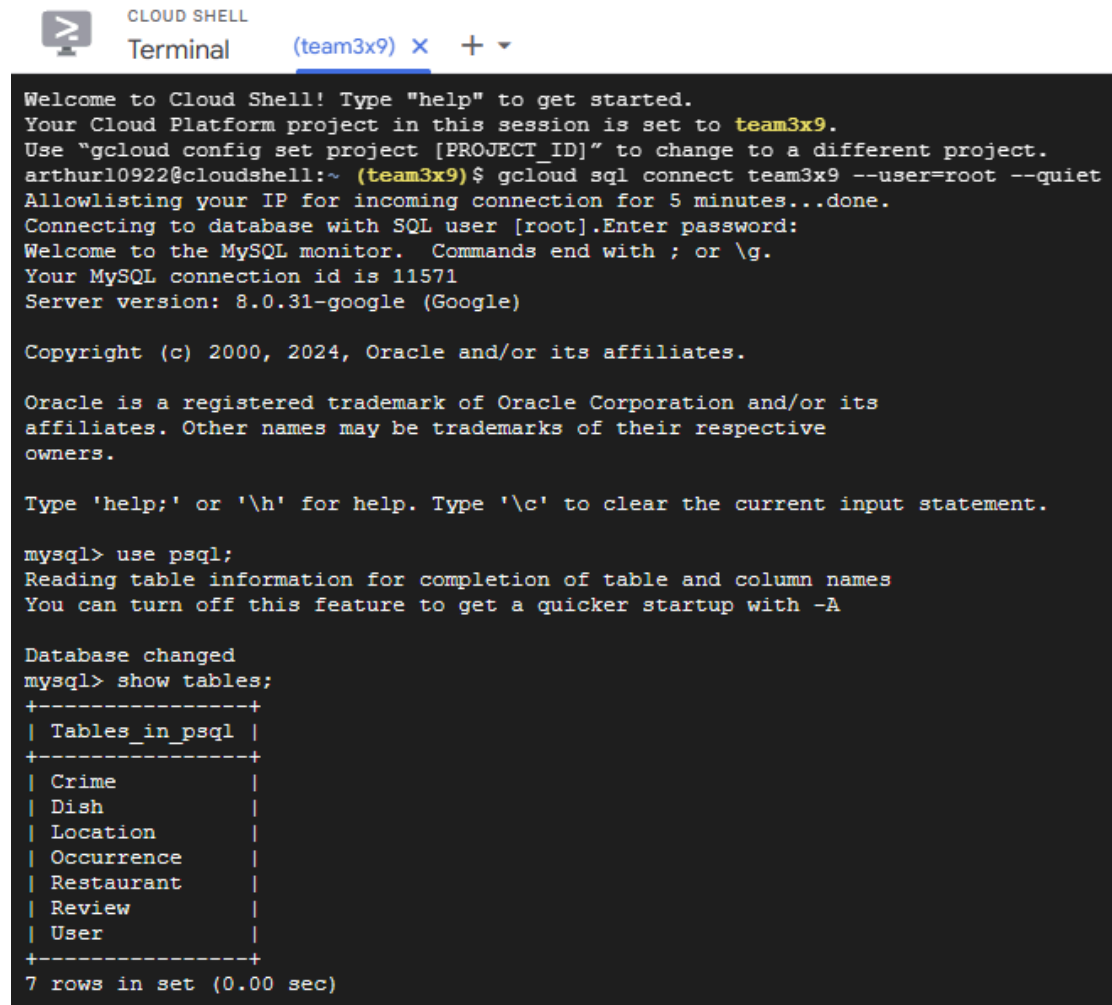


Database Design

❖ Database Implementation:

➤ Screenshot For Connection:



```
CLOUD SHELL
Terminal (team3x9) x + v

Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to team3x9.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
arthurl0922@cloudshell:~ (team3x9)$ gcloud sql connect team3x9 --user=root --quiet
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 11571
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, 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> use psql;
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
mysql> show tables;
+-----+
| Tables_in_psql |
+-----+
| Crime          |
| Dish           |
| Location       |
| Occurrence     |
| Restaurant     |
| Review         |
| User           |
+-----+
7 rows in set (0.00 sec)
```

➤ DDL Commands For Tables:

■ Restaurant:

```
CREATE TABLE Restaurant (
    RestaurantId INT PRIMARY KEY,
    LocationId INT,
    Hours VARCHAR(255),
```

Stars REAL,
Category VARCHAR(127),
RestaurantName VARCHAR(255),

FOREIGN KEY(LocationId) REFERENCES

Location(LocationId)

ON DELETE CASCADE

ON UPDATE CASCADE

);

■ Review:

CREATE TABLE Review (

RestaurantId INT,

UserId INT,

Stars REAL,

Text MEDIUMTEXT,

Date DATETIME,

PRIMARY KEY(RestaurantId, UserId, Date),

FOREIGN KEY(RestaurantId) REFERENCES

Restaurant(RestaurantId)

ON DELETE CASCADE

ON UPDATE CASCADE,

FOREIGN KEY(UserId) REFERENCES User(UserId)

ON DELETE CASCADE

ON UPDATE CASCADE

);

■ User:

```
CREATE TABLE User (  
    UserId INT PRIMARY KEY,  
    Name VARCHAR(31),  
    Taste VARCHAR(127),  
    Password VARCHAR(31)  
);
```

■ Location:

```
CREATE TABLE Location (  
    LocationId INT PRIMARY KEY,  
    Latitude REAL,  
    Longitude REAL,  
    PostalCode VARCHAR(31),  
    State VARCHAR(15),  
    City VARCHAR(31)  
);
```

■ Dish:

```
CREATE TABLE Dish (  
    DishId INT,  
    RestaurantId INT,  
    Price REAL,
```

Name VARCHAR(127),

PRIMARY KEY(DishId, RestaurantId),

FOREIGN KEY(RestaurantId) REFERENCES

Restaurant(RestaurantId)

ON DELETE CASCADE

ON UPDATE CASCADE

);

■ Crime:

CREATE TABLE Crime (

CrimeId INT PRIMARY KEY,

Count INT,

Type INT,

Cleared INT,

LocationId INT,

FOREIGN KEY(LocationId) REFERENCES

Location(LocationId)

ON DELETE CASCADE

ON UPDATE CASCADE

);

■ Occurrence:

CREATE TABLE Occurrence (

RestaurantId INT,

CrimeId INT,

PRIMARY KEY(RestaurantId, CrimeId),

FOREIGN KEY(RestaurantId) REFERENCES

Restaurant(RestaurantId)

ON DELETE CASCADE

ON UPDATE CASCADE,

FOREIGN KEY(CrimeId) REFERENCES Crime(CrimeId)

ON DELETE CASCADE

ON UPDATE CASCADE

);

➤ Screenshot for Rows of the Tables:

■ Restaurant:

```
mysql> SELECT COUNT(*) FROM Restaurant;
+-----+
| COUNT(*) |
+-----+
|      10000 |
+-----+
1 row in set (0.01 sec)
```

■ Review:

```
mysql> SELECT COUNT(*) FROM Review;
+-----+
| COUNT(*) |
+-----+
|       1000 |
+-----+
1 row in set (0.22 sec)
```

■ User:

```
mysql> SELECT COUNT(*) FROM User;
+-----+
| COUNT(*) |
+-----+
|      1000 |
+-----+
1 row in set (0.00 sec)
```

■ Location:

```
mysql> SELECT COUNT(*) FROM Location;
+-----+
| COUNT(*) |
+-----+
|      1000 |
+-----+
1 row in set (0.01 sec)
```

■ Dish:

```
mysql> SELECT COUNT(*) FROM Dish;
+-----+
| COUNT(*) |
+-----+
|     22854 |
+-----+
1 row in set (0.15 sec)
```

■ Crime:

```
mysql> SELECT COUNT(*) FROM Crime;
+-----+
| COUNT(*) |
+-----+
|     10000 |
+-----+
1 row in set (0.00 sec)
```

