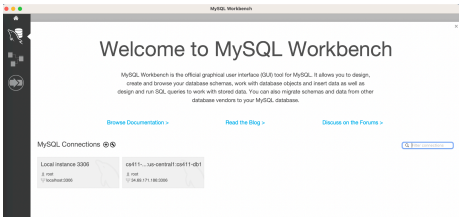
Stage3 Report

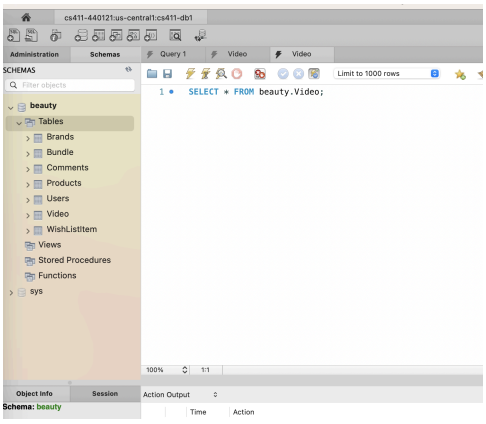
1. Database Implementation

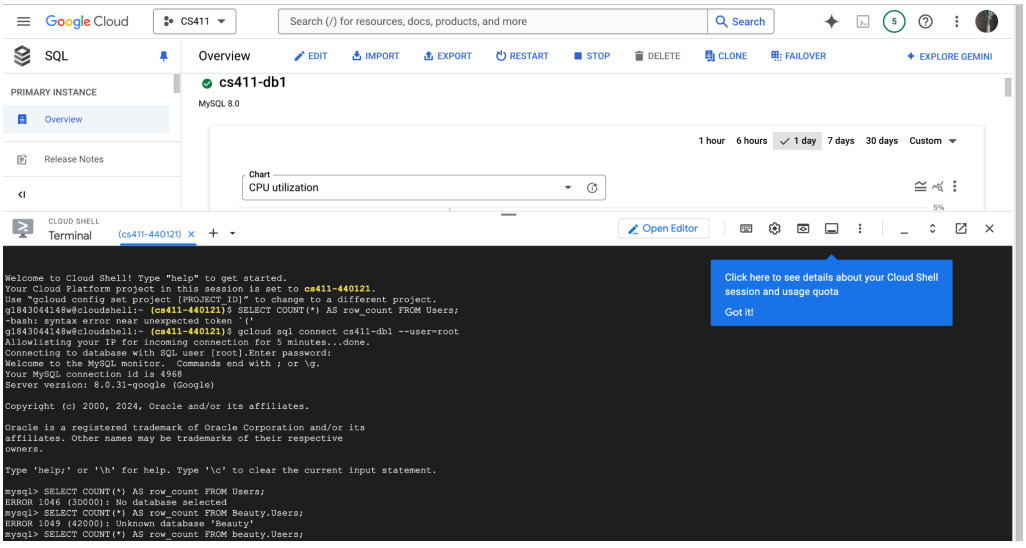
a. Database Connection

Platform: The database is implemented on the Google Cloud Platform (GCP) with a MySQL instance.

Connection Screenshot:





