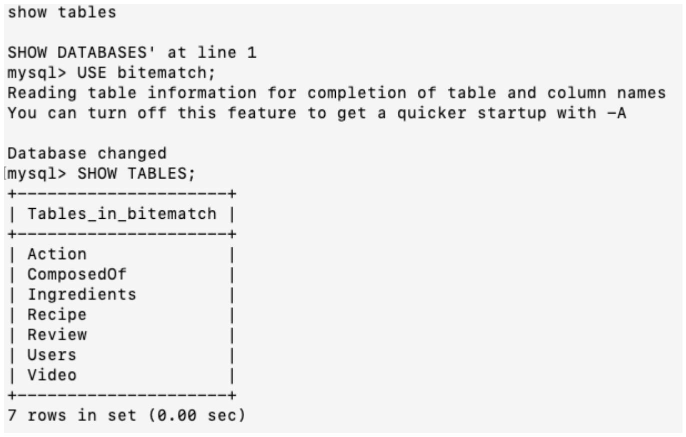
Stage 3: Database Implementation and Indexing

Table Creation DDL Commands:

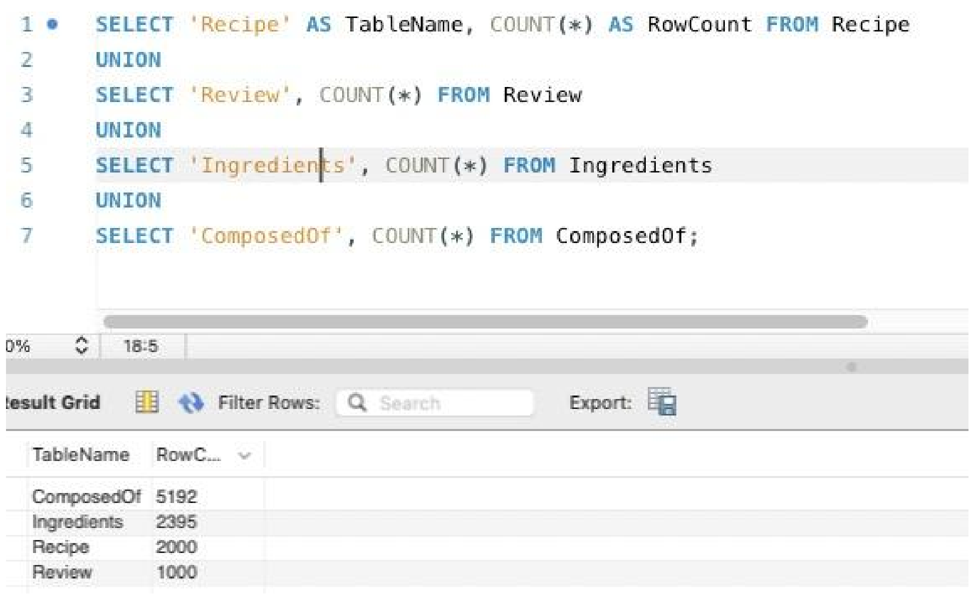
```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Username VARCHAR(255) NOT NULL,
    Email VARCHAR(255) NOT NULL
);

CREATE TABLE Recipe (
    RecipeID INT PRIMARY KEY,
    RecipeTitle VARCHAR(255) NOT NULL,
    Directions TEXT,
    Source VARCHAR(255)
);

CREATE TABLE Ingredients (
    IngredientID INT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Calories DECIMAL(10,2)
);

CREATE TABLE Video (
    VideoID INT PRIMARY KEY,
    Title VARCHAR(255) NOT NULL,
    Link VARCHAR(255) NOT NULL,
    RecipeID INT,
    FOREIGN KEY (RecipeID) REFERENCES Recipe(RecipeID)
);

CREATE TABLE ComposedOf (
    RecipeID INT,
    IngredientID INT,
    Quantity VARCHAR(255),
    PRIMARY KEY (RecipeID, IngredientID),
    FOREIGN KEY (RecipeID) REFERENCES Recipe(RecipeID),
    FOREIGN KEY (IngredientID) REFERENCES Ingredients(IngredientID)
);

CREATE TABLE Review (
    ReviewID INT PRIMARY KEY,
    RecipeID INT,
    UserID INT,
    Stars INT,
    Comments VARCHAR(1000),
    FOREIGN KEY (RecipeID) REFERENCES Recipe(RecipeID),
    FOREIGN KEY (UserID) REFERENCES Users(UserID)
);

CREATE TABLE Action (
    ActionType VARCHAR(255),
    Time TIMESTAMP,
    UserID INT,
    RecipeID INT,
    FOREIGN KEY (UserID) REFERENCES Users(UserID),
    FOREIGN KEY (RecipeID) REFERENCES Recipe(RecipeID)
);
```

Terminal information



```
show tables

SHOW DATABASES' at line 1
mysql> USE bitematch;
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_bitematch |
+---------------------+
| Action              |
| ComposedOf          |
| Ingredients         |
| Recipe              |
| Review              |
| Users               |
| Video               |
+---------------------+
7 rows in set (0.00 sec)
```

Count query: insert at least 1000 rows



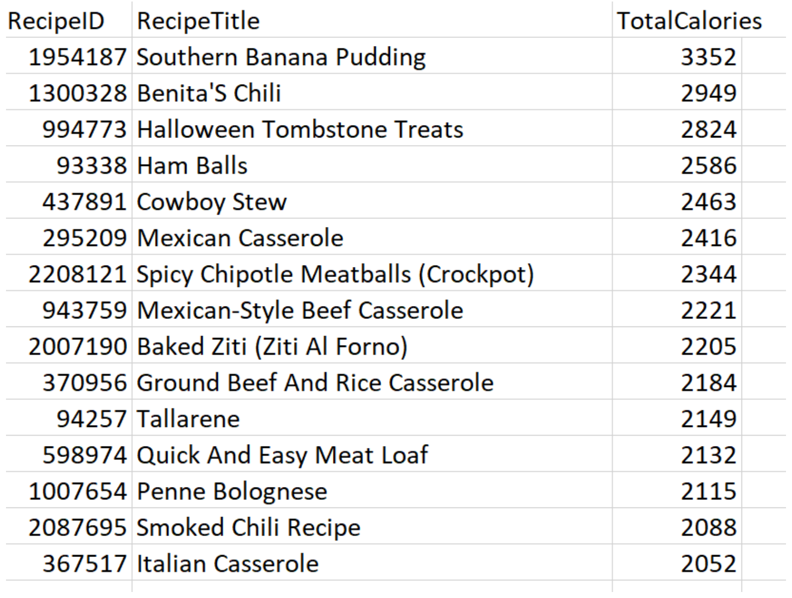
```sql
SELECT 'Recipe' AS TableName, COUNT(*) AS RowCount FROM Recipe
UNION
SELECT 'Review', COUNT(*) FROM Review
UNION
SELECT 'Ingredients', COUNT(*) FROM Ingredients
UNION
SELECT 'ComposedOf', COUNT(*) FROM ComposedOf;
```

```
TableName    RowC...
ComposedOf   5192
Ingredients  2395
Recipe       2000
Review       1000
```

