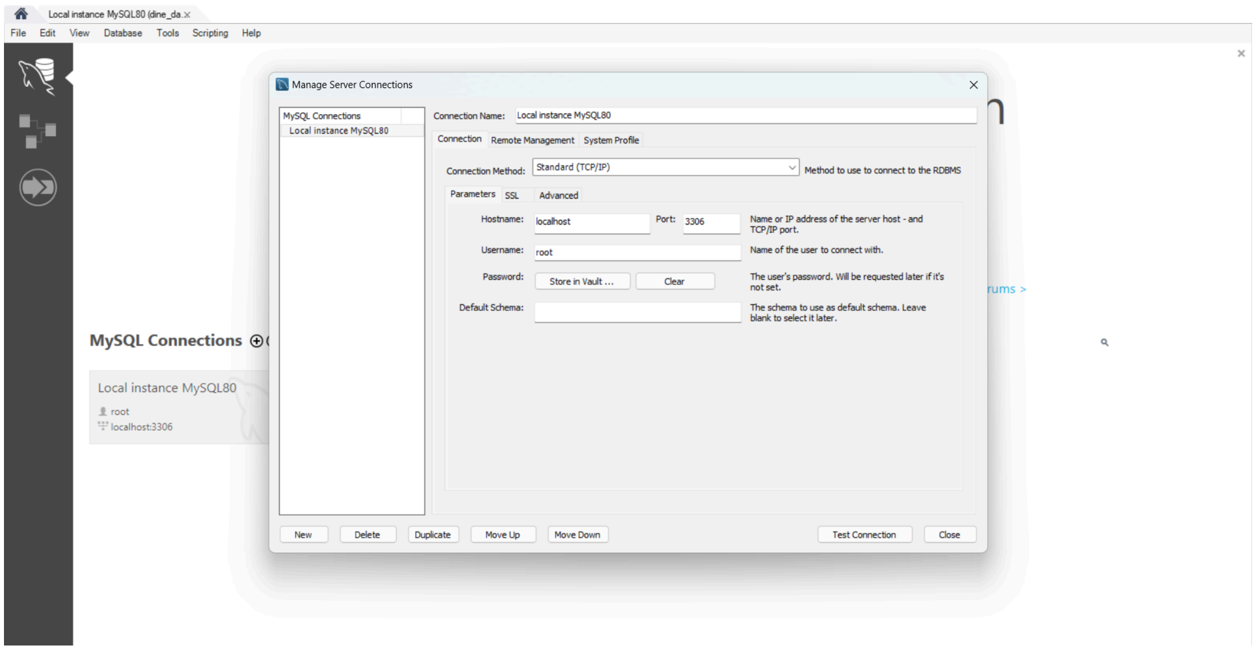
Stage 3: Database Implementation (MySQL WorkBench)

Local Connection:


Local instance MySQL80 (dine_da
File Edit View Database Tools Scripting Help
MySQL Connections
Local instance MySQL80
root
localhost:3306
Manage Server Connections
MySQL Connections
Local instance MySQL80
Connection Remote Management System Profile
Parameters SSL Advanced
Store in Vault ... Clear
New Delete Duplicate Move Up Move Down Test Connection Close

Connection Name: Local instance MySQL80

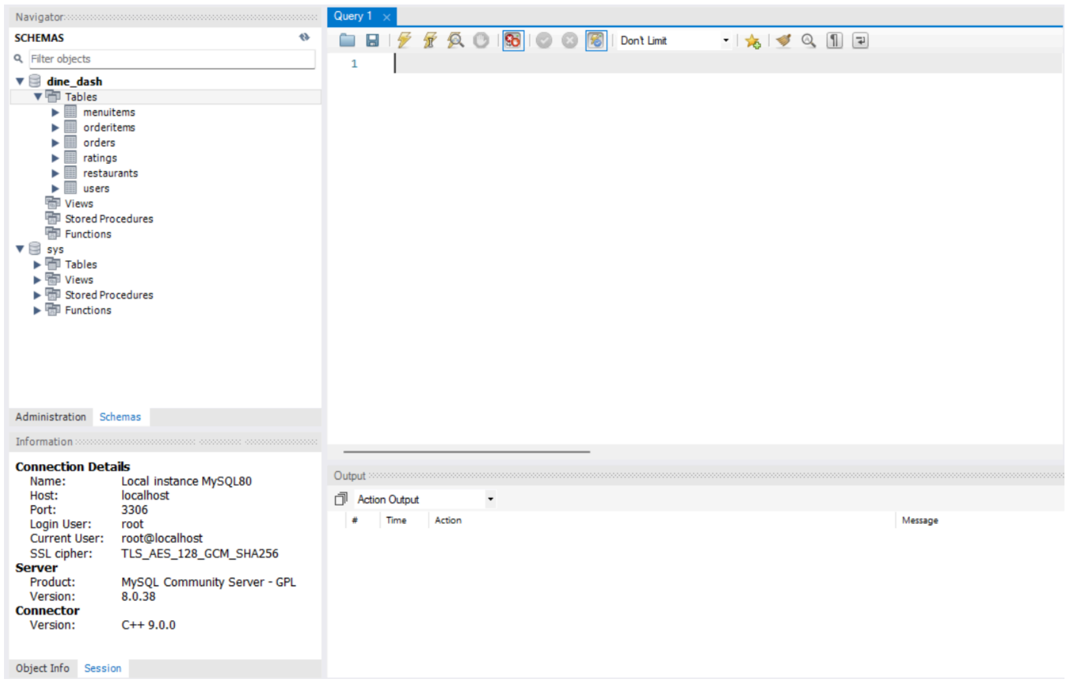
Connection Method: Standard (TCP/IP) — Method to use to connect to the RDBMS

Hostname: localhost Port: 3306 — Name or IP address of the server host - and TCP/IP port.

Username: root — Name of the user to connect with.

Password: — The user's password. Will be requested later if it's not set.

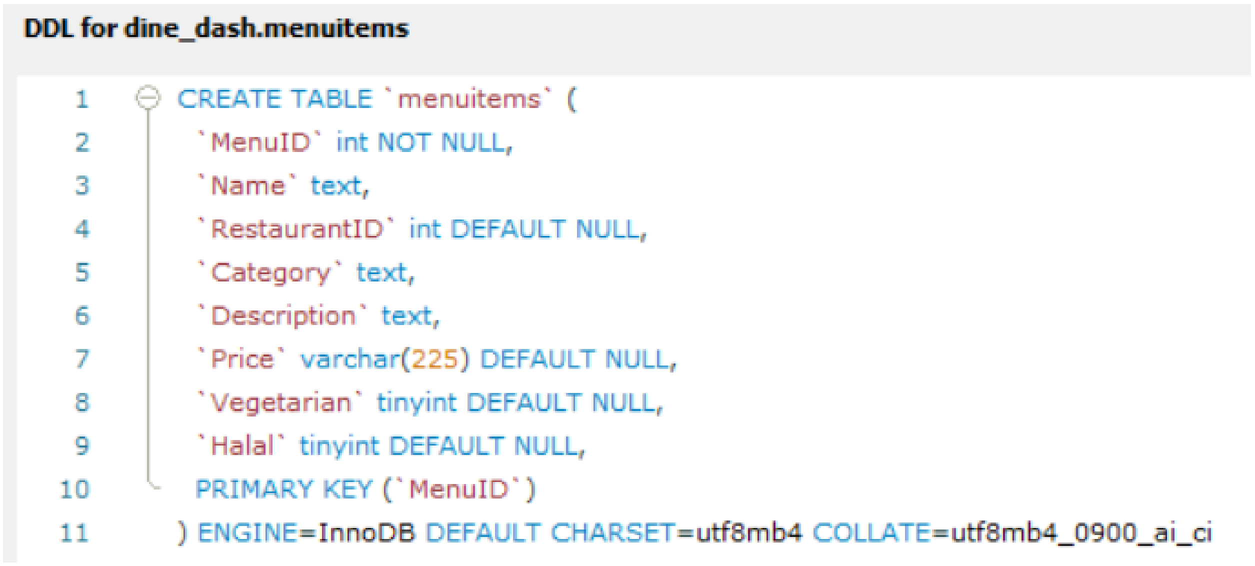
Default Schema: — The schema to use as default schema. Leave blank to select it later.



Connection Details
Name: Local instance MySQL80
Host: localhost
Port: 3306
Login User: root
Current User: root@localhost
SSL cipher: TLS_AES_128_GCM_SHA256
Server
Product: MySQL Community Server - GPL
Version: 8.0.38
Connector
Version: C++ 9.0.0

Database Design DDL(s):

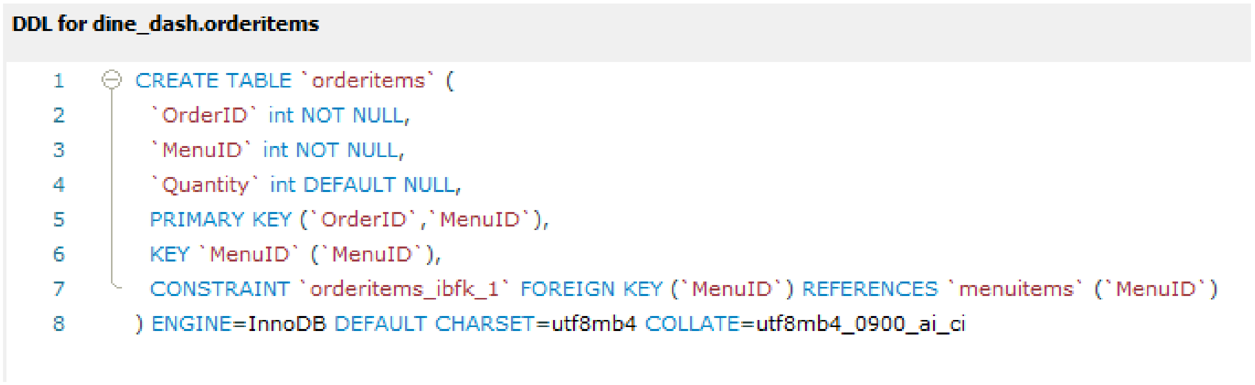
MenuItems



DDL for dine_dash.menuitems

```sql
CREATE TABLE `menuitems` (
  `MenuID` int NOT NULL,
  `Name` text,
  `RestaurantID` int DEFAULT NULL,
  `Category` text,
  `Description` text,
  `Price` varchar(225) DEFAULT NULL,
  `Vegetarian` tinyint DEFAULT NULL,
  `Halal` tinyint DEFAULT NULL,
  PRIMARY KEY (`MenuID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

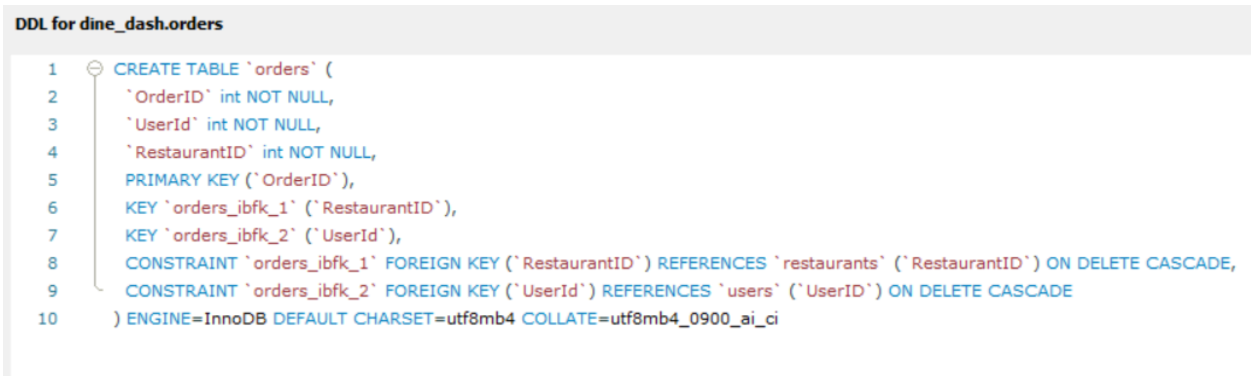
OrderItems



DDL for dine_dash.orderitems

```sql
CREATE TABLE `orderitems` (
  `OrderID` int NOT NULL,
  `MenuID` int NOT NULL,
  `Quantity` int DEFAULT NULL,
  PRIMARY KEY (`OrderID`,`MenuID`),
  KEY `MenuID` (`MenuID`),
  CONSTRAINT `orderitems_ibfk_1` FOREIGN KEY (`MenuID`) REFERENCES `menuitems` (`MenuID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