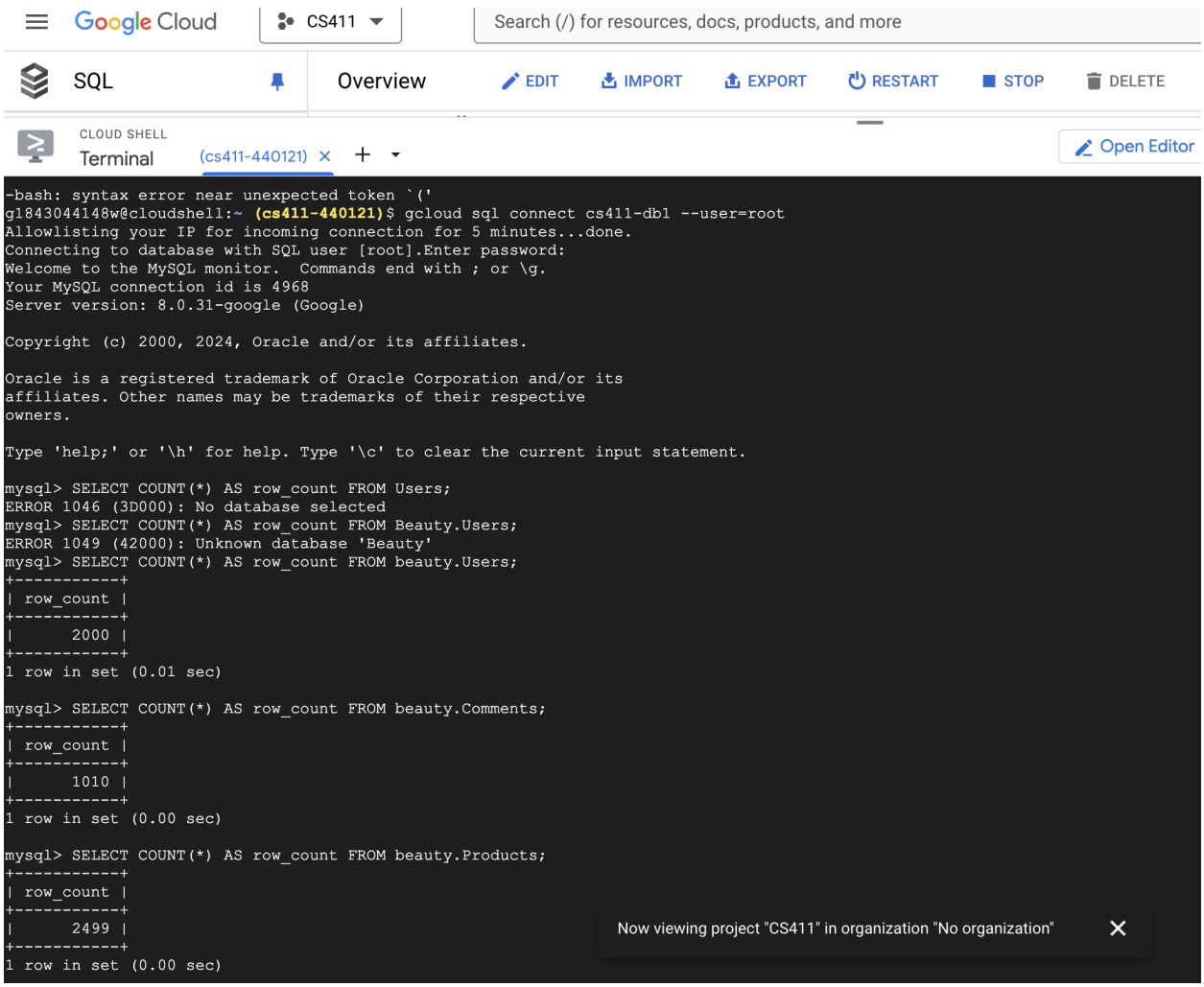


b. DDL Commands

```
CREATE TABLE Users (
    UserId VARCHAR(50) PRIMARY KEY,
    UserName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) NOT NULL,
    Age INT,
    Gender VARCHAR(20),
    Password VARCHAR(255) NOT NULL,
    SkinType VARCHAR(20)
);

CREATE TABLE Products (
    ProductId INT PRIMARY KEY,
    ProductName VARCHAR(255) NOT NULL,
    Category VARCHAR(50),
    Price DECIMAL(8, 2),
    BrandId INT,
    UsageFrequency VARCHAR(30),
    SkinType VARCHAR(20),
    NumberOfReviews INT,
    Rating DECIMAL(3, 2),
    GenderTarget VARCHAR(20),
    FOREIGN KEY (BrandId) REFERENCES Brands(BrandId)
);

CREATE TABLE Brands (
    BrandId INT PRIMARY KEY,
    BrandCountry VARCHAR(255)
);

CREATE TABLE Video (
    VideoId INT PRIMARY KEY,
    ProductId INT,
    VideoLink VARCHAR(255),
    FOREIGN KEY (ProductId) REFERENCES Products(ProductId)
);

CREATE TABLE WishListItem (
    RecordId INT PRIMARY KEY,
    UserId VARCHAR(50),
    ProductId INT,
    FOREIGN KEY (UserId) REFERENCES Users(UserId),
    FOREIGN KEY (ProductId) REFERENCES Products(ProductId)
);
```

```
CREATE TABLE Comments (
    CommentId INT PRIMARY KEY,
    ProductId INT,
    UserId VARCHAR(50),
    Date DATE,
    Rating DECIMAL(3, 2),
    CommentContent VARCHAR(255),
    FOREIGN KEY (ProductId) REFERENCES Products(ProductId),
    FOREIGN KEY (UserId) REFERENCES Users(UserId)
);

CREATE TABLE Bundle (
    RecordId INT PRIMARY KEY,
    UserId VARCHAR(50),
    ProductId INT,
    BundledId INT,
    FOREIGN KEY (UserId) REFERENCES Users(UserId),
    FOREIGN KEY (ProductId) REFERENCES Products(ProductId)
);
```

