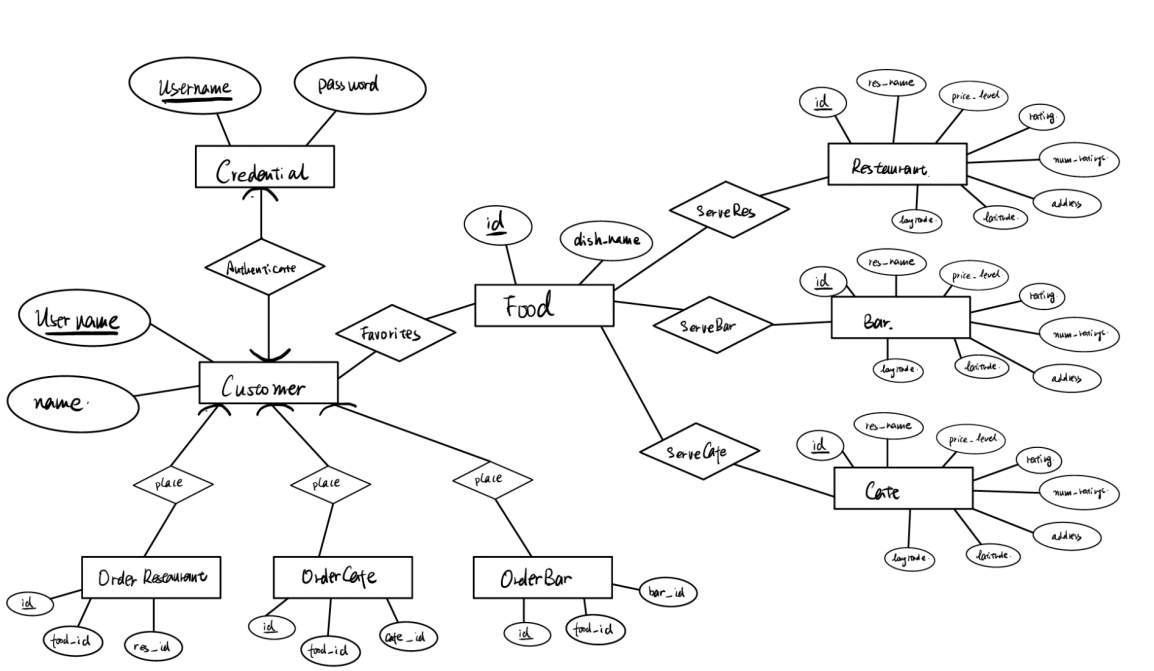
CS411
PT1 Stage3 Grp39

1. A few modifications to our original design

We slightly updated our original design of the schema based on what kind of actual data we can pull from the internet and what kind we need to generate artificially. To avoid the possible confusion that the later sections of creating tables etc. seem out of sync with our stage 2 design, we describe again our schema below:

Diagram:



Relations:
Customer(username: VARCHAR(35)[PK], name: VARCHAR(35));
Credential(username: VARCHAR(35)[PK, FK REF Customer.username], password);
# This is a one-to-one relationship between customer and credential. It could simply be an attribute to the customer, but a separate relation protects the customer's personal information.

Food(id: INT [PK], dish_name: VARCHAR(255));
Favorites(username [PK, FK REF Customer.username], food_id [PK, FK REF Food.id]);

Restaurant(id:INT [PK], res_name: VARCHAR(255), price_level: INT, rating: REAL, num_rating: INT, address: VARCHAR(255), longitude: REAL, latitude: REAL);
ServeRes(res_id:INT [PK, FK REF Restaurant.id], food_id:INT [PK, FK REF Food.id]);

Bar(id:INT [PK], bar_name: VARCHAR(255), price_level: INT, rating: REAL, num_rating: INT, address: VARCHAR(255), longitude: REAL, latitude: REAL);
ServeBar(bar_id:INT [PK, FK REF Bar.id], food_id:INT [PK, FK REF Food.id]);
Cafe(id:INT [PK], cafe_name: VARCHAR(255), price_level: INT, rating: REAL, num_rating: INT, address: VARCHAR(255), longitude: REAL, latitude: REAL);
ServeCafe(cafe_id:INT [PK, FK REF Cafe.id], food_id:INT [PK, FK REF Food.id]);

-> We would like to justify here why we intentionally split them into these 3 entities: We separated these 3 major types of places to eat because there are different functionalities associated with each, and in the future when we build recommendation systems, we will not need to filter by type everytime we query, so the performance will be much better.

ServeRes, ServeBar, ServeCafe are used to describe the menu in different places.

OrderRestaurant(id: INT[PK], username VARCHAR(35) [FK REF Customer.username], food_id: INT [FK REF Food.id], res_id INT [FK REF Restaurant.id]);
OrderBar(id: INT[PK], username VARCHAR(35) [FK REF Customer.username], food_id: INT [FK REF Food.id], bar_id INT [FK REF Bar.id]);
OrderCafe(id: INT[PK], username VARCHAR(35) [FK REF Customer.username], food_id: INT [FK REF Food.id], cafe_id INT [FK REF Cafe.id]);

Orders are one-to-many relationships that can be used to track what food each customer
orders from which place. Note that we only ask customers to give one dish or the best dish
from their entire order to get a better sense of their preference.