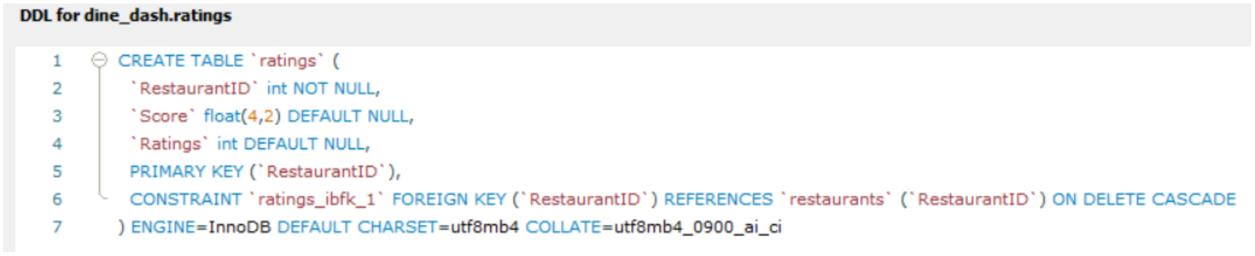
Orders



DDL for dine_dash.orders

```sql
CREATE TABLE `orders` (
  `OrderID` int NOT NULL,
  `UserId` int NOT NULL,
  `RestaurantID` int NOT NULL,
  PRIMARY KEY (`OrderID`),
  KEY `orders_ibfk_1` (`RestaurantID`),
  KEY `orders_ibfk_2` (`UserId`),
  CONSTRAINT `orders_ibfk_1` FOREIGN KEY (`RestaurantID`) REFERENCES `restaurants` (`RestaurantID`) ON DELETE CASCADE,
  CONSTRAINT `orders_ibfk_2` FOREIGN KEY (`UserId`) REFERENCES `users` (`UserID`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

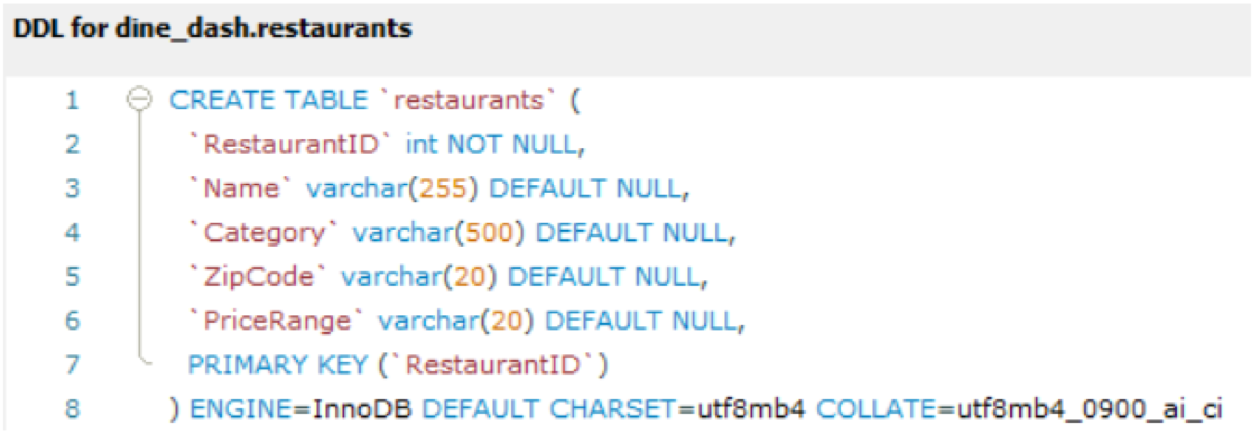
Ratings



DDL for dine_dash.ratings

```
CREATE TABLE `ratings` (
  `RestaurantID` int NOT NULL,
  `Score` float(4,2) DEFAULT NULL,
  `Ratings` int DEFAULT NULL,
  PRIMARY KEY (`RestaurantID`),
  CONSTRAINT `ratings_ibfk_1` FOREIGN KEY (`RestaurantID`) REFERENCES `restaurants` (`RestaurantID`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

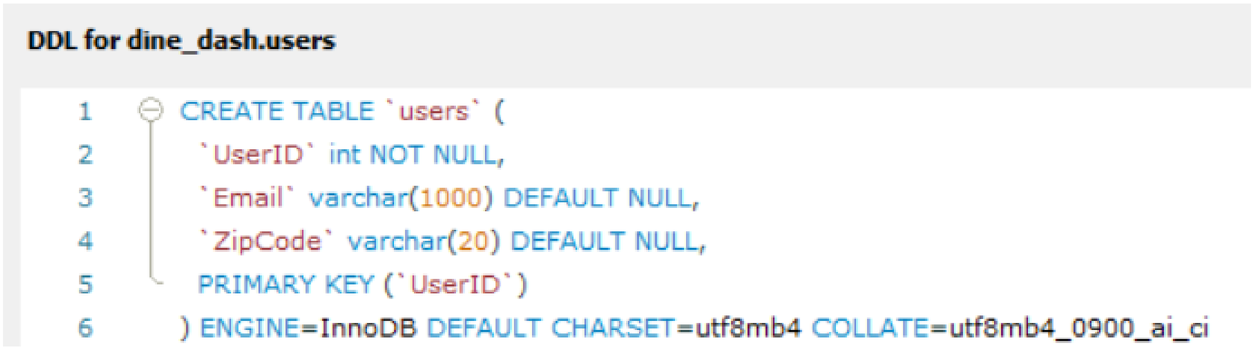
Restaurants



DDL for dine_dash.restaurants

```
CREATE TABLE `restaurants` (
  `RestaurantID` int NOT NULL,
  `Name` varchar(255) DEFAULT NULL,
  `Category` varchar(500) DEFAULT NULL,
  `ZipCode` varchar(20) DEFAULT NULL,
  `PriceRange` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`RestaurantID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

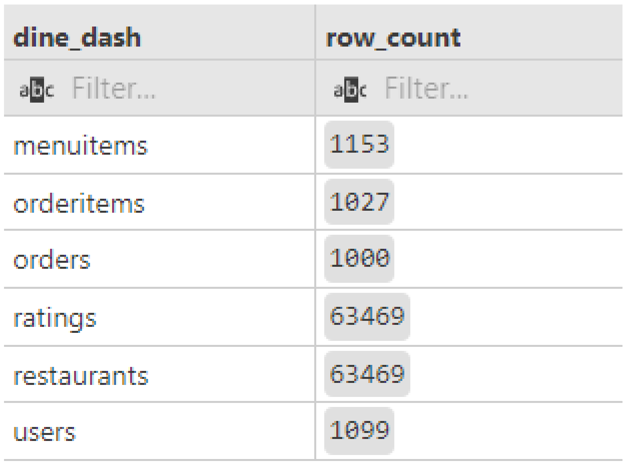
Users



DDL for dine_dash.users

```
CREATE TABLE `users` (
  `UserID` int NOT NULL,
  `Email` varchar(1000) DEFAULT NULL,
  `ZipCode` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`UserID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Table Sizes



dine_dash	row_count
Filter...	Filter...
menuitems	1153
orderitems	1027
orders	1000
ratings	63469
restaurants	63469
users	1099

Currently MySQL Workbench will not allow us to load all the data. There are roughly 1,048,576 rows but CSV Formatting has made it difficult to import. Our data is a bit skewed due to the inclusion of a smaller part of MenuItems. We realized that in the Description Column, some items have new lines which cause the CSV to insert new lines there and change the format of the data when Workbench interprets it.

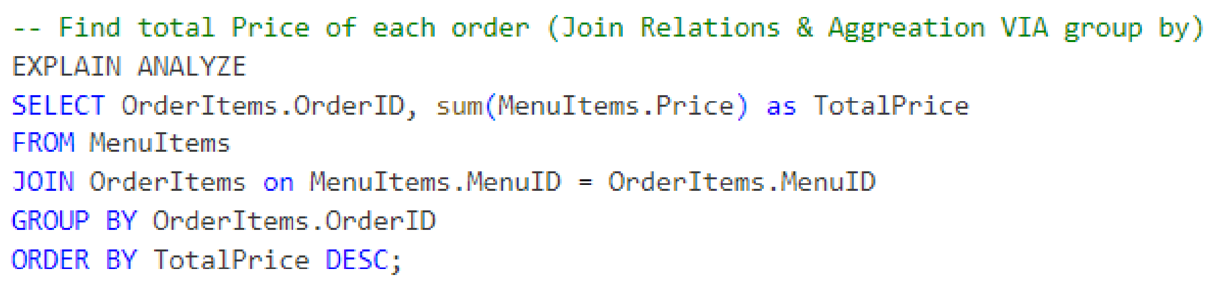
Some tables were populated with data through the Table Data Import Wizard feature with foreign and primary keys added

MenuItem and OrderItems were loaded through LOAD DATA INIT (changing table_name appropriately):

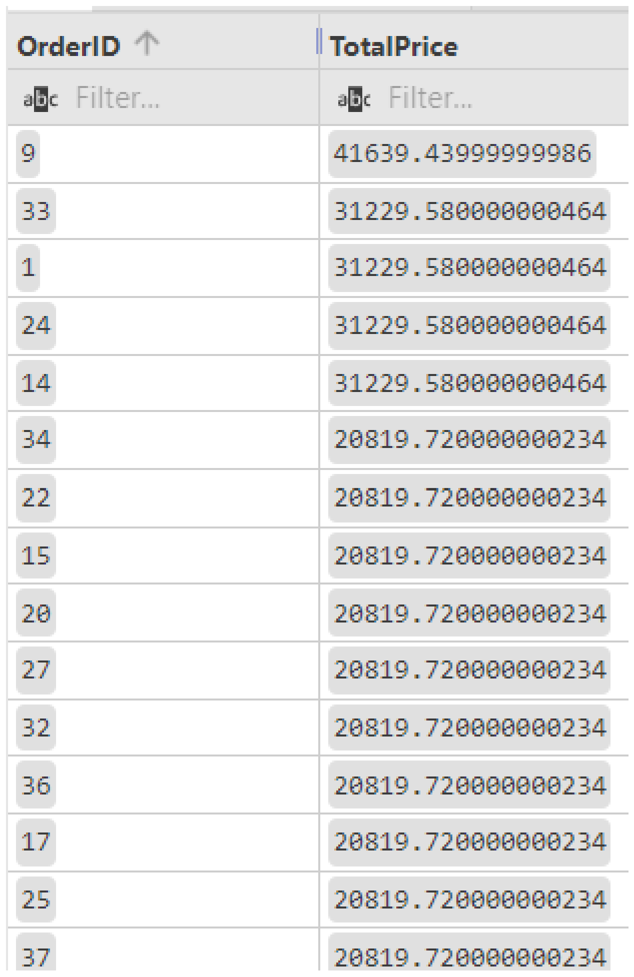
```
LOAD DATA INFILE "C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/table_name.csv"
INTO TABLE dine_dash.table_name
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES;
SELECT * FROM table_name;
```

Advanced Queries

Query 1: Find Total Price of Each Order (Join Relation & Aggregation via Group By) (Output > 15)

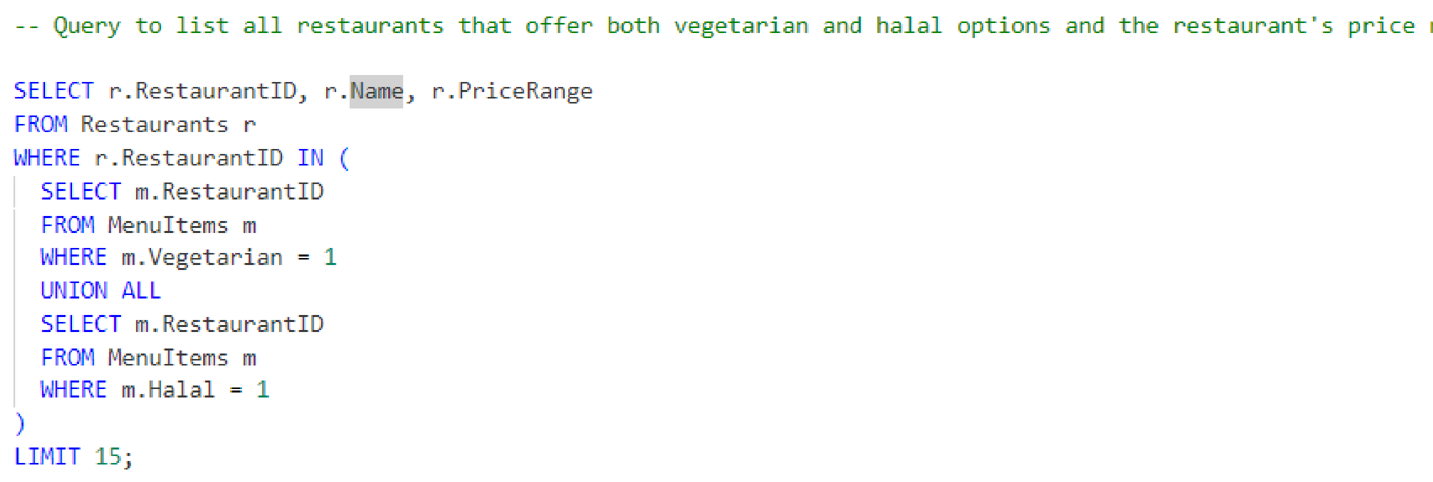


```
-- Find total Price of each order (Join Relations & Aggreation VIA group by)
EXPLAIN ANALYZE
SELECT OrderItems.OrderID, sum(MenuItems.Price) as TotalPrice
FROM MenuItems
JOIN OrderItems on MenuItems.MenuID = OrderItems.MenuID
GROUP BY OrderItems.OrderID
ORDER BY TotalPrice DESC;
```



