Stage 3: Database Implementation and Indexing (PART TWO)

Advanced SQL Queries

QUERY #1 (join multiple tables and set operators)

```
# Find users that have Pokemon with a primary type (pType_1) or secondary type
(pType_2) of 'Psychic'

SELECT u.user_id, u.user_name, ut.team_name,p.name
FROM users u
NATURAL JOIN user_teams ut
NATURAL JOIN user_poke_team_members uptm
JOIN pokedex_entries p
USING (pokedex_id)
WHERE p.pType_1 = "psychic"

UNION

SELECT u1.user_id,u1.user_name,ut1.team_name,p1.name
FROM users u1
NATURAL JOIN user_teams ut1
NATURAL JOIN user_poke_team_members uptm1
JOIN pokedex_entries p1
USING (pokedex_id)
WHERE p1.pType_2 = "psychic"
ORDER BY user_id;
```

Query results: first 15 rows (as it appeared in Cloud SQL Studio)

user_id	user_name	team_name	name
4	Gina Martinez	Team Raiders 1	Drowzee
7	Robin Anderson DDS	Nova Dragons	Drowzee
14	Ricky Warren	The Wolves	Mew
15	Scott Schaefer	Cyber Guardians 2	Mewtwo
16	Sabrina Whitehead	The Hunters 3	Solosis
18	Kara Walter	Squad Snipers	Alakazam
23	Frederick Garza	Ninja Wolves 2	Kadabra
25	Christopher Peck	Team Hunters 1	Drowzee
25	Christopher Peck	Team Hunters 1	Mewtwo
30	Michael King	Team Storm 3	Mewtwo
32	Barry Pruitt	Elite Guardians 1	Kadabra
58	Paul Mooney	Arcade Rangers	Mew
64	Kelli Peterson	Squad Legends 2	Abra
64	Kelli Peterson	Squad Legends 2	Hypno
75	Tonya Crosby	Nova Snipers 1	Kadabra

Explain Analyze results on this query with no added index:

```
| -> Sort: user_id  (cost=1534..1534 rows=146) (actual time=5.77..5.85 rows=310 loops=1)
    -> Table scan on <union temporary>  (cost=1410..1415 rows=146) (actual time=5.39..5.51 rows=310 loops=1)
        -> Union materialize with deduplication  (cost=1410..1410 rows=146) (actual time=5.39..5.39 rows=310 loops=1)
            -> Nested loop inner join  (cost=698 rows=72.8) (actual time=0.221..3.78 rows=310 loops=1)
                -> Nested loop inner join  (cost=443 rows=728) (actual time=0.205..2.6 rows=344 loops=1)
                    -> Nested loop inner join  (cost=188 rows=728) (actual time=0.184..1.56 rows=344 loops=1)
                        -> Filter: (p.pType_1 = 'psychic')  (cost=81.6 rows=80.1) (actual time=0.141..0.92 rows=53 loops=1)
                            -> Table scan on p  (cost=81.6 rows=801) (actual time=0.0976..0.682 rows=801 loops=1)
                        -> Covering index lookup on uptm using pokedex_id (pokedex_id=p.pokedex_id)  (cost=0.433 rows=9.09) (actual time=0.0063..0.0108 rows=6.49 loops=53)
                    -> Single-row index lookup on ut using PRIMARY (user_team_id=uptm.user_team_id)  (cost=0.25 rows=1) (actual time=0.00256..0.00262 rows=1 loops=344)
                -> Filter: (u.is_active = ut.is_active)  (cost=0.25 rows=0.1) (actual time=0.00285..0.00302 rows=0.901 loops=344)
                    -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=0.00236..0.00242 rows=1 loops=344)
            -> Nested loop inner join  (cost=698 rows=72.8) (actual time=0.847..0.847 rows=0 loops=1)
                -> Nested loop inner join  (cost=443 rows=728) (actual time=0.847..0.847 rows=0 loops=1)
                    -> Nested loop inner join  (cost=188 rows=728) (actual time=0.846..0.846 rows=0 loops=1)
                        -> Filter: (p1.pType_2 = 'psychic')  (cost=81.6 rows=80.1) (actual time=0.845..0.845 rows=0 loops=1)
                            -> Table scan on p1  (cost=81.6 rows=801) (actual time=0.0633..0.608 rows=801 loops=1)
                        -> Covering index lookup on uptm1 using pokedex_id (pokedex_id=p1.pokedex_id)  (cost=0.433 rows=9.09) (never executed)
                    -> Single-row index lookup on ut1 using PRIMARY (user_team_id=uptm1.user_team_id)  (cost=0.25 rows=1) (never executed)
                -> Filter: (u1.is_active = ut1.is_active)  (cost=0.25 rows=0.1) (never executed)
                    -> Single-row index lookup on u1 using PRIMARY (user_id=ut1.user_id)  (cost=0.25 rows=1) (never executed)
|
```

The cost at the outermost level (for sort: user_id) is 1534. There are non-primary attributes in the Where clause (pType_1 and pType2) that can be indexed on to see if it improves the cost. We can also index on user_name that is in the select clause but is not a primary key to see if it has any effect.

Explain analyze with index on pokedex_entries(PType_1). (This is a B+ Tree index because when the user creates their own index without specifying the type that is what is used).

```
| -> Sort: user_id  (cost=1227..1227 rows=121) (actual time=4.31..4.33 rows=310 loops=1)
    -> Table scan on <union temporary>  (cost=1128..1132 rows=121) (actual time=4.11..4.16 rows=310 loops=1)
        -> Union materialize with deduplication  (cost=1128..1128 rows=121) (actual time=4.11..4.11 rows=310 loops=1)
            -> Nested loop inner join  (cost=418 rows=48.2) (actual time=1.43..3.1 rows=310 loops=1)
                -> Nested loop inner join  (cost=249 rows=482) (actual time=1.42..2.39 rows=344 loops=1)
                    -> Nested loop inner join  (cost=80.3 rows=482) (actual time=1.41..1.87 rows=344 loops=1)
                        -> Index lookup on p using pokedex_pType1 (pType_1='psychic')  (cost=9.8 rows=53) (actual time=1.38..1.53 rows=53 loops=1)
                        -> Covering index lookup on uptm using pokedex_id (pokedex_id=p.pokedex_id)  (cost=0.439 rows=9.09) (actual time=0.00364..0.00578 rows=6.49 loops=53
)
                    -> Single-row index lookup on ut using PRIMARY (user_team_id=uptm.user_team_id)  (cost=0.25 rows=1) (actual time=0.00131..0.00134 rows=1 loops=344)
                -> Filter: (u.is_active = ut.is_active)  (cost=0.25 rows=0.1) (actual time=0.0018..0.00188 rows=0.901 loops=344)
                    -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=0.00163..0.00166 rows=1 loops=344)
            -> Nested loop inner join  (cost=698 rows=72.8) (actual time=0.397..0.397 rows=0 loops=1)
                -> Nested loop inner join  (cost=443 rows=728) (actual time=0.396..0.396 rows=0 loops=1)
                    -> Nested loop inner join  (cost=188 rows=728) (actual time=0.396..0.396 rows=0 loops=1)
                        -> Filter: (p1.pType_2 = 'psychic')  (cost=81.6 rows=80.1) (actual time=0.396..0.396 rows=0 loops=1)
                            -> Table scan on p1  (cost=81.6 rows=801) (actual time=0.0413..0.322 rows=801 loops=1)
                        -> Covering index lookup on uptm1 using pokedex_id (pokedex_id=p1.pokedex_id)  (cost=0.433 rows=9.09) (never executed)
                    -> Single-row index lookup on ut1 using PRIMARY (user_team_id=uptm1.user_team_id)  (cost=0.25 rows=1) (never executed)
                -> Filter: (u1.is_active = ut1.is_active)  (cost=0.25 rows=0.1) (never executed)
                    -> Single-row index lookup on u1 using PRIMARY (user_id=ut1.user_id)  (cost=0.25 rows=1) (never executed)
```

