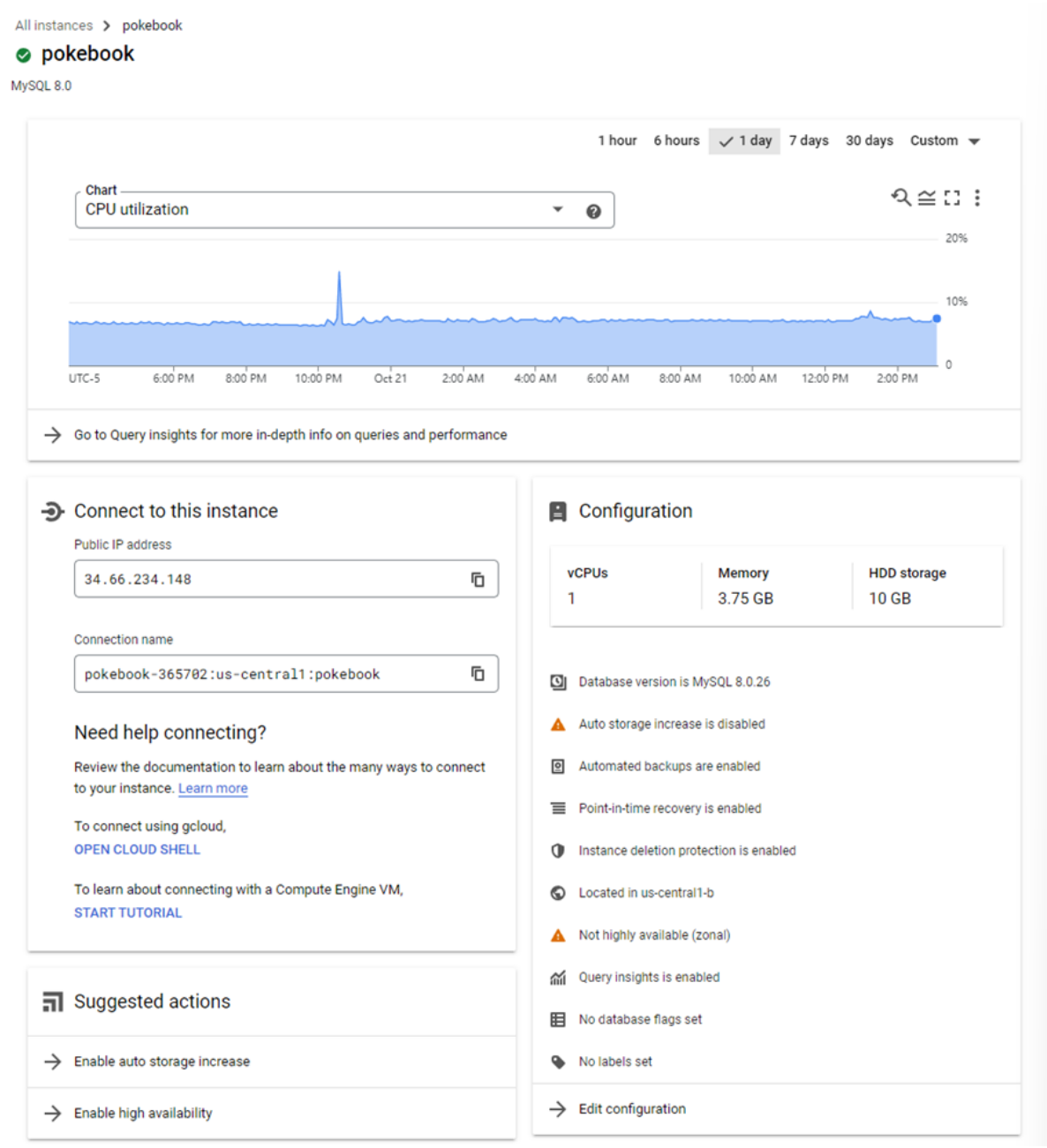
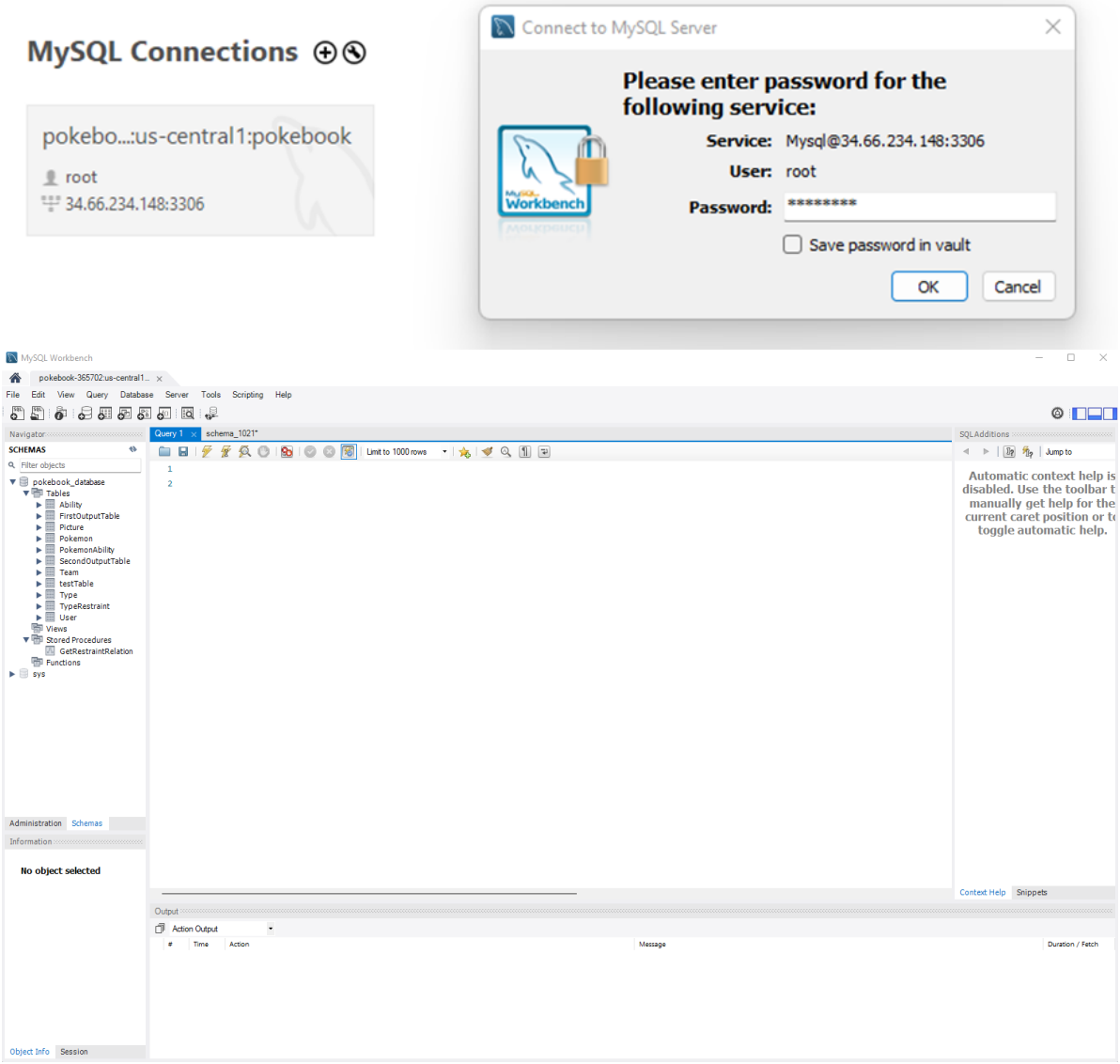
Database implementation:

Screenshot of the connection

GCP



MySQL connection:



