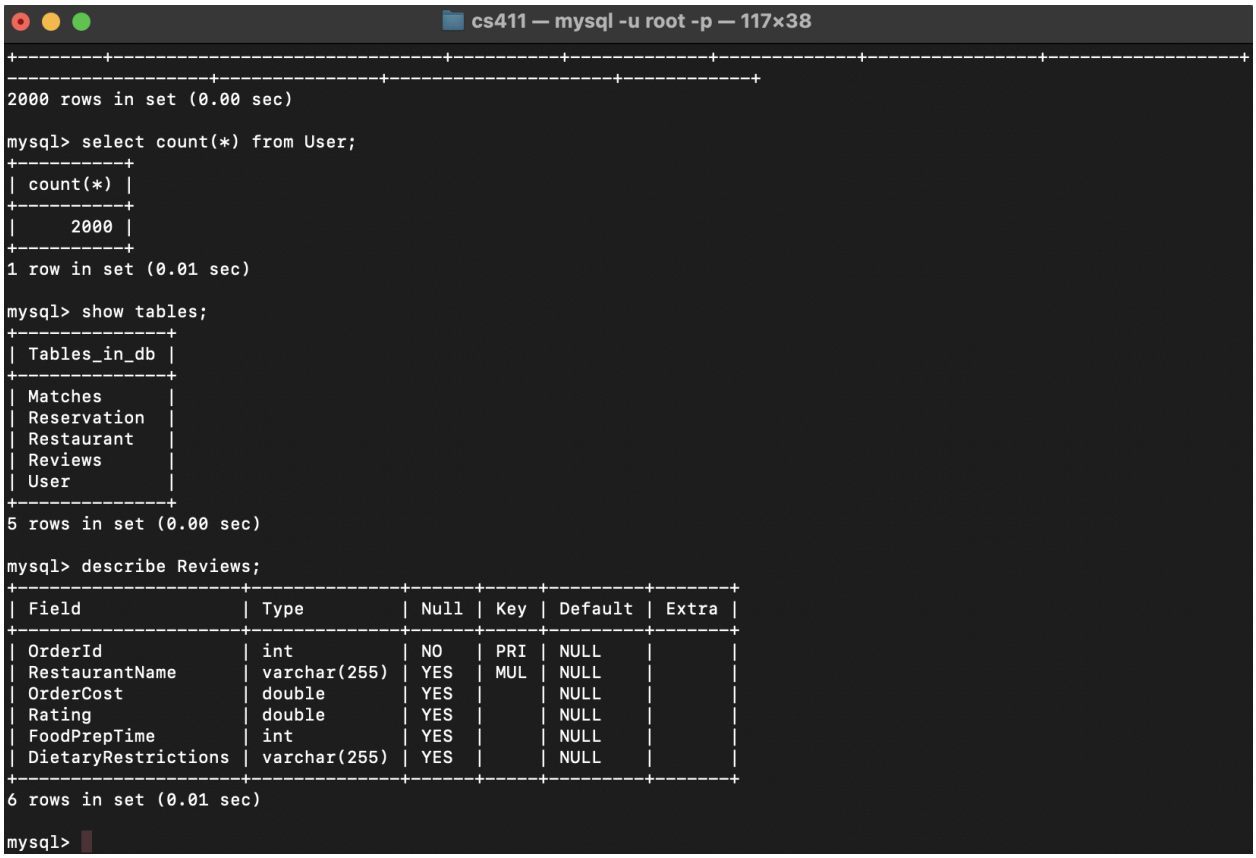
DDL Commands:

```
CREATE TABLE User (UserId INT PRIMARY KEY,
Email VARCHAR(255),
Password VARCHAR(255),
FirstName VARCHAR(255),
LastName VARCHAR(255),
GenderIdentity VARCHAR(255),
GenderPreference VARCHAR(255),
CuisinePreference VARCHAR(255),
MaximumBudget VARCHAR(255),
OptimalLengthOfDate INT,
Allergies VARCHAR(500));

CREATE TABLE Restaurant (RestaurantName VARCHAR(255) PRIMARY KEY,
Cuisine VARCHAR(255),
AverageRating REAL,
Address VARCHAR(500),
Email VARCHAR(255)
);

CREATE TABLE Matches (MatchId INT PRIMARY KEY,
UserIdA INT,
UserIdB INT,
Description VARCHAR(255),

FOREIGN KEY(UserIdA) REFERENCES User(UserId)
    ON DELETE CASCADE,

FOREIGN KEY(UserIdB) REFERENCES User(UserId)
    ON DELETE CASCADE);

CREATE TABLE Reservation (ReservationId INT PRIMARY KEY,
RestaurantName VARCHAR(255),
MatchId INT,
Time VARCHAR(255),
Date VARCHAR(255),

FOREIGN KEY(RestaurantName) REFERENCES Restaurant(RestaurantName)
    ON DELETE CASCADE,
FOREIGN KEY(MatchId) REFERENCES Matches(MatchId)
```

```
    ON DELETE CASCADE);

CREATE TABLE Reviews(OrderId INT PRIMARY KEY,
RestaurantName VARCHAR(255),
OrderCost REAL,
Rating REAL,
FoodPrepTime INT,
DietaryRestrictions VARCHAR(255),

FOREIGN KEY(RestaurantName) REFERENCES Restaurant(RestaurantName)
    ON DELETE CASCADE
);
```

Screenshot of working connection:



Three Tables with 1000+ Rows (User, Reviews, Restaurant):

```
mysql> select count(*) from User;
+----------+
| count(*) |
+----------+
|     2000 |
+----------+
1 row in set (0.01 sec)
```

```
mysql> select count(*) from Reviews;
+----------+
| count(*) |
+----------+
|     1895 |
+----------+
1 row in set (0.00 sec)
```

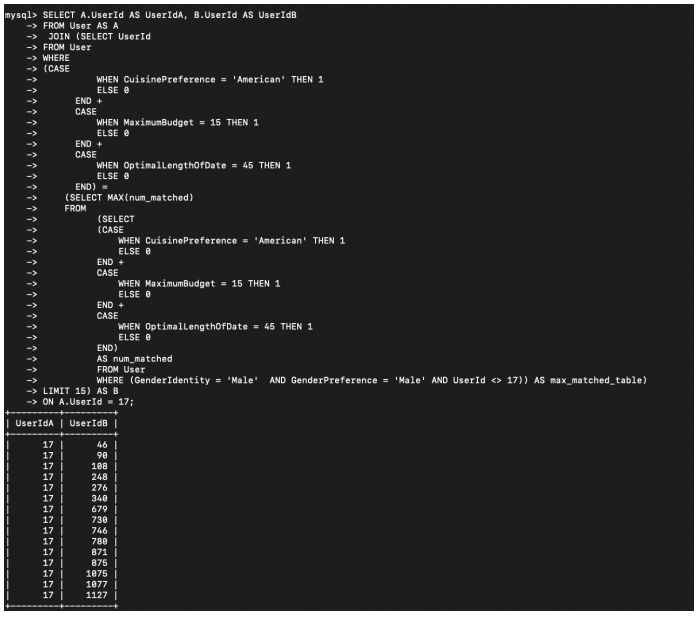
```
mysql> select count(*) from Restaurant;
+----------+
| count(*) |
+----------+
|     1678 |
+----------+
1 row in set (0.00 sec)
```

