Database Design

1. Data Definition Language

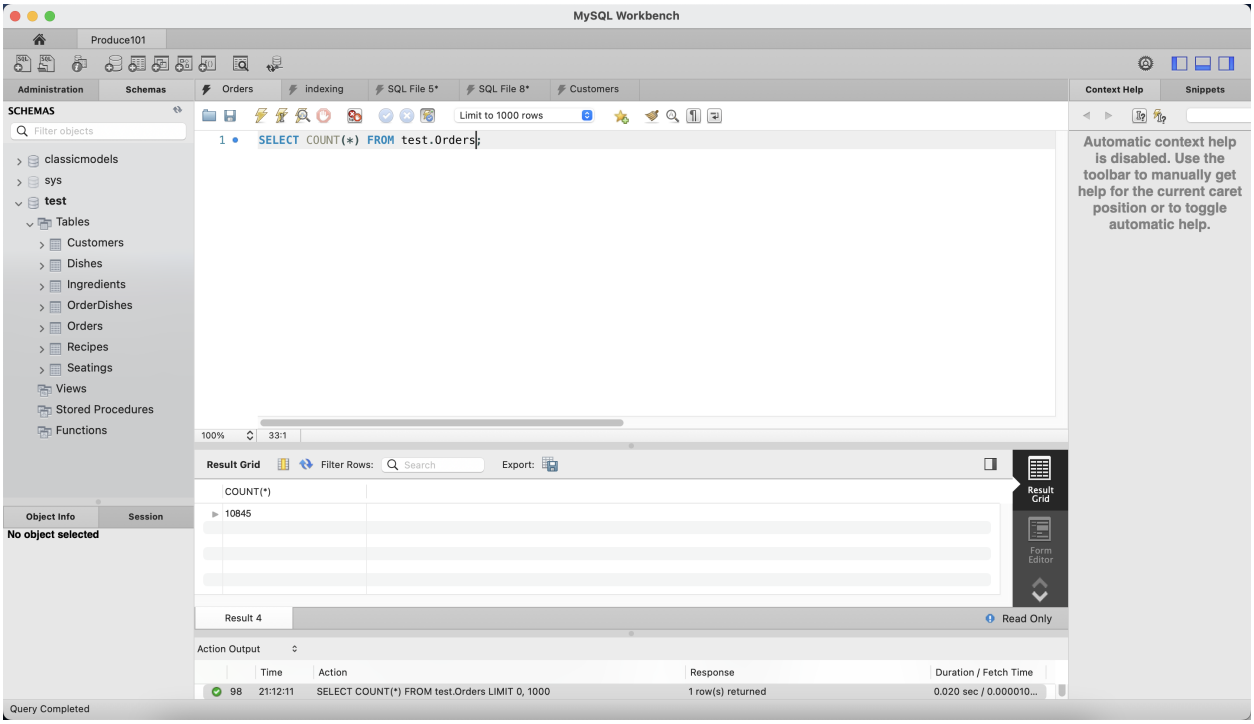
```
USE test;

CREATE TABLE Dishes(
    DishId INT PRIMARY KEY NOT NULL,
    DishName VARCHAR(32) NOT NULL,
    Price REAL NOT NULL,
    ImageUrl VARCHAR(256),
    Description VARCHAR(256)
);

CREATE TABLE Ingredients(
    IngredientId INT PRIMARY KEY NOT NULL,
    IngredientName VARCHAR(32) NOT NULL,
    Amount INT NOT NULL
);

CREATE TABLE Seatings(
    SeatingId INT PRIMARY KEY NOT NULL,
    Capacity INT NOT NULL,
    Occupied BOOLEAN
);

CREATE TABLE Customers(
    CustomerId INT PRIMARY KEY NOT NULL,
    Password VARCHAR(32) NOT NULL,
    Name VARCHAR(32) NOT NULL,
    Phone VARCHAR(16) DEFAULT NULL,
    FavoriteFood INT,
    FOREIGN KEY (FavoriteFood) REFERENCES Dishes(DishId) ON DELETE SET NULL
);

CREATE TABLE Orders(
    OrderId INT PRIMARY KEY NOT NULL,
    StartTime DATETIME NOT NULL,
    FinishTime DATETIME DEFAULT NULL,
    TotalPrice REAL NOT NULL,
    CustomerId INT NOT NULL,
    SeatingId INT,
    FOREIGN KEY (CustomerId) REFERENCES Customers(CustomerId) ON UPDATE CASCADE,
    FOREIGN KEY (SeatingId) REFERENCES Seatings(SeatingId) ON UPDATE CASCADE
);

CREATE TABLE Recipes(
    DishId INT NOT NULL,
    IngredientId INT NOT NULL,
    PRIMARY KEY (DishId, IngredientId),
    FOREIGN KEY (DishId) REFERENCES Dishes(DishId) ON DELETE CASCADE,
    FOREIGN KEY (IngredientId) REFERENCES Ingredients(IngredientId) ON DELETE CASCADE
);

CREATE TABLE OrderDishes(
    OrderId INT NOT NULL,
    DishId INT,
    Amount INT,
    PRIMARY KEY (OrderId, DishId),
    FOREIGN KEY (OrderId) REFERENCES Orders(OrderId) ON DELETE CASCADE,
    FOREIGN KEY (DishId) REFERENCES Dishes(DishId) ON DELETE CASCADE
);
```