■ Occurrence:

```
mysql> SELECT COUNT(*) FROM Occurrence;
+-----+
| COUNT(*) |
+-----+
|     10000 |
+-----+
1 row in set (0.00 sec)
```

❖ Advanced Queries:

- Find (“Popular”) Restaurants that Has At Least 3 Users that Likes Its Category & Has Above 3.5 Stars:

```
SELECT R.RestaurantId, R.RestaurantName, R.Category, R.Stars
FROM Restaurant R JOIN User U ON R.Category = U.Taste
WHERE R.Stars >= 3.5
GROUP BY R.RestaurantId
HAVING COUNT(U.UserId) >= 3
ORDER BY R.Stars DESC;
```

- Screenshot for the Top 15 Rows:

```
mysql> SELECT R.RestaurantId, R.RestaurantName, R.Category, R.Stars
-> FROM Restaurant R JOIN User U ON R.Category = U.Taste
-> WHERE R.Stars >= 3.5
-> GROUP BY R.RestaurantId
-> HAVING COUNT(U.UserId) >= 3
-> ORDER BY R.Stars DESC
-> LIMIT 15;
```

RestaurantId	RestaurantName	Category	Stars
6419	Tucson's Loop Bicycle Shop	Earthy	5
6421	Unity Shoppe Inc	Salty	5
6437	Tag Inspections	Chewy	5
6439	Oliver's Antiques	Dry	5
6443	Our Lady of Mount Carmel Church	Tender	5
6446	Pat Moyer Wedding Photography and Films	Smokey	5
6456	T A Mahoney	Zesty	5
6477	Meagan Q Macklin	Crispy	5
6478	Liferoot Acupuncture & Healing Arts	Fiery	5
6483	New Systems HVAC	Fruity	5
6504	Podiatry Associates of Indiana Foot and Ankle Institute	Sweet	5
6510	MacDill Marina	Moist	5
6511	Simply Sweet Sugaring	Crunchy	5
6518	Hop MD	Creamy	5
6524	America's Mattress of Tucson	Nutty	5

15 rows in set (0.47 sec)

- Find Restaurants In the State of Illinois (IL) Where the Crimes that Happen Around the Restaurant <= 100 & Stars >= 3:

```
SELECT RestaurantId, RestaurantName, Category, Stars
FROM Restaurant R NATURAL JOIN Location L
WHERE R.Stars >= 3 AND L.State = 'IL' AND (SELECT SUM(C.Count)
FROM Crime C
```

WHERE C.LocationId = L.LocationId) <= 100

ORDER BY Stars DESC, RestaurantName ASC;

- Screenshot for the Top 15 Rows:

```
mysql> SELECT RestaurantId, RestaurantName, Category, Stars
-> FROM Restaurant R NATURAL JOIN Location L
-> WHERE R.Stars >= 3 AND L.State = 'IL' AND (SELECT SUM(C.Count)
-> FROM Crime C
-> WHERE C.LocationId = L.LocationId) <= 100
-> ORDER BY Stars DESC, RestaurantName ASC
-> LIMIT 15;
```

RestaurantId	RestaurantName	Category	Stars
4898	Auto Appearance	Umami	4.5
4251	Buff Beauty Bar	Zesty	4.5
176	Collision XS	Auto Repair, Body Shops, Automotive	4.5
3911	Palm Pavilion Inn	Smokey	4.5
2650	Petite Clouet Cafe	Crispy	4.5
9128	Royal Street Stones	Zesty	4.5
837	Triple R's Smokehouse	Restaurants, Barbeque	4.5
4867	WANG'S TABLE	Juicy	4.5
5548	Buddy Brew Coffee	Buttery	4
1982	Kimpton Hotel Monaco Philadelphia	Moist	4
4742	Rolling Stars	Mild	4
8330	Shop N Go II	Mild	4
7121	Trader Joe's	Juicy	4
7759	Agustin Brasserie	Frothy	3.5
8967	Froyo Fresh	Sweet	3.5

15 rows in set (0.00 sec)

- Find Locations that Has At Least 5 Restaurants Around It With Average Stars of Those Restaurants Being At Least 3 & Having Uncleared Crimes of Type 4, 5 and 6 Under 50 (Note: Crime Type Are Represented As Integers Where These Integers Will Each Map to a Specific Type (E.g. Type 1 = Robbery)):

SELECT State, City, PostalCode, COUNT(R.RestaurantId) AS

NumRestaurants, AVG(R.Stars) AS AverageStars

FROM Location L NATURAL JOIN Restaurant R NATURAL JOIN

Crime C

WHERE C.Type IN (4, 5, 6) AND C.Cleared = 0

GROUP BY L.LocationId

HAVING COUNT(R.RestaurantId) >= 5 AND AVG(R.Stars) >= 3 AND

SUM(C.Count) <= 50

ORDER BY AverageStars DESC, NumRestaurants DESC;

■ Screenshot for the Top 15 Rows:

```
mysql> SELECT State, City, PostalCode, COUNT(R.RestaurantId) AS NumRestaurants, AVG(R.Stars) AS AverageStars
-> FROM Location L NATURAL JOIN Restaurant R NATURAL JOIN Crime C
-> WHERE C.Type IN (4, 5, 6) AND C.Cleared = 0
-> GROUP BY L.LocationId
-> HAVING COUNT(R.RestaurantId) >= 5 AND AVG(R.Stars) >= 3 AND SUM(C.Count) <= 50
-> ORDER BY AverageStars DESC, NumRestaurants DESC
-> LIMIT 15;
```

State	City	PostalCode	NumRestaurants	AverageStars
FL	St. Petersburg	33713	11	4.2272727272727275
PA	Philadelphia	19125	7	4.214285714285714
FL	Clearwater	33767	9	4.166666666666667
FL	Wesley Chapel	33544	6	4.166666666666667
NJ	Haddonfield	08033	10	4.15
TN	Nashville	37211	27	4.111111111111111
PA	Norristown	19401	12	4
ID	Boise	83705	10	4
NJ	Medford	08055	6	4
MO	Normandy	63121	16	3.875
FL	Tampa	33618	8	3.875
AZ	Tucson	85705	8	3.8125
FL	Clearwater	33755	7	3.7857142857142856
TN	Goodlettsville	37072	10	3.75
PA	Philadelphia	19103	8	3.75

15 rows in set (0.07 sec)

- Find Restaurants that Open on Friday From 11:00-23:00 Where the “Cleared Crime Rate” (I.e. Cleared Crimes / Total Number of Crimes) ≥ 0.1 and Restaurants that Open on Monday From 11:00-17:00 Where the Average Dish Price ≤ 10 (E.g. User Wants Safer Place on Friday Nights & Cheaper Places on Monday Noons):

```
(SELECT RestaurantId, RestaurantName, Stars
FROM Restaurant R
WHERE R.Hours LIKE '%Friday: 11:0-23:0%' AND ((SELECT
SUM(C.Count)
FROM Occurrence O NATURAL JOIN Crime C
WHERE O.RestaurantId = R.RestaurantId AND C.Cleared = 1) /
(SELECT SUM(C.Count)
FROM Occurrence O NATURAL JOIN Crime C
WHERE O.RestaurantId = R.RestaurantId)) >= 0.1)
UNION
(SELECT RestaurantId, RestaurantName, Stars
```

```

FROM Restaurant R

WHERE R.Hours LIKE '%Monday: 11:0-17:0%' AND (SELECT

AVG(D.Price)

FROM Dish D

WHERE D.RestaurantId = R.RestaurantId) <= 10);

```

■ Screenshot for the Top 15 Rows:

```

mysql> (SELECT RestaurantId, RestaurantName, Stars
-> FROM Restaurant R
-> WHERE R.Hours LIKE '%Friday: 11:0-23:0%' AND ((SELECT SUM(C.Count)
-> FROM Occurrence O NATURAL JOIN Crime C
-> WHERE O.RestaurantId = R.RestaurantId AND C.Cleared = 1) / (SELECT SUM(C.Count)
-> FROM Occurrence O NATURAL JOIN Crime C
-> WHERE O.RestaurantId = R.RestaurantId)) >= 0.1)
-> UNION
-> (SELECT RestaurantId, RestaurantName, Stars
-> FROM Restaurant R
-> WHERE R.Hours LIKE '%Monday: 11:0-17:0%' AND (SELECT AVG(D.Price)
-> FROM Dish D
-> WHERE D.RestaurantId = R.RestaurantId) <= 10)
-> LIMIT 15;