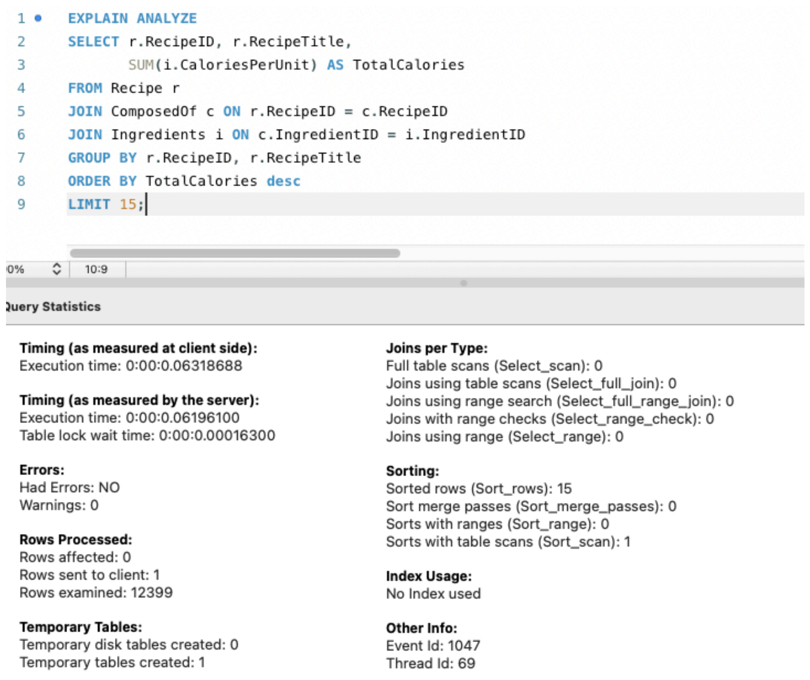
Advanced Query

1. Calculate the total calories of each recipe based on its ingredients

```sql
SELECT r.RecipeID, r.RecipeTitle,
    SUM(i.Calories) AS TotalCalories
FROM Recipe r
JOIN ComposedOf c ON r.RecipeID = c.RecipeID
JOIN Ingredients i ON c.IngredientID = i.IngredientID
GROUP BY r.RecipeID, r.RecipeTitle;
```



RecipeID	RecipeTitle	TotalCalories
1954187	Southern Banana Pudding	3352
1300328	Benita'S Chili	2949
994773	Halloween Tombstone Treats	2824
93338	Ham Balls	2586
437891	Cowboy Stew	2463
295209	Mexican Casserole	2416
2208121	Spicy Chipotle Meatballs (Crockpot)	2344
943759	Mexican-Style Beef Casserole	2221
2007190	Baked Ziti (Ziti Al Forno)	2205
370956	Ground Beef And Rice Casserole	2184
94257	Tallarene	2149
598974	Quick And Easy Meat Loaf	2132
1007654	Penne Bolognese	2115
2087695	Smoked Chili Recipe	2088
367517	Italian Casserole	2052



```
EXPLAIN ANALYZE
SELECT r.RecipeID, r.RecipeTitle,
       SUM(i.CaloriesPerUnit) AS TotalCalories
FROM Recipe r
JOIN ComposedOf c ON r.RecipeID = c.RecipeID
JOIN Ingredients i ON c.IngredientID = i.IngredientID
GROUP BY r.RecipeID, r.RecipeTitle
ORDER BY TotalCalories desc
LIMIT 15;
```

uery Statistics

Timing (as measured at client side):
Execution time: 0:00:0.06318688

Timing (as measured by the server):
Execution time: 0:00:0.06196100
Table lock wait time: 0:00:0.00016300

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1
Rows examined: 12399

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 1

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 15
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 1

Index Usage:
No Index used

Other Info:
Event Id: 1047
Thread Id: 69

Index	
1	CREATE INDEX idx_composed_recipeid ON ComposedOf(RecipeID);
2	CREATE INDEX idx_composed_ingredientid ON ComposedOf(IngredientID);
3	CREATE INDEX idx_composed_both ON ComposedOf(RecipeID, IngredientID);

To optimize the query calculating total calories per recipe, we tested multiple indexing strategies using EXPLAIN ANALYZE. Without any index, the estimated cost was **109572.55** due to full table scans. Adding an index on ComposedOf(RecipeID) significantly improved performance, reducing the estimated cost to **4142.53**, as the query planner adopted index-based nested loop joins. A composite index on (RecipeID, IngredientID) also worked but had a slightly higher cost of **4557.87**, while indexing IngredientID alone resulted in no improvement (**109572.55**). We selected the index on RecipeID as the final choice, as it provided the most substantial reduction in query cost with minimal complexity.

Analysis Reports

Index	Indexing Analysis
base	-> Limit: 15 row(s) (actual time=30.342..30.387 rows=15 loops=1)

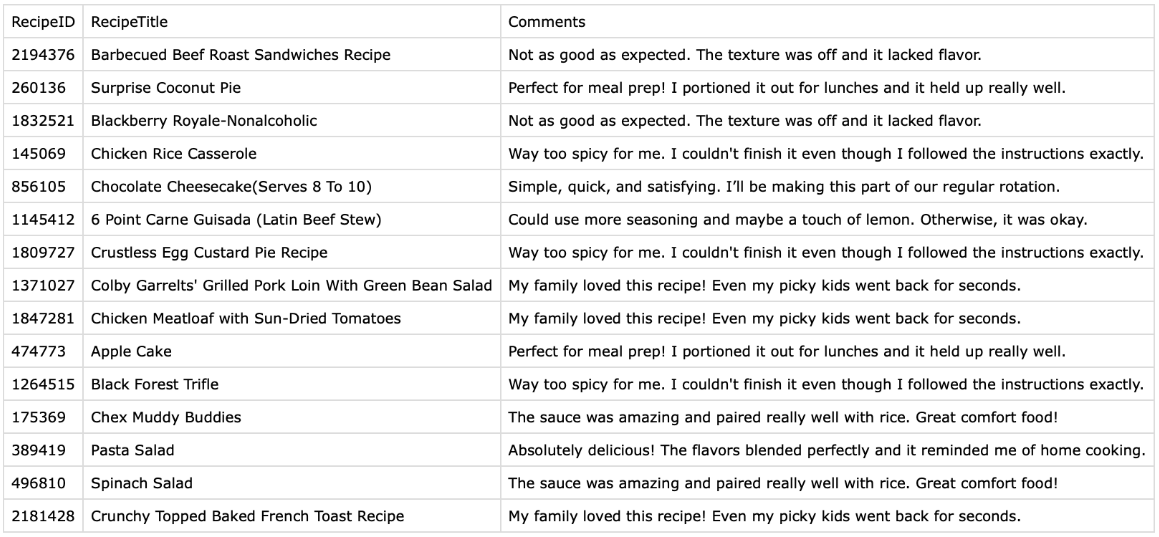
	-> Sort: TotalCalories DESC, limit input to 15 row(s) per chunk (actual time=30.341..30.386 rows=15 loops=1) -> Table scan on <temporary> (actual time=29.018..29.220 rows=1771 loops=1) -> Aggregate using temporary table (actual time=29.017..29.017 rows=1771 loops=1) -> Nested loop inner join (cost=109572.55 rows=99251) (actual time=5.714..18.025 rows=5192 loops=1) -> Inner hash join (c.RecipeID = r.RecipeID) (cost=99520.36 rows=99251) (actual time=5.538..10.812 rows=5192 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.05 rows=508) (actual time=0.055..3.739 rows=5192 loops=1) -> Table scan on c (cost=0.05 rows=5082) (actual time=0.053..3.336 rows=5192 loops=1) -> Hash -> Table scan on r (cost=219.55 rows=1953) (actual time=0.159..3.908 rows=2000 loops=1) -> Filter: (c.IngredientID = i.IngredientID) (cost=0.00 rows=1) (actual time=0.001..0.001 rows=1 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.00 rows=1) (actual time=0.001..0.001 rows=1 loops=5192)
1	-> Limit: 15 row(s) (actual time=23.449..23.451 rows=15 loops=1) -> Sort: TotalCalories DESC, limit input to 15 row(s) per chunk (actual time=23.448..23.450 rows=15 loops=1) -> Table scan on <temporary> (actual time=22.492..22.639 rows=1771 loops=1) -> Aggregate using temporary table (actual time=22.491..22.491 rows=1771 loops=1) -> Nested loop inner join (cost=4142.53 rows=5604) (actual time=0.204..14.585 rows=5192 loops=1) -> Nested loop inner join (cost=2181.04 rows=5604) (actual time=0.172..9.522 rows=5192 loops=1) -> Filter: (r.RecipeID is not null) (cost=219.55 rows=1953) (actual time=0.083..1.623 rows=2000 loops=1) -> Table scan on r (cost=219.55 rows=1953) (actual time=0.082..1.506 rows=2000 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.72 rows=3) (actual time=0.002..0.004 rows=3 loops=2000)