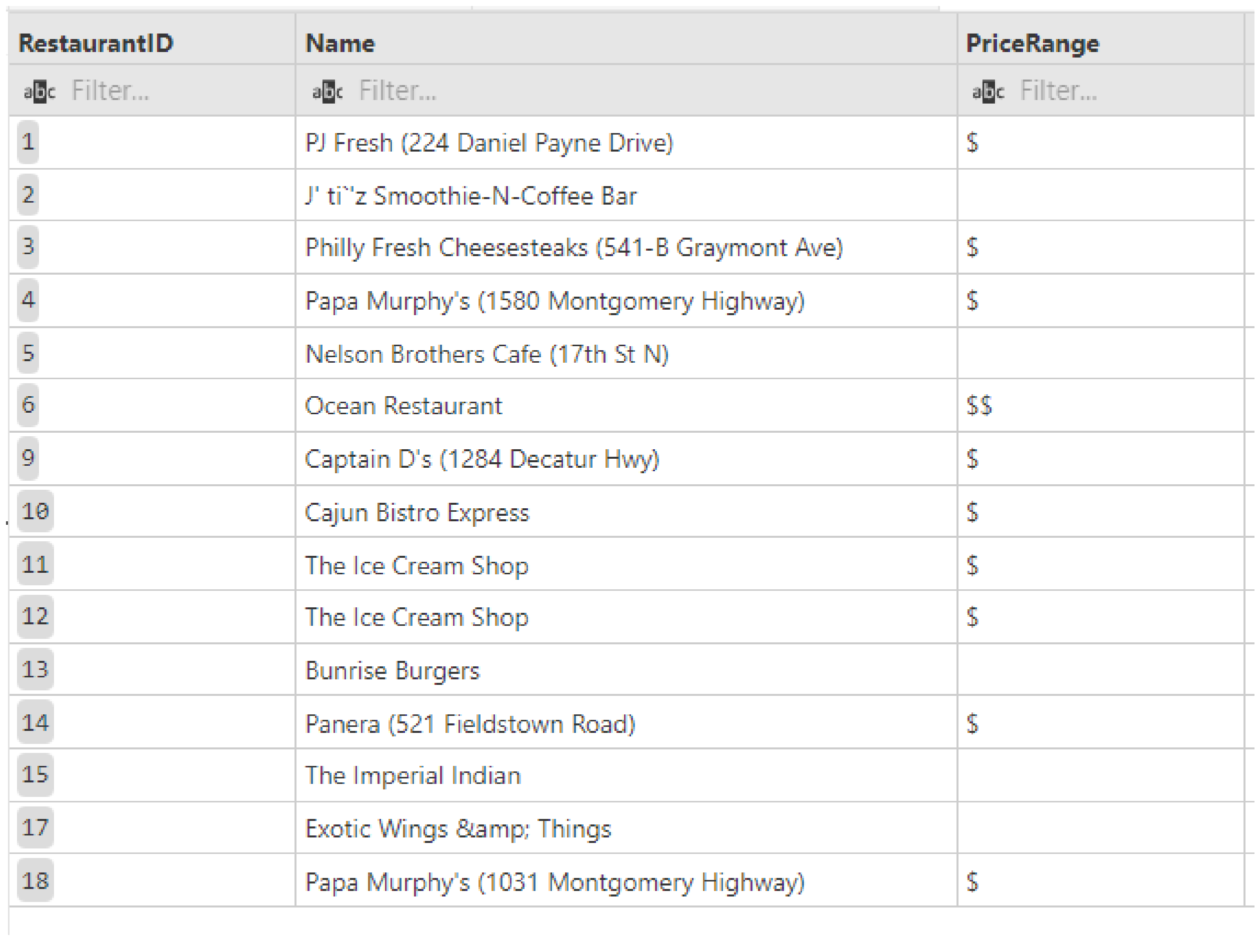
OrderID	TotalPrice
Filter...	Filter...
9	41639.43999999986
33	31229.580000000464
1	31229.580000000464
24	31229.580000000464
14	31229.580000000464
34	20819.720000000234
22	20819.720000000234
15	20819.720000000234
20	20819.720000000234
27	20819.720000000234
32	20819.720000000234
36	20819.720000000234
17	20819.720000000234
25	20819.720000000234
37	20819.720000000234

Query 2: Query to list all restaurants that offer both vegetarian and halal options and their price range (Output > 15)



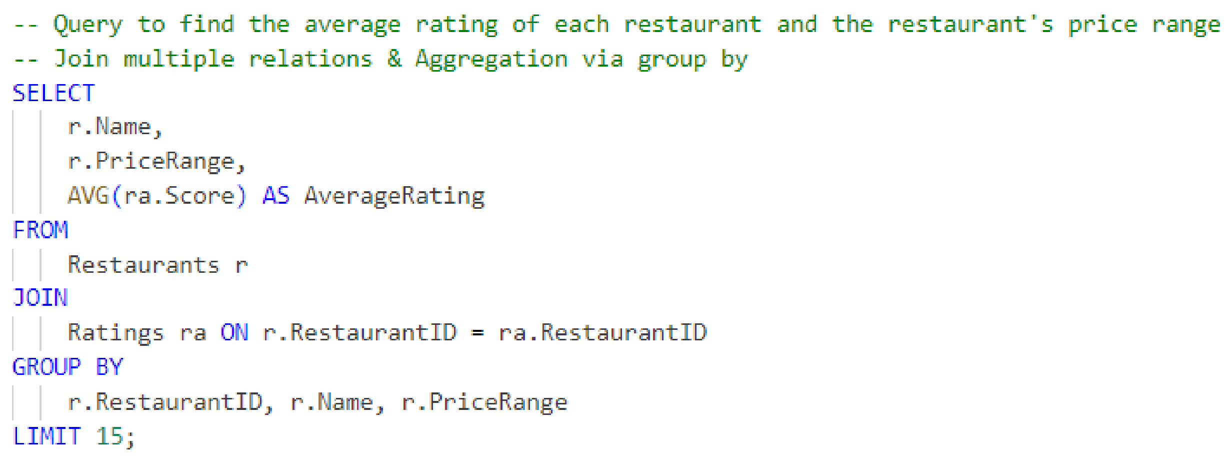
```
-- Query to list all restaurants that offer both vegetarian and halal options and the restaurant's price

SELECT r.RestaurantID, r.Name, r.PriceRange
FROM Restaurants r
WHERE r.RestaurantID IN (
  SELECT m.RestaurantID
  FROM MenuItems m
  WHERE m.Vegetarian = 1
  UNION ALL
  SELECT m.RestaurantID
  FROM MenuItems m
  WHERE m.Halal = 1
)
LIMIT 15;
```

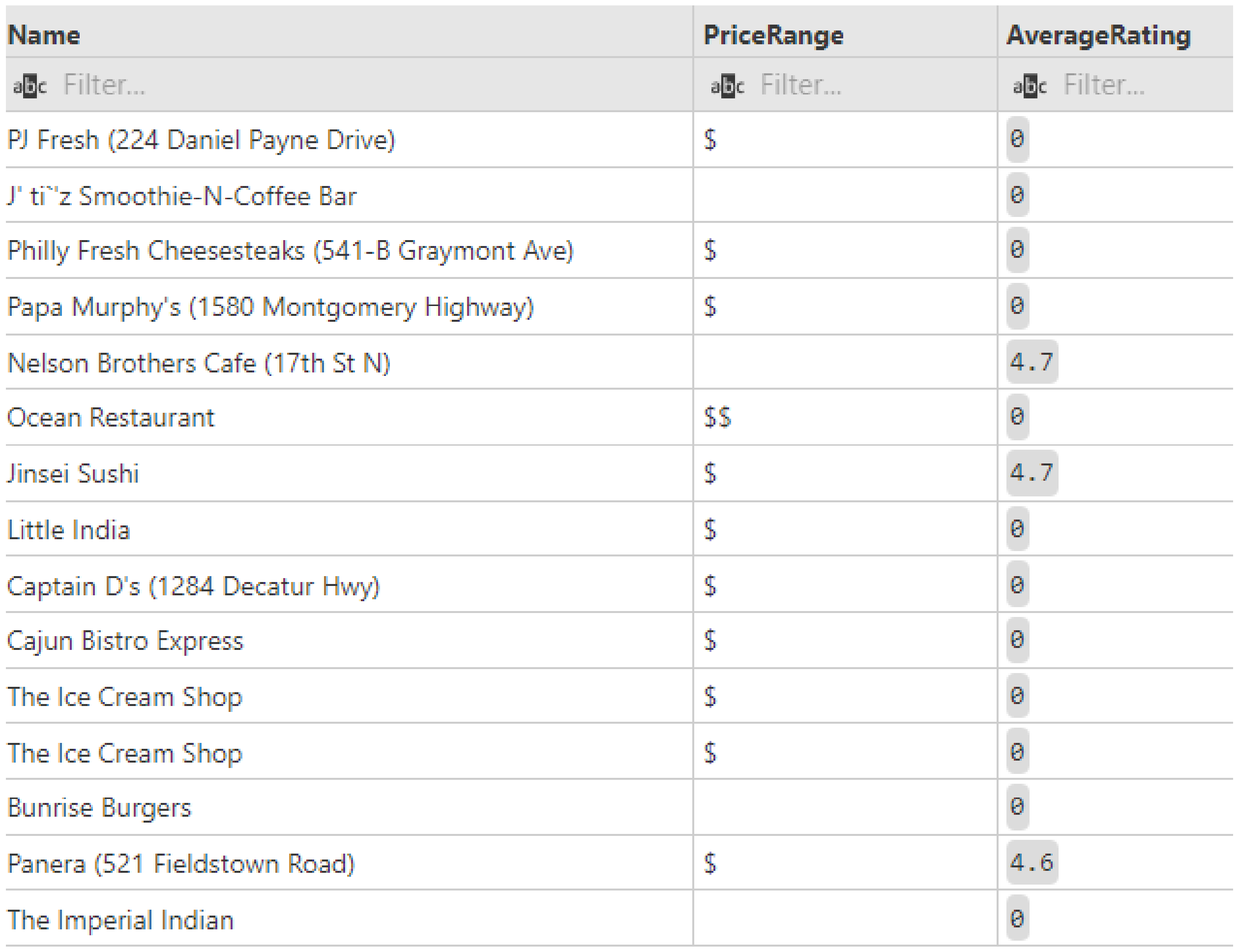


RestaurantID	Name	PriceRange
Filter...	Filter...	Filter...
1	PJ Fresh (224 Daniel Payne Drive)	$
2	J' ti`'z Smoothie-N-Coffee Bar	
3	Philly Fresh Cheesesteaks (541-B Graymont Ave)	$
4	Papa Murphy's (1580 Montgomery Highway)	$
5	Nelson Brothers Cafe (17th St N)	
6	Ocean Restaurant	$$
9	Captain D's (1284 Decatur Hwy)	$
10	Cajun Bistro Express	$
11	The Ice Cream Shop	$
12	The Ice Cream Shop	$
13	Bunrise Burgers	
14	Panera (521 Fieldstown Road)	$
15	The Imperial Indian	
17	Exotic Wings &amp; Things	
18	Papa Murphy's (1031 Montgomery Highway)	$

Query 3:Query to find average rating of each restaurant and the restaurant's price range