Query 1: For each person, find the top 15 people that matches in terms of CuisinePreference, MaximumBudget, OptimalLengthOfDate, Gender Identity, and Gender Preference

```
SELECT A.UserId AS UserIdA, B.UserId AS UserIdB
FROM User AS A
 JOIN (SELECT UserId
FROM User
WHERE
         (CASE
        WHEN CuisinePreference = 'American' THEN 1
        ELSE 0
    END +
    CASE
        WHEN MaximumBudget = 15 THEN 1
        ELSE 0
    END +
    CASE
```

```
        WHEN OptimalLengthOfDate = 45 THEN 1
        ELSE 0
      END) =
    (SELECT MAX(num_matched)
    FROM
        (SELECT
        (CASE
            WHEN CuisinePreference = 'American' THEN 1
            ELSE 0
        END +
        CASE
            WHEN MaximumBudget = 15 THEN 1
            ELSE 0
        END +
        CASE
            WHEN OptimalLengthOfDate = 45 THEN 1
            ELSE 0
        END)
        AS num_matched
        FROM User
        WHERE (GenderIdentity = 'Male'  AND GenderPreference = 'Male' AND UserId <> 17))
AS max_matched_table)
LIMIT 15) AS B
ON A.UserId = 17;
```

First 15 Rows:



```
mysql> SELECT A.UserId AS UserIdA, B.UserId AS UserIdB
    -> FROM User AS A
    ->  JOIN (SELECT UserId
    -> FROM User
    -> WHERE
    -> (CASE
    ->          WHEN CuisinePreference = 'American' THEN 1
    ->          ELSE 0
    ->      END +
    ->      CASE
    ->          WHEN MaximumBudget = 15 THEN 1
    ->          ELSE 0
    ->      END +
    ->      CASE
    ->          WHEN OptimalLengthOfDate = 45 THEN 1
    ->          ELSE 0
    ->      END) =
    ->    (SELECT MAX(num_matched)
    ->    FROM
    ->          (SELECT
    ->          (CASE
    ->              WHEN CuisinePreference = 'American' THEN 1
    ->              ELSE 0
    ->          END +
    ->          CASE
    ->              WHEN MaximumBudget = 15 THEN 1
    ->              ELSE 0
    ->          END +
    ->          CASE
    ->              WHEN OptimalLengthOfDate = 45 THEN 1
    ->              ELSE 0
    ->          END)
    ->          AS num_matched
    ->          FROM User
    ->          WHERE (GenderIdentity = 'Male'  AND GenderPreference = 'Male' AND UserId <> 17)) AS max_matched_table)
    -> LIMIT 15) AS B
    -> ON A.UserId = 17;
+---------+---------+
| UserIdA | UserIdB |
+---------+---------+
|      17 |      46 |
|      17 |      90 |
|      17 |     108 |
|      17 |     248 |
|      17 |     276 |
|      17 |     340 |
|      17 |     679 |
|      17 |     730 |
|      17 |     746 |
|      17 |     780 |
|      17 |     871 |
|      17 |     875 |
|      17 |    1075 |
|      17 |    1077 |
|      17 |    1127 |
+---------+---------+
```

Explain Analyze:

```
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
---+
| -> Table scan on B  (cost=206..208 rows=15) (actual time=3.41..3.41 rows=15 loops=1)
    -> Materialize  (cost=206..206 rows=15) (actual time=3.41..3.41 rows=15 loops=1)
        -> Limit: 15 row(s)  (cost=204 rows=15) (actual time=2.03..3.37 rows=15 loops=1)
            -> Filter: ((((case when (`user`.CuisinePreference = 'American') then 1 else 0 end) + (case when (`user`.
MaximumBudget = 15) then 1 else 0 end)) + (case when (`user`.OptimalLengthOfDate = 45) then 1 else 0 end)) = (select
#3))  (cost=204 rows=2000) (actual time=2..3.34 rows=15 loops=1)
                -> Table scan on User  (cost=204 rows=2000) (actual time=0.0295..0.522 rows=1128 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Aggregate: max((((case when (`user`.CuisinePreference = 'American') then 1 else 0 end) + (case
 when (`user`.MaximumBudget = 15) then 1 else 0 end)) + (case when (`user`.OptimalLengthOfDate = 45) then 1 else 0 en
d)))  (cost=205 rows=1) (actual time=1.81..1.81 rows=1 loops=1)
                        -> Filter: ((`user`.GenderPreference = 'Male') and (`user`.GenderIdentity = 'Male') and (`use
r`.UserId <> 17))  (cost=204 rows=10.2) (actual time=0.0178..1.47 rows=283 loops=1)
                            -> Index range scan on User using PRIMARY over (UserId < 17) OR (17 < UserId)  (cost=204
rows=1017) (actual time=0.0117..1.16 rows=1999 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Index 1 (Cuisine Preference):

```
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
----+
| -> Table scan on B  (cost=206..208 rows=15) (actual time=6.34..6.35 rows=15 loops=1)
    -> Materialize  (cost=206..206 rows=15) (actual time=6.34..6.34 rows=15 loops=1)
        -> Limit: 15 row(s)  (cost=204 rows=15) (actual time=3.74..6.31 rows=15 loops=1)
            -> Filter: ((((case when (`user`.CuisinePreference = 'American') then 1 else 0 end) + (case when (`user`.
MaximumBudget = 15) then 1 else 0 end)) + (case when (`user`.OptimalLengthOfDate = 45) then 1 else 0 end)) = (select
#3))  (cost=204 rows=2000) (actual time=3.74..6.3 rows=15 loops=1)
                -> Table scan on User  (cost=204 rows=2000) (actual time=0.0284..1.02 rows=1128 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Aggregate: max((((case when (`user`.CuisinePreference = 'American') then 1 else 0 end) + (case
 when (`user`.MaximumBudget = 15) then 1 else 0 end)) + (case when (`user`.OptimalLengthOfDate = 45) then 1 else 0 en
d)))  (cost=205 rows=1) (actual time=3.53..3.53 rows=1 loops=1)
                        -> Filter: ((`user`.GenderPreference = 'Male') and (`user`.GenderIdentity = 'Male') and (`use
r`.UserId <> 17))  (cost=204 rows=10.2) (actual time=0.0215..3.04 rows=283 loops=1)
                            -> Index range scan on User using PRIMARY over (UserId < 17) OR (17 < UserId)  (cost=204
rows=1017) (actual time=0.0128..2.39 rows=1999 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Index 2 (CuisinePreference, MaximumBudget, OptimalLengthofDate):

```
-> Table scan on B  (cost=206..208 rows=15) (actual time=6.19..6.19 rows=15 loops=1)
    -> Materialize  (cost=206..206 rows=15) (actual time=6.18..6.18 rows=15 loops=1)
        -> Limit: 15 row(s)  (cost=204 rows=15) (actual time=3.55..6.15 rows=15 loops=1)
            -> Filter: ((((case when (`user`.CuisinePreference = 'American') then 1 else 0 end) + (case when (`user`.
MaximumBudget = 15) then 1 else 0 end)) + (case when (`user`.OptimalLengthOfDate = 45) then 1 else 0 end)) = (select
#3))  (cost=204 rows=2000) (actual time=3.54..6.14 rows=15 loops=1)
                -> Table scan on User  (cost=204 rows=2000) (actual time=0.0265..0.991 rows=1128 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Aggregate: max((((case when (`user`.CuisinePreference = 'American') then 1 else 0 end) + (case
 when (`user`.MaximumBudget = 15) then 1 else 0 end)) + (case when (`user`.OptimalLengthOfDate = 45) then 1 else 0 en
d)))  (cost=205 rows=1) (actual time=3.36..3.36 rows=1 loops=1)
                        -> Filter: ((`user`.GenderPreference = 'Male') and (`user`.GenderIdentity = 'Male') and (`use
r`.UserId <> 17))  (cost=204 rows=10.2) (actual time=0.0228..2.82 rows=283 loops=1)
                            -> Index range scan on User using PRIMARY over (UserId < 17) OR (17 < UserId)  (cost=204
rows=1017) (actual time=0.0143..2.21 rows=1999 loops=1)
 |
```