DDL Command:

```
USE pokebook_database;

CREATE TABLE User (
 UserId INTEGER NOT NULL,
 UserName VARCHAR(20),
 UserEmail VARCHAR(30),
 UserPassword VARCHAR(15),
 PRIMARY KEY (UserId)
);

CREATE TABLE Pokemon (
 PokemonId INTEGER NOT NULL,
 PokemonName VARCHAR(20),
 Generation INTEGER,
 Height FLOAT,
 Weight FLOAT,
 Total INTEGER,
 Hp INTEGER,
 Attack INTEGER,
```

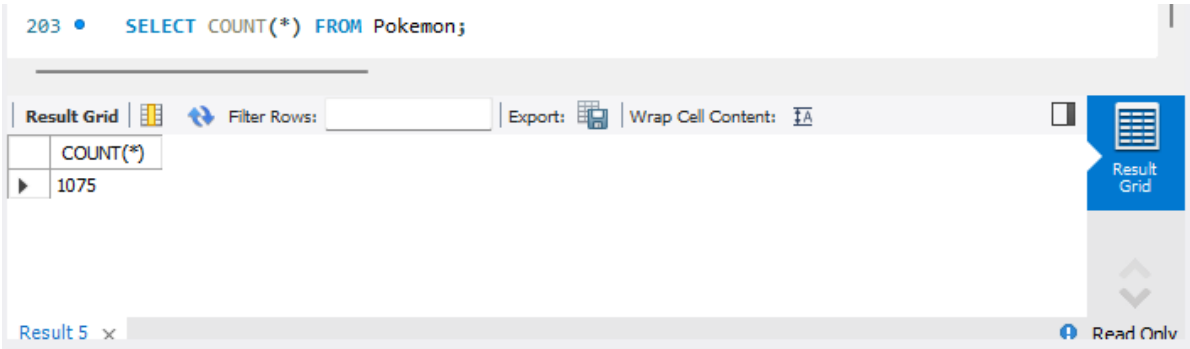
```
 Defense INTEGER,
 SpecialAttack INTEGER,
 SpecialDefense INTEGER,
 Speed INTEGER,
 FirstTypeId INTEGER,
 SecondTypeId INTEGER,
 PRIMARY KEY (PokemonId)
);

CREATE TABLE Team (
 UserId INTEGER,
 PokemonId INTEGER,
 PRIMARY KEY (UserId,PokemonId)
);

CREATE TABLE Picture (
 PictureId INTEGER NOT NULL,
 PictureType VARCHAR(10),
 PictureLocation VARCHAR(200),
 PokemonId INTEGER,
 PRIMARY KEY (PictureId)
);

CREATE TABLE Ability (
 AbilityId INTEGER NOT NULL,
 AbilityName VARCHAR(20),
 PRIMARY KEY (AbilityId)
);

CREATE TABLE PokemonAbility (
 PokemonId INTEGER,
 AbilityId INTEGER,
 PRIMARY KEY (PokemonId,AbilityId)
);

CREATE TABLE Type (
 TypeId INTEGER NOT NULL,
 TypeName VARCHAR(15),
 PRIMARY KEY (TypeId)
);


CREATE TABLE TypeRestrain (
 TheTypeId INTEGER,
 TheRestraintId INTEGER,
 Power FLOAT,
 PRIMARY KEY (TheTypeId,TheRestraintId)
);

ALTER TABLE Team ADD CONSTRAINT Team_PokemonId FOREIGN KEY(PokemonId)
REFERENCES Pokemon(PokemonId);

ALTER TABLE Team ADD CONSTRAINT Team_UserId FOREIGN KEY (UserId)
REFERENCES User(UserId);
```

```
ALTER TABLE Picture ADD CONSTRAINT Picture_PokemonId FOREIGN KEY(PokemonId)
REFERENCES Pokemon(PokemonId);

ALTER TABLE PokemonAbility ADD CONSTRAINT PokemonAbility_PokemonId FOREIGN
KEY(PokemonId)
REFERENCES Pokemon(PokemonId);

ALTER TABLE PokemonAbility ADD CONSTRAINT PokemonAbility_AbilityId FOREIGN
KEY(AbilityId)
REFERENCES Ability(AbilityId);

ALTER TABLE TypeRestrain ADD CONSTRAINT TypeRestraint_Type FOREIGN
KEY(TheTypeId)
REFERENCES Type(TypeId);

ALTER TABLE TypeRestrain ADD CONSTRAINT TypeRestraint_Type FOREIGN
KEY(TheRestraintId)
REFERENCES Type(TypeId);

LOAD DATA LOCAL INFILE
'D:/School/UIUC/Fall22/CS411/Project/Stage3/User_data_1.1.csv'
INTO TABLE User
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(UserId,UserName,UserEmail,UserPassword);

LOAD DATA LOCAL INFILE
'D:/School/UIUC/Fall22/CS411/Project/Stage3/Pokemon_data_1.1.csv'
INTO TABLE Pokemon
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(PokemonId,PokemonName,Generation,Height,Weight,Total,Hp,Attack,Defense,SpecialA
ttack,SpecialDefense,Speed);

LOAD DATA LOCAL INFILE
'D:/School/UIUC/Fall22/CS411/Project/Stage3/Abilities_data_1.1.csv'
INTO TABLE Ability
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(AbilityId,AbilityName);

LOAD DATA LOCAL INFILE
'D:/School/UIUC/Fall22/CS411/Project/Stage3/Type_data_1.1.csv'
INTO TABLE Type
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(TypeId,TypeName);
```

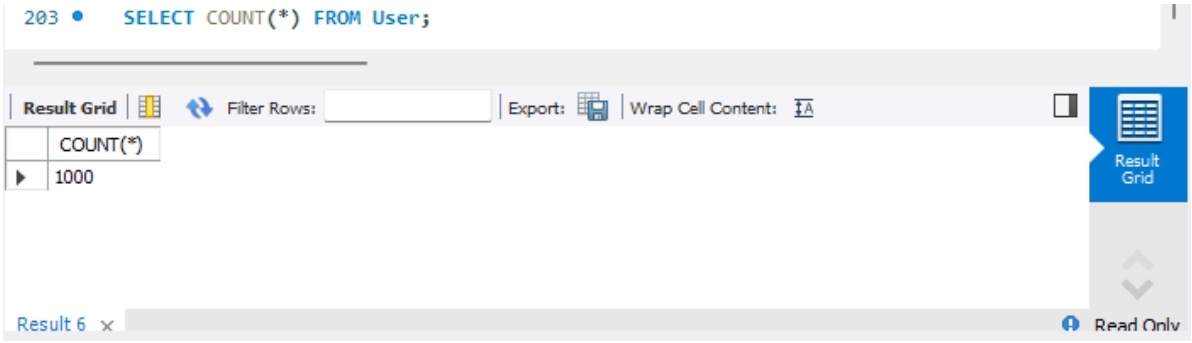
```
LOAD DATA LOCAL INFILE
'D:/School/UIUC/Fall22/CS411/Project/Stage3/Pokemon_Abilities_data_1.1.csv'
INTO TABLE PokemonAbility
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(PokemonId,AbilityId);

LOAD DATA LOCAL INFILE
'D:/School/UIUC/Fall22/CS411/Project/Stage3/TypeRestraint_1.1.csv'
INTO TABLE TypeRestraint
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(TheTypeId,TheRestraintId,Power);

LOAD DATA LOCAL INFILE
'D:/School/UIUC/Fall22/CS411/Project/Stage3/Picture_data_1.2.csv'
INTO TABLE Picture
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(PictureId,PictureType,PictureLocation,PokemonId);
```

Count Query :

