodude |
| 22 | 438 | bonsly |
| 23 | 204 | pineco |
| 24 | 81 | magnemite |

As we can see, the heaviest species with ‘sturdy’ ability match with our output.

Now that we have the heaviest species, we can check if the *defense* and *special_defense* values are correct for ‘steelix’ (for example). If we filter by its ID in the ***has_stats*** table, we get that the abilities it has are the following ones:

WHERE id_species = 208 × ORDER

| | id_species | id_statistics |
|---|---|---|
| 1 | 208 | 1243 |
| 2 | 208 | 1244 |
| 3 | 208 | 1245 |
| 4 | 208 | 1246 |
| 5 | 208 | 1247 |
| 6 | 208 | 1248 |

Now, if we go to the table ***base_statistics***, we can see that the values match with what we got.

| | id | effort | base_stat | name |
|---|---|---|---|---|
| 1 | 1245 | 2 | 200 | defense |

| | id | effort | base_stat | name |
|---|---|---|---|---|
| 1 | 1247 | 0 | 65 | special defense |

## *5.2 Query 2*

### *5.2.1 Solution*

```
SELECT AVG(CASE WHEN (bs.name = 'attack' OR bs.name = 'special
attack' ) THEN bs.base_stat END) AS average_damage_bs,
```

```
        MAX(CASE WHEN (bs.name = 'attack' OR bs.name = 'special
attack' ) THEN bs.base_stat END) AS max_damage_bs,
        MIN(CASE WHEN (bs.name = 'attack' OR bs.name = 'special
attack' ) THEN bs.base_stat END) AS min_damage_bs
FROM species AS s
JOIN has_stat ON s.id = has_stat.id_species
JOIN base_statistics bs ON has_stat.id_statistics = bs.id
WHERE s.id IN (SELECT id_species
               FROM has_type
               WHERE type_name = 'fire');
```

| | average_damage_bs | max_damage_bs | min_damage_bs |
|---|---|---|---|
| 1 | 84.8636363636363636 | 130 | 40 |

### 5.2.2 Explanation

In this query we want to get the *average*, *maximum* and *minimum damage base_statistic* of the species of *type* 'fire'. Note that since we want to consider only those base stats related to damage, we must do a CASE when selecting the information. Then, we *JOIN* ***species***, ***has_stat*** and ***base_statistics*** tables. In the *WHERE* clause, we state that we only want those species whose ID is included in ***has_type*** table and that the *type* is fire. To specify this, we use a subquery since we just want to take into account the species that have type 'fire'.

### 5.2.3 Query validation

We can check that the MAX and MIN values are correct by doing the following query:

```
SELECT s.name, ht.id_species, MAX(CASE WHEN (bs.name = 'attack' OR
bs.name = 'special attack' ) THEN bs.base_stat END),
         MIN(CASE WHEN (bs.name = 'attack' OR bs.name = 'special
attack' ) THEN bs.base_stat END)
FROM species AS s
JOIN has_type AS ht ON s.id = ht.id_species
JOIN has_stat hs on s.id = hs.id_species
JOIN base_statistics bs on hs.id_statistics = bs.id
WHERE ht.type_name = 'fire'
GROUP BY(s.name, ht.id_species);
```

If we order the MAX stat from highest to lowest and the MIN stat from lowest to highest, we get this:

| | name | id_species | max | min |
|---|---|---|---|---|
| 1 | flareon | 136 | 130 | 95 |
| 2 | ho-oh | 250 | 130 | 110 |
| 3 | heatran | 485 | 130 | 90 |
| 4 | magmortar | 467 | 125 | 95 |

| | name | id_species | max | min |
|---|---|---|---|---|
| 1 | slugma | 218 | 70 | 40 |
| 2 | vulpix | 37 | 50 | 41 |
| 3 | magcargo | 219 | 90 | 50 |
| 4 | charmander | 4 | 60 | 52 |

As the screenshots show, the min base stat is 40 and the max base stat is 130.

As for the AVG value, by computing the following query we can see that it coincides:

```
SELECT AVG(CASE WHEN (bs.name = 'attack' OR bs.name = 'special
attack' ) THEN bs.base_stat END)
FROM species AS s
JOIN has_stat hs on s.id = hs.id_species
JOIN base_statistics bs on hs.id_statistics = bs.id
WHERE s.id IN (SELECT id_species
               FROM has_type
               WHERE type_name = 'fire');
```

| | avg |
|---|---|
| 1 | 84.8636363636363636 |

## 5.3 Query 3

### 5.3.1 Solution

```
SELECT a.name AS ability_name, a.short_description, s.name AS species_name
FROM species AS s JOIN has_ability AS ha ON s.id = ha.id_species
JOIN ability AS a ON a.id = ha.id_ability
WHERE s.basic_experience > (SELECT AVG(basic_experience)
                            FROM species)
ORDER BY (a.name, s.name) DESC;
```

1-500 of 501+ CSV

| | ability_name | short_description | species_name |
|---|---|---|---|
| 1 | wonder skin | Lowers incoming non-damaging moves' base accuracy to exactly 50%. | venomoth |
| 2 | white smoke | Prevents stats from being lowered by other Pokémon. | torkoal |
| 3 | weak armor | Raises Speed and lowers Defense by one stage each upon being hit by a physical move. | skarmory |
| 4 | weak armor | Raises Speed and lowers Defense by one stage each upon being hit by a physical move. | omastar |
| 5 | weak armor | Raises Speed and lowers Defense by one stage each upon being hit by a physical move. | magcargo |
| 6 | weak armor | Raises Speed and lowers Defense by one stage each upon being hit by a physical move. | kabutops |
| 7 | water veil | Prevents burns. | wailord |
| 8 | water veil | Prevents burns. | seaking |
| 9 | water veil | Prevents burns. | mantine |
| 10 | water veil | Prevents burns. | lumineon |

As you can see in the picture, the query returns 595 rows (too many to fit in a screenshot).

### 5.3.2 Explanation

In here we were asked to provide the *name* of the ability, its corresponding *description* and the *species name* of those species whose starting *base experience* is greater than the *average base experience* of all the species. To obtain this, we need to *JOIN* ***species***, ***has_ability*** and ***ability*** tables. In the *WHERE* clause we specify that we want to display the information of only those species whose starting *base_experience* is greater than the

average of the *basic_experience* of all the species. Since the AVG function gets the *basic_experience* of all the species, a subquery is needed.

### 5.3.3 Query validation

In order to check that the query returns what is expected, we can get the average of the *basic_experience* of all the species. Once we have that, we can create another query that shows the species with a basic experience greater than this average value. If we start checking name by name we see that it matches with our initial query.

```
SELECT AVG(basic_experience) AS average
FROM species;
```

| average |
| --- |
| 145.9553752535496957 |

```
SELECT s.name, s.basic_experience
FROM species AS s
WHERE s.basic_experience > 145.95
ORDER BY s.basic_experience DESC;
```

| | name | basic_experience |
| --- | --- | --- |
| 1 | blissey | 635 |
| 2 | chansey | 395 |
| 3 | rayquaza | 340 |
| 4 | ho-oh | 340 |
| 5 | giratina-altered | 340 |
| 6 | palkia | 340 |
| 7 | dialga | 340 |
| 8 | lugia | 340 |
| 9 | mewtwo | 340 |
| 10 | kyogre | 335 |

## 5.4 Query 4

### 5.4.1 Solution

```
SELECT stat_name, max_base_stat, evolved_species
FROM (
   SELECT   evolved_species.name   AS   evolved_species,   bs.name   AS   stat_name,
bs.base_stat AS max_base_stat,
          ROW_NUMBER() OVER (PARTITION BY bs.name ORDER BY bs.base_stat DESC,
evolved_species.id) AS rn
 FROM Species AS baby_species
 JOIN evolve ON baby_species.id = evolve.id_current_species
 JOIN Species AS evolved_species ON evolve.id_next_species = evolved_species.id
 JOIN Pokemon_Creature ON baby_species.id = Pokemon_Creature.species
 JOIN has_stat ON evolved_species.id = has_stat.id_species
 JOIN Base_Statistics AS bs ON has_stat.id_statistics = bs.id
 WHERE Pokemon_Creature.is_baby = TRUE
) AS subquery
WHERE rn = 1
```

```
ORDER BY max_base_stat DESC, stat_name;
```

| | stat_name | max_base_stat | evolved_species |
|---|---|---|---|
| 1 | hp | 250 | chansey |
| 2 | special defense | 140 | mantine |
| 3 | attack | 120 | hitmonlee |
| 4 | defense | 115 | sudowoodo |
| 5 | special attack | 115 | jynx |
| 6 | speed | 105 | electabuzz |

### *5.4.2 Explanation*

This query obtains the maximum base stat of each statistics among the evolved species of baby pokemon. It uses a subquery to *JOIN* first the tables ***species*** with ***evolve*** and again with ***species*** in order to know the evolution from a baby, and then, with ***pokemon creature***, ***has_stat*** and ***base_statistics*** to know the *maximum base stat*. Note that we need the ***pokemon_creature*** table because we stored if it is a baby or not there.

The subquery also uses row number ( ) that calculates for each combination of statistics and evolved species. This number, which we call rn, is determined by ordering the *basic_statistics* in descending order and breaking ties according to the evolved species ID.

Then, PARTITION BY is used to ensure that the row number is calculated for each base_satatistics.

The main query filters the subquery results to only include rows with rn = 1(meaning the row with the highest base_stat value for each base_statistics). Finally, the output is ordered by base_stat value in descending order.

### *5.4.3 Query validation*

If we check all the max base statististics of each of the pokemons, we can see if some are repeated and should appear twice.

| evolved_species | stat_name | max_base_stat | rn |
|---|---|---|---|
| hitmonlee | attack | 120 | 1 |
| hitmonlee | attack | 120 | 34 |
| snorlax | attack | 110 | 35 |
| sudowoodo | defense | 115 | 1 |
| sudowoodo | defense | 115 | 30 |
| hitmontop | defense | 95 | 31 |
| 1105 chansey | hp | 250 | 1 |
| chansey | hp | 250 | 22 |
| wobbuffet | hp | 190 | 23 |

| | | | |
|---|---|---|---|
| jynx | special attack | 115 | 1 |
| jynx | special attack | 115 | 28 |
| lucario | special attack | 115 | 29 |
| mantine | special defense | 140 | 1 |
| mantine | special defense | 140 | 25 |
| mr-mime | special defense | 120 | 26 |
| electabuzz | speed | 105 | 1 |
| electabuzz | speed | 105 | 27 |
| jynx | speed | 95 | 28 |

With this results we can see that the max base statistic of each of them is correct and that the only ones that have the same max stat are jynx and lucario (special attack).

## 5.5 Query 5

### 5.5.1 Solution

```
(SELECT
      'Strength' AS type, attacker.name AS strongest_type, COUNT(*) AS
strength_count
FROM affect AS a
JOIN Type AS attacker ON attacker.name = a.name_type1
WHERE a.multiplier = 'x2'
GROUP BY attacker.name
ORDER BY strength_count DESC
LIMIT 2)

UNION ALL

(SELECT
      'Resistance' AS type, defender.name AS resistant_type, COUNT(*) AS
resistance_count
FROM affect AS a
JOIN Type AS defender ON defender.name = a.name_type2
WHERE a.multiplier = 'x0.5'
GROUP BY defender.name
ORDER BY resistance_count DESC
LIMIT 1);
```

| | type | strongest_type | strength_count |
|---|---|---|---|
| 1 | Strength | ground | 5 |
| 2 | Strength | fighting | 5 |
| 3 | Resistance | steel | 10 |

### 5.5.2 Explanation

This query provides us information about the strongest attacking type and the most resistant defending type. In the first part of the query, we find the attacking type (strongest type) based on the count of moves (strength_count) that have a multiplier of x2 in the ***affect*** table. It groups the results by attacker name and orders them in descending order of strength_count. It is limited to 2 since there are two types that have the maximum strength.

In the second part of the query, it identifies the most resistant defending type (resistant type) based on the count of moves (resistance_count) that have a multiplier of x0.5 in the ***affect*** table. It groups the results by defender name and orders them in descending order of resistance_count. It is just limited to 1 since there is only one type with the most resistance.

A *UNION ALL* is used since we want to combine the results of both queries. It basically adds a new column called 'type' that indicates whether the row represents the strongest type or the most resistant type.

#### *5.5.3 Query validation*

The validation of this query can be done by running the queries separately.

| | strongest_type | strength_count |
|---|---|---|
| 1 | ground | 5 |
| 2 | fighting | 5 |
| 3 | ice | 4 |
| 4 | fire | 4 |
| 5 | rock | 4 |
| 6 | fairy | 3 |
| 7 | water | 3 |
| 8 | flying | 3 |
| 9 | grass | 3 |
| 10 | steel | 3 |
| 11 | bug | 3 |
| 12 | ghost | 2 |
| 13 | electric | 2 |
| 14 | dark | 2 |
| 15 | psychic | 2 |
| 16 | poison | 2 |
| 17 | dragon | 1 |

| | resistant_type | resistance_count |
|---|---|---|
| 1 | steel | 10 |
| 2 | fire | 6 |
| 3 | poison | 5 |
| 4 | dragon | 4 |
| 5 | grass | 4 |
| 6 | water | 4 |
| 7 | rock | 4 |
| 8 | fairy | 3 |
| 9 | flying | 3 |
| 10 | fighting | 3 |
| 11 | electric | 3 |
| 12 | bug | 3 |
| 13 | ghost | 2 |
| 14 | dark | 2 |
| 15 | ground | 2 |
| 16 | psychic | 2 |
| 17 | ice | 1 |

As we can see, there are two strength types with the maximum number of strengths and just one with the maximum resistances.

## *5.6 Query 6*

#### *5.6.1 Solution*

```
SELECT sc1.id AS species_id, sc1.name AS species_name, sc1.basic_experience AS
species_experience,
            sc2.id AS evolved_species_id, sc2.name AS evolved_species_name,
(sc1.basic_experience + sc2.basic_experience) AS evolved_species_experience,
                    sc3.id AS final_evolved_species_id, sc3.name AS
final_evolved_species_name,
        (sc1.basic_experience + sc2.basic_experience + sc3.basic_experience) AS
final_evolved_species_experience
FROM Species AS sc1
```

```
JOIN evolve AS ev1 ON sc1.id = ev1.id_current_species
JOIN Species AS sc2 ON ev1.id_next_species = sc2.id
JOIN evolve AS ev2 ON sc2.id = ev2.id_current_species
JOIN Species AS sc3 ON ev2.id_next_species = sc3.id
WHERE ev1.trigger = 'level up'
 AND ev1.level <> 0
 AND ev2.trigger = 'level up'
 AND ev2.level <> 0
ORDER BY final_evolved_species_experience DESC
LIMIT 1;
```

| | species_id | species_name | species_experience | evolved_species_id | evolved_species_name | evolved_species_experience |
|---|---|---|---|---|---|---|
| 1 | 147 | dratini | 60 | 148 | dragonair | 207 |

| final_evolved_species_id | final_evolved_species_name | final_evolved_species_experience |
|---|---|---|
| 149 | dragonite | 507 |

#### 5.6.2 Explanation

Now we want to get the evolution chain that requires more experience to reach its biggest form. To reach this goal, we select the id of the species, the species name and the species experience. We do this also for the evolved and for the final evolved species. We *JOIN* ***species*** with ***evolve*** multiple times since we want to know the evolution relationships. We link the *species.id* with the *current_species*, the *next_species* with the *second species id*, the *second species id* with the *current species of the second evolution* and finally, the *next_species id of the second evolution* with the *third species id.*

In the WHERE clause, we state that in order to evolve, the trigger in evolution must be ‘level up’ and that the level is different from 0.

#### 5.6.3 Query validation

In this case, we have UPDATED the basic_experience of bulbasaur to 1 million in order to see it we get that as the biggest evolution chain.

```
UPDATE species
SET basic_experience = 10000000
WHERE id = 1;
```

After the modification, we see that we get bulbasur.

| | species_id | species_name | species_experience | evolved_species_id | evolved_species_name | evolved_species_experience |
|---|---|---|---|---|---|---|
| 1 | 1 | bulbasaur | 10000000 | 2 | ivysaur | 10000142 |

| final_evolved_species_id | final_evolved_species_name | final_evolved_species_experience |
|---|---|---|
| 3 | venusaur | 10000405 |

## 5.7 Trigger 1

#### 5.7.1 Solution

```
DROP TABLE IF EXISTS warnings CASCADE;
CREATE TABLE warnings (
   affected_table VARCHAR(30) NOT NULL,
   date DATE NOT NULL,
   error_message VARCHAR(255) NOT NULL,
```

```
   "user" VARCHAR(255) NOT NULL
);

-- Trigger maximum num types x species
DROP FUNCTION IF EXISTS check_max_types CASCADE;
CREATE OR REPLACE FUNCTION check_max_types() RETURNS TRIGGER AS $$
DECLARE
   types_count INT;
   total_entries INT;
BEGIN
   SELECT COUNT(id_species) INTO types_count
   FROM has_type
   WHERE id_species = NEW.id_species;

   total_entries := types_count + 1;

   IF types_count > 2 THEN
       INSERT INTO warnings (affected_table, date, error_message, "user")
         VALUES ('has_type', CURRENT_DATE, 'A ' || (total_entries - 1 )|| 'th
entry has been inserted into the has_type table', current_user);

   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS max_types_trigger ON has_type CASCADE;
CREATE TRIGGER max_types_trigger
AFTER INSERT OR UPDATE ON has_type
FOR EACH ROW
EXECUTE FUNCTION check_max_types();

-- Trigger maximum num abilities x species
DROP FUNCTION IF EXISTS check_max_abilities CASCADE;
CREATE OR REPLACE FUNCTION check_max_abilities() RETURNS TRIGGER AS $$
DECLARE
   abilities_count INT;
   total_entries INT;
BEGIN
   SELECT COUNT(id_ability) INTO abilities_count
   FROM has_ability
   WHERE id_species = NEW.id_species AND hidden = FALSE;

   total_entries := abilities_count + 1;

   IF abilities_count > 2 THEN
       INSERT INTO warnings (affected_table, date, error_message, "user")
        VALUES ('has_ability', CURRENT_DATE, 'A ' || (total_entries - 1) || 'th
entry has been inserted into the has_ability table', current_user);

   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS max_abilities_trigger ON has_ability CASCADE;
CREATE TRIGGER max_abilities_trigger
AFTER INSERT OR UPDATE ON has_ability
FOR EACH ROW
EXECUTE FUNCTION check_max_abilities();
```

### 5.7.2 Explanation

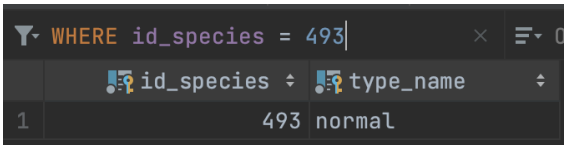
This triggers are basically activated after inserting or updating the tables ***has_ability*** and ***has_type***. Two variables are declared, one that counts the number of abilities or types that a specific species has and the other one to notice the number of entries that have been inserted into the table. If the number of either types or abilities per species is greater than 2, an INSERT INTO the table warnings is done warning that x entries have been inserted into the ***has_ability*** table. In this warnings table, the affected table, the current date of the insertion and the user who did that operation is also registered.

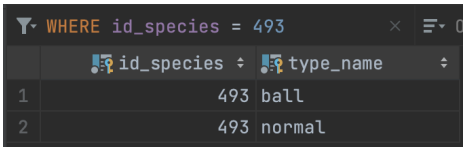
### 5.7.3 Query validation

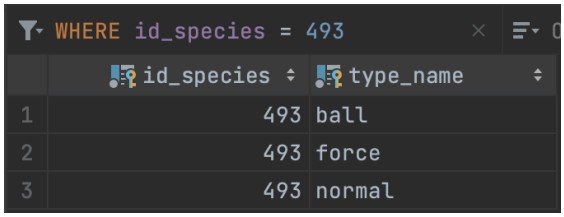
Since there are two different triggers, we must do two different validations, one for the types and one for the abilities.

For the first trigger to activate, we first check how many types a particular species has. In this case, we are going to use the species id 493. We go to the ***has_type*** table and see that it has only one type associated with it. Since the trigger is only triggered when it detects more than two different types associated to the same species, the next step is to insert two more types into the ***has_type*** and ***type*** tables.

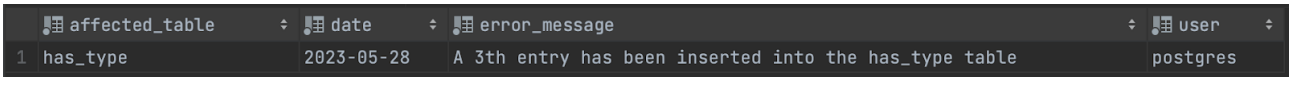
```
INSERT INTO Type (id, name)
VALUES (22, 'ball');
INSERT INTO has_type (id_species, type_name)
VALUES (1, 'ball');
```




After each insertion we run the trigger and check that in the second insertion nothing happens while in the third one, the corresponding error message appears.




We repeat the same process but this time inserting new abilities in the table ***has_ability*** and ***ability***.

## 5.8 Trigger 2

### 5.8.1 Solution

### 5.8.2 Explanation

### 5.8.3 Query validation

We faced some problems when doing this trigger and didn’t have enough time to fix it.

# 6 I'm not a gamer, I'm a Pokémon trainer

## 6.1 Query 1

### 6.1.1 Solution

```
SELECT tr.name AS trainer, pc.id AS firstPokemon
      FROM trainer tr JOIN team t ON t.trainer = tr.id
      JOIN battle b on t.id = b.winner JOIN team glt ON glt.id = b.loser
      JOIN gym_leader gl ON glt.trainer = gl.id
      JOIN pokemon_creature pc ON pc.trainer = tr.id
      GROUP BY tr.id, pc.id
      ORDER BY COUNT(gl.id) DESC, MIN(pc.date_time)
      LIMIT 1;
```

| | trainer | firstpokemon |
|---|---|---|
| 1 | Olinda | 6167 |

### 6.1.2 Explanation

To find the trainer who had defeated the most Gym Leaders in combat we had to select its name from the table ***Trainer***, then join this one with its ***Team*** to see in which battles he had fought. We were able to get only the battles in which they won by linking its team to the *winner* attribute in the ***Battle*** entity. To find only those battles in which the trainer had defeated a Gym Leader, we joined the *loser* attribute of the ***Battle*** entity with a new instance of the ***Team*** entity. Then, we linked this team to the ***Gym_Leader entity*** (we could skip the ***Trainer*** table as the ***Gym_Leader*** has as its *id* a FK to the ***Trainer*** table).

The above mentioned steps return us all the trainers that have defeated a Gym Leader. To get the amount of wins of each trainer, we used a *GROUP BY* clause to join all the rows from the same trainer. To order them by the one that had defeated most, we used an *ORDER BY* clause in which we specified we wanted the output to be from the one with a highest count of defeated Gym Leaders (*COUNT* operation of Gym Leaders) to the one that had the least (*DESC*). We added a *LIMIT* 1 to only get the trainer that had defeated the most.

Lastly, to show the first Pokémon's id of the trainer, we joined the ***Pokemon_Creature*** entity with the ***Trainer*** (adding it on the *GROUP BY* clause) and ordering the output using the *ORDER BY* clause (as a second parameter, giving it less priority) with the Pokémon that had been captured earlier. To get this, we used the *MIN* operation to check which of their Pokémons had the smallest *date_time* attribute.

### 6.1.3 Query validation

It is hard to check that this query works with the given dataset, as if we remove the *LIMIT* of the previous query and show the *COUNT* of defeated Gym Leaders, and the *date_time* of when the Pokémon was captured, we can see how all trainers have defeated a maximum of one Gym Leader. Olinda (output of the query) just happens to have the Pokémon who was captured the earliest of all (second condition of the *ORDER BY* clause). Therefore, she is returned as the output.

```
SELECT tr.name AS trainer, pc.id AS firstPokemon, pc.date_time AS captured,
COUNT( gl.id) AS defeated
```

```
    FROM trainer tr JOIN team t ON t.trainer = tr.id
    JOIN battle b on t.id = b.winner JOIN team glt ON glt.id = b.loser
    JOIN gym_leader gl ON glt.trainer = gl.id
    JOIN pokemon_creature pc ON pc.trainer = tr.id
    GROUP BY tr.id, pc.id ORDER BY COUNT( gl.id) DESC, MIN(pc.date_time);
```

| | trainer | firstpokemon | captured | defeated |
|---|---|---|---|---|
| 1 | Olinda | 6167 | 1996-08-18 19:17:20.000000 | 1 |
| 2 | Garrett | 12469 | 1996-09-08 10:33:00.000000 | 1 |
| 3 | Karol | 829 | 1996-09-14 07:53:04.000000 | 1 |
| 4 | Falkner | 10490 | 1996-09-28 12:27:59.000000 | 1 |
| 5 | Nannie | 6020 | 1996-10-08 09:26:24.000000 | 1 |
| 6 | Rozanne | 6375 | 1996-12-18 11:06:06.000000 | 1 |
| 7 | Garrett | 12460 | 1996-12-21 04:25:18.000000 | 1 |
| 8 | Mac | 6865 | 1996-12-26 10:30:57.000000 | 1 |

To check if our query actually works, we can try manually inserting some data. Let's see we want the trainer *Karol* to defeat another Gym Leader. We will insert the following values to the ***Battle*** table:

```
INSERT INTO battle(id, winner, loser)
VALUES (700, 177, 424);
```

As you can see on the screenshots below, the *id* 177 corresponds to the trainer *Karol* (*id = 30*) and we set this one as the winner of the battle. Then, we can also identify the team 428 as property of the trainer 504 (first gym leader) and we set is as the loser.