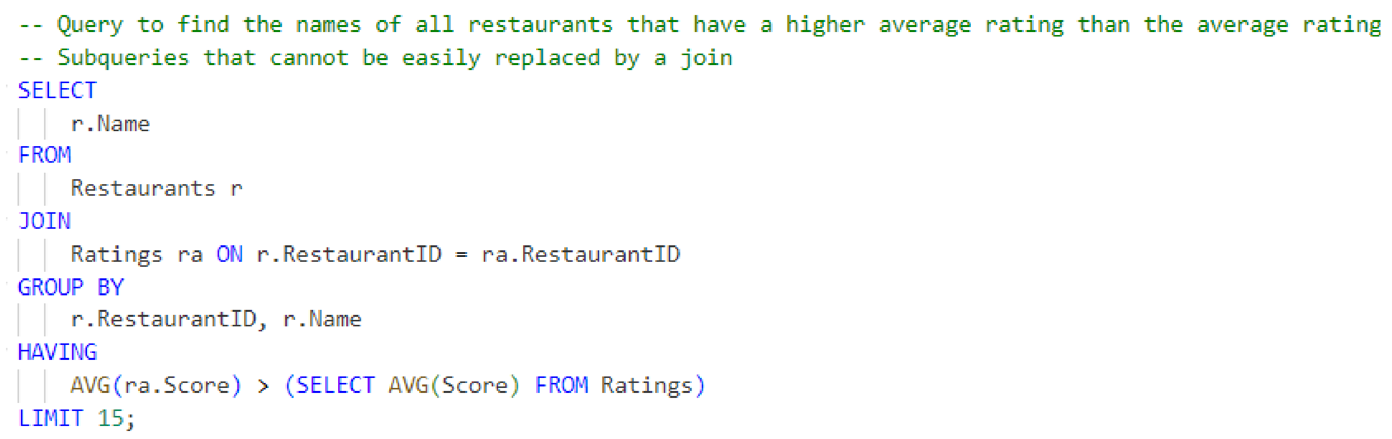


```sql
-- Query to find the average rating of each restaurant and the restaurant's price range
-- Join multiple relations & Aggregation via group by
SELECT
    r.Name,
    r.PriceRange,
    AVG(ra.Score) AS AverageRating
FROM
    Restaurants r
JOIN
    Ratings ra ON r.RestaurantID = ra.RestaurantID
GROUP BY
    r.RestaurantID, r.Name, r.PriceRange
LIMIT 15;
```

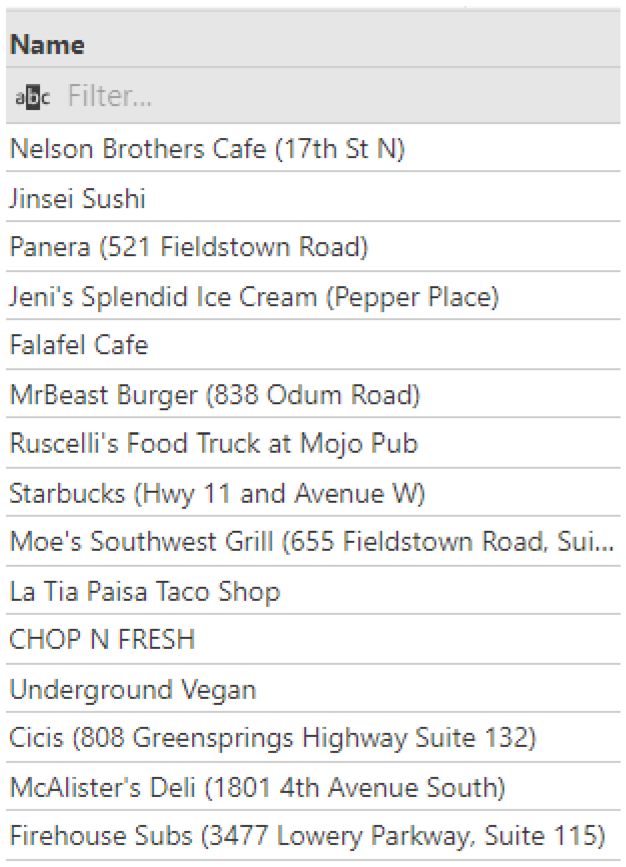


Name	PriceRange	AverageRating
Filter...	Filter...	Filter...
PJ Fresh (224 Daniel Payne Drive)	$	0
J' ti`'z Smoothie-N-Coffee Bar		0
Philly Fresh Cheesesteaks (541-B Graymont Ave)	$	0
Papa Murphy's (1580 Montgomery Highway)	$	0
Nelson Brothers Cafe (17th St N)		4.7
Ocean Restaurant	$$	0
Jinsei Sushi	$	4.7
Little India	$	0
Captain D's (1284 Decatur Hwy)	$	0
Cajun Bistro Express	$	0
The Ice Cream Shop	$	0
The Ice Cream Shop	$	0
Bunrise Burgers		0
Panera (521 Fieldstown Road)	$	4.6
The Imperial Indian		0

Query 4: Query to find the names of all restaurants that have a higher average rating than the average rating of all restaurants (Output > 15)



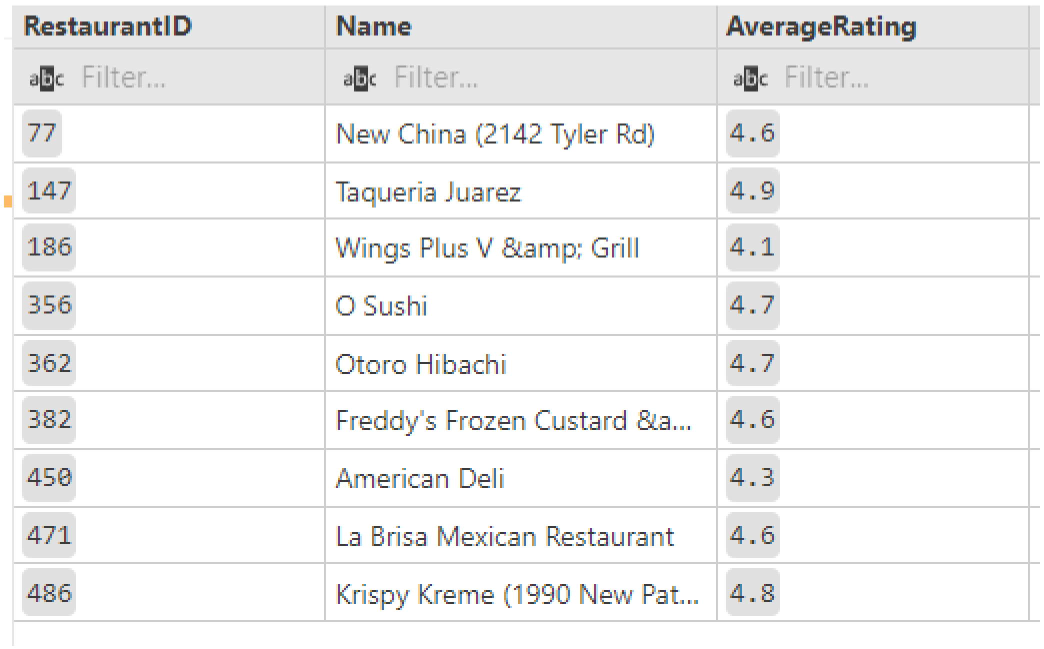
```sql
-- Query to find the names of all restaurants that have a higher average rating than the average rating
-- Subqueries that cannot be easily replaced by a join
SELECT
    r.Name
FROM
    Restaurants r
JOIN
    Ratings ra ON r.RestaurantID = ra.RestaurantID
GROUP BY
    r.RestaurantID, r.Name
HAVING
    AVG(ra.Score) > (SELECT AVG(Score) FROM Ratings)
LIMIT 15;
```



Name
Filter...
Nelson Brothers Cafe (17th St N)
Jinsei Sushi
Panera (521 Fieldstown Road)
Jeni's Splendid Ice Cream (Pepper Place)
Falafel Cafe
MrBeast Burger (838 Odum Road)
Ruscelli's Food Truck at Mojo Pub
Starbucks (Hwy 11 and Avenue W)
Moe's Southwest Grill (655 Fieldstown Road, Sui...
La Tia Paisa Taco Shop
CHOP N FRESH
Underground Vegan
Cicis (808 Greensprings Highway Suite 132)
McAlister's Deli (1801 4th Avenue South)
Firehouse Subs (3477 Lowery Parkway, Suite 115)

Query 5: Find the restaurants with ratings above the average in zipcode 35226

```sql
EXPLAIN ANALYZE
SELECT r.RestaurantID, r.Name, AVG(ra.Score) AS AverageRating
FROM Restaurants r
JOIN Ratings ra ON r.RestaurantID = ra.RestaurantID
WHERE r.Zipcode = '35226'
GROUP BY r.RestaurantID, r.Name
HAVING AVG(ra.Score) > (
    SELECT AVG(ra2.Score)
    FROM Restaurants r2
    JOIN Ratings ra2 ON r2.RestaurantID = ra2.RestaurantID
    WHERE r2.Zipcode = '35226'
);
```



RestaurantID	Name	AverageRating
Filter...	Filter...	Filter...
77	New China (2142 Tyler Rd)	4.6
147	Taqueria Juarez	4.9
186	Wings Plus V &amp; Grill	4.1
356	O Sushi	4.7
362	Otoro Hibachi	4.7
382	Freddy's Frozen Custard &a...	4.6
450	American Deli	4.3
471	La Brisa Mexican Restaurant	4.6
486	Krispy Kreme (1990 New Pat...	4.8

Explain Analysis on Queries (Without LIMIT)

Query 1:

```
-> Sort: TotalPrice DESC  (actual time=1.86..1.88 rows=999 loops=1)
    -> Stream results  (cost=566 rows=999) (actual time=0.0521..1.67
rows=999 loops=1)
        -> Group aggregate: sum(menuitems.Price)  (cost=566 rows=999)
(actual time=0.0501..1.55 rows=999 loops=1)
            -> Nested loop inner join  (cost=464 rows=1027) (actual
time=0.0391..1.34 rows=1027 loops=1)
                -> Covering index scan on OrderItems using OrderID
(cost=104 rows=1027) (actual time=0.0277..0.223 rows=1027 loops=1)
                -> Single-row index lookup on MenuItems using PRIMARY
(MenuID=orderitems.ItemID)  (cost=0.25 rows=1) (actual time=986e-6..0.001
rows=1 loops=1027)
```

Query 2:

```
-> Filter: <in_optimizer>(r.RestaurantID,<exists>(select #2))  (cost=6233
rows=61123) (actual time=0.248..54979 rows=21 loops=1)
    -> Table scan on r  (cost=6233 rows=61123) (actual time=0.122..62
rows=63469 loops=1)
    -> Select #2 (subquery in condition; dependent)
        -> Limit: 1 row(s)  (cost=217 rows=1) (actual time=0.863..0.863
rows=331e-6 loops=63469)
            -> Append  (cost=217 rows=2) (actual time=0.862..0.862
rows=331e-6 loops=63469)
                -> Stream results  (cost=109 rows=1) (actual
time=0.436..0.436 rows=331e-6 loops=63469)
                    -> Limit: 1 row(s)  (cost=109 rows=1) (actual
time=0.436..0.436 rows=331e-6 loops=63469)
                        -> Filter: ((m.Vegetarian = 1) and
(<cache>(r.RestaurantID) = m.RestaurantID))  (cost=109 rows=11.5) (actual
time=0.436..0.436 rows=331e-6 loops=63469)
                            -> Table scan on m  (cost=109 rows=1153)
(actual time=0.0214..0.353 rows=1153 loops=63469)
                -> Stream results  (cost=109 rows=1) (actual
time=0.425..0.425 rows=0 loops=63448)
```

```
                    -> Limit: 1 row(s)  (cost=109 rows=1) (actual
time=0.425..0.425 rows=0 loops=63448)
                        -> Filter: ((m.Halal = 1) and
(<cache>(r.RestaurantID) = m.RestaurantID))  (cost=109 rows=11.5) (actual
time=0.425..0.425 rows=0 loops=63448)
                            -> Table scan on m  (cost=109 rows=1153)
(actual time=0.0215..0.353 rows=1153 loops=63448)
```

Query 3:

```
-> Table scan on <temporary>  (actual time=596..624 rows=63469 loops=1)
    -> Aggregate using temporary table  (actual time=596..596 rows=63468
loops=1)
        -> Nested loop inner join  (cost=27626 rows=61123) (actual
time=0.0721..131 rows=63469 loops=1)
            -> Table scan on r  (cost=6233 rows=61123) (actual
time=0.0592..29.7 rows=63469 loops=1)
            -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual
time=0.00143..0.00145 rows=1 loops=63469)
```

Query 4:

```
-> Filter: (??? > (select #2))  (actual time=241..254 rows=35281 loops=1)
    -> Table scan on <temporary>  (actual time=226..234 rows=63469
loops=1)
        -> Aggregate using temporary table  (actual time=226..226
rows=63469 loops=1)
            -> Nested loop inner join  (cost=27626 rows=61123) (actual
time=0.058..121 rows=63469 loops=1)
                -> Table scan on r  (cost=6233 rows=61123) (actual
time=0.0463..27.7 rows=63469 loops=1)
                -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual
time=0.0013..0.00133 rows=1 loops=63469)
    -> Select #2 (subquery in condition; run only once)
```

```
        -> Aggregate: avg(ratings.Score)  (cost=12765 rows=1) (actual
time=14.2..14.2 rows=1 loops=1)
            -> Table scan on Ratings  (cost=6403 rows=63624) (actual
time=0.0248..11 rows=63469 loops=1)
```