2. Connection & Table Information:

We build our database on GCP. Here shows our connection and database information.

```
lijingyu1222@cloudshell:~ (produce101-order-sytem)$ gcloud sql connect produce101instance --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2015
Server version: 8.0.26-google (Google)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| classicmodels      |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| test               |
+--------------------+
6 rows in set (0.00 sec)

mysql> use test;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```



We collect the dish data from: https://www.kaggle.com/datasets/pes12017000148/food-ingredients-and-recipe-dataset-with-images

We collect the customer data from the 411 course websites: https://canvas.illinois.edu/courses/30041/assignments/syllabus

We auto-generate other data for our application.

The following shows the table count larger than 1000.

```
mysql> show tables;
+----------------+
| Tables_in_test |
+----------------+
| Customers      |
| Dishes         |
| Ingredients    |
| OrderDishes    |
| Orders         |
| Recipes        |
| Seatings       |
+----------------+
7 rows in set (0.00 sec)

mysql> select count(*) from Customers;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from Orders;
+----------+
| count(*) |
+----------+
|    10845 |
+----------+
1 row in set (0.01 sec)

mysql> select count(*) from OrderDishes;
+----------+
| count(*) |
+----------+
|    65118 |
+----------+
1 row in set (0.00 sec)
```

3. Advanced SQL queries:

1 - STAR Customers ⭐