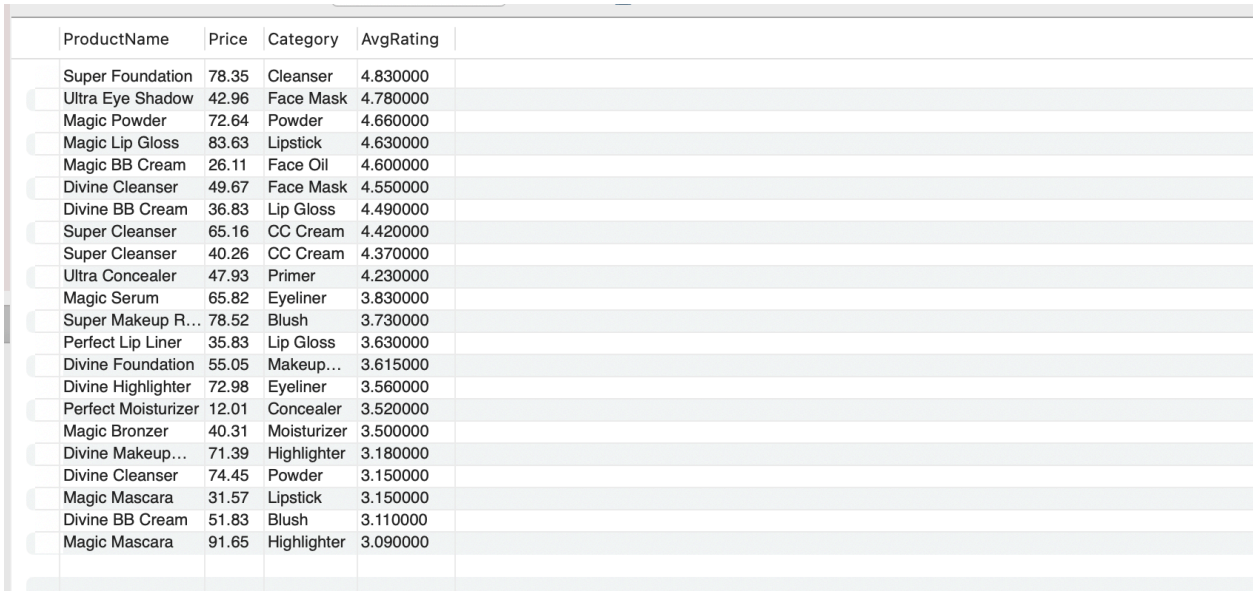
c. Data Insertion



2. Advanced Queries

Query 1: Top Recommended Products Based on User Preferences and Ratings

```
SELECT p.ProductName, p.Price, p.Category, AVG(c.Rating) AS AvgRating
FROM Products p
JOIN Comments c ON p.ProductId = c.ProductId
WHERE p.SkinType = 'Oily'
  AND p.Price <= 100
  AND (p.GenderTarget = 'Female' OR p.GenderTarget IS NULL)
GROUP BY p.ProductName, p.Price, p.Category
HAVING AVG(c.Rating) > 3.0
ORDER BY AvgRating DESC
```