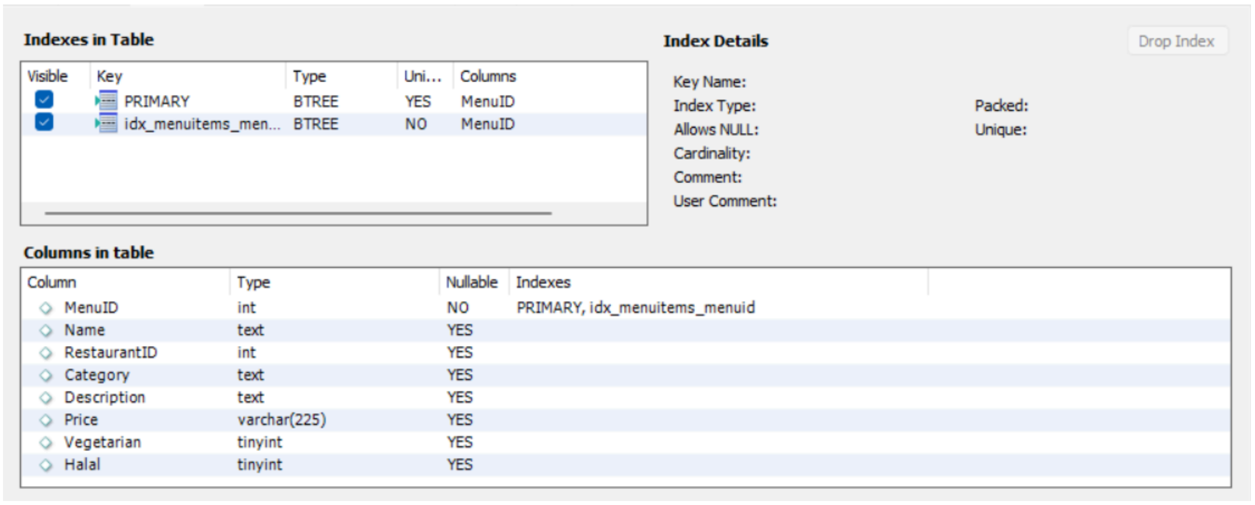
Query 5:

```
-> Filter: (??? > (select #2))  (actual time=51.7..51.7 rows=9 loops=1)
    -> Table scan on <temporary>  (actual time=25.1..25.1 rows=14 loops=1)
        -> Aggregate using temporary table  (actual time=25.1..25.1
rows=14 loops=1)
            -> Nested loop inner join  (cost=8372 rows=6112) (actual
time=0.0815..25.1 rows=14 loops=1)
                -> Filter: (r.ZipCode = '35226')  (cost=6233 rows=6112)
(actual time=0.0658..25 rows=14 loops=1)
                    -> Table scan on r  (cost=6233 rows=61123) (actual
time=0.0601..21 rows=63469 loops=1)
                -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual
time=0.00268..0.00272 rows=1 loops=14)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(ra2.Score)  (cost=8983 rows=1) (actual
time=26.5..26.5 rows=1 loops=1)
            -> Nested loop inner join  (cost=8372 rows=6112) (actual
time=0.0441..26.5 rows=14 loops=1)
                -> Filter: (r2.ZipCode = '35226')  (cost=6233 rows=6112)
(actual time=0.0316..26.5 rows=14 loops=1)
                    -> Table scan on r2  (cost=6233 rows=61123) (actual
time=0.0304..21.6 rows=63469 loops=1)
                -> Single-row index lookup on ra2 using PRIMARY
(RestaurantID=r2.RestaurantID)  (cost=0.25 rows=1) (actual
time=0.00169..0.0017 rows=1 loops=14)
```

Advanced Queries Analyzed Post Indices

Query 1:

Index on MenuItems.MenuID



Indexes in Table

Visible	Key	Type	Uni...	Columns
✓	PRIMARY	BTREE	YES	MenuID
✓	idx_menuitems_men...	BTREE	NO	MenuID

Index Details

Key Name:
Index Type:
Allows NULL:
Cardinality:
Comment:
User Comment:
Packed:
Unique:

Columns in table

Column	Type	Nullable	Indexes
MenuID	int	NO	PRIMARY, idx_menuitems_menuid
Name	text	YES	
RestaurantID	int	YES	
Category	text	YES	
Description	text	YES	
Price	varchar(225)	YES	
Vegetarian	tinyint	YES	
Halal	tinyint	YES	

```
-- Find total Price of each order (Join Relations & Aggreation VIA group by)
CREATE INDEX idx_menuitems_ID ON MenuItems(MenuID);
EXPLAIN ANALYZE
SELECT OrderItems.OrderID, sum(MenuItems.Price) as TotalPrice
FROM MenuItems
JOIN OrderItems on MenuItems.MenuID = OrderItems.MenuID
GROUP BY OrderItems.OrderID
ORDER BY TotalPrice DESC;
DROP INDEX idx_menuitems_ID ON MenuItems;
```

Output of Analyze:

```
-> Sort: TotalPrice DESC  (actual time=2.06..2.09 rows=999 loops=1)
    -> Stream results  (cost=566 rows=999) (actual time=0.0518..1.88
rows=999 loops=1)
        -> Group aggregate: sum(menuitems.Price)  (cost=566 rows=999)
(actual time=0.0495..1.74 rows=999 loops=1)
            -> Nested loop inner join  (cost=464 rows=1027) (actual
time=0.0368..1.52 rows=1027 loops=1)
                -> Covering index scan on OrderItems using OrderID
(cost=104 rows=1027) (actual time=0.024..0.236 rows=1027 loops=1)
                -> Single-row index lookup on MenuItems using PRIMARY
(MenuID=orderitems.ItemID)  (cost=0.25 rows=1) (actual
time=0.00114..0.00116 rows=1 loops=1027)
```

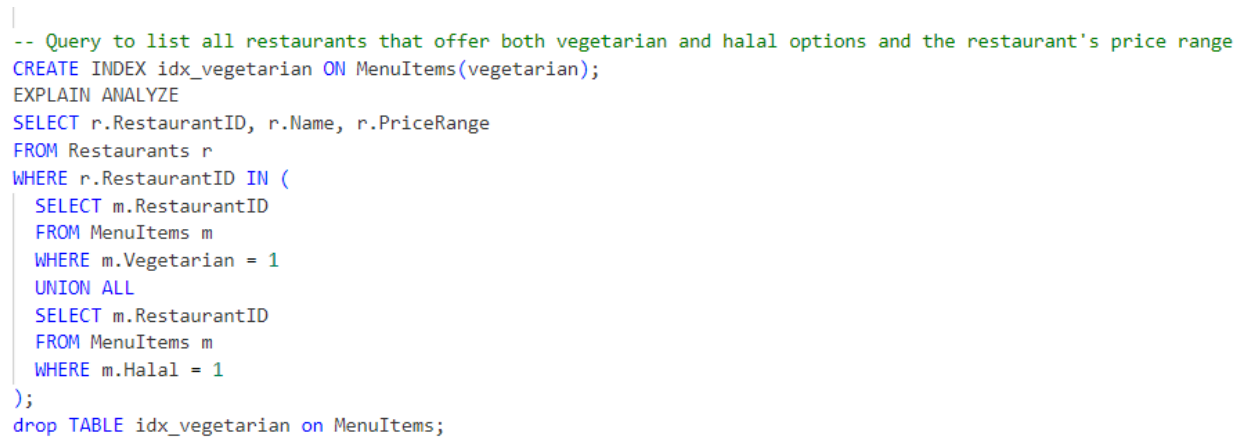
Analysis of Query 1: Index on MenuItems.MenuID

We chose an index on MenuItems.MenuID because we were hoping to improve the efficiency within the nested loop join operation. This index was intending to facilitate a faster lookup of matching rows in MenuItems that correspond to ItemID' in OrderItems.The query involves aggregating and summing prices from MenuItems and joins MenuItems to OrderItems on ItemID and MenuID. It sorts the results based on TotalPrice in descending order.

The cost and time of the indexed query are comparable, but the index query has a minorly better result. This is evident through the actual time improvement for operations including the nested loop join.

The index on MenuItems.MenuID was intended to optimize the nested loop join and speedup the lookup process. While the index enabled fast single-row lookups, the overall impact on the join operation's cost was limited. The aggregate operations had similar execution time, but the index did show a minor improvement in execution time. There was some variation in execution times which could be due to caching mechanisms within the database. The index and original query might have similar performance levels due to fairly efficiently accessing tables and not including excessive data scanning.

Query 2: Index on MenuItems.Vegetarian:



```
-- Query to list all restaurants that offer both vegetarian and halal options and the restaurant's price range
CREATE INDEX idx_vegetarian ON MenuItems(vegetarian);
EXPLAIN ANALYZE
SELECT r.RestaurantID, r.Name, r.PriceRange
FROM Restaurants r
WHERE r.RestaurantID IN (
  SELECT m.RestaurantID
  FROM MenuItems m
  WHERE m.Vegetarian = 1
  UNION ALL
  SELECT m.RestaurantID
  FROM MenuItems m
  WHERE m.Halal = 1
);
drop TABLE idx_vegetarian on MenuItems;
```

Output on Analyze:

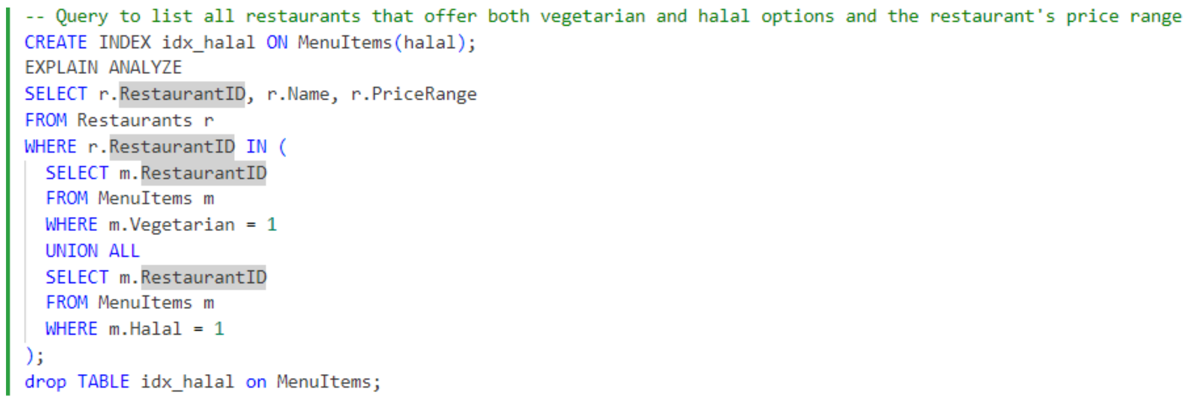
```
-> Filter: <in_optimizer>(r.RestaurantID,<exists>(select #2))  (cost=6233
rows=61123) (actual time=0.368..36896 rows=21 loops=1)
    -> Table scan on r  (cost=6233 rows=61123) (actual time=0.115..68.1
rows=63469 loops=1)
    -> Select #2 (subquery in condition; dependent)
        -> Limit: 1 row(s)  (cost=121 rows=1) (actual time=0.577..0.577
rows=331e-6 loops=63469)
            -> Append  (cost=121 rows=2) (actual time=0.577..0.577
rows=331e-6 loops=63469)
```

```
                    -> Stream results  (cost=12.2 rows=1) (actual
time=0.141..0.141 rows=331e-6 loops=63469)
                         -> Limit: 1 row(s)  (cost=12.2 rows=1) (actual
time=0.141..0.141 rows=331e-6 loops=63469)
                              -> Filter: (<cache>(r.RestaurantID) =
m.RestaurantID)  (cost=12.2 rows=9.7) (actual time=0.141..0.141
rows=331e-6 loops=63469)
                                   -> Index lookup on m using veg_idx
(Vegetarian=1)  (cost=12.2 rows=97) (actual time=0.122..0.135 rows=97
loops=63469)
                    -> Stream results  (cost=109 rows=1) (actual
time=0.435..0.435 rows=0 loops=63448)
                         -> Limit: 1 row(s)  (cost=109 rows=1) (actual
time=0.435..0.435 rows=0 loops=63448)
                              -> Filter: ((m.Halal = 1) and
(<cache>(r.RestaurantID) = m.RestaurantID))  (cost=109 rows=11.5) (actual
time=0.434..0.434 rows=0 loops=63448)
                                   -> Table scan on m  (cost=109 rows=1153)
(actual time=0.0219..0.359 rows=1153 loops=63448)
```

Analysis of Query 2 index on MenuItems.Vegetarian:

This query aims to list out the restaurants that provide both vegetarian, as well as halal options. In order to achieve this, the query searches for Restaurants that provide vegetarian options, Restaurants that provide halal options, and returns the set of Restaurants that occur in both result sets.
Due to the various number of times the ‘vegetarian’ attribute is accessed in this query’s WHERE clause, we experimented with adding an index on it. We hoped to optimize the data retrieval operations performed on this field, and hence improve the overall performance of the query as a whole.
As per our analysis, comparing the costs of the various subqueries of this advanced query shows a significant reduction, especially for the subquery filtering out all non- vegetarian items. Due to its noticeable improvement, we decided to include it as part of our final design.