Karol's team: (***Trainer*** and ***Team*** tables respectively)

| id | class | name | experience | gold |
|---|---|---|---|---|
| 30 | Spy | Karol | 121717 | -109 |

| id | trainer |
|---|---|
| 177 | 30 |

Gym Leader's team: (***Gym_leader*** and ***Team*** tables respectively)

| id | gym |
|---|---|
| 504 | 527 |

| id | trainer |
|---|---|
| 424 | 504 |

Therefore, if we run the query again we get the following output:

| | trainer | firstpokemon |
|---|---|---|
| 1 | Karol | 829 |

As you can see, now the trainer that appears is Karol, as these one has defeated 2 Gym Leaders in total.

## 6.2 Query 2

### 6.2.1 Solution

```
SELECT pc.nickname AS pokemon, tr.name AS trainer, COUNT(b.id) AS defeats
      FROM pokemon_creature pc JOIN fight f on f.id_creature = pc.id
      JOIN battle b ON f.id_battle = b.id
      JOIN trainer tr ON tr.id = pc.trainer
      WHERE b.winner = pc.team AND f.life_remaining = 0
      GROUP BY pc.id, tr.id
      ORDER BY COUNT(b.id) DESC, MIN(pc.date_time)
      LIMIT 5;
```

| | pokemon | trainer | defeats |
|---|---|---|---|
| 1 | Mac | Norberto | 5 |
| 2 | Rhett | Wiley | 4 |
| 3 | Friday | Mark | 4 |
| 4 | Nibbles | Wiley | 4 |
| 5 | Ginger | Karoline | 4 |

### 6.2.2 Explanation

To get the Pokémon that have been defeated the most times in battles they have won we have started by joining the tables ***Pokemon_Creature*** and ***Battle*** through the N:M relation table ***Fight*** (which holds the statistics of the Pokémon after each battle). With this, we get all the Pokémon and the battles they have fought in. To identify the ones they won we add a *WHERE* clause linking thePokémon's *team* to the *winner* attribute of the ***Battle*** entity.

Also, to get the battles in which they were particularly defeated (even though their team won) we add a second condition to the *WHERE* clause stating that their hp after finishing the battle (*life_reamining* attribute in ***Fight*** table) has to be equal to 0.

To show the ones that have been defeated the most times, we add a *GROUP BY* clause to join rows with the same Pokémon and trainer (to count them) and add a *GROUP BY* condition to order the output from the one with most defeats (using the *COUNT* operator to count the number of battles) to the one that has less (DESC). We also limit the query to only show five Pokémon using the *LIMIT* operator.

Lastly, to give priority to the Pokémon that were captured the earliest, we add a second condition to the *ORDER BY* clause (less priority) that indicates we want to order them by the minimum capture time (using the *MIN* operator and accessing the *date_time* attribute of the ***Pokemon_Creature*** entity). We also have to do a *JOIN* with the ***Trainer*** entity to display the Pokémon trainer's name.

### 6.2.3 Query validation

First, we will check if the results we got match the criteria by displaying the *HP* attribute of the Pokémon, their team *id* (so that we can later check if the count of lost battles makes sense), and the *date_time* when the Pokémon was captured. By executing the following query, we get the result below.

```
SELECT pc.id, pc.nickname AS pokemon, tr.name AS trainer, COUNT(b.id) AS
defeats, pc.hp AS hp, pc.team AS team, pc.date_time AS captured
```

```
      FROM pokemon_creature pc JOIN fight f on f.id_creature = pc.id
      JOIN battle b ON f.id_battle = b.id
      JOIN trainer tr ON tr.id = pc.trainer
      WHERE b.winner = pc.team AND f.life_remaining = 0
      GROUP BY pc.id, tr.id ORDER BY COUNT(b.id) DESC, MIN(pc.date_time) LIMIT
5;
```

| id | pokemon | trainer | defeats | hp | team | captured |
|---|---|---|---|---|---|---|
| 3575 | Mac | Norberto | 5 | 0 | 189 | 2005-03-13 04:16:39.000000 |
| 4940 | Rhett | Wiley | 4 | 0 | 436 | 2000-02-17 13:00:11.000000 |
| 425 | Friday | Mark | 4 | 0 | 115 | 2004-11-14 14:00:16.000000 |
| 4939 | Nibbles | Wiley | 4 | 0 | 436 | 2007-01-03 13:10:57.000000 |
| 8111 | Ginger | Karoline | 4 | 0 | 391 | 2007-08-22 16:35:04.000000 |

As you can see, all the Pokémon have an *HP* of zero, matching the criteria specified by the statement. Moreover, see that for those Pokémons that have been defeated the same number of times (4), these ones are ordered by the ones that were captured before (starting in 2000 and going up to 2007).

To validate the count of battles is correct, we will take the first Pokémon as an example (*id* of 3575 and *team* with *id* 189), and execute the following query which will display all the battles in which this Pokémon's team won but he had 0 HP:

```
SELECT pc.nickname, b.id
   FROM pokemon_creature pc JOIN fight f on pc.id = f.id_creature
   JOIN battle b on f.id_battle = b.id
   WHERE f.life_remaining = 0 AND pc.team = b.winner AND pc.team = 189
   AND pc.id = 3575;
```

5 rows

| | nickname | id |
|---|---|---|
| 1 | Mac | 326 |
| 2 | Mac | 180 |
| 3 | Mac | 470 |
| 4 | Mac | 469 |
| 5 | Mac | 457 |

As you can see in the screenshot above, this query retrieved five rows, which is exactly the count of battles displayed on the previous query. Meaning that, our count of battles works correctly and is able to identify the desired Pokémon.

## 6.3 Query 3

### 6.3.1 Solution

```
SELECT tr.id AS trainer, COUNT(pc.id) AS numpokemon
      FROM trainer tr JOIN team t on tr.id = t.trainer
      LEFT JOIN battle b ON t.id = b.winner
      JOIN pokemon_creature pc ON pc.trainer = tr.id
      WHERE b.winner IS NULL AND tr.experience > tr.gold^2
```

```
    GROUP BY tr.id;
```

| | trainer | numpokemon |
|---|---|---|
| 1 | 426 | 10 |
| 2 | 59 | 26 |
| 3 | 65 | 27 |
| 4 | 121 | 41 |
| 5 | 239 | 38 |
| 6 | 74 | 33 |
| 7 | 34 | 12 |
| 8 | 67 | 43 |
| 9 | 484 | 17 |
| 10 | 36 | 21 |
| 11 | 277 | 31 |
| 12 | 128 | 33 |
| 13 | 432 | 40 |

### 6.3.2 Explanation

To show the trainers that had never won a battle we did a *LEFT JOIN* with the ***Battle*** table (connecting the *id* of the ***Trainer*** with the *winner* attribute of the ***Battle*** entity) and specified in the *WHERE* clause that the *winner* attribute had to be *NULL*. We used a *LEFT JOIN* for this so that we could get the non-existing relationships between these two tables.

Also, to narrow down even more our output and display only those trainers that have more experience than squared amount of gold we added a second *WHERE* clause in which we specified the relationship between these two attributes.

Lastly, to count the number of Pokémons each trainer owns, we did a *JOIN* with the ***Pokemon_Creature*** entity (linking the *trainer* attribute of this one with the ***Trainer***'s *id*) and added a *GROUP BY* clause to join all the rows of the same trainer. This way, we could use the *COUNT()* operator to check how many Pokémons the trainer had.

### 6.3.3 Query validation

If execute the same query as before, but now displaying the *experience and gold* attributes from the ***Trainer*** entity, we will be able to check if the relationship between these is the expected:

```
SELECT tr.id AS trainer, COUNT(pc.id) AS numpokemon, tr.experience, tr.gold
    FROM trainer tr JOIN team t on tr.id = t.trainer
    LEFT JOIN battle b ON t.id = b.winner
    JOIN pokemon_creature pc ON pc.trainer = tr.id
    WHERE b.winner IS NULL AND tr.experience > tr.gold^2
    GROUP BY tr.id;
```

| | trainer | numpokemon | experience | gold |
|---|---|---|---|---|
| 1 | 426 | 10 | 365665 | 308 |
| 2 | 59 | 26 | 537855 | -275 |
| 3 | 65 | 27 | 494951 | -557 |
| 4 | 121 | 41 | 1880445 | 351 |
| 5 | 239 | 38 | 348903 | 192 |
| 6 | 74 | 33 | 1821347 | -563 |
| 7 | 34 | 12 | 1888246 | 1283 |
| 8 | 67 | 43 | 1672810 | 882 |
| 9 | 484 | 17 | 1451825 | -342 |
| 10 | 36 | 21 | 1875336 | 885 |
| 11 | 277 | 31 | 1347591 | 378 |
| 12 | 128 | 33 | 1237790 | 419 |
| 13 | 432 | 40 | 329603 | 387 |

As you can see, all the trainers displayed have more experience than squared their amount of gold.

Now, if we insert a new Pokémon to one of the trainers displayed (the first one for example) we can see how the count of Pokemon changes.

```
INSERT INTO pokemon_creature(id, trainer) VALUES (15000, 426);
```

| | trainer | numpokemon | experience | gold |
|---|---|---|---|---|
| 1 | 426 | 11 | 365665 | 308 |
| 2 | 59 | 26 | 537855 | -275 |
| 3 | 65 | 27 | 494951 | -557 |

You can see how the count was increased by one unit on the first trainer. Meaning that we were able to correctly count the number of Pokémons each trainer has.

To ensure the displayed trainers match the criteria of never having won a battle, we will take the first displayed trainer (*id* = 426) and show all the battles they have been in, and see if they won or lost:

```
SELECT tr.id AS trainer, bw.winner, bl.loser AS loser
   FROM trainer tr JOIN team t ON tr.id = t.trainer
   LEFT JOIN battle bl on t.id = bl.loser
   LEFT JOIN battle bw ON t.id = bw.winner
   WHERE tr.id = 426;
```

4 rows

| | trainer | winner | loser |
|---|---|---|---|
| 1 | 426 | <null> | 283 |
| 2 | 426 | <null> | 283 |
| 3 | 426 | <null> | 283 |
| 4 | 426 | <null> | 283 |

As we can see in the screenshot above, the trainer with *id* = 426 has fought in four battles, but he didn't win any (*winner* attribute is null). Notice how the *loser* attribute matches the trainer's team:

| id | trainer |
|---|---|
| 283 | 426 |

Therefore, we can conclude that the displayed trainer matched the criteria and are giving us the expected result.

## 6.4 Query 4

### 6.4.1 Solution

```
SELECT o.name AS organzation, t.name AS villain, ec.money_steal AS stolen
   FROM criminal_organization o
   JOIN evil_criminal ec ON o.name = ec.criminal_organization
   JOIN trainer t ON t.id = ec.id
   WHERE ec.money_steal IN
   (SELECT evc.money_steal
       FROM evil_criminal evc
       WHERE o.name = evc.criminal_organization
       ORDER BY evc.money_steal DESC
       LIMIT 2)
   ORDER BY o.name, money_steal DESC;
```

| | o.name | t.name | money_steal |
|---|---|---|---|
| 1 | Aqua | Mia | 1581 |
| 2 | Aqua | Willian | 1576 |
| 3 | Cipher | Carlton | 1497 |
| 4 | Cipher | Jaye | 1398 |
| 5 | Galactic | Jade | 1669 |
| 6 | Galactic | Erin | 1535 |
| 7 | Magma | Bennett | 1530 |
| 8 | Magma | Kerri | 1399 |
| 9 | Plasma | Ghetsis | 159 |
| 10 | Rocket | Giovanni | 69697000000000 |
| 11 | Rocket | Fidel | 1318 |

### 6.4.2 Explanation

To show the villains from each villainous organization we have joined the tables ***Criminal_Organization*** and ***Evil_Criminal*** (linking the *criminal_organization* attribute of the criminal with the organization). To get their names, we have also joined the ***Trainer*** entity (equaling the *id* of the evil criminal with the trainer's).

Moreover, to get the two criminals that have stolen the most money we had to do a subquery that returned the two criminals from each organization with the most stolen money and use it in the *WHERE* clause. We had to take this approach as there was no other way for us to check if the stolen money was one of the desired values. If the query

had only wanted the criminal with the most stolen money, we could have omitted this subquery and just use a *LIMIT 1* operator.

This subquery selected the stolen money of the two evil criminals that belonged to the organization being evaluated (specified in the inner *WHERE* clause), ordered them from the greatest to the lowest (*DESC*) and limited the output to only 2 (using the *LIMIT* operator).

To show the output with the desired format, we used an ORDER *BY* expression to force the output to show the names of the criminal organizations in alphabetical order and (as a secondary condition) from the greatest money stolen to the least (*DESC*).

### *6.4.3 Query validation*

To validate this query, we will display all the criminals from a criminal organization and sort them by the one with the most stolen money to the least using the following query:

```
SELECT o.name, t.name, ec.money_steal
      FROM criminal_organization o
      JOIN evil_criminal ec on o.name = ec.criminal_organization
      JOIN trainer t ON t.id = ec.id
      ORDER BY ec.money_steal DESC;
```

| | o.name | t.name | money_steal |
|---|---|---|---|
| 1 | Rocket | Giovanni | 69697000000000 |
| 2 | Galactic | Jade | 1669 |
| 3 | Aqua | Mia | 1581 |
| 4 | Aqua | Willian | 1576 |
| 5 | Aqua | Gregorio | 1567 |
| 6 | Galactic | Erin | 1535 |
| 7 | Magma | Bennett | 1530 |
| 8 | Aqua | Lela | 1502 |
| 9 | Cipher | Carlton | 1497 |
| 10 | Aqua | Ralph | 1472 |
| 11 | Galactic | Marisol | 1458 |
| 12 | Aqua | Hermine | 1417 |
| 13 | Magma | Kerri | 1399 |
| 14 | Cipher | Jaye | 1398 |
| 15 | Magma | Preston | 1350 |
| 16 | Cipher | Jonah | 1326 |
| 17 | Rocket | Fidel | 1318 |
| 18 | Aqua | Douglass | 1308 |
| 87 | Plasma | Ghetsis | 159 |

As you can see, the output matches with the one on the previous section. Notice how the Plasma criminal organization only gave us one row on the previous output. We will check how many evil criminals this one has with the following query:

```
SELECT ec.id FROM criminal_organization co
   JOIN evil_criminal ec on co.name = ec.criminal_organization
   WHERE co.name = 'Plasma';
```


As you can see, there is only one trainer in the criminal organization called ‘Plasma’. This explains why we only got one row for this organization in the previous outputs.

Now we will try to insert a new evil criminal into the ‘Aqua’ organization and give it a greater number of money stolen with the following query:

```
INSERT INTO evil_criminal(id, money_steal, criminal_organization)
VALUES (1, 9999999, 'Aqua');
```

We execute the initial query again and we get the following:

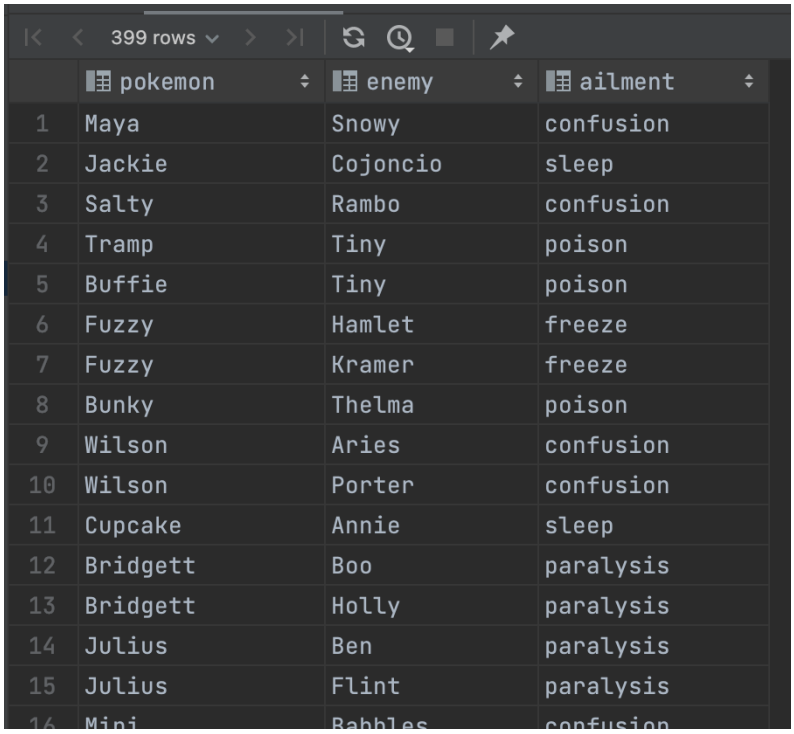
| | organzation | villain | stolen |
|---|---|---|---|
| 1 | Aqua | Genie | 9999999 |
| 2 | Aqua | Mia | 1581 |
| 3 | Cipher | Carlton | 1497 |
| 4 | Cipher | Jaye | 1398 |

As you can see, the villains being displayed for the ‘Aqua’ organization changed and now include this new criminal, as it has a greater number of stolen money.

## *6.5 Query 5*

### *6.5.1 Solution*

```
SELECT pc.nickname AS pokemon, e.nickname AS enemy, pc.status_condition AS
ailment
      FROM pokemon_creature pc JOIN fight f ON f.id_creature = pc.id
      JOIN battle b ON f.id_battle = b.id JOIN fight fe ON fe.id_battle = b.id
      JOIN pokemon_creature e ON fe.id_creature = e.id
      JOIN cause_or_receive cor ON e.id = cor.creature
      JOIN status s ON cor.move = s.id
      WHERE pc.status_condition NOT LIKE 'none' AND e.id != pc.id
      AND s.special_effect = pc.status_condition AND pc.team != e.team
      GROUP BY pc.id, e.id, s.special_effect;
```



399 rows

| | pokemon | enemy | ailment |
|---|---|---|---|
| 1 | Maya | Snowy | confusion |
| 2 | Jackie | Cojoncio | sleep |
| 3 | Salty | Rambo | confusion |
| 4 | Tramp | Tiny | poison |
| 5 | Buffie | Tiny | poison |
| 6 | Fuzzy | Hamlet | freeze |
| 7 | Fuzzy | Kramer | freeze |
| 8 | Bunky | Thelma | poison |
| 9 | Wilson | Aries | confusion |
| 10 | Wilson | Porter | confusion |
| 11 | Cupcake | Annie | sleep |
| 12 | Bridgett | Boo | paralysis |
| 13 | Bridgett | Holly | paralysis |
| 14 | Julius | Ben | paralysis |
| 15 | Julius | Flint | paralysis |
| 16 | Mini | Babbles | confusion |

As you can see in the picture, the query returns 399 rows (too many to fit in the screenshot).

#### *6.5.2 Explanation*

Firstly, to show all the Pokémon that are under an ailment state we selected the *nickname* of the ***Pokemon_Creature*** table and added in a *WHERE* clause that its *status_condition* attribute could not be ‘none’. Next, to get all the battles they fought in we joined the***Pokemon_Creature*** and ***Battle*** tables (by linking them with the ***Fight*** entity).

Now, to get all the Pokémons they fought against we did the same joins as before but in an inverse order (linking the **Battle** *id* of the ones gotten before to a new ***Fight*** instance and a new ***Pokemon_Creature*** to get the Pokémons that also fought on those battles). To ensure we don’t get duplicates (same Pokémon as it did fight in the battle) or Pokémons from the same team as the one we are evaluating, we added two conditions in the *WHERE* clause specifying that the two Pokémons had to be different (*pc.id != e.id)* and that they couldn't belong to the same team (*pc.team != e.team*).

Lastly, instead of getting all the Pokémons they fought against, we join these Pokémon with the ***Status*** (move) table (through the ***Cause_Or_Receive*** table). This way, we can add an extra condition in the *WHERE* clause specifying that the Pokémon must inflict an ailment of the same status as the affected Pokémon is in, discarding like this all those Pokémon that couldn’t have caused the damage.

To display the information, we add a *GROUP BY* clause that joins those rows that have the same two Pokémons with the same special effect (eliminating repetitions for those Pokemons that have two moves of the same type).

#### *6.5.3 Query validation*

To validate this query we will take the first displayed Pokémon of an example (Maya). We will display the query again showing the Pokémon *id* so we can use in further queries and only Maya's Pokemons by adding in the *WHERE* clause that the Pokémons *id* has to be 134:

1 row

| | id | pokemon | enemy | ailment |
|---|---|---|---|---|
| 1 | 134 | Maya | Snowy | confusion |

Now that we know Maya's *id* and have seen that there is only one row retrieved by the query, we will execute the following query to show all the Pokémon Maya has encountered in all its battles:

```
SELECT pc.id, pc.nickname, e.nickname, e.id, s.special_effect
   FROM pokemon_creature pc JOIN fight f ON f.id_creature = pc.id
   JOIN battle b ON f.id_battle = b.id JOIN fight fe ON fe.id_battle = b.id
   JOIN pokemon_creature e ON fe.id_creature = e.id
   JOIN cause_or_receive cor on e.id = cor.creature
   JOIN status s ON cor.move = s.id
   WHERE pc.id = 134
   ORDER BY s.special_effect;
```

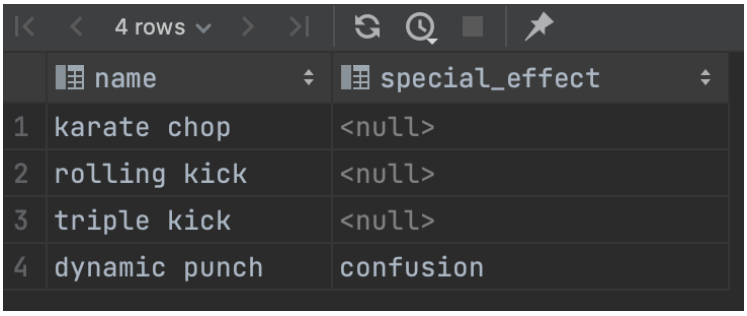
40 rows

| | pc.id | pc.nickname | e.nickname | e.id | special_effect |
|---|---|---|---|---|---|
| 1 | 134 | Maya | Lexus | 12328 | burn |
| 2 | 134 | Maya | Maximus | 8128 | burn |
| 3 | 134 | Maya | Winter | 8842 | burn |
| 4 | 134 | Maya | Winter | 8842 | burn |
| 5 | 134 | Maya | Winter | 8842 | burn |
| 6 | 134 | Maya | Lexus | 12328 | burn |
| 7 | 134 | Maya | Snowy | 8843 | confusion |
| 8 | 134 | Maya | Bumper | 9320 | freeze |
| 9 | 134 | Maya | Atlas | 8130 | freeze |
| 10 | 134 | Maya | Karma | 132 | infatuation |

As you can see, there is only one Pokémon (Snowy) that has a special effect that causes *confusion*, which we saw was the ailment state that Maya is in. Therefore, we can conclude that the output retrieved by the initial query makes sense as it only retrieved the one out of 40 Pokémons encountered that have a move that inflicts their ailment state.

Just to double check that the *Snow* Pokémon has that specific move, we will display all the moves that this Pokémon (*id* = 8843) has with the following query:

```
SELECT m.name, s.special_effect
   FROM move m JOIN cause_or_receive cor ON m.id = cor.move
   LEFT JOIN status s ON m.id = s.id
   JOIN pokemon_creature pc ON cor.creature = pc.id
   WHERE pc.id = 8843;
```



| | name | special_effect |
|---|---|---|
| 1 | karate chop | <null> |
| 2 | rolling kick | <null> |
| 3 | triple kick | <null> |
| 4 | dynamic punch | confusion |

As you can see, this Pokémon has four moves, out of which one can inflict an ailment state, which is in fact *confusion*, the ailment the Maya Pokémon has.