2. Data

Our application needs actual restaurant, bar, and cafe information from the Chicago area. We first found a food inspection database: https://data.cityofchicago.org/Health-Human-Services/Restaurant/5udb-dr6f. Pulled the name of all places and split them into 3 categories and then used the google maps places API: https://developers.google.com/maps/documentation/places/web-service/overview to scrape all the data we need for each place: name, address, longitude, and latitude for us to calculate distances, price-level, ratings and the number of ratings for recommendations.

We generated artificial data for the rest relations because we do not have a user base and the menu items were not easy to scrape. We will consider adding the actual menu info before the final demo, but artificial data should be adequate for the purpose of stage 3.

Specific details, data scrape and generation scripts can be found in the repo: ./data folder.

3. Table Implementation

We implemented all tables except for the Credential one, which is auxiliary information that is only used during login.

a. Customer(username: VARCHAR(35)[PK], name: VARCHAR(35));
# “name” attribute seems to overlap with an existent command, so replaced with “real_name”.

```
CREATE TABLE Customer (
username VARCHAR(35) PRIMARY KEY,
real_name VARCHAR(255)
);
```

b. Food(id: INT [PK], dish_name: VARCHAR(255));

```
CREATE TABLE Food (
id INT NOT NULL PRIMARY KEY,
dish_name VARCHAR(255)
);
```

"dish_name" could be a candidate key, but considering that dish name, in reality, could be really long, which is not ideal if we were to create index for this, we added an additional id.

c. Favorites(username [PK, FK REF Customer.username], food_id [PK, FK REF Food.id]);

```
CREATE TABLE Favorites (
username VARCHAR(35),
food_id INT,
PRIMARY KEY (username, food_id),
FOREIGN KEY (username) REFERENCES Customer(username),
FOREIGN KEY (food_id) REFERENCES Food(id)
);
```

This is a many-to-many relationship with 2 foreign keys.

d. Restaurant(id:INT [PK], res_name: VARCHAR(255), price_level: INT, rating: REAL, num_rating: INT, address: VARCHAR(255), longitude: REAL, latitude: REAL);

```
CREATE TABLE Restaurant(
id INT NOT NULL PRIMARY KEY,
res_name VARCHAR(255),
price_level INT CHECK ((price_level IN (1, 2, 3, 4)) OR (price_level IS NULL)),
rating REAL CHECK ((rating IS NULL) OR (rating <= 5) OR (rating >= 0)),
num_ratings INT CHECK ((num_ratings IS NULL) OR (num_ratings >= 0)),
address VARCHAR(500),
latitude REAL,
longitude REAL
);
```

price-level: corresponds to the number of the dollar signs on the google map. So it can only be 1,2,3,4 or NULL
rating: The stars given by the reviewers on the google map, so it has to be between 0 and 5.
num_ratings: number of reviews, a rating of 5-star given by 1 reviewer is apparently different from a 5-star given by a thousand reviewers.

*The other 2 types of places have the same structure:
Bar(id:INT [PK], bar_name: VARCHAR(255), price_level: INT, rating: REAL, num_rating: INT, address: VARCHAR(255), longitude: REAL, latitude: REAL);
Cafe(id:INT [PK], cafe_name: VARCHAR(255), price_level: INT, rating: REAL, num_rating: INT, address: VARCHAR(255), longitude: REAL, latitude: REAL);

```
CREATE TABLE Bar(
id INT NOT NULL PRIMARY KEY,
res_name VARCHAR(255),
price_level INT CHECK ((price_level IN (1, 2, 3, 4)) OR (price_level IS NULL)),
rating REAL CHECK ((rating <= 5) OR (rating >= 0) OR (rating IS NULL)),
num_ratings INT CHECK ((num_ratings >= 0) OR (num_ratings IS NULL)),
address VARCHAR(500),
latitude REAL,
longitude REAL
);
```

```
CREATE TABLE Cafe(
id INT NOT NULL PRIMARY KEY,
res_name VARCHAR(255),
price_level INT CHECK ((price_level IN (1, 2, 3, 4)) OR (price_level IS NULL)),
rating REAL CHECK ((rating IS NULL) OR (rating <= 5) OR (rating >= 0)),
num_ratings INT CHECK ((num_ratings IS NULL) OR (num_ratings >= 0)),
address VARCHAR(500),
latitude REAL,
longitude REAL
);
```

e. ServeRes(res_id:INT [PK, FK REF Restaurant.id], food_id:INT [PK, FK REF Food.id]);

```
CREATE TABLE ServeRestaurant(
food_id INT REFERENCES Food(id),
res_id INT REFERENCES Restaurant(id),
PRIMARY KEY(food_id, res_id)
);
```

A many-to-many relationship that describes what food items are served in a restaurant.
Although the main menu usually looks different for the 3 types of places, there usually are overlaps such as all places serve soft drinks.