Query 2: Index on MenuItems.Halal



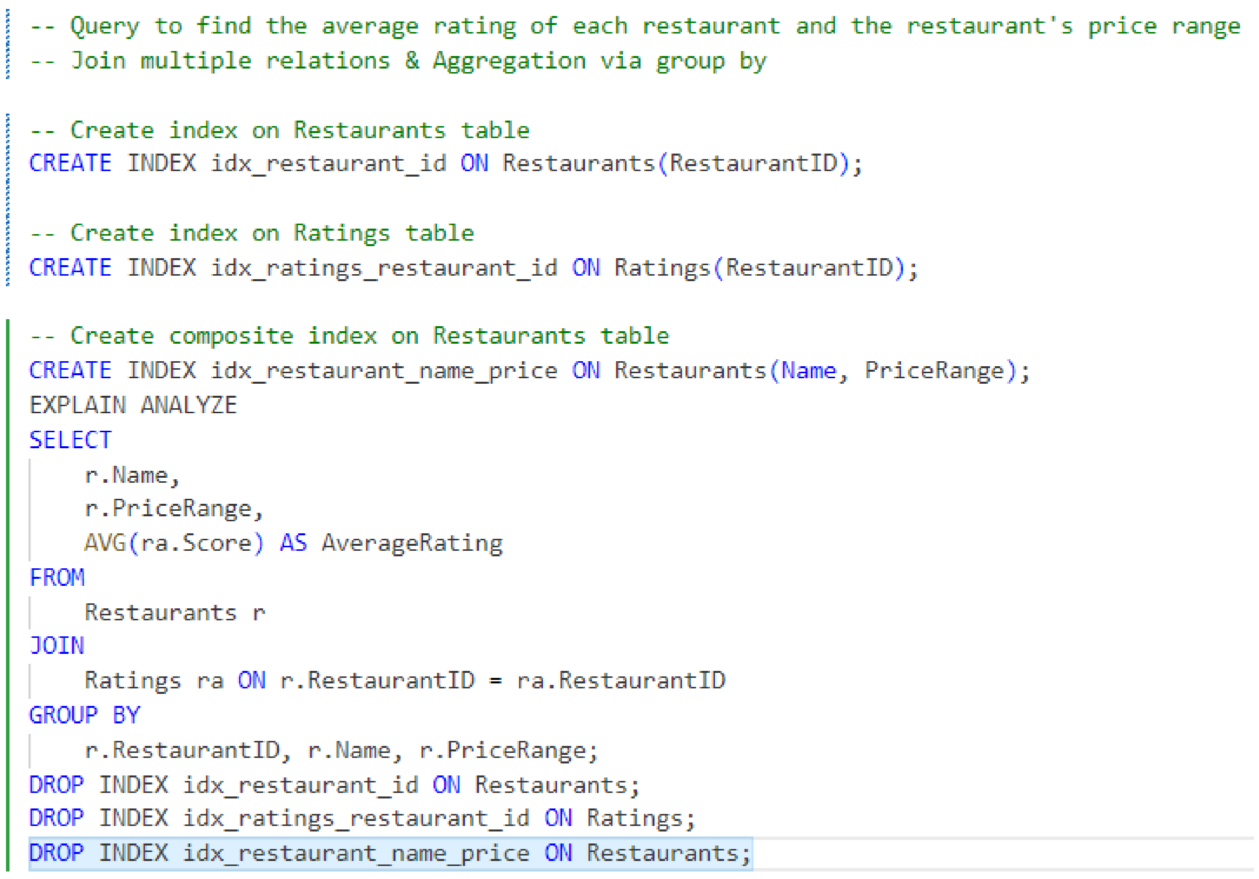
```sql
-- Query to list all restaurants that offer both vegetarian and halal options and the restaurant's price range
CREATE INDEX idx_halal ON MenuItems(halal);
EXPLAIN ANALYZE
SELECT r.RestaurantID, r.Name, r.PriceRange
FROM Restaurants r
WHERE r.RestaurantID IN (
  SELECT m.RestaurantID
  FROM MenuItems m
  WHERE m.Vegetarian = 1
  UNION ALL
  SELECT m.RestaurantID
  FROM MenuItems m
  WHERE m.Halal = 1
);
drop TABLE idx_halal on MenuItems;
```

```
-> Filter: <in_optimizer>(r.RestaurantID,<exists>(select #2))  (cost=6233
rows=61123) (actual time=0.385..29258 rows=21 loops=1)
    -> Table scan on r  (cost=6233 rows=61123) (actual time=0.278..63.9
rows=63469 loops=1)
    -> Select #2 (subquery in condition; dependent)
        -> Limit: 1 row(s)  (cost=109 rows=1) (actual time=0.458..0.458
rows=331e-6 loops=63469)
            -> Append  (cost=109 rows=1.1) (actual time=0.458..0.458
rows=331e-6 loops=63469)
                -> Stream results  (cost=109 rows=1) (actual
time=0.452..0.452 rows=331e-6 loops=63469)
                    -> Limit: 1 row(s)  (cost=109 rows=1) (actual
time=0.452..0.452 rows=331e-6 loops=63469)
                        -> Filter: ((m.Vegetarian = 1) and
(<cache>(r.RestaurantID) = m.RestaurantID))  (cost=109 rows=11.5) (actual
time=0.452..0.452 rows=331e-6 loops=63469)
                            -> Table scan on m  (cost=109 rows=1153)
(actual time=0.021..0.365 rows=1153 loops=63469)
                -> Stream results  (cost=0.26 rows=0.1) (actual
time=0.0048..0.0048 rows=0 loops=63448)
                    -> Limit: 1 row(s)  (cost=0.26 rows=0.1) (actual
time=0.00439..0.00439 rows=0 loops=63448)
                        -> Filter: (<cache>(r.RestaurantID) =
m.RestaurantID)  (cost=0.26 rows=0.1) (actual time=0.00423..0.00423 rows=0
loops=63448)
```

```
                                    -> Index lookup on m using halal_idx (Halal=1)
(cost=0.26 rows=1) (actual time=0.00405..0.00405 rows=0 loops=63448)
```

Similarly, due to the number of times the 'halal' attribute of MenuItems is accessed by the subquery of the WHERE clause, we experimented with creating an index on it.
Our analysis showed a significant decrease in the cost of subqueries retrieving data pertinent to the halal attribute. The cost was reduced by a larger margin compared to the reduction provided by the index on the vegetarian attribute, which is likely due to the fewer number of entries having a notable value for the halal attribute in our dataset.

Query 3: Indices on Restaurants.RestaurantID, Ratings.RestaurantID, Restaurants.Name, Restaurants.PriceRange



```sql
-- Query to find the average rating of each restaurant and the restaurant's price range
-- Join multiple relations & Aggregation via group by

-- Create index on Restaurants table
CREATE INDEX idx_restaurant_id ON Restaurants(RestaurantID);

-- Create index on Ratings table
CREATE INDEX idx_ratings_restaurant_id ON Ratings(RestaurantID);

-- Create composite index on Restaurants table
CREATE INDEX idx_restaurant_name_price ON Restaurants(Name, PriceRange);
EXPLAIN ANALYZE
SELECT
    r.Name,
    r.PriceRange,
    AVG(ra.Score) AS AverageRating
FROM
    Restaurants r
JOIN
    Ratings ra ON r.RestaurantID = ra.RestaurantID
GROUP BY
    r.RestaurantID, r.Name, r.PriceRange;
DROP INDEX idx_restaurant_id ON Restaurants;
DROP INDEX idx_ratings_restaurant_id ON Ratings;
DROP INDEX idx_restaurant_name_price ON Restaurants;
```

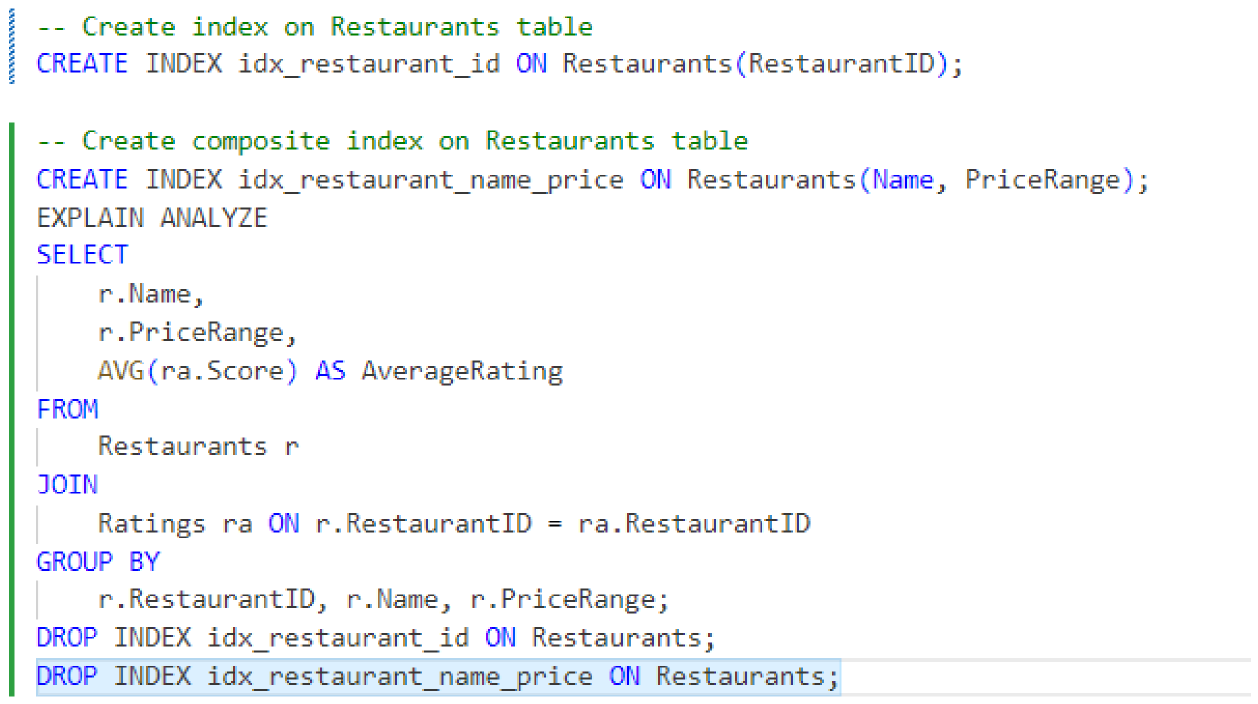
```
-> Table scan on <temporary>  (actual time=1133..1158 rows=63469 loops=1)
    -> Aggregate using temporary table  (actual time=1133..1133 rows=63468 
loops=1)
        -> Nested loop inner join  (cost=27626 rows=61123) (actual 
time=2.08..289 rows=63469 loops=1)
            -> Covering index scan on r using idx_restaurant_name_price 
(cost=6233 rows=61123) (actual time=0.791..110 rows=63469 loops=1)
            -> Single-row index lookup on ra using PRIMARY 
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual 
time=0.00262..0.00265 rows=1 loops=63469)
```

Analysis of Query 3: Indices on Restaurants.RestaurantID, Ratings.RestaurantID, Restaurants.Name, Restaurants.PriceRange

We implemented indices on Restaurants.RestaurantID, Ratings.RestaurantID, Restaurants.Name, Restaurants.PriceRange to optimize the performance involving the aggregation and joins. The goal of indexing these was to facilitate data retrieval and aggregation operations by having more efficient index scans. The query itself functions by performing aggregations and joins on Restaurants and Ratings.

The indexes on Restaurants.RestaurantID and Ratings.RestaurantID allow for efficient single row lookups in the nested loop joins. It contributes to faster data retrieval and aggregation. The index scan using idx_restaurant_name_price creates an efficient process to retrieve relevant rows based on Name and PriceRange. This improves the query performance. The efficiency gains observed validated the selected indices because they were able to minimize costs and improve execution time.