## 6.6 Query 6

### 6.6.1 Solution

```
SELECT ty.name, COUNT(b.loser)
      FROM team t LEFT JOIN battle b on b.loser = t.id
      JOIN gym_leader gl ON gl.id = t.trainer JOIN gym g ON gl.gym = g.id
      JOIN type ty ON g.id_type = ty.id
      GROUP BY ty.name ORDER BY COUNT(b.loser)
      LIMIT 1;
```

| | name | count |
|---|---|---|
| 1 | dragon | 0 |

### 6.6.2 Explanation

To get the Gym Leaders that belong to each Gym type we first join the ***Gym_Leader*** and ***Gym*** entities to get the gym each leader has. Then, we join the ***Gym*** and ***Type*** tables to get the type of the leader's gyms. Lastly, we use the *GROUP BY* clause to join all those trainers that belong to the same gym type. With this, we are able to use the *COUNT* operator to check how many Gym Leaders belong to each gym type.

To figure out which of the selected types is the hardest to beat, we check which has the least number of defeated trainers. To do so, we first link the ***Team*** and ***Gym_Leader*** entities to then be able to join the ***Team*** table with the *loser* attribute of the ***Battle*** entity (we use a *LEFT JOIN* for this relationship as we want to allow *NULL* relations to count those types that don't have any leader who has been defeated). Now we can use the *COUNT* operator to check how many losers each type has. We also use this operation in an *ORDER BY* clause so that we can sort them by the ones with the least number of defeats. Lastly, we add a *LIMIT* 1 to only get the type with the least number of defeats.

### 6.6.3 Query validation

As happened in other triggers, there are many rows that satisfy the given conditions, as you can see with the following query. If we use the same query but take out the *LIMIT* we get the following:

| | name | count |
|---|---|---|
| 1 | dragon | 0 |
| 2 | ice | 0 |
| 3 | grass | 0 |
| 4 | psychic | 0 |
| 5 | flying | 1 |
| 6 | rock | 1 |
| 7 | fire | 1 |
| 8 | normal | 1 |
| 9 | ground | 1 |
| 10 | water | 1 |
| 11 | fighting | 1 |
| 12 | steel | 2 |
| 13 | electric | 2 |
| 14 | ghost | 3 |
| 15 | bug | 3 |
| 16 | poison | 4 |

As you can see, there are four types whose leaders have not been defeated even once. Therefore, to check that our query actually works we will insert three new battles which define trainers from the ‘dragon’, ‘ice’ and ‘grass’ types as losers.

First, we identify these type’s *id* on the ***Type*** table and find a gym that identifies with that type in the ***Gym*** table:

| id | name |
|---|---|
| 12 | grass |
| 15 | ice |
| 16 | dragon |

| id | medal | id_item | id_type |
|---|---|---|---|
| 553 | Forest Badge | 306 | 12 |
| 551 | Glacier Badge | 352 | 15 |
| 545 | Rising Badge | 329 | 16 |

We will now find the gym leaders that own these three gyms.

| id | gym |
|---|---|
| 525 | 553 |

| id | gym |
|---|---|
| 514 | 551 |

| id | gym |
|---|---|
| 515 | 545 |

Finally, let’s find these trainer’s teams in the ***Team*** table:

| id | trainer |
|---|---|
| 321 | 525 |

| id | trainer |
|---|---|
| 413 | 514 |

| id | trainer |
|---|---|
| 468 | 515 |

Now we execute the following query to insert three new battles defining the found teams as losers and the team with *id* = 196 as the winner.

```
INSERT INTO battle(id, winner, loser) VALUES (650, 196, 321);
INSERT INTO battle(id, winner, loser) VALUES (651, 196, 413);
INSERT INTO battle(id, winner, loser) VALUES (652, 196, 468);
```

| name | count |
|---|---|
| psychic | 0 |
| normal | 1 |
| ground | 1 |
| water | 1 |
| flying | 1 |
| dragon | 1 |
| fire | 1 |
| ice | 1 |

As you can see, if we run the query again we now get that the count of defeats has increased for the selected types, having *'psychic'* as the only type without any defeated gym leader. This proves that the counting of the defeats works and that the query would be able to identify the type with less defeated leaders.

## 6.7 Trigger 1

### 6.7.1 Solution

```
DROP FUNCTION IF EXISTS teach_move CASCADE;
CREATE OR REPLACE FUNCTION teach_move() RETURNS TRIGGER AS $$
BEGIN
   IF(
      EXISTS
            (SELECT *
                  FROM trainer t JOIN pokemon_creature pc on pc.trainer = t.id
                  JOIN owns_items o on t.id = o.trainer_id
                  JOIN move_machine mm ON o.item_id = mm.id
                  WHERE mm.move = NEW.move AND pc.id = NEW.creature
            )
   )THEN
      --remove move machine
       DELETE FROM owns_items oi WHERE oi.trainer_id IN
            (SELECT t.id
                  FROM trainer t
                  JOIN pokemon_creature p ON t.id = p.trainer
                  WHERE p.id = NEW.creature)
       AND oi.item_id =
           (SELECT mm.id
                  FROM move_machine mm
                  JOIN move m ON mm.move = m.id
                  WHERE m.id = NEW.move);

      --delete previous move
       DELETE FROM cause_or_receive WHERE slot = NEW.slot
       AND creature = NEW.creature AND move != NEW.move;


   ELSE
      --error message
       INSERT INTO warnings(affected_table, error_message, date, user_name)
       SELECT 'trainer', concat('<',t.class, '><', t.name,
      '> attempted to teach his <', s.id, '> the move <', m.name,
       '> without the necessary move machine.'), CURRENT_DATE, t.id
           FROM trainer t JOIN pokemon_creature pc ON pc.trainer = t.id
           JOIN species s ON pc.species = s.id
           JOIN move m ON m.id = NEW.move
           WHERE pc.id = NEW.creature;
```

```
      --revert insert
       DELETE FROM cause_or_receive WHERE id = NEW.id
       AND move = NEW.move AND creature = NEW.creature;

   END IF;
   RETURN NEW;
END
$$ LANGUAGE plpgsql;


DROP TRIGGER IF EXISTS teach_new_move ON cause_or_receive CASCADE;
CREATE TRIGGER teach_new_move AFTER INSERT ON cause_or_receive
FOR EACH ROW
EXECUTE FUNCTION teach_move();
```

### 6.7.2 Explanation

We want to check if a trainer has the required move machine to teach a Pokémon a specific move. Therefore, we designed this trigger to be executed after a new *INSERT* was done to the ***Cause_Or_Recieve*** table, which links the ***Pokemon_Creature*** and ***Move*** entities. We were required to do it *AFTER* the insertion as in one of the options we had to revert the change (remove the inserted row).

Whenever the trigger gets activated, the function *teach_move()* is executed. This one has two main scenarios: the trainer has the necessary move machine to teach the move to their Pokémon, or if they haven't.

To check if we are on the first case, we use the *EXISTS* clause and a subquery to find out if the entered Pokémon's *id* corresponds to that of a trainer who has the required move machine. We do so by joining the ***Trainer*** and ***Pokemon_Creature*** entities and specifying in a *WHERE* clause that the Pokémons *id* has to be equal to the one just entered (using the *NEW* command). Then, we join the ***Trainer*** and ***Move_Machine*** entities (by connecting them through the ***Owns_Item*** table) and add as a second condition to the *WHERE* clause that forces the *move* the machine teaches to be the same as the one introduced. This query will return a row if the trainer does have the necessary move machine or nothing otherwise. If it returns a row, the function will execute the first *IF* statement. If not, it will execute the *ELSE.*

Once we have identified the trainer of the entered *Pokémon* to meet the requirements, we must do two operations. Firstly, we remove the move machine from the trainer's items by using the *DELETE FROM* command stating we want to eliminate information from the ***Owns_Item*** table. To specify which rows we want to eliminate, we add a *WHERE* clause in which we equal the *trainer_id* attribute of this table to the entered Pokémons *trainer* (which we find by means of a subquery by joining the ***Trainer*** and ***Pokemon_Creature*** entities and adding a *WHERE* clause to select only the Pokémon that has just been inserted) and also equal the *item_id* attribute to the move machine we want to delete (which we get by using a subquery that links the ***Move*** and ***Move_Machine*** entities and adds a *WHERE* condition to only get the move machine that teaches the newly entered move).

Also, when teaching a new move we want to remove the one that used to take up the same slot (if there was any). To do so we use the *DELETE FROM* command and specify (using a *WHERE* clause) we want to delete those rows from the ***Cause_Or_Recieve*** table

(linking ***Pokemon_Creature*** and ***Move***) that have the same slot as the newly entered move, belong to the same Pokémon creature, but connect to a different move. This way, we will first insert a new move with the same *id* and then remove the one that has that same *id* but a different move from the one we just inserted.

On the other hand, if the trainer doesn't have the necessary move machine to teach the move, the function performs two actions. First, we create a new entry in the ***Warnings*** table by using the *INSERT INTO* command and selecting '*trainer*' as the affected table, concatenating a string with the desired message, adding the current date, and selecting the trainer's id as the user. To do so, we link the ***Trainer*** and ***Pokemon_Creature*** tables and specify in the *WHERE* clause that we want this Pokémon to be the newly inserted one (*NEW* command). This already allows us to get the trainer *class*, *name*, and *id* which we need for the message. However, to get the specie's *id* and the name of the move we join the tables ***Species*** (linking it to the Pokémon) and ***Move*** (stating we want this to be the newly inserted one). Also, we use the *concat()* method to get the desired *error_message* attribute to insert it.

Lastly, if the trainer didn't meet the requirements, we are asked to revert the change and delete the newly inserted row to the ***Cause_Or_Receive*** table. Hence, we use the *DELETE FROM* command combined with a *WHERE* clause to specify we want to delete the row that has the same id as the just introduced one.

### *6.7.3 Query validation*

To validate this trigger, we will insert two new moves: one to a Pokémon whose trainer has the required move machine and another one that doesn't. Hence, we will see if our trigger works in both cases.

For example, if we check the following row of the ***Move_Machine*** table we can see how the move with *id* = 68 is taught by the machine with *id* = 322.

| | id | move |
|---|---|---|
| 21 | 322 | 68 |

If we now check the ***Owns_Item*** table, the trainer with *id* = 0 owns that move machine.

| trainer_id | item_id | obtention_method |
|---|---|---|
| 0 | 322 | REWARD |

And by checking the ***Pokemon_Creature*** table we can see that the same trainer owns the Pokémon with *id* = 8.

| | id | species | nickname | trainer |
|---|---|---|---|---|
| 1 | 8 | 387 | Blue | 0 |

Now if we check the moves that this Pokemon has in the ***Cause_Or_Recieve*** we get the following:

| creature | move | id |
|---|---|---|
| 8 | 79 | 4 |
| 8 | 147 | 3 |
| 8 | 71 | 1 |
| 8 | 73 | 2 |

Now we try to insert the move with *id* = 68 on its first position by executing the following query:

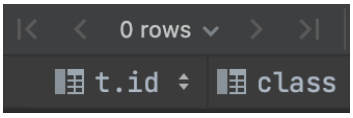
```
INSERT INTO cause_or_receive(creature, move, id) VALUES (8, 68, 1);
```

If we now check the ***Cause_Or_Receive*** table again, we can see how this one was updated as this Pokémon's trainer had the necessary move machine to teach the move. Notice how the first move of the Pokémon (*id* = 1) now represents the move with *id* = 68.

| creature | move | id |
|---|---|---|
| 8 | 79 | 4 |
| 8 | 147 | 3 |
| 8 | 68 | 1 |
| 8 | 73 | 2 |

Also, if we check if the trainer with *id* = 0 still has the move machine by executing the following simple query, we can see how no rows are returned, meaning the move machine was correctly deleted from the trainer's items:

```
SELECT *
   FROM trainer t JOIN owns_items oi ON oi.trainer_id = t.id
   JOIN move_machine mm ON oi.item_id = mm.id
   WHERE t.id = 0 AND mm.id = 322;
```




As we can see, the operation was successful for a Pokémon whose trainer met the requirements. Now we will do the opposite case by performing the same query again (now that the trainer with *id* = 0 no longer has the move machine).

```
INSERT INTO cause_or_receive(creature, move, id) VALUES (8, 68, 2);
```

We can see how the ***Cause_Or_Recieve*** table stayed the same (the move with *id* = 68 wasn't added on the second position or anywhere else, meaning it didn't replace the past move and its insertion was reverted).

| creature | move | id |
|---|---|---|
| 8 | 79 | 4 |
| 8 | 147 | 3 |
| 8 | 68 | 1 |
| 8 | 73 | 2 |

Also, if we check the warnings table we can see how we get the appropriate error message.

| | aff... | error_message | date | user_name |
|---|---|---|---|---|
| 1 | trainer | <Expert><Gema> attempted to teach his <387> the move <counter> without the necessary move machine. | 2023-05-20 | 0 |

## 6.8 Trigger 2

### 6.8.1 Solution

```
DROP FUNCTION IF EXISTS battle_reward CASCADE;
CREATE OR REPLACE FUNCTION battle_reward() RETURNS TRIGGER AS $$
BEGIN
       --Add winner gold and xp
       UPDATE trainer
           SET gold = gold + NEW.gold,
           experience = experience + 2.0/3.0 * NEW.xp
           FROM team tm
           WHERE tm.id = NEW.winner AND tm.trainer = trainer.id;

       --Add loser xp
       UPDATE trainer
           SET experience = experience + 1.0/3.0 * NEW.xp
           FROM team tm
           WHERE tm.id = NEW.loser AND tm.trainer = trainer.id;

       --Subtract gold if defeated by villain
       UPDATE trainer
           SET gold = gold - NEW.gold
           FROM evil_criminal ec
           JOIN team tec ON ec.id = tec.trainer
           JOIN team tt ON tt.id = NEW.loser
           WHERE tec.id = NEW.winner AND tt.trainer = trainer.id;

       --Give Gym Leader item to its defeater
       INSERT INTO owns_items(trainer_id, item_id, obtention_method, date_time)
           SELECT NEW.winner, i.id, 'WON IN BATTLE', CURRENT_TIMESTAMP
               FROM team t JOIN gym_leader gl ON t.trainer = gl.id
               JOIN gym g ON gl.gym = g.id JOIN item i ON g.id_item = i.id
               WHERE t.id = NEW.loser;

   RETURN NEW;
END
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS battle_result ON battle CASCADE;
CREATE TRIGGER battle_result AFTER INSERT ON battle
FOR EACH ROW
EXECUTE FUNCTION battle_reward();
```

### 6.8.2 Explanation

This trigger was defined to make the result of battles consistent. Therefore, we designed it to be activated right after a new battle is inserted. When this happens, the function *battle_reward()* gets executed.

The function we designed has been split into four different tasks. Firstly, we *UPDATE* the ***Trainer*** entity by adding to the winner of the battle the *gold* reward and increase their *experience* by ⅔ of the battle's *xp*. To do so, we just add to these attributes the newly inserted ones (using the *NEW* command). To identify which trainer we need to add it to, we do a *JOIN* with the ***Team*** entity and add a *WHERE* clause to specify that the row we want to update is that of the trainer that belongs to the team who was victorious.

Similarly, to add ⅓ of the battle's XP to the loser we *UPDATE* the ***Trainer*** entity by adding the inserted attribute (*NEW*) to the one they had before. To identify which row we want to

change, we add a *WHERE* clause that links the trainer's team with the *loser* attribute of the newly inserted battle.

Moreover, if the *winner* of the battle was a criminal, the gold reward they receive must be subtracted from the other trainer. To do so, we *UPDATE* again the ***Trainer*** table and *SET* their *gold* attribute to its previous value minus the inserted one (*NEW*). To ensure we only do this whenever the winner is a criminal, we join the ***Evil_Criminal*** and ***Team*** entities and add a *WHERE* clause that the criminal's team has to be the same as the *winner* attribute of the inserted battle. This will only leave us with the battles in which a criminal was victorious. However, to specify which row of the defeated trainer we want to change, we join the entity ***Team*** once more (now linking it to the *loser* attribute of the new battle) and add a second condition to the *WHERE* clause that links the team to the ***Trainer*** table we are updating.

Lastly, if a Gym Leader is beated, we want to give its item (the one on their gym) to its defeater. To do so, we use the *INSERT INTO* command to add a new row to the ***Owns_Item*** table (create a new relationship between a ***Trainer*** and an ***Item***), in which we insert the *winner* attribute of the new battle (using the *NEW* command) as the owner of the item, specify the obtention method as '*WON IN BATTLE*', and add the current time as the *date_time* of obtention (using the *CURRENT_TIMESTAMP* method). Lastly, to get the *item* of the Gym Leader and add it to the table in the *item_id* column (to correctly link the trainer and the item) we need to do a set of *JOIN*s. First, we join the ***Team*** and ***Gym_Leader*** entities to get the team of the Gym Leader, then we join the ***Gym_Leader*** to its ***Gym*** and lastly we link the ***Gym*** to the ***Item*** it has. To specify we want to get the Gym Leader who lost this battle we add a *WHERE* clause linking the leader's team with the *loser* of the newly inserted battle.

### *6.8.3 Query validation*

To validate this trigger we will insert four different battles (one of each of the mentioned cases) and see their effects. First, we will look at four different trainers:

Regulat trainers and their teams (***Trainer*** and ***Team*** entities): :

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 1 | 0 | Expert | Gema | 536957 | 3820 |
| 2 | 1 | Passionate Rider | Genie | 915295 | 8092 |

| | id | trainer |
|---|---|---|
| 1 | 196 | 0 |
| 2 | 392 | 1 |

Evil criminal and their team (***Evil_Criminal, Trainer,*** and ***Team*** entities):

| | id | money_steal | signat... | is_leader | sidekick | criminal_organization |
|---|---|---|---|---|---|---|
| 1 | 406 | 242 | You want a to… | false | 400 | Rocket |

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 407 | 406 | Rocket Executive | Lucien | 330109 | 9634 |

| | id | trainer |
|---|---|---|
| 407 | 13 | 406 |

Gym Leader and their team (***Trainer, Gym_Leader,*** and ***Team*** entities):

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 505 | 504 | Gym Leader | Koga | 1365804 | 5981 |

| | id | gym |
|---|---|---|
| 1 | 504 | 527 |

| | id | trainer |
|---|---|---|
| 505 | 424 | 504 |

Now that we have correctly identified these four trainers we can proceed to insert the battles.

First, we will insert a regular battle in which one regular trainer beats another one.

```
INSERT INTO battle(id, winner, loser, gold, xp)
VALUES(600, 196, 392, 100, 100);
```

If we run the trigger, then do the insert above and check the tables we get the following:
*Before:*

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 1 | 0 | Expert | Gema | 536957 | 3820 |
| 2 | 1 | Passionate Rider | Genie | 915295 | 8092 |

*After:*

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 1 | 0 | Expert | Gema | 537024 | 3920 |
| 2 | 1 | Passionate Rider | Genie | 915328 | 8092 |

As you can see, the winner of the battle (trainer with *id* = 0) increased their gold by 100 units (*gold* attribute of inserted battle) while the other trainer stayed with the same amount. Also, notice how the experience of the first trainer got increased by 67 units (⅔ of 100, *experience* of the new battle) and the second trainer increased its experience by 33 units (⅓ of a 100).

If we now insert a new battle in which an evil criminal defeats a regular trainer we obtain the following:

```
INSERT INTO battle(id, winner, loser, gold, xp)
VALUES(601, 13, 392, 100, 100);
```

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 2 | 1 | Passionate Rider | Genie | 915361 | 7992 |

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 407 | 406 | Rocket Executive | Lucien | 330176 | 9734 |

As you can see, the first trainer has decreased its gold by 100 units, while the second trainer has increased it by the same amount. Meaning that the evil criminal took the gold reward from the defeated trainer. Also if you check, the *experience* attribute has also been updated accordingly as the previous example.

Lastly, we insert a battle in which the defeated trainer is a Gym Leader and obtain the following:

```
INSERT INTO battle(id, winner, loser, gold, xp)
VALUES(603, 196, 424, 100, 100);
```

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 1 | 0 | Expert | Gema | 537091 | 4020 |

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 505 | 504 | Gym Leader | Koga | 1365837 | 5981 |

The first thing we can see is that, as in the previous examples, the *gold* and *experience* attributes have been updated correctly (increasing *gold* by a 100 for the first trainer and staying the same for the second one; and increasing ⅔ of 100 of experience for the winner and ⅓ for the loser).

However, we should now check if the winner has obtained the corresponding item. If we check the previous images we can see how the gym of the Gym Leader with an *id* = 504 has an *id* of 527. From the table ***Gym*** we can see how this one gives the item with *id* = 353.

| | id | medal | id_item | id_type |
|---|---|---|---|---|
| 1 | 527 | Sould Badge | 353 | 4 |

Now if we check the ***Owns_Item*** table we can see there is a new entry linking the winner trainer (*id* = 196) with the gym's item (*id* = 353) with the designated obtention method.

| | trainer_id | item_id | obtention_method | date_time |
|---|---|---|---|---|
| 1 | 196 | 353 | WON IN BATTLE | 2023-05-20 08:01:16.565669 |

All of these inserts represent the different cases our trigger manages and prove its correct functioning by updating the trainers as expected.

# 7 Let's go Pokéshopping

## 7.1 Query 1

### 7.1.1 Solution

```
SELECT item_id, i.name, p.trainer_id, t.name
FROM purchase as p
JOIN item as i on p.item_id = i.id
JOIN trainer as t ON p.trainer_id = t.id
WHERE trainer_id = (SELECT p.trainer_id
FROM purchase as p
GROUP BY p.trainer_id
ORDER BY SUM(p.final_price * amount) DESC
LIMIT 1)
ORDER BY amount DESC
LIMIT 1;
```

| | item_id | i.name | trainer_id | t.name |
|---|---|---|---|---|
| 1 | 316 | tm12 | 70 | Freeman |

### 7.1.2 Explanation

For this query, we have to filter by the trainer ID that has spent the most amount of money in stores. To do that, we have done a subquery that selects the trainer ID from the **Purchase** table group and does a group by **Trainer** so we could sum the final_price and amount by Trainer. In order to obtain the one that spent the most, we organized the result by the sum count in descendent order and limited by one. Finally, to select its favorite item we order by amount in descendent order and also limit by 1.

### 7.1.3 Query validation

| | store_id | item_id | trainer_id | amount | final_price | discount | dat |
|---|---|---|---|---|---|---|---|
| 11 | 581 | 56 | 0 | 1 | 1000 | 0 | 2008-1 |
| 12 | 572 | 312 | 0 | 1 | 50000 | 0 | 2007-0 |
| 13 | 593 | 62 | 0 | 2 | 2000 | 0 | 2012-1 |
| 14 | 569 | 441 | 1 | 2 | 0 | 10 | 2011- |
| 15 | 564 | 555 | 1 | 2 | 0 | 56 | 1996- |
| 16 | 581 | 544 | 1 | 3 | 0 | 0 | 2019- |
| 17 | 561 | 252 | 1 | 2 | 4000 | 0 | 2003- |
| 18 | 592 | 256 | 1 | 3 | 4000 | 0 | 2003- |
| 19 | 590 | 66 | 1 | 2 | 20 | 0 | 2001- |
| 20 | 565 | 409 | 2 | 1 | 0 | 0 | 2017- |

If we select everything from the **Purchase** table and order by trainer_id, we obtain that for instance Trainer with ID = 1, has purchased a total of 8000 + 12000+40. If we do the sum up just with the following query, we obtain the same result. This confirms obtaining the player that has spent the most amount of money in stores. The rest are simply joins that were validated in phase 2 and an order by amount in descending order.