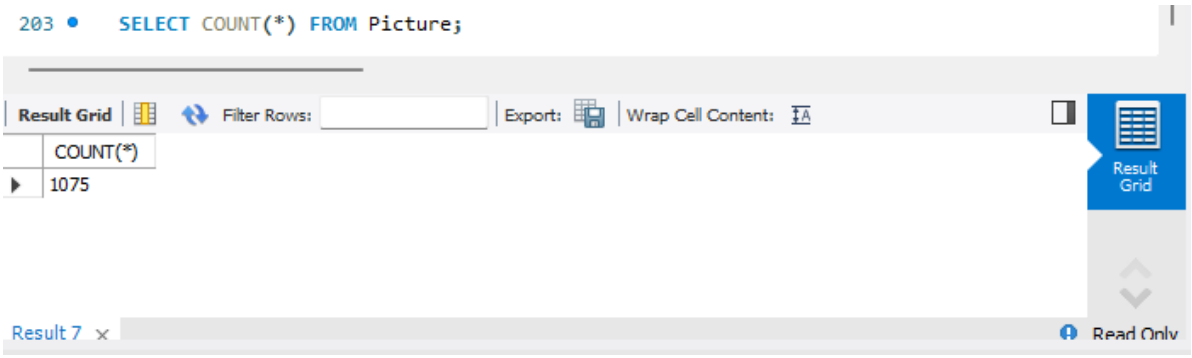
Pokemon Table



User Table



Picture Table



Advanced Queries:

First Query

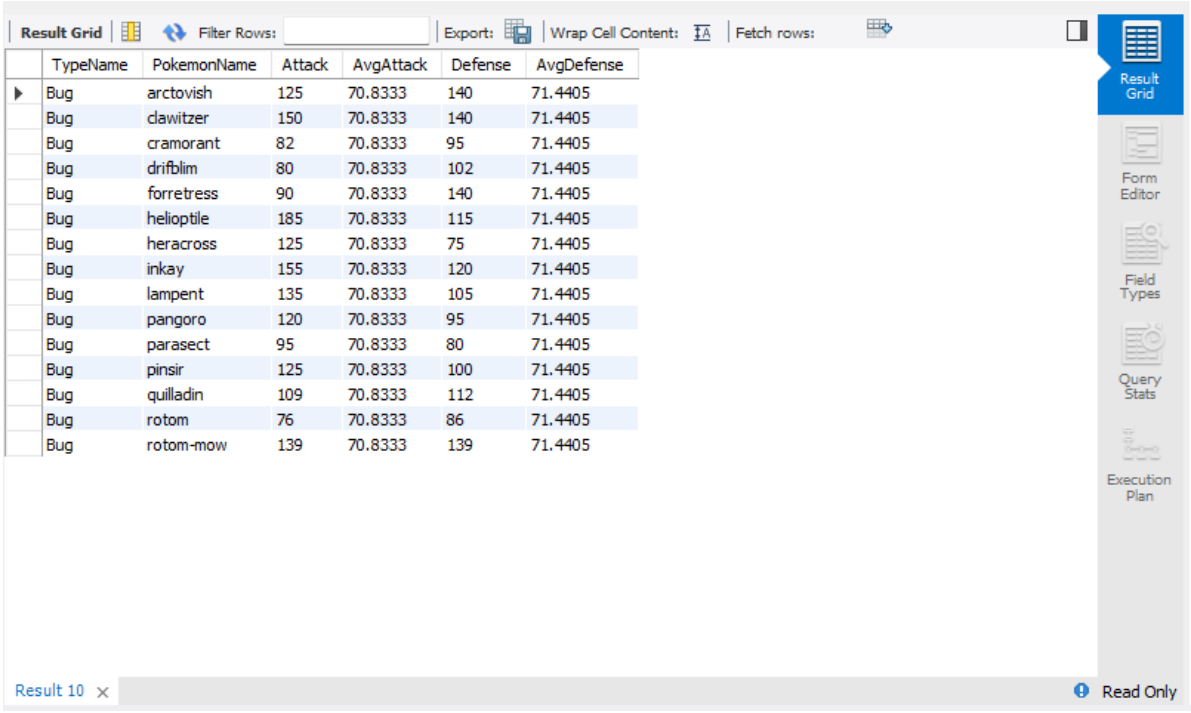
Description:

The first query wants Pokémons whose attack and defense value are greater than average attack and average defense for first type of Pokémon. The query returns *TypeName, PokemonName, Attack, AvgAttack, Defense, AvgDefense.*

```
SELECT t1.TypeName, p1.PokemonName, p1.Attack,
 (SELECT AVG(p2.Attack)
 FROM (Pokemon p2 LEFT JOIN Type t2 on p2.FirstTypeId =
 t2.TypeId)
 Where t1.TypeId = t2.TypeId
 GROUP BY t2.TypeId) as AvgAttack,
p1.Defense,
 (SELECT AVG(p2.Defense)
 FROM (Pokemon p2 LEFT JOIN Type t2 ON p2.FirstTypeId = t2.TypeId)
 Where t1.TypeId = t2.TypeId
 GROUP BY t2.TypeId) as AvgDefense
FROM (Pokemon p1 LEFT JOIN Type t1 on p1.FirstTypeId = t1.TypeId)
WHERE p1.Attack > (SELECT AVG(p2.Attack)
      FROM (Pokemon p2 LEFT JOIN Type t2 on p2.FirstTypeId =
      t2.TypeId)
      Where t1.TypeId = t2.TypeId
      GROUP BY t2.TypeId)
AND
 p1.Defense > (SELECT AVG(p2.Defense)
      FROM (Pokemon p2 LEFT JOIN Type t2 ON p2.FirstTypeId = t2.TypeId)
```

```
    Where t1.TypeId = t2.TypeId
    GROUP BY t2.TypeId)
GROUP BY t1.TypeId, t1.TypeName, p1.PokemonName, p1.Attack, p1.Defense
ORDER BY t1.TypeName, p1.PokemonName
Limit 15;
```

Screenshot:



TypeName	PokemonName	Attack	AvgAttack	Defense	AvgDefense
Bug	arctovish	125	70.8333	140	71.4405
Bug	clawitzer	150	70.8333	140	71.4405
Bug	cramorant	82	70.8333	95	71.4405
Bug	drifblim	80	70.8333	102	71.4405
Bug	forretress	90	70.8333	140	71.4405
Bug	helioptile	185	70.8333	115	71.4405
Bug	heracross	125	70.8333	75	71.4405
Bug	inkay	155	70.8333	120	71.4405
Bug	lampent	135	70.8333	105	71.4405
Bug	pangoro	120	70.8333	95	71.4405
Bug	parasect	95	70.8333	80	71.4405
Bug	pinsir	125	70.8333	100	71.4405
Bug	quilladin	109	70.8333	112	71.4405
Bug	rotom	76	70.8333	86	71.4405
Bug	rotom-mow	139	70.8333	139	71.4405

Second Query

Description:

The second query wants information of Pokémons whose type is fire, grass, or water and generation is greater or equal to 4. The query returns *PokemonName, FirstTypeName, SecondTypeName, Hp, Attack, Defense, SpecialAttack, SpecialDefense, Speed, Generation.*

```
((SELECT p2.Total, p2.PokemonName, t1.TypeName AS FirstTypeName, t2.TypeName AS
SecondTypeName, p2.Hp, p2.Attack, p2.Defense, p2.SpecialAttack,
p2.SpecialDefense, p2.Speed, p2.Generation
FROM (Pokemon p2 LEFT JOIN Type t1 ON p2.FirstTypeId = t1.TypeId) JOIN Type t2
ON (p2.SecondTypeId =
t2.TypeId)
WHERE (t1.TypeName like '%Fire%' OR t2.TypeName like '%Fire%') AND p2.Generation
>= 4
ORDER BY p2.Total ASC)
UNION
(SELECT p2.Total, p2.PokemonName, t1.TypeName AS FirstTypeName, t2.TypeName AS
SecondTypeName, p2.Hp, p2.Attack, p2.Defense, p2.SpecialAttack,
p2.SpecialDefense, p2.Speed, p2.Generation
FROM (Pokemon p2 LEFT JOIN Type t1 ON p2.FirstTypeId = t1.TypeId) JOIN Type t2
ON (p2.SecondTypeId =
t2.TypeId)
WHERE (t1.TypeName like '%Grass%' OR t2.TypeName like '%Grass%') AND
p2.Generation
```

```
>= 4
ORDER BY p2.Total ASC)
UNION
(SELECT p2.Total, p2.PokemonName, t1.TypeName AS FirstTypeName, t2.TypeName AS
SecondTypeName, p2.Hp, p2.Attack, p2.Defense, p2.SpecialAttack,
p2.SpecialDefense, p2.Speed, p2.Generation
FROM (Pokemon p2 LEFT JOIN Type t1 ON p2.FirstTypeId = t1.TypeId) JOIN Type t2
ON (p2.SecondTypeId =
t2.TypeId)
WHERE (t1.TypeName like '%Water%' OR t2.TypeName like '%Water%') AND
p2.Generation
>= 4
ORDER BY p2.Total ASC) ORDER BY Total LIMIT 15)
;
```