The cost at the outermost step (sort:user id) is now reduced to 1227.

explain analyze on hash index on pokedex_entries(ptype_1)

```
mysql> DROP INDEX pokedex_pType1 ON  pokedex_entries;
Query OK, 0 rows affected (0.25 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX pokedex_pType1_hash on pokedex_entries(pType_1) USING HASH;
Query OK, 0 rows affected, 1 warning (0.20 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> explain analyze SELECT u.user_id, u.user_name, ut.team_name,p.name
    -> FROM users u
    -> NATURAL JOIN user_teams ut
    -> NATURAL JOIN user_poke_team_members uptm
    -> JOIN pokedex_entries p
    -> USING (pokedex_id)
    -> WHERE p.pType_1 = "psychic"
    ->
    ->
    -> UNION
    ->
    ->
    -> SELECT u1.user_id,u1.user_name,ut1.team_name,p1.name
    -> FROM users u1
    -> NATURAL JOIN user_teams ut1
    -> NATURAL JOIN user_poke_team_members uptm1
    -> JOIN pokedex_entries p1
    -> USING (pokedex_id)
    -> WHERE p1.pType_2 = "psychic"
    -> ORDER BY user_id;
```

```
| -> Sort: user_id  (cost=1227..1227 rows=121) (actual time=3.02..3.04 rows=310 loops=1)
    -> Table scan on <union temporary>  (cost=1128..1132 rows=121) (actual time=2.86..2.91 rows=310 loops=1)
        -> Union materialize with deduplication  (cost=1128..1128 rows=121) (actual time=2.86..2.86 rows=310 loops=1)
            -> Nested loop inner join  (cost=418 rows=48.2) (actual time=0.752..2.23 rows=310 loops=1)
                -> Nested loop inner join  (cost=249 rows=482) (actual time=0.741..1.71 rows=344 loops=1)
                    -> Nested loop inner join  (cost=80.3 rows=482) (actual time=0.73..1.2 rows=344 loops=1)
                        -> Index lookup on p using pokedex_pType1_hash (pType_1='psychic')  (cost=9.8 rows=53) (actual time=0.702..0.877 rows=53 loops=1)
                        -> Covering index lookup on uptm using pokedex_id (pokedex_id=p.pokedex_id)  (cost=0.439 rows=9.09) (actual time=0.00326..0.00545 rows=6.49 loops=53)
                    -> Single-row index lookup on ut using PRIMARY (user_team_id=uptm.user_team_id)  (cost=0.25 rows=1) (actual time=0.00129..0.00132 rows=1 loops=344)
                -> Filter: (u.is_active = ut.is_active)  (cost=0.25 rows=0.1) (actual time=0.00126..0.00134 rows=0.901 loops=344)
                    -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=0.0011..0.00113 rows=1 loops=344)
            -> Nested loop inner join  (cost=698 rows=72.8) (actual time=0.354..0.354 rows=0 loops=1)
                -> Nested loop inner join  (cost=443 rows=728) (actual time=0.354..0.354 rows=0 loops=1)
                    -> Nested loop inner join  (cost=188 rows=728) (actual time=0.353..0.353 rows=0 loops=1)
                        -> Filter: (p1.pType_2 = 'psychic')  (cost=81.6 rows=80.1) (actual time=0.353..0.353 rows=0 loops=1)
                            -> Table scan on p1  (cost=81.6 rows=801) (actual time=0.0402..0.286 rows=801 loops=1)
                        -> Covering index lookup on uptm1 using pokedex_id (pokedex_id=p1.pokedex_id)  (cost=0.433 rows=9.09) (never executed)
                    -> Single-row index lookup on ut1 using PRIMARY (user_team_id=uptm1.user_team_id)  (cost=0.25 rows=1) (never executed)
                -> Filter: (u1.is_active = ut1.is_active)  (cost=0.25 rows=0.1) (never executed)
                    -> Single-row index lookup on u1 using PRIMARY (user_id=ut1.user_id)  (cost=0.25 rows=1) (never executed)
```

Choosing a hash table index instead of a B+ Tree index does not change the effect on

the cost at the outermost level. It is reduced but by the same amount to 1227.

Explain analyze on username from users.

```
mysql> CREATE INDEX username ON users(user_name);
Query OK, 0 rows affected (0.25 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze SELECT u.user_id, u.user_name, ut.team_name,p.name
    -> FROM users u
    -> NATURAL JOIN user_teams ut
    -> NATURAL JOIN user_poke_team_members uptm
    -> JOIN pokedex_entries p
    -> USING (pokedex_id)
    -> WHERE p.pType_1 = "psychic"
    ->
    ->
    -> UNION
    ->
    ->
    -> SELECT u1.user_id,u1.user_name,ut1.team_name,p1.name
    -> FROM users u1
    -> NATURAL JOIN user_teams ut1
    -> NATURAL JOIN user_poke_team_members uptm1
    -> JOIN pokedex_entries p1
    -> USING (pokedex_id)
    -> WHERE p1.pType_2 = "psychic"
    -> ORDER BY user_id;
```

```
| -> Sort: user_id  (cost=1534..1534 rows=146) (actual time=3.59..3.62 rows=310 loops=1)
    -> Table scan on <union temporary>  (cost=1410..1415 rows=146) (actual time=3.2..3.28 rows=310 loops=1)
        -> Union materialize with deduplication  (cost=1410..1410 rows=146) (actual time=3.2..3.2 rows=310 loops=1)
            -> Nested loop inner join  (cost=698 rows=72.8) (actual time=0.304..2.19 rows=310 loops=1)
                -> Nested loop inner join  (cost=443 rows=728) (actual time=0.289..1.47 rows=344 loops=1)
                    -> Nested loop inner join  (cost=188 rows=728) (actual time=0.273..0.945 rows=344 loops=1)
                        -> Filter: (p.pType_1 = 'psychic')  (cost=81.6 rows=80.1) (actual time=0.229..0.557 rows=53 loops=1)
                            -> Table scan on p  (cost=81.6 rows=801) (actual time=0.207..0.474 rows=801 loops=1)
                        -> Covering index lookup on uptm using pokedex_id (pokedex_id=p.pokedex_id)  (cost=0.433 rows=9.09) (actual time=0.00444..0.00664 rows=6.49 loops=53)
                    -> Single-row index lookup on ut using PRIMARY (user_team_id=uptm.user_team_id)  (cost=0.25 rows=1) (actual time=0.00131..0.00134 rows=1 loops=344)
                -> Filter: (u.is_active = ut.is_active)  (cost=0.25 rows=0.1) (actual time=0.00156..0.00164 rows=0.901 loops=344)
                    -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=0.00138..0.0014 rows=1 loops=344)
            -> Nested loop inner join  (cost=698 rows=72.8) (actual time=0.664..0.664 rows=0 loops=1)
                -> Nested loop inner join  (cost=443 rows=728) (actual time=0.663..0.663 rows=0 loops=1)
                    -> Nested loop inner join  (cost=188 rows=728) (actual time=0.663..0.663 rows=0 loops=1)
                        -> Filter: (p1.pType_2 = 'psychic')  (cost=81.6 rows=80.1) (actual time=0.661..0.661 rows=0 loops=1)
                            -> Table scan on p1  (cost=81.6 rows=801) (actual time=0.241..0.584 rows=801 loops=1)
                        -> Covering index lookup on uptm1 using pokedex_id (pokedex_id=p1.pokedex_id)  (cost=0.433 rows=9.09) (never executed)
                    -> Single-row index lookup on ut1 using PRIMARY (user_team_id=uptm1.user_team_id)  (cost=0.25 rows=1) (never executed)
                -> Filter: (u1.is_active = ut1.is_active)  (cost=0.25 rows=0.1) (never executed)
                    -> Single-row index lookup on u1 using PRIMARY (user_id=ut1.user_id)  (cost=0.25 rows=1) (never executed)
|
```