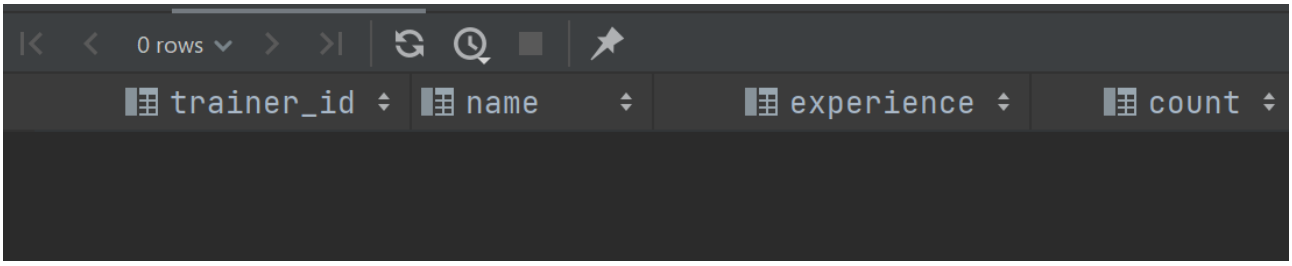
```
SELECT p.trainer_id, SUM(p.final_price * amount) as money
FROM purchase as p
GROUP BY p.trainer_id
ORDER BY SUM(p.final_price * amount) DESC;
```

| | trainer_id | money |
|---|---|---|
| 1 | 0 | 78800 |
| 2 | 1 | 20040 |
| 3 | 2 | 8500 |
| 4 | 3 | 46420 |
| 5 | 4 | 152081 |
| 6 | 5 | 45363 |

## 7.2 Query 2

### 7.2.1 Solution

```
SELECT o.trainer_id, t.name, t.experience, COUNT(DISTINCT
o.item_id)
FROM owns_items as o
JOIN trainer as t ON o.trainer_id = t.id
JOIN item as i ON o.item_id = i.id
WHERE t.id IN (SELECT t.id
FROM trainer AS t
LEFT JOIN purchase AS p ON t.id = p.trainer_id
WHERE p.trainer_id IS NULL
ORDER BY t.name)
GROUP BY o.trainer_id, t.name, t.experience
HAVING COUNT(DISTINCT o.item_id) > 10 and t.experience > 3000
ORDER BY COUNT(DISTINCT o.item_id);
```

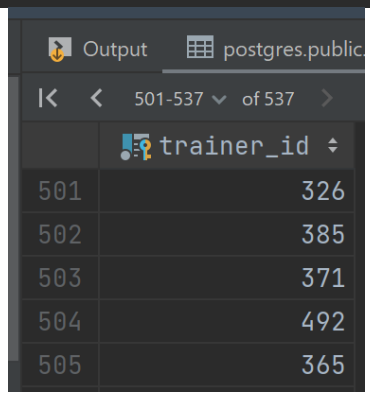



### 7.2.2 Explanation

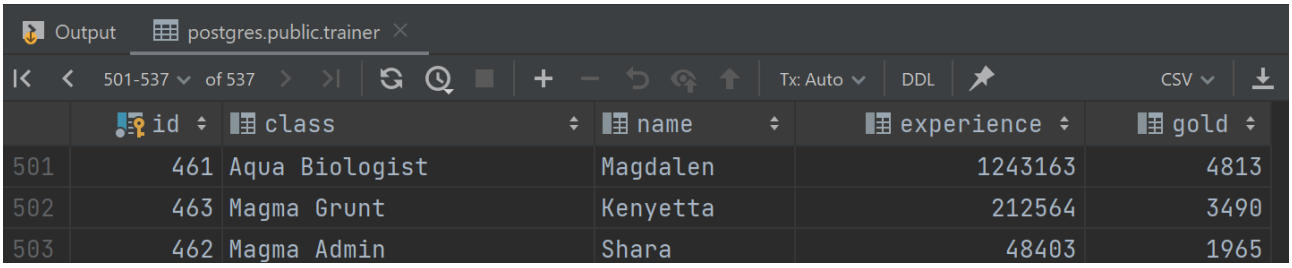
Since the query needs to list all the players that have never bought an item, we do a WHERE clause and a subquery to select them. To do so, we do a left join on the trainer ID. Finally, we do a group by ID, name and experience to filter those that have over 10 different items and experience is greater than 3000.

### 7.2.3 Query validation

```
SELECT trainer_id
FROM purchase
group by trainer_id;
```




```
SELECT *
FROM trainer;
```

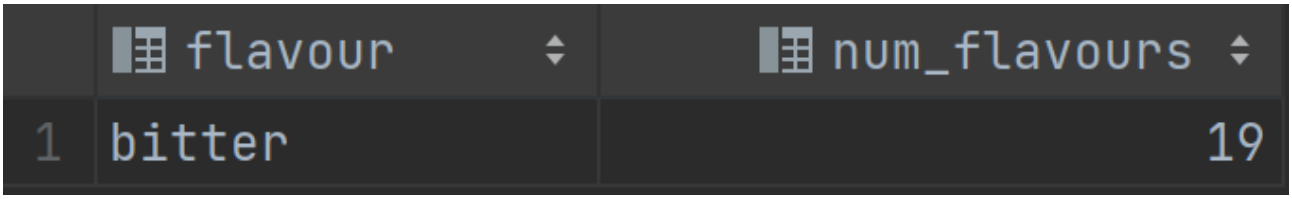



Since this query didn't return anything, we do a group by trainer_id in **Purchase** and compare the number of rows to the number of **Trainers**. In both cases, we obtain 537 rows meaning that all trainers stored in the system have at least purchased something.

## 7.3 Query 3

### 7.3.1 Solution

```
SELECT DISTINCT hf.flavour, COUNT(DISTINCT i.id) as num_flavours
FROM sell as s
JOIN item as i on s.item_id = i.id
JOIN berrie as b on i.id = b.id_item
JOIN has_flavour hf on b.id_item = hf.id_berrie
JOIN flavour as f on hf.flavour = f.name
GROUP BY hf.flavour
ORDER BY  COUNT(DISTINCT i.id)
LIMIT 1;
```



### 7.3.2 Explanation

To start, we do a join from **sell** with **item** since we want the items that can be purchased in stores and **berrie** because those are the only elements that can have flavors. Then, we join berrie with flavor through has_flavour to be able to access flavors information. We do a group by flavor, count the distinct IDs that appear and order by that in ascending order. Finally, we limit by 1 to have the least possible.

### 7.3.3 Query validation

To validate it, we have eliminated the LIMIT by 1, so all the flavors appeared together with the number of times used.

| | flavour | num_flavours |
|---|---|---|
| 1 | bitter | 19 |
| 2 | spicy | 21 |
| 3 | sour | 22 |
| 4 | sweet | 24 |
| 5 | dry | 27 |

For each flavor, we have done the following query in order to know the number of rows. In our case, dry gives us 27 rows, sour 22, spicy 21, sweet 24 and bitter 19. These results are equal to the ones obtained before.

Query for dry flavour:

```
SELECT DISTINCT i.id, i.name, hf.flavour, hf.strength
FROM sell as s
JOIN item as i on s.item_id = i.id
JOIN berrie as b on i.id = b.id_item
JOIN has_flavour hf on b.id_item = hf.id_berrie
JOIN flavour as f on hf.flavour = f.name
WHERE hf.flavour LIKE 'dry'
ORDER BY i.id;
```

Query for sour flavour:

```
SELECT DISTINCT i.id, i.name, hf.flavour, hf.strength
FROM sell as s
JOIN item as i on s.item_id = i.id
JOIN berrie as b on i.id = b.id_item
JOIN has_flavour hf on b.id_item = hf.id_berrie
JOIN flavour as f on hf.flavour = f.name
WHERE hf.flavour LIKE 'sour'
ORDER BY i.id;
```

Query for spice flavour:

```
SELECT DISTINCT i.id, i.name, hf.flavour, hf.strength
FROM sell as s
JOIN item as i on s.item_id = i.id
JOIN berrie as b on i.id = b.id_item
JOIN has_flavour hf on b.id_item = hf.id_berrie
JOIN flavour as f on hf.flavour = f.name
WHERE hf.flavour LIKE 'spicy'
ORDER BY i.id;
```

Query for sweet flavour:

```
SELECT DISTINCT i.id, i.name, hf.flavour, hf.strength
FROM sell as s
JOIN item as i on s.item_id = i.id
JOIN berrie as b on i.id = b.id_item
JOIN has_flavour hf on b.id_item = hf.id_berrie
JOIN flavour as f on hf.flavour = f.name
WHERE hf.flavour LIKE 'sweet'
ORDER BY i.id;
```

Last query for dry flavour:

```
SELECT DISTINCT i.id, i.name, hf.flavour, hf.strength
FROM sell as s
JOIN item as i on s.item_id = i.id
JOIN berrie as b on i.id = b.id_item
JOIN has_flavour hf on b.id_item = hf.id_berrie
JOIN flavour as f on hf.flavour = f.name
WHERE hf.flavour LIKE 'bitter'
ORDER BY i.id;
```

## 7.4 Query 4

### 7.4.1 Solution

```
SELECT combined.item_id, MAX(combined.trainers) AS trainers,
MAX(combined.stock) AS stock, MAX(combined.sum) AS sum
FROM
(
   SELECT item_id, trainers, 0 AS stock, 0 AS sum
   FROM
   (
       SELECT item_id, COUNT(DISTINCT trainer_id) AS trainers
       FROM owns_items
       GROUP BY item_id
   ) AS trainers_query

   UNION
```

```sql
    SELECT item_id, 0 AS trainers, stock, 0 AS sum
    FROM sell
    WHERE stock = (SELECT MIN(stock) FROM sell)

    UNION

    SELECT item_id, 0 AS trainers, 0 AS stock, SUM(amount) AS sum
    FROM purchase
    GROUP BY item_id
) AS combined
GROUP BY combined.item_id
ORDER BY trainers, stock, sum
LIMIT 1;
```

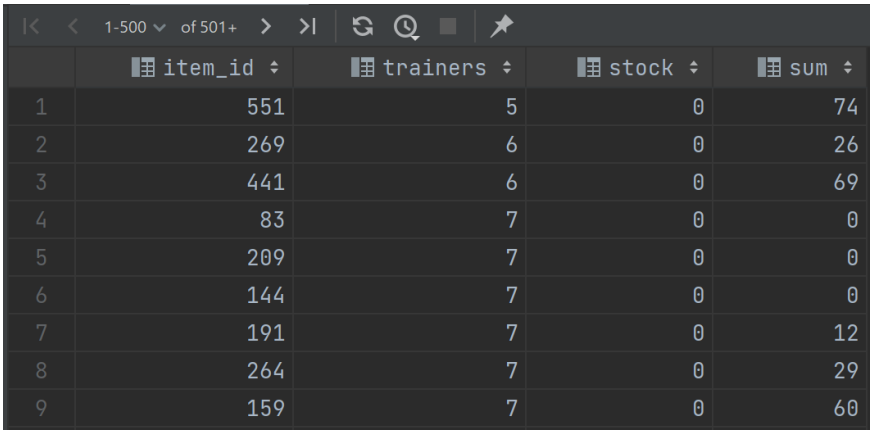
| | item_id | trainers | stock | sum |
|---|---|---|---|---|
| 1 | 551 | 5 | 0 | 74 |

### 7.4.2 Explanation

This query aims to join three different queries with different criteria each. This is the reason why we have all combined with a UNION and from the result, we have selected the maximum value of all three. For example in stock when looking at the number of times each Item has been purchased, we fill this column with zeros, so at the end we want to have the maximum value which will correspond to the query that gives this solution.

### 7.4.3 Query validation

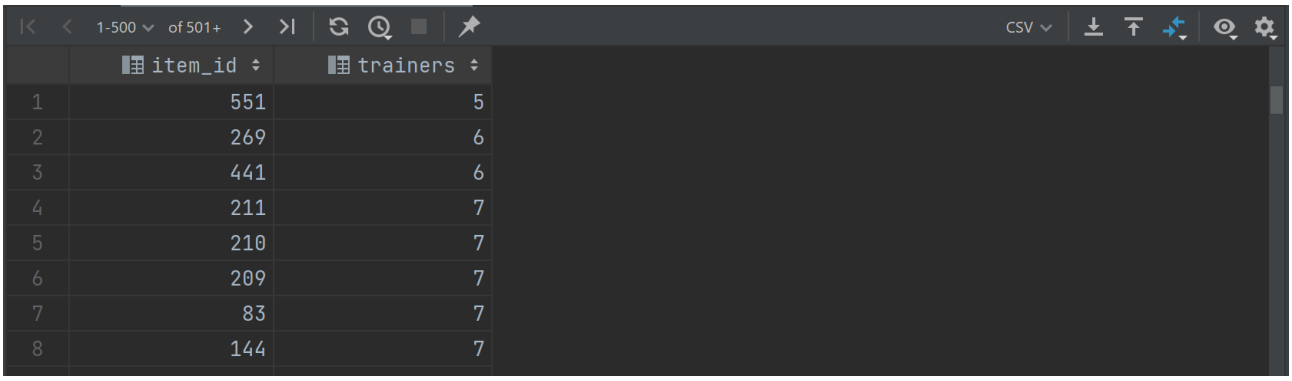
If we eliminate the LIMIT by 1, we will see that columns are combined and correspond to the three queries done separetely.




| | item_id | trainers | stock | sum |
|---|---|---|---|---|
| 1 | 551 | 5 | 0 | 74 |
| 2 | 269 | 6 | 0 | 26 |
| 3 | 441 | 6 | 0 | 69 |
| 4 | 83 | 7 | 0 | 0 |
| 5 | 209 | 7 | 0 | 0 |
| 6 | 144 | 7 | 0 | 0 |
| 7 | 191 | 7 | 0 | 12 |
| 8 | 264 | 7 | 0 | 29 |
| 9 | 159 | 7 | 0 | 60 |

Looking at all the rows, we'll see that stock is always zero but that happens because we select the minimum stock from sell since the objective of this query is to select the rarest item and that value is zero.

```sql
--number of items owned by trainers
SELECT item_id, COUNT(DISTINCT trainer_id) as trainers
FROM owns_items
GROUP BY item_id
ORDER BY trainers;
```

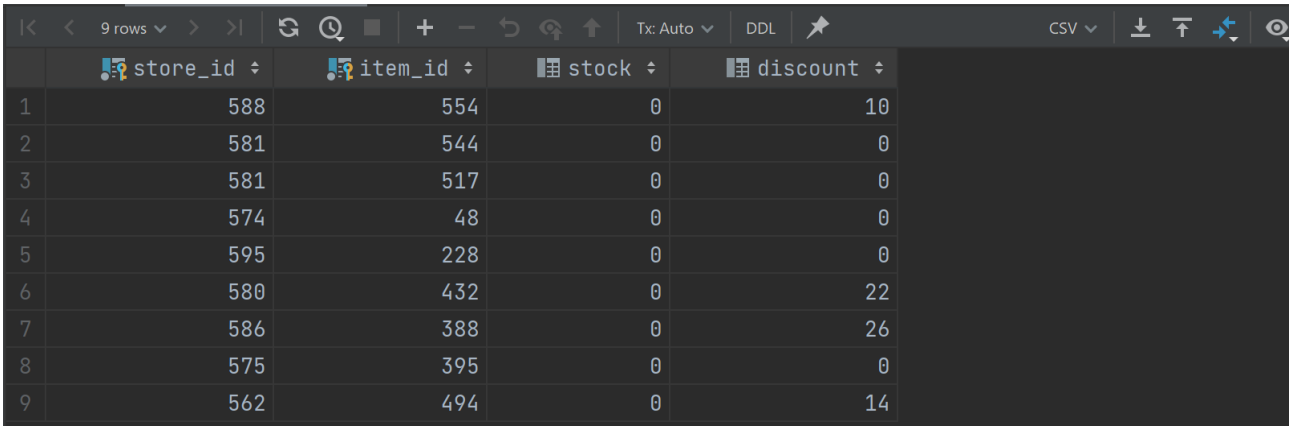


| | item_id | trainers |
|---|---|---|
| 1 | 551 | 5 |
| 2 | 269 | 6 |
| 3 | 441 | 6 |
| 4 | 211 | 7 |
| 5 | 210 | 7 |
| 6 | 209 | 7 |
| 7 | 83 | 7 |
| 8 | 144 | 7 |

```sql
--store that each store has
SELECT *
FROM sell
WHERE stock = (SELECT MIN(stock)
FROM sell)
ORDER BY stock ;
```

Or

```sql
SELECT *
FROM sell
WHERE stock = (SELECT stock
FROM sell
GROUP BY stock
ORDER BY stock
LIMIT 1)
ORDER BY stock ;
```



| | store_id | item_id | stock | discount |
|---|---|---|---|---|
| 1 | 588 | 554 | 0 | 10 |
| 2 | 581 | 544 | 0 | 0 |
| 3 | 581 | 517 | 0 | 0 |
| 4 | 574 | 48 | 0 | 0 |
| 5 | 595 | 228 | 0 | 0 |
| 6 | 580 | 432 | 0 | 22 |
| 7 | 586 | 388 | 0 | 26 |
| 8 | 575 | 395 | 0 | 0 |
| 9 | 562 | 494 | 0 | 14 |

```sql
-- number of times each item has been purchased
SELECT item_id, SUM(amount) as sum
FROM purchase
GROUP BY item_id
ORDER BY sum;
```

457 rows

| | item_id | sum |
|---|---|---|
| 447 | 551 | 74 |
| 448 | 552 | 51 |
| 449 | 553 | 57 |
| 450 | 554 | 51 |
| 451 | 555 | 18 |
| 452 | 556 | 48 |
| 453 | 557 | 51 |
| 454 | 558 | 93 |

Overall, we see that results match with the combined query.

## 7.5 Query 5

### 7.5.1 Solution

```sql
SELECT
    ( SUM(money) - SUM(total_discount) ) AS total_discount_paid
FROM (
    SELECT
        (i.base_price * p.amount) AS money,
        (p.final_price * p.amount) AS total_discount
    FROM purchase AS p
    JOIN item AS i ON i.id = p.item_id
    WHERE EXTRACT(YEAR FROM p.date_time) = 2020
) AS discounts;
```

1 row

| | total_discount_paid |
|---|---|
| 1 | 429937 |

### 7.5.2 Explanation

To give the total discounts, we have done a subquery with a WHERE clause to filter date_time by year 2020, and from that JOIN with **Item** to calculate the price that would be paid without the discount, multiplying the base_price times the amount for each **purchase**. Then, we do a similar operation to get the final_price with the discount. From those results, we sum all the total amount paid with and without the discount, subtract them and we get the total_discounts in that year.

### 7.5.3 Query validation

```
SELECT i.id, base_price, p.amount,
       (i.base_price * p.amount) AS money,
       p.final_price,
       (p.final_price * p.amount) AS discount
   FROM purchase AS p
   JOIN item AS i ON i.id = p.item_id
   WHERE EXTRACT(YEAR FROM p.date_time) = 2020;
```

304 rows — CSV

| | id | base_price | amount | money | final_price | discount |
|---|---|---|---|---|---|---|
| 1 | 26 | 700 | 3 | 2100 | 315 | 945 |
| 2 | 110 | 2000 | 2 | 4000 | 2000 | 4000 |
| 3 | 492 | 0 | 10 | 0 | 0 | 0 |
| 4 | 123 | 50 | 2 | 100 | 50 | 100 |
| 5 | 213 | 1000 | 10 | 10000 | 1000 | 10000 |
| 6 | 107 | 3000 | 2 | 6000 | 3000 | 6000 |
| 7 | 282 | 1000 | 2 | 2000 | 1000 | 2000 |
| 8 | 79 | 400 | 2 | 800 | 348 | 696 |

This query helps us check that we properly obtain the money paid without and with the discount. For example, the item with ID 26 was three times purchased in the same date_time, so the amount paid will be equal to its base_price times the amount. Same for the discount but multiplying final_price instead of base_price.

## 7.6 Query 6

### 7.6.1 Solution 1

```
SELECT *
FROM berrie
WHERE berrie.id_item = (
SELECT item_id
FROM owns_items
JOIN item i on i.id = owns_items.item_id
JOIN berrie b on i.id = b.id_item
WHERE obtention_method LIKE '%HARVESTED'
GROUP BY item_id
ORDER BY COUNT(obtention_method) DESC, item_id DESC
LIMIT 1);
```

1 row — Tx: Auto — DDL — CSV

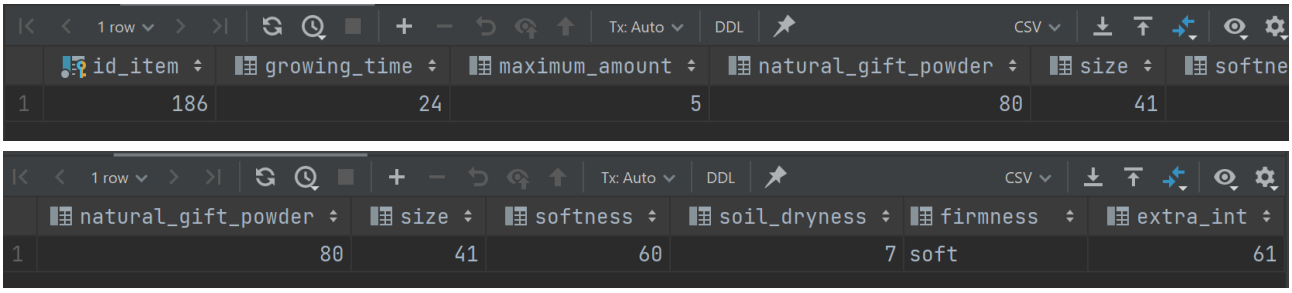
| | id_item | growing_time | maximum_amount | natural_gift_powder | size | softne |
|---|---|---|---|---|---|---|
| 1 | 186 | 24 | 5 | 80 | 41 | |

1 row — Tx: Auto — DDL — CSV

| | natural_gift_powder | size | softness | soil_dryness | firmness | extra_int |
|---|---|---|---|---|---|---|
| 1 | 80 | 41 | 60 | 7 | soft | 61 |

### 7.6.2 Solution 2

```
SELECT b.id_item, growing_time, maximum_amount,
natural_gift_powder, size, softness, soil_dryness, firmness,
extra_int
FROM owns_items oi
JOIN berrie b ON oi.item_id = b.id_item
WHERE oi.obtention_method LIKE '%HARVESTED'
GROUP BY oi.item_id, b.id_item, growing_time, maximum_amount,
natural_gift_powder, size, softness, soil_dryness, firmness,
extra_int
ORDER BY COUNT(DISTINCT oi.trainer_id) DESC, item_id DESC
LIMIT 1;
```



| | id_item | growing_time | maximum_amount | natural_gift_powder | size | softne |
|---|---|---|---|---|---|---|
| 1 | 186 | 24 | 5 | 80 | 41 | |



| | natural_gift_powder | size | softness | soil_dryness | firmness | extra_int |
|---|---|---|---|---|---|---|
| 1 | 80 | 41 | 60 | 7 | soft | 61 |

### 7.6.3 Explanation

This last query has two approaches. On the first one, there's a subquery that searches the id of the **berrie** that has been harvested most times. We obtain this with a WHERE clause to filter the obtention method of **owns_items** to those that contain "HARVESTED". Then, it is grouped by item ID for the counting and also ordered by item to be able to obtain the same results in both approaches since it is limited by 1.

Second approach eliminates the subquery and adds attributes to consider when printing a **berrie** in the group by. Also, instead of counting harvested, we count DISTINCT trainer_id because we still have a WHERE clause that selects those obtention_method that contain HARVESTED.

### 7.6.4 Query validation

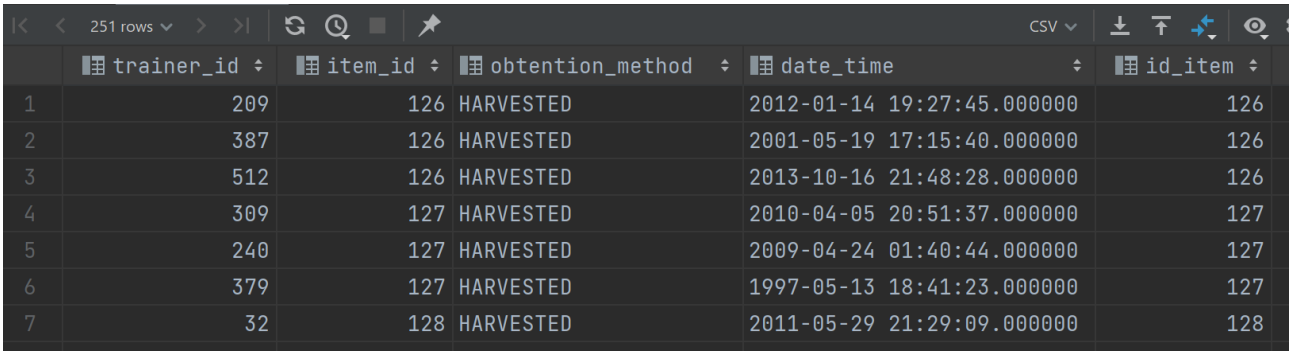
```
SELECT oi.item_id, COUNT(DISTINCT oi.trainer_id)
FROM owns_items oi
JOIN berrie b ON oi.item_id = b.id_item
WHERE oi.obtention_method LIKE '%HARVESTED'
GROUP BY oi.item_id
ORDER BY item_id;
```

To validate it, we have done this query that does the group by item_id and filters by those containing harvested in obtention_method. It counts DISTINCT trainer_ids and it is ordered by item_id to ease the work.

| | item_id | count |
|---|---|---|
| 1 | 126 | 3 |
| 2 | 127 | 3 |
| 3 | 128 | 4 |
| 4 | 129 | 5 |
| 5 | 130 | 3 |
| 6 | 131 | 8 |
| 7 | 132 | 6 |
| 8 | 133 | 8 |

To validate the counting, we have compiled the following query.

```
SELECT *
FROM owns_items
JOIN berrie b ON item_id = b.id_item
WHERE obtention_method LIKE '%HARVESTED'
ORDER BY item_id ;
```



| | trainer_id | item_id | obtention_method | date_time | id_item |
|---|---|---|---|---|---|
| 1 | 209 | 126 | HARVESTED | 2012-01-14 19:27:45.000000 | 126 |
| 2 | 387 | 126 | HARVESTED | 2001-05-19 17:15:40.000000 | 126 |
| 3 | 512 | 126 | HARVESTED | 2013-10-16 21:48:28.000000 | 126 |
| 4 | 309 | 127 | HARVESTED | 2010-04-05 20:51:37.000000 | 127 |
| 5 | 240 | 127 | HARVESTED | 2009-04-24 01:40:44.000000 | 127 |
| 6 | 379 | 127 | HARVESTED | 1997-05-13 18:41:23.000000 | 127 |
| 7 | 32 | 128 | HARVESTED | 2011-05-29 21:29:09.000000 | 128 |

As we can observe item with ID 126 has been harvested three times, so this confirms the three obtained in the first query, which at the end is signaling that if it's ordered by that count and limited by 1, we will obtain the berry that has been harvested the most times.