Similar DDL for the bar and cafe:
ServeBar(bar_id:INT [PK, FK REF Bar.id], food_id:INT [PK, FK REF Food.id]);
ServeCafe(cafe_id:INT [PK, FK REF Cafe.id], food_id:INT [PK, FK REF Food.id]);

```
CREATE TABLE ServeBar(
food_id INT REFERENCES Food(id),
bar_id INT REFERENCES Bar(id),
PRIMARY KEY(food_id, bar_id)
);
```

```
CREATE TABLE ServeCafe(
food_id INT REFERENCES Food(id),
cafe_id INT REFERENCES Cafe(id),
PRIMARY KEY(food_id, cafe_id)
);
```

f. OrderRestaurant(id: INT[PK], username VARCHAR(35) [FK REF Customer.username], food_id: INT [FK REF Food.id], res_id INT [FK REF Restaurant.id]);

```
CREATE TABLE OrderRestaurant(
id INT NOT NULL PRIMARY KEY,
username VARCHAR(35) REFERENCES Customer(username),
food_id INT REFERENCES Food(id),
res_id INT REFERENCES Restaurant(id)
);
```

A one-to-many relationship since each order can only be placed by one user, but a user can place many orders.
each order contains a food item that the user chooses to include and where the food is from.

Similar DDL for the bar and cafe:
OrderBar(id: INT[PK], username VARCHAR(35) [FK REF Customer.username], food_id: INT [FK REF Food.id], bar_id INT [FK REF Bar.id]);
OrderCafe(id: INT[PK], username VARCHAR(35) [FK REF Customer.username], food_id: INT [FK REF Food.id], cafe_id INT [FK REF Cafe.id]);

```
CREATE TABLE OrderBar(
id INT NOT NULL PRIMARY KEY,
username VARCHAR(35) REFERENCES Customer(username),
food_id INT REFERENCES Food(id),
bar_id INT REFERENCES Bar(id)
);
```

```
CREATE TABLE OrderCafe(
id INT NOT NULL PRIMARY KEY,
username VARCHAR(35) REFERENCES Customer(username),
food_id INT REFERENCES Food(id),
cafe_id INT REFERENCES Cafe(id)
);
```

4. Number of entries in each table:

The tables with more than 1000 entries have been marked reds

Customer: 200

```
mysql> SELECT COUNT(*) FROM Customer;
+----------+
| COUNT(*) |
+----------+
|      200 |
+----------+
1 row in set (0.04 sec)
```

Food: 1147

```
mysql> SELECT COUNT(*) FROM Food;
+----------+
| COUNT(*) |
+----------+
|     1187 |
+----------+
1 row in set (0.04 sec)
```

Favorites:394

```
mysql> SELECT COUNT(*) FROM Favorites;
+----------+
| COUNT(*) |
+----------+
|      394 |
+----------+
1 row in set (0.04 sec)
```

Restaurant: 1401

```
mysql> SELECT COUNT(*) FROM Restaurant;
+----------+
| COUNT(*) |
+----------+
|     1401 |
+----------+
1 row in set (0.04 sec)
```

Bar: 1275

```
mysql> SELECT COUNT(*) FROM Bar;
+----------+
| COUNT(*) |
+----------+
|     1275 |
+----------+
1 row in set (0.05 sec)
```

Cafe:1348

```
mysql> SELECT COUNT(*) FROM Cafe;
+----------+
| COUNT(*) |
+----------+
|     1348 |
+----------+
1 row in set (0.04 sec)
```

ServeRestaurant: 1998

```
mysql> SELECT COUNT(*) FROM ServeRestaurant;
+----------+
| COUNT(*) |
+----------+
|     1998 |
+----------+
1 row in set (0.04 sec)
```

ServeBar: 2000

```
mysql> SELECT COUNT(*) FROM ServeBar;
+----------+
| COUNT(*) |
+----------+
|     2000 |
+----------+
1 row in set (0.04 sec)
```

ServeCafe: 1998

```
mysql> SELECT COUNT(*) FROM ServeCafe;
+----------+
| COUNT(*) |
+----------+
|     1998 |
+----------+
1 row in set (0.04 sec)
```

OrderRestaurant: 1500

```
mysql> SELECT COUNT(*) FROM OrderRestaurant;
+----------+
| COUNT(*) |
+----------+
|     1500 |
+----------+
1 row in set (0.04 sec)
```

OrderBar:1500

```
mysql> SELECT COUNT(*) FROM OrderBar;
+----------+
| COUNT(*) |
+----------+
|     1500 |
+----------+
1 row in set (0.05 sec)
```

OrderCafe:1500

```
mysql> SELECT COUNT(*) FROM OrderCafe;
+----------+
| COUNT(*) |
+----------+
|     1500 |
+----------+
1 row in set (0.04 sec)
```

5. Advanced Queries

Note that in our case (a place-to-eat roulette web app), we would only generate 1 place in the end and give it back to the user. However, stage 3 requires that we return the first 15 lines, so below will be queries that align with our logic and may be used as intermediate steps in the future, but may not be used in the exact same manner for final results.