```

RestaurantId	RestaurantName	Stars
88	Copper Vine	4.5
201	China Pearl	1.5
214	530 Pub & Grill	4.5
223	Zesty Tsunami	4
243	Lee Roy Selmon's	3.5
254	Mellow Mushroom	3.5
346	G Peppers Grill & Tavern	4
416	Tokyo Mandarin	4
677	Katie's Pizza & Pasta Osteria	4
680	Kuzina By Sofia	3
779	Rosa Mezcal	4
785	Candida's Pizza	2.5
812	HotBox Pizza	3.5
929	Akira	3
93	James Dant	5

15 rows in set (0.02 sec)

❖ Indexing:

- Find (“Popular”) Restaurants that Has At Least 3 Users that Likes Its Category & Has Above 3.5 Stars:

```

EXPLAIN ANALYZE SELECT R.RestaurantId, R.RestaurantName,
R.Category, R.Stars
FROM Restaurant R JOIN User U ON R.Category = U.Taste
WHERE R.Stars >= 3.5

```

GROUP BY R.RestaurantId

HAVING COUNT(U.UserId) >= 3

ORDER BY R.Stars DESC;

1.0 No index, cost 335435.51:

```
| -> Sort: R.Stars DESC (actual time=512.418..513.112 rows=6105 loops=1)
  -> Filter: (count(U.UserId) >= 3) (actual time=505.814..508.620 rows=6105 loops=1)
    -> Table scan on <temporary> (actual time=505.809..508.058 rows=6105 loops=1)
      -> Aggregate using temporary table (actual time=505.801..505.801 rows=6105 loops=1)
        -> Filter: (R.Category = U.Taste) (cost=35435.51 rows=11080) (actual time=1.566..74.696 rows=203447 loops=1)
          -> Inner hash join (<hash>(R.Category)=<hash>(U.Taste)) (cost=35435.51 rows=11080) (actual time=1.563..28.113 rows=203447 loops=1)
            -> Filter: (R.Stars >= 3.5) (cost=2.12 rows=332) (actual time=0.079..8.771 rows=6795 loops=1)
              -> Table scan on R (cost=2.12 rows=9974) (actual time=0.076..7.410 rows=10000 loops=1)
            -> Hash
              -> Table scan on U (cost=101.50 rows=1000) (actual time=0.071..0.376 rows=1000 loops=1)
|
```

1.1 Index on Restaurant(Category), cost 3908.37

CREATE INDEX Rest_Cat on Restaurant(Category);

```
| -> Sort: R.Stars DESC (actual time=1191.388..1192.113 rows=6105 loops=1)
  -> Filter: (count(U.UserId) >= 3) (actual time=1185.126..1187.910 rows=6105 loops=1)
    -> Table scan on <temporary> (actual time=1185.122..1187.363 rows=6105 loops=1)
      -> Aggregate using temporary table (actual time=1185.114..1185.114 rows=6105 loops=1)
        -> Nested loop inner join (cost=3908.37 rows=3625) (actual time=0.292..635.491 rows=203447 loops=1)
          -> Filter: (U.Taste is not null) (cost=101.50 rows=1000) (actual time=0.066..1.622 rows=1000 loops=1)
            -> Table scan on U (cost=101.50 rows=1000) (actual time=0.065..1.202 rows=1000 loops=1)
          -> Filter: (R.Stars >= 3.5) (cost=2.72 rows=4) (actual time=0.031..0.618 rows=203 loops=1000)
            -> Index lookup on R using Rest_Cat (Category=U.Taste) (cost=2.72 rows=11) (actual time=0.031..0.588 rows=300 loops=1000)
|
```

operations were replaced with lower-cost operations :

```
-> Filter: (R.Category = U.Taste) (cost=35435.51 rows=11080) (actual time=2.145..71.899 rows=203447 loops=1)
  -> Inner hash join (<hash>(R.Category)=<hash>(U.Taste)) (cost=35435.51 rows=11080) (actual time=2.140..25.722 rows=203447 loops=1)
```

Replaced by

```
-> Nested loop inner join (cost=3908.37 rows=3625) (actual time=0.186..538.725 rows=203447 loops=1)
```

Adding an index on the Category column is good as it dramatically reduces the cost

because Inner Hash join(cost 35435.51) is replaced by Nested Loop inner join(cost

3908.37), which costs way more less.

1.2 Index on User(Taste), cost=29476.29

Drop index Rest_Cat on Restaurant;

CREATE INDEX User_Tas on User(Taste);

```

-----+
| -> Sort: R.Stars DESC (actual time=139.447..140.100 rows=6105 loops=1)
|   -> Filter: (count(U.UserId) >= 3) (actual time=2.926..135.181 rows=6105 loops=1)
|     -> Stream results (cost=29476.29 rows=110811) (actual time=2.852..134.292 rows=6105 loops=1)
|       -> Group aggregate: count(U.UserId) (cost=29476.29 rows=110811) (actual time=2.847..131.658 rows=6105 loops=1)
|         -> Nested loop inner join (cost=18395.18 rows=110811) (actual time=2.817..115.807 rows=203447 loops=1)
|           -> Filter: ((R.Stars >= 3.5) and (R.Category is not null)) (cost=1037.65 rows=3324) (actual time=0.081..6.652 rows=6795 loops=1)
|             -> Index scan on R using PRIMARY (cost=1037.65 rows=9974) (actual time=0.078..4.830 rows=10000 loops=1)
|             -> Covering index lookup on U using User_Tas (Taste=R.Category) (cost=1.89 rows=33) (actual time=0.010..0.014 rows=30 loops=6795)
|
|
-----+
1 row in set (0.14 sec)

```

operations were replaced with lower-cost operations :

```

-> Inner hash join (<hash>(R.Category)=<hash>(U.Taste)) (cost=35435.51 rows=11080) (actual time=1.563..28.113 rows=203447 loops=1)
  -> Filter: (R.Stars >= 3.5) (cost=2.12 rows=332) (actual time=0.079..8.771 rows=6795 loops=1)
    -> Table scan on R (cost=2.12 rows=9974) (actual time=0.076..7.410 rows=10000 loops=1)
  -> Hash
    -> Table scan on U (cost=101.50 rows=1000) (actual time=0.071..0.376 rows=1000 loops=1)

```

Is replaced by

```

-> Nested loop inner join (cost=18395.18 rows=110811) (actual time=2.817..115.807 rows=203447 loops=1)
  -> Filter: ((R.Stars >= 3.5) and (R.Category is not null)) (cost=1037.65 rows=3324) (actual time=0.081..6.652 rows=6795 loops=1)
    -> Index scan on R using PRIMARY (cost=1037.65 rows=9974) (actual time=0.078..4.830 rows=10000 loops=1)
    -> Covering index lookup on U using User_Tas (Taste=R.Category) (cost=1.89 rows=33) (actual time=0.010..0.014 rows=30 loops=6795)

```

Introducing an index on User(Taste) is good as it reduces the total cost to 29476.29. Because the Inner Hash join(cost 35435.51) is replaced by Nested Loop inner join(cost 18395.18), which costs way more less. Also the Hash Table scan(cost=101.50) is replaced on U is replaced by index lookup on U(cost=1.89).