The outermost cost here is unchanged from when there was no added index: 1534. It makes sense that the effective indices would be on an attribute in the where clause, because it speeds up the searching for those Pokemon in the pokedex_entries table whose type is “psychic”.

The index design we would choose would be either a B+ Tree or Hash Table index on Ptype_1 in pokedex_entries. If we had more time, we would also have indexed on pType_2 in pokedex_entries to see if that was more or less effective than indexing on pType_1.

QUERY #2

```
ADVANCED SQL CONCEPTS: Join multiple relations, Subquery not easily replaced by a join
# Get the user name, team name, email, total battle count and last_battle_id for ALL active user teams
SELECT u.user_name, u.email, ut.team_name, COUNT(b.battle_id) AS total_battles,
(
SELECT MAX(b2.battle_id)
FROM battles b2
WHERE b2.user_team_id = ut.user_team_id
) AS last_battle_id
FROM users u
JOIN user_teams ut ON u.user_id = ut.user_id
JOIN battles b ON ut.user_team_id = b.user_team_id
WHERE ut.is_active = TRUE
GROUP BY u.user_id, u.user_name, u.email, ut.user_team_id, ut.team_name
ORDER BY u.user_name, ut.team_name;
```

user_name	email	team_name	total_battles	last_battle_id
Aaron James	tracy62@yahoo.com	Arcade Legends 2	2	670
Aaron Roberts	lloydjames@hotmail.com	Team Killers 4	1	980
Abigail Hale	emily05@yahoo.com	Steel Warriors 1	2	470
Adam Caldwell	luke53@yahoo.com	Dark Dragons	3	175
Adam Wallace	kimberly95@gmail.com	Omega Hunters	2	142
Alan Crawford	marqueznathan@morris.info	Clan Legends 3	2	766
Alan Crawford	marqueznathan@morris.info	Dark Bots 2	2	665
Alexa Williams	woodsnancy@aguirre.com	Clan Destroyers 2	1	504
Alicia Russell	vsmith@hotmail.com	Crimson Guardians	1	120
Alicia Russell	vsmith@hotmail.com	Steel Destroyers 2	2	738
Allen Mcclain	benjamin28@hotmail.com	Clan Legends	1	14
Allen Mcclain	benjamin28@hotmail.com	Pixel Rangers	2	202

A) Analysis of Query #2

Initially, the overall cost for the original query is 784. The cost for performing the filter on **is_active** is 0.25. The actual time for sorting on **user_name** and **team_name** is 7.36ms, but the overall cost isn't part of the output. Please see below for additional information from EXPLAIN ANALYZE:

```
------------------+
| -> Sort: u.user_name, ut.team_name  (actual time=7.36..7.42 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=6.75..6.86 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=6.75..6.75 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.111..2.65 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.102..1.67 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.0843..0.343 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.0826..0.261 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.00108..0.00116 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=915e-6..944e
-6 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=823e-6..851e-6 rows=1 loops=95
2)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00276..0.00279 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00177..0.0023 rows=
1.56 loops=612)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------
```

B) Indexing on is_active (BTree and Hash)

The overall cost for the original query is still 784. The cost for performing the filter on **is_active** is 0.25, which is the same as the original estimate. Please see below for additional information from EXPLAIN ANALYZE:

```
-------------+
| -> Sort: u.user_name, ut.team_name  (actual time=8.38..8.44 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=6.7..6.87 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=6.7..6.7 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.126..2.63 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.113..1.63 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.091..0.361 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.0884..0.28 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.001..0.00108 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=831e-6..859e
-6 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=835e-6..864e-6 rows=1 loops=95
2)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00274..0.00277 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00175..0.00227 rows
=1.56 loops=612)
 |
```

```
----------------+
| -> Sort: u.user_name, ut.team_name  (actual time=7.98..8.05 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=7.5..7.61 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=7.5..7.5 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.121..2.93 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.106..1.79 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.0914..0.372 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.0891..0.284 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.00116..0.00124 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=972e-6..0.00
1 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=965e-6..995e-6 rows=1 loops=95
2)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00305..0.00309 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00195..0.00252 rows
=1.56 loops=612)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------
```

C) Indexing on team_name (BTree and Hash)

The overall cost for the original query is still 784, so there was no change. While the cost for sorting on **user_name** and **team_name** is not explicitly stated, the **nested loop inner join** (784) is the best representation for the estimated cost. Please see below for additional information from EXPLAIN ANALYZE:

```
-----------------------------------------------------------------------------------------------------------------------------------------------
-----------------+
| -> Sort: u.user_name, ut.team_name  (actual time=9.18..9.25 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=7.53..7.67 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=7.53..7.53 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.223..2.97 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.208..1.86 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.171..0.468 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.16..0.376 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.00113..0.00122 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=946e-6..976e
-6 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=937e-6..967e-6 rows=1 loops=95
2)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00307..0.00311 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00196..0.00256 rows
=1.56 loops=612)
```

```
----------------------+
| -> Sort: u.user_name, ut.team_name  (actual time=9.59..9.65 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=9.04..9.2 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=9.04..9.04 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.101..3.39 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.0929..2.1 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.0718..0.447 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.0701..0.339 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.00134..0.00144 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=0.00113..0.0
0116 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=0.0011..0.00113 rows=1 loops=9
52)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00377..0.00381 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00239..0.00304 rows
=1.56 loops=612)
```

D) Indexing on user_name (BTree and Hash)

The overall cost for the original query is still 784, so there was no change. While the cost for sorting on **user_name** and **team_name** is not explicitly stated, the **nested loop inner join** (784) is the best representation for the estimated cost. Please see below for additional information from EXPLAIN ANALYZE:

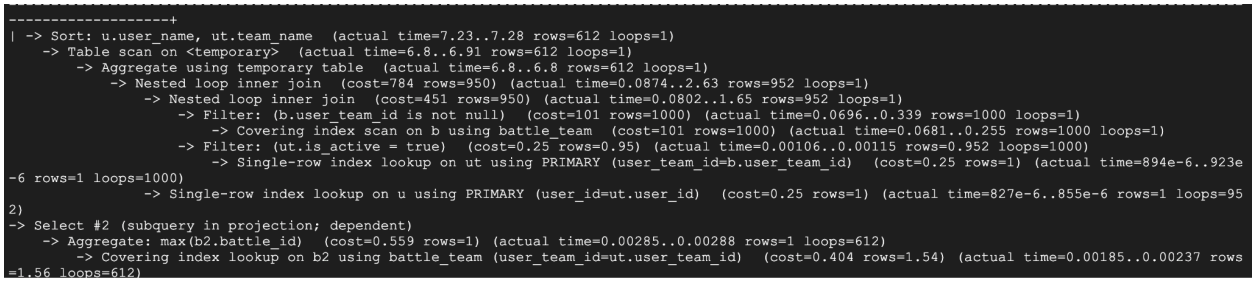
```
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------+
| -> Sort: u.user_name, ut.team_name  (actual time=7.19..7.25 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=6.76..6.86 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=6.76..6.76 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.0901..2.7 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.0828..1.72 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.0716..0.368 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.0699..0.287 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.00109..0.00118 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=931e-6..959e
-6 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=830e-6..858e-6 rows=1 loops=95
2)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00278..0.00281 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00178..0.00229 rows
=1.56 loops=612)
```

```
----------------------+
| -> Sort: u.user_name, ut.team_name  (actual time=7.55..7.6 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=7.03..7.19 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=7.02..7.02 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.0938..2.75 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.0862..1.68 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.0743..0.377 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.0723..0.285 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.00104..0.00112 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=869e-6..898e
-6 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=907e-6..936e-6 rows=1 loops=95
2)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00288..0.00292 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00185..0.00239 rows
=1.56 loops=612)
```

E) Indexing on both user_name AND team_name (BTree and Hash)

The overall cost for the original query is still 784, so there was no change. While the cost for sorting on **user_name** and **team_name** is not explicitly stated, the **nested loop inner join** (784) is the best representation for the estimated cost. Please see below for additional information from EXPLAIN ANALYZE:

```
----------------------+
| -> Sort: u.user_name, ut.team_name  (actual time=7.81..7.86 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=7.36..7.46 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=7.35..7.35 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.0974..2.85 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.0884..1.85 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.0741..0.413 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.0725..0.312 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.00118..0.00128 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=953e-6..994e
-6 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=813e-6..843e-6 rows=1 loops=95
2)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00313..0.00316 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00207..0.00264 rows
=1.56 loops=612)
```



```
-> Sort: u.user_name, ut.team_name  (actual time=7.23..7.28 rows=612 loops=1)
    -> Table scan on <temporary>  (actual time=6.8..6.91 rows=612 loops=1)
        -> Aggregate using temporary table  (actual time=6.8..6.8 rows=612 loops=1)
            -> Nested loop inner join  (cost=784 rows=950) (actual time=0.0874..2.63 rows=952 loops=1)
                -> Nested loop inner join  (cost=451 rows=950) (actual time=0.0802..1.65 rows=952 loops=1)
                    -> Filter: (b.user_team_id is not null)  (cost=101 rows=1000) (actual time=0.0696..0.339 rows=1000 loops=1)
                        -> Covering index scan on b using battle_team  (cost=101 rows=1000) (actual time=0.0681..0.255 rows=1000 loops=1)
                    -> Filter: (ut.is_active = true)  (cost=0.25 rows=0.95) (actual time=0.00106..0.00115 rows=0.952 loops=1000)
                        -> Single-row index lookup on ut using PRIMARY (user_team_id=b.user_team_id)  (cost=0.25 rows=1) (actual time=894e-6..923e
-6 rows=1 loops=1000)
                -> Single-row index lookup on u using PRIMARY (user_id=ut.user_id)  (cost=0.25 rows=1) (actual time=827e-6..855e-6 rows=1 loops=95
2)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: max(b2.battle_id)  (cost=0.559 rows=1) (actual time=0.00285..0.00288 rows=1 loops=612)
        -> Covering index lookup on b2 using battle_team (user_team_id=ut.user_team_id)  (cost=0.404 rows=1.54) (actual time=0.00185..0.00237 rows
=1.56 loops=612)
```

Overall conclusion for the indexing analysis of Query #2 is that the **estimated cost remained at 784**. This indicates that none of the indexing strategies applied to non-primary key attributes that were used in the query for sorting or filtering significantly changed the optimization.

QUERY # 3

```
# find the pokemon that have the highest type_matchup multiplier defending against
fire type moves with power at least 100
SELECT p.name, p.pokedex_id, m.move_id, m.move_name, m.move_power, MAX(tm.multiplier)
FROM moves m JOIN type_matchups tm ON m.move_type = tm.attacking_type
JOIN pokedex_entries p
ON ((tm.defending_type = p.pType_1) OR (tm.defending_type = p.pType_2))
WHERE m.move_type = 'fire'
AND m.move_power >= '100'
GROUP BY p.name, p.pokedex_id, m.move_id, m.move_name, m.move_power
ORDER BY p.name;
```

Query results (first 15 rows):

```
mysql> SELECT p.name, p.pokedex_id, m.move_id, m.move_name, m.move_power, MAX(tm.multipli
    -> FROM moves m JOIN type_matchups tm ON m.move_type = tm.attacking_type
    -> JOIN pokedex_entries p
    -> ON ((tm.defending_type = p.pType_1) OR (tm.defending_type = p.pType_2))
    -> WHERE m.move_type = 'fire'
    -> AND m.move_power >= '100'
    -> GROUP BY p.name, p.pokedex_id, m.move_id, m.move_name, m.move_power
    -> ORDER BY p.name LIMIT 15;
+-----------+------------+---------+--------------+------------+--------------------+
| name      | pokedex_id | move_id | move_name    | move_power | MAX(tm.multiplier) |
+-----------+------------+---------+--------------+------------+--------------------+
| Abomasnow |        460 |     126 | Fire Blast   |        110 |                  2 |
| Abomasnow |        460 |     221 | Sacred Fire  |        100 |                  2 |
| Abomasnow |        460 |     284 | Eruption     |        150 |                  2 |
| Abomasnow |        460 |     307 | Blast Burn   |        150 |                  2 |
| Abomasnow |        460 |     315 | Overheat     |        130 |                  2 |
| Abomasnow |        460 |     394 | Flare Blitz  |        120 |                  2 |
| Abomasnow |        460 |     463 | Magma Storm  |        100 |                  2 |
| Abomasnow |        460 |     517 | Inferno      |        100 |                  2 |
| Abomasnow |        460 |     545 | Searing Shot |        100 |                  2 |
| Abomasnow |        460 |     551 | Blue Flare   |        130 |                  2 |
| Abomasnow |        460 |     557 | V-create     |        180 |                  2 |
| Abomasnow |        460 |     558 | Fusion Flare |        100 |                  2 |
| Abomasnow |        460 |     682 | Burn Up      |        130 |                  2 |
| Abomasnow |        460 |     704 | Shell Trap   |        150 |                  2 |
| Abomasnow |        460 |     720 | Mind Blown   |        150 |                  2 |
+-----------+------------+---------+--------------+------------+--------------------+
```