Scenario 1: Given a user who wants to eat at a restaurant with a history of preference for cheap places, we may want to first pull out all restaurants with the following conditions:

- high ratings (>4.5) and many reviews (>100)
- price levels 1 and 2
- At least some of the food items are some user's favorites

The following query involves both JOIN and subquery:

```
3 •  SELECT DISTINCT res_name, price_level, rating, num_ratings, address
4    FROM Restaurant r JOIN ServeRestaurant s ON r.id = s.res_id
5    WHERE price_level IN (1, 2)
6          AND rating > 4.5
7          AND num_ratings > 100
8          AND food_id IN (SELECT food_id FROM Favorites)
9    ORDER BY rating DESC, num_ratings DESC
10   LIMIT 15;
11
```

100% 17:9

Result Grid Filter Rows: Search Export: Fetch rows:

res_name	price_level	rating	num_ratings	address
Offset BBQ	2	4.9	313	1720 N California Ave, Chicago, IL 60647, USA
BASE CAFE (8A5E)	2	4.9	121	1200 W 35th St, Chicago, IL 60609, USA
Kie-Gol-Lanee	2	4.8	467	5004 N Sheridan Rd, Chicago, IL 60640, USA
Ritual Coffeehouse	1	4.8	300	1821 W Irving Park Rd, Chicago, IL 60613, USA
Torteria San Lenchito	1	4.8	295	4810 N Drake Ave, Chicago, IL 60625, USA
Barbaro Taqueria	1	4.8	250	2525 W North Ave, Chicago, IL 60647, USA
Sinya Mediterranean	2	4.8	209	3224 N Damen Ave, Chicago, IL 60618, USA
Somethin' Sweet Donuts	1	4.8	181	4456 N Kedzie Ave, Chicago, IL 60625, USA
Mario's Italian Lemonade	1	4.7	2013	1068 W Taylor St, Chicago, IL 60607, USA
Tweet	2	4.7	1575	5020 N Sheridan Rd, Chicago, IL 60640, USA
Pasta D'Arte Trattoria It…	2	4.7	929	6311 N Milwaukee Ave, Chicago, IL 60646, USA
Scooter's Frozen Custard	1	4.7	654	1658 W Belmont Ave, Chicago, IL 60657, USA
Goree Cuisine	2	4.7	647	1126 E 47th St, Chicago, IL 60653, USA
Viaggio Restaurant Chi…	2	4.7	556	1330 W Madison St, Chicago, IL 60607, USA
The California Clipper	2	4.7	497	1002 N California Ave, Chicago, IL 60622, USA

Scenario 2: Let's say that we would like to obtain some summary statistics to gain a better picture of our user's behavior. We would like to know the number of orders users, if they do have this kind of order, placed where the food item ordered is NOT their favorite, so they stepped out of their comfort zone using our web app. And we would like to know these for cafe and restaurant.
Note that the username seems to be random characters because they are indeed randomly generated.

The following query gives us the report using set operations, jOIN, subqueries and GROUP BY:

```
(SELECT username, COUNT(*) AS num_order_brave
FROM Customer c NATURAL JOIN OrderRestaurant o
WHERE o.food_id IN (
    SELECT food_id FROM Customer c1 NATURAL JOIN Favorites
    WHERE c.username != c1.username
    )
GROUP by username)
UNION
(SELECT username, COUNT(*) AS num_order_brave
FROM Customer c NATURAL JOIN OrderCafe o
WHERE o.food_id IN (
    SELECT food_id FROM Customer c1 NATURAL JOIN Favorites
    WHERE c.username != c1.username
    )
GROUP by username)
ORDER BY num_order_brave DESC
LIMIT 15;
```

100% 46:21

Result Grid Filter Rows: Search Export:

username	num_order_bra...
INatGfXESRBFRsSFLiiMmtu...	7
AirAqtPZmwmcultOmlhIBPp...	7
ikpyc	7
OtjVroCHOurtYEogxoHfTcM...	6
rsAfbTasNlAXKxbmvuQc	6
TjC	6
RORXPanDEAFKReQZcbd...	6
AiFGSatLHRMpzhpzChpLJaY	6
pkZM	5
RqJuqWHqtCvIC	5
ikpyc	5
YfdlxkRbjbSbDasiLkUxKZRM	5
zUCPZClqognFGgaClxWu	5
tojDLmHDlGOviOGicQppbA...	5
XvamqNpwSEhG	5

6. Indexing Analysis

Scenario 1: Looking for cheap but good restaurants:

- The original query:

```
3 •   EXPLAIN ANALYZE SELECT DISTINCT res_name, price_level, rating, num_ratings, address
4     FROM Restaurant r JOIN ServeRestaurant s ON r.id = s.res_id
5     WHERE price_level IN (1, 2)
6           AND rating > 4.5
7           AND num_ratings > 100
8           AND food_id IN (SELECT food_id FROM Favorites)
9     ORDER BY rating DESC, num_ratings DESC
10    LIMIT 15;
```