	-> Index lookup on c using idx_composed_recipeid (RecipeID=r.RecipeID) (cost=0.72 rows=3) (actual time=0.002..0.003 rows=3 loops=2000) -> Filter: (c.IngredientID = i.IngredientID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=5192)
2	-> Limit: 15 row(s) (actual time=31.237..31.240 rows=15 loops=1) -> Sort: TotalCalories DESC, limit input to 15 row(s) per chunk (actual time=31.236..31.238 rows=15 loops=1) -> Table scan on <temporary> (actual time=30.607..30.790 rows=1771 loops=1) -> Aggregate using temporary table (actual time=30.605..30.605 rows=1771 loops=1) -> Nested loop inner join (cost=109572.55 rows=99251) (actual time=4.968..20.255 rows=5192 loops=1) -> Inner hash join (c.RecipeID = r.RecipeID) (cost=99520.36 rows=99251) (actual time=4.903..11.436 rows=5192 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.05 rows=508) (actual time=0.018..4.542 rows=5192 loops=1) -> Table scan on c (cost=0.05 rows=5082) (actual time=0.017..4.082 rows=5192 loops=1) -> Hash -> Table scan on r (cost=219.55 rows=1953) (actual time=0.088..3.666 rows=2000 loops=1) -> Filter: (c.IngredientID = i.IngredientID) (cost=0.00 rows=1) (actual time=0.001..0.001 rows=1 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.00 rows=1) (actual time=0.001..0.001 rows=1 loops=5192)
3	-> Limit: 15 row(s) (actual time=37.568..37.578 rows=15 loops=1) -> Sort: TotalCalories DESC, limit input to 15 row(s) per chunk (actual time=37.567..37.576 rows=15 loops=1) -> Table scan on <temporary> (actual time=36.473..36.626 rows=1771 loops=1) -> Aggregate using temporary table (actual time=36.472..36.472 rows=1771 loops=1)

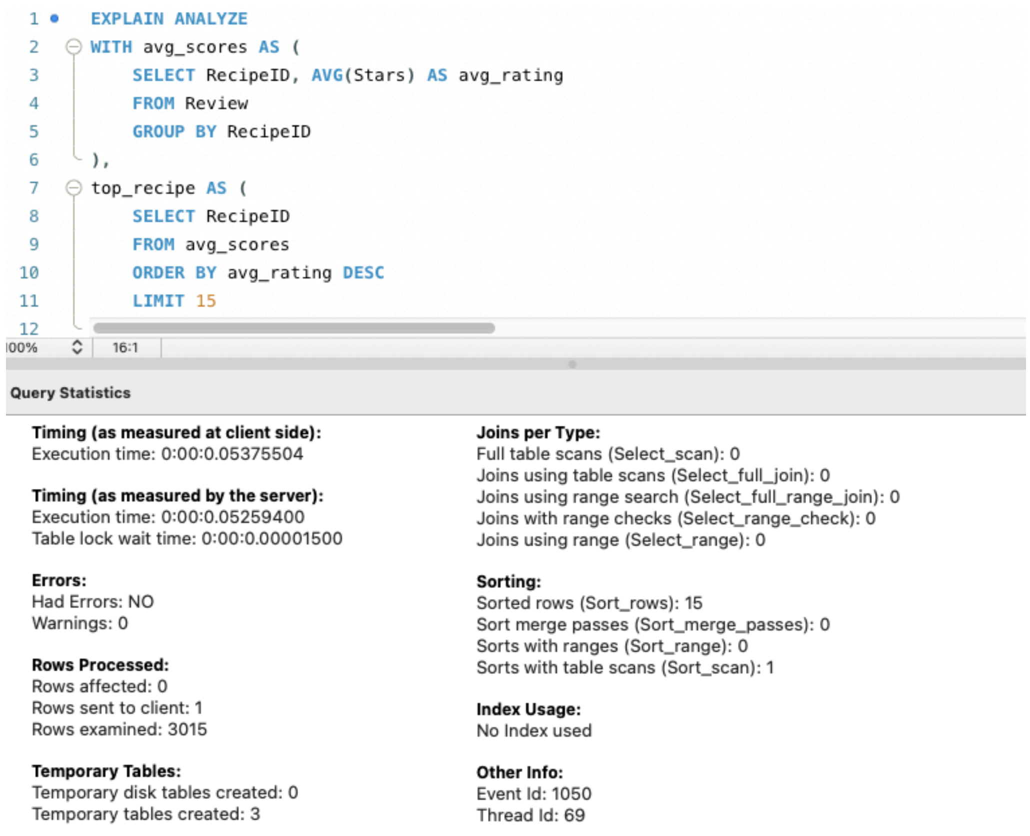
	-> Nested loop inner join (cost=4557.87 rows=5604) (actual time=0.192..24.806 rows=5192 loops=1) -> Nested loop inner join (cost=2596.38 rows=5604) (actual time=0.122..15.273 rows=5192 loops=1) -> Filter: (r.RecipeID is not null) (cost=219.55 rows=1953) (actual time=0.082..3.591 rows=2000 loops=1) -> Table scan on r (cost=219.55 rows=1953) (actual time=0.081..3.388 rows=2000 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.93 rows=3) (actual time=0.003..0.005 rows=3 loops=2000) -> Covering index lookup on c using idx_composed_both (RecipeID=r.RecipeID) (cost=0.93 rows=3) (actual time=0.003..0.005 rows=3 loops=2000) -> Filter: (c.IngredientID = i.IngredientID) (cost=0.25 rows=1) (actual time=0.001..0.002 rows=1 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=5192)

2.Find the recipe with the highest average rating and display all its reviews along with the recipe name

```
WITH avg_scores AS (
   SELECT RecipeID, AVG(Stars) AS avg_rating
   FROM Review
   GROUP BY RecipeID
),
top_recipe AS (
   SELECT RecipeID
   FROM avg_scores
   ORDER BY avg_rating DESC
   LIMIT 15
)
SELECT r.RecipeID, s.RecipeTitle, rv.Comments
FROM top_recipe r
JOIN Recipe s ON r.RecipeID = s.RecipeID
JOIN Review rv ON r.RecipeID = rv.RecipeID;
```



RecipeID	RecipeTitle	Comments
2194376	Barbecued Beef Roast Sandwiches Recipe	Not as good as expected. The texture was off and it lacked flavor.
260136	Surprise Coconut Pie	Perfect for meal prep! I portioned it out for lunches and it held up really well.
1832521	Blackberry Royale-Nonalcoholic	Not as good as expected. The texture was off and it lacked flavor.
145069	Chicken Rice Casserole	Way too spicy for me. I couldn't finish it even though I followed the instructions exactly.
856105	Chocolate Cheesecake(Serves 8 To 10)	Simple, quick, and satisfying. I'll be making this part of our regular rotation.
1145412	6 Point Carne Guisada (Latin Beef Stew)	Could use more seasoning and maybe a touch of lemon. Otherwise, it was okay.
1809727	Crustless Egg Custard Pie Recipe	Way too spicy for me. I couldn't finish it even though I followed the instructions exactly.
1371027	Colby Garrelts' Grilled Pork Loin With Green Bean Salad	My family loved this recipe! Even my picky kids went back for seconds.
1847281	Chicken Meatloaf with Sun-Dried Tomatoes	My family loved this recipe! Even my picky kids went back for seconds.
474773	Apple Cake	Perfect for meal prep! I portioned it out for lunches and it held up really well.
1264515	Black Forest Trifle	Way too spicy for me. I couldn't finish it even though I followed the instructions exactly.
175369	Chex Muddy Buddies	The sauce was amazing and paired really well with rice. Great comfort food!
389419	Pasta Salad	Absolutely delicious! The flavors blended perfectly and it reminded me of home cooking.
496810	Spinach Salad	The sauce was amazing and paired really well with rice. Great comfort food!
2181428	Crunchy Topped Baked French Toast Recipe	My family loved this recipe! Even my picky kids went back for seconds.



```
EXPLAIN ANALYZE
WITH avg_scores AS (
    SELECT RecipeID, AVG(Stars) AS avg_rating
    FROM Review
    GROUP BY RecipeID
),
top_recipe AS (
    SELECT RecipeID
    FROM avg_scores
    ORDER BY avg_rating DESC
    LIMIT 15
```