Explain analyze on the query with no index

```
| -> Sort: p.`name`, p.pokedex_id, m.move_id, m.move_name, m.move_power  (actual time=142..146 rows=12015 loops=1)
    -> Table scan on <temporary>  (actual time=117..120 rows=12015 loops=1)
        -> Aggregate using temporary table  (actual time=117..117 rows=12015 loops=1)
            -> Filter: ((p.pType_1 = tm.defending_type) or (p.pType_2 = tm.defending_type))  (cost=6484 rows=11965) (actual time=0.862..83.7 rows=12030 loops=1)
                -> Inner hash join (no condition)  (cost=6484 rows=11965) (actual time=0.805..37.1 rows=216270 loops=1)
                    -> Table scan on p  (cost=0.0241 rows=801) (actual time=0.0427..1.08 rows=801 loops=1)
                    -> Hash
                        -> Inner hash join (no condition)  (cost=183 rows=78.6) (actual time=0.446..0.629 rows=270 loops=1)
                            -> Filter: (tm.attacking_type = 'fire')  (cost=1.37 rows=32.4) (actual time=0.0261..0.171 rows=18 loops=1)
                                -> Table scan on tm  (cost=1.37 rows=324) (actual time=0.0222..0.138 rows=324 loops=1)
                            -> Hash
                                -> Filter: ((m.move_type = 'fire') and (m.move_power >= 100))  (cost=74.1 rows=24.3) (actual time=0.134..0.39 rows=15 loops=1)
                                    -> Table scan on m  (cost=74.1 rows=728) (actual time=0.0779..0.314 rows=728 loops=1)
|
```

The outermost cost is 6484. We will try to reduce it by indexing on the join attributes.

B+Tree index on moves(move_type)

```
mysql> create index mtype on moves(move_type);
Query OK, 0 rows affected (0.11 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Sort: p.`name`, p.pokedex_id, m.move_id, m.move_name, m.move_power  (actual time=119..120 rows=12015 loops=1)
    -> Table scan on <temporary>  (actual time=102..105 rows=12015 loops=1)
        -> Aggregate using temporary table  (actual time=102..102 rows=12015 loops=1)
            -> Filter: ((p.pType_1 = tm.defending_type) or (p.pType_2 = tm.defending_type))  (cost=3190 rows=5917) (actual time=1.21..72.6 rows=12030 loops=1)
                -> Inner hash join (no condition)  (cost=3190 rows=5917) (actual time=1.1..32.4 rows=216270 loops=1)
                    -> Table scan on p  (cost=0.0459 rows=801) (actual time=0.0604..1.07 rows=801 loops=1)
                    -> Hash
                        -> Inner hash join (no condition)  (cost=73.6 rows=38.9) (actual time=0.593..0.829 rows=270 loops=1)
                            -> Filter: (tm.attacking_type = 'fire')  (cost=2.75 rows=32.4) (actual time=0.0412..0.233 rows=18 loops=1)
                                -> Table scan on tm  (cost=2.75 rows=324) (actual time=0.0328..0.189 rows=324 loops=1)
                            -> Hash
                                -> Filter: (m.move_power >= 100)  (cost=4.95 rows=12) (actual time=0.336..0.525 rows=15 loops=1)
                                    -> Index lookup on m using mtype (move_type='fire')  (cost=4.95 rows=36) (actual time=0.327..0.514 rows=36 loops=1)
|
```

This reduces the outermost cost to 3190.

B+Tree index on type_matchups(attacking_type)

```
mysql> create index tmtype on type_matchups(attacking_type);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Sort: p.`name`, p.pokedex_id, m.move_id, m.move_name, m.move_power  (actual time=127..128 rows=12015 loops=1)
    -> Table scan on <temporary>  (actual time=113..115 rows=12015 loops=1)
        -> Aggregate using temporary table  (actual time=113..113 rows=12015 loops=1)
            -> Filter: ((p.pType_1 = tm.defending_type) or (p.pType_2 = tm.defending_type))  (cost=1288 rows=2215) (actual time=1.71..80.8 rows=12030 loops=1)
                -> Inner hash join (no condition)  (cost=1288 rows=2215) (actual time=1.7..36.3 rows=216270 loops=1)
                    -> Table scan on p  (cost=0.0412 rows=801) (actual time=0.0751..1.12 rows=801 loops=1)
                    -> Hash
                        -> Inner hash join (no condition)  (cost=119 rows=14.6) (actual time=0.343..1.53 rows=270 loops=1)
                            -> Filter: ((m.move_type = 'fire') and (m.move_power >= 100))  (cost=4.18 rows=24.3) (actual time=0.171..1.28 rows=15 loops=1)
                                -> Table scan on m  (cost=4.18 rows=728) (actual time=0.0747..1.14 rows=728 loops=1)
                            -> Hash
                                -> Index lookup on tm using tmtype (attacking_type='fire')  (cost=2.55 rows=18) (actual time=0.131..0.141 rows=18 loops=1)
```

This reduces the outermost cost to 1288.

B+Tree index on type_matchups(defending_type)

```
mysql> create index dtype on type_matchups(defending_type);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Sort: p.`name`, p.pokedex_id, m.move_id, m.move_name, m.move_power  (actual time=596..597 rows=12015 loops=1)
    -> Table scan on <temporary>  (actual time=582..585 rows=12015 loops=1)
        -> Aggregate using temporary table  (actual time=582..582 rows=12015 loops=1)
            -> Nested loop inner join  (cost=633116 rows=68025) (actual time=2.9..547 rows=12030 loops=1)
                -> Inner hash join (no condition)  (cost=2019 rows=19436) (actual time=0.446..3.83 rows=12015 loops=1)
                    -> Table scan on p  (cost=3.36 rows=801) (actual time=0.0398..0.91 rows=801 loops=1)
                    -> Hash
                        -> Filter: ((m.move_type = 'fire') and (m.move_power >= 100))  (cost=74.1 rows=24.3) (actual time=0.132..0.391 rows=15 loops=1)
                            -> Table scan on m  (cost=74.1 rows=728) (actual time=0.0756..0.316 rows=728 loops=1)
                -> Filter: ((tm.attacking_type = 'fire') and ((tm.defending_type = p.pType_1) or (tm.defending_type = p.pType_2)))  (cost=0.0711 rows=3.5) (actual time=0.038
8..0.0449 rows=1 loops=12015)
                    -> Index range scan on tm (re-planned for each iteration)  (cost=0.0711 rows=324) (actual time=0.0378..0.0426 rows=18 loops=12015)
```

This increases the outermost cost to 633116! This may be because there are many type matchups with the same defending_type.

B+tree index on pokedex_entries(pType_1)

```
mysql> create index ptype1 on pokedex_entries(pType_1);
Query OK, 0 rows affected (0.18 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Sort: p.`name`, p.pokedex_id, m.move_id, m.move_name, m.move_power  (actual time=99.9..101 rows=12015 loops=1)
    -> Table scan on <temporary>  (actual time=85.7..88.1 rows=12015 loops=1)
        -> Aggregate using temporary table  (actual time=85.7..85.7 rows=12015 loops=1)
            -> Filter: ((p.pType_1 = tm.defending_type) or (p.pType_2 = tm.defending_type))  (cost=6484 rows=9446) (actual time=0.811..59.3 rows=12030 loops=1)
                -> Inner hash join (no condition)  (cost=6484 rows=9446) (actual time=0.761..26.2 rows=216270 loops=1)
                    -> Table scan on p  (cost=0.0201 rows=801) (actual time=0.0383..0.718 rows=801 loops=1)
                    -> Hash
                        -> Inner hash join (no condition)  (cost=183 rows=78.6) (actual time=0.504..0.654 rows=270 loops=1)
                            -> Filter: (tm.attacking_type = 'fire')  (cost=1.37 rows=32.4) (actual time=0.026..0.152 rows=18 loops=1)
                                -> Table scan on tm  (cost=1.37 rows=324) (actual time=0.022..0.125 rows=324 loops=1)
                            -> Hash
                                -> Filter: ((m.move_type = 'fire') and (m.move_power >= 100))  (cost=74.1 rows=24.3) (actual time=0.121..0.464 rows=15 loops=1)
                                    -> Table scan on m  (cost=74.1 rows=728) (actual time=0.0668..0.385 rows=728 loops=1)
```

This leaves the outermost cost unchanged at 6484.