Screenshot:

Total	PokemonName	FirstTypeName	SecondTypeName	Hp	Attack	Defense	SpecialAttack	SpecialDefense	Speed	Generation
175	morgrem	Water	None	45	20	20	25	25	40	7
210	indeedee-male	Grass	None	42	30	38	30	38	32	7
230	dracovish	Bug	Water	25	35	40	20	30	80	7
250	milcery	Grass	None	40	55	35	50	35	35	7
250	salamence-mega	Grass	None	40	40	60	40	60	10	8
260	raichu-alola	Grass	Dragon	40	40	80	40	40	20	8
269	mr-rime	Water	Bug	38	40	52	40	72	27	7
275	bouffalant	Ghost	Fire	50	30	55	65	55	20	5
275	honchkrow	Grass	None	45	35	45	62	53	35	4
280	dugtrio-alola	Water	None	41	63	40	40	30	66	8
280	carracosta	Grass	None	45	35	50	70	50	30	5
280	cofagrigus	Grass	Fairy	40	27	60	37	50	66	5
280	burmy	Grass	Poison	40	30	35	50	70	55	4
284	raticate-totem-...	Water	None	50	64	50	38	38	44	8
285	falinks	Grass	Fairy	40	35	55	65	75	15	7

(*: SecondTypeId could be None because not every Pokemon has two types.)

Indexing Analysis:

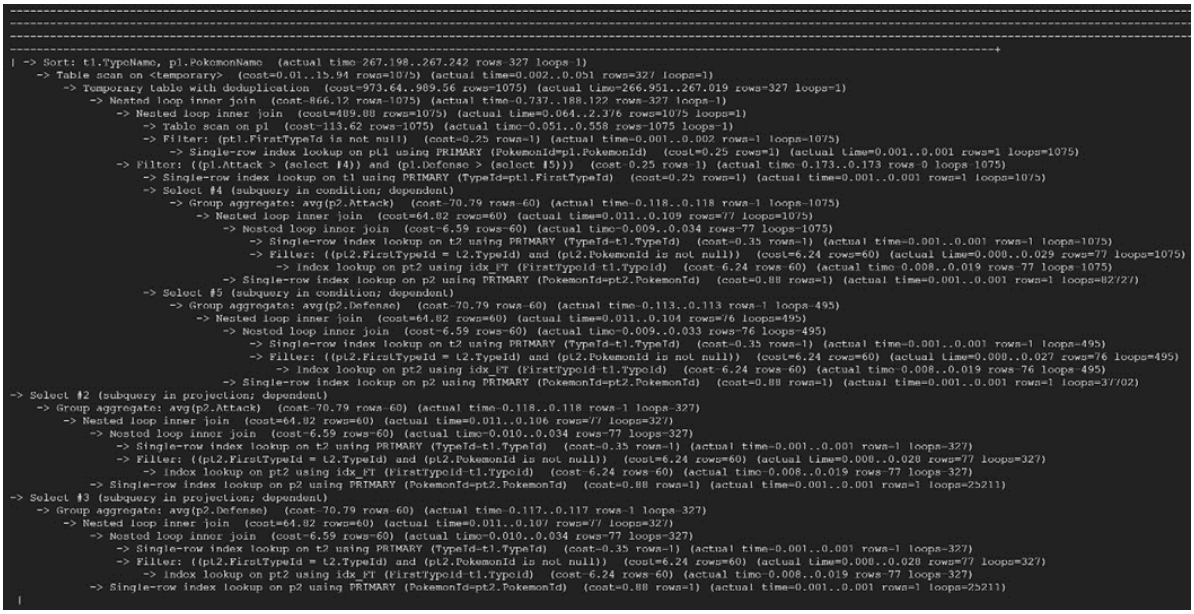
First Query

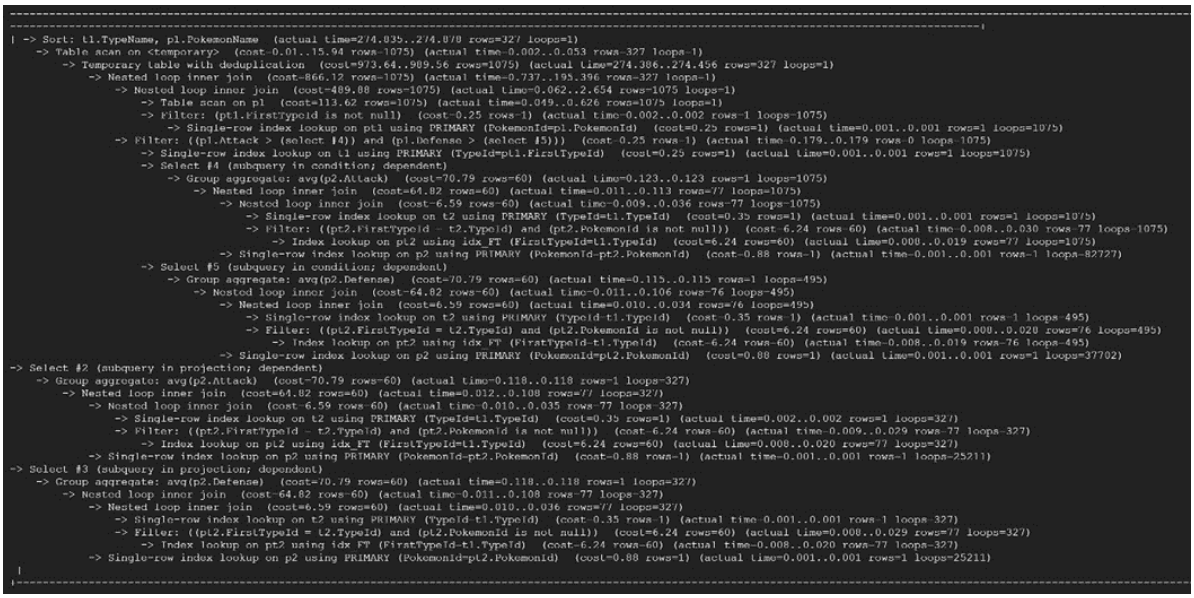
```
SELECT t1.TypeName, p1.PokemonName, p1.Attack,
 (SELECT AVG(p2.Attack)
 FROM (Pokemon p2 LEFT JOIN Type t2 on p2.FirstTypeId =
 t2.TypeId)
 Where t1.TypeId = t2.TypeId
 GROUP BY t2.TypeId) as AvgAttack,
p1.Defense,
 (SELECT AVG(p2.Defense)
 FROM (Pokemon p2 LEFT JOIN Type t2 ON p2.FirstTypeId = t2.TypeId)
 Where t1.TypeId = t2.TypeId
 GROUP BY t2.TypeId) as AvgDefense
FROM (Pokemon p1 LEFT JOIN Type t1 on p1.FirstTypeId = t1.TypeId)
WHERE p1.Attack > (SELECT AVG(p2.Attack)
      FROM (Pokemon p2 LEFT JOIN Type t2 on p2.FirstTypeId =
      t2.TypeId)
      Where t1.TypeId = t2.TypeId
      GROUP BY t2.TypeId)
AND
```

```
  p1.Defense > (SELECT AVG(p2.Defense)
      FROM (Pokemon p2 LEFT JOIN Type t2 ON p2.FirstTypeId = t2.TypeId)
      Where t1.TypeId = t2.TypeId
      GROUP BY t2.TypeId)
GROUP BY t1.TypeId, t1.TypeName, p1.PokemonName, p1.Attack, p1.Defense
ORDER BY t1.TypeName, p1.PokemonName
Limit 15;
```