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.05375504

Timing (as measured by the server):
Execution time: 0:00:0.05259400
Table lock wait time: 0:00:0.00001500

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1
Rows examined: 3015

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 3

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 15
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 1

Index Usage:
No Index used

Other Info:
Event Id: 1050
Thread Id: 69

Index	
1	CREATE INDEX idx_review_recipeid ON Review(RecipeID);

2	CREATE INDEX idx_recipe_recipeid ON Recipe(RecipeID);
3	CREATE INDEX idx_review_recipeid_stars ON Review(RecipeID, Stars);

To improve the performance of a query that retrieves the top 15 highest-rated recipes along with their titles and comments, we evaluated three indexing strategies on the Review and Recipe tables. Without indexing, the estimated query cost was relatively low (**cost=55.03**) due to small data size and use of hash joins. After creating an index on Review(RecipeID), the optimizer adopted index lookups, and the query was restructured into nested loop joins, with an increased estimated cost of **6051.25**. We then added an index on Recipe(RecipeID), which further optimized specific join paths, reducing the estimated cost to **952.75**. Finally, we implemented a composite index on Review(RecipeID, Stars) to support both aggregation and joins; however, the cost remained **6051.25**, indicating no additional benefit over the single-column index. Based on this analysis, we selected the index on Recipe(RecipeID) as the most efficient design, offering the lowest cost and streamlined join performance.

Index	Indexing Analysis

base	
	```
	-> Inner hash join (s.RecipeID = r.RecipeID)  (cost=55.03 rows=0) (actual time=5.055..6.549 rows=15 loops=1)
	    -> Table scan on s  (cost=0.05 rows=1953) (actual time=0.043..1.773 rows=2000 loops=1)
	    -> Hash
	        -> Inner hash join (rv.RecipeID = r.RecipeID)  (cost=5.25 rows=0) (actual time=4.309..4.347 rows=15 loops=1)
	            -> Table scan on rv  (cost=0.85 rows=1000) (actual time=0.021..0.302 rows=1000 loops=1)
	            -> Hash
	                -> Table scan on r  (cost=2.50..2.50 rows=0) (actual time=3.970..3.972 rows=15 loops=1)
	                    -> Materialize CTE top_recipe  (cost=2.50..2.50 rows=0) (actual time=3.970..3.970 rows=15 loops=1)
	                        -> Limit: 15 row(s)  (cost=0.00..0.00 rows=0) (actual time=3.927..3.928 rows=15 loops=1)
	                            -> Sort: avg_scores.avg_rating DESC, limit input to 15 row(s) per chunk  (cost=0.00..0.00 rows=0) (actual time=3.926..3.927 rows=15 loops=1)
	                                -> Table scan on avg_scores  (cost=115.00 rows=1000)
	```

	(actual time=3.689..3.752 rows=787 loops=1) -> Materialize CTE avg_scores (cost=0.00..0.00 rows=0) (actual time=3.689..3.689 rows=787 loops=1) -> Table scan on <temporary> (actual time=3.484..3.547 rows=787 loops=1) -> Aggregate using temporary table (actual time=3.483..3.483 rows=787 loops=1) -> Table scan on Review (cost=102.75 rows=1000) (actual time=2.691..3.140 rows=1000 loops=1)
1	-> Nested loop inner join (cost=6051.25 rows=37224) (actual time=5.087..18.355 rows=15 loops=1) -> Nested loop inner join (cost=1088.10 rows=2482) (actual time=0.157..12.459 rows=1000 loops=1) -> Filter: (s.RecipeID is not null) (cost=219.55 rows=1953) (actual time=0.080..3.878 rows=2000 loops=1) -> Table scan on s (cost=219.55 rows=1953) (actual time=0.078..3.634 rows=2000 loops=1) -> Index lookup on rv using idx_review_recipeid (RecipeID=s.RecipeID) (cost=0.32 rows=1) (actual time=0.003..0.004 rows=0 loops=2000) -> Covering index lookup on r using <auto_key0> (RecipeID=s.RecipeID) (actual time=0.006..0.006 rows=0 loops=1000) -> Materialize CTE top_recipe (cost=304.25..304.25 rows=15) (actual time=4.739..4.739 rows=15 loops=1) -> Limit: 15 row(s) (cost=302.75..302.75 rows=15) (actual time=4.666..4.670 rows=15 loops=1) -> Sort: avg_scores.avg_rating DESC, limit input to 15 row(s) per chunk (cost=302.75..302.75 rows=1000) (actual time=4.665..4.667 rows=15 loops=1) -> Table scan on avg_scores (cost=115.00 rows=1000) (actual time=4.046..4.247 rows=787 loops=1) -> Materialize CTE avg_scores (cost=302.75..302.75 rows=1000) (actual time=4.045..4.045 rows=787 loops=1) -> Group aggregate: avg(review.Stars) (cost=202.75 rows=1000) (actual time=0.038..3.435 rows=787 loops=1) -> Index scan on Review using idx_review_recipeid (cost=102.75 rows=1000) (actual time=0.014..2.941 rows=1000 loops=1)
2	-> Nested loop inner join (cost=952.75 rows=0) (actual time=14.142..14.943 rows=15 loops=1)

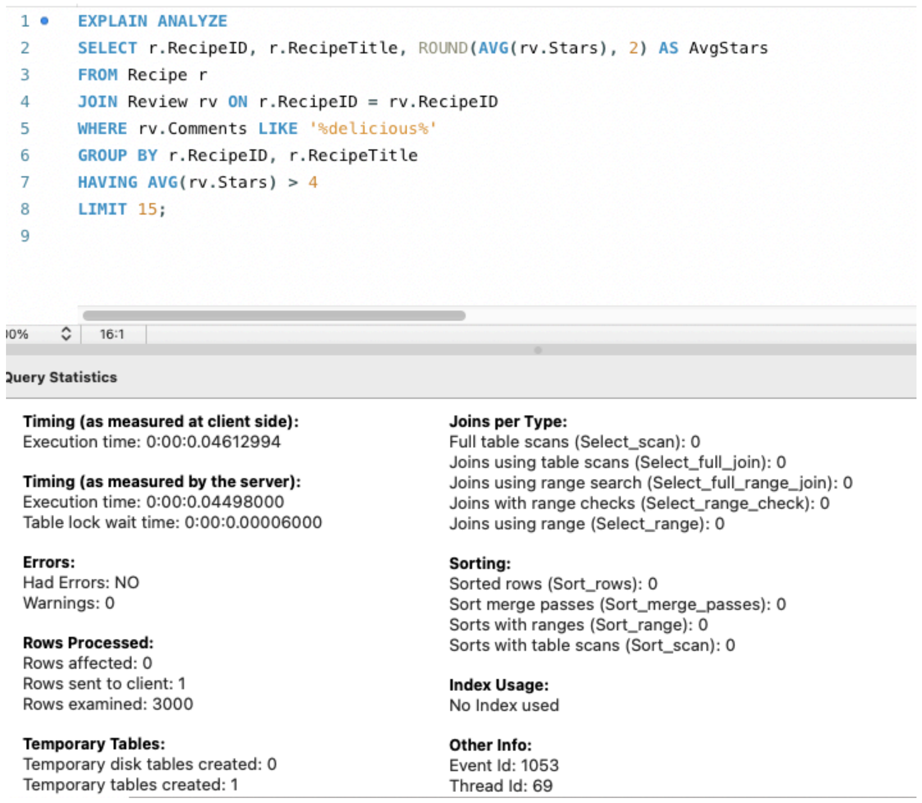
	-> Nested loop inner join (cost=452.75 rows=1000) (actual time=0.164..9.809 rows=1000 loops=1) -> Filter: (rv.RecipeID is not null) (cost=102.75 rows=1000) (actual time=0.104..1.464 rows=1000 loops=1) -> Table scan on rv (cost=102.75 rows=1000) (actual time=0.102..1.336 rows=1000 loops=1) -> Index lookup on s using idx_recipe_recipeid (RecipeID=rv.RecipeID) (cost=0.25 rows=1) (actual time=0.007..0.008 rows=1 loops=1000) -> Covering index lookup on r using <auto_key0> (RecipeID=rv.RecipeID) (actual time=0.005..0.005 rows=0 loops=1000) -> Materialize CTE top_recipe (cost=0.00..0.00 rows=0) (actual time=3.849..3.849 rows=15 loops=1) -> Limit: 15 row(s) (cost=0.00..0.00 rows=0) (actual time=3.771..3.776 rows=15 loops=1) -> Sort: avg_scores.avg_rating DESC, limit input to 15 row(s) per chunk (cost=0.00..0.00 rows=0) (actual time=3.771..3.773 rows=15 loops=1) -> Table scan on avg_scores (cost=115.00 rows=1000) (actual time=3.108..3.313 rows=787 loops=1) -> Materialize CTE avg_scores (cost=0.00..0.00 rows=0) (actual time=3.107..3.107 rows=787 loops=1) -> Table scan on <temporary> (actual time=2.314..2.513 rows=787 loops=1) -> Aggregate using temporary table (actual time=2.312..2.312 rows=787 loops=1) -> Table scan on Review (cost=102.75 rows=1000) (actual time=0.012..1.356 rows=1000 loops=1)
3	-> Nested loop inner join (cost=6051.25 rows=37224) (actual time=3.965..17.352 rows=15 loops=1) -> Nested loop inner join (cost=1088.10 rows=2482) (actual time=0.390..13.010 rows=1000 loops=1) -> Filter: (s.RecipeID is not null) (cost=219.55 rows=1953) (actual time=0.188..3.917 rows=2000 loops=1) -> Table scan on s (cost=219.55 rows=1953) (actual time=0.187..3.673 rows=2000 loops=1) -> Index lookup on rv using idx_review_recipeid_stars (RecipeID=s.RecipeID) (cost=0.32 rows=1) (actual time=0.004..0.004 rows=0 loops=2000) -> Covering index lookup on r using <auto_key0> (RecipeID=s.RecipeID) (actual time=0.004..0.004 rows=0 loops=1000)