- Results after running EXPLAIN ANALYZE:

```
-> Limit: 15 row(s)  (actual time=2.339..2.342 rows=15 loops=1)
    -> Sort: r.rating DESC, r.num_ratings DESC, limit input to 15 row(s) per chunk  (actual time=2.338..2.340 rows=15 loops=1)
        -> Table scan on <temporary>  (cost=2.50..2.50 rows=1) (actual time=0.001..0.006 rows=48 loops=1)
            -> Temporary table with deduplication  (cost=435.53..435.53 rows=1) (actual time=2.302..2.311 rows=48 loops=1)
                -> Inner hash join (r.id = s.res_id)  (cost=432.97 rows=1) (actual time=1.357..2.224 rows=59 loops=1)
                    -> Filter: ((r.price_level in (1,2)) and (r.rating > 4.5) and (r.num_ratings > 100))  (cost=69.82 rows=1) (actual time=0.039..0.877 rows=155 loops=1)
                        -> Table scan on r  (cost=69.82 rows=1401) (actual time=0.035..0.682 rows=1401 loops=1)
                    -> Hash
                        -> Nested loop inner join  (cost=213.53 rows=792) (actual time=0.118..1.123 rows=566 loops=1)
                            -> Remove duplicates from input sorted on food_id  (cost=35.64 rows=394) (actual time=0.095..0.257 rows=336 loops=1)
                                -> Index scan on Favorites using food_id  (cost=35.64 rows=394) (actual time=0.094..0.211 rows=394 loops=1)
                            -> Index lookup on s using PRIMARY (food_id=Favorites.food_id)  (cost=98.89 rows=2) (actual time=0.002..0.002 rows=2 loops=336)

1 row in set (0.04 sec)
```

- Index designs:

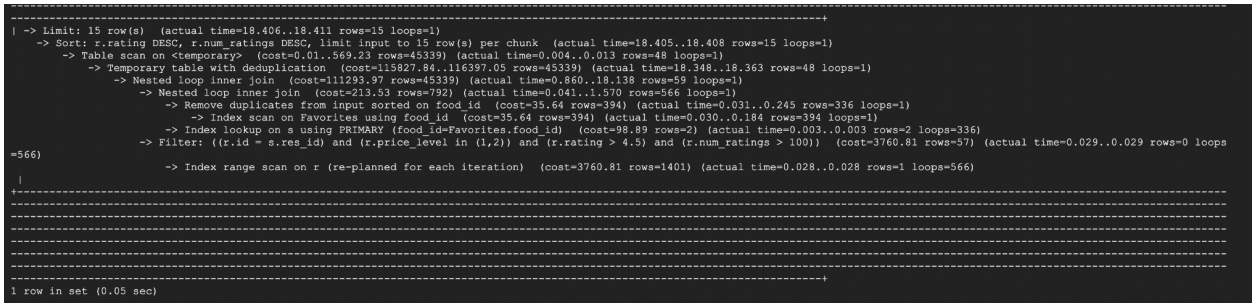
Importantly, primary keys are automatically indexed. In our case, there are the 3 relations involved in this query, all attributes in Favorite and ServeRestaurant are primary keys, so that leaves us with attributes in the Restaurant table.

One observation is that there is a “table scan” on table r that costs 143.5 in the results. This seems to arise from the filtering of the table based on the boolean expressions in the WHERE clause. Therefore, it seems that the query may benefit from 3 additional indices from the 3 boolean conditions on r that could potentially speed up the filtering process. However, if all 3 filtering attributes are indexed, then 4 out of 8 attributes will be indices, this could potentially create a problem because there are too many indices.

Therefore, we would like to start by indexing all of them and dropping one-by-one to see what combination gives the best results and whether there exists an optimal number of indices.

- Index design 1: Restaurant.price_level + Restaurant.rating + Restaurant.num_rating

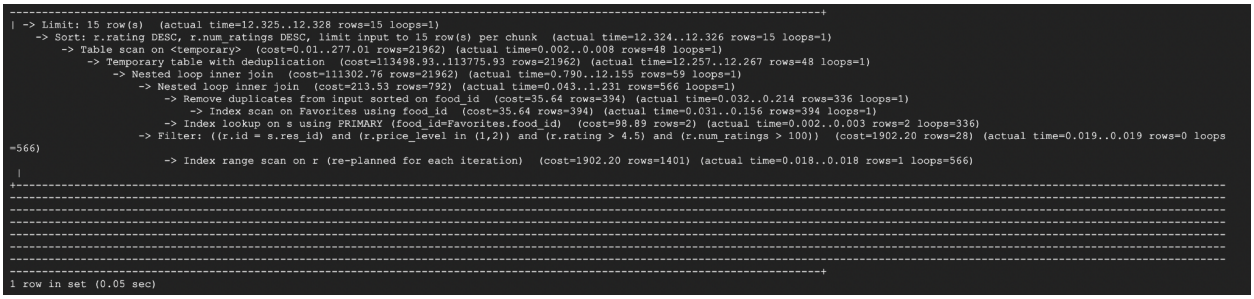
  As expected, adding all 3 indices did not have benefits but made the query slower than the default one. We see that a lot of steps have changed including how the JOIN



happened, and the costs for each new step have risen significantly, so it could be that there are too many indices for the Restaurant relation