## 7.7 Trigger 1

### 7.7.1 Solution

```
DROP FUNCTION IF EXISTS calculate_healing CASCADE;
CREATE OR REPLACE FUNCTION calculate_healing() RETURNS TRIGGER AS
$$
DECLARE
 heal INTEGER;
 maxHeal INTEGER;
 currentHP INTEGER;
BEGIN
   IF OLD.item_id IN (SELECT id From healing_item) THEN

       currentHP := (SELECT hp FROM pokemon_creature
```

```
        WHERE pokemon_creature.id = (SELECT p.id FROM
pokemon_creature as p
        WHERE p.position = 1 and p.trainer = OLD.trainer_id));

        heal :=  currentHP + (SELECT hi.status_heal
        from healing_item as hi
        WHERE OLD.item_id = hi.id);

        maxHeal := (SELECT ((2 * hp + (level ^ 2) / 4) / 100 + 5) *
nature
        FROM pokemon_creature
        WHERE pokemon_creature.id = (SELECT p.id FROM
pokemon_creature as p
        WHERE p.position = 1 and p.trainer = OLD.trainer_id));

        IF heal <= maxHeal THEN
            UPDATE pokemon_creature
            SET hp = heal
            WHERE pokemon_creature.id = (SELECT p.id
            FROM pokemon_creature as p
            WHERE p.position = 1 and p.trainer = OLD.trainer_id);
        ELSE
            UPDATE pokemon_creature
            SET hp = maxHeal
            WHERE pokemon_creature.id = (SELECT p.id
            FROM pokemon_creature as p
            WHERE p.position = 1 and p.trainer = OLD.trainer_id);

        end if;


    END IF;

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS healing_result ON owns_items CASCADE;
CREATE TRIGGER healing_result AFTER DELETE ON owns_items
FOR EACH ROW
EXECUTE FUNCTION calculate_healing();
```

### 7.7.2 Explanation

Whenever a delete is done in owns_items, we check whether this OLD.id was a healing item by doing a subquery. If it is, we set the current HP by first finding the pokemon_creature that is in the first position and whose trainer was the one we have just deleted. Then, we want to sum up that value plus the status heal of the Item we have removed specifying this in the WHERE clause. In order not to surpass the maximum hp we calculate it with the formula for the same pokemom_creature as before. If the sum is less than the maximum, we update this hp of the pokemon_creature to the sum, otherwise we set it to the maximum Heal calculated.

### 7.7.3 Query validation

For the query validation, whenever I delete an item I want to make sure I am getting it stauts_heal and maxHeal correctly, so I insert those values into the warnings table.

```
INSERT INTO warnings
SELECT hi.status_heal
from healing_item as hi
WHERE OLD.item_id = hi.id;
```

Here are the tests done to test the trigger.

```
DELETE FROM owns_items
WHERE item_id = 26 and trainer_id = 4;
```

For item_id 26, we found in healing_item that status_heal is 77 and calculating with the level, base stat and nature of the pokemon_creature we obtain maxHeal 117 which corresponds to the **warnings** table. Here it's important to note that the pokemon_creature chosen is Noel since it has trainer_id = 4 and position =1.

| | id | hp_recovered | status_heal | hp_percentage |
|---|---|---|---|---|
| 1 | 32 | 80 | 28 | 90 |
| 2 | 26 | 50 | 77 | 92 |

| | message |
|---|---|
| 1 | 77 |
| 2 | 117 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 128 | 120 | 426 | Noel | 4 | 9 | 6 |

Finally, we'll check whether it has updated or not the hp. In our trigger, we modify Noel's row in pokemon_creature and set his hp to 83, which was previously defined as 6.

| | | | | | | |
|---|---|---|---|---|---|---|
| 128 | false | 1 | 83 | none | 59 | |

## 7.8 Trigger 2

### 7.8.1 Solution

```
DROP TABLE IF EXISTS warnings CASCADE;
CREATE TABLE warnings (
```

```
   message TEXT
);

DROP FUNCTION IF EXISTS modify_purchase CASCADE;
CREATE OR REPLACE FUNCTION modify_purchase() RETURNS TRIGGER AS $$
DECLARE
 price INTEGER;
 newStock INTEGER;
 newGold INTEGER;
BEGIN

   price := (SELECT i.base_price
                     FROM item as i
                     WHERE NEW.item_id = i.id) -
     (SELECT i.base_price
                     FROM item as i
                       WHERE NEW.item_id = i.id) * NEW.discount /
100;

   UPDATE purchase
   SET final_price = price
   WHERE NEW.trainer_id = trainer_id AND NEW.item_id = item_id AND
         NEW.store_id = store_id AND NEW.date_time = date_time;


    newGold := (SELECT gold FROM trainer WHERE NEW.trainer_id =
trainer.id) - ( price * NEW.amount);
   IF newGold < 0 THEN
       INSERT INTO warnings
       SELECT CONCAT(
           'Trainer #', t.id,
           ' tried to buy ', i.name,
           ' but his card has been declined, insufficient funds.'
       )
       FROM purchase
       JOIN trainer as t ON purchase.trainer_id = t.id
       JOIN item as i ON purchase.item_id = i.id
       JOIN shop as s ON purchase.store_id = s.id
          WHERE NEW.trainer_id = t.id and NEW.item_id = i.id AND
NEW.store_id=s.id and NEW.date_time = date_time;

       DELETE FROM purchase
        WHERE trainer_id = NEW.trainer_id and item_id = NEW.item_id
and store_id = NEW.store_id and
              date_time = NEW.date_time and amount=NEW.amount and
discount=NEW.discount;
   else

        newStock = (SELECT stock FROM SELL
            where NEW.store_id = sell.store_id and NEW.item_id =
sell.item_id) - NEW.AMOUNT;
       IF newStock < 0 THEN
```

```
            INSERT INTO warnings
            SELECT CONCAT(
                'Trainer #', t.id,
                ' tried to buy ', i.name,
                ' at ', s.name, ' store in ',
                sub.name, ', however the store ran out of stock.'
            )
            FROM purchase
            JOIN trainer as t ON purchase.trainer_id = t.id
            JOIN item as i ON purchase.item_id = i.id
            JOIN shop as s ON purchase.store_id = s.id
            JOIN subarea as sub on s.id = sub.id
             WHERE NEW.trainer_id = t.id and NEW.item_id = i.id and
NEW.store_id = s.id;

            DELETE FROM purchase
                 WHERE trainer_id = NEW.trainer_id and item_id =
NEW.item_id and store_id = NEW.store_id and
              date_time = NEW.date_time and amount=NEW.amount and
discount=NEW.discount;

       else

            INSERT INTO owns_items
                 VALUES (NEW.trainer_id, NEW.item_id, 'PURCHASED',
NEW.date_time);
             UPDATE trainer
             SET gold = newGold
             WHERE NEW.trainer_id = trainer.id;
             UPDATE sell
             SET stock = sell.stock - NEW.amount
               WHERE NEW.store_id = sell.store_id and NEW.item_id =
sell.item_id ;
       end if;


   end if;


   RETURN NEW;
END
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS insert_purchase ON purchase CASCADE;
CREATE TRIGGER insert_purchase AFTER INSERT ON purchase
FOR EACH ROW
EXECUTE FUNCTION modify_purchase();
```

### 7.8.2 Explanation

After an insert on the **purchase** table, we calculate the final_price by applying the discount thanks to the base_price of the Item and the amount the trainer has purchased it. Then, we update this value in the corresponding row with the new trainer_id, store_id, item_id and date_time. As a third step, we calculate the remaining gold the trainer will have.

If it's negative, it means the trainer cannot buy this, so we insert a message indicating this into the **warnings** table and delete that row from the **purchase** table. Otherwise, we calculate the remaining stock by computing the difference between the stock for a specific item and store, and the amount of items purchased. Again, if it's negative, we insert the corresponding message into **warnings** and delete that row in **purchase**. Otherwise, it will mean that we need to create an entry in owns_items, update trainer's gold and also store's stock.

### 7.8.3 Query validation

First case: Trainer doesn't have enough gold to do this purchase.

```
INSERT INTO purchase(store_id, item_id, trainer_id, amount,
discount, date_time)
VALUES (582,367,407,1,30,'2000-01-01');
```

Before doing the insert, Trainer with ID 407 has 2084 gold.

| | | | | | |
|---|---|---|---|---|---|
| 130 | 407 | Magma Geologist | Brenton | 823483 | 2084 |

A part of the purchase table looks like this.

| | | | | | |
|---|---|---|---|---|---|
| 4739 | 582 | 347 | 434 | 3 | 1000 |
| 4740 | 582 | 347 | 446 | 2 | 660 |
| 4741 | 582 | 347 | 459 | 2 | 1000 |
| 4742 | 582 | 367 | 104 | 2 | 50000 |
| 4743 | 582 | 367 | 106 | 1 | 39500 |
| 4744 | 582 | 367 | 188 | 2 | 50000 |
| 4745 | 582 | 367 | 224 | 2 | 40000 |
| 4746 | 582 | 367 | 273 | 6 | 50000 |
| 4747 | 582 | 367 | 407 | 9 | 50000 |
| 4748 | 582 | 367 | 457 | 2 | 50000 |

In this case, he wants to buy the item with ID 367 that costs 50000, so in this case the trainer has not enough gold to buy it.

| | | | | | |
|---|---|---|---|---|---|
| 201 | 367 | tm63 | 50000 | Teaches Embargo to a compat… | fa |

As a result, we see that it is not added in the purchase table.

| | | | | | |
|---|---|---|---|---|---|
| 4739 | 582 | 347 | 434 | 3 | 1000 |
| 4740 | 582 | 347 | 446 | 2 | 660 |
| 4741 | 582 | 347 | 459 | 2 | 1000 |
| 4742 | 582 | 367 | 104 | 2 | 50000 |
| 4743 | 582 | 367 | 106 | 1 | 39500 |
| 4744 | 582 | 367 | 188 | 2 | 50000 |
| 4745 | 582 | 367 | 224 | 2 | 40000 |
| 4746 | 582 | 367 | 273 | 6 | 50000 |
| 4747 | 582 | 367 | 407 | 9 | 50000 |
| 4748 | 582 | 367 | 457 | 2 | 50000 |

Also, it has appeared the message in the **warnings** table.

| | message |
|---|---|
| 1 | Trainer #407 tried to buy tm63 but his card has been declined, insufficient fu… |

Second case: Trainer has enough gold but there's not enough stock

```
INSERT INTO purchase(store_id, item_id, trainer_id, amount,
discount, date_time)
VALUES (561,67,230,2,0,'2000-01-03');
```

Trainer's gold is 8921.

| | | | | |
|---|---|---|---|---|
| 230 | Cowgirl | Ashlee | 936598 | 8921 |

Item's stock in store 561 is 1.

| | | | |
|---|---|---|---|
| 561 | 67 | 1 | 0 |

In the **warnings** table, it appears the corresponding message and if we want to look for this row in **purchase**, it doesn't exist.

| | message |
|---|---|
| 1 | Trainer #407 tried to buy tm63 but his card has been declined, insufficient fu… |
| 2 | Trainer #230 tried to buy red flute at Koelpin, Lindgren and Hintz store in Koelpin, Lindgren and Hintz, however |

postgres@localhost
console

| | | | | | |
|---|---|---|---|---|---|
| 438 | 561 | 67 | 137 | 1 | 20 |
| 439 | 561 | 67 | 367 | 5 | 20 |

Third case: Trainer has gold and there's enough stock

```
INSERT INTO purchase(store_id, item_id, trainer_id, amount,
discount, date_time)
VALUES (562,320,229,4,10,'2000-01-01');
```

Trainer gold is 11460:

| | | | | | |
|---|---|---|---|---|---|
| 8 | 229 | Guy | Vernie | 840327 | 11460 |

Store stock is 100:

| | | | | |
|---|---|---|---|---|
| 1 | 562 | 320 | 100 | 0 |

Base price of the item is 1000.

| | | | | | |
|---|---|---|---|---|---|
| 248 | 320 | tm16 | 1000 | Teaches Light Screen to a c… | f |

| | store_id | item_id | trainer_id | amount | final_price |
|---|---|---|---|---|---|
| 686 | 562 | 320 | 120 | 1 | 1000 |
| 687 | 562 | 320 | 127 | 2 | 820 |
| 688 | 562 | 320 | 219 | 4 | 1000 |
| 689 | 562 | 320 | 229 | 4 | 900 |

| | | | | | |
|---|---|---|---|---|---|
| 689 | 29 | 4 | 900 | 10 | 2000-01-01 00:00:00.000000 |

As we can see, the purchase is added in the corresponding table, and the final_price corresponds to the 10% discount of the base_price of the Item with ID 320.

Then, in **trainers** gold is updated to 7860 , and in **sell** the stock to 96.

| | | | | | |
|---|---|---|---|---|---|
| 308 | 229 | Guy | Vernie | 840327 | 7860 |

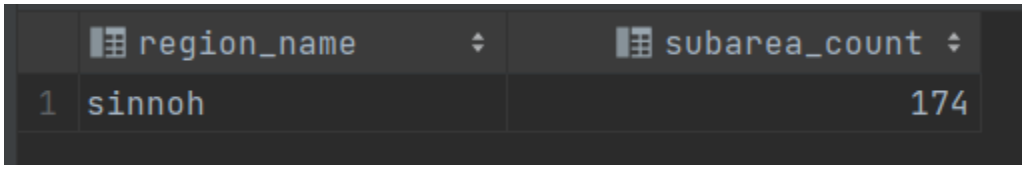
| | | | | |
|---|---|---|---|---|
| 562 | 320 | 96 | 0 | |

# 8 Time to Explore

## 8.1 Query 1

### 8.1.1 Solution 1

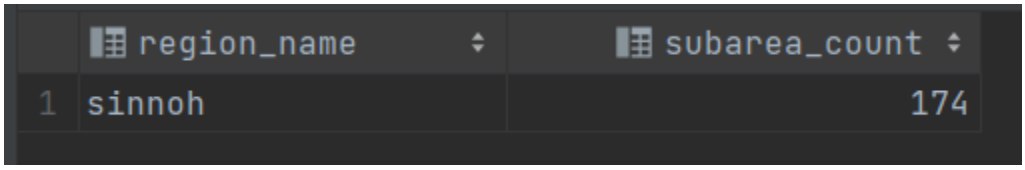
```
SELECT region.name AS region_name, COUNT(subarea.id) AS subarea_count
FROM region
JOIN area ON region.name = area.name_region
JOIN subarea ON area.id = subarea.id_area
GROUP BY region.name
ORDER BY subarea_count DESC
LIMIT 1;
```



| | region_name | subarea_count |
|---|---|---|
| 1 | sinnoh | 174 |

#### 8.1.2 Solution 2

```
SELECT region.name AS region_name, subarea_count
FROM region
JOIN (
   SELECT area.name_region, COUNT(subarea.id) AS subarea_count
   FROM area
   JOIN subarea ON area.id = subarea.id_area
   GROUP BY area.name_region
) AS subarea_counts ON region.name = subarea_counts.name_region
ORDER BY subarea_count DESC
LIMIT 1;
```



| | region_name | subarea_count |
|---|---|---|
| 1 | sinnoh | 174 |

#### 8.1.3 Explanation

In the case of the first query, it selects the name from the ***Region*** table and counts the number of ***Subareas*** using the COUNT function. It then joins the ***region*** table with the ***Area*** table based on the *name_region* column and it joins the ***area*** table with the ***subarea*** table based on the *id* and *id_area* columns, respectively. The result is grouped by *region.name* in order to make sure that the COUNT() function calculates the amount of ***subareas*** for each unique region. The ORDER BY clause orders the result in descending order based on the count of ***subareas*** so that, finally, the LIMIT 1 ensures that only the ***region*** with the highest count is returned.

As for the second query, it achieves the same result by using a subquery to calculate the count of ***subareas*** for each ***area*** before joining with the ***region*** table. The subquery in the JOIN clause calculates the count of ***subareas*** for each area and associates it with the *name_region* column.

#### 8.1.4 Query validation

The queries use some simple JOINS based on their relationships as location. The GROUP BY clause makes sure that all of the locations are specifically related to each ***region***. In addition, by performing a similar query we can see all of the different ***subareas*** related to each ***region*** and we can make sure that Sinnoh is the one with the most instances of ***subareas***.

| | region_name | subarea_count |
|---|---|---|
| 1 | sinnoh | 174 |
| 2 | kanto | 166 |
| 3 | hoenn | 142 |
| 4 | johto | 118 |

## 8.2 Query 2

### 8.2.1 Solution

```
SELECT s.name AS gym_name, a.name AS city_it_belongs_to,
      ar.name AS route_directly_connected, oa.name AS area_on_the_other_side
FROM subarea s
JOIN gym g ON s.id = g.id
JOIN area a ON s.id_area = a.id
JOIN city c ON a.id = c.id
JOIN connect_ co ON a.id = co.id_area
JOIN route r ON co.id_route = r.id
JOIN area ar ON ar.id = r.id
JOIN connect_ co2 ON r.id = co2.id_route
JOIN area oa ON oa.id = co2.id_area
WHERE a.name <> oa.name;
```

90 rows CSV

| | gym_name | city_it_belongs_to | route_directly_connected | area_on_the_other_side |
|---|---|---|---|---|
| 1 | 6th Indigo League Gym | saffron city | route 5 | cerulean city |
| 2 | 8th Sinnoh League Gym | sunyshore city | route 223 | sinnoh pokemon league |
| 3 | 4th Sinnoh League Gym | pastoria city | route 212 | hearthome city |
| 4 | 1st Johto League Gym | violet city | route 31 | dark cave |
| 5 | 3rd Indigo League Gym | vermilion city | route 6 | saffron city |
| 6 | 2nd Indigo League Gym | cerulean city | route 24 | route 25 |
| 7 | 2nd Sinnoh League Gym | eterna city | route 205 | eterna forest |
| 8 | 1st Hoenn League Gym | rustboro city | route 115 | meteor falls |
| 9 | 3rd Johto League Gym | goldenrod city | route 35 | national park |
| 10 | 4th Sinnoh League Gym | pastoria city | route 213 | valor lakefront |
| 11 | 2nd Sinnoh League Gym | eterna city | route 211 | mt coronet |
| 12 | 6th Johto League Gym | olivine city | route 39 | route 38 |
| 13 | 3rd Hoenn League Gym | mauville city | route 111 | route 113 |
| 14 | 1st Johto League Gym | violet city | route 36 | route 37 |
| 15 | 5th Sinnoh League Gym | hearthome city | route 209 | solaceon town |
| 16 | 3rd Hoenn League Gym | mauville city | route 118 | route 119 |
| 17 | 7th Indigo League Gym | cinnabar island | route 21 | pallet town |
| 18 | 1st Sinnoh League Gym | oreburgh city | route 207 | route 206 |
| 19 | 2nd Hoenn League Gym | dewford town | route 106 | route 105 |
| 20 | 4th Johto League Gym | ecruteak city | route 42 | mt mortar |

### 8.2.2 Explanation

What this query has in particular is that we need to do several JOINS including three times with the ***Area*** table. Since the ***Gyms*** are a generalization of ***Subareas***, we need to connect the table ***Subarea*** with the ***Area*** table and this one with the ***City*** table in order to get the city of each ***gym***. In order to get the ***Route*** of each ***City*** we need to do several JOINS including the table ***Connect_*** and another one with area (since ***Routes*** are a generalization of ***Areas***). Finally, so as to get the ***areas*** to the other side of the ***routes*** we need to do another connection with the tables ***Connect_*** and ***Area***. As for the WHERE

clause, it ensures that the area name is not the same as the other side area name and avoids duplications.

#### 8.2.3 Query validation

In order to validate this query we can take a look at the information given to us and corroborate some of the information. For example:

| | | | | | |
|---|---|---|---|---|---|
| Route 104 | Rustboro City | Petalburg City | | Route 105 | Grass |
| Route 105 | Route 104 | | | Route 106 | Water |
| Route 106 | Route 105 | | | Dewford Town | Water |
| Route 107 | | Route 108 | Dewford Town | | Water |
| Route 108 | | Route 109 | Route 107 | | Water |
| Route 109 | Slateport City | | Route 108 | | Water |
| Route 110 | Mauville City | Route 103 | | Slateport City | Grass |
| Route 111 | Route 113 | | Route 112 | Mauville City | Sand |
| Route 112 | | Route 111 | Lavaridge Town | | Sand |
| Route 113 | | Route 111 | Fallarbor Town | | Mud |
| Route 114 | | Fallarbor Town | | Meteor Falls | Rocks |
| Route 115 | Meteor Falls | | | Rustboro City | Gravel |
| Route 116 | | Rusturf Tunnel | Rustboro City | | Grass |
| Route 117 | | Mauville City | Verdanturf Town | | Snow |

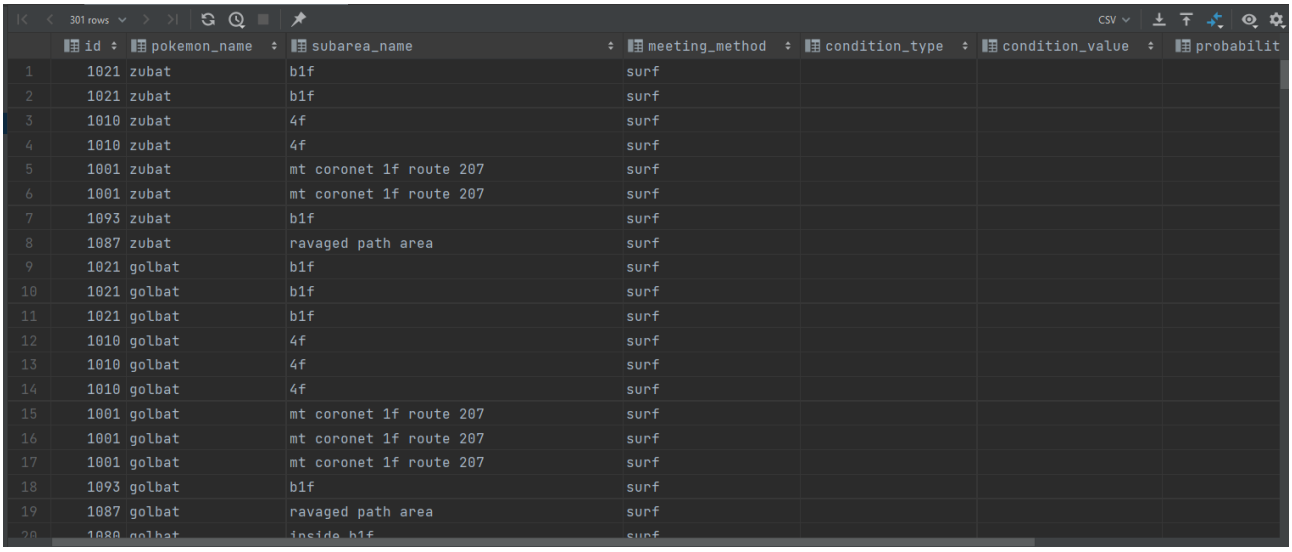
The 1st Hoenn League Gym is placed in Rustboro City. That city would be directly connected through a Route to Petalburg City, Route 105, Meteor Falls and Rusturf Tunnel. This is exactly what we get as a result from the query:

| | gym_name | city_it_belongs_to | route_directly_connected | area_on_the_other_side |
|---|---|---|---|---|
| 1 | 1st Hoenn League Gym | rustboro city | route 115 | meteor falls |
| 2 | 1st Hoenn League Gym | rustboro city | route 104 | petalburg city |
| 3 | 1st Hoenn League Gym | rustboro city | route 116 | rusturf tunnel |
| 4 | 1st Hoenn League Gym | rustboro city | route 104 | route 105 |

## 8.3 Query 3

#### 8.3.1 Solution

```
SELECT e.id, p.name AS pokemon_name, s.name AS subarea_name, e.meeting_method,
f.condition_type, f.condition_value, f.probability,
      f.min_level, f.max_level
FROM encounter e
JOIN subarea s ON e.subarea = s.id
JOIN area a ON s.id_area = a.id
JOIN region r ON a.name_region = r.name
JOIN find f ON f.id_encounter = e.id
JOIN species p ON f.id_species = p.id
WHERE r.name = 'sinnoh' AND e.meeting_method = 'surf';
```



| | id | pokemon_name | subarea_name | meeting_method | condition_type | condition_value | probabilit |
|---|---|---|---|---|---|---|---|
| 1 | 1021 | zubat | b1f | surf | | | |
| 2 | 1021 | zubat | b1f | surf | | | |
| 3 | 1010 | zubat | 4f | surf | | | |
| 4 | 1010 | zubat | 4f | surf | | | |
| 5 | 1001 | zubat | mt coronet 1f route 207 | surf | | | |
| 6 | 1001 | zubat | mt coronet 1f route 207 | surf | | | |
| 7 | 1093 | zubat | b1f | surf | | | |
| 8 | 1087 | zubat | ravaged path area | surf | | | |
| 9 | 1021 | golbat | b1f | surf | | | |
| 10 | 1021 | golbat | b1f | surf | | | |
| 11 | 1021 | golbat | b1f | surf | | | |
| 12 | 1010 | golbat | 4f | surf | | | |
| 13 | 1010 | golbat | 4f | surf | | | |
| 14 | 1010 | golbat | 4f | surf | | | |
| 15 | 1001 | golbat | mt coronet 1f route 207 | surf | | | |
| 16 | 1001 | golbat | mt coronet 1f route 207 | surf | | | |
| 17 | 1001 | golbat | mt coronet 1f route 207 | surf | | | |
| 18 | 1093 | golbat | b1f | surf | | | |
| 19 | 1087 | golbat | ravaged path area | surf | | | |
| 20 | 1080 | golbat | inside b1f | surf | | | |

The query result is too long to be able to show it completely in this report format.