	-> Materialize CTE top_recipe (cost=304.25..304.25 rows=15) (actual time=3.183..3.183 rows=15 loops=1) -> Limit: 15 row(s) (cost=302.75..302.75 rows=15) (actual time=3.117..3.121 rows=15 loops=1) -> Sort: avg_scores.avg_rating DESC, limit input to 15 row(s) per chunk (cost=302.75..302.75 rows=1000) (actual time=3.116..3.118 rows=15 loops=1) -> Table scan on avg_scores (cost=115.00 rows=1000) (actual time=2.415..2.612 rows=787 loops=1) -> Materialize CTE avg_scores (cost=302.75..302.75 rows=1000) (actual time=2.413..2.413 rows=787 loops=1) -> Group aggregate: avg(review.Stars) (cost=202.75 rows=1000) (actual time=0.084..1.826 rows=787 loops=1) -> Covering index scan on Review using idx_review_recipeid_stars (cost=102.75 rows=1000) (actual time=0.072..1.343 rows=1000 loops=1)

3. Find all recipes with a rating higher than 4 and containing the phrase 'delicious'

```
SELECT r.RecipeID, r.RecipeTitle, ROUND(AVG(rv.Stars), 2) AS AvgStars
FROM Recipe r
JOIN Review rv ON r.RecipeID = rv.RecipeID
WHERE rv.Comments LIKE '%delicious%'
GROUP BY r.RecipeID, r.RecipeTitle
HAVING AVG(rv.Stars) > 4
LIMIT 15;
```

RecipeID	RecipeTitle
842610	Hollandaz Sauce For Vegetables
710401	Beef Stew
1905565	Pasta with Anchovies and Tomatoes
1688430	Tropical Fruit Salsa
997733	Spinach Dip With Homemade Pita Chips
272493	Best Oatmeal Cookies
13528	Chicken Fajita Sandwiches
933113	Hamusta Soup
1555470	Smurf Cake (Ombre Cake)
1803319	Mr. Jim's Salsa
467249	Chinese Pepper Steak
1441359	Carrot-Apple Soup
1534776	Bacon-Sausage Quiche Tarts
1205264	Cabbage And Yukon Gold Potato Casserole
1748354	Milwaukee's Best Spaghetti Sauce Recipe



```
EXPLAIN ANALYZE
SELECT r.RecipeID, r.RecipeTitle, ROUND(AVG(rv.Stars), 2) AS AvgStars
FROM Recipe r
JOIN Review rv ON r.RecipeID = rv.RecipeID
WHERE rv.Comments LIKE '%delicious%'
GROUP BY r.RecipeID, r.RecipeTitle
HAVING AVG(rv.Stars) > 4
LIMIT 15;
```

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.04612994

Timing (as measured by the server):
Execution time: 0:00:0.04498000
Table lock wait time: 0:00:0.00006000

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1
Rows examined: 3000

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 1

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Other Info:
Event Id: 1053
Thread Id: 69

Index	
1	CREATE INDEX idx_review_recipeid ON Review (RecipeID);
2	CREATE INDEX idx_review_recipeid_stars ON Review (RecipeID, Stars);
3	CREATE FULLTEXT INDEX idx_review_comments_fulltext ON Review (Comments);

The analysis of the indexing strategies revealed that the cost of using the `idx_review_recipid` index is higher at 6051.25 compared to the base query's cost of 219.55 due to fewer rows. However, it significantly enhances query performance by reducing the need for full table scans. The index allows for quicker data retrieval, especially with large datasets. Despite the increased cost, the efficiency gained through indexed lookups justifies its use, demonstrating that well-designed indexing can effectively optimize complex queries.