```
SELECT CustomerId, c.Name, COUNT(*) as ArrivalCount, SUM(OrderCost) AS TotalPrice
FROM Orders o NATURAL JOIN Customers c NATURAL JOIN
(SELECT OrderId, SUM(Price) AS OrderCost FROM OrderDishes NATURAL JOIN Dishes GROUP BY OrderId) AS DishPrice
GROUP BY CustomerId
ORDER BY ArrivalCount DESC, c.Name;
```

- Explanation: This query selects customer id, customer name, the customer's arrival times to the restaurant, and their total cost on all dishes (not final cost) ranked decreasingly by arrival times and increasingly by customer name.

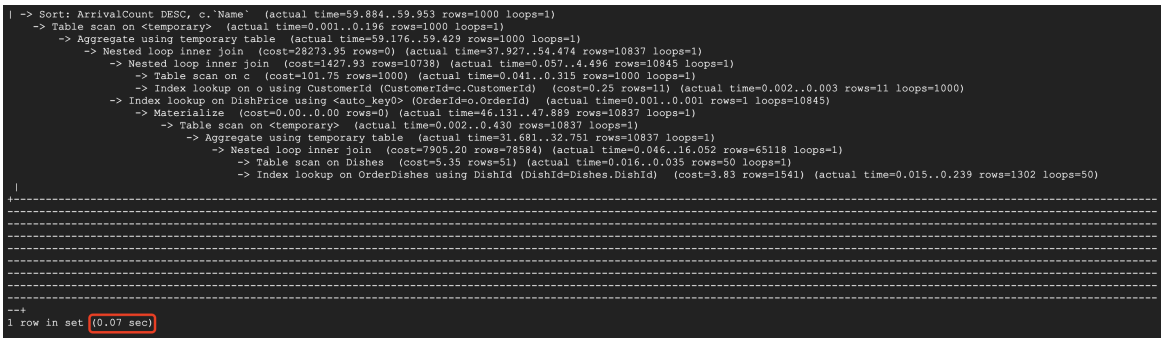
- Application: The customer list will be displayed on the menu page of our application as the **STAR** Customer to encourage more consumption.

- SQL concepts included: Join of multiple relations; Aggregation via GROUP BY; Subqueries

- Screenshot:

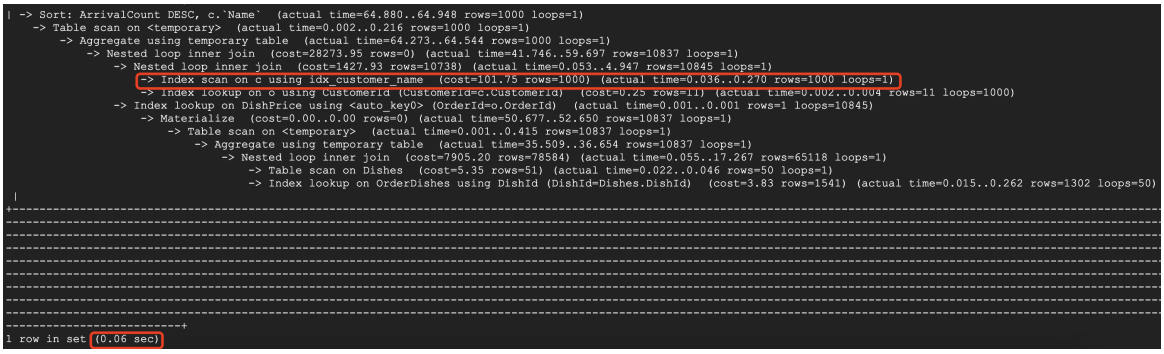
CustomerId	Name	ArrivalCount	TotalPrice
687	Young Julie	23	4206
142	Cruz Christina	21	3601
127	Koskitalo Peter	21	2856
693	Bennett Brian	20	2861
570	Hernandez Anna	20	2882
802	Tonini Akiko	20	3309
736	Bennett Leslie	19	2810
782	Hashimoto Dominique	19	2510
285	Holz Janine	19	2886
148	Lebihan Yoshi	19	3135
638	Moroni Jeff	19	2868
331	Müller Mel	19	2907
931	Piestrzeniewicz Palle	19	2782
426	Camino Jean	18	3192
541	Franco Matti	18	2752

- Index Performance Measurement
  i. Query performance before adding indexes



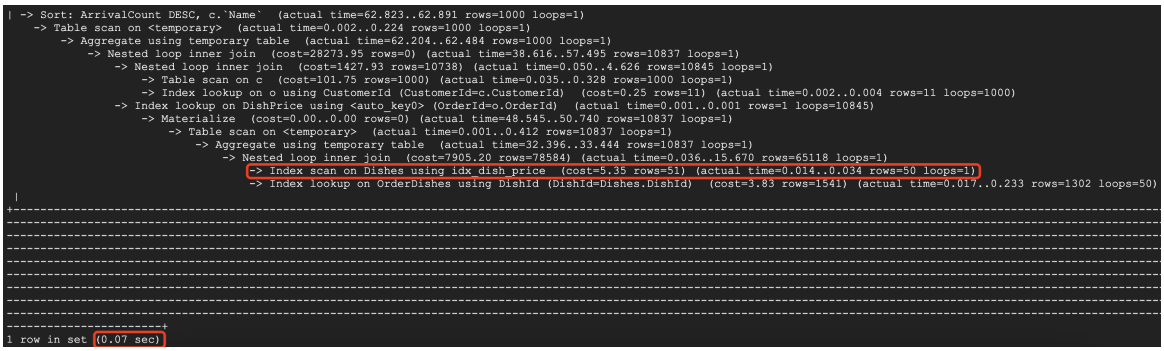
  ii. Query performance after adding indexes for Customer.name