### *8.3.2 Explanation*

The query uses several JOINS in order to relate the information of the ***encounters*** (which is separated between the table ***Encounter*** and the table ***Find***) with the ***region***. It joins the ***encounter*** table with the ***subarea*** table based on the id of the ***subarea***. It further joins the ***subarea*** table with the ***area*** table based on the id of the ***area***. It finally reaches the ***region*** by joining this last table with ***Region*** using the region name. Then it joins the ***find*** table with the ***encounter*** table based on the id of the encounter and the ***species*** of the pokemons based on their id as well. The WHERE clause filters the result to only include encounters in the 'Sinnoh' region (r.name = 'sinnoh') and with the meeting method set as 'surf' (e.meeting_method = 'surf').

### *8.3.3 Query validation*

We can use the canalave city area (subarea) as an example. Its id is the number 371. It has several surf encounters such as the following:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| pelipper | 371 | surf | | | 4 | 20 | 40 |
| tentacool | 371 | surf | | | 60 | 20 | 30 |
| tentacruel | 371 | surf | | | 5 | 20 | 40 |

We can also safely say that it belongs to the Sinnoh region.

| | | | |
|---|---|---|---|
| sinnoh | canalave city | canalave city area | 371 |

Finally, we can see that all of this information is included in the result of the query:

| | id | pokemon_name | subarea_name | meeting_method |
|---|---|---|---|---|
| 1 | 967 | tentacool | canalave city area | surf |
| 2 | 967 | tentacruel | canalave city area | surf |
| 3 | 967 | wingull | canalave city area | surf |
| 4 | 967 | pelipper | canalave city area | surf |
| 5 | 967 | pelipper | canalave city area | surf |
| 6 | 967 | shellos | canalave city area | surf |
| 7 | 967 | gastrodon | canalave city area | surf |

## 8.4 Query 4

### 8.4.1 Solution

```
SELECT e.id, p.name AS pokemon_name, f.probability, f.min_level
FROM encounter e
JOIN find f on e.id = f.id_encounter
JOIN species p on f.id_species = p.id
WHERE f.probability <= 1 AND f.min_level =
(SELECT f2.min_level FROM find f2 WHERE f2.probability <= 1
ORDER BY min_level ASC LIMIT 1);
```

| | id | pokemon_name | probability | min_level |
|---|---|---|---|---|
| 1 | 1160 | nidoran-m | 1 | 2 |
| 2 | 1160 | nidoran-m | 1 | 2 |
| 3 | 1161 | sentret | 1 | 2 |
| 4 | 1161 | sentret | 1 | 2 |
| 5 | 779 | wingull | 1 | 2 |
| 6 | 1161 | starly | 1 | 2 |
| 7 | 1160 | starly | 1 | 2 |
| 8 | 1161 | bidoof | 1 | 2 |
| 9 | 1160 | bidoof | 1 | 2 |

### 8.4.2 Explanation

We first need to join the tables containing all the information of the ***encounters*** with the ***species*** to be able to show the pokemon's *names*. The WHERE clause filters those results based on the ones that have a *probability* of 1 or lower and also that have a *min level* equal to the result of the subquery. The subquery finds the *minimum level* out of the encounters with 1% chance or lower.

### 8.4.3 Query validation

We can directly see the information in the data provided and we can see that all of the possible results are included in the query:

| pokemon | subareaID | method | condition_ | condition_value | chance% | min_level | max_level |
|---|---|---|---|---|---|---|---|
| togepi | 373 | gift | | | 100 | 1 | 1 |
| togepi | 232 | gift | | | 100 | 1 | 1 |
| mareep | 233 | gift | | | 100 | 1 | 1 |
| wooper | 233 | gift | | | 100 | 1 | 1 |
| slugma | 233 | gift | | | 100 | 1 | 1 |
| happiny | 518 | gift | | | 100 | 1 | 1 |
| riolu | 455 | gift | | | 100 | 1 | 1 |
| nidoran-m | 474 | walk | radar | on | 1 | 2 | 2 |
| nidoran-m | 474 | walk | radar | on | 1 | 2 | 2 |
| sentret | 475 | walk | radar | on | 1 | 2 | 2 |
| sentret | 475 | walk | radar | on | 1 | 2 | 2 |
| wingull | 300 | walk | | | 1 | 2 | 2 |
| starly | 474 | walk | radar | off | 1 | 2 | 2 |
| starly | 475 | walk | radar | off | 1 | 2 | 2 |
| bidoof | 474 | walk | radar | off | 1 | 2 | 2 |
| bidoof | 475 | walk | radar | off | 1 | 2 | 2 |
| growlithe | 474 | walk | item | pp up | 4 | 2 | 2 |
| growlithe | 474 | walk | item | sun stone | 4 | 2 | 2 |
| growlithe | 475 | walk | item | potion | 4 | 2 | 2 |
| growlithe | 475 | walk | item | twisted spoon | 4 | 2 | 2 |

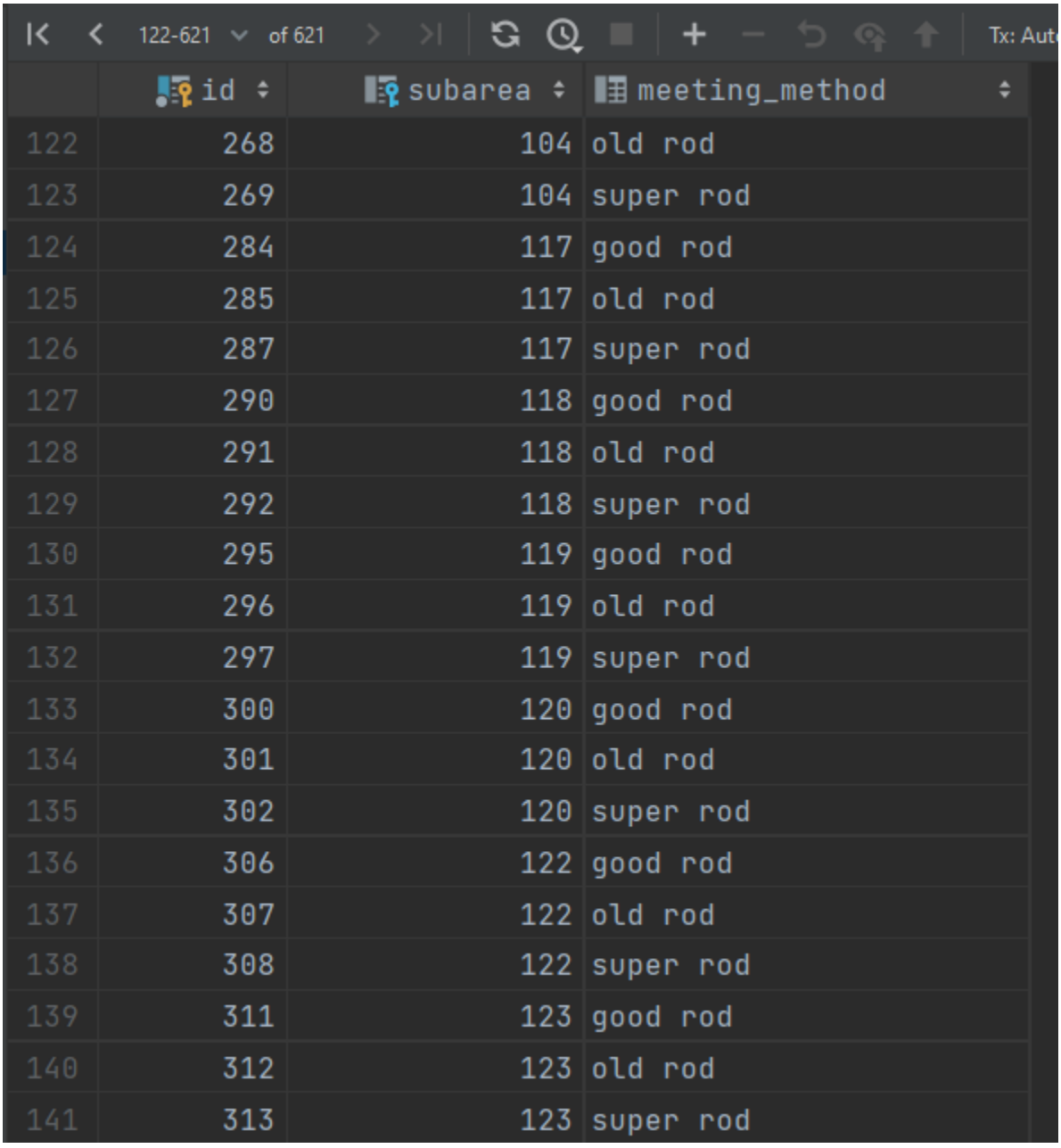
## 8.5 Query 5

### 8.5.1 Solution

```
UPDATE encounter
SET meeting_method =
   CASE
       WHEN meeting_method = 'good' THEN 'good rod'
       WHEN meeting_method = 'old' THEN 'old rod'
       WHEN meeting_method = 'super' THEN 'super rod'
   END
WHERE meeting_method IN ('good', 'old', 'super');

CREATE TEMPORARY TABLE available_methods AS
SELECT DISTINCT meeting_method
FROM encounter
WHERE meeting_method <> 'rock';

UPDATE encounter
SET meeting_method = (
   SELECT meeting_method
   FROM available_methods
   ORDER BY RANDOM()
   LIMIT 1
)
WHERE meeting_method = 'rock';
```



122-621 of 621 Tx: Aut

| | id | subarea | meeting_method |
|---|---|---|---|
| 122 | 268 | 104 | old rod |
| 123 | 269 | 104 | super rod |
| 124 | 284 | 117 | good rod |
| 125 | 285 | 117 | old rod |
| 126 | 287 | 117 | super rod |
| 127 | 290 | 118 | good rod |
| 128 | 291 | 118 | old rod |
| 129 | 292 | 118 | super rod |
| 130 | 295 | 119 | good rod |
| 131 | 296 | 119 | old rod |
| 132 | 297 | 119 | super rod |
| 133 | 300 | 120 | good rod |
| 134 | 301 | 120 | old rod |
| 135 | 302 | 120 | super rod |
| 136 | 306 | 122 | good rod |
| 137 | 307 | 122 | old rod |
| 138 | 308 | 122 | super rod |
| 139 | 311 | 123 | good rod |
| 140 | 312 | 123 | old rod |
| 141 | 313 | 123 | super rod |

And if we search based on the meeting_method ‘rock’ we would get 0 results.

#### *8.5.2 Explanation*

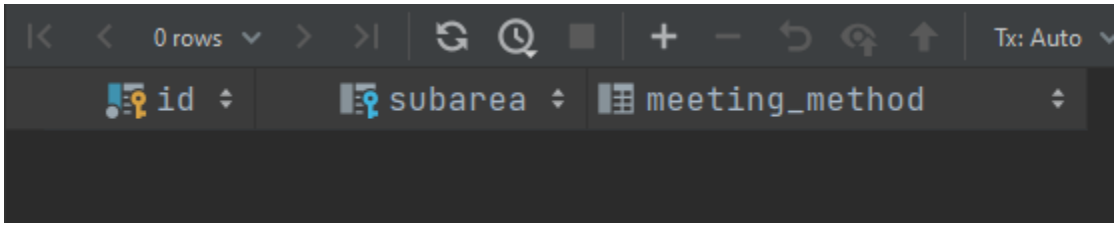
First what we simply do is replace all of the mentions of ‘old’, ‘good’ or ‘super’ with ‘old rod’, ‘good rod’ and ‘super rod’ respectively by creating an UPDATE that takes into account all of those cases. Then, we create a temporary table which contains all of the *meeting_methods* except ‘rock’. This table is then used in the subquery of the second UPDATE in order to randomly select one of the other methods and insert it wherever the method ‘rock’ appears.

#### *8.5.3 Query validation*

Firstly, we can see that if we do the following query:

```
SELECT * FROM encounter WHERE meeting_method = 'rock';
```

We get 0 results.




Secondly, we can also take a look at some of the instances in which the ‘rock’ method was used and see that they have all now changed to the same new method.

| | id | subarea | meeting_method |
|---|---|---|---|
| 1 | 34 | 14 | rock |
| 2 | 156 | 51 | rock |
| 3 | 162 | 52 | rock |
| 4 | 169 | 53 | rock |

| | id | subarea | meeting_method |
|---|---|---|---|
| 1 | 34 | 14 | walk |
| 2 | 156 | 51 | walk |
| 3 | 162 | 52 | walk |
| 4 | 169 | 53 | walk |

## 8.6 Query 6

### 8.6.1 Solution

```
SELECT DISTINCT a.name, r.pavement, f.condition_value
FROM area a
JOIN subarea s ON a.id = s.id_area
JOIN encounter e ON e.subarea = s.id
JOIN find f ON e.id = f.id_encounter
JOIN route r ON a.id = r.id
WHERE r.pavement = 'Grass' AND (f.condition_value like 'rain' OR
f.condition_value like 'night ');
```

| | name | pavement | condition_value |
|---|---|---|---|
| 1 | route 1 | Grass | night |
| 2 | route 14 | Grass | night |
| 3 | route 16 | Grass | night |
| 4 | route 201 | Grass | night |
| 5 | route 204 | Grass | night |
| 6 | route 209 | Grass | night |
| 7 | route 212 | Grass | night |
| 8 | route 26 | Grass | night |
| 9 | route 28 | Grass | night |
| 10 | route 34 | Grass | night |
| 11 | route 39 | Grass | night |
| 12 | route 47 | Grass | night |
| 13 | route 47 | Grass | rain |
| 14 | route 5 | Grass | night |
| 15 | route 7 | Grass | night |

#### *8.6.2 Explanation*

In this query we use the DISTINCT clause since there we would have had repeated values otherwise. We then use several JOINS to be able to relate all the information from the ***encounters*** to the ***routes***. The WHERE clauses make sure that we only take into consideration those ***Routes*** that have 'grass' *pavement* and only those encounters that happen at night or while it is raining.

#### *8.6.3 Query validation*

We can see that there are a couple of instances with the id_encounter 553 that happen during rainy conditions:

| | | | | | |
|---|---|---|---|---|---|
| 47 | 9605 | 553 | 223 | weather | rain |
| 48 | 6926 | 553 | 129 | weather | rain |

This encounters happen in a route where the pavement is snow:

| | id | pavement |
|---|---|---|
| 1 | 348 | Snow |

However, if we were to change it to Grass we would be able to see that those encounters are added to our results (although only one result is added since we don't want repeated results)

```
UPDATE route
SET pavement = 'Grass'
WHERE id = 348;
```

16 rows

| | name | pavement | condition_value |
|---|---|---|---|
| 1 | route 1 | Grass | night |
| 2 | route 14 | Grass | night |
| 3 | route 16 | Grass | night |
| 4 | route 201 | Grass | night |
| 5 | route 204 | Grass | night |
| 6 | route 209 | Grass | night |
| 7 | route 212 | Grass | night |
| 8 | route 26 | Grass | night |
| 9 | route 28 | Grass | night |
| 10 | route 34 | Grass | night |
| 11 | route 39 | Grass | night |
| 12 | route 44 | Grass | rain |
| 13 | route 47 | Grass | night |
| 14 | route 47 | Grass | rain |
| 15 | route 5 | Grass | night |
| 16 | route 7 | Grass | night |

## 8.7 Trigger 1

### 8.7.1 Solution

```
CREATE OR REPLACE FUNCTION add_city_info_function()
RETURNS TRIGGER AS $$
DECLARE
 region_name VARCHAR(255);
 gym_name VARCHAR(255);
 gym_leader_name VARCHAR(255);
 gym_leader_id INT;
 experience INT;
 gold INT;
 gym_medal VARCHAR(255);
 item_id INT;
 type_id INT;
   city_name VARCHAR(255);
```

```
    subarea_id INT;
BEGIN
 -- Check if the region Lasalia exists
 SELECT name INTO region_name FROM region WHERE name = 'Lasalia';

 -- If not, create it
 IF region_name IS NULL THEN
   INSERT INTO region (name) VALUES ('Lasalia') RETURNING name INTO region_name;
 END IF;

  -- Generate random values
 city_name := 'City ' || (floor(random() * 1000) + 1)::text;
 gym_name := (floor(random() * 1000) + 1)::text || ' Gym';
 gym_medal := (floor(random() * 1000) + 1)::text || ' Badge';
 gym_leader_name := 'Gym Leader ' || (floor(random() * 1000) + 1)::text;
 experience := floor(random() * 1000) + 1;
 gold := floor(random() * 10000) + 1;

 -- Random item
 SELECT id FROM item ORDER BY random() LIMIT 1 INTO item_id;

 -- Random type
 SELECT id FROM type ORDER BY random() LIMIT 1 INTO type_id;

 -- Insert the city into area
  INSERT INTO area (id, name, name_region) VALUES (NEW.id, city_name,
region_name);

 -- Get the next value from the subareas
 DROP SEQUENCE IF EXISTS sub_area_id_seq;
 CREATE SEQUENCE sub_area_id_seq;
 PERFORM setval('sub_area_id_seq', (SELECT MAX(id) FROM subarea));
 SELECT nextval('sub_area_id_seq') INTO subarea_id;

 -- Create the subarea associated with the gym
 INSERT INTO subarea (id, name, id_area) VALUES (subarea_id, gym_name, NEW.id);

 -- Create the gym associated with the city
 INSERT INTO gym (id, medal, id_item, id_type)
 VALUES (subarea_id, gym_medal, item_id, type_id);

 -- Get the next value from the trainers
 DROP SEQUENCE IF EXISTS trainer_id_seq;
 CREATE SEQUENCE trainer_id_seq;
 PERFORM setval('trainer_id_seq', (SELECT MAX(id) FROM trainer));
 SELECT nextval('trainer_id_seq') INTO gym_leader_id;

 -- Insert the gym leader into the trainer table
   INSERT INTO trainer (id, class, name, experience, gold) VALUES
(gym_leader_id,'Gym Leader',gym_leader_name, experience, gold);

 -- Create the gym leader associated with the gym
 INSERT INTO gym_leader (id, gym) VALUES (gym_leader_id, subarea_id);

 RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER add_city_info_trigger
BEFORE INSERT ON city
FOR EACH ROW
EXECUTE FUNCTION add_city_info_function();
```

### 8.7.2 Explanation

The trigger first checks whether the Lasalia ***region*** already exists, if it does not, it creates it. Then, we generate random values for the city name and everything related to the ***Gym*** and to the ***Gym_leader***. These will compose the full information regarding the ***city*** in the different tables. Later, we have to start inserting into each corresponding table but it has to be in order so as not to mess up any foreign keys.We first, insert the information of the ***city*** into ***area*** since cities are a generalization of areas. Then we grab the next possible value in the subarea's *id* using a sequence and we add an instance of a ***subarea*** which will be used for the ***Gym***. Again, ***gym*** represent a generalization of ***subarea*** so we need to add it there first. Finally, we repeat this process but for the ***trainer*** and ***gym_leader*** since ***gym_leader*** is a generalization from ***trainer***.

### 8.7.3 Query validation

The trigger should have an effect everytime we look to insert a new ***city***. Since we need to make sure that there is an area associated to the ***city***, the trigger will take palace before the insert. We can validate the trigger by doing the following insert:

```
INSERT INTO city (id, population) VALUES (400, 100000);
```

If we check all the information from ***city*** where the *id* is 400. We get the following information.

| | id | population |
|---|---|---|
| 1 | 400 | 100000 |

If we check the information of the ***area*** where the *id* is 400.

| | id | name | name_region |
|---|---|---|---|
| 1 | 400 | City 561 | Lasalia |

If we check the information about ***subarea*** where the *id_area* is 400.

| | id | name | id_area |
|---|---|---|---|
| 1 | 651 | 566 Gym | 400 |

If we check the information of the ***gym_leader*** whose gym is 651.

| | id | gym |
|---|---|---|
| 1 | 537 | 651 |

Finally, if we see the information of the ***trainer*** whose id is 537.

| | id | class | name | experience | gold |
|---|---|---|---|---|---|
| 1 | 537 | Gym Leader | Gym Leader 387 | 942 | 9461 |

## 8.8 Trigger 2

### 8.8.1 Solution

```
CREATE TABLE IF NOT EXISTS Warnings (
    id SERIAL PRIMARY KEY,
    trainer_id INT,
    species_name VARCHAR(255),
    route_name VARCHAR(255),
    message VARCHAR(255),
    date_obtained DATE
);

CREATE OR REPLACE FUNCTION check_new_pokemon()
RETURNS TRIGGER AS $$
DECLARE
    pavement_type VARCHAR(255);
    pokemon_min_level INT;
    pokemon_max_level INT;
    capture_chance INT;
    trainer_name VARCHAR(255);
    species_name VARCHAR(255);
    route_name VARCHAR(255);
BEGIN
    SELECT r.pavement INTO pavement_type
    FROM route r JOIN subarea s ON s.id_area = r.id
    WHERE s.id = NEW.subarea;

    SELECT p.name INTO species_name
    FROM species p WHERE p.id = NEW.species;

    SELECT t.name INTO trainer_name
    FROM trainer t WHERE t.id = NEW.trainer;

    SELECT f.min_level, f.max_level, f.probability
    INTO pokemon_min_level, pokemon_max_level, capture_chance
    FROM find f
    JOIN encounter e ON f.id_encounter = e.id
    WHERE f.id_species = NEW.species AND e.subarea = NEW.subarea
    LIMIT 1;

    SELECT a.name INTO route_name FROM area a
    JOIN subarea s ON a.id = s.id_area WHERE s.id = NEW.subarea;

    IF (pavement_type = 'water' AND NEW.method <> 'surf') THEN
            INSERT INTO Warnings (trainer_id, species_name, route_name, message,
date_obtained)
        VALUES (NEW.trainer, species_name, route_name,
            CONCAT('Trainer ', trainer_name, ' captured ', species_name,
            ' in ', route_name,
                    ' with an incorrect method, level, or capture chance.'),
NEW.date_time);
    ELSIF (pavement_type <> 'water' AND (NEW.method <> 'walk' AND NEW.method <>
'headbutt')) THEN
            INSERT INTO Warnings (trainer_id, species_name, route_name, message,
date_obtained)
        VALUES (NEW.trainer, species_name, route_name,
```

```
            CONCAT('Trainer ', trainer_name, ' captured ', species_name,
            ' in ', route_name,
                    ' with an incorrect method, level, or capture chance.'),
NEW.date_time);
    ELSIF (NEW.level < pokemon_min_level OR NEW.level > pokemon_max_level OR
capture_chance <= 0) THEN
          INSERT INTO Warnings (trainer_id, species_name, route_name, message,
date_obtained)
       VALUES (NEW.trainer, species_name, route_name,
            CONCAT('Trainer ', trainer_name, ' captured ', species_name,
            ' in ', route_name,
                    ' with an incorrect method, level, or capture chance.'),
NEW.date_time);
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_new_pokemon_trigger
AFTER INSERT ON pokemon_creature
FOR EACH ROW
EXECUTE FUNCTION check_new_pokemon();
```

### 8.8.2 Explanation

Firstly, we need to create the table ***Warnings*** which will contain the corresponding information of the wrong insert into the ***pokemon_cretaure*** table. We then do a series of queries to obtain the information regarding the pavement of the ***route*** where the pokemon was obtained, the name of its ***species***, the name of the ***trainer***, and the min level, *max level* and chance of it being obtained in that ***subarea***. All of these queries are performed with the corresponding information of the newly obtained pokemon. After this, we do the corresponding checks and if at any moment the information inserted does not correspond with the conditions stated then we add a new warning. The conditions include, that if it was captured in water it has to have happened while surfing, if it isn't water it should have been with the headbutt or walking methods and the level should be between the min and max levels and with a probability higher than 0%.