Index	Indexing Analysis

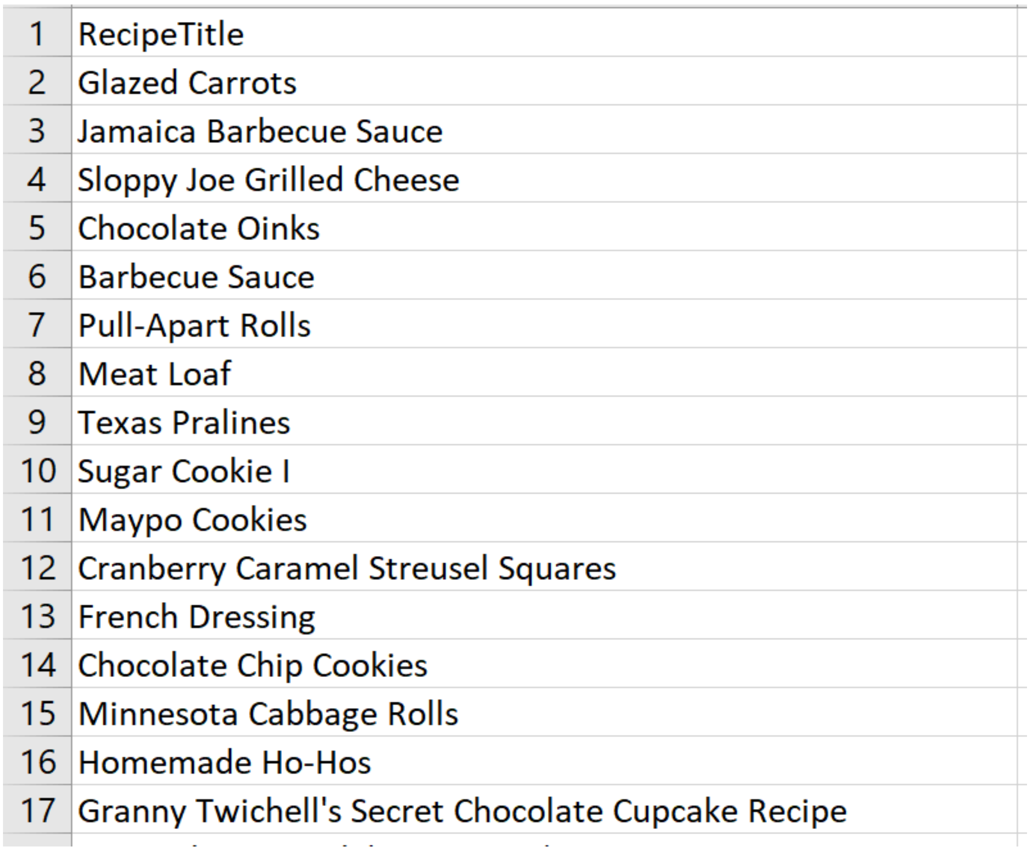
base	-> Limit: 15 row(s) (actual time=9.167..9.203 rows=15 loops=1)
	-> Filter: (avg(rv.Stars) > 4) (actual time=9.164..9.198 rows=15 loops=1)
	-> Table scan on <temporary> (actual time=9.153..9.170 rows=72 loops=1)
	-> Aggregate using temporary table (actual time=9.152..9.152 rows=171 loops=1)
	-> Inner hash join (rv.RecipeID = r.RecipeID) (cost=3035.35 rows=241) (actual time=5.305..8.629 rows=178 loops=1)
	-> Filter: (rv.Comments like '%delicious%') (cost=0.33 rows=11) (actual time=0.926..4.134 rows=178 loops=1)
	-> Table scan on rv (cost=0.33 rows=1000) (actual time=0.108..1.907 rows=1000 loops=1)
	-> Hash
	-> Table scan on r (cost=219.55 rows=1953) (actual time=0.250..3.267 rows=2000 loops=1)
1	-> Nested loop inner join (cost=6051.25 rows=37224) (actual time=3.870..16.788 rows=15 loops=1)
	-> Nested loop inner join (cost=1088.10 rows=2482) (actual time=0.425..12.385 rows=1000 loops=1)
	-> Filter: (s.RecipeID is not null) (cost=219.55 rows=1953) (actual time=0.066..3.552 rows=2000 loops=1)
	-> Table scan on s (cost=219.55 rows=1953) (actual time=0.065..3.322 rows=2000 loops=1)
	-> Index lookup on rv using idx_review_recipeid (RecipeID=s.RecipeID) (cost=0.32 rows=1) (actual time=0.004..0.004 rows=0 loops=2000)
	-> Covering index lookup on r using <auto_key0> (RecipeID=s.RecipeID) (actual time=0.004..0.004 rows=0 loops=1000)
	-> Materialize CTE top_recipe (cost=304.25..304.25 rows=15) (actual time=3.248..3.248 rows=15 loops=1)
	-> Limit: 15 row(s) (cost=302.75..302.75 rows=15) (actual time=3.177..3.181 rows=15 loops=1)
	-> Sort: avg_scores.avg_rating DESC, limit input to 15 row(s) per chunk (cost=302.75..302.75 rows=1000) (actual time=3.176..3.179 rows=15 loops=1)
	-> Table scan on avg_scores (cost=115.00 rows=1000) (actual time=2.550..2.752 rows=787 loops=1)
	-> Materialize CTE avg_scores (cost=302.75..302.75 rows=1000) (actual time=2.548..2.548 rows=787 loops=1)
	-> Group aggregate: avg(review.Stars) (cost=202.75

	rows=1000) (actual time=0.024..2.186 rows=787 loops=1) -> Index scan on Review using idx_review_recipeid (cost=102.75 rows=1000) (actual time=0.010..1.870 rows=1000 loops=1)
2	-> Limit: 15 row(s) (actual time=38.557..38.562 rows=15 loops=1) -> Sort: TotalCalories DESC, limit input to 15 row(s) per chunk (actual time=38.556..38.560 rows=15 loops=1) -> Table scan on <temporary> (actual time=37.432..37.772 rows=1771 loops=1) -> Aggregate using temporary table (actual time=37.431..37.431 rows=1771 loops=1) -> Nested loop inner join (cost=109572.55 rows=99251) (actual time=5.053..24.652 rows=5192 loops=1) -> Inner hash join (c.RecipeID = r.RecipeID) (cost=99520.36 rows=99251) (actual time=4.988..13.499 rows=5192 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.05 rows=508) (actual time=0.031..5.991 rows=5192 loops=1) -> Table scan on c (cost=0.05 rows=5082) (actual time=0.030..5.412 rows=5192 loops=1) -> Hash -> Table scan on r (cost=219.55 rows=1953) (actual time=0.076..3.728 rows=2000 loops=1) -> Filter: (c.IngredientID = i.IngredientID) (cost=0.00 rows=1) (actual time=0.002..0.002 rows=1 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.00 rows=1) (actual time=0.002..0.002 rows=1 loops=5192)
3	-> Limit: 15 row(s) (actual time=28.251..28.254 rows=15 loops=1) -> Sort: TotalCalories DESC, limit input to 15 row(s) per chunk (actual time=28.250..28.252 rows=15 loops=1) -> Table scan on <temporary> (actual time=27.648..27.812 rows=1771 loops=1) -> Aggregate using temporary table (actual time=27.647..27.647 rows=1771 loops=1) -> Nested loop inner join (cost=109572.55 rows=99251) (actual time=5.547..18.923 rows=5192 loops=1) -> Inner hash join (c.RecipeID = r.RecipeID) (cost=99520.36 rows=99251) (actual time=5.445..11.389 rows=5192 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.05 rows=508)

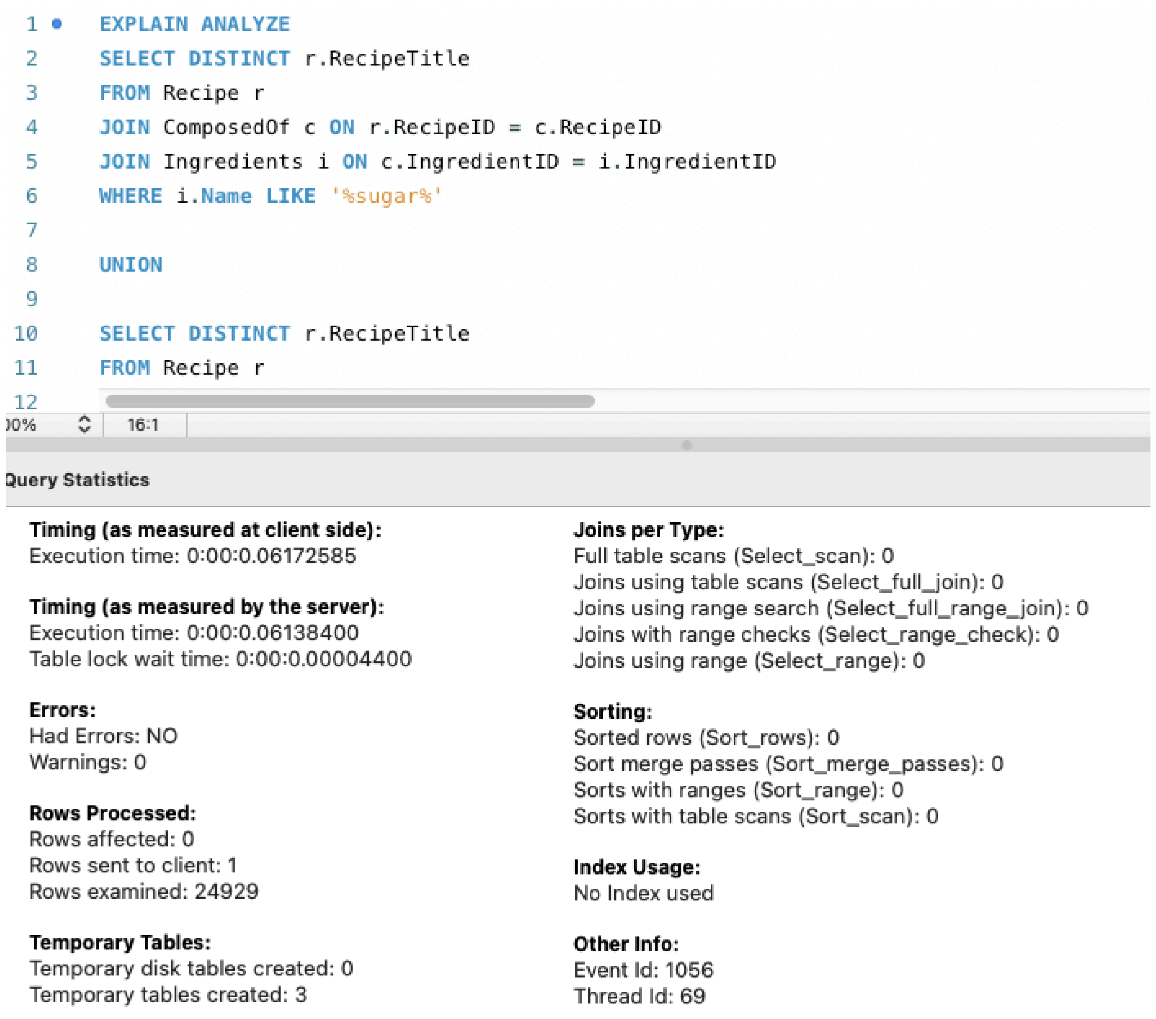
	(actual time=0.035..4.220 rows=5192 loops=1) -> Table scan on c (cost=0.05 rows=5082) (actual time=0.033..3.820 rows=5192 loops=1) -> Hash -> Table scan on r (cost=219.55 rows=1953) (actual time=0.091..4.150 rows=2000 loops=1) -> Filter: (c.IngredientID = i.IngredientID) (cost=0.00 rows=1) (actual time=0.001..0.001 rows=1 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.00 rows=1) (actual time=0.001..0.001 rows=1 loops=5192)