Index 3 (Cuisine Preference, MaximumBudget, OptimalLengthofDate, GenderIdentity, GenderPreference):

```
-> Table scan on B  (cost=206..208 rows=15) (actual time=5.99..5.99 rows=15 loops=1)
    -> Materialize  (cost=206..206 rows=15) (actual time=5.99..5.99 rows=15 loops=1)
        -> Limit: 15 row(s)  (cost=204 rows=15) (actual time=3.34..5.95 rows=15 loops=1)
            -> Filter: ((((case when (`user`.CuisinePreference = 'American') then 1 else 0 end) + (case when (`user`.
MaximumBudget = 15) then 1 else 0 end)) + (case when (`user`.OptimalLengthOfDate = 45) then 1 else 0 end)) = (select
#3))  (cost=204 rows=2000) (actual time=3.34..5.95 rows=15 loops=1)
                -> Table scan on User  (cost=204 rows=2000) (actual time=0.0275..1.14 rows=1128 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Aggregate: max((((case when (`user`.CuisinePreference = 'American') then 1 else 0 end) + (case
 when (`user`.MaximumBudget = 15) then 1 else 0 end)) + (case when (`user`.OptimalLengthOfDate = 45) then 1 else 0 en
d)))  (cost=221 rows=1) (actual time=3.17..3.17 rows=1 loops=1)
                        -> Filter: ((`user`.GenderPreference = 'Male') and (`user`.GenderIdentity = 'Male') and (`use
r`.UserId <> 17))  (cost=204 rows=170) (actual time=0.0216..2.74 rows=283 loops=1)
                            -> Index range scan on User using PRIMARY over (UserId < 17) OR (17 < UserId)  (cost=204
rows=1017) (actual time=0.0123..2.14 rows=1999 loops=1)
 |
```

We indexed several attributes present in the JOIN and WHERE clauses and found that indexing did not affect the cost of running the query. This is likely because indexing is not helpful on a small number of items, and the EXPLAIN ANALYZE command shows that the query is only running a single loop. For example, when we indexed GenderIdentity and GenderPreference, the query only retrieved the values stored in these attributes for 283 rows, which only took 0.0216 milliseconds. Because the query is so quick, indexing doesn't affect the cost/time. We ultimately decided to go with our third indexing scheme (index Cuisine Preference, MaximumBudget, OptimalLengthofDate, GenderIdentity, GenderPreference), because it took the least amount of time, even though the cost remained the same.

Query 2: Use Reviews and Restaurant info to match a match with a restaurant:

```
SELECT RestaurantName, Score
FROM (
   SELECT Res.RestaurantName,
      CASE
         WHEN UA.MaximumBudget >= (SELECT AVG(OrderCost) FROM Reviews Rev
WHERE Rev.RestaurantName = Res.RestaurantName) AND UB.MaximumBudget >=
(SELECT AVG(OrderCost) FROM Reviews Rev WHERE Rev.RestaurantName =
Res.RestaurantName) THEN 1
         ELSE 0
      END +
      CASE
         WHEN EXISTS (SELECT RestaurantName FROM Reviews Rev WHERE
Rev.RestaurantName = Res.RestaurantName AND (UA.Allergies = Rev.DietaryRestrictions OR
UB.Allergies = Rev.DietaryRestrictions) AND Res.AverageRating > 3) THEN 1
         ELSE 0
      END +
      CASE
         WHEN UA.CuisinePreference = Res.Cuisine AND UB.CuisinePreference = Res.Cuisine
THEN 1
         ELSE 0
      END AS Score
   FROM Restaurant Res
   CROSS JOIN Matches M
   JOIN User UA ON M.UserIdA = UA.UserId
   JOIN User UB ON M.UserIdB = UB.UserId
) AS ScoredRestaurants
ORDER BY Score DESC
LIMIT 15;
```

```
mysql> SELECT RestaurantName, Score FROM (     SELECT Res.RestaurantName,         CASE             WHEN UA.MaximumBudget >= (SELECT AVG(OrderCost) FROM Reviews Rev
taurantName = Res.RestaurantName) AND UB.MaximumBudget >= (SELECT AVG(OrderCost) FROM Reviews Rev WHERE Rev.RestaurantName = Res.RestaurantName) THEN 1
  END +         CASE             WHEN EXISTS (SELECT RestaurantName FROM Reviews Rev WHERE Rev.RestaurantName = Res.RestaurantName AND (UA.Allergies = Rev.DietaryR
 UB.Allergies = Rev.DietaryRestrictions) AND Res.AverageRating > 3) THEN 1             ELSE 0         END +         CASE             WHEN UA.CuisinePreference = Re
UB.CuisinePreference = Res.Cuisine THEN 1             ELSE 0         END AS Score     FROM Restaurant Res     CROSS JOIN Matches M     JOIN User UA ON M.UserIdA =
JOIN User UB ON M.UserIdB = UB.UserId ) AS ScoredRestaurants ORDER BY Score DESC LIMIT 15;
+----------------------------+-------+
| RestaurantName             | Score |
+----------------------------+-------+
| Bright Café Lebanese       |     1 |
| Charming Bistro Italian    |     1 |
| Bright Harbor Japanese     |     1 |
| Bright Grill Chinese       |     1 |
| Charming Garden Lebanese   |     1 |
| Charming Café Thai         |     1 |
| Ancient Terrace Vietnamese |     1 |
| Bright Lane American       |     1 |
| Bright Harbor Argentinian  |     1 |
| Ancient Market Brazilian   |     1 |
| Charming Bistro Chinese    |     1 |
| Charming Diner Brazilian   |     1 |
| Bright Point Lebanese      |     1 |
| Ancient House Russian      |     1 |
| Ancient Diner Mexican      |     1 |
+----------------------------+-------+
15 rows in set, 12716 warnings (0.09 sec)
```