ProductName	Price	Category	AvgRating
Super Foundation	78.35	Cleanser	4.830000
Ultra Eye Shadow	42.96	Face Mask	4.780000
Magic Powder	72.64	Powder	4.660000
Magic Lip Gloss	83.63	Lipstick	4.630000
Magic BB Cream	26.11	Face Oil	4.600000
Divine Cleanser	49.67	Face Mask	4.550000
Divine BB Cream	36.83	Lip Gloss	4.490000
Super Cleanser	65.16	CC Cream	4.420000
Super Cleanser	40.26	CC Cream	4.370000
Ultra Concealer	47.93	Primer	4.230000
Magic Serum	65.82	Eyeliner	3.830000
Super Makeup R…	78.52	Blush	3.730000
Perfect Lip Liner	35.83	Lip Gloss	3.630000
Divine Foundation	55.05	Makeup…	3.615000
Divine Highlighter	72.98	Eyeliner	3.560000
Perfect Moisturizer	12.01	Concealer	3.520000
Magic Bronzer	40.31	Moisturizer	3.500000
Divine Makeup…	71.39	Highlighter	3.180000
Divine Cleanser	74.45	Powder	3.150000
Magic Mascara	31.57	Lipstick	3.150000
Divine BB Cream	51.83	Blush	3.110000
Magic Mascara	91.65	Highlighter	3.090000

Query 2: Most Popular Products by Hit Rate and Reviews

```
SELECT p.ProductName, p.Category, p.Price, p.NumberOfReviews, COUNT(c.CommentId) AS
TotalReviews
FROM Products p
LEFT JOIN Comments c ON p.ProductId = c.ProductId
GROUP BY p.ProductName, p.Category, p.Price, p.NumberOfReviews
HAVING COUNT(c.CommentId) > 2 AND p.NumberOfReviews > 10
ORDER BY p.NumberOfReviews DESC, TotalReviews DESC;
```

ProductName	Category	Price	NumberOfReviews	TotalReviews
Perfect Lip Gloss	Lip Liner	34.52	9621	3
Super Face Mask	Face Oil	57.86	8935	3
Perfect Bronzer	Blush	38.49	8316	3
Magic Blush	Cleanser	37.87	7907	3
Ultra Eyeliner	Setting Spray	96.61	7153	3
Perfect Lip Gloss	Serum	50.29	6974	3
Perfect Setting Spray	Lip Liner	133.45	6070	3
Divine Lip Liner	Exfoliator	31.40	5359	4
Ultra BB Cream	Makeup Remover	38.06	5339	3
Divine Primer	Eyeliner	96.00	3557	3
Perfect Lip Liner	Face Oil	15.93	3204	3
Magic Serum	Powder	63.46	2493	4
Divine Moisturizer	Cleanser	127.69	2302	3
Super Mascara	Face Mask	83.01	1312	3
Super Foundation	Face Oil	60.96	1214	3
Ultra Setting Spray	Eyeliner	27.09	1182	3
Divine Concealer	Face Mask	109.61	506	3
Divine Highlighter	Lip Gloss	75.43	407	3

Query 3: Compare Average Price of Products by Category Using Set Operators

```
SELECT p.Category, AVG(p.Price) AS AvgPrice, b.BrandName
FROM Products p
JOIN Brands b ON p.BrandId = b.BrandId
GROUP BY p.Category, b.BrandName
HAVING p.Category IN ('Highlighter', 'Blush', 'Powder', 'Lip Gloss', 'CC Cream', 'Eye Shadow');
```

Category	AvgPrice	BrandName
Blush	39.820000	Drunk Elephant
Lip Gloss	57.410000	Drunk Elephant
Eye Shadow	100.623333	Drunk Elephant
Powder	50.510000	Drunk Elephant
CC Cream	71.120000	Drunk Elephant
Highlighter	43.690000	Drunk Elephant
CC Cream	58.583333	Laura Mercier
Eye Shadow	100.297500	Laura Mercier
Lip Gloss	76.500000	Laura Mercier
Blush	109.712000	Laura Mercier
Highlighter	108.050000	Laura Mercier
Powder	46.110000	Laura Mercier
Highlighter	50.343333	Natasha Den…
Powder	80.780000	Natasha Den…
CC Cream	70.761667	Natasha Den…
Eye Shadow	73.100000	Natasha Den…
Lip Gloss	34.965000	Natasha Den…
Blush	141.060000	Natasha Den…
CC Cream	102.220000	Ilia Beauty

Query 4: Identify Trending Products Based on Recent Reviews

```
SELECT p.ProductName, p.Category, COUNT(c.CommentId) AS RecentReviews
FROM Products p
```

```
JOIN Comments c ON p.ProductId = c.ProductId
WHERE c.Date >= CURDATE() - INTERVAL 365 DAY
GROUP BY p.ProductName, p.Category
HAVING RecentReviews > 3
ORDER BY RecentReviews DESC;
```