4. Find all recipes that containing 'sugar' or 'brown sugar'

```
SELECT DISTINCT r.RecipeTitle
FROM Recipe r
JOIN ComposedOf c ON r.RecipeID = c.RecipeID
JOIN Ingredients i ON c.IngredientID = i.IngredientID
WHERE i.Name LIKE '%sugar%'

UNION

SELECT DISTINCT r.RecipeTitle
FROM Recipe r
JOIN ComposedOf c ON r.RecipeID = c.RecipeID
JOIN Ingredients i ON c.IngredientID = i.IngredientID
WHERE i.Name LIKE '%brown sugar%';
```



1	RecipeTitle
2	Glazed Carrots
3	Jamaica Barbecue Sauce
4	Sloppy Joe Grilled Cheese
5	Chocolate Oinks
6	Barbecue Sauce
7	Pull-Apart Rolls
8	Meat Loaf
9	Texas Pralines
10	Sugar Cookie I
11	Maypo Cookies
12	Cranberry Caramel Streusel Squares
13	French Dressing
14	Chocolate Chip Cookies
15	Minnesota Cabbage Rolls
16	Homemade Ho-Hos
17	Granny Twichell's Secret Chocolate Cupcake Recipe


```
EXPLAIN ANALYZE
SELECT DISTINCT r.RecipeTitle
FROM Recipe r
JOIN ComposedOf c ON r.RecipeID = c.RecipeID
JOIN Ingredients i ON c.IngredientID = i.IngredientID
WHERE i.Name LIKE '%sugar%'

UNION

SELECT DISTINCT r.RecipeTitle
FROM Recipe r
```

```
Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.06172585

Timing (as measured by the server):
Execution time: 0:00:0.06138400
Table lock wait time: 0:00:0.00004400

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1
Rows examined: 24929

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 3

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Other Info:
Event Id: 1056
Thread Id: 69
```

Index	
1	CREATE INDEX idx_ingredient_name ON Ingredients(Name);
2	CREATE INDEX idx_ingredient_name ON Ingredients(Name);
3	CREATE INDEX idx_composedof_ingredientid ON ComposedOf(IngredientID);

With baseline indexes, the query shows a reported cost of 0.01..278.16. Adding an index on Ingredients(Name) maintains the same cost of 0.01..278.16, indicating no cost improvement for the condition. An index on ComposedOf(IngredientID) pushes the cost range up to 0.01..12554.44, which makes it appear more expensive rather than the baseline. In contrast, adding an index on ComposedOf(RecipeID) lowers the reported cost all the way down to 0.01..18.06, suggesting a significant improvement based on the optimizer’s cost calculations.

Index	Indexing Analysis
base	-> Table scan on <union temporary> (cost=0.01..278.16 rows=22054) (actual time=29.655..29.668 rows=161 loops=1) -> Union materialize with deduplication (cost=223836.51..224114.66 rows=22054) (actual time=29.655..29.655 rows=161 loops=1) -> Table scan on <temporary> (cost=0.01..140.32 rows=11027) (actual time=20.630..20.643 rows=161 loops=1) -> Temporary table with deduplication (cost=110675.25..110815.56 rows=11027) (actual time=20.629..20.629 rows=161 loops=1) -> Nested loop inner join (cost=109572.55 rows=11027) (actual time=8.261..20.219 rows=187 loops=1) -> Inner hash join (c.RecipeID = r.RecipeID) (cost=99520.36 rows=99251) (actual time=6.705..11.364 rows=5192 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.05 rows=508) (actual time=0.037..3.279 rows=5192 loops=1) -> Table scan on c (cost=0.05 rows=5082) (actual time=0.035..2.938 rows=5192 loops=1) -> Hash -> Table scan on r (cost=219.55 rows=1953) (actual time=0.229..5.013 rows=2000 loops=1) -> Limit: 1 row(s) (cost=0.00 rows=0.1) (actual time=0.002..0.002 rows=0 loops=5192) -> Filter: ((i.`Name` like '%sugar%') and (c.IngredientID = i.IngredientID)) (cost=0.00 rows=0.1) (actual time=0.001..0.001 rows=0 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.00 rows=1) (actual time=0.001..0.001 rows=1 loops=5192) -> Table scan on <temporary> (cost=0.01..140.32 rows=11027) (actual time=8.898..8.905 rows=93 loops=1) -> Temporary table with deduplication (cost=110675.25..110815.56 rows=11027) (actual time=8.897..8.897 rows=93 loops=1) -> Nested loop inner join (cost=109572.55 rows=11027) (actual time=1.846..8.777 rows=100 loops=1) -> Inner hash join (c.RecipeID = r.RecipeID) (cost=99520.36 rows=99251) (actual time=1.810..4.408 rows=5192 loops=1)

	-> Filter: (c.IngredientID is not null) (cost=0.05 rows=508) (actual time=0.028..1.805 rows=5192 loops=1) -> Table scan on c (cost=0.05 rows=5082) (actual time=0.027..1.543 rows=5192 loops=1) -> Hash -> Table scan on r (cost=219.55 rows=1953) (actual time=0.037..0.829 rows=2000 loops=1) -> Limit: 1 row(s) (cost=0.00 rows=0.1) (actual time=0.001..0.001 rows=0 loops=5192) -> Filter: ((i.\`Name\` like '%brown sugar%') and (c.IngredientID = i.IngredientID)) (cost=0.00 rows=0.1) (actual time=0.001..0.001 rows=0 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=5192)
1	-> Table scan on <union temporary> (cost=0.01..278.16 rows=22054) (actual time=39.460..39.487 rows=161 loops=1) -> Union materialize with deduplication (cost=223836.51..224114.66 rows=22054) (actual time=39.460..39.460 rows=161 loops=1) -> Table scan on <temporary> (cost=0.01..140.32 rows=11027) (actual time=28.986..29.000 rows=161 loops=1) -> Temporary table with deduplication (cost=110675.25..110815.56 rows=11027) (actual time=28.985..28.985 rows=161 loops=1) -> Nested loop inner join (cost=109572.55 rows=11027) (actual time=5.922..28.444 rows=187 loops=1) -> Inner hash join (c.RecipeID = r.RecipeID) (cost=99520.36 rows=99251) (actual time=5.084..14.341 rows=5192 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.05 rows=508) (actual time=0.046..6.611 rows=5192 loops=1) -> Table scan on c (cost=0.05 rows=5082) (actual time=0.044..6.008 rows=5192 loops=1) -> Hash -> Table scan on r (cost=219.55 rows=1953) (actual time=0.074..3.769 rows=2000 loops=1) -> Limit: 1 row(s) (cost=0.00 rows=0.1) (actual time=0.003..0.003 rows=0 loops=5192) -> Filter: ((i.\`Name\` like '%sugar%') and (c.IngredientID = i.IngredientID)) (cost=0.00 rows=0.1) (actual time=0.002..0.002 rows=0 loops=5192)