```
CREATE INDEX idx_customer_name ON Customers (Name);
```

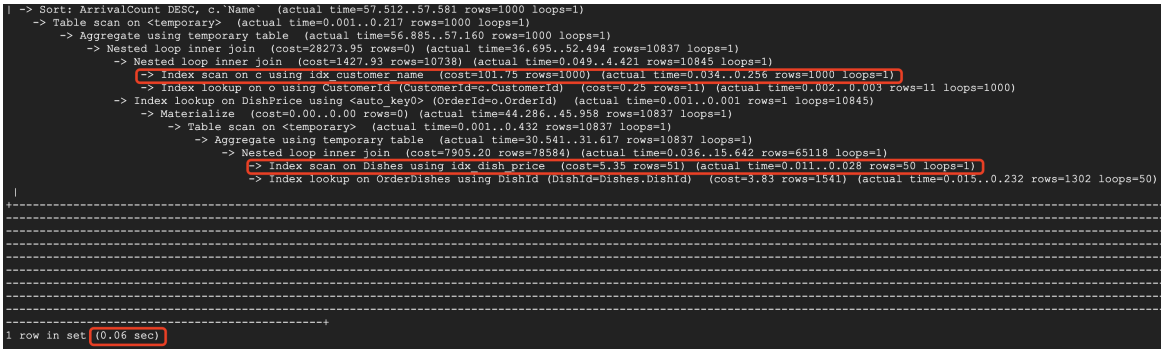


  iii. Query performance after adding indexes for Dishes.Price

```
CREATE INDEX idx_dish_price ON Dishes (Price);
```



  iv. Query performance after adding indexes for both Customer.name and Dishes.Price



- Index Analysis

  For this advanced query, we have 3 index design: Add an index on Customers.name; Add an index on Dishes.Price; Add both indices. We choose to add an index related to customer name since the attribute is included in the select list and the final results are sorted according to the name. The index can help make the customer name in sequence and accelerate the order process. We choose to add an index related to dish price since there is a SUM of the price in our sub-query and we wonder whether it can help speed up the aggregation. Finally we check whether adding both indices can do better than adding separately.

  As can be seen from the previous analysis screenshot. Adding an index on customer name brings a slight improvement on total execution time. In theory, when we include an ORDER BY clause in a SELECT statement, we can improve the performance by creating an index on the ORDER BY attribute. The database server uses the index to sort the query result in an efficient manner. However, the analysis shows that the index is only used when joining Customer table and does not bring essential effect. An index scan replaces the previous table scan. We have no way to know whether the index exactly help speed up the ordering procedure. We have the following reasons to explain the minor improvement: the data size is too small to reflect a large speedup; the final result is sorted by the count first and then by customer name, so the sort range decreases to customers having the same arrival times, which is small; there is no improvement on ORDER BY at all and the time is caused by varied server environment.

  For the index on Dish price, it brings no improvement on query performance. This fits the theoretical result. Generally speaking, indices are useful when finding a few targeted rows among many rows. They improve query by limiting how many hits to rows are required. An aggregation function like SUM has to search through all the entries in the price column, so adding the index does not help. From the analysis screenshot, the index is only used for joining the dish table and the cost is close to the original case.

  As for adding both indices, it brings slight improvement because of the customer name index and the other index does not make influence. The query time is the same as the second case.

2. Most Popular and Unpopular Food 🍛

```
(SELECT d.DishName, d.Price, COUNT(FavoriteFood) as count, Ingredient_Count
FROM Customers c JOIN Dishes d ON c.FavoriteFood = d.DishId JOIN
(SELECT DishId, COUNT(IngredientId) as Ingredient_Count FROM Recipes GROUP BY DishId) as t ON c.FavoriteFood = t.DishId
GROUP BY c.FavoriteFood
ORDER BY count DESC, d.Price, d.DishName, Ingredient_Count
LIMIT 10)
UNION
(SELECT d.DishName, d.Price, COUNT(FavoriteFood) as count, Ingredient_Count
FROM Customers c JOIN Dishes d ON c.FavoriteFood = d.DishId JOIN
(SELECT DishId, COUNT(IngredientId) as Ingredient_Count FROM Recipes GROUP BY DishId) as t ON c.FavoriteFood = t.DishId
GROUP BY c.FavoriteFood
ORDER BY count ASC, d.Price, d.DishName, Ingredient_Count
LIMIT 10)
ORDER BY count DESC;
```

- Explanation: This query selects most and least popular food (each 10) according to customers' preference. The results contain their name, price, how many ingredients do each of them contains, and their appearance frequency. The results are ordered by frequency in descending order, ordered by price in ascending order and ordered by dish name in ascending order.
- Application: The food list will be displayed on the menu page of our application, in order to provide ordering guidance for customers, especially the new ones.
- SQL concepts included: Join of multiple relations & Aggregation via GROUP BY & SET Operation
- Results:

DishName	Price	count	Ingredient_Count
Salt-and-Pepper Fish	24	30	8
Ginger and Tamarind Refresher	16	27	4
Maple Barbecue Grilled Chicken	19	27	9
Tomato Brown Butter	27	27	4
Instant Pot Lamb Haleem	5	26	11
Tomato Pie With Sour Cream Crust	13	26	17
Tomato and Roasted Garlic Pie	32	26	12
Peach and Sesame Crumble	35	25	10
Spicy Coconut Pumpkin Soup	40	25	11
Veselka's Famous Borscht	24	24	9
Shrimp Creole	6	17	11
Enfrijoladas	9	17	10
Tropi-Cobb Salad	2	16	10
Yogurt and Spice Roasted Salmon	40	15	10
Peanut Butter Brookies	13	14	10
Crispy Salt and Pepper Potatoes	37	14	8
Chicken Pelau	40	14	16
Caramelized Plantain Parfait	25	13	4
Green Seasoning	36	10	5
Caesar Salad Roast Chicken	14	8	10

- Index Performance Measurement

  i. Before adding indexes