First try of indexing design:

Index on **Attack**





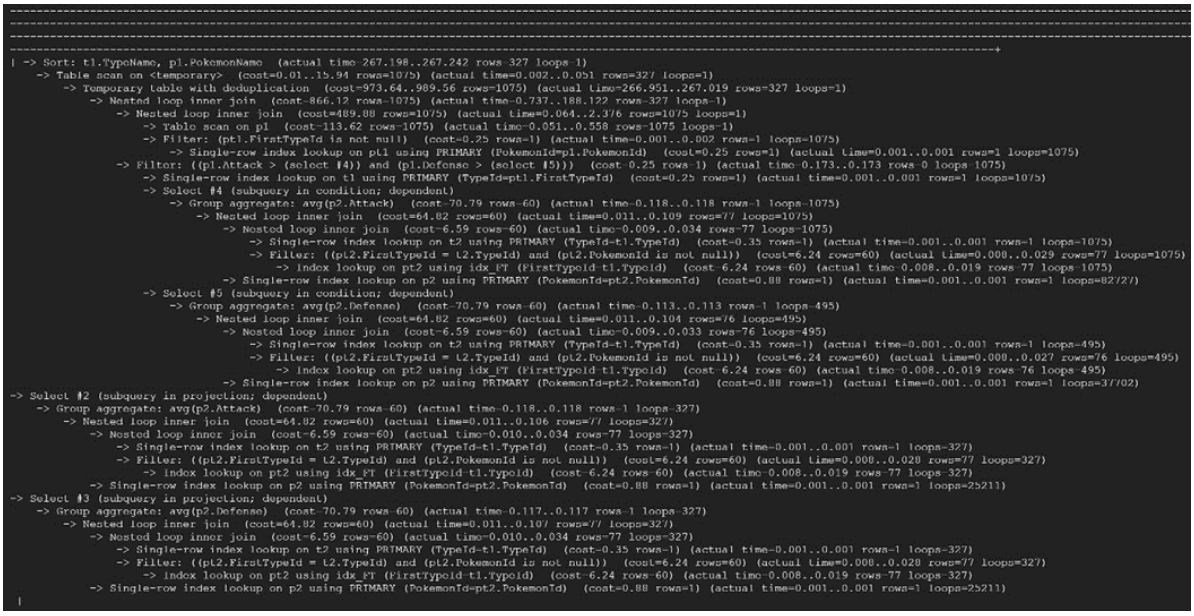
Since we wish to filter Pokemon with higher-than-average attack and defense, we have decided to create an index on Attack. However, the results are quite unsatisfactory, the running time became worse. The main reason is possibly that when we have attack as index, we are distorting the original data structure and make it more redundant.

Running time before indexing: 267.198, 0.118, 0.117

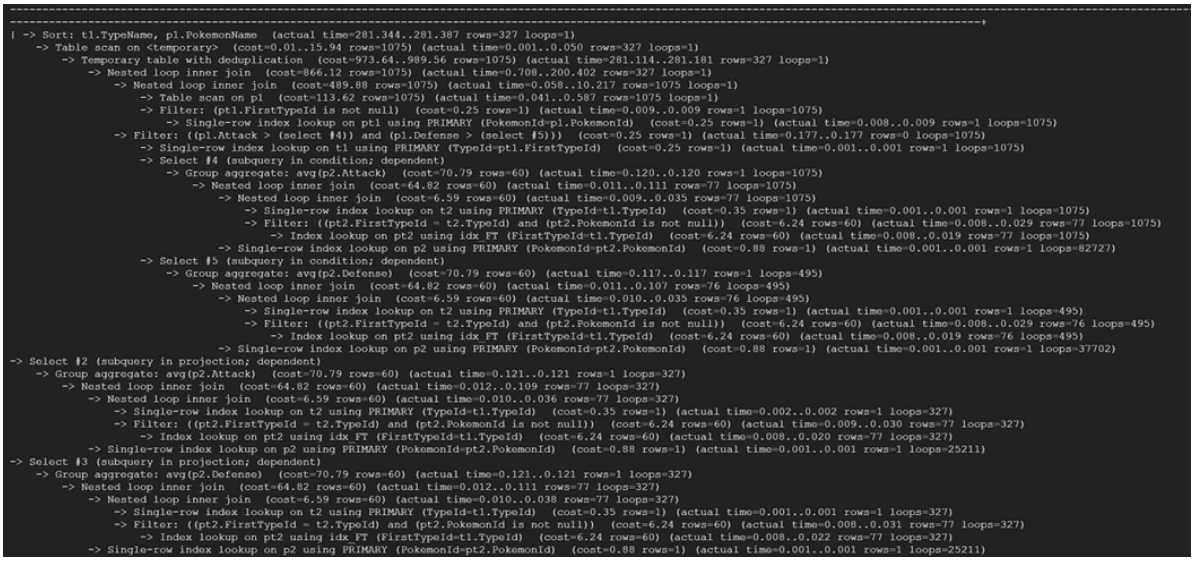
Running time after indexing: 274.386, 0.118, 0.117

Second try of indexing design:

Index on TypeName



```
-> Sort: t1.TypeName, p1.PokemonName  (actual time=267.198..267.242 rows=327 loops=1)
    -> Table scan on <temporary>  (cost=0.01..15.94 rows=1075) (actual time=0.002..0.051 rows=327 loops=1)
        -> Temporary table with deduplication  (cost=973.64..989.56 rows=1075) (actual time=266.951..267.019 rows=327 loops=1)
            -> Nested loop inner join  (cost=866.12 rows=1075) (actual time=0.737..188.122 rows=327 loops=1)
                -> Nested loop inner join  (cost=489.88 rows=1075) (actual time=0.064..2.376 rows=1075 loops=1)
                    -> Table scan on p1  (cost=113.62 rows=1075) (actual time=0.051..0.558 rows=1075 loops=1)
                    -> Filter: (pt1.FirstTypeId is not null)  (cost=0.25 rows=1) (actual time=0.001..0.002 rows=1 loops=1075)
                        -> Single-row index lookup on pt1 using PRIMARY (PokemonId=p1.PokemonId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1075)
                -> Filter: ((p1.Attack > (select #4)) and (p1.Defense > (select #5)))  (cost=0.25 rows=1) (actual time=0.173..0.173 rows=0 loops=1075)
                    -> Single-row index lookup on t1 using PRIMARY (TypeId=pt1.FirstTypeId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1075)
                    -> Select #4 (subquery in condition; dependent)
                        -> Group aggregate: avg(p2.Attack)  (cost=70.79 rows=60) (actual time=0.118..0.118 rows=1 loops=1075)
                            -> Nested loop inner join  (cost=64.82 rows=60) (actual time=0.011..0.109 rows=77 loops=1075)
                                -> Nested loop inner join  (cost=6.59 rows=60) (actual time=0.009..0.034 rows=77 loops=1075)
                                    -> Single-row index lookup on t2 using PRIMARY (TypeId=t1.TypeId)  (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=1075)
                                    -> Filter: ((pt2.FirstTypeId = t2.TypeId) and (pt2.PokemonId is not null))  (cost=6.24 rows=60) (actual time=0.008..0.029 rows=77 loops=1075)
                                        -> Index lookup on pt2 using idx_FT (FirstTypeId=t1.TypeId)  (cost=6.24 rows=60) (actual time=0.008..0.019 rows=77 loops=1075)
                                -> Single-row index lookup on p2 using PRIMARY (PokemonId=pt2.PokemonId)  (cost=0.88 rows=1) (actual time=0.001..0.001 rows=1 loops=82727)
                    -> Select #5 (subquery in condition; dependent)
                        -> Group aggregate: avg(p2.Defense)  (cost=70.79 rows=60) (actual time=0.113..0.113 rows=1 loops=495)
                            -> Nested loop inner join  (cost=64.82 rows=60) (actual time=0.011..0.104 rows=76 loops=495)
                                -> Nested loop inner join  (cost=6.59 rows=60) (actual time=0.009..0.033 rows=76 loops=495)
                                    -> Single-row index lookup on t2 using PRIMARY (TypeId=t1.TypeId)  (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=495)
                                    -> Filter: ((pt2.FirstTypeId = t2.TypeId) and (pt2.PokemonId is not null))  (cost=6.24 rows=60) (actual time=0.008..0.027 rows=76 loops=495)
                                        -> Index lookup on pt2 using idx_FT (FirstTypeId=t1.TypeId)  (cost=6.24 rows=60) (actual time=0.008..0.019 rows=76 loops=495)
                                -> Single-row index lookup on p2 using PRIMARY (PokemonId=pt2.PokemonId)  (cost=0.88 rows=1) (actual time=0.001..0.001 rows=1 loops=37702)
-> Select #2 (subquery in projection; dependent)
    -> Group aggregate: avg(p2.Attack)  (cost=70.79 rows=60) (actual time=0.118..0.118 rows=1 loops=327)
        -> Nested loop inner join  (cost=64.82 rows=60) (actual time=0.011..0.106 rows=77 loops=327)
            -> Nested loop inner join  (cost=6.59 rows=60) (actual time=0.010..0.034 rows=77 loops=327)
                -> Single-row index lookup on t2 using PRIMARY (TypeId=t1.TypeId)  (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=327)
                -> Filter: ((pt2.FirstTypeId = t2.TypeId) and (pt2.PokemonId is not null))  (cost=6.24 rows=60) (actual time=0.008..0.029 rows=77 loops=327)
                    -> Index lookup on pt2 using idx_FT (FirstTypeId=t1.TypeId)  (cost=6.24 rows=60) (actual time=0.008..0.019 rows=77 loops=327)
            -> Single-row index lookup on p2 using PRIMARY (PokemonId=pt2.PokemonId)  (cost=0.88 rows=1) (actual time=0.001..0.001 rows=1 loops=25211)
-> Select #3 (subquery in projection; dependent)
    -> Group aggregate: avg(p2.Defense)  (cost=70.79 rows=60) (actual time=0.117..0.117 rows=1 loops=327)
        -> Nested loop inner join  (cost=64.82 rows=60) (actual time=0.011..0.107 rows=77 loops=327)
            -> Nested loop inner join  (cost=6.59 rows=60) (actual time=0.010..0.034 rows=77 loops=327)
                -> Single-row index lookup on t2 using PRIMARY (TypeId=t1.TypeId)  (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=327)
                -> Filter: ((pt2.FirstTypeId = t2.TypeId) and (pt2.PokemonId is not null))  (cost=6.24 rows=60) (actual time=0.008..0.029 rows=77 loops=327)
                    -> Index lookup on pt2 using idx_FT (FirstTypeId=t1.TypeId)  (cost=6.24 rows=60) (actual time=0.008..0.019 rows=77 loops=327)
            -> Single-row index lookup on p2 using PRIMARY (PokemonId=pt2.PokemonId)  (cost=0.88 rows=1) (actual time=0.001..0.001 rows=1 loops=25211)
```



```
-> Sort: t1.TypeName, p1.PokemonName  (actual time=281.344..281.387 rows=327 loops=1)
    -> Table scan on <temporary>  (cost=0.01..15.94 rows=1075) (actual time=0.001..0.050 rows=327 loops=1)
        -> Temporary table with deduplication  (cost=973.64..989.56 rows=1075) (actual time=281.114..281.181 rows=327 loops=1)
            -> Nested loop inner join  (cost=866.12 rows=1075) (actual time=0.708..200.402 rows=327 loops=1)
                -> Nested loop inner join  (cost=489.88 rows=1075) (actual time=0.058..10.217 rows=1075 loops=1)
                    -> Table scan on p1  (cost=113.62 rows=1075) (actual time=0.041..0.587 rows=1075 loops=1)
                    -> Filter: (pt1.FirstTypeId is not null)  (cost=0.25 rows=1) (actual time=0.009..0.009 rows=1 loops=1075)
                        -> Single-row index lookup on pt1 using PRIMARY (PokemonId=p1.PokemonId)  (cost=0.25 rows=1) (actual time=0.008..0.009 rows=1 loops=1075)
                -> Filter: ((p1.Attack > (select #4)) and (p1.Defense > (select #5)))  (cost=0.25 rows=1) (actual time=0.177..0.177 rows=0 loops=1075)
                    -> Single-row index lookup on t1 using PRIMARY (TypeId=pt1.FirstTypeId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1075)
                    -> Select #4 (subquery in condition; dependent)
                        -> Group aggregate: avg(p2.Attack)  (cost=70.79 rows=60) (actual time=0.120..0.120 rows=1 loops=1075)
                            -> Nested loop inner join  (cost=64.82 rows=60) (actual time=0.011..0.111 rows=77 loops=1075)
                                -> Nested loop inner join  (cost=6.59 rows=60) (actual time=0.009..0.035 rows=77 loops=1075)
                                    -> Single-row index lookup on t2 using PRIMARY (TypeId=t1.TypeId)  (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=1075)
                                    -> Filter: ((pt2.FirstTypeId = t2.TypeId) and (pt2.PokemonId is not null))  (cost=6.24 rows=60) (actual time=0.008..0.029 rows=77 loops=1075)
                                        -> Index lookup on pt2 using idx_FT (FirstTypeId=t1.TypeId)  (cost=6.24 rows=60) (actual time=0.008..0.019 rows=77 loops=1075)
                                -> Single-row index lookup on p2 using PRIMARY (PokemonId=pt2.PokemonId)  (cost=0.88 rows=1) (actual time=0.001..0.001 rows=1 loops=82727)
                    -> Select #5 (subquery in condition; dependent)
                        -> Group aggregate: avg(p2.Defense)  (cost=70.79 rows=60) (actual time=0.117..0.117 rows=1 loops=495)
                            -> Nested loop inner join  (cost=64.82 rows=60) (actual time=0.011..0.107 rows=76 loops=495)
                                -> Nested loop inner join  (cost=6.59 rows=60) (actual time=0.010..0.035 rows=76 loops=495)
                                    -> Single-row index lookup on t2 using PRIMARY (TypeId=t1.TypeId)  (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=495)
                                    -> Filter: ((pt2.FirstTypeId = t2.TypeId) and (pt2.PokemonId is not null))  (cost=6.24 rows=60) (actual time=0.008..0.029 rows=76 loops=495)
                                        -> Index lookup on pt2 using idx_FT (FirstTypeId=t1.TypeId)  (cost=6.24 rows=60) (actual time=0.008..0.019 rows=76 loops=495)
                                -> Single-row index lookup on p2 using PRIMARY (PokemonId=pt2.PokemonId)  (cost=0.88 rows=1) (actual time=0.001..0.001 rows=1 loops=37702)
-> Select #2 (subquery in projection; dependent)
    -> Group aggregate: avg(p2.Attack)  (cost=70.79 rows=60) (actual time=0.121..0.121 rows=1 loops=327)
        -> Nested loop inner join  (cost=64.82 rows=60) (actual time=0.012..0.109 rows=77 loops=327)
            -> Nested loop inner join  (cost=6.59 rows=60) (actual time=0.010..0.036 rows=77 loops=327)
                -> Single-row index lookup on t2 using PRIMARY (TypeId=t1.TypeId)  (cost=0.35 rows=1) (actual time=0.002..0.002 rows=1 loops=327)
                -> Filter: ((pt2.FirstTypeId = t2.TypeId) and (pt2.PokemonId is not null))  (cost=6.24 rows=60) (actual time=0.009..0.030 rows=77 loops=327)
                    -> Index lookup on pt2 using idx_FT (FirstTypeId=t1.TypeId)  (cost=6.24 rows=60) (actual time=0.008..0.020 rows=77 loops=327)
            -> Single-row index lookup on p2 using PRIMARY (PokemonId=pt2.PokemonId)  (cost=0.88 rows=1) (actual time=0.001..0.001 rows=1 loops=25211)
-> Select #3 (subquery in projection; dependent)
    -> Group aggregate: avg(p2.Defense)  (cost=70.79 rows=60) (actual time=0.121..0.121 rows=1 loops=327)
        -> Nested loop inner join  (cost=64.82 rows=60) (actual time=0.012..0.111 rows=77 loops=327)
            -> Nested loop inner join  (cost=6.59 rows=60) (actual time=0.010..0.038 rows=77 loops=327)
                -> Single-row index lookup on t2 using PRIMARY (TypeId=t1.TypeId)  (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=327)
                -> Filter: ((pt2.FirstTypeId = t2.TypeId) and (pt2.PokemonId is not null))  (cost=6.24 rows=60) (actual time=0.008..0.031 rows=77 loops=327)
                    -> Index lookup on pt2 using idx_FT (FirstTypeId=t1.TypeId)  (cost=6.24 rows=60) (actual time=0.008..0.022 rows=77 loops=327)
            -> Single-row index lookup on p2 using PRIMARY (PokemonId=pt2.PokemonId)  (cost=0.88 rows=1) (actual time=0.001..0.001 rows=1 loops=25211)
```