B+tree index on pokedex_entries(pType_2). This

```
mysql> create index ptype2 on pokedex_entries(pType_2);
Query OK, 0 rows affected (0.20 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
-> Sort: p.`name`, p.pokedex_id, m.move_id, m.move_name, m.move_power  (actual time=95.9..97.2 rows=12015 loops=1)
  -> Table scan on <temporary>  (actual time=79.9..82.1 rows=12015 loops=1)
      -> Aggregate using temporary table  (actual time=79.9..79.9 rows=12015 loops=1)
          -> Filter: ((p.pType_1 = tm.defending_type) or (p.pType_2 = tm.defending_type))  (cost=6484 rows=9131) (actual time=0.788..57.8 rows=12030 loops=1)
              -> Inner hash join (no condition)  (cost=6484 rows=9131) (actual time=0.738..25.3 rows=216270 loops=1)
                  -> Table scan on p  (cost=0.0196 rows=801) (actual time=0.0394..0.748 rows=801 loops=1)
                  -> Hash
                      -> Inner hash join (no condition)  (cost=183 rows=78.6) (actual time=0.498..0.649 rows=270 loops=1)
                          -> Filter: (tm.attacking_type = 'fire')  (cost=1.37 rows=32.4) (actual time=0.0257..0.153 rows=18 loops=1)
                              -> Table scan on tm  (cost=1.37 rows=324) (actual time=0.0217..0.124 rows=324 loops=1)
                          -> Hash
                              -> Filter: ((m.move_type = 'fire') and (m.move_power >= 100))  (cost=74.1 rows=24.3) (actual time=0.188..0.458 rows=15 loops=1)
                                  -> Table scan on m  (cost=74.1 rows=728) (actual time=0.131..0.381 rows=728 loops=1)
```

Analysis of Query 4

```
# The players has a team whose average current hp is higher than the average hp of all pokemons
SELECT user_id, user_team_id, team_name
FROM user_teams NATURAL JOIN user_poke_team_members
WHERE current_hp > 100
GROUP BY user_team_id
HAVING AVG(current_hp) > (SELECT AVG(hp) AS avg_hp FROM pokedex_entries)
ORDER BY user_id;
```

```
mysql> SELECT user_id, user_team_id, team_name
    -> FROM user_teams NATURAL JOIN user_poke_team_members
    -> WHERE current_hp > 100
    -> GROUP BY user_team_id
    -> HAVING AVG(current_hp) > (SELECT AVG(hp) AS avg_hp FROM pokedex_entries)
    -> ORDER BY user_id;
+---------+--------------+----------------------+
| user_id | user_team_id | team_name            |
+---------+--------------+----------------------+
|         |          925 | Cyber Storm 3        |
|         |          596 | The Titans 2         |
|       4 |          276 | Team Raiders 1       |
|         |          140 | Alpha Snipers        |
|         |          111 | Cyber Raiders        |
|      15 |          658 | Cyber Guardians 2    |
|       8 |          586 | The Warriors 2       |
|       5 |          277 | Team Hunters 1       |
|         |          820 | Team Storm 3         |
|      32 |          483 | Pixel Legends 1      |
|      38 |          537 | Steel Blasters 1     |
|       2 |          197 | Arcade Killers       |
|         |          704 | Omega Bots 2         |
|      49 |          568 | Clan Guardians 2     |
|      54 |          968 | Omega Dragons 3      |
|         |          295 | Clan Storm 1         |
|         |           52 | The Hunters          |
|       8 |          210 | Arcade Rangers       |
|      64 |          207 | Arcade Blasters      |
|         |          173 | Ninja Dragons        |
```

```
         | |          504 | Nova Wolves 1
       |7 |          992 | Dark Killers 3
      |67 |          981 | Ninja Raiders 3
        | |          180 | Ninja Rangers
       |8 |         1095 | Team Rangers 4
      |71 |          527 | Steel Killers 1
      |77 |          922 | Cyber Hunters 3
      |77 |          598 | The Guardians 2
        | |          910 | Ghost Storm 3
       |0 |          557 | Clan Killers 2
 |   1091 |          394 | Crimson Destroyers 1
      |91 |          103 | Ghost Guardians
      |95 |          425 | Omega Snipers 1
         ||            7 | Team Hunters
      |98 |          975 | Omega Rangers 3
        | |          158 | Omega Dragons
     |100 |         1007 | Arcade Killers 3
        | |          731 | Dark Titans 2
+---------+--------------+----------------------+
386 rows in set (0.25 sec)
```

user_id	user_team_id	team_name
1	925	Cyber Storm 3
2	596	The Titans 2
4	276	Team Raiders 1
6	111	Cyber Raiders
6	140	Alpha Snipers
15	658	Cyber Guardians 2
18	586	The Warriors 2
25	277	Team Hunters 1
30	820	Team Storm 3
32	483	Pixel Legends 1
38	537	Steel Blasters 1
42	197	Arcade Killers
43	704	Omega Bots 2
49	568	Clan Guardians 2
54	968	Omega Dragons 3

Analysis before adding index:

The query runs slow mainly because it joins user_teams with user_poke_team_members using a nested loop. For each of the 1100 teams, it looks up matching members and filters by current_hp > 100, which takes time. Then it groups the results and calculates the average HP for each team. This part alone takes over 5 seconds. The subquery that gets the average HP from pokedex_entries is fast. Overall, most of the time is spent on the join and the group-by calculation.

```
------+
| -> Sort: user_teams.user_id  (actual time=6.44..6.46 rows=386 loops=1)
    -> Filter: (`avg(user_poke_team_members.current_hp)` > (select #2))  (actual time=0.665..6.27 rows=386 loops=1)
        -> Stream results  (cost=1270 rows=1100) (actual time=0.165..5.71 rows=386 loops=1)
            -> Group aggregate: avg(user_poke_team_members.current_hp)  (cost=1270 rows=1100) (actual time=0.158..5.56 rows=386 loops=1)
                -> Nested loop inner join  (cost=1050 rows=2200) (actual time=0.135..5.42 rows=463 loops=1)
                    -> Index scan on user_teams using PRIMARY  (cost=111 rows=1100) (actual time=0.0702..0.405 rows=1100 loops=1)
                    -> Filter: (user_poke_team_members.current_hp > 100)  (cost=0.254 rows=2) (actual time=0.004..0.00441 rows=0.421 loops=1100)
                        -> Index lookup on user_poke_team_members using PRIMARY (user_team_id=user_teams.user_team_id)  (cost=0.254 rows=6) (actual time=0.0028
4..0.00389 rows=6 loops=1100)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(pokedex_entries.hp)  (cost=162 rows=1) (actual time=0.337..0.337 rows=1 loops=1)
                -> Table scan on pokedex_entries  (cost=81.6 rows=801) (actual time=0.0495..0.262 rows=801 loops=1)
```

A) Analysis after adding index on current_hp:

After adding an index on current_hp, the query becomes much faster and more efficient. Instead of scanning all teams first, it now starts by using the index to quickly find only the user_poke_team_members where current_hp > 100. This uses an index range scan, which is much quicker than checking every row. Then, for each matching member, it does a fast index lookup to get their team info. Compared to before, the nested loop runs fewer times (only 463 instead of 1100), and the total time drops from over 6 seconds to under 2 seconds. The main improvement comes from filtering early with the index and reducing unnecessary lookups.

```
mysql> create index chp on user_poke_team_members(current_hp);
Query OK, 0 rows affected (0.40 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze SELECT user_id, user_team_id, team_name
    -> FROM user_teams NATURAL JOIN user_poke_team_members
    -> WHERE current_hp > 100
    -> GROUP BY user_team_id
    -> HAVING AVG(current_hp) > (SELECT AVG(hp) AS avg_hp FROM pokedex_entries)
    -> ORDER BY user_id;
+----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN
|
|
|
|
|                                                                                                                                                                |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: user_teams.user_id  (actual time=2.05..2.08 rows=386 loops=1)
    -> Filter: (??? > (select #2))  (actual time=1.79..1.89 rows=386 loops=1)
        -> Table scan on <temporary>  (actual time=1.44..1.5 rows=386 loops=1)
            -> Aggregate using temporary table  (actual time=1.44..1.44 rows=386 loops=1)
                -> Nested loop inner join  (cost=256 rows=463) (actual time=0.124..0.981 rows=463 loops=1)
                    -> Filter: (user_poke_team_members.current_hp > 100)  (cost=94.4 rows=463) (actual time=0.101..0.27 rows=463 loops=1)
                        -> Covering index range scan on user_poke_team_members using chp over (100 < current_hp)  (cost=94.4 rows=463) (actual time=0.0957..0.2
24 rows=463 loops=1)
                    -> Single-row index lookup on user_teams using PRIMARY (user_team_id=user_poke_team_members.user_team_id)  (cost=0.25 rows=1) (actual time=
0.00132..0.00135 rows=1 loops=463)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(pokedex_entries.hp)  (cost=162 rows=1) (actual time=0.318..0.318 rows=1 loops=1)
                -> Table scan on pokedex_entries  (cost=81.6 rows=801) (actual time=0.0303..0.244 rows=801 loops=1)
```

B) Analysis on adding index on current_hp using hash:

With a hash index on current_hp, the query is faster than the original (no index) but slower than the B-tree index version. The plan still uses an index scan, but hash indexes are optimized for exact matches, not range conditions like current_hp > 100. As a result, the filtering step (current_hp > 100) is less efficient, taking around

0.9–1.0 seconds versus 0.1–0.27 seconds with the B-tree index. The total time is around 3.5 seconds, which is better than the original 6+ seconds but nearly 2x slower than the B-tree index version.

```
mysql> create index chp_hash on user_poke_team_members(current_hp);
Query OK, 0 rows affected (0.60 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze SELECT user_id, user_team_id, team_name
    -> FROM user_teams NATURAL JOIN user_poke_team_members
    -> WHERE current_hp > 100
    -> GROUP BY user_team_id
    -> HAVING AVG(current_hp) > (SELECT AVG(hp) AS avg_hp FROM pokedex_entries)
    -> ORDER BY user_id;
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------+
| EXPLAIN
                                                                            |
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------+
| -> Sort: user_teams.user_id  (actual time=3.54..3.57 rows=386 loops=1)
    -> Filter: (??? > (select #2))  (actual time=3.23..3.35 rows=386 loops=1)
        -> Table scan on <temporary>  (actual time=2.17..2.24 rows=386 loops=1)
            -> Aggregate using temporary table  (actual time=2.16..2.16 rows=386 loops=1)
                -> Nested loop inner join  (cost=256 rows=463) (actual time=0.935..1.78 rows=463 loops=1)
                    -> Filter: (user_poke_team_members.current_hp > 100)  (cost=94.4 rows=463) (actual time=0.914..1.07 rows=463 loops=1)
                        -> Covering index range scan on user_poke_team_members using chp_hash over (100 < current_hp)  (cost=94.4 rows=463) (actual time=0.104.
.0.223 rows=463 loops=1)
                    -> Single-row index lookup on user_teams using PRIMARY (user_team_id=user_poke_team_members.user_team_id)  (cost=0.25 rows=1) (actual time=
0.00132..0.00135 rows=1 loops=463)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(pokedex_entries.hp)  (cost=162 rows=1) (actual time=0.315..0.316 rows=1 loops=1)
```

C) Analysis on adding index on team_member:

Since the query doesn’t filter, group, or sort by team_name, and it isn’t used in any WHERE, HAVING, or ORDER BY clauses, the index on team_name is not used. As a result, the execution plan, cost, and runtime stay the same, with sorting still relying on the primary key (user_id).

```
mysql> create index name_index on user_teams(team_name);
Query OK, 0 rows affected, 1 warning (0.29 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> explain analyze SELECT user_id, user_team_id, team_name
    -> FROM user_teams NATURAL JOIN user_poke_team_members
    -> WHERE current_hp > 100
    -> GROUP BY user_team_id
    -> HAVING AVG(current_hp) > (SELECT AVG(hp) AS avg_hp FROM pokedex_entries)
    -> ORDER BY user_id;
+---------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
------+
| EXPLAIN





                                                                          |
+---------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
------+
| -> Sort: user_teams.user_id  (actual time=6.41..6.43 rows=386 loops=1)
    -> Filter: (`avg(user_poke_team_members.current_hp)` > (select #2))  (actual time=0.593..6.23 rows=386 loops=1)
        -> Stream results  (cost=1270 rows=1100) (actual time=0.163..5.75 rows=386 loops=1)
            -> Group aggregate: avg(user_poke_team_members.current_hp)  (cost=1270 rows=1100) (actual time=0.155..5.6 rows=386 loops=1)
                -> Nested loop inner join  (cost=1050 rows=2200) (actual time=0.134..5.42 rows=463 loops=1)
                    -> Index scan on user_teams using PRIMARY  (cost=111 rows=1100) (actual time=0.0779..0.408 rows=1100 loops=1)
                    -> Filter: (user_poke_team_members.current_hp > 100)  (cost=0.254 rows=2) (actual time=0.00405..0.00441 rows=0.421 loops=1100)
                        -> Index lookup on user_poke_team_members using PRIMARY (user_team_id=user_teams.user_team_id)  (cost=0.254 rows=6) (actual time=0.0029
..0.00391 rows=6 loops=1100)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(pokedex_entries.hp)  (cost=162 rows=1) (actual time=0.407..0.407 rows=1 loops=1)
                -> Table scan on pokedex_entries  (cost=81.6 rows=801) (actual time=0.0279..0.292 rows=801 loops=1)
```

D) Analysis on adding hash index on team_member:

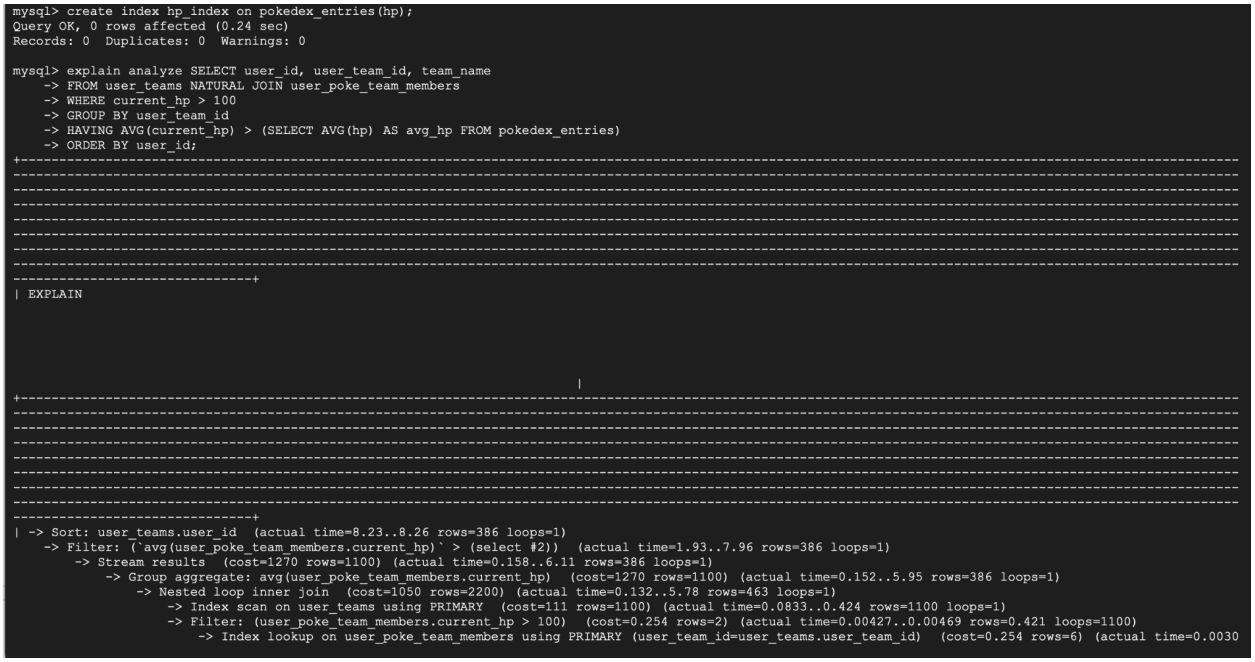
After adding a hash index on team_member, the query plan and performance remain unchanged. The query does not filter, group, or join using the team_member column, nor is it referenced in any WHERE, HAVING, or ORDER BY clauses. As a result, the hash index is not used. The total execution time stays around 6 seconds, and the plan structure is identical to the one without the hash index. This shows that indexing a column not involved in the query logic has no impact on performance.

```
mysql> create index name_hash on user_teams(team_name);
Query OK, 0 rows affected, 1 warning (0.26 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> explain analyze SELECT user_id, user_team_id, team_name
    -> FROM user_teams NATURAL JOIN user_poke_team_members
    -> WHERE current_hp > 100
    -> GROUP BY user_team_id
    -> HAVING AVG(current_hp) > (SELECT AVG(hp) AS avg_hp FROM pokedex_entries)
    -> ORDER BY user_id;
+---------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
-------+
| EXPLAIN
                                                                                  |
+---------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
-------+
| -> Sort: user_teams.user_id  (actual time=6.16..6.19 rows=386 loops=1)
    -> Filter: (`avg(user_poke_team_members.current_hp)` > (select #2))  (actual time=0.413..5.92 rows=386 loops=1)
        -> Stream results  (cost=1270 rows=1100) (actual time=0.11..5.57 rows=386 loops=1)
            -> Group aggregate: avg(user_poke_team_members.current_hp)  (cost=1270 rows=1100) (actual time=0.106..5.39 rows=386 loops=1)
                -> Nested loop inner join  (cost=1050 rows=2200) (actual time=0.0885..5.25 rows=463 loops=1)
                    -> Index scan on user_teams using PRIMARY  (cost=111 rows=1100) (actual time=0.0487..0.356 rows=1100 loops=1)
                    -> Filter: (user_poke_team_members.current_hp > 100)  (cost=0.254 rows=2) (actual time=0.00397..0.00432 rows=0.421 loops=1100)
                        -> Index lookup on user_poke_team_members using PRIMARY (user_team_id=user_teams.user_team_id)  (cost=0.254 rows=6) (actual time=0.0028
5..0.00382 rows=6 loops=1100)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(pokedex_entries.hp)  (cost=162 rows=1) (actual time=0.289..0.289 rows=1 loops=1)
                -> Table scan on pokedex_entries  (cost=81.6 rows=801) (actual time=0.024..0.218 rows=801 loops=1)
  |
```

E) Analysis on adding index on pokedex_entries(hp):

Adding an index on hp made the query slower because the query just calculates the average of all hp values ,as it doesn’t filter or search by hp. Before the index, the database used a simple full table scan, which is fast for reading all rows. After adding the index, it used the index to scan, but that can be slower when reading the whole table because it doesn’t access the data in a straight line. So in this case, the index wasn’t helpful and made the scan a bit less efficient.



F) Final index design

The best index for current_hp is a regular B-tree index because it supports range conditions like > 100, which this query uses. In testing, it gave the best performance, cutting execution time to around 2 seconds, much faster than using no index or a hash index. Hash indexes are only good for exact matches and were slower in this case. The B-tree index also helps with sorting or other future queries, making it the most effective and flexible choice.

For other attributes such as team_name, has no effect by index because the query doesn't filter, group, or sort by those columns, so the database never uses those indexes. Attribute like hp, since the subquery scan of all hp values in pokedex_entries, therefore, index is not helpful or even worse in this case.

How we implemented changes from feedback and chat in Campuswire:

TA Comments from Stage 2:

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

2. Instead of connecting type matchup with pokemon id, you can have a matrix that you can filter by pokemon type 1 and 2. Refer to this: https://pokemondb.net/type. If this does not help, feel free to ignore.

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

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

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

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

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

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

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

```
CREATE TABLE type_matchups(
matchup_id INT PRIMARY KEY AUTO_INCREMENT,
attacking_type VARCHAR(30),
defending_type VARCHAR(30),
multiplier REAL
);
```

The attacking type refers to the "type" in the moves section (ie, it is the type of the attacking move and not necessarily the ptype_1 or ptype_2 of the attacking Pokemon). The defending type refers to either the ptype_1 or the ptype_2 of the defending pokemon (whichever type matchup you are looking up). Calculating the type effect of an attack on a Pokemon involves looking up two

multipliers from the pokemon_moves table: one multiplier for the match between the attacking move type and the defending Pokemon's ptype_1 and another lookup for the match between the attacking move type and the defending Pokemon's ptype_2, if any (this can be null so you may want to do a right/left join when querying for it). These two multipliers are then multiplied by each other to give the total_defensive_multiplier as affected by types. Here is a sample query:

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

This query may not be worded in the best way, and we may change it to involve a set operator instead, but I hope it gives you an idea of how the type matchups work. We may need to add rows to the table for null types (which can occur with ptype2) and a multiplier of 1 so that there will be no effect on the total defensive multiplier from a null ptype2.

There is a question remaining though of what relationships the type matchups table has with the pokedex pntries table (where PType1 and Ptype2 are) and the moves table. If these relationships are many to many, will we need additional tables for both of them with the primary keys of two tables in each relationship table?

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

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

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

it) which four moves they want each of their Pokemon to have in battle when they assemble their team.

Please see our updated UML diagram at the top of Part 1 of our Stage 3 Database Design submission for more details.