- Index design 2: Restaurant.price_level + Restaurant.rating

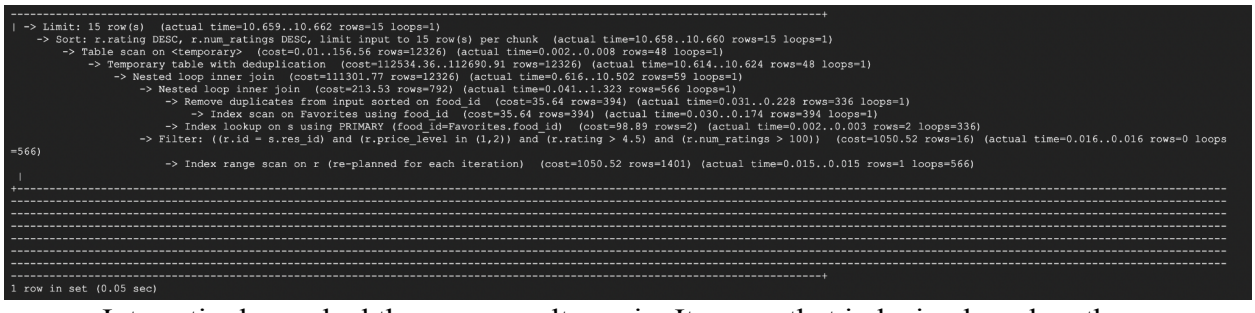
  Next, we kept only the price_level and rating. We should expect slightly better performance if fewer indices mean less confusion or a much better performance if the number of indices is right within the appropriate range.



Unfortunately, the result is almost the same as the previous one, showing really slow performance. It could be that 2 additional indices on the same relation are still too much, so we kept only the price_level.

- Index design 3: Restaurant.price_level

  The reason that we kept the price_level is that price level only allows integer values from 1 to 4, so indexing on this column alone should be efficient and lightweight.



Interestingly, we had the same results again. It seems that indexing based on the where clause does not help the filtering process at all in this case and changed the whole query process. We observed that there is a “re-planned” process that was not here in the

default design. It is possible that in this case, the default indices are already near the optimal case and the extra indices are only adding confusing information to complicate the process.

Scenario 2: Brave user statistics report:

- The original query:

```
EXPLAIN ANALYZE (SELECT username, COUNT(*) AS num_order_brave
FROM Customer c NATURAL JOIN OrderRestaurant o
WHERE o.food_id IN (
    SELECT food_id FROM Customer c1 NATURAL JOIN Favorites
    WHERE c.username != c1.username
    )
GROUP by username)
UNION
(SELECT username, COUNT(*) AS num_order_brave
FROM Customer c NATURAL JOIN OrderCafe o
WHERE o.food_id IN (
    SELECT food_id FROM Customer c1 NATURAL JOIN Favorites
    WHERE c.username != c1.username
    )
GROUP by username)
ORDER BY num_order_brave DESC
LIMIT 15;
```