1.3 Index on Restaurant(Category) and User(Taste), cost 3908.37

Drop index User_Tas on User;

CREATE INDEX Rest_Cat on Restaurant(Category);

CREATE INDEX User_Tas on User(Taste);

```

-----+
| -> Sort: R.Stars DESC (actual time=964.839..965.469 rows=6105 loops=1)
|   -> Filter: (count(U.UserId) >= 3) (actual time=958.355..961.272 rows=6105 loops=1)
|     -> Table scan on <temporary> (actual time=958.350..960.720 rows=6105 loops=1)
|       -> Aggregate using temporary table (actual time=958.339..958.339 rows=6105 loops=1)
|         -> Nested loop inner join (cost=3908.37 rows=3625) (actual time=0.138..478.101 rows=203447 loops=1)
|           -> Filter: (U.Taste is not null) (cost=101.50 rows=1000) (actual time=0.062..1.635 rows=1000 loops=1)
|             -> Covering index scan on U using User_Tas (cost=101.50 rows=1000) (actual time=0.061..1.116 rows=1000 loops=1)
|           -> Filter: (R.Stars >= 3.5) (cost=2.72 rows=4) (actual time=0.019..0.460 rows=203 loops=1000)
|             -> Index lookup on R using Rest_Cat (Category=U.Taste) (cost=2.72 rows=11) (actual time=0.018..0.431 rows=300 loops=1000)
|
|
-----+
1 row in set (0.97 sec)

```

operations were replaced with lower-cost operations :

```
=> Filter: (R.Category = U.Taste) (cost=35435.51 rows=11080) (actual time=2.145..71.899 rows=203447 loops=1)
-> Inner hash join (<hash>(R.Category)=<hash>(U.Taste)) (cost=35435.51 rows=11080) (actual time=2.140..25.722 rows=203447 loops=1)
```

Replaced by

```
-> Nested loop inner join (cost=3908.37 rows=3625) (actual time=0.139..449.761 rows=203447 loops=1)
```

Adding an index on both the Restaurant(Category) and User(Taste) column is good as it brings the total cost back down to 3908.37. The Inner Hash join(cost 35435.51) is replaced by Nested Loop inner join(cost 3908.37), which costs way more less.(Though it is same as with only the Restaurant(Category) index. This demonstrates that while both indexes contribute to query optimization, but the Restaurant(Category) index is the primary driver of the cost reduction in this query, maybe because of the size of User is small(only 1000 rows).)

➤ Find Restaurants In the State of Illinois (IL) Where the Crimes that Happen

Around the Restaurant <= 100 & Stars >= 3:

```
EXPLAIN ANALYZE SELECT RestaurantId, RestaurantName, Category,
Stars
```

```
FROM Restaurant R NATURAL JOIN Location L1
```

```
WHERE R.Stars >= 3 AND L1.State = 'IL' AND (SELECT
SUM(C.Count)
```

```
FROM Location L2 NATURAL JOIN Crime C
```

```
WHERE L2.LocationId = L1.LocationId) <= 100
```

```
ORDER BY Stars DESC, RestaurantName ASC;
```

2.0: No indexing: cost 450.24

```
| -> Sort: R.Stars DESC, R.RestaurantName (actual time=1.128..1.130 rows=19 loops=1)
    -> Stream results (cost=450.24 rows=332) (actual time=0.235..1.102 rows=19 loops=1)
        -> Nested loop inner join (cost=450.24 rows=332) (actual time=0.230..1.091 rows=19 loops=1)
            -> Filter: ((L1.State = 'IL') and ((select #2) <= 100)) (cost=101.50 rows=100) (actual time=0.148..0.945 rows=2 loops=1)
                -> Table scan on L1 (cost=101.50 rows=1000) (actual time=0.058..0.343 rows=1000 loops=1)
                    -> Select #2 (subquery in condition; dependent)
                        -> Aggregate: sum(C.Count) (cost=4.79 rows=1) (actual time=0.033..0.033 rows=1 loops=14)
                            -> Nested loop inner join (cost=3.80 rows=10) (actual time=0.028..0.032 rows=9 loops=14)
                                -> Single-row covering index lookup on L2 using PRIMARY (LocationId=L1.LocationId) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=14)
                                    -> Index lookup on C using LocationId (LocationId=L1.LocationId), with index condition: (C.LocationId = L2.LocationId) (cost=3.45 rows=10) (actual time=0.025..0.028 rows=9 loops=14)
                                -> Filter: (R.Stars >= 3) (cost=2.49 rows=3) (actual time=0.065..0.071 rows=10 loops=2)
                                    -> Index lookup on R using LocationId (LocationId=L1.LocationId) (cost=2.49 rows=10) (actual time=0.064..0.069 rows=12 loops=2)
```

2.1: Indexing on Restaurant(Stars): cost 450.24

Create: CREATE INDEX Stars_idx ON Restaurant(Stars);

```
| -> Sort: R.Stars DESC, R.RestaurantName (actual time=1.060..1.062 rows=19 loops=1)
    -> Stream results (cost=450.24 rows=797) (actual time=0.185..1.036 rows=19 loops=1)
        -> Nested loop inner join (cost=450.24 rows=797) (actual time=0.184..1.025 rows=19 loops=1)
            -> Filter: ((L1.State = 'IL') and ((select #2) <= 100)) (cost=101.50 rows=100) (actual time=0.135..0.916 rows=2 loops=1)
                -> Table scan on L1 (cost=101.50 rows=1000) (actual time=0.055..0.309 rows=1000 loops=1)
                    -> Select #2 (subquery in condition; dependent)
                        -> Aggregate: sum(C.Count) (cost=4.79 rows=1) (actual time=0.033..0.033 rows=1 loops=14)
                            -> Nested loop inner join (cost=3.80 rows=10) (actual time=0.029..0.032 rows=9 loops=14)
                                -> Single-row covering index lookup on L2 using PRIMARY (LocationId=L1.LocationId) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=14)
                                    -> Index lookup on C using LocationId (LocationId=L1.LocationId), with index condition: (C.LocationId = L2.LocationId) (cost=3.45 rows=10) (actual time=0.026..0.028 rows=9 loops=14)
                                -> Filter: (R.Stars >= 3) (cost=2.50 rows=8) (actual time=0.048..0.053 rows=10 loops=2)
                                    -> Index lookup on R using LocationId (LocationId=L1.LocationId) (cost=2.50 rows=10) (actual time=0.047..0.051 rows=12 loops=2)
```

where

```
-> Stream results (cost=450.24 rows=332) (actual time=0.720..1.503 rows=21 loops=1)
    -> Nested loop inner join (cost=450.24 rows=332) (actual time=0.716..1.482 rows=21 loops=1)
        -> Filter: ((L1.State = 'IL') and ((select #2) <= 100)) (cost=101.50 rows=100) (actual time=0.650..1.350 rows=2 loops=1)
```

is replaced by

```
-> Stream results (cost=450.24 rows=797) (actual time=0.448..1.038 rows=21 loops=1)
    -> Nested loop inner join (cost=450.24 rows=797) (actual time=0.444..1.022 rows=21 loops=1)
        -> Filter: ((L1.State = 'IL') and ((select #2) <= 100)) (cost=101.50 rows=100) (actual time=0.362..0.857 rows=2 loops=1)
```

The intuition of adding an index on variable Stars of Restaurants is based on the filter Restaurant R.Stars >= 3 in the where clause. However, the result turns out to be a total surprise as this indexing does not help to make any improvement in total. More specifically, we observed the number of rows indeed increases, from 332 when there is no indexing to 797 after adding index to Stars. We suspect that R.Stars >= 3 is a really loose constraint and data with R.Stars >= 3 is a wide range of candidates. In contrast, L.State = 'IL' probably is more constraint and gives back a fewer number of candidates so that indexing on the state will have better performance.