Original cost: 2133

```
----------------------+
| -> Limit: 15 row(s)  (cost=2133..2133 rows=15) (actual time=94.3..94.3 rows=15 loops=1)
    -> Sort: scoredrestaurants.Score DESC, limit input to 15 row(s) per chunk  (cost=2133..2133 rows=15) (actual time=94.3..94.3 rows=15 loops=1)
        -> Table scan on ScoredRestaurants  (cost=1363..1450 rows=6768) (actual time=92.3..93.5 rows=6712 loops=1)
            -> Materialize  (cost=1363..1363 rows=6768) (actual time=92.3..92.3 rows=6712 loops=1)
                -> Inner hash join (no condition)  (cost=686 rows=6768) (actual time=0.0815..2.7 rows=6712 loops=1)
                    -> Table scan on Res  (cost=43.8 rows=1692) (actual time=0.0192..1.04 rows=1678 loops=1)
                    -> Hash
                        -> Nested loop inner join  (cost=3.45 rows=4) (actual time=0.0386..0.0512 rows=4 loops=1)
                            -> Nested loop inner join  (cost=2.05 rows=4) (actual time=0.0349..0.0442 rows=4 loops=1)
                                -> Filter: ((m.UserIdA is not null) and (m.UserIdB is not null))  (cost=0.65 rows=4) (actual time=0.0217..0.0249 rows=4 loops=1)
                                    -> Table scan on M  (cost=0.65 rows=4) (actual time=0.0203..0.023 rows=4 loops=1)
                                -> Single-row index lookup on UA using PRIMARY (UserId=m.UserIdA)  (cost=0.275 rows=1) (actual time=0.00437..0.00441 rows=1 loops=4)
                            -> Single-row index lookup on UB using PRIMARY (UserId=m.UserIdB)  (cost=0.275 rows=1) (actual time=0.00137..0.00141 rows=1 loops=4)
                -> Select #3 (subquery in projection; dependent)
                    -> Aggregate: avg(rev.OrderCost)  (cost=4.82 rows=1) (actual time=0.00383..0.00387 rows=1 loops=6712)
                        -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=3.75 rows=10.7) (actual time=0.0027..0.00345 rows=1.13 loops=671
2)
                -> Select #4 (subquery in projection; dependent)
                    -> Aggregate: avg(rev.OrderCost)  (cost=4.82 rows=1) (actual time=0.00254..0.00258 rows=1 loops=6004)
                        -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=3.75 rows=10.7) (actual time=0.00225..0.00225 rows=0 loops=6004)
                -> Select #5 (subquery in projection; dependent)
                    -> Limit: 1 row(s)  (cost=2.88 rows=1) (actual time=0.00388..0.00388 rows=0 loops=6712)
                        -> Filter: (((ua.Allergies = rev.DietaryRestrictions) or (ub.Allergies = rev.DietaryRestrictions)) and (res.AverageRating > 3))  (cost=2.88 rows=2.03) (a
ctual time=0.00371..0.00371 rows=0 loops=6712)
                            -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=2.88 rows=10.7) (actual time=0.00262..0.00334 rows=1.13 loop
s=6712)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

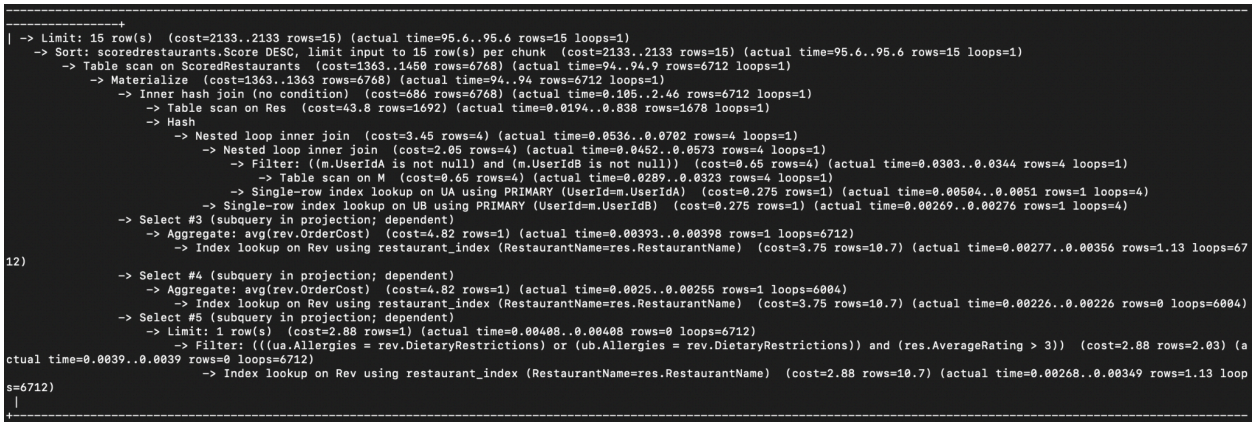
Added index on Reservation(RestaurantName)

Cost: 2133

```
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------+
| -> Limit: 15 row(s)  (cost=2133..2133 rows=15) (actual time=99.4..99.4 rows=15 loops=1)
    -> Sort: scoredrestaurants.Score DESC, limit input to 15 row(s) per chunk  (cost=2133..2133 rows=15) (actual time=99.4..99.4 rows=15 loops=1)
        -> Table scan on ScoredRestaurants  (cost=1363..1450 rows=6768) (actual time=97.8..98.7 rows=6712 loops=1)
            -> Materialize  (cost=1363..1363 rows=6768) (actual time=97.7..97.7 rows=6712 loops=1)
                -> Inner hash join (no condition)  (cost=686 rows=6768) (actual time=0.206..2.74 rows=6712 loops=1)
                    -> Table scan on Res  (cost=43.8 rows=1692) (actual time=0.0378..1.04 rows=1678 loops=1)
                    -> Hash
                        -> Nested loop inner join  (cost=3.45 rows=4) (actual time=0.0917..0.123 rows=4 loops=1)
                            -> Nested loop inner join  (cost=2.05 rows=4) (actual time=0.084..0.107 rows=4 loops=1)
                                -> Filter: ((m.UserIdA is not null) and (m.UserIdB is not null))  (cost=0.65 rows=4) (actual time=0.0574..0.0643 rows=4 loops=1)
                                    -> Table scan on M  (cost=0.65 rows=4) (actual time=0.0541..0.0598 rows=4 loops=1)
                                -> Single-row index lookup on UA using PRIMARY (UserId=m.UserIdA)  (cost=0.275 rows=1) (actual time=0.00937..0.00948 rows=1 loops=4)
                            -> Single-row index lookup on UB using PRIMARY (UserId=m.UserIdB)  (cost=0.275 rows=1) (actual time=0.00311..0.00323 rows=1 loops=4)
                -> Select #3 (subquery in projection; dependent)
                    -> Aggregate: avg(rev.OrderCost)  (cost=4.82 rows=1) (actual time=0.00404..0.00409 rows=1 loops=6712)
                        -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=3.75 rows=10.7) (actual time=0.00284..0.00365 rows=1.13 loops=67
12)
                -> Select #4 (subquery in projection; dependent)
                    -> Aggregate: avg(rev.OrderCost)  (cost=4.82 rows=1) (actual time=0.00267..0.00271 rows=1 loops=6004)
                        -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=3.75 rows=10.7) (actual time=0.00239..0.00239 rows=0 loops=6004)
                -> Select #5 (subquery in projection; dependent)
                    -> Limit: 1 row(s)  (cost=2.88 rows=1) (actual time=0.00422..0.00422 rows=0 loops=6712)
                        -> Filter: (((ua.Allergies = rev.DietaryRestrictions) or (ub.Allergies = rev.DietaryRestrictions)) and (res.AverageRating > 3))  (cost=2.88 rows=2.03) (a
ctual time=0.00402..0.00402 rows=0 loops=6712)
                            -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=2.88 rows=10.7) (actual time=0.00282..0.00362 rows=1.13 loop
s=6712)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