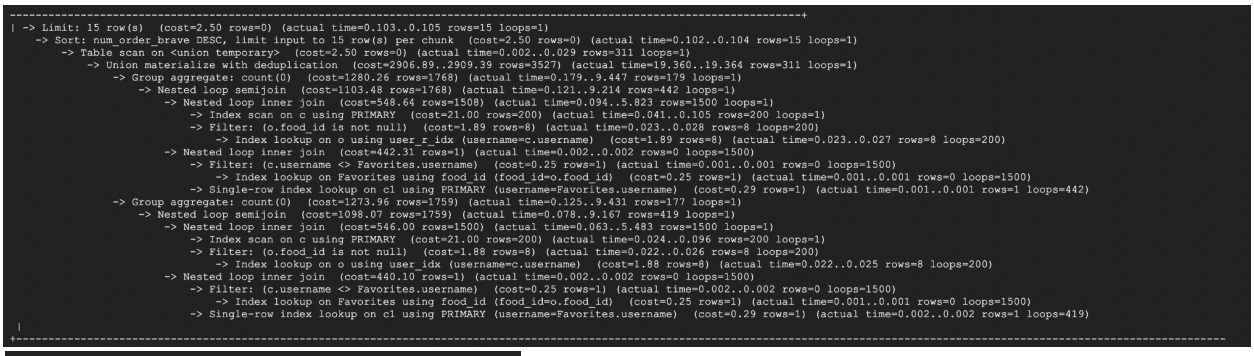
- Results after running EXPLAIN ANALYZE:

```
-> Limit: 15 row(s)  (cost=2.50 rows=0) (actual time=0.094..0.097 rows=15 loops=1)
    -> Sort: num_order_brave DESC, limit input to 15 row(s) per chunk  (cost=2.50 rows=0) (actual time=0.093..0.095 rows=15 loops=1)
        -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.001..0.027 rows=311 loops=1)
            -> Union materialize with deduplication  (cost=2.50..2.50 rows=0) (actual time=15.165..15.169 rows=311 loops=1)
                -> Table scan on <temporary>  (actual time=0.001..0.016 rows=179 loops=1)
                    -> Aggregate using temporary table  (actual time=8.004..8.034 rows=179 loops=1)
                        -> Nested loop semijoin  (cost=1228.82 rows=1759) (actual time=0.116..7.587 rows=442 loops=1)
                            -> Nested loop inner join  (cost=676.75 rows=1500) (actual time=0.091..3.720 rows=1500 loops=1)
                                -> Filter: ((o.username is not null) and (o.food_id is not null))  (cost=151.75 rows=1500) (actual time=0.047..0.859 rows=1500 loops=1)
                                    -> Table scan on o  (cost=151.75 rows=1500) (actual time=0.046..0.600 rows=1500 loops=1)
                                -> Single-row index lookup on c using PRIMARY (username=o.username)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1500)
                            -> Nested loop inner join  (cost=440.10 rows=1) (actual time=0.002..0.002 rows=0 loops=1500)
                                -> Filter: (o.username <> Favorites.username)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=0 loops=1500)
                                    -> Index lookup on Favorites using food_id (food_id=o.food_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1500)
                                -> Single-row index lookup on c1 using PRIMARY (username=Favorites.username)  (cost=0.29 rows=1) (actual time=0.002..0.002 rows=1 loops=442)
                -> Table scan on <temporary>  (actual time=0.000..0.015 rows=177 loops=1)
                    -> Aggregate using temporary table  (actual time=6.858..6.887 rows=177 loops=1)
                        -> Nested loop semijoin  (cost=1228.82 rows=1759) (actual time=0.051..6.489 rows=419 loops=1)
                            -> Nested loop inner join  (cost=676.75 rows=1500) (actual time=0.040..3.179 rows=1500 loops=1)
                                -> Filter: ((o.username is not null) and (o.food_id is not null))  (cost=151.75 rows=1500) (actual time=0.035..0.817 rows=1500 loops=1)
                                    -> Table scan on o  (cost=151.75 rows=1500) (actual time=0.034..0.577 rows=1500 loops=1)
                                -> Single-row index lookup on c using PRIMARY (username=o.username)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1500)
                            -> Nested loop inner join  (cost=440.10 rows=1) (actual time=0.002..0.002 rows=0 loops=1500)
                                -> Filter: (o.username <> Favorites.username)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1500)
                                    -> Index lookup on Favorites using food_id (food_id=o.food_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1500)
                                -> Single-row index lookup on c1 using PRIMARY (username=Favorites.username)  (cost=0.29 rows=1) (actual time=0.001..0.001 rows=1 loops=419)
```

```
1 row in set, 2 warnings (0.06 sec)
```

- Index design 1: OrderRestaurant.username + OrderCafe.username

  We see that the NATURAL JOIN between Customer and the Order relations take up much time and has a high cost. In addition, the 2 subqueries unioned are both grouped by username. So we think that perhaps indices on the username column of the Order_ relations could help with the JOIN and group process (username is not PK in the Order_ relations.)

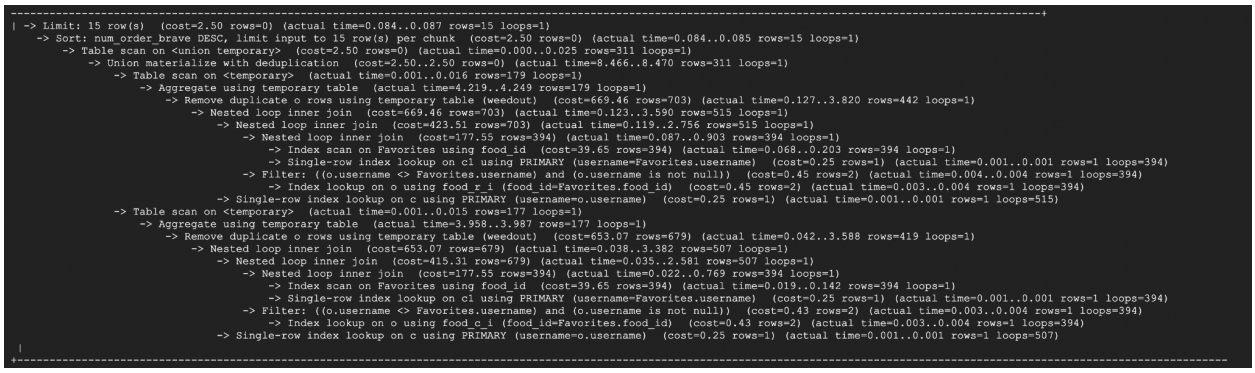



```
1 row in set, 2 warnings (0.06 sec)
```

Unfortunately, the total time spent did not change. However, we see that the username index was indeed used: the table scan in the default mode changed to index lookup and the number of rows scanned decreased from 1500 to 8. This is a great improvement, but not enough to reflect on the global scale.