### 8.8.3 Query validation

If we perform the following insertion:

```
INSERT INTO pokemon_creature (id, species, nickname, trainer, level, experience,
gender, nature, date_time, subarea, method, pokeball, is_baby, position, hp,
status_condition, team, attack, defense, special_attack, special_defense, speed,
evasion)
VALUES (13713, 16, 'bird', 24, 23, 5423, 'male', 23, '2021-08-29
04:24:54.000000', 22, 'surf', 3, false, null, 70, null, null,
100,100,100,100,100,100);
```

We get the following result in the ***Warnings*** table:

| | id | trainer_id | species_name | route_name | message | date_obtained |
|---|---|---|---|---|---|---|
| 1 | 1 | 24 | pidgey | route 15 | Trainer Wally captured pidgey in route 15 with an inco… | 2021-08-29 |

In here we added a pokemon caught in a route which pavement is dirt but we use the surf method so that its why it has been added to the warnings table.

| | id | name | id_area |
|---|---|---|---|
| 1 | 22 | route 15 area | 285 |

| | id | pavement |
|---|---|---|
| 1 | 285 | Dirt |

If the do another test:

```
INSERT INTO pokemon_creature (id, species, nickname, trainer, level, experience, gender, nature, date_time, subarea, method, pokeball, is_baby, position, hp, status_condition, team, attack, defense, special_attack, special_defense, speed, evasion)
VALUES (13715, 98, 'crab', 24, 23, 5423, 'male', 23, '2021-08-29 04:24:54.000000', 193, 'walk', 3, false, null, 70, null, null, 100,100,100,100,100,100);
```

We get the following result in the ***Warnings*** table:

| | r_id | species_name | route_name | message |
|---|---|---|---|---|
| 1 | 24 | pidgey | route 15 | Trainer Wally captured pidgey in route 15 with an incorrect method, level, or capture chance. |
| 2 | 24 | krabby | route 40 | Trainer Wally captured krabby in route 40 with an incorrect method, level, or capture chance. |

In this case the pavement is water but we have used the method walk.

| | id | name | id_area |
|---|---|---|---|
| 1 | 193 | route 40 area | 22 |

| | id | pavement |
|---|---|---|
| 1 | 22 | Water |

And if we do another test:

```
INSERT INTO pokemon_creature (id, species, nickname, trainer, level, experience, gender, nature, date_time, subarea, method, pokeball, is_baby, position, hp, status_condition, team, attack, defense, special_attack, special_defense, speed, evasion)
VALUES (13716, 72, 'octopus', 24, 25, 5423, 'male', 23, '2021-08-29 04:24:54.000000', 193, 'surf', 3, false, null, 70, null, null, 100,100,100,100,100,100);
```

We get this result:

| | id | trainer_id | species_name | route_name | message | date_obtained |
|---|---|---|---|---|---|---|
| 1 | 1 | 24 | pidgey | route 15 | Trainer Wally captured pidgey in route 15 with an inco… | 2021-08-29 |
| 2 | 2 | 24 | krabby | route 40 | Trainer Wally captured krabby in route 40 with an inco… | 2021-08-29 |
| 3 | 3 | 24 | tentacool | route 40 | Trainer Wally captured tentacool in route 40 with an i… | 2021-08-29 |

In this case it is the same area as before so it satisfies the condition of ir being water and the method being surf but the level surpasses the max level.

| | id | id_encounter | id_species | condition_type | condition_value | probability | min_level | max_level |
|---|---|---|---|---|---|---|---|---|
| 1 | 3526 | 536 | 72 | | | 60 | 20 | 24 |
| 2 | 3527 | 536 | 72 | | | 30 | 15 | 19 |
| 3 | 3706 | 536 | 73 | | | 10 | 20 | 24 |

# 9 Cross queries.

## 9.1 Query 1

### 9.1.1 Solution

```
SELECT r.name
   FROM region r JOIN criminal_organization co ON r.name = co.region
   JOIN evil_criminal ec ON co.name = ec.criminal_organization
   JOIN team tc ON tc.trainer = ec.id JOIN battle b ON tc.id = b.winner
   JOIN team t ON t.id = b.loser
   GROUP BY r.name
   ORDER BY COUNT(t.id) DESC
   LIMIT 1;
```

| | name |
|---|---|
| 1 | hoenn |

### 9.1.2 Explanation

To find out the region that is most affected by a criminal region we start by joining the tables ***Region*** and ***Criminal_Organization*** (linking the *region* attribute of the criminal organization to the region's id). Then, to find all the battles that criminals have won we join the ***Evil_Criminal***'s ***Team*** with the *winner* attribute of the ***Battle*** entity (by means of two *JOIN*s). Once we have these battles, we link them with the ***Teams*** who lost (*loser* attribute of the ***Battle*** table). This gives us the list of all the trainers defeated by criminals. To count how many happened in each area, we use a *GROUP BY* clause and merge all the rows of trainers defeated in the same region. Then, by means of the *COUNT()* operator, we are able to check how many defeated trainers each region had. Lastly, to show only the one that had the most criminal activity, we use the *ORDER BY* command to sort them by the number of defeated trainers and use the *LIMIT 1* operation to get only one row.

### 9.1.3 Query validation

Fristly, if we remove the *LIMIT* on the previous query and also select the *COUNT(t.id)* we use to sort the result, we can check the number of defeated trainers in each region:

| | name | count |
|---|---|---|
| 1 | hoenn | 53 |
| 2 | johto | 33 |
| 3 | kanto | 21 |
| 4 | sinnoh | 12 |

As you can see, we only get four regions in which criminal members defeated other trainers. To check if this makes sense, we can look at the ***Criminal_Organization*** table and count how many different regions they can be found in. As you can see below, there are only four regions in which the organizations are located. Therefore, the result of our query is consistent.

| | name | building | base_location | region |
|---|---|---|---|---|
| 1 | Plasma | N's Castle | 3 | kanto |
| 2 | Rocket | Silph Co. | 89 | kanto |
| 3 | Galactic | Galactic Veilstone Building | 142 | sinnoh |
| 4 | Magma | Magma Hideout | 224 | hoenn |
| 5 | Aqua | Aqua Hideout | 224 | hoenn |
| 6 | Cipher | Souvenir Shop | 351 | johto |

We will now add a new battle in which a criminal from an organization located at 'johto' region defeats another trainer and see how this affects the output of our query. As you can see in the table above, the 'Cipher' criminal organization is the one located in the 'johto' region. Below you can see that one of the evil criminals that belongs to this organization is the one with *id* = 434.

| id | money_steal | sig... | is_leader | sidekick | criminal_organization |
|---|---|---|---|---|---|
| 434 | 139 | I mean, sa… | false | 427 | Cipher |

We will execute the following query to insert a new battle in which the criminal with *id* = 434 defeats the trainer with *id* = 0 by first checking their respective teams in the ***Team*** table:

| id | trainer |
|---|---|
| 196 | 0 |

| id | trainer |
|---|---|
| 527 | 434 |

```
INSERT INTO battle(id, winner, loser) VALUES (700, 527, 196);
```

If we execute the query again, we get the following output:

| name | count |
|---|---|
| hoenn | 53 |
| johto | 34 |
| kanto | 21 |
| sinnoh | 12 |

As you can see, the 'johto' region now has increased its defeated trainers by one unit. Meaning that the way we count the defeated trainers of each region is valid.

## *9.2 Query 2*

### *9.2.1 Solution*

```
SELECT pc.id, s.name AS species_name, a.name AS area_name
FROM pokemon_creature pc
JOIN species s on pc.species = s.id
JOIN subarea sa on pc.subarea = sa.id
JOIN area a ON sa.id_area = a.id
WHERE sa.id NOT IN (SELECT sa2.id FROM subarea sa2
   JOIN encounter e on pc.subarea = e.subarea
   JOIN find f on e.id = f.id_encounter
   JOIN species s2 on f.id_species = s2.id
   WHERE s2.id = s.id);
```

#### 9.2.2 Explanation

The essential logic behind the query is that we want to find those pokemon that have been captured in an ***area*** where it would have been impossible to find them there. Basically, in the subquery we find all the possible ***subareas*** where each pokemon could be found. Once we have that, we try and find all the pokemon caught in a ***subarea*** that did not exist in the possibilities of the ***encounters***. We basically compare if the ***subarea*** where the pokemon was caught with the ***subareas*** that exist in the ***encounters***. We took this approach since we noticed that there are no null values in the ***probability*** column.

#### 9.2.3 Query validation

Initially, the query returned no results. However once we perform a insert we can see that the query works.

```
INSERT INTO pokemon_creature (id, species, nickname, trainer, level, experience,
gender, nature, date_time, subarea, method, pokeball, is_baby, position, hp,
status_condition, team, attack, defense, special_attack, special_defense, speed,
evasion)
VALUES (13760, 460, 'Pedro', 1, 40, 150664, 'male', 18, '2022-02-01
07:58:51.000000', 41, 'walk', 450, false, null, 506, null, null,
50,50,50,50,50,50);
```

In this case, we are inserting the pokemon abomasnow that has been caught in subarea 41 when its only possible subareas where it can be caught are 385 and 386.

| | id | species_name | area_name |
|---|---|---|---|
| 1 | 13760 | abomasnow | route 5 |

If we were to change the area where it was caught from subarea 41 to 385 then it will not be included in the results.

### 9.3 Query 3

#### 9.3.1 Solution

#### 9.3.2 Explanation

#### 9.3.3 Query validation

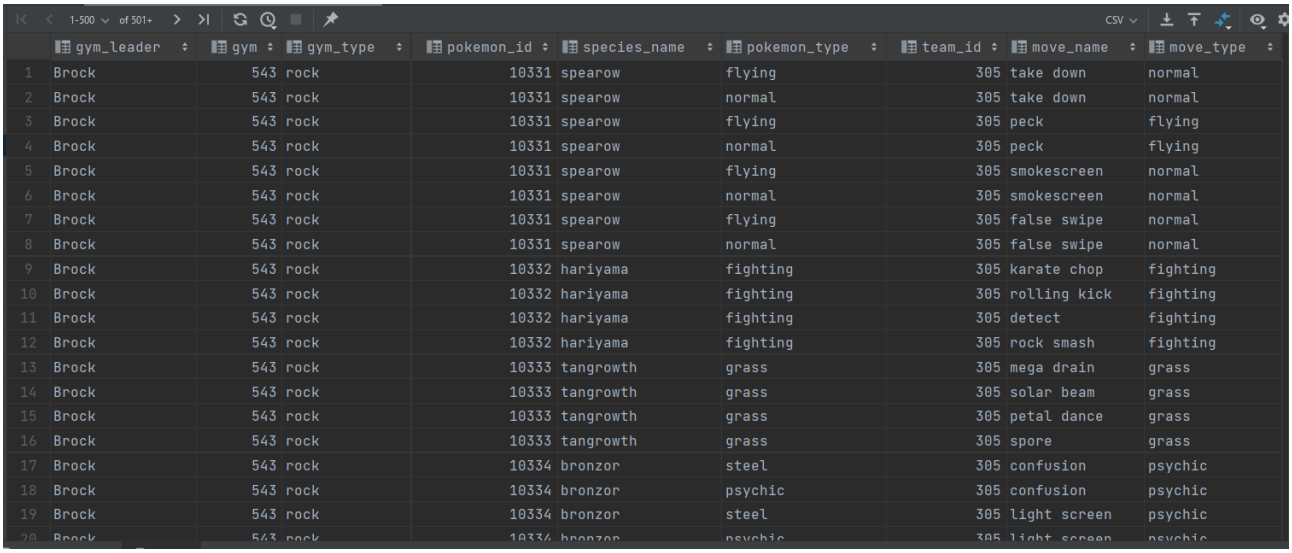
We faced some problems when doing this trigger and didn't have enough time to fix it.

### 9.4 Query 4

#### 9.4.1 Solution

```
SELECT tr.name AS gym_leader, gl.gym AS gym, t.name AS gym_type,
      p.id AS pokemon_id, s.name AS species_name, ht.type_name AS pokemon_type,
      tm.id AS team_id, m.name AS move_name, m.type AS move_type
FROM gym_Leader gl
JOIN gym g ON gl.gym = g.id
JOIN trainer tr ON gl.id = tr.id
JOIN team tm ON tr.id = tm.trainer
JOIN pokemon_Creature p ON tm.id = p.team
JOIN species s ON p.species = s.id
```

```
JOIN has_type ht ON p.species = ht.id_species
JOIN type t ON g.id_type = t.id
JOIN cause_or_receive cr ON p.id = cr.creature
JOIN move m ON cr.move = m.id
WHERE NOT EXISTS (
   SELECT 1
   FROM move mv
   WHERE mv.type = t.name
   AND mv.id = m.id
)
AND NOT EXISTS (
   SELECT 1
   FROM has_type ht2
   WHERE ht2.id_species = p.species
   AND ht2.type_name = t.name
)
ORDER BY gl.id, p.id, m.id;
```



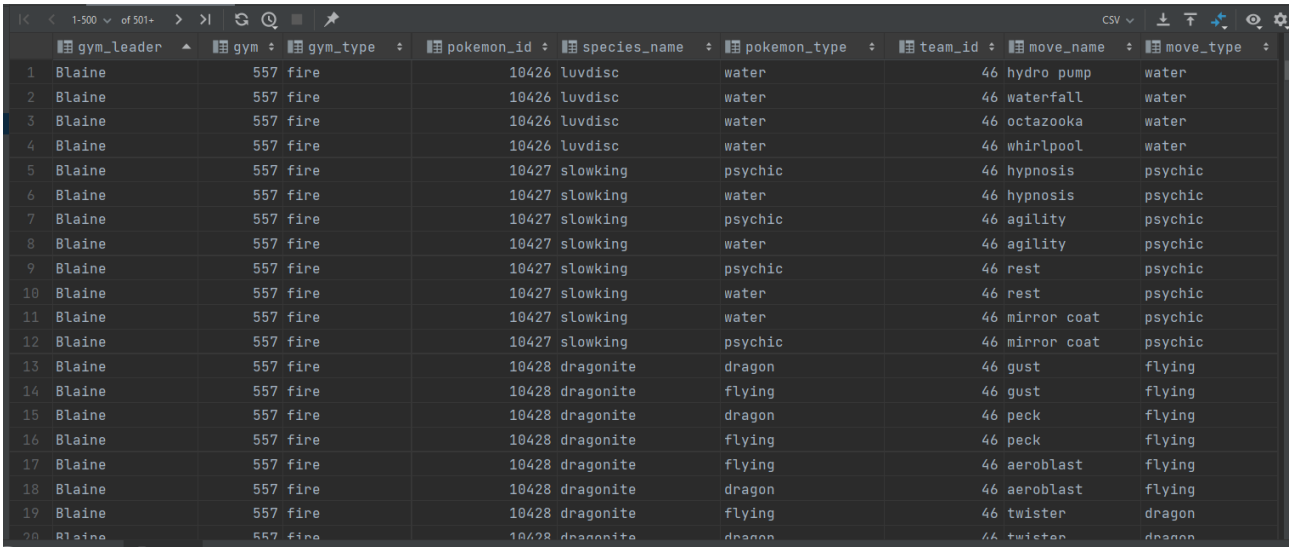
| | gym_leader | gym | gym_type | pokemon_id | species_name | pokemon_type | team_id | move_name | move_type |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Brock | 543 | rock | 10331 | spearow | flying | 305 | take down | normal |
| 2 | Brock | 543 | rock | 10331 | spearow | normal | 305 | take down | normal |
| 3 | Brock | 543 | rock | 10331 | spearow | flying | 305 | peck | flying |
| 4 | Brock | 543 | rock | 10331 | spearow | normal | 305 | peck | flying |
| 5 | Brock | 543 | rock | 10331 | spearow | flying | 305 | smokescreen | normal |
| 6 | Brock | 543 | rock | 10331 | spearow | normal | 305 | smokescreen | normal |
| 7 | Brock | 543 | rock | 10331 | spearow | flying | 305 | false swipe | normal |
| 8 | Brock | 543 | rock | 10331 | spearow | normal | 305 | false swipe | normal |
| 9 | Brock | 543 | rock | 10332 | hariyama | fighting | 305 | karate chop | fighting |
| 10 | Brock | 543 | rock | 10332 | hariyama | fighting | 305 | rolling kick | fighting |
| 11 | Brock | 543 | rock | 10332 | hariyama | fighting | 305 | detect | fighting |
| 12 | Brock | 543 | rock | 10332 | hariyama | fighting | 305 | rock smash | fighting |
| 13 | Brock | 543 | rock | 10333 | tangrowth | grass | 305 | mega drain | grass |
| 14 | Brock | 543 | rock | 10333 | tangrowth | grass | 305 | solar beam | grass |
| 15 | Brock | 543 | rock | 10333 | tangrowth | grass | 305 | petal dance | grass |
| 16 | Brock | 543 | rock | 10333 | tangrowth | grass | 305 | spore | grass |
| 17 | Brock | 543 | rock | 10334 | bronzor | steel | 305 | confusion | psychic |
| 18 | Brock | 543 | rock | 10334 | bronzor | psychic | 305 | confusion | psychic |
| 19 | Brock | 543 | rock | 10334 | bronzor | steel | 305 | light screen | psychic |

### 9.4.2 Explanation

In order to perform this query we need to relate a lot of different tables. We start at ***gym_leader*** and we go on to find their pokemon that belong to a ***team*** and which ***moves*** do they have. The central part of the query are the subqueries used in the WHERE clause. Both conditions in the first subquery aim to relate first the ***species*** with their ***type*** and then it makes sure that the *gym type* and the *pokemon type* match. With this we filter out rows where a match exists in the ***has_type*** table for the current pokemon ***species*** and the gym type and we end up displaying only those pokemon that do not share the *gym's type*. We do something similar in the second subquery where we aim to exclude every move that has the same *type* as the gym.

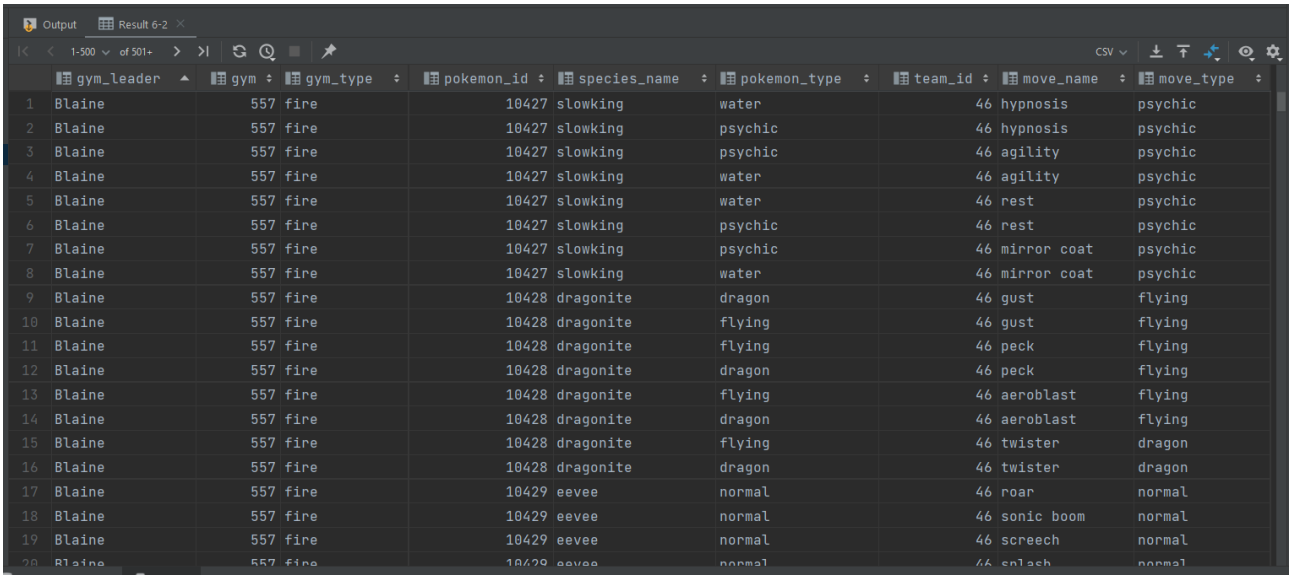
### 9.4.3 Query validation

For example, we can see that Blaine has a luvdisc in his team which is water type. But if we changed that luvdisc's type to fire then it will not appear:



| | gym_leader | gym | gym_type | pokemon_id | species_name | pokemon_type | team_id | move_name | move_type |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Blaine | 557 | fire | 10426 | luvdisc | water | 46 | hydro pump | water |
| 2 | Blaine | 557 | fire | 10426 | luvdisc | water | 46 | waterfall | water |
| 3 | Blaine | 557 | fire | 10426 | luvdisc | water | 46 | octazooka | water |
| 4 | Blaine | 557 | fire | 10426 | luvdisc | water | 46 | whirlpool | water |
| 5 | Blaine | 557 | fire | 10427 | slowking | psychic | 46 | hypnosis | psychic |
| 6 | Blaine | 557 | fire | 10427 | slowking | water | 46 | hypnosis | psychic |
| 7 | Blaine | 557 | fire | 10427 | slowking | psychic | 46 | agility | psychic |
| 8 | Blaine | 557 | fire | 10427 | slowking | water | 46 | agility | psychic |
| 9 | Blaine | 557 | fire | 10427 | slowking | psychic | 46 | rest | psychic |
| 10 | Blaine | 557 | fire | 10427 | slowking | water | 46 | rest | psychic |
| 11 | Blaine | 557 | fire | 10427 | slowking | water | 46 | mirror coat | psychic |
| 12 | Blaine | 557 | fire | 10427 | slowking | psychic | 46 | mirror coat | psychic |
| 13 | Blaine | 557 | fire | 10428 | dragonite | dragon | 46 | gust | flying |
| 14 | Blaine | 557 | fire | 10428 | dragonite | flying | 46 | gust | flying |
| 15 | Blaine | 557 | fire | 10428 | dragonite | dragon | 46 | peck | flying |
| 16 | Blaine | 557 | fire | 10428 | dragonite | flying | 46 | peck | flying |
| 17 | Blaine | 557 | fire | 10428 | dragonite | flying | 46 | aeroblast | flying |
| 18 | Blaine | 557 | fire | 10428 | dragonite | dragon | 46 | aeroblast | flying |
| 19 | Blaine | 557 | fire | 10428 | dragonite | flying | 46 | twister | dragon |
| 20 | Blaine | 557 | fire | 10428 | dragonite | dragon | 46 | twister | dragon |

```
UPDATE has_type
SET type_name = 'fire'
WHERE id_species = 370;
```