```
| -> Sort: count DESC  (cost=2.50 rows=0) (actual time=0.011..0.013 rows=20 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.001..0.002 rows=20 loops=1)
        -> Union materialize with deduplication  (cost=2.50..2.50 rows=0) (actual time=2.239..2.241 rows=20 loops=1)
            -> Limit: 10 row(s)  (actual time=1.292..1.294 rows=10 loops=1)
                -> Sort: count DESC, d.Price, d.DishName, t.Ingredient_Count, limit input to 10 row(s) per chunk  (actual time=1.292..1.292 rows=10 loops=1)
                    -> Table scan on <temporary>  (actual time=0.001..0.010 rows=49 loops=1)
                        -> Aggregate using temporary table  (actual time=1.247..1.259 rows=49 loops=1)
                            -> Nested loop inner join  (cost=1283.84 rows=8837) (actual time=0.292..0.742 rows=1000 loops=1)
                                -> Nested loop inner join  (cost=289.61 rows=433) (actual time=0.274..0.348 rows=50 loops=1)
                                    -> Table scan on t  (cost=0.02..7.91 rows=433) (actual time=0.001..0.006 rows=50 loops=1)
                                        -> Materialize  (cost=130.17..138.06 rows=433) (actual time=0.184..0.191 rows=50 loops=1)
                                            -> Group aggregate: count(Recipes.IngredientId)  (cost=86.85 rows=433) (actual time=0.070..0.168 rows=50 loops=1)
                                                -> Index scan on Recipes using PRIMARY  (cost=43.55 rows=433) (actual time=0.066..0.140 rows=433 loops=1)
                                    -> Single-row index lookup on d using PRIMARY (DishId=t.DishId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=50)
                                -> Index lookup on c using FavoriteFood (FavoriteFood=t.DishId)  (cost=0.26 rows=20) (actual time=0.004..0.006 rows=20 loops=50)
        -> Limit: 10 row(s)  (actual time=0.915..0.916 rows=10 loops=1)
            -> Sort: count, d.Price, d.DishName, t.Ingredient_Count, limit input to 10 row(s) per chunk  (actual time=0.914..0.915 rows=10 loops=1)
                -> Table scan on <temporary>  (actual time=0.000..0.006 rows=49 loops=1)
                    -> Aggregate using temporary table  (actual time=0.875..0.884 rows=49 loops=1)
                        -> Nested loop inner join  (cost=1283.84 rows=8837) (actual time=0.139..0.503 rows=1000 loops=1)
                            -> Nested loop inner join  (cost=289.61 rows=433) (actual time=0.131..0.191 rows=50 loops=1)
                                -> Table scan on t  (cost=0.02..7.91 rows=433) (actual time=0.000..0.005 rows=50 loops=1)
                                    -> Materialize  (cost=130.17..138.06 rows=433) (actual time=0.125..0.132 rows=50 loops=1)
                                        -> Group aggregate: count(Recipes.IngredientId)  (cost=86.85 rows=433) (actual time=0.029..0.113 rows=50 loops=1)
                                            -> Index scan on Recipes using PRIMARY  (cost=43.55 rows=433) (actual time=0.026..0.087 rows=433 loops=1)
                                -> Single-row index lookup on d using PRIMARY (DishId=t.DishId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=50)
                            -> Index lookup on c using FavoriteFood (FavoriteFood=t.DishId)  (cost=0.26 rows=20) (actual time=0.003..0.005 rows=20 loops=50)
|
```