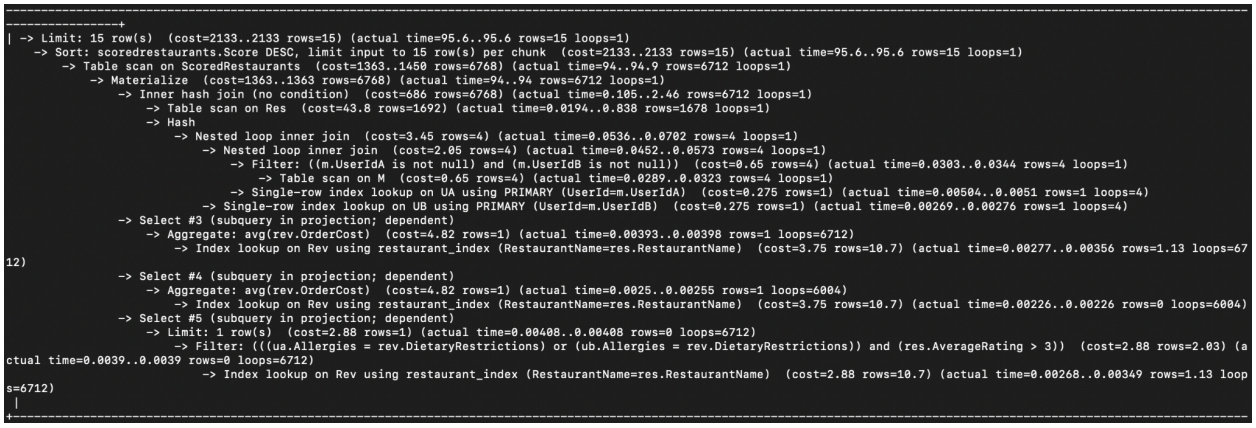
Added index on Matches(UserIdA)

Cost: 2133



```
-> Limit: 15 row(s)  (cost=2133..2133 rows=15) (actual time=95.6..95.6 rows=15 loops=1)
    -> Sort: scoredrestaurants.Score DESC, limit input to 15 row(s) per chunk  (cost=2133..2133 rows=15) (actual time=95.6..95.6 rows=15 loops=1)
        -> Table scan on ScoredRestaurants  (cost=1363..1450 rows=6768) (actual time=94..94.9 rows=6712 loops=1)
            -> Materialize  (cost=1363..1363 rows=6768) (actual time=94..94 rows=6712 loops=1)
                -> Inner hash join (no condition)  (cost=686 rows=6768) (actual time=0.105..2.46 rows=6712 loops=1)
                    -> Table scan on Res  (cost=43.8 rows=1692) (actual time=0.0194..0.838 rows=1678 loops=1)
                    -> Hash
                        -> Nested loop inner join  (cost=3.45 rows=4) (actual time=0.0536..0.0702 rows=4 loops=1)
                            -> Nested loop inner join  (cost=2.05 rows=4) (actual time=0.0452..0.0573 rows=4 loops=1)
                                -> Filter: ((m.UserIdA is not null) and (m.UserIdB is not null))  (cost=0.65 rows=4) (actual time=0.0303..0.0344 rows=4 loops=1)
                                    -> Table scan on M  (cost=0.65 rows=4) (actual time=0.0289..0.0323 rows=4 loops=1)
                                -> Single-row index lookup on UA using PRIMARY (UserId=m.UserIdA)  (cost=0.275 rows=1) (actual time=0.00504..0.0051 rows=1 loops=4)
                            -> Single-row index lookup on UB using PRIMARY (UserId=m.UserIdB)  (cost=0.275 rows=1) (actual time=0.00269..0.00276 rows=1 loops=4)
                -> Select #3 (subquery in projection; dependent)
                    -> Aggregate: avg(rev.OrderCost)  (cost=4.82 rows=1) (actual time=0.00393..0.00398 rows=1 loops=6712)
                        -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=3.75 rows=10.7) (actual time=0.00277..0.00356 rows=1.13 loops=67
12)
                -> Select #4 (subquery in projection; dependent)
                    -> Aggregate: avg(rev.OrderCost)  (cost=4.82 rows=1) (actual time=0.0025..0.00255 rows=1 loops=6004)
                        -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=3.75 rows=10.7) (actual time=0.00226..0.00226 rows=0 loops=6004)
                -> Select #5 (subquery in projection; dependent)
                    -> Limit: 1 row(s)  (cost=2.88 rows=1) (actual time=0.00408..0.00408 rows=0 loops=6712)
                        -> Filter: (((ua.Allergies = rev.DietaryRestrictions) or (ub.Allergies = rev.DietaryRestrictions)) and (res.AverageRating > 3))  (cost=2.88 rows=2.03) (a
ctual time=0.0039..0.0039 rows=0 loops=6712)
                            -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=2.88 rows=10.7) (actual time=0.00268..0.00349 rows=1.13 loop
s=6712)
 |
```