Drop: DROP INDEX Stars_idx ON Restaurant;

2.2: Index on Location(State): cost 50.50

Create: CREATE INDEX State_idx ON Location(State);

```
| -> Sort: R.Stars DESC, R.RestaurantName (actual time=0.584..0.586 rows=19 loops=1)
|   -> Stream results (cost=50.50 rows=46) (actual time=0.133..0.563 rows=19 loops=1)
|     -> Nested loop inner join (cost=50.50 rows=46) (actual time=0.130..0.554 rows=19 loops=1)
|       -> Filter: ((select #2) <= 100) (cost=1.68 rows=14) (actual time=0.076..0.444 rows=2 loops=1)
|         -> Covering index lookup on L1 using State_idx (State='IL') (cost=1.68 rows=14) (actual time=0.016..0.019 rows=14 loops=1)
|       -> Select #2 (subquery in condition; dependent)
|         -> Aggregate: sum(C.Count) (cost=4.79 rows=1) (actual time=0.029..0.029 rows=1 loops=14)
|           -> Nested loop inner join (cost=3.80 rows=10) (actual time=0.024..0.028 rows=9 loops=14)
|             -> Single-row covering index lookup on L2 using PRIMARY (LocationId=L1.LocationId) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=14)
|             -> Index lookup on C using LocationId (LocationId=L1.LocationId), with index condition: (C.LocationId = L2.LocationId) (cost=3.45 rows=10) (actual time=0.021..0.024 rows=9 loops=14)
|           -> Filter: (R.Stars >= 3) (cost=2.51 rows=3) (actual time=0.048..0.053 rows=10 loops=2)
|             -> Index lookup on R using LocationId (LocationId=L1.LocationId) (cost=2.51 rows=10) (actual time=0.047..0.051 rows=12 loops=2)
```

where (original)

```
-> Stream results (cost=450.24 rows=332) (actual time=0.450..1.036 rows=21 loops=1)
-> Nested loop inner join (cost=450.24 rows=332) (actual time=0.446..1.025 rows=21 loops=1)
-> Filter: ((L1.State = 'IL') and ((select #2) <= 100)) (cost=101.50 rows=100) (actual time=0.383..0.896 rows=2 loops=1)
-> Table scan on L1 (cost=101.50 rows=1000) (actual time=0.037..0.292 rows=1000 loops=1)
```

is replaced by

```
-> Stream results (cost=50.50 rows=46) (actual time=0.335..0.624 rows=21 loops=1)
-> Nested loop inner join (cost=50.50 rows=46) (actual time=0.332..0.614 rows=21 loops=1)
-> Filter: ((select #2) <= 100) (cost=1.68 rows=14) (actual time=0.278..0.489 rows=2 loops=1)
-> Covering index lookup on L1 using State_idx (State='IL') (cost=1.68 rows=14) (actual time=0.017..0.020 rows=14 loops=1)
```

(The “Table scan on L1” is replaced by “Covering index lookup on L1”)

We can see the result of indexing on variable States of the Location does indeed demonstrate the previous assumption is correct. Compared to indexing on Restaurant Stars, it significantly brings down the cost from 450.34 to 50.50 and the number of rows from 332 to 46. This result is also consistent with common sense since the number of restaurants in Illinois should be much fewer than the number of restaurants that have a star greater than 3 all across the United States. Thus, indexing on the “State” variable will have better performance.

Drop: DROP INDEX State_idx ON Location;

2.3: Indexing on both Restaurant(Stars) and Location(State): cost 50.50

Create:

CREATE INDEX Stars_idx ON Restaurant(Stars);

CREATE INDEX State_idx ON Location(State);

```
| -> Sort: R.Stars DESC, R.RestaurantName (actual time=0.646..0.647 rows=19 loops=1)
    -> Stream results (cost=50.50 rows=112) (actual time=0.168..0.621 rows=19 loops=1)
        -> Nested loop inner join (cost=50.50 rows=112) (actual time=0.165..0.611 rows=19 loops=1)
            -> Filter: ((select #2) <= 100) (cost=1.68 rows=14) (actual time=0.105..0.490 rows=2 loops=1)
                -> Covering index lookup on L1 using State_idx (State='IL') (cost=1.68 rows=14) (actual time=0.032..0.035 rows=14 loops=1)
                    -> Select #2 (subquery in condition/ dependent)
                        -> Aggregate: sum(C.Count) (cost=4.79 rows=1) (actual time=0.031..0.031 rows=1 loops=14)
                            -> Nested loop inner join (cost=3.80 rows=10) (actual time=0.026..0.029 rows=9 loops=14)
                                -> Single-row covering index lookup on L2 using PRIMARY (LocationId=L1.LocationId) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=14)
                                    -> Index lookup on C using LocationId (LocationId=L1.LocationId), with index condition: (C.LocationId = L2.LocationId) (cost=3.45 rows=10) (actual time=0.023..0.026 rows=9 loops=14)
                                -> Filter: (R.Stars >= 3) (cost=2.55 rows=8) (actual time=0.053..0.059 rows=10 loops=2)
                                    -> Index lookup on R using LocationId (LocationId=L1.LocationId) (cost=2.55 rows=10) (actual time=0.053..0.057 rows=12 loops=2)
                                    |
```

where (original)

```
-> Stream results (cost=450.24 rows=332) (actual time=0.450..1.036 rows=21 loops=1)
    -> Nested loop inner join (cost=450.24 rows=332) (actual time=0.446..1.025 rows=21 loops=1)
        -> Filter: ((L1.State = 'IL') and ((select #2) <= 100)) (cost=101.50 rows=100) (actual time=0.383..0.896 rows=2 loops=1)
            -> Table scan on L1 (cost=101.50 rows=1000) (actual time=0.037..0.292 rows=1000 loops=1)
```

is replaced by

```
-> Stream results (cost=50.50 rows=112) (actual time=0.396..0.671 rows=21 loops=1)
    -> Nested loop inner join (cost=50.50 rows=112) (actual time=0.391..0.659 rows=21 loops=1)
        -> Filter: ((select #2) <= 100) (cost=1.68 rows=14) (actual time=0.327..0.541 rows=2 loops=1)
            -> Covering index lookup on L1 using State_idx (State='IL') (cost=1.68 rows=14) (actual time=0.032..0.036 rows=14 loops=1)
```

Again, this indexing schema also proves the assumption that the location in the where clause takes the lead. The total cost after indexing on both Stars and State is exactly the same as only indexing on State. Additionally, we can see the row number next to the cost has increased compared to only indexing on states (112 vs 46). As a result, indexing only on States is the best option for this advanced query with this particular input. It is also worth mentioning that there should be no other indexing candidates that can potentially improve the performance of this query: both of the natural join is joining on LocationId and so does the nested where, but LocationId has already been an index since it is one of the primary keys.

Drop:

DROP INDEX Stars_idx ON Restaurant;

DROP INDEX State_idx ON Location;

- Find Locations that Has At Least 5 Restaurants Around It With Average Stars of Those Restaurants Being At Least 3 & Having Uncleared Crimes of Type 4, 5 and 6 Under 50 (Note: Crime Type Are Represented As Integers Where These Integers Will Each Map to a Specific Type (E.g. Type 1 = Robbery)):

```
Explain analyze SELECT State, City, PostalCode,  
COUNT(R.RestaurantId) AS NumRestaurants, AVG(R.Stars) AS  
AverageStars  
FROM Location L NATURAL JOIN Restaurant R NATURAL JOIN  
Crime C  
WHERE C.Type IN (4, 5, 6) AND C.Cleared = 0  
GROUP BY L.LocationId  
HAVING COUNT(R.RestaurantId) >= 5 AND AVG(R.Stars) >= 3 AND  
SUM(C.Count) <= 50  
ORDER BY AverageStars DESC, NumRestaurants DESC;
```

3.0 Default index : cost=2122.51

```
| -> Sort: AverageStars DESC, NumRestaurants DESC (actual time=118.400..118.416 rows=45 loops=1)  
-> Filter: ((count(R.RestaurantId) >= 5) and (avg(R.Stars) >= 3) and (sum(C.Count) <= 50)) (actual time=117.845..118.352 rows=45 loops=1)  
-> Table scan on <temporary> (actual time=117.830..118.195 rows=862 loops=1)  
-> Aggregate using temporary table (actual time=117.827..117.827 rows=862 loops=1)  
-> Nested loop inner join (cost=2122.51 rows=2940) (actual time=13.658..95.745 rows=21254 loops=1)  
-> Nested loop inner join (cost=1093.66 rows=295) (actual time=0.081..12.210 rows=2134 loops=1)  
-> Filter: ((C.Cleared = 0) and (C.'Type' in (4,5,6)) and (C.LocationId is not null)) (cost=990.40 rows=295) (actual time=0.064..4.956 rows=2134 loops=1)  
-> Table scan on C (cost=990.40 rows=9834) (actual time=0.060..3.598 rows=10000 loops=1)  
-> Single-row index lookup on L using PRIMARY (LocationId=C.LocationId) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=2134)  
-> Index lookup on R using LocationId (LocationId=C.LocationId) (cost=2.49 rows=10) (actual time=0.035..0.038 rows=10 loops=2134)  
|
```