```
+----------------------+
1 row in set (0.01 sec)
```

ii. Index on Dishes.Price

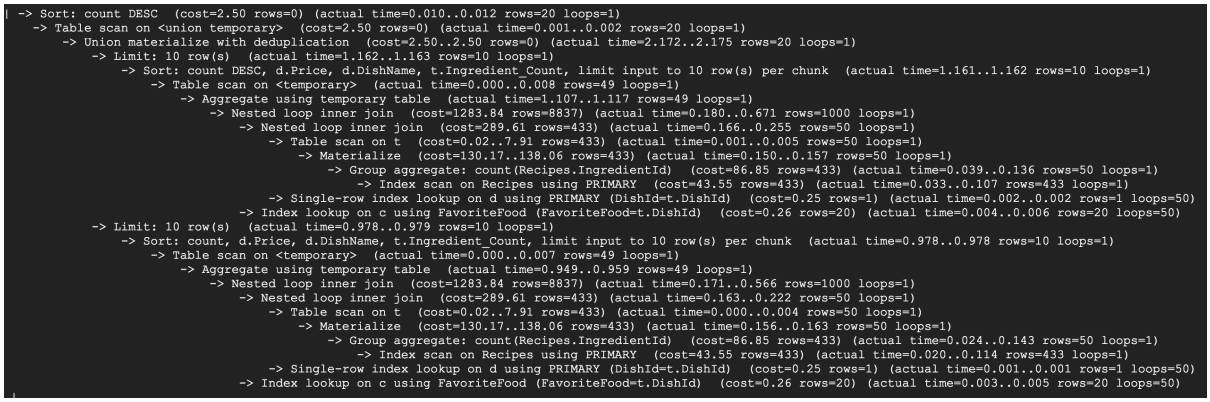
```
CREATE INDEX price ON Dishes(Price);
```

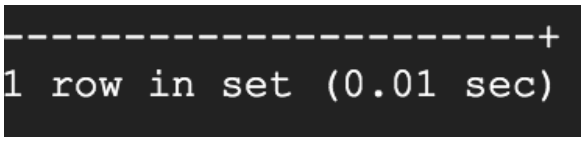
```
| -> Sort: count DESC  (cost=2.50 rows=0) (actual time=0.009..0.010 rows=20 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.000..0.002 rows=20 loops=1)
        -> Union materialize with deduplication  (cost=2.50..2.50 rows=0) (actual time=1.971..1.974 rows=20 loops=1)
            -> Limit: 10 row(s)  (actual time=1.047..1.049 rows=10 loops=1)
                -> Sort: count DESC, d.Price, d.DishName, t.Ingredient_Count, limit input to 10 row(s) per chunk  (actual time=1.046..1.047 rows=10 loops=1)
                    -> Table scan on <temporary>  (actual time=0.001..0.008 rows=49 loops=1)
                        -> Aggregate using temporary table  (actual time=1.009..1.019 rows=49 loops=1)
                            -> Nested loop inner join  (cost=1283.84 rows=8837) (actual time=0.164..0.601 rows=1000 loops=1)
                                -> Nested loop inner join  (cost=289.61 rows=433) (actual time=0.152..0.223 rows=50 loops=1)
                                    -> Table scan on t  (cost=0.02..7.91 rows=433) (actual time=0.001..0.004 rows=50 loops=1)
                                        -> Materialize  (cost=130.17..138.06 rows=433) (actual time=0.141..0.147 rows=50 loops=1)
                                            -> Group aggregate: count(Recipes.IngredientId)  (cost=86.85 rows=433) (actual time=0.036..0.128 rows=50 loops=1)
                                                -> Index scan on Recipes using PRIMARY  (cost=43.55 rows=433) (actual time=0.032..0.101 rows=433 loops=1)
                                    -> Single-row index lookup on d using PRIMARY (DishId=t.DishId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=50)
                                -> Index lookup on c using FavoriteFood (FavoriteFood=t.DishId)  (cost=0.26 rows=20) (actual time=0.004..0.006 rows=20 loops=50)
        -> Limit: 10 row(s)  (actual time=0.897..0.898 rows=10 loops=1)
            -> Sort: count, d.Price, d.DishName, t.Ingredient_Count, limit input to 10 row(s) per chunk  (actual time=0.897..0.898 rows=10 loops=1)
                -> Table scan on <temporary>  (actual time=0.000..0.007 rows=49 loops=1)
                    -> Aggregate using temporary table  (actual time=0.871..0.881 rows=49 loops=1)
                        -> Nested loop inner join  (cost=1283.84 rows=8837) (actual time=0.126..0.504 rows=1000 loops=1)
                            -> Nested loop inner join  (cost=289.61 rows=433) (actual time=0.121..0.178 rows=50 loops=1)
                                -> Table scan on t  (cost=0.02..7.91 rows=433) (actual time=0.000..0.004 rows=50 loops=1)
                                    -> Materialize  (cost=130.17..138.06 rows=433) (actual time=0.117..0.123 rows=50 loops=1)
                                        -> Group aggregate: count(Recipes.IngredientId)  (cost=86.85 rows=433) (actual time=0.016..0.107 rows=50 loops=1)
                                            -> Index scan on Recipes using PRIMARY  (cost=43.55 rows=433) (actual time=0.014..0.081 rows=433 loops=1)
                                -> Single-row index lookup on d using PRIMARY (DishId=t.DishId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=50)
                            -> Index lookup on c using FavoriteFood (FavoriteFood=t.DishId)  (cost=0.26 rows=20) (actual time=0.003..0.005 rows=20 loops=50)
|
```