Added index on Matches(UserIdB)
Cost: 2133



```
-> Limit: 15 row(s)  (cost=2133..2133 rows=15) (actual time=95.6..95.6 rows=15 loops=1)
    -> Sort: scoredrestaurants.Score DESC, limit input to 15 row(s) per chunk  (cost=2133..2133 rows=15) (actual time=95.6..95.6 rows=15 loops=1)
        -> Table scan on ScoredRestaurants  (cost=1363..1450 rows=6768) (actual time=94..94.9 rows=6712 loops=1)
            -> Materialize  (cost=1363..1363 rows=6768) (actual time=94..94 rows=6712 loops=1)
                -> Inner hash join (no condition)  (cost=686 rows=6768) (actual time=0.105..2.46 rows=6712 loops=1)
                    -> Table scan on Res  (cost=43.8 rows=1692) (actual time=0.0194..0.838 rows=1678 loops=1)
                    -> Hash
                        -> Nested loop inner join  (cost=3.45 rows=4) (actual time=0.0536..0.0702 rows=4 loops=1)
                            -> Nested loop inner join  (cost=2.05 rows=4) (actual time=0.0452..0.0573 rows=4 loops=1)
                                -> Filter: ((m.UserIdA is not null) and (m.UserIdB is not null))  (cost=0.65 rows=4) (actual time=0.0303..0.0344 rows=4 loops=1)
                                    -> Table scan on M  (cost=0.65 rows=4) (actual time=0.0289..0.0323 rows=4 loops=1)
                                -> Single-row index lookup on UA using PRIMARY (UserId=m.UserIdA)  (cost=0.275 rows=1) (actual time=0.00504..0.0051 rows=1 loops=4)
                            -> Single-row index lookup on UB using PRIMARY (UserId=m.UserIdB)  (cost=0.275 rows=1) (actual time=0.00269..0.00276 rows=1 loops=4)
                -> Select #3 (subquery in projection; dependent)
                    -> Aggregate: avg(rev.OrderCost)  (cost=4.82 rows=1) (actual time=0.00393..0.00398 rows=1 loops=6712)
                        -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=3.75 rows=10.7) (actual time=0.00277..0.00356 rows=1.13 loops=67
12)
                -> Select #4 (subquery in projection; dependent)
                    -> Aggregate: avg(rev.OrderCost)  (cost=4.82 rows=1) (actual time=0.0025..0.00255 rows=1 loops=6004)
                        -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=3.75 rows=10.7) (actual time=0.00226..0.00226 rows=0 loops=6004)
                -> Select #5 (subquery in projection; dependent)
                    -> Limit: 1 row(s)  (cost=2.88 rows=1) (actual time=0.00408..0.00408 rows=0 loops=6712)
                        -> Filter: (((ua.Allergies = rev.DietaryRestrictions) or (ub.Allergies = rev.DietaryRestrictions)) and (res.AverageRating > 3))  (cost=2.88 rows=2.03) (a
ctual time=0.0039..0.0039 rows=0 loops=6712)
                            -> Index lookup on Rev using restaurant_index (RestaurantName=res.RestaurantName)  (cost=2.88 rows=10.7) (actual time=0.00268..0.00349 rows=1.13 loop
s=6712)
 |
```

In order to find top restaurant matches for each user match, we indexed several different attributes in the JOIN and WHERE clauses and found no difference in cost. For example, the original cost of the query without indexing was 2133. This number remained the same after adding indexes on Reservation(RestaurantName), Matches(UserIdA), and Matches(UserIdB), which should have sped up the subqueries and join operations. However, because this query involves multiple subqueries and conditional logic, it is relatively more computationally intensive. In this case, indexes don't have a significant impact on the overall cost since the main cost lies in these complex computations rather than table access.

Query 3: Calculate Average Restaurant Rating (displayed in the UI)

```
SELECT RestaurantName, AVG(rating)
FROM
  (SELECT RestaurantName, Rating as rating
```

```
    FROM Restaurant NATURAL JOIN Reviews) AS joined_table
GROUP BY RestaurantName
HAVING RestaurantName = 'Hangawi';
```

*Hangawi will be replaced with the restaurant that is displayed to the user

Original Query:

```
mysql>
mysql> SELECT RestaurantName, AVG(rating)
    -> FROM
    ->   (SELECT RestaurantName, Rating as rating
    ->       FROM Restaurant NATURAL JOIN Reviews) AS joined_table
    -> GROUP BY RestaurantName
    -> HAVING RestaurantName = 'Hangawi';
+----------------+-------------+
| RestaurantName | AVG(rating) |
+----------------+-------------+
| Hangawi        |           4 |
+----------------+-------------+
1 row in set (0.00 sec)
```

Original cost - 856

```
-> Filter: (restaurant.RestaurantName = 'Hangawi')  (actual time=9.62..9.64 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=9.51..9.56 rows=177 loops=1)
        -> Aggregate using temporary table  (actual time=9.51..9.51 rows=177 loops=1)
            -> Nested loop inner join  (cost=856 rows=1895) (actual time=0.0844..7.25 rows=1895 loops=1)
                -> Filter: (reviews.RestaurantName is not null)  (cost=193 rows=1895) (actual time=0.0629..1.5 rows=1895 loops=1)
                    -> Table scan on Reviews  (cost=193 rows=1895) (actual time=0.0612..1.17 rows=1895 loops=1)
                -> Single-row covering index lookup on Restaurant using PRIMARY (RestaurantName=reviews.RestaurantName)  (cost=0.25 rows=1) (actual time=0.00259..0.00267 rows=1
loops=1895)
 |
```

Cost of index on Restaurant(RestaurantName) - 856

```
-> Filter: (restaurant.RestaurantName = 'Hangawi')  (actual time=12.4..12.4 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=12.3..12.4 rows=177 loops=1)
        -> Aggregate using temporary table  (actual time=12.3..12.3 rows=177 loops=1)
            -> Nested loop inner join  (cost=856 rows=1895) (actual time=0.0971..9.25 rows=1895 loops=1)
                -> Filter: (reviews.RestaurantName is not null)  (cost=193 rows=1895) (actual time=0.0752..1.88 rows=1895 loops=1)
                    -> Table scan on Reviews  (cost=193 rows=1895) (actual time=0.0734..1.5 rows=1895 loops=1)
                -> Single-row covering index lookup on Restaurant using PRIMARY (RestaurantName=reviews.RestaurantName)  (cost=0.25 rows=1) (actual time=0.00333..0.00342 rows=1
loops=1895)
 |
```

Cost of index on Reviews(Rating) - 856

```
-> Filter: (restaurant.RestaurantName = 'Hangawi')  (actual time=10.8..10.8 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=10.7..10.8 rows=177 loops=1)
        -> Aggregate using temporary table  (actual time=10.7..10.7 rows=177 loops=1)
            -> Nested loop inner join  (cost=856 rows=1895) (actual time=0.0704..8.23 rows=1895 loops=1)
                -> Filter: (reviews.RestaurantName is not null)  (cost=193 rows=1895) (actual time=0.0532..1.66 rows=1895 loops=1)
                    -> Table scan on Reviews  (cost=193 rows=1895) (actual time=0.052..1.31 rows=1895 loops=1)
                -> Single-row covering index lookup on Restaurant using PRIMARY (RestaurantName=reviews.RestaurantName)  (cost=0.25 rows=1) (actual time=0.00296..0.00304 rows=1
loops=1895)
 |
```

Cost of index on both Reviews(Rating) and Restaurant(RestaurantName) - 856

```
-> Filter: (restaurant.RestaurantName = 'Hangawi')  (actual time=13..13 rows=1 loops=1)
    -> Table scan on <temporary>  (actual time=13..13 rows=177 loops=1)
        -> Aggregate using temporary table  (actual time=13..13 rows=177 loops=1)
            -> Nested loop inner join  (cost=856 rows=1895) (actual time=0.12..9.75 rows=1895 loops=1)
                -> Filter: (reviews.RestaurantName is not null)  (cost=193 rows=1895) (actual time=0.0803..1.97 rows=1895 loops=1)
                    -> Table scan on Reviews  (cost=193 rows=1895) (actual time=0.0788..1.62 rows=1895 loops=1)
                -> Single-row covering index lookup on Restaurant using PRIMARY (RestaurantName=reviews.RestaurantName)  (cost=0.25 rows=1) (actual time=0.00346..0.00358 rows=1
loops=1895)
 |
```

The cost of the query did not change, as it was 856 for many combinations of indices of the query. The query mainly used RestaurantName (from Restaurants) and Rating (from Reviews), but adding indices for these attributes did not have an impact on the cost. This is probably due to the inner join that comprises the majority of the cost, which requires a nested loop to loop through every pair. In this case, adding an index would not help, as a nested join does not need to efficiently get a specific row because it needs to go through every single row at least once.