3.1 Indexing on Crime(Type)

CREATE INDEX Crime_Type on Crime(Type); cost=2638.19

```
| -> Sort: AverageStars DESC, NumRestaurants DESC (actual time=76.760..76.766 rows=45 loops=1)  
-> Filter: ((count(R.RestaurantId) >= 5) and (avg(R.Stars) >= 3) and (sum(C.Count) <= 50)) (actual time=76.114..76.718 rows=45 loops=1)  
-> Table scan on <temporary> (actual time=76.092..76.554 rows=862 loops=1)  
-> Aggregate using temporary table (actual time=76.089..76.089 rows=862 loops=1)  
-> Nested loop inner join (cost=2638.19 rows=4279) (actual time=0.146..53.449 rows=21254 loops=1)  
-> Nested loop inner join (cost=1140.69 rows=429) (actual time=0.065..9.098 rows=2134 loops=1)  
-> Filter: ((C.Cleared = 0) and (C.'Type' in (4,5,6)) and (C.LocationId is not null)) (cost=990.40 rows=429) (actual time=0.053..5.094 rows=2134 loops=1)  
-> Table scan on C (cost=990.40 rows=9834) (actual time=0.048..3.794 rows=10000 loops=1)  
-> Single-row index lookup on L using PRIMARY (LocationId=C.LocationId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2134)  
-> Index lookup on R using LocationId (LocationId=C.LocationId) (cost=2.49 rows=10) (actual time=0.017..0.020 rows=10 loops=2134)  
|
```

Adding an index on the Type column in the Crime table is not useful as the total cost increased to 2638.19.

```
-> Nested loop inner join (cost=2122.51 rows=2940) (actual time=13.658..95.745 rows=21254 loops=1)
```

Is replaced by

```
-> Nested loop inner join (cost=2638.19 rows=4279) (actual time=0.146..53.449 rows=21254 loops=1)
```

It might initially seem to improve performance because it directly supports the WHERE C.Type IN (4, 5, 6) condition. However, the index didn't lead to the expected performance gains. The cost increase could be due to the overhead of maintaining the index during the query execution.

3.2 Indexing on Crime(Cleared) and Crime(Type)

```
CREATE INDEX Crime_Type on Crime(Type);
```

```
CREATE INDEX Crime_Clear on Crime(Cleared); cost: 8423.05
```

```
| -> Sort: AverageStars DESC, NumRestaurants DESC (actual time=69.319..69.325 rows=45 loops=1)
|   -> Filter: ((count(R.RestaurantId) >= 5) and (avg(R.Stars) >= 3) and (sum(C.Count) <= 50)) (actual time=68.702..69.279 rows=45 loops=1)
|   -> Table scan on <temporary> (actual time=68.686..69.108 rows=862 loops=1)
|   -> Aggregate using temporary table (actual time=68.683..68.683 rows=862 loops=1)
|   -> Nested loop inner join (cost=8423.05 rows=21262) (actual time=0.290..47.953 rows=21254 loops=1)
|     -> Nested loop inner join (cost=981.26 rows=2134) (actual time=0.256..10.665 rows=2134 loops=1)
|       -> Filter: ((C.`Type` in (4,5,6)) and (C.LocationId is not null)) (cost=234.39 rows=2134) (actual time=0.245..6.948 rows=2134 loops=1)
|       -> Index lookup on C using Crime_Clear (Cleared=0) (cost=234.39 rows=4887) (actual time=0.240..6.274 rows=4887 loops=1)
|       -> Single-row index lookup on L using PRIMARY (LocationId=C.LocationId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2134)
|       -> Index lookup on R using LocationId (LocationId=C.LocationId) (cost=2.49 rows=10) (actual time=0.014..0.016 rows=10 loops=2134)
|
```

Adding an index on both Type and Clear column in the Crime table is not useful as the total cost increased to 8423.05.

Although

```
-> Filter: ((C.Cleared = 0) and (C.`Type` in (4,5,6)) and (C.LocationId is not null)) (cost=990.40 rows=295) (actual time=0.064..4.956 rows=2134 loops=1)
-> Table scan on C (cost=990.40 rows=9834) (actual time=0.060..3.598 rows=10000 loops=1)
```

Is replaced by

```
-> Filter: ((C.`Type` in (4,5,6)) and (C.LocationId is not null)) (cost=234.39 rows=2134) (actual time=0.245..6.948 rows=2134 loops=1)
-> Index lookup on C using Crime_Clear (Cleared=0) (cost=234.39 rows=4887) (actual time=0.240..6.274 rows=4887 loops=1)
```

Although after indexing The “index lookup” cost less than “Table scan on C”, the filter also cost less,

```
-> Nested loop inner join (cost=2122.51 rows=2940) (actual time=13.658..95.745 rows=21254 loops=1)
```

Is replaced by

```
-> Nested loop inner join (cost=8423.05 rows=21262) (actual time=0.290..47.953 rows=21254 loops=1)
```

which increases the cost badly. The cost increase could be due to the overhead of maintaining the index during the query execution.

3.3 Indexing on Crime(Cleared)

CREATE INDEX Crime_Clear on Crime(Cleared); cost=5793.64

```
| -> Sort: AverageStars DESC, NumRestaurants DESC (actual time=118.400..118.416 rows=45 loops=1)
-> Filter: ((count(R.RestaurantId) >= 5) and (avg(R.Stars) >= 3) and (sum(C.Count) <= 50)) (actual time=117.845..118.352 rows=45 loops=1)
-> Table scan on <temporary> (actual time=117.830..118.195 rows=862 loops=1)
-> Aggregate using temporary table (actual time=117.827..117.827 rows=862 loops=1)
-> Nested loop inner join (cost=2122.51 rows=2940) (actual time=13.658..95.745 rows=21254 loops=1)
-> Nested loop inner join (cost=1093.66 rows=295) (actual time=0.081..12.210 rows=2134 loops=1)
-> Filter: ((C.Cleared = 0) and (C.Type in (4,5,6)) and (C.LocationId is not null)) (cost=990.40 rows=295) (actual time=0.064..4.956 rows=2134 loops=1)
-> Table scan on C (cost=990.40 rows=9834) (actual time=0.060..3.598 rows=10000 loops=1)
-> Single-row index lookup on L using PRIMARY (LocationId=C.LocationId) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=2134)
-> Index lookup on R using LocationId (LocationId=C.LocationId) (cost=2.49 rows=10) (actual time=0.035..0.038 rows=10 loops=2134)
|
```

```
| -> Sort: AverageStars DESC, NumRestaurants DESC (actual time=67.443..67.449 rows=45 loops=1)
-> Filter: ((count(R.RestaurantId) >= 5) and (avg(R.Stars) >= 3) and (sum(C.Count) <= 50)) (actual time=66.836..67.404 rows=45 loops=1)
-> Table scan on <temporary> (actual time=66.822..67.192 rows=862 loops=1)
-> Aggregate using temporary table (actual time=66.818..66.819 rows=862 loops=1)
-> Nested loop inner join (cost=5793.64 rows=14608) (actual time=0.289..46.532 rows=21254 loops=1)
-> Nested loop inner join (cost=680.75 rows=1466) (actual time=0.248..10.519 rows=2134 loops=1)
-> Filter: ((C.Type in (4,5,6)) and (C.LocationId is not null)) (cost=167.61 rows=1466) (actual time=0.237..6.963 rows=2134 loops=1)
-> Index lookup on C using Crime Clear (Cleared=0) (cost=167.61 rows=4887) (actual time=0.232..6.340 rows=4887 loops=1)
-> Single-row index lookup on L using PRIMARY (LocationId=C.LocationId) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=2134)
-> Index lookup on R using LocationId (LocationId=C.LocationId) (cost=2.49 rows=10) (actual time=0.013..0.016 rows=10 loops=2134)
|
```

Adding an index on both Type and Clear columns in the Crime table is not useful as the total cost increased to 5793.64.