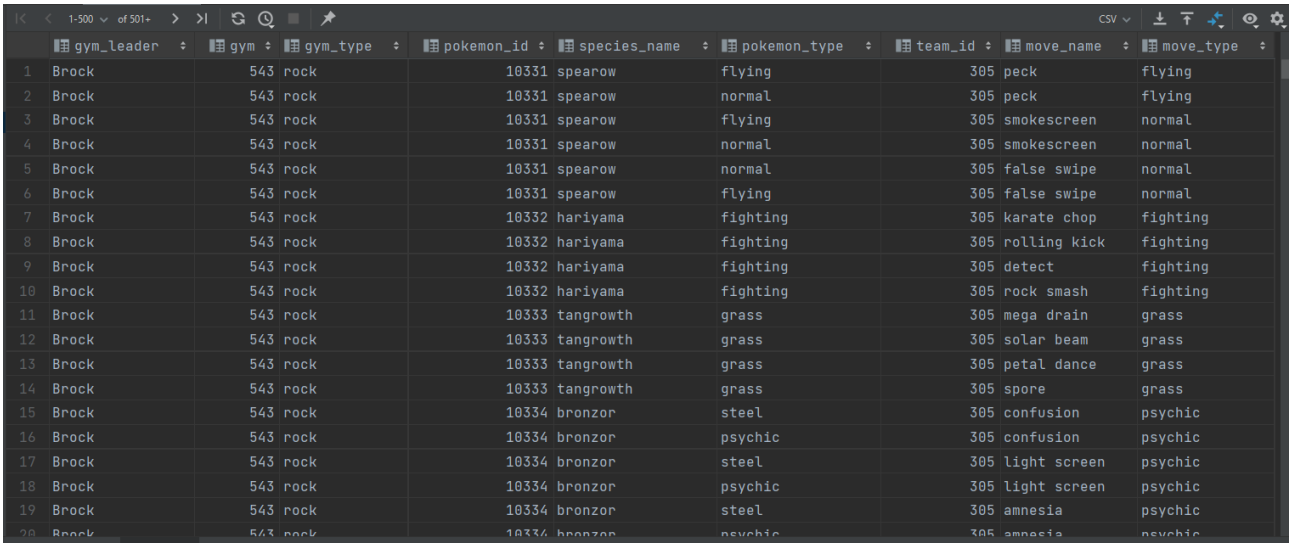


| | gym_leader | gym | gym_type | pokemon_id | species_name | pokemon_type | team_id | move_name | move_type |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Blaine | 557 | fire | 10427 | slowking | water | 46 | hypnosis | psychic |
| 2 | Blaine | 557 | fire | 10427 | slowking | psychic | 46 | hypnosis | psychic |
| 3 | Blaine | 557 | fire | 10427 | slowking | psychic | 46 | agility | psychic |
| 4 | Blaine | 557 | fire | 10427 | slowking | water | 46 | agility | psychic |
| 5 | Blaine | 557 | fire | 10427 | slowking | water | 46 | rest | psychic |
| 6 | Blaine | 557 | fire | 10427 | slowking | psychic | 46 | rest | psychic |
| 7 | Blaine | 557 | fire | 10427 | slowking | psychic | 46 | mirror coat | psychic |
| 8 | Blaine | 557 | fire | 10427 | slowking | water | 46 | mirror coat | psychic |
| 9 | Blaine | 557 | fire | 10428 | dragonite | dragon | 46 | gust | flying |
| 10 | Blaine | 557 | fire | 10428 | dragonite | flying | 46 | gust | flying |
| 11 | Blaine | 557 | fire | 10428 | dragonite | flying | 46 | peck | flying |
| 12 | Blaine | 557 | fire | 10428 | dragonite | dragon | 46 | peck | flying |
| 13 | Blaine | 557 | fire | 10428 | dragonite | flying | 46 | aeroblast | flying |
| 14 | Blaine | 557 | fire | 10428 | dragonite | dragon | 46 | aeroblast | flying |
| 15 | Blaine | 557 | fire | 10428 | dragonite | flying | 46 | twister | dragon |
| 16 | Blaine | 557 | fire | 10428 | dragonite | dragon | 46 | twister | dragon |
| 17 | Blaine | 557 | fire | 10429 | eevee | normal | 46 | roar | normal |
| 18 | Blaine | 557 | fire | 10429 | eevee | normal | 46 | sonic boom | normal |
| 19 | Blaine | 557 | fire | 10429 | eevee | normal | 46 | screech | normal |
| 20 | Blaine | 557 | fire | 10429 | eevee | normal | 46 | splash | normal |

And if we change the type of a move:

```
UPDATE move
SET type = 'rock'
WHERE name = 'take down';
```

We can see that Brock's spearrow still appears but only three of its moves do:

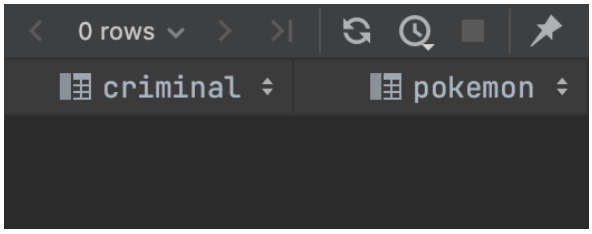


| | gym_leader | gym | gym_type | pokemon_id | species_name | pokemon_type | team_id | move_name | move_type |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Brock | 543 | rock | 10331 | spearow | flying | 305 | peck | flying |
| 2 | Brock | 543 | rock | 10331 | spearow | normal | 305 | peck | flying |
| 3 | Brock | 543 | rock | 10331 | spearow | flying | 305 | smokescreen | normal |
| 4 | Brock | 543 | rock | 10331 | spearow | normal | 305 | smokescreen | normal |
| 5 | Brock | 543 | rock | 10331 | spearow | normal | 305 | false swipe | normal |
| 6 | Brock | 543 | rock | 10331 | spearow | flying | 305 | false swipe | normal |
| 7 | Brock | 543 | rock | 10332 | hariyama | fighting | 305 | karate chop | fighting |
| 8 | Brock | 543 | rock | 10332 | hariyama | fighting | 305 | rolling kick | fighting |
| 9 | Brock | 543 | rock | 10332 | hariyama | fighting | 305 | detect | fighting |
| 10 | Brock | 543 | rock | 10332 | hariyama | fighting | 305 | rock smash | fighting |
| 11 | Brock | 543 | rock | 10333 | tangrowth | grass | 305 | mega drain | grass |
| 12 | Brock | 543 | rock | 10333 | tangrowth | grass | 305 | solar beam | grass |
| 13 | Brock | 543 | rock | 10333 | tangrowth | grass | 305 | petal dance | grass |
| 14 | Brock | 543 | rock | 10333 | tangrowth | grass | 305 | spore | grass |
| 15 | Brock | 543 | rock | 10334 | bronzor | steel | 305 | confusion | psychic |
| 16 | Brock | 543 | rock | 10334 | bronzor | psychic | 305 | confusion | psychic |
| 17 | Brock | 543 | rock | 10334 | bronzor | steel | 305 | light screen | psychic |
| 18 | Brock | 543 | rock | 10334 | bronzor | psychic | 305 | light screen | psychic |
| 19 | Brock | 543 | rock | 10334 | bronzor | steel | 305 | amnesia | psychic |

## 9.5 Query 5

### 9.5.1 Solution

```
SELECT ec.id AS criminal, pc.id AS pokemon
   FROM evil_criminal ec JOIN pokemon_creature pc ON pc.trainer = ec.id
   JOIN cause_or_receive cor ON pc.id = cor.creature
   JOIN move m ON cor.move = m.id JOIN healing h on m.id = h.id
   WHERE pc.method = 'gift' AND ec.is_leader = TRUE
   AND m.accuracy_percentage != 0 AND pc.subarea IS NOT NULL;
```




As you can see, the query does not return any rows. We will explain the reason why below.

### 9.5.2 Explanation

To get the Pokémon that have been obtained through a ‘gift’ by a criminal we first join these two tables (***Evil_Criminal*** and ***Pokemon_Creature***) and specify in the *WHERE* clause that the Pokémon’s *method* (representing how they were obtained) equals ‘gift’. To ensure now these were obtained in a subarea, we add a second condition to the *WHERE* clause specifying that the *subarea* attribute of the ***Pokemon_Creature*** table cannot be *NULL*. Also, to show only the criminal leaders we add another condition to the *WHERE* clause stating that their *is_leader* attribute has to be *TRUE.*

Lastly, to check if the Pokémon have a move that can drain other Pokémons HP we join the ***Pokemon_Creature*** and ***Move*** entities (through the ***Cause_Or_Receive*** table). Now, as we classified the drain moves as ***Healing*** moves on the previous phase (as the Pokémon use them to heal themselves) we join this table with the ***Move***’s *id*. This will give us all the moves that are of type healing. However, we realized that the drain moves are those that have a none zero in their *accuracy_percentage* attribute (as opposed to all other healing moves that don’t drain). Hence, we added this condition to the *WHERE* clause.

### 9.5.3 Query validation

This query did not return any rows because there aren't any Pokémon in the given datasets that satisfy all these conditions. Therefore, to prove that the query works we will insert a new Pokémon that does by executing the query below. We will first add the Pokémon to the ***Pokemon_Creature*** table by adding a big *id* (not already added), giving it a *nickname*, allocate it a valid *subarea*, stating its obtention method as 'gift', and assign it to an evil criminal that is a leader (*id* = 507 as you can see on the screenshot below).

```
INSERT INTO pokemon_creature(id, species, nickname, subarea, method, trainer)
VALUES(14000, 1, 'ultimatePokemon', 127, 'gift', 507);
```

| id | money_steal | sig... | is_leader |
|---|---|---|---|
| 507 | 69697000000000 | Hit the gy… | • true |

Also, we must ensure this Pokémon has a move that drains other Pokémons HP, so we create a new entry at the ***Cause_Or_Recieve*** table linking this Pokémon to the move with *id* = 71, which as you can double check in the screenshot below is a draining move.

```
INSERT INTO cause_or_receive(creature, move, id) VALUES (14000, 71, 14000);
```

| id | name | ... | effect |
|---|---|---|---|
| 71 | absorb | 100 | Drains half the damage inflicted to heal the user. |

Therefore, after inserting these two rows and executing the previous query again, we get the following:

| criminal | pokemon |
|---|---|
| 507 | 14000 |

As you can see, the query now does return the newly inserted Pokémon, meaning the conditions of the query are checked correctly, but before there weren't any Pokémon satisfying them.

Also, to prove that the drain moves are those we stored as *Healing* and have an accuracy percentage different than zero we will execute the following query:

```
SELECT m.name, m.effect
      FROM move m
      JOIN healing h on m.id = h.id
      WHERE accuracy_percentage != 0;
```

This is the output we get. Notice how all these are draining other Pokémons HP.

| name | effect |
|---|---|
| absorb | Drains half the damage inflicted to heal the user. |
| mega drain | Drains half the damage inflicted to heal the user. |
| dream eater | Only works on sleeping Pokémon.  Drains half the damage inflicted to heal the user. |
| leech life | Drains half the damage inflicted to heal the user. |
| giga drain | Drains half the damage inflicted to heal the user. |
| drain punch | Drains half the damage inflicted to heal the user. |
| horn leech | Drains half the damage inflicted to heal the user. |
| parabolic charge | Heals the user for half the total damage dealt to all targets. |
| draining kiss | Drains 75% of the damage inflicted to heal the user. |
| oblivion wing | Drains 75% of the damage inflicted to heal the user. |

However, if we invert the *WHERE* condition of the previous query and make it equal to zero we obtain these:

| name | effect |
|---|---|
| recover | Heals the user by half its max HP. |
| soft boiled | Heals the user by half its max HP. |
| milk drink | Heals the user by half its max HP. |
| morning sun | Heals the user by half its max HP.  Affected by weather. |
| synthesis | Heals the user by half its max HP.  Affected by weather. |
| moonlight | Heals the user by half its max HP.  Affected by weather. |
| swallow | Recovers 1/4 HP after one Stockpile, 1/2 HP after two Stockpiles, or full HP after thr |
| slack off | Heals the user by half its max HP. |
| roost | Heals the user by half its max HP. |
| heal order | Heals the user by half its max HP. |
| heal pulse | Heals the target for half its max HP. |
| shore up | Heals the user for ½ its max HP, or ⅔ during a sandstorm. |
| floral healing | Heals the target for ½ its max HP, or ⅔ on Grassy Terrain. |
| purify | Cures the target of a major status ailment and heals the user for 50% of its max HP. |
| jungle healing | Inflicts regular damage with no additional effect. |

Notice how all these aren't drain moves.

## 9.6 Query 6

### 9.6.1 Solution

### 9.6.2 Explanation

### 9.6.3 Query validation

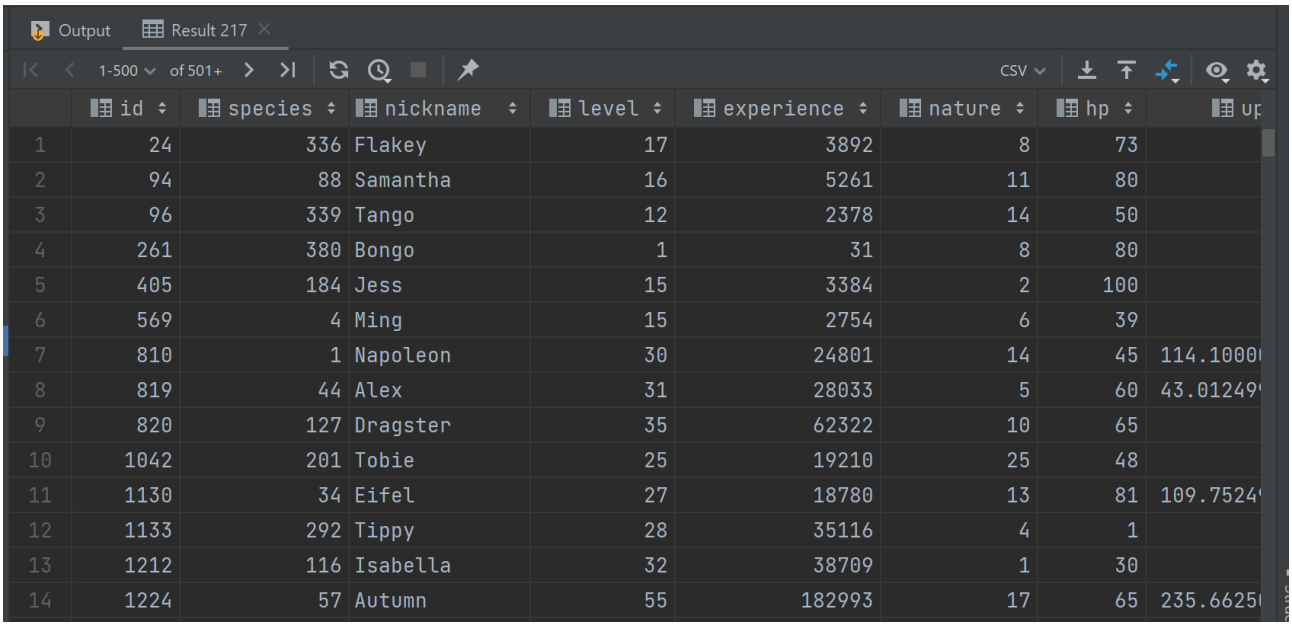
We faced some problems when doing this trigger and didn't have enough time to fix it.

## 9.7 Query 7

### 9.7.1 Solution

```
SELECT   p.id,   p.species,   p.nickname,   p.level,   p.experience,
p.nature,
       p.hp, (((2 * hp + (level ^ 2) / 4) / 100 + 5) * nature) as
update_hp,
       p.attack, (((2 * p.attack + (level ^ 2) / 4) / 100 + 5) *
nature) as updated_attack,
      p.defense, (((2 * p.defense + (level ^ 2) / 4) / 100 + 5) *
nature) as updated_defense,
```

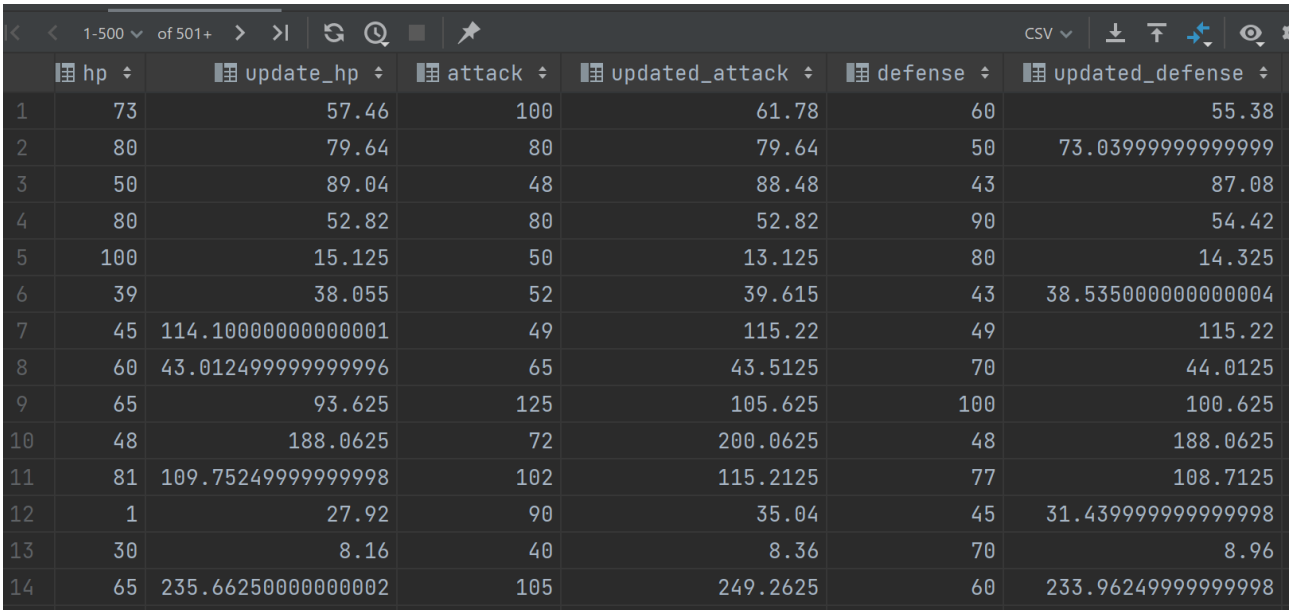
```
       p.special_attack, (((2 * p.special_attack + (level ^ 2) / 4)
/ 100 + 5) * nature) as updated_special_attack,
       p.special_defense, (((2 * p.special_defense + (level ^ 2) /
4) / 100 + 5) * nature) as updated_special_defense,
       p.speed, (((2 * p.speed + (level ^ 2) / 4) / 100 + 5) *
nature) as updated_speed
FROM pokemon_creature as p
WHERE EXTRACT(YEAR FROM p.date_time) >= 2022 and EXTRACT(YEAR FROM
p.date_time) <= 2023;
```



Output | Result 217

1-500 of 501+ | CSV

| | id | species | nickname | level | experience | nature | hp | up |
|---|---|---|---|---|---|---|---|---|
| 1 | 24 | 336 | Flakey | 17 | 3892 | 8 | 73 | |
| 2 | 94 | 88 | Samantha | 16 | 5261 | 11 | 80 | |
| 3 | 96 | 339 | Tango | 12 | 2378 | 14 | 50 | |
| 4 | 261 | 380 | Bongo | 1 | 31 | 8 | 80 | |
| 5 | 405 | 184 | Jess | 15 | 3384 | 2 | 100 | |
| 6 | 569 | 4 | Ming | 15 | 2754 | 6 | 39 | |
| 7 | 810 | 1 | Napoleon | 30 | 24801 | 14 | 45 | 114.1000 |
| 8 | 819 | 44 | Alex | 31 | 28033 | 5 | 60 | 43.01249 |
| 9 | 820 | 127 | Dragster | 35 | 62322 | 10 | 65 | |
| 10 | 1042 | 201 | Tobie | 25 | 19210 | 25 | 48 | |
| 11 | 1130 | 34 | Eifel | 27 | 18780 | 13 | 81 | 109.7524 |
| 12 | 1133 | 292 | Tippy | 28 | 35116 | 4 | 1 | |
| 13 | 1212 | 116 | Isabella | 32 | 38709 | 1 | 30 | |
| 14 | 1224 | 57 | Autumn | 55 | 182993 | 17 | 65 | 235.6625 |



1-500 of 501+ | CSV

| | hp | update_hp | attack | updated_attack | defense | updated_defense |
|---|---|---|---|---|---|---|
| 1 | 73 | 57.46 | 100 | 61.78 | 60 | 55.38 |
| 2 | 80 | 79.64 | 80 | 79.64 | 50 | 73.03999999999999 |
| 3 | 50 | 89.04 | 48 | 88.48 | 43 | 87.08 |
| 4 | 80 | 52.82 | 80 | 52.82 | 90 | 54.42 |
| 5 | 100 | 15.125 | 50 | 13.125 | 80 | 14.325 |
| 6 | 39 | 38.055 | 52 | 39.615 | 43 | 38.535000000000004 |
| 7 | 45 | 114.10000000000001 | 49 | 115.22 | 49 | 115.22 |
| 8 | 60 | 43.012499999999996 | 65 | 43.5125 | 70 | 44.0125 |
| 9 | 65 | 93.625 | 125 | 105.625 | 100 | 100.625 |
| 10 | 48 | 188.0625 | 72 | 200.0625 | 48 | 188.0625 |
| 11 | 81 | 109.75249999999998 | 102 | 115.2125 | 77 | 108.7125 |
| 12 | 1 | 27.92 | 90 | 35.04 | 45 | 31.439999999999998 |
| 13 | 30 | 8.16 | 40 | 8.36 | 70 | 8.96 |
| 14 | 65 | 235.66250000000002 | 105 | 249.2625 | 60 | 233.96249999999998 |

| | special_attack | updated_special_attack | special_defense | updated_special_defens |
|---|---|---|---|---|
| 1 | 100 | 61.78 | 60 | 5 |
| 2 | 40 | 70.83999999999999 | 50 | 73.0399999999 |
| 3 | 46 | 87.92 | 41 | 8 |
| 4 | 110 | 57.620000000000005 | 130 | 6 |
| 5 | 60 | 13.525 | 80 | 14 |
| 6 | 60 | 40.575 | 50 | 39 |
| 7 | 65 | 119.70000000000002 | 65 | 119.7000000000 |
| 8 | 85 | 45.512499999999996 | 75 | 44. |
| 9 | 55 | 91.625 | 70 | 94 |
| 10 | 72 | 200.0625 | 48 | 188. |
| 11 | 85 | 110.79250000000002 | 75 | 108. |
| 12 | 30 | 30.240000000000002 | 30 | 30.240000000000 |
| 13 | 70 | 8.96 | 25 | |
| 14 | 60 | 233.96249999999998 | 70 | 237. |

1-500 of 501+ CSV

| | cial_attack | special_defense | updated_special_defense | speed | updated_speed |
|---|---|---|---|---|---|
| 1 | 61.78 | 60 | 55.38 | 65 | 56.18 |
| 2 | 3999999999999 | 50 | 73.03999999999999 | 25 | 67.53999999999999 |
| 3 | 87.92 | 41 | 86.52 | 60 | 91.84 |
| 4 | 0000000000005 | 130 | 60.82 | 110 | 57.620000000000005 |
| 5 | 13.525 | 80 | 14.325 | 50 | 13.125 |
| 6 | 40.575 | 50 | 39.375 | 65 | 41.175 |
| 7 | 0000000000002 | 65 | 119.70000000000002 | 45 | 114.10000000000001 |
| 8 | .2499999999996 | 75 | 44.5125 | 40 | 41.0125 |
| 9 | 91.625 | 70 | 94.625 | 85 | 97.625 |
| 10 | 200.0625 | 48 | 188.0625 | 48 | 188.0625 |
| 11 | 9250000000002 | 75 | 108.1925 | 85 | 110.79250000000002 |
| 12 | 0000000000002 | 30 | 30.240000000000002 | 40 | 31.04 |
| 13 | 8.96 | 25 | 8.06 | 60 | 8.76 |
| 14 | 6249999999998 | 70 | 237.3625 | 95 | 245.8625 |

#### 9.7.2 Explanation

This query simply looks for the pokemon_creature captured between 2022 and 2023 with the WHERE clause and extract function from PostgreSql to take the year from the datetime attribute. Then, we simply took the same formula and put the updated statistic.

#### 9.7.3 Query validation

To validate it, we have simply printed the level, nature and the statistics. If we apply the formula for hp 73, level 17 and nature 8, we will get approximately 57.46. Since this query has the same logic applied to each statistic, we can know that the rest of the statistics is fine as well.

### 9.8 Query 8

#### 9.8.1 Solution

#### 9.8.2 Explanation

#### 9.8.3 Query validation

We faced some problems when doing this trigger and didn't have enough time to fix it.

# 10 Conclusions

## *10.1 Use of resources*

| Stage | Valèria | Eugènia | Sebastián | Marc | Total |
|---|---|---|---|---|---|
| **ER/RM model update** | 0h | 0h | 0h | ¼ h | ¼ h |
| **Data type selection** | 0h | 0h | 0h | 0h | 0h |
| **Physical model codification** | 0h | 0h | 0h | 0h | 0h |
| **Database population** | 0h | 0h | 0h | 0 h | 0 h |
| **Implementation and validation of the queries and triggers** | 10h | 10h | 10h | 10h | 40h |
| **Reporting** | 10h | 7h | 7h | 7h | 28h |
| **Total:** | 20 h | 20 h | 20 h | 20.25 h | 80.25h |

In general, the parts that took us longer to complete were the implementation of the queries and triggers, as we got stuck with some of them. Also, reporting everything that we did and explaining the validation of our queries was a long process.

We believed we all worked approximately the same amount of ours as we distributed the work equally between all the team members and helped one another when any of us encountered an issue they couldn't solve by themselves.

## *10.2 IA use (if required, 1-2 pages)*

We did not use any AI tools to develop this project.

## *10.3 Lessons learnt and conclusions (1 page)*

This stage of the project has allowed us to put into practice everything we have learnt throughout the course. Firstly, we were able to code hard queries in SQL that required a large set of JOINS and/or subqueries. In fact, it has forced us to review concepts that we had forgotten a bit such as the different kinds of joins or subqueries.

Secondly, we have had to review triggers as it was a topic we did very briefly in the first semester and we had happened to forget it by now. However, reviewing the slides made us remember the previously learned concepts and we were able to successfully implement almost all the required triggers.

Lastly, we have learned the importance of organization in a team in order to successfully carry out a project of this size. It is true that we already had to do it in the previous phase, but this one required more coordination between all the members of the team because of the little time we had to implement this one, considering the enormous amount of work we all have to do at the end of the semester. If we had had more time, we believe we could have finished all the work, as we faced some problems with some queries and didn't have time to fix them.

To conclude, we can say that this project has helped us to reinforce our SQL skills and taught us how to operate a database from its creation to its population and usage. Moreover, it has allowed us to have a satisfactory first contact with database programming.