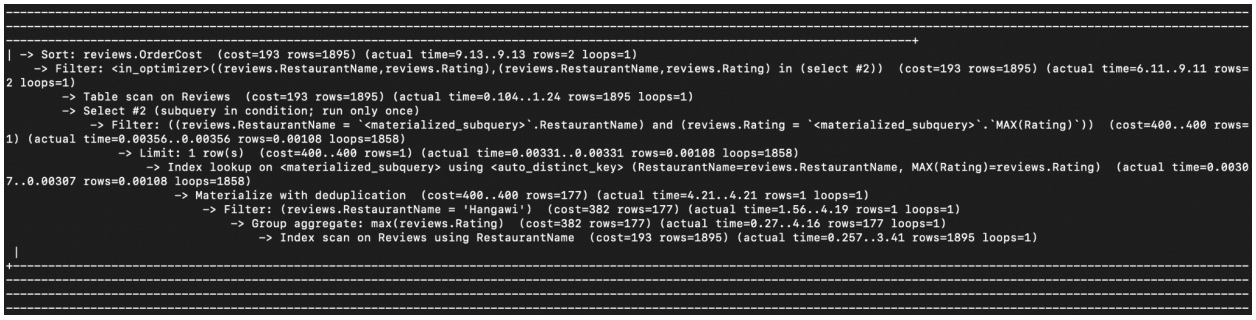
Query 4: Find Top Review (displayed in the UI)

```
SELECT *
FROM Reviews
WHERE
        (RestaurantName, Rating)
        IN
        (SELECT RestaurantName, MAX(Rating)
        FROM
                Reviews
        GROUP BY RestaurantName
        HAVING RestaurantName = 'Hangawi')
ORDER BY OrderCost;
```

First 15 Rows (Only 2 in the output):

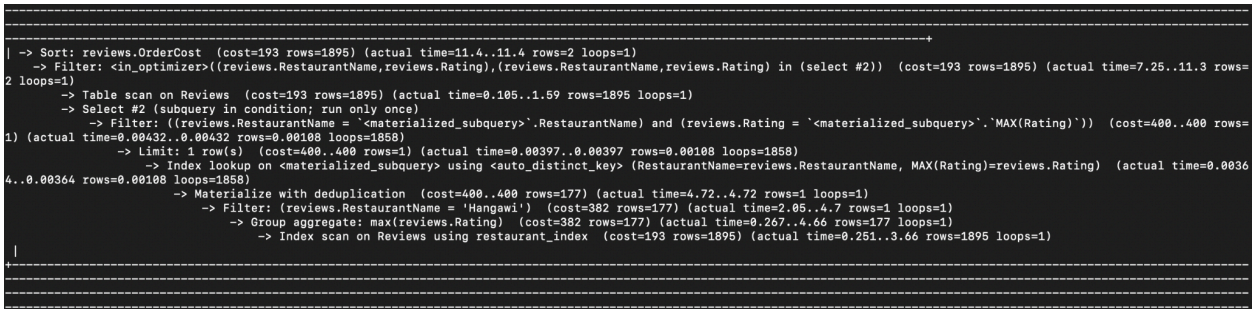
```
mysql> SELECT *
    -> FROM Reviews
    -> WHERE
    -> (RestaurantName, Rating)
    -> IN
    -> (SELECT RestaurantName, MAX(Rating)
    -> FROM
    -> Reviews
    -> GROUP BY RestaurantName
    -> HAVING RestaurantName = 'Hangawi')
    -> ORDER BY OrderCost;
+---------+----------------+-----------+--------+-------------+--------------------+
| OrderId | RestaurantName | OrderCost | Rating | FoodPrepTime | DietaryRestrictions |
+---------+----------------+-----------+--------+-------------+--------------------+
| 1477600 | Hangawi        |      6.74 |      4 |           21 | Milk               |
| 1477147 | Hangawi        |     30.75 |      4 |           25 | Gluten             |
+---------+----------------+-----------+--------+-------------+--------------------+
2 rows in set (0.01 sec)
```

Explain Analyze:
Initial Cost = 193, 400



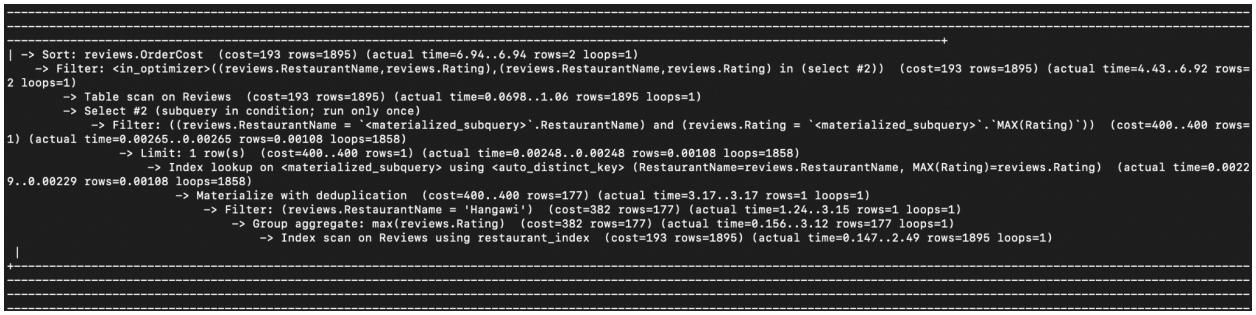
```
| -> Sort: reviews.OrderCost  (cost=193 rows=1895) (actual time=9.13..9.13 rows=2 loops=1)
    -> Filter: <in_optimizer>((reviews.RestaurantName,reviews.Rating),(reviews.RestaurantName,reviews.Rating) in (select #2))  (cost=193 rows=1895) (actual time=6.11..9.11 rows=
2 loops=1)
        -> Table scan on Reviews  (cost=193 rows=1895) (actual time=0.104..1.24 rows=1895 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Filter: ((reviews.RestaurantName = `<materialized_subquery>`.RestaurantName) and (reviews.Rating = `<materialized_subquery>`.`MAX(Rating)`))  (cost=400..400 rows=
1) (actual time=0.00356..0.00356 rows=0.00108 loops=1858)
                -> Limit: 1 row(s)  (cost=400..400 rows=1) (actual time=0.00331..0.00331 rows=0.00108 loops=1858)
                    -> Index lookup on <materialized_subquery> using <auto_distinct_key> (RestaurantName=reviews.RestaurantName, MAX(Rating)=reviews.Rating)  (actual time=0.0030
7..0.00307 rows=0.00108 loops=1858)
                        -> Materialize with deduplication  (cost=400..400 rows=177) (actual time=4.21..4.21 rows=1 loops=1)
                            -> Filter: (reviews.RestaurantName = 'Hangawi')  (cost=382 rows=177) (actual time=1.56..4.19 rows=1 loops=1)
                                -> Group aggregate: max(reviews.Rating)  (cost=382 rows=177) (actual time=0.27..4.16 rows=177 loops=1)
                                    -> Index scan on Reviews using RestaurantName  (cost=193 rows=1895) (actual time=0.257..3.41 rows=1895 loops=1)
 |
```