ProductName	Category	RecentReviews
Divine Lip Liner	Exfoliator	6
Perfect Lip Liner	Face Oil	5
Ultra Face Oil	Primer	4
Divine Primer	Eyeliner	4
Divine Highlighter	Lip Gloss	4
Magic BB Cream	CC Cream	4
Magic Lip Gloss	Setting Spray	4
Magic BB Cream	Blush	4
Divine Primer	Bronzer	4
Divine Setting Spray	Face Mask	4
Super Mascara	Face Mask	4
Magic Mascara	Mascara	4
Divine Highlighter	Eyeliner	4
Magic Serum	Powder	4
Ultra Eyeliner	Setting Spray	4
Perfect Lip Gloss	Serum	4
Perfect Lip Liner	Lip Gloss	4
Ultra Cleanser	Lip Gloss	4
Ultra Contour	Blush	4

3.Indexing Analysis

Query 1: Top Recommended Products Based on User Preferences and Ratings

```
SELECT p.ProductName, p.Price, p.Category, AVG(c.Rating) AS AvgRating
FROM Products p
JOIN Comments c ON p.ProductId = c.ProductId
WHERE p.SkinType = 'Oily'
AND p.Price <= 100
AND (p.GenderTarget = 'Female' OR p.GenderTarget IS NULL)
GROUP BY p.ProductName, p.Price, p.Category
HAVING AVG(c.Rating) > 3.0
ORDER BY AvgRating DESC;
```

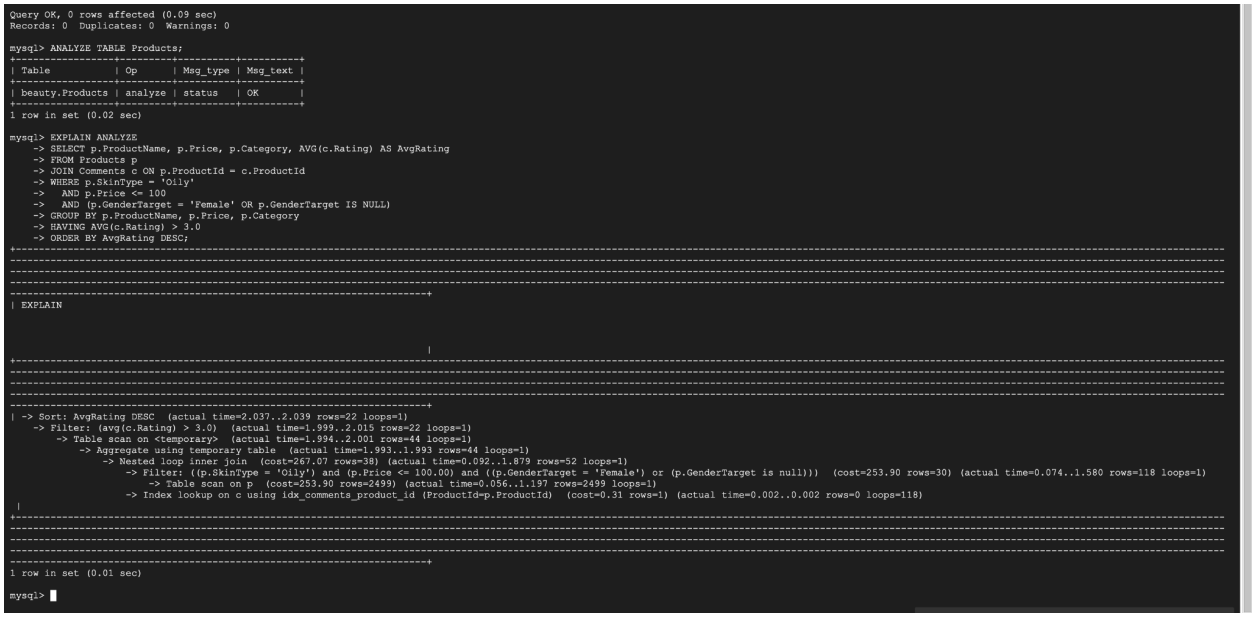
Original:

```
| -> Sort: AvgRating DESC  (actual time=2.170..2.173 rows=22 loops=1)
```