- Index design 2: OrderRestaurant.food_id + OrderCafe.food_id

  Another combination of indices we tried are the 2 food ids in the 2 Order_ tables. This is because the WHERE clause in each subqueries unioned involves looking up the Order_'s food_id, so perhaps indexing it will speed up the query. Importantly we dropped the previous 2 indices to isolate the effect of food_ids.




```
1 row in set, 2 warnings (0.05 sec)
```

The food_id indices are again used here. Interestingly, The performance improvement this time is due to the fact that we can rely on the reduced number of entries due to index for the Order_ table compared to the native number of rows in Favorites. Previously, without the defined food_index, the filtering used the Favorite table's native primary key, and there there were 1500 loops corresponding to the 1500 rows. Now with the defined food_idx on Order_, the number of loops decreased to 394. We see that the final performance increased with a total time of 0.05 sec compared to the 0.06 previously.

- Index design 3: OrderRestaurant.username + OrderCafe.username + OrderRestaurant.food_id + OrderCafe.food_id

Finally, as a follow-up to our attempts in the Scenario 1, what would happen if we use both indices in each Order_ table? Will we see a better, because of the useful indices, or worse performance because the number of indices starts to create confusion? We therefore applied all four indices:

```
-> Limit: 15 row(s)  (cost=2.50 rows=0) (actual time=0.093..0.096 rows=15 loops=1)
    -> Sort: num_order_brave DESC, limit input to 15 row(s) per chunk  (cost=2.50 rows=0) (actual time=0.092..0.094 rows=15 loops=1)
        -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.000..0.027 rows=311 loops=1)
            -> Union materialize with deduplication  (cost=2.50..2.50 rows=0) (actual time=8.394..8.398 rows=311 loops=1)
                -> Table scan on <temporary>  (actual time=0.001..0.016 rows=179 loops=1)
                    -> Aggregate using temporary table  (actual time=4.128..4.157 rows=179 loops=1)
                        -> Remove duplicate o rows using temporary table (weedout)  (cost=669.46 rows=703) (actual time=0.089..3.743 rows=442 loops=1)
                            -> Nested loop inner join  (cost=669.46 rows=703) (actual time=0.085..3.537 rows=515 loops=1)
                                -> Nested loop inner join  (cost=423.51 rows=703) (actual time=0.082..2.686 rows=515 loops=1)
                                    -> Nested loop inner join  (cost=177.55 rows=394) (actual time=0.051..0.845 rows=394 loops=1)
                                        -> Index scan on Favorites using food_id  (cost=39.65 rows=394) (actual time=0.034..0.167 rows=394 loops=1)
                                        -> Single-row index lookup on c1 using PRIMARY (username=Favorites.username)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=394)
                                    -> Filter: ((o.username <> Favorites.username) and (o.username is not null))  (cost=0.45 rows=2) (actual time=0.004..0.004 rows=1 loops=394)
                                        -> Index lookup on o using food_r_i (food_id=Favorites.food_id)  (cost=0.45 rows=2) (actual time=0.003..0.004 rows=1 loops=394)
                                -> Single-row index lookup on c using PRIMARY (username=o.username)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=515)
                -> Table scan on <temporary>  (actual time=0.001..0.016 rows=177 loops=1)
                    -> Aggregate using temporary table  (actual time=3.944..3.974 rows=177 loops=1)
                        -> Remove duplicate o rows using temporary table (weedout)  (cost=653.07 rows=679) (actual time=0.044..3.584 rows=419 loops=1)
                            -> Nested loop inner join  (cost=653.07 rows=679) (actual time=0.041..3.386 rows=507 loops=1)
                                -> Nested loop inner join  (cost=415.31 rows=679) (actual time=0.038..2.586 rows=507 loops=1)
                                    -> Nested loop inner join  (cost=177.55 rows=394) (actual time=0.023..0.783 rows=394 loops=1)
                                        -> Index scan on Favorites using food_id  (cost=39.65 rows=394) (actual time=0.019..0.142 rows=394 loops=1)
                                        -> Single-row index lookup on c1 using PRIMARY (username=Favorites.username)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=394)
                                    -> Filter: ((o.username <> Favorites.username) and (o.username is not null))  (cost=0.43 rows=2) (actual time=0.003..0.004 rows=1 loops=394)
                                        -> Index lookup on o using food_c_i (food_id=Favorites.food_id)  (cost=0.43 rows=2) (actual time=0.003..0.004 rows=1 loops=394)
                                -> Single-row index lookup on c using PRIMARY (username=o.username)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=507)
```

```
1 row in set, 2 warnings (0.04 sec)
```

It seems that our performance increased further with all 4 indices: total time reduced to 0.04 from 0.06 secs. However, after examining the results, it seems that the query was run in the same manner as the previous one with only food_id indices. The username indices were not used at all, so the change in total time could be subject to other factors that are irrelevant to indexing. It seems, therefore, the food_id index could help with this particular query.

7. Implementing the database tables locally or on GCP

As you can see from the screenshots above, we used both mysql benchmark and the GCP to complete this stage. We switched platform so that the screenshots look more clearly in some questions. The mysql workbench was run using a local machine (see below). All datasets were imported into tables using the workbench's import feature. Then the imported database was dumped into a sql file by the work bench and we uploaded this sql onto GCP. We performed analysis on GCP in a shared project mysql39 (see below for a screenshot).