Index 1 (RestaurantName):
CREATE INDEX restaurant_index ON Reviews(RestaurantName);



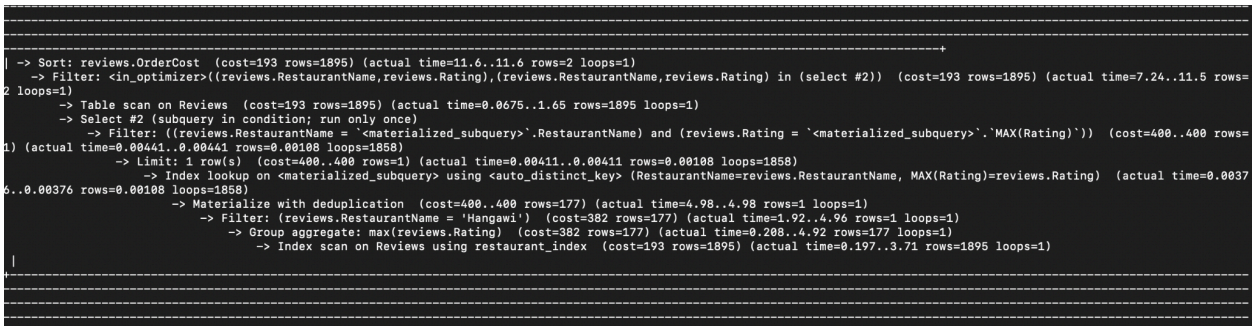
```
| -> Sort: reviews.OrderCost  (cost=193 rows=1895) (actual time=11.4..11.4 rows=2 loops=1)
    -> Filter: <in_optimizer>((reviews.RestaurantName,reviews.Rating),(reviews.RestaurantName,reviews.Rating) in (select #2))  (cost=193 rows=1895) (actual time=7.25..11.3 rows=
2 loops=1)
        -> Table scan on Reviews  (cost=193 rows=1895) (actual time=0.105..1.59 rows=1895 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Filter: ((reviews.RestaurantName = `<materialized_subquery>`.RestaurantName) and (reviews.Rating = `<materialized_subquery>`.`MAX(Rating)`))  (cost=400..400 rows=
1) (actual time=0.00432..0.00432 rows=0.00108 loops=1858)
                -> Limit: 1 row(s)  (cost=400..400 rows=1) (actual time=0.00397..0.00397 rows=0.00108 loops=1858)
                    -> Index lookup on <materialized_subquery> using <auto_distinct_key> (RestaurantName=reviews.RestaurantName, MAX(Rating)=reviews.Rating)  (actual time=0.0036
4..0.00364 rows=0.00108 loops=1858)
                        -> Materialize with deduplication  (cost=400..400 rows=177) (actual time=4.72..4.72 rows=1 loops=1)
                            -> Filter: (reviews.RestaurantName = 'Hangawi')  (cost=382 rows=177) (actual time=2.05..4.7 rows=1 loops=1)
                                -> Group aggregate: max(reviews.Rating)  (cost=382 rows=177) (actual time=0.267..4.66 rows=177 loops=1)
                                    -> Index scan on Reviews using restaurant_index  (cost=193 rows=1895) (actual time=0.251..3.66 rows=1895 loops=1)
 |
```

Index 2 (RestaurantName, Rating):
CREATE INDEX restaurant_index ON Reviews(RestaurantName);
CREATE INDEX rating_index ON Reviews(Rating);



```
| -> Sort: reviews.OrderCost  (cost=193 rows=1895) (actual time=6.94..6.94 rows=2 loops=1)
    -> Filter: <in_optimizer>((reviews.RestaurantName,reviews.Rating),(reviews.RestaurantName,reviews.Rating) in (select #2))  (cost=193 rows=1895) (actual time=4.43..6.92 rows=
2 loops=1)
        -> Table scan on Reviews  (cost=193 rows=1895) (actual time=0.0698..1.06 rows=1895 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Filter: ((reviews.RestaurantName = `<materialized_subquery>`.RestaurantName) and (reviews.Rating = `<materialized_subquery>`.`MAX(Rating)`))  (cost=400..400 rows=
1) (actual time=0.00265..0.00265 rows=0.00108 loops=1858)
                -> Limit: 1 row(s)  (cost=400..400 rows=1) (actual time=0.00248..0.00248 rows=0.00108 loops=1858)
                    -> Index lookup on <materialized_subquery> using <auto_distinct_key> (RestaurantName=reviews.RestaurantName, MAX(Rating)=reviews.Rating)  (actual time=0.0022
9..0.00229 rows=0.00108 loops=1858)
                        -> Materialize with deduplication  (cost=400..400 rows=177) (actual time=3.17..3.17 rows=1 loops=1)
                            -> Filter: (reviews.RestaurantName = 'Hangawi')  (cost=382 rows=177) (actual time=1.24..3.15 rows=1 loops=1)
                                -> Group aggregate: max(reviews.Rating)  (cost=382 rows=177) (actual time=0.156..3.12 rows=177 loops=1)
                                    -> Index scan on Reviews using restaurant_index  (cost=193 rows=1895) (actual time=0.147..2.49 rows=1895 loops=1)
 |
```

Index 3 (Indexing RestaurantName, Rating, and OrderCost):
CREATE INDEX restaurant_index ON Reviews(RestaurantName);
CREATE INDEX rating_index ON Reviews(Rating);
CREATE INDEX order_index ON Reviews(OrderCost);



This query is used to display the top review for a given restaurant based on the rating. The reviews are ordered by the order cost as a tiebreaker. In our app, this will be useful for users to see the best reviews for a restaurant they are interested in dining at. For example purposes, we used the restaurant named “Hangawi” in our query.

The initial cost, calculated using the EXPLAIN ANALYZE command on our query was 400 for the inner select and 193 for the outer, and was also the cost for all three of our indexing designs. The first two indexing designs were on the RestaurantName and Rating attributes, which are in the WHERE clause, and the final indexing design was also on OrderCost which was in the ORDER BY clause. The cost likely remained the same because when indexing on these attributes, only a few rows are being filtered, as we can see rows = 1 for the filterings. This is likely because we have set the desired restaurant name, which in this example is Hangawi. Thus, the indexing did not reduce the search space and was not significant to the cost. Therefore, we can choose any of our indexing designs, since we are evaluating based on cost.

Stage 2 Fix:

IsInA(UserId:INT [PK, FK to User.UserId], MatchId:INT [PK, FK to Matches.MatchId])

IsInA is a many-to-many relationship, therefore, we made a relational schema for it. UserId and MatchId for the primary key together, and they are foreign keys.