Although

```
-> Nested loop inner join (cost=1093.66 rows=295) (actual time=0.081..12.210 rows=2134 loops=1)
-> Filter: ((C.Cleared = 0) and (C.Type in (4,5,6)) and (C.LocationId is not null)) (cost=990.40 rows=295) (actual time=0.064..4.956 rows=2134 loops=1)
-> Table scan on C (cost=990.40 rows=9834) (actual time=0.060..3.598 rows=10000 loops=1)
```

Is replaced by

```
-> Nested loop inner join (cost=680.75 rows=1466) (actual time=0.248..10.519 rows=2134 loops=1)
-> Filter: ((C.Type in (4,5,6)) and (C.LocationId is not null)) (cost=167.61 rows=1466) (actual time=0.237..6.963 rows=2134 loops=1)
-> Index lookup on C using Crime Clear (Cleared=0) (cost=167.61 rows=4887) (actual time=0.232..6.340 rows=4887 loops=1)
```

Although after indexing The “index lookup” cost less than “Table scan on C”, the cost of filter and Nested loop inner join also decreased,

```
-> Nested loop inner join (cost=2122.51 rows=2940) (actual time=13.658..95.745 rows=21254 loops=1)
```

Is replaced by

```
-> Nested loop inner join (cost=5793.64 rows=14608) (actual time=0.289..46.532 rows=21254 loops=1)
```

which increases the cost badly. The cost increase could be due to the overhead of maintaining the index during the query execution.

- Find Restaurants that Open on Friday From 11:00-23:00 Where the “Cleared Crime Rate” (I.e. Cleared Crimes / Total Number of Crimes) ≥ 0.1 and Restaurants that Open on Monday From 11:00-17:00 Where the Average Dish Price ≤ 10 (E.g. User Wants Safer Place on Friday Nights & Cheaper Places on Monday Noons):

```
EXPLAIN ANALYZE (SELECT RestaurantId, RestaurantName, Stars
FROM Restaurant R
WHERE R.Hours LIKE '%Friday: 11:0-23:0%' AND ((SELECT
SUM(C.Count)
FROM Occurrence O NATURAL JOIN Crime C
WHERE O.RestaurantId = R.RestaurantId AND C.Cleared = 1) /
(SELECT SUM(C.Count)
FROM Occurrence O NATURAL JOIN Crime C
WHERE O.RestaurantId = R.RestaurantId))  $\geq 0.1$ )
```

UNION

(SELECT RestaurantId, RestaurantName, Stars

FROM Restaurant R

WHERE R.Hours LIKE '%Monday: 11:0-17:0%' AND (SELECT
AVG(D.Price)

FROM Dish D

WHERE D.RestaurantId = R.RestaurantId) <= 10);

4.0: No indexing: cost 2296.94..2327.12

```
| -> Table scan on <Union temporary> (cost=2296.94..2327.12 rows=2216) (actual time=42.096..42.101 rows=27 loops=1)
    -> Union materialize with deduplication (cost=2296.92..2296.92 rows=2216) (actual time=42.093..42.093 rows=27 loops=1)
        -> Filter: ((R.Hours like '%Friday: 11:0-23:0%') and (((select #2) / (select #3)) >= 0.1)) (cost=1037.65 rows=1108) (actual time=0.444..20.874 rows=14 loops=1)
            -> Table scan on R (cost=1037.65 rows=9974) (actual time=0.076..3.982 rows=10000 loops=1)
                -> Select #2 (subquery in condition; dependent)
                    -> Aggregate: sum(C.Count) (cost=4.85 rows=1) (actual time=0.006..0.006 rows=1 loops=167)
                        -> Nested loop inner join (cost=4.75 rows=1) (actual time=0.004..0.006 rows=0 loops=167)
                            -> Covering index lookup on O using PRIMARY (RestaurantId=R.RestaurantId) (cost=1.25 rows=10) (actual time=0.003..0.003 rows=1 loops=167)
                            -> Filter: (C.Cleared = 1) (cost=0.25 rows=0.1) (actual time=0.002..0.002 rows=0 loops=151)
                                -> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=151)
                    -> Select #3 (subquery in condition; dependent)
                        -> Aggregate: sum(C.Count) (cost=5.75 rows=1) (actual time=0.016..0.016 rows=1 loops=15)
                            -> Nested loop inner join (cost=4.75 rows=10) (actual time=0.005..0.015 rows=10 loops=15)
                                -> Covering index lookup on O using PRIMARY (RestaurantId=R.RestaurantId) (cost=1.25 rows=10) (actual time=0.004..0.006 rows=10 loops=15)
                                -> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=151)
        -> Filter: ((R.Hours like '%Monday: 11:0-17:0%') and ((select #5) <= 10)) (cost=1037.65 rows=1108) (actual time=0.277..21.053 rows=13 loops=1)
            -> Table scan on R (cost=1037.65 rows=9974) (actual time=0.049..4.096 rows=10000 loops=1)
                -> Select #5 (subquery in condition; dependent)
                    -> Aggregate: avg(D.Price) (cost=2.55 rows=1) (actual time=0.047..0.047 rows=1 loops=27)
                        -> Index lookup on D using RestaurantId (RestaurantId=R.RestaurantId) (cost=2.29 rows=3) (actual time=0.045..0.046 rows=2 loops=27)
```

4.1: Indexing on Restaurant(Hours): cost 2296.94..2327.12

Create: CREATE INDEX Hours_idx ON Restaurant(Hours);

```
| -> Table scan on <Union temporary> (cost=2296.94..2327.12 rows=2216) (actual time=44.887..44.892 rows=27 loops=1)
    -> Union materialize with deduplication (cost=2296.92..2296.92 rows=2216) (actual time=44.883..44.883 rows=27 loops=1)
        -> Filter: ((R.Hours like '%Friday: 11:0-23:0%') and (((select #2) / (select #3)) >= 0.1)) (cost=1037.65 rows=1108) (actual time=0.665..24.895 rows=14 loops=1)
            -> Table scan on R (cost=1037.65 rows=9974) (actual time=0.362..5.203 rows=10000 loops=1)
                -> Select #2 (subquery in condition; dependent)
                    -> Aggregate: sum(C.Count) (cost=4.85 rows=1) (actual time=0.008..0.008 rows=1 loops=167)
                        -> Nested loop inner join (cost=4.75 rows=1) (actual time=0.006..0.008 rows=0 loops=167)
                            -> Covering index lookup on O using PRIMARY (RestaurantId=R.RestaurantId) (cost=1.25 rows=10) (actual time=0.005..0.005 rows=1 loops=167)
                            -> Filter: (C.Cleared = 1) (cost=0.25 rows=0.1) (actual time=0.002..0.002 rows=0 loops=151)
                                -> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=151)
                    -> Select #3 (subquery in condition; dependent)
                        -> Aggregate: sum(C.Count) (cost=5.75 rows=1) (actual time=0.016..0.016 rows=1 loops=15)
                            -> Nested loop inner join (cost=4.75 rows=10) (actual time=0.005..0.015 rows=10 loops=15)
                                -> Covering index lookup on O using PRIMARY (RestaurantId=R.RestaurantId) (cost=1.25 rows=10) (actual time=0.004..0.005 rows=10 loops=15)
                                -> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=151)
        -> Filter: ((R.Hours like '%Monday: 11:0-17:0%') and ((select #5) <= 10)) (cost=1037.65 rows=1108) (actual time=0.271..19.919 rows=13 loops=1)
            -> Table scan on R (cost=1037.65 rows=9974) (actual time=0.046..3.952 rows=10000 loops=1)
                -> Select #5 (subquery in condition; dependent)
                    -> Aggregate: avg(D.Price) (cost=2.55 rows=1) (actual time=0.015..0.015 rows=1 loops=27)
                        -> Index lookup on D using RestaurantId (RestaurantId=R.RestaurantId) (cost=2.29 rows=3) (actual time=0.013..0.014 rows=2 loops=27)
```

(No change)

The intuition of adding an index on Restaurant Hours is based on there are two where clause

involves Restaurant Hours. Therefore, it is a total surprise that it turns out the cost and rows are

staying exactly the same before and after adding this index. We suspect that indexing does not really help when doing some pattern matching of strings in the where clause. This is because we observe that in the last block, “Filter: ((R.Hours like ‘%Monday: 11:0-17:0%’)) and ...” is totally the same before and after adding an index but we do expect for some change after adding a new index on Restaurant(Hours).