	-> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.00 rows=1) (actual time=0.002..0.002 rows=1 loops=5192) -> Table scan on <temporary> (cost=0.01..140.32 rows=11027) (actual time=10.300..10.315 rows=93 loops=1) -> Temporary table with deduplication (cost=110675.25..110815.56 rows=11027) (actual time=10.299..10.299 rows=93 loops=1) -> Nested loop inner join (cost=109572.55 rows=11027) (actual time=1.084..10.227 rows=100 loops=1) -> Inner hash join (c.RecipeID = r.RecipeID) (cost=99520.36 rows=99251) (actual time=1.063..4.468 rows=5192 loops=1) -> Filter: (c.IngredientID is not null) (cost=0.05 rows=508) (actual time=0.005..2.329 rows=5192 loops=1) -> Table scan on c (cost=0.05 rows=5082) (actual time=0.004..1.988 rows=5192 loops=1) -> Hash -> Table scan on r (cost=219.55 rows=1953) (actual time=0.008..0.664 rows=2000 loops=1) -> Limit: 1 row(s) (cost=0.00 rows=0.1) (actual time=0.001..0.001 rows=0 loops=5192) -> Filter: ((i.\`Name\` like '%brown sugar%') and (c.IngredientID = i.IngredientID)) (cost=0.00 rows=0.1) (actual time=0.001..0.001 rows=0 loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.00 rows=1) (actual time=0.001..0.001 rows=1 loops=5192)
2	-> Table scan on <union temporary> (cost=0.01..12554.44 rows=1004155) (actual time=14.581..14.607 rows=161 loops=1) -> Union materialize with deduplication (cost=1220370.59..1232925.02 rows=1004155) (actual time=14.580..14.580 rows=161 loops=1) -> Table scan on <temporary> (cost=0.01..6278.46 rows=502078) (actual time=8.681..8.713 rows=161 loops=1) -> Temporary table with deduplication (cost=553699.09..559977.54 rows=502078) (actual time=8.679..8.679 rows=161 loops=1) -> Inner hash join (r.RecipeID = c.RecipeID) (cost=503491.33 rows=502078) (actual time=4.702..8.330 rows=187 loops=1) -> Table scan on r (cost=0.11 rows=1953) (actual time=0.043..3.239 rows=2000 loops=1) -> Hash

	-> Nested loop inner join (cost=1141.53 rows=2571) (actual time=1.118..4.513 rows=187 loops=1) -> Filter: (i.\`Name\` like '%sugar%') (cost=241.75 rows=266) (actual time=0.355..3.606 rows=14 loops=1) -> Table scan on i (cost=241.75 rows=2395) (actual time=0.147..2.030 rows=2395 loops=1) -> Index lookup on c using idx_composedof_ingredientid (IngredientID=i.IngredientID), with index condition: (c.IngredientID = i.IngredientID) (cost=2.42 rows=10) (actual time=0.010..0.062 rows=13 loops=14) -> Table scan on <temporary> (cost=0.01..6278.46 rows=502078) (actual time=5.575..5.590 rows=93 loops=1) -> Temporary table with deduplication (cost=553699.09..559977.54 rows=502078) (actual time=5.573..5.573 rows=93 loops=1) -> Inner hash join (r.RecipeID = c.RecipeID) (cost=503491.33 rows=502078) (actual time=2.725..5.405 rows=100 loops=1) -> Table scan on r (cost=0.11 rows=1953) (actual time=0.010..2.321 rows=2000 loops=1) -> Hash -> Nested loop inner join (cost=1141.53 rows=2571) (actual time=0.710..2.650 rows=100 loops=1) -> Filter: (i.\`Name\` like '%brown sugar%') (cost=241.75 rows=266) (actual time=0.689..2.418 rows=1 loops=1) -> Table scan on i (cost=241.75 rows=2395) (actual time=0.071..1.358 rows=2395 loops=1) -> Index lookup on c using idx_composedof_ingredientid (IngredientID=i.IngredientID), with index condition: (c.IngredientID = i.IngredientID) (cost=2.42 rows=10) (actual time=0.020..0.220 rows=100 loops=1)
3	-> Table scan on <union temporary> (cost=0.01..18.06 rows=1245) (actual time=44.320..44.335 rows=161 loops=1) -> Union materialize with deduplication (cost=8554.68..8572.73 rows=1245) (actual time=44.320..44.320 rows=161 loops=1) -> Table scan on <temporary> (cost=0.02..10.28 rows=623) (actual time=31.990..32.005 rows=161 loops=1) -> Temporary table with deduplication (cost=4204.81..4215.07 rows=623) (actual time=31.989..31.989 rows=161 loops=1) -> Nested loop inner join (cost=4142.53 rows=623) (actual time=0.476..31.576 rows=187 loops=1)

```
                    -> Nested loop inner join  (cost=2181.04 rows=5604) (actual
time=0.130..20.367 rows=5192 loops=1)
                        -> Filter: (r.RecipeID is not null)  (cost=219.55 rows=1953)
(actual time=0.068..5.430 rows=2000 loops=1)
                            -> Table scan on r  (cost=219.55 rows=1953) (actual
time=0.067..5.229 rows=2000 loops=1)
                        -> Filter: (c.IngredientID is not null)  (cost=0.72 rows=3) (actual
time=0.004..0.007 rows=3 loops=2000)
                            -> Index lookup on c using idx_composedof_recipeid
(RecipeID=r.RecipeID)  (cost=0.72 rows=3) (actual time=0.003..0.007 rows=3
loops=2000)
                    -> Limit: 1 row(s)  (cost=0.25 rows=0.1) (actual
time=0.002..0.002 rows=0 loops=5192)
                        -> Filter: ((i.`Name` like '%sugar%') and (c.IngredientID =
i.IngredientID))  (cost=0.25 rows=0.1) (actual time=0.002..0.002 rows=0
loops=5192)
                            -> Single-row index lookup on i using PRIMARY
(IngredientID=c.IngredientID)  (cost=0.25 rows=1) (actual time=0.001..0.001
rows=1 loops=5192)
        -> Table scan on <temporary>  (cost=0.02..10.28 rows=623) (actual
time=12.188..12.197 rows=93 loops=1)
            -> Temporary table with deduplication  (cost=4204.81..4215.07
rows=623) (actual time=12.187..12.187 rows=93 loops=1)
                -> Nested loop inner join  (cost=4142.53 rows=623) (actual
time=0.040..12.112 rows=100 loops=1)
                    -> Nested loop inner join  (cost=2181.04 rows=5604) (actual
time=0.017..7.119 rows=5192 loops=1)
                        -> Filter: (r.RecipeID is not null)  (cost=219.55 rows=1953)
(actual time=0.009..0.988 rows=2000 loops=1)
                            -> Table scan on r  (cost=219.55 rows=1953) (actual
time=0.009..0.877 rows=2000 loops=1)
                        -> Filter: (c.IngredientID is not null)  (cost=0.72 rows=3) (actual
time=0.001..0.003 rows=3 loops=2000)
                            -> Index lookup on c using idx_composedof_recipeid
(RecipeID=r.RecipeID)  (cost=0.72 rows=3) (actual time=0.001..0.003 rows=3
loops=2000)
                    -> Limit: 1 row(s)  (cost=0.25 rows=0.1) (actual
time=0.001..0.001 rows=0 loops=5192)
                        -> Filter: ((i.`Name` like '%brown sugar%') and (c.IngredientID
= i.IngredientID))  (cost=0.25 rows=0.1) (actual time=0.001..0.001 rows=0
```

	loops=5192) -> Single-row index lookup on i using PRIMARY (IngredientID=c.IngredientID) (cost=0.25 rows=1) (actual time=0.000..0.001 rows=1 loops=5192)