```
+----------------------+
1 row in set (0.01 sec)
```

iii. Index on Dishes.DishName

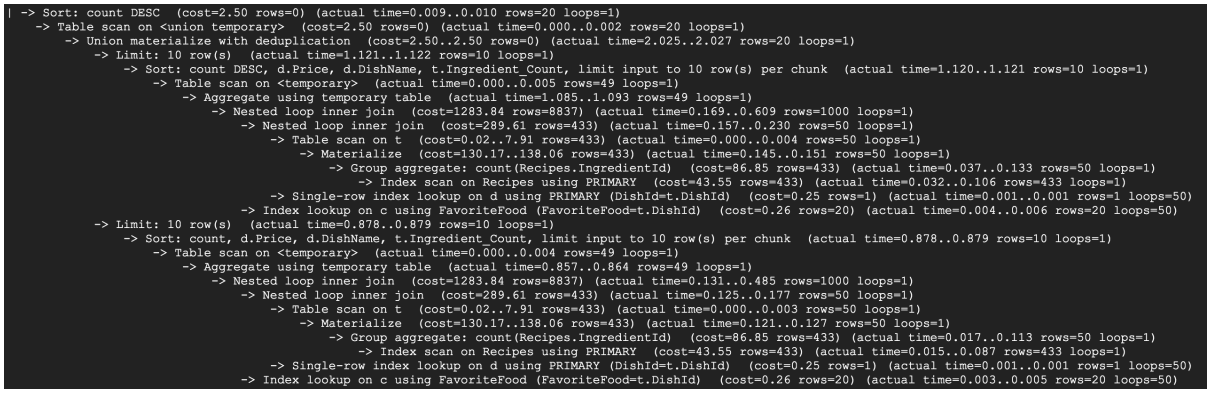
```
CREATE INDEX dishname ON Dishes(DishName);
```

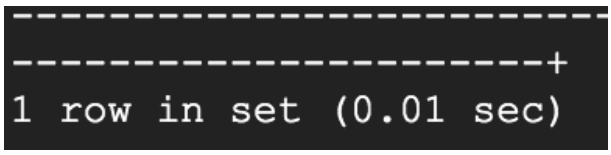




iv. Index on both Dishes.Price and Dishes.DishName

```
CREATE INDEX dishname ON Dishes(DishName);
CREATE INDEX price ON Dishes(Price);
```





- Index Analysis:

For this advanced query, we have 3 index designs: Add an index on Dishes.Price; Add an index on Dishes.DishName; Add both indices. We choose to add an index related to Dish name and price since the attributes are included in the select list and the final results are sorted according to the DishName and Price. Then we check whether adding both indices can do better than adding separately.

As can be seen from the previous analysis screenshot. None of 3 designs bring improvement on execution time. As mentioned in the last index analysis, when we include an ORDER BY clause in a SELECT statement, we hope that the database server uses the index to sort the query result in an efficient manner, thus improve the performance by creating an index on the ORDER BY attribute. But according to the results, the query only apply index scan and index look-up using already exist PRIMARY. It indicates that none of 3 index designs is used. Compared with the previous analysis, we found that this is because there is no table scan related to DishName or Price in original non-indexed query. Thus, these indices cannot bring expected improvement. From this index trial, we learn a lesson that not to create such index. Because not only they cannot bring improvement, they require extra storage and may become the burden in query execution.