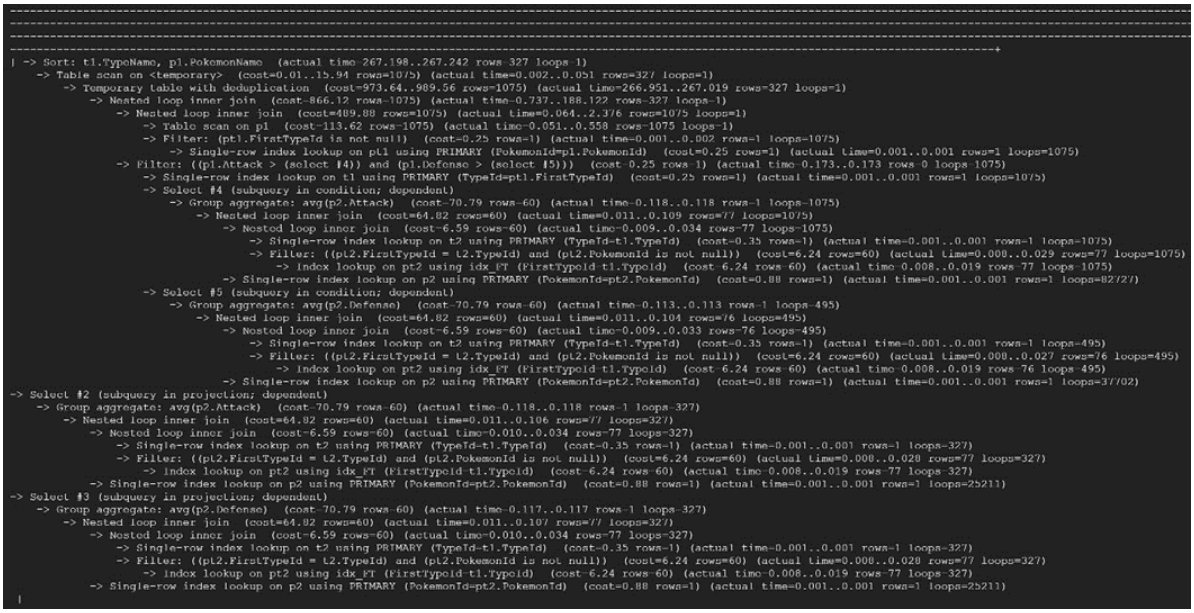
We are trying to optimize performance by putting types of Pokemons into category since it might cost less to traverse between data points. However, the truth turns out to be otherwise and none of the results are better. In addition to distorting original data structure, we consider there is another reason: the index type is B tree and sorting the data and there might be too much rotation and splits on the tree in order to maintain the new index.

Running time before indexing: 267.198, 0.118, 0.117

Running time after indexing: 281.334, 0.121, 0.121

Third try of indexing design:

Index on Generation



We have decided to choose generation as another index because as the Pokemon evolves, their abilities are getting better (and their type remains the same). As the data shows, there are slight improvements in terms of some of the nested loops joins. Overall, the performance is similar, and we think that our premises might not be entirely correct. Different generations of Pokemon does not mean they are getting better, it's more of an adjustment rather than evolution.