Drop: DROP INDEX Hours_idx ON Restaurant;

4.2: Indexing on Crime(Cleared): cost 2296.94..2327.12

Create: CREATE INDEX Cleared_idx ON Crime(Cleared);

```
| -> Table scan on Union temporary? (cost=2296.94..2327.12 rows=2216) (actual time=43.618..43.623 rows=27 loops=1)
|   -> Union materialize with deduplication (cost=2296.92..2296.92 rows=2216) (actual time=43.614..43.614 rows=27 loops=1)
|     -> Filter: ((R.Hours like 'Friday: 11:0-23:0%') and ((select #2) / (select #3)) >= 0.1)) (cost=1037.65 rows=1108) (actual time=0.302..21.770 rows=14 loops=1)
|       -> Table scan on R (cost=1037.65 rows=9974) (actual time=0.047..4.180 rows=10000 loops=1)
|         -> Select #2 (subquery in condition; dependent)
|           -> Aggregate: sum(C.Count) (cost=5.27 rows=1) (actual time=0.006..0.006 rows=1 loops=167)
|             -> Nested loop inner join (cost=4.75 rows=5) (actual time=0.004..0.005 rows=0 loops=167)
|               -> Covering index lookup on O using PRIMARY (RestaurantId=R.RestaurantId) (cost=1.25 rows=10) (actual time=0.003..0.004 rows=1 loops=167)
|               -> Filter: (C.Cleared = 1) (cost=0.26 rows=1) (actual time=0.002..0.002 rows=0 loops=151)
|                 -> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=151)
|           -> Select #3 (subquery in condition; dependent)
|             -> Aggregate: sum(C.Count) (cost=5.75 rows=1) (actual time=0.017..0.017 rows=1 loops=15)
|               -> Nested loop inner join (cost=4.75 rows=10) (actual time=0.005..0.016 rows=10 loops=15)
|                 -> Covering index lookup on O using PRIMARY (RestaurantId=R.RestaurantId) (cost=1.25 rows=10) (actual time=0.004..0.006 rows=10 loops=15)
|                 -> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=151)
|           -> Filter: ((R.Hours like 'Monday: 11:0-17:0%') and ((select #5) <= 10)) (cost=1037.65 rows=1108) (actual time=0.268..21.760 rows=13 loops=1)
|             -> Table scan on R (cost=1037.65 rows=9974) (actual time=0.047..4.194 rows=10000 loops=1)
|               -> Select #5 (subquery in condition; dependent)
|                 -> Aggregate: avg(D.Price) (cost=2.55 rows=1) (actual time=0.018..0.018 rows=1 loops=27)
|                   -> Index lookup on D using RestaurantId (RestaurantId=R.RestaurantId) (cost=2.29 rows=3) (actual time=0.016..0.017 rows=2 loops=27)
```

where

```
-> Filter: (C.Cleared = 1) (cost=0.25 rows=0.1) (actual time=0.002..0.002 rows=0 loops=151)
-> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=151)
```

is replaced by

```
-> Filter: (C.Cleared = 1) (cost=0.26 rows=1) (actual time=0.002..0.002 rows=0 loops=151)
-> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.26 rows=1) (actual time=0.002..0.002 rows=1 loops=151)
```

The intuition of adding an index on Crime Cleared is based on the constraint “C.Cleared = 1” in the where clause. However, it seems that the total cost does not change and there is even a decrease in performance when Filtering out “C.Cleared = 1”. As observed when filtering out C.Cleared only uses primary key, we propose using the primary key instead the newly created index is more efficient and in turn, we should rather not add this index.

Drop: DROP INDEX Cleared_idx ON Crime;

4.3: Indexing on RestaurantName: Cost 2296.94..2327.12

Create: CREATE INDEX RestaurantName_idx ON Restaurant(RestaurantName);

```
| -> Table scan on <union temporary> (cost=2296.94..2327.12 rows=2216) (actual time=47.431..47.436 rows=27 loops=1)
  -> Union materialize with deduplication (cost=2296.92..2296.92 rows=2216) (actual time=47.427..47.427 rows=27 loops=1)
    -> Filter: ((R.Hours like '%Friday: 11:0-23:0%') and (((select #2) / (select #3)) >= 0.1)) (cost=1037.65 rows=1108) (actual time=0.355..22.850 rows=14 loops=1)
    -> Table scan on R (cost=1037.65 rows=9974) (actual time=0.044..4.358 rows=10000 loops=1)
    -> Select #2 (subquery in condition; dependent)
      -> Aggregate: sum(C.Count) (cost=4.85 rows=1) (actual time=0.007..0.007 rows=1 loops=167)
        -> Nested loop inner join (cost=4.75 rows=1) (actual time=0.004..0.006 rows=0 loops=167)
          -> Covering index lookup on O using PRIMARY (RestaurantId=R.RestaurantId) (cost=1.25 rows=10) (actual time=0.003..0.004 rows=1 loops=167)
          -> Filter: (C.Cleared = 1) (cost=0.25 rows=0.1) (actual time=0.003..0.003 rows=0 loops=151)
            -> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=151)
        -> Select #3 (subquery in condition; dependent)
          -> Aggregate: sum(C.Count) (cost=5.75 rows=1) (actual time=0.025..0.025 rows=1 loops=15)
            -> Nested loop inner join (cost=4.75 rows=10) (actual time=0.009..0.023 rows=10 loops=15)
              -> Covering index lookup on O using PRIMARY (RestaurantId=R.RestaurantId) (cost=1.25 rows=10) (actual time=0.007..0.009 rows=10 loops=15)
              -> Single-row index lookup on C using PRIMARY (CrimeId=O.CrimeId) (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=151)
          -> Filter: ((R.Hours like '%Monday: 11:0-17:0%') and ((select #5) <= 10)) (cost=1037.65 rows=1108) (actual time=0.269..24.478 rows=13 loops=1)
    -> Table scan on R (cost=1037.65 rows=9974) (actual time=0.054..4.698 rows=10000 loops=1)
    -> Select #5 (subquery in condition; dependent)
      -> Aggregate: avg(D.Price) (cost=1.17 rows=1) (actual time=0.046..0.046 rows=1 loops=27)
        -> Index lookup on D using RestaurantId (RestaurantId=R.RestaurantId) (cost=0.91 rows=3) (actual time=0.043..0.045 rows=2 loops=27)
```

The intuition of trying to index on Restaurant Name is because it is one of the attributes that involves set operation union. However, we observed the overall cost is still the same. This might be a result that among the attributes involving this set operation union, there exists a primary key 'RestaurantId' such that using this primary key can make the set operation done faster so that adding an index on RestaurantName is not helpful. Alternatively, it is possible that it indeed does improve the performance of this query but is so limited, since we can see the cost on the most bottom line truly decreased but it only affects 1+3 rows so it is negligible compared to the cost in total.

Therefore, for this query, as there are no other plausible indexing options (all natural join are based on primary keys), concluding from all three different index schemas above, having no indexing seems to be the best option for this query since none of them really make any improvement on the total cost.

Drop: DROP INDEX RestaurantName_idx ON Restaurant;