Query 3: Indices on Restaurants.RestaurantID, Restaurants.Name



```sql
-- Create index on Restaurants table
CREATE INDEX idx_restaurant_id ON Restaurants(RestaurantID);

-- Create composite index on Restaurants table
CREATE INDEX idx_restaurant_name_price ON Restaurants(Name, PriceRange);
EXPLAIN ANALYZE
SELECT
    r.Name,
    r.PriceRange,
    AVG(ra.Score) AS AverageRating
FROM
    Restaurants r
JOIN
    Ratings ra ON r.RestaurantID = ra.RestaurantID
GROUP BY
    r.RestaurantID, r.Name, r.PriceRange;
DROP INDEX idx_restaurant_id ON Restaurants;
DROP INDEX idx_restaurant_name_price ON Restaurants;
```

```
-> Table scan on <temporary>  (actual time=724..757 rows=63469 loops=1)
    -> Aggregate using temporary table  (actual time=724..724 rows=63468
loops=1)
        -> Nested loop inner join  (cost=27626 rows=61123) (actual
time=0.409..237 rows=63469 loops=1)
```

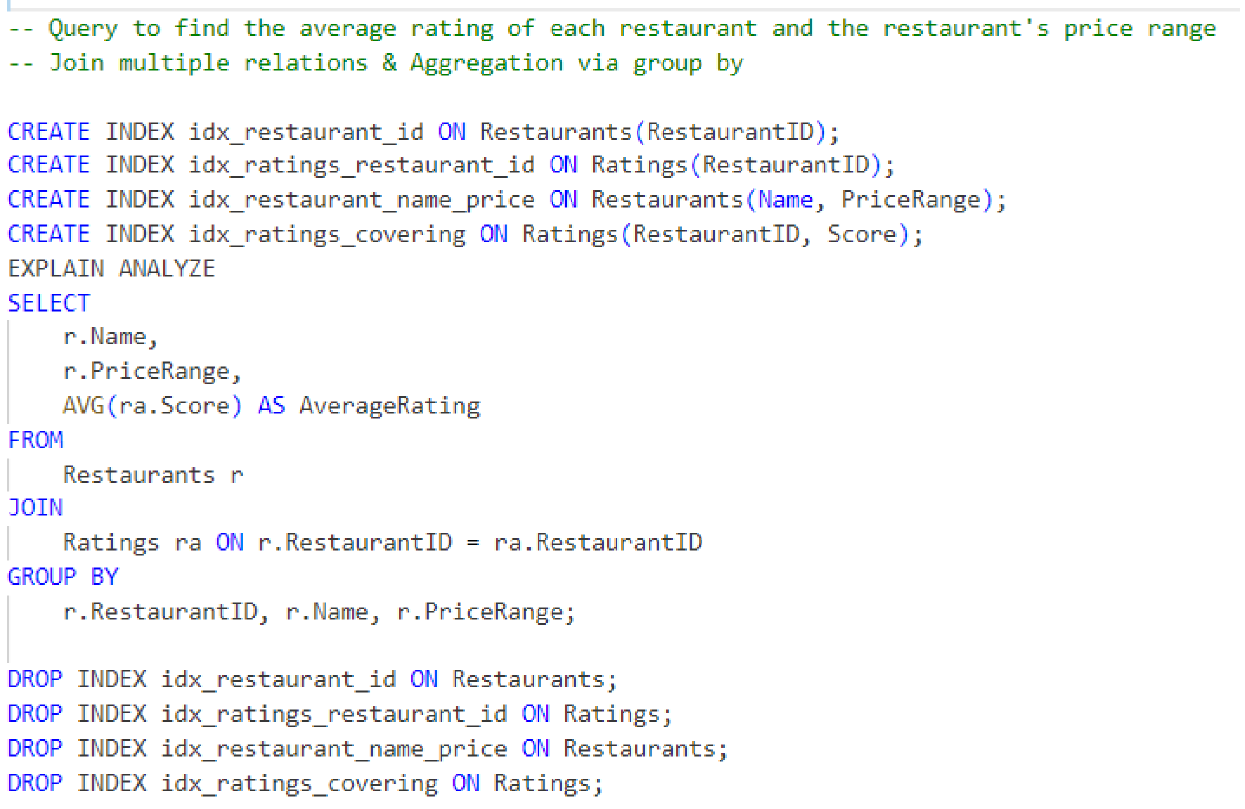
```
            -> Covering index scan on r using idx_restaurant_name_price
(cost=6233 rows=61123) (actual time=0.399..86.8 rows=63469 loops=1)
            -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual
time=0.00218..0.00221 rows=1 loops=63469)
```

<u>Analysis of Query 3: Indices on Restaurants.RestaurantID, Restaurants.Name</u>

We chose indexes on Restaurants.RestaurantID and Restaurants.Name to optimize the performance of aggregations and joins. The indexes were chosen based on their relevance ot the filtering and join conditions and we hoped it would improve efficiency.

The query involved table scans, aggregation utilizing temporary tables, and nested loop joins. The aggregation was facilitated by the index on Restaurants.RestaurantID and Restaurants.Name and allowed for a more efficient data retrieval. The covering index scan effectively uses the index on Restaurants.Name and allows for a more optimal retrieval of rows. These indexes were chosen to address the filtering and join requirements and to minimize the data access costs. The improvements in the nested loop join allowed for an overall improvement in the query performance.

<u>Query 3: Create a covering index: include all the data required for the query without having to access the table</u>



```
-- Query to find the average rating of each restaurant and the restaurant's price range
-- Join multiple relations & Aggregation via group by

CREATE INDEX idx_restaurant_id ON Restaurants(RestaurantID);
CREATE INDEX idx_ratings_restaurant_id ON Ratings(RestaurantID);
CREATE INDEX idx_restaurant_name_price ON Restaurants(Name, PriceRange);
CREATE INDEX idx_ratings_covering ON Ratings(RestaurantID, Score);
EXPLAIN ANALYZE
SELECT
    r.Name,
    r.PriceRange,
    AVG(ra.Score) AS AverageRating
FROM
    Restaurants r
JOIN
    Ratings ra ON r.RestaurantID = ra.RestaurantID
GROUP BY
    r.RestaurantID, r.Name, r.PriceRange;

DROP INDEX idx_restaurant_id ON Restaurants;
DROP INDEX idx_ratings_restaurant_id ON Ratings;
DROP INDEX idx_restaurant_name_price ON Restaurants;
DROP INDEX idx_ratings_covering ON Ratings;
```

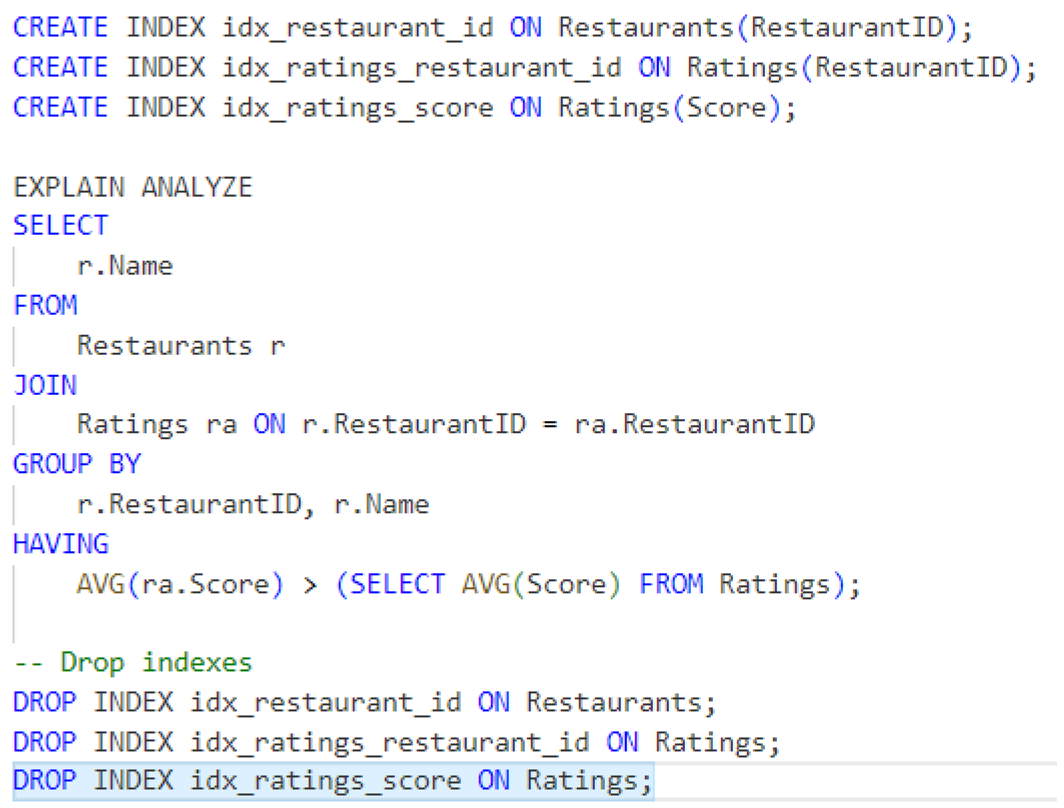
```
-> Table scan on <temporary>  (actual time=616..643 rows=63469 loops=1)
    -> Aggregate using temporary table  (actual time=616..616 rows=63468
loops=1)
        -> Nested loop inner join  (cost=27626 rows=61123) (actual
time=0.038..201 rows=63469 loops=1)
            -> Covering index scan on r using idx_restaurant_name_price
(cost=6233 rows=61123) (actual time=0.0286..80.9 rows=63469 loops=1)
            -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual
time=0.00173..0.00175 rows=1 loops=63469)
```

Analysis of Query 3: Create a covering index: include all the data required for the query without having to access the table

We chose to have a cover index to include all of the necessary data for the query without requiring access to the table itself. This indexing strategy was used in an attempt to optimize query performance by minimizing disk I/O and utilizing efficient data retrieval directly from the index. This query involves aggregation and nested loop joins.

The covering index scan on Restaurants using idx_restaurant_name_price includes all required data without accessing the table. This minimizes disk I/O and improves the query performance by reducing the need for additional data fetches during nested loop joins. The covering index scan optimized data retrieval by directly accessing necessary columns from the index which eliminates the overhead that is associated with accessing the table. By including the required data in the covering index the data was able to result in faster data retrieval.

Query 4: Apply indices on Restaurants.RestaurantID, Ratings.RestaurantID, RatingsScore



```sql
CREATE INDEX idx_restaurant_id ON Restaurants(RestaurantID);
CREATE INDEX idx_ratings_restaurant_id ON Ratings(RestaurantID);
CREATE INDEX idx_ratings_score ON Ratings(Score);

EXPLAIN ANALYZE
SELECT
    r.Name
FROM
    Restaurants r
JOIN
    Ratings ra ON r.RestaurantID = ra.RestaurantID
GROUP BY
    r.RestaurantID, r.Name
HAVING
    AVG(ra.Score) > (SELECT AVG(Score) FROM Ratings);

-- Drop indexes
DROP INDEX idx_restaurant_id ON Restaurants;
DROP INDEX idx_ratings_restaurant_id ON Ratings;
DROP INDEX idx_ratings_score ON Ratings;
```

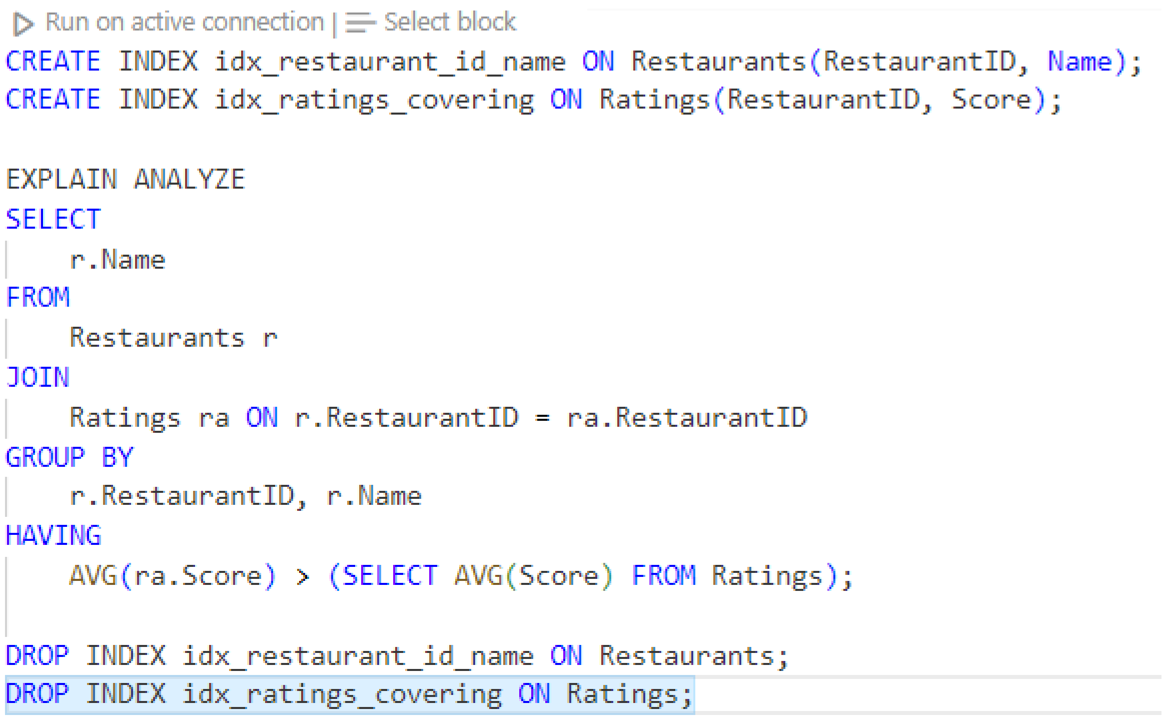
```
-> Filter: (avg(ra.Score) > (select #2))  (cost=33738 rows=61123) (actual
time=23.6..180 rows=35281 loops=1)
    -> Group aggregate: avg(ra.Score)  (cost=33738 rows=61123) (actual
time=0.075..147 rows=63469 loops=1)
        -> Nested loop inner join  (cost=27626 rows=61123) (actual
time=0.0671..127 rows=63469 loops=1)
            -> Covering index scan on r using idx_restaurant_id_name
(cost=6233 rows=61123) (actual time=0.0594..50.8 rows=63469 loops=1)
            -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual
time=0.00106..0.00108 rows=1 loops=63469)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(ratings.Score)  (cost=12765 rows=1) (actual
time=23.3..23.3 rows=1 loops=1)
            -> Covering index scan on Ratings using idx_ratings_score
(cost=6403 rows=63624) (actual time=0.0149..20.3 rows=63469 loops=1)
```

Analysis of Query 4: Apply indices on Restaurants.RestaurantID, Ratings.RestaurantID, RatingsScore

The indexing strategy we implemented for the restaurant rating query aimed to optimize the performance by creating three specific indexes: a composite index on the Restaurants table covering RestaurantID and Name (idx_restaurant_id_name), and two indexes on the Ratings table - one on RestaurantID (idx_ratings_restaurant_id) and another on Score (idx_ratings_score). This approach was designed to address the main operations in the query: the join between Restaurants and Ratings, the grouping by restaurant, and the score calculations. Upon analyzing the query execution plan after applying these indexes, we observed some improvements in certain aspects of the query execution, but the overall cost remained largely unchanged, indicating that the query's complexity presents challenges that go beyond what can be solved by indexing alone.