Running time before indexing: 267.198, 0.118, 0.117

Running time after indexing: 269.036, 0.117, 0.117

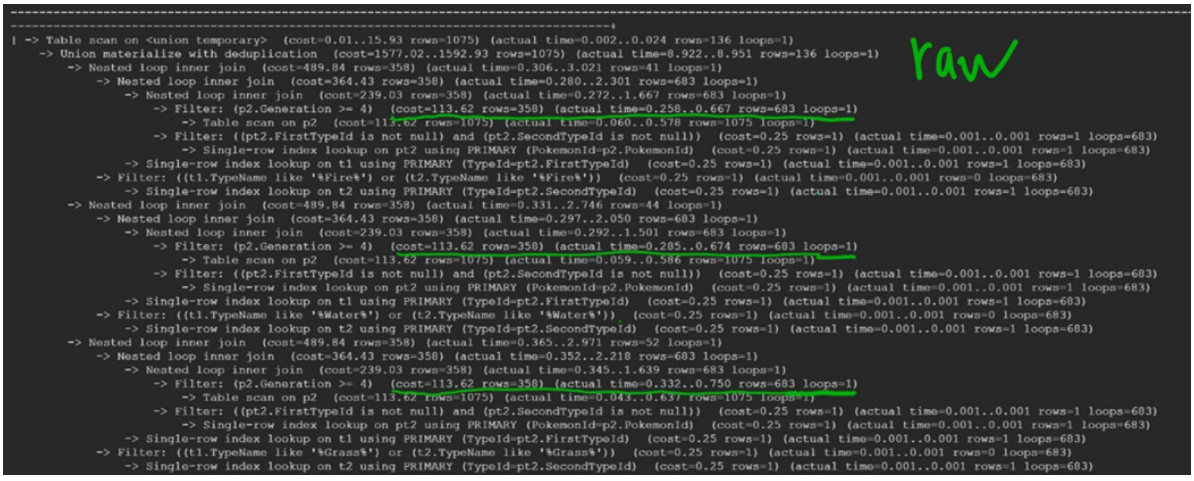
Second Query

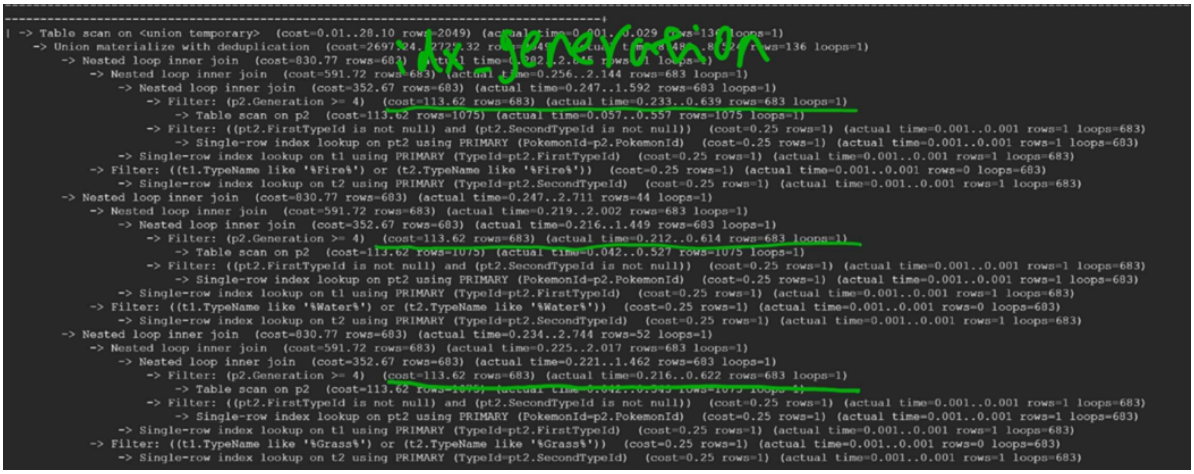
```
((SELECT p2.Total, p2.PokemonName, t1.TypeName AS FirstTypeName, t2.TypeName AS
SecondTypeName, p2.Hp, p2.Attack, p2.Defense, p2.SpecialAttack,
p2.SpecialDefense, p2.Speed, p2.Generation
FROM (Pokemon p2 LEFT JOIN Type t1 ON p2.FirstTypeId = t1.TypeId) JOIN Type t2
ON (p2.SecondTypeId =
t2.TypeId)
WHERE (t1.TypeName like '%Fire%' OR t2.TypeName like '%Fire%') AND p2.Generation
>= 4
ORDER BY p2.Total ASC)
```

```
UNION
(SELECT p2.Total, p2.PokemonName, t1.TypeName AS FirstTypeName, t2.TypeName AS
SecondTypeName, p2.Hp, p2.Attack, p2.Defense, p2.SpecialAttack,
p2.SpecialDefense, p2.Speed, p2.Generation
FROM (Pokemon p2 LEFT JOIN Type t1 ON p2.FirstTypeId = t1.TypeId) JOIN Type t2
ON (p2.SecondTypeId =
t2.TypeId)
WHERE (t1.TypeName like '%Grass%' OR t2.TypeName like '%Grass%') AND
p2.Generation
>= 4
ORDER BY p2.Total ASC)
UNION
(SELECT p2.Total, p2.PokemonName, t1.TypeName AS FirstTypeName, t2.TypeName AS
SecondTypeName, p2.Hp, p2.Attack, p2.Defense, p2.SpecialAttack,
p2.SpecialDefense, p2.Speed, p2.Generation
FROM (Pokemon p2 LEFT JOIN Type t1 ON p2.FirstTypeId = t1.TypeId) JOIN Type t2
ON (p2.SecondTypeId =
t2.TypeId)
WHERE (t1.TypeName like '%Water%' OR t2.TypeName like '%Water%') AND
p2.Generation
>= 4
ORDER BY p2.Total ASC) ORDER BY Total LIMIT 15)
;
```

First try of indexing design:

Index on **Generation**





In this experiment we have used generation as an index, since one important filters in the query is whether generation >= 4. As seen from the pictures shown above, in terms of filtering, the efficiency has greatly been improved compared to the original method. This can be explained by the less redundant pattern in searching (since nodes are categorized by generation and generations < 4 are separated)

Running time before indexing: 0.258, 0.285, 0.332

Running time after indexing: 0.233, 0.212, 0.216

Second try of indexing design:

Index on **FirstTypeId**

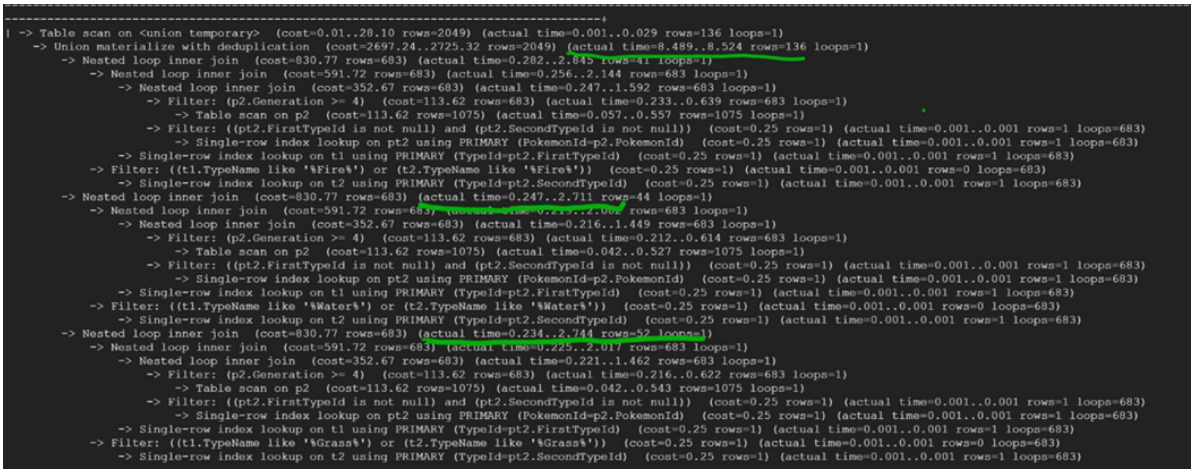
In this experiment we have used FirstTypeId as an index, in order to cluster similar types together. When it comes to the filtering results on our index, there is not too much difference since the actual time are too small to tell apart (both with values of 0.001). However, the nested loop join efficiencies are quite intriguing. Two of them became faster but one became significantly slower. We are not sure about the causation but we are quite certain that this issue is correlated with the distribution of “Fire” Pokemon in our database.

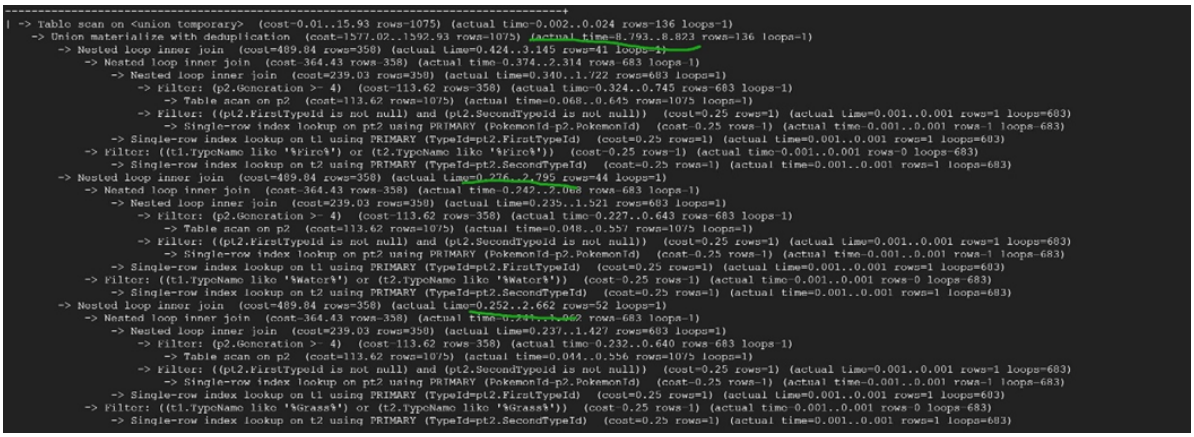
Running time before indexing: 8.92, 0.331, 0.365

Running time after indexing: 13.56, 0.236, 0.237

Third try of indexing design:

Index on **PokemonName**





Since Pokemons are like animals, their naming conventions are also categorical. We are trying to explore whether the names of Pokemon can be of value to optimizing our query. It turns out that the results are worse than the original result. So, it can be concluded that naming conventions do not have effects on the searching of certain types or generations of Pokemon. However, we believe that with more advanced “identifying methods” such as prefixes, the estimated result would be better than the present one, in which we are only using alphabetical order as index.

Running time before indexing: 9.489, 0.247, 0.234

Running time after indexing: 8.793, 0.276, 0.252