Looking at the main query join operations, we see nested loop join's cost remained constant at 27626, suggesting that the fundamental complexity of joining the tables wasn't reduced. However, a notable improvement occurred in the scan of the Restaurants table. The original query showed a full table scan with cost of 6233, but after indexing, this was replaced by a covering index scan. This change allowed the database to retrieve the necessary information (RestaurantID and Name) directly from the index without accessing the table data. The fact that the cost remained the same despite this change can indicate that the cost estimation does not capture the benefits of avoiding table data access or the number of rows being processed is the dominant factor in this cost.

Query 4: Covering Indexes on Restaurants.RestaurantID, Restaurants.Name, Ratings.RestaurantID, Ratings.Score



```
Run on active connection | Select block
CREATE INDEX idx_restaurant_id_name ON Restaurants(RestaurantID, Name);
CREATE INDEX idx_ratings_covering ON Ratings(RestaurantID, Score);

EXPLAIN ANALYZE
SELECT
    r.Name
FROM
    Restaurants r
JOIN
    Ratings ra ON r.RestaurantID = ra.RestaurantID
GROUP BY
    r.RestaurantID, r.Name
HAVING
    AVG(ra.Score) > (SELECT AVG(Score) FROM Ratings);

DROP INDEX idx_restaurant_id_name ON Restaurants;
DROP INDEX idx_ratings_covering ON Ratings;
```

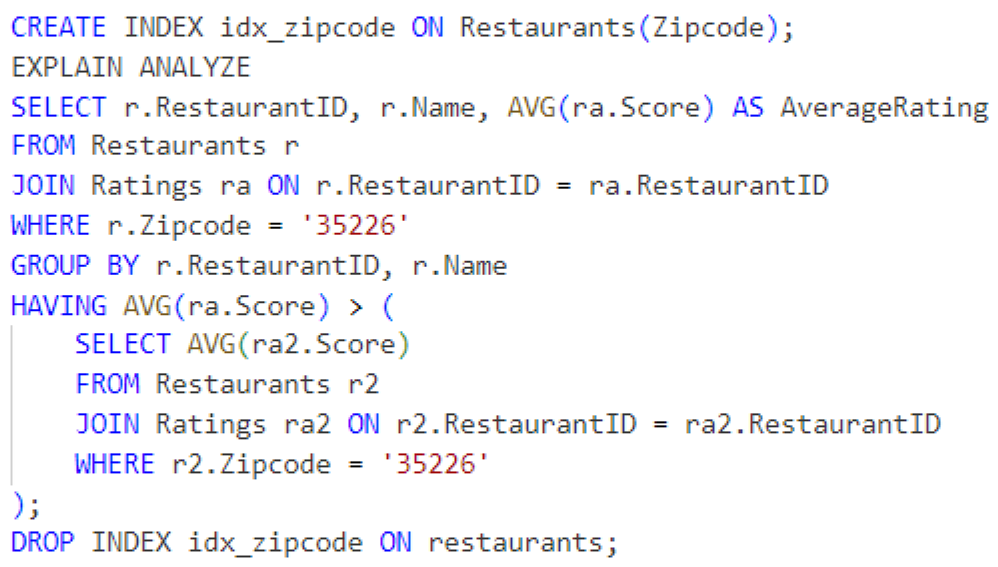
```
-> Filter: (avg(ra.Score) > (select #2))  (cost=33738 rows=61123) (actual
time=17.5..162 rows=35281 loops=1)
    -> Group aggregate: avg(ra.Score)  (cost=33738 rows=61123) (actual
time=0.451..137 rows=63469 loops=1)
        -> Nested loop inner join  (cost=27626 rows=61123) (actual
time=0.439..120 rows=63469 loops=1)
            -> Covering index scan on r using idx_restaurant_id_name
(cost=6233 rows=61123) (actual time=0.426..54.4 rows=63469 loops=1)
            -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual
time=911e-6..927e-6 rows=1 loops=63469)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(ratings.Score)  (cost=12765 rows=1) (actual
time=17..17 rows=1 loops=1)
            -> Covering index scan on Ratings using
idx_ratings_restaurant_score  (cost=6403 rows=63624) (actual
time=0.0122..14.3 rows=63469 loops=1)
```

Analysis of Query 4: Covering Indexes on Restaurants.RestaurantID, Restaurants.Name, Ratings.RestaurantID, Ratings.Score

The aim of this index was to try out a different composition of indices to see if we could achieve further improvements. We added the Ratings.RestaurantID to our original index to further enhance the efficiency of the aggregation step between the Restaurants and Ratings table, and hence speed the look-up time of the entries present.

The covering indexes on Restaurants.RestaurantID, Restaurants.Name, Ratings.RestaurantID, Ratings.Score optimized data retrieval and aggregations operations. It allows for reduced execution times for nested loop joins and demonstrates the increased efficiency from using covering indexes. By using the covering index scan we were able to eliminate the need for additional table access during the runtime of the query. By having efficient utilization of the covering indexes the query has better data access and resulted in a better query execution across large datasets.

<u>Query 5: Index on Restaurants.Zipcode</u>



```
CREATE INDEX idx_zipcode ON Restaurants(Zipcode);
EXPLAIN ANALYZE
SELECT r.RestaurantID, r.Name, AVG(ra.Score) AS AverageRating
FROM Restaurants r
JOIN Ratings ra ON r.RestaurantID = ra.RestaurantID
WHERE r.Zipcode = '35226'
GROUP BY r.RestaurantID, r.Name
HAVING AVG(ra.Score) > (
    SELECT AVG(ra2.Score)
    FROM Restaurants r2
    JOIN Ratings ra2 ON r2.RestaurantID = ra2.RestaurantID
    WHERE r2.Zipcode = '35226'
);
DROP INDEX idx_zipcode ON restaurants;
```

```
-> Filter: (??? > (select #2))  (actual time=0.237..0.239 rows=9 loops=1)
    -> Table scan on <temporary>  (actual time=0.18..0.182 rows=14
loops=1)
        -> Aggregate using temporary table  (actual time=0.179..0.179
rows=14 loops=1)
            -> Nested loop inner join  (cost=9.8 rows=14) (actual
time=0.0632..0.0888 rows=14 loops=1)
```

```
                -> Index lookup on r using idx_restaurants_zipcode
(ZipCode='35226')  (cost=4.9 rows=14) (actual time=0.0557..0.0609 rows=14
loops=1)
                -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.257 rows=1) (actual
time=0.00173..0.00176 rows=1 loops=14)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(ra2.Score)  (cost=8.84 rows=1) (actual
time=0.0429..0.043 rows=1 loops=1)
            -> Nested loop inner join  (cost=7.44 rows=14) (actual
time=0.0267..0.039 rows=14 loops=1)
                -> Covering index lookup on r2 using
idx_restaurants_zipcode (ZipCode='35226')  (cost=2.54 rows=14) (actual
time=0.024..0.0265 rows=14 loops=1)
                -> Single-row index lookup on ra2 using PRIMARY
(RestaurantID=r2.RestaurantID)  (cost=0.257 rows=1) (actual
time=786e-6..807e-6 rows=1 loops=14)


time=0.00167..0.00168 rows=1 loops=14)
```

Analysis of Query 5: Index on Restaurants.Zipcode

We implemented an index on Restaurants.Zipcode to optimize the performance involving wheres and joins. The goal of indexing these was to facilitate data retrieval and aggregation operations by having more efficient index scans.

Adding an index on Restaurants.Zipcode had significantly reduced the cost of nested inner joins from 8372 to 9.8 and improved our query performance. This is because by indexing, we are reducing the number of rows required to be joined by only accessing the restaurants in a particular zipcode. We need not filter the large dataset to get relevant data as we are directly accessing the restaurants in the relevant zipcode and making comparisons only using those records.

The difference in performance between the original query vs the modified query with the index is vast as the modified query has a much lower cost and faster execution time compared to the original query. This optimization betters the system's performance and delivers quicker results.

Query 5: Index on Ratings.RestaurantID

```sql
CREATE INDEX idx_ratings_ID ON Ratings(RestaurantID);
EXPLAIN ANALYZE
SELECT r.RestaurantID, r.Name, AVG(ra.Score) AS AverageRating
FROM Restaurants r
JOIN Ratings ra ON r.RestaurantID = ra.RestaurantID
WHERE r.Zipcode = '35226'
GROUP BY r.RestaurantID, r.Name
HAVING AVG(ra.Score) > (
    SELECT AVG(ra2.Score)
    FROM Restaurants r2
    JOIN Ratings ra2 ON r2.RestaurantID = ra2.RestaurantID
    WHERE r2.Zipcode = '35226'
);
DROP INDEX idx_ratings_ID ON Ratings;
```

```
-> Filter: (??? > (select #2))  (actual time=36.3..36.3 rows=9 loops=1)
    -> Table scan on <temporary>  (actual time=18.7..18.7 rows=14 loops=1)
        -> Aggregate using temporary table  (actual time=18.7..18.7
rows=14 loops=1)
            -> Nested loop inner join  (cost=8372 rows=6112) (actual
time=0.0727..18.7 rows=14 loops=1)
                -> Filter: (r.ZipCode = '35226')  (cost=6233 rows=6112)
(actual time=0.0621..18.6 rows=14 loops=1)
                    -> Table scan on r  (cost=6233 rows=61123) (actual
time=0.0574..15.8 rows=63469 loops=1)
                -> Single-row index lookup on ra using PRIMARY
(RestaurantID=r.RestaurantID)  (cost=0.25 rows=1) (actual
time=0.00179..0.00179 rows=1 loops=14)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(ra2.Score)  (cost=8983 rows=1) (actual
time=17.6..17.6 rows=1 loops=1)
            -> Nested loop inner join  (cost=8372 rows=6112) (actual
time=0.0446..17.6 rows=14 loops=1)
                -> Filter: (r2.ZipCode = '35226')  (cost=6233 rows=6112)
(actual time=0.0326..17.6 rows=14 loops=1)
                    -> Table scan on r2  (cost=6233 rows=61123) (actual
time=0.0309..14.2 rows=63469 loops=1)
                -> Single-row index lookup on ra2 using PRIMARY
(RestaurantID=r2.RestaurantID)  (cost=0.25 rows=1) (actual
```

Analysis of Query 5: Index on Ratings.RestaurantID

We chose an index on Ratings.RestauantID believing it may improve the aggregation on ratings. Since the cost did not go down by adding an Index on Ratings.RestaurantID it did not improve the amount of resources used. The query requires a full table scan on the restaurants table to filter by zip code then it calculates the average score of the zipcode and compares each restaurant's score to the average.

The cost output between the original query vs the query with the index on Ratings is the same. There are small differences in the execution time which could be due to improvement in the subquery execution and main nested loop join. The index could allow for better caching as the data would already be in memory during the second execution and the database could have used statistics from the first execution to optimize the second. The cost for the query may still be the same due to Restaurants not having a rating thus no noticeable difference in computing the average.

When reviewing our data, Restaurants and Ratings have 63,469 rows so each score and rating has an associated restaurant. The query does not access more data beyond 2 columns which may be a reason why adding indexes did not improve cost performance as there was low cost in general. If we were to add an additional column to our query like where rating> avg(rating), adding an index may improve in that situation as there are two aggregates and 3 columns.