-> Filter: (avg(c.Rating) > 3.0) (actual time=2.123..2.139 rows=22 loops=1)
    -> Table scan on <temporary> (actual time=2.118..2.124 rows=44 loops=1)
        -> Aggregate using temporary table (actual time=2.116..2.116 rows=44 loops=1)
            -> Nested loop inner join (cost=260.75 rows=20) (actual time=0.094..1.985 rows=52 loops=1)
                -> Filter: ((p.SkinType = 'Oily') and (p.Price <= 100.00) and ((p.GenderTarget = 'Female') or (p.GenderTarget is null))) (cost=253.90 rows=16) (actual time=0.075..1.575 rows=118 loops=1)
                    -> Table scan on p (cost=253.90 rows=2499) (actual time=0.057..1.200 rows=2499 loops=1)
                -> Index lookup on c using idx_comments_product_id (ProductId=p.ProductId) (cost=0.32 rows=1) (actual time=0.003..0.003 rows=0 loops=118)

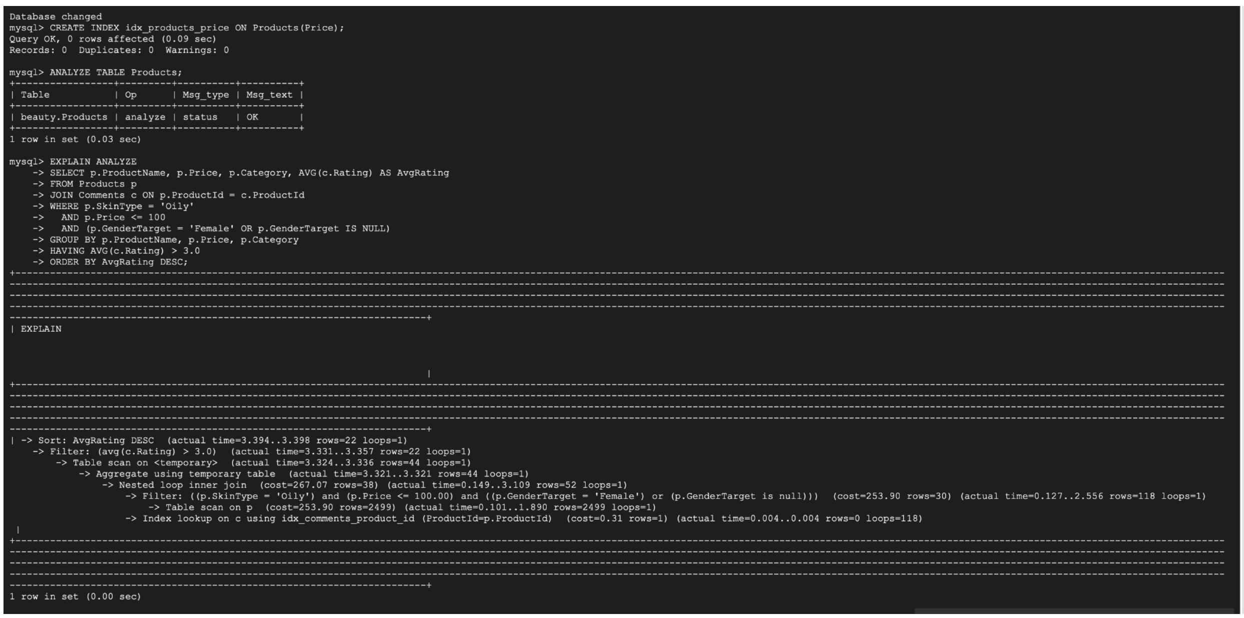
After using date as index

| -> Sort: AvgRating DESC (actual time=2.037..2.039 rows=22 loops=1)
    -> Filter: (avg(c.Rating) > 3.0) (actual time=1.999..2.015 rows=22 loops=1)
        -> Table scan on <temporary> (actual time=1.994..2.001 rows=44 loops=1)
            -> Aggregate using temporary table (actual time=1.993..1.993 rows=44 loops=1)
                -> Nested loop inner join (cost=267.07 rows=38) (actual time=0.092..1.879 rows=52 loops=1)
                    -> Filter: ((p.SkinType = 'Oily') and (p.Price <= 100.00) and ((p.GenderTarget = 'Female') or (p.GenderTarget is null))) (cost=253.90 rows=30) (actual time=0.074..1.580 rows=118 loops=1)
                        -> Table scan on p (cost=253.90 rows=2499) (actual time=0.056..1.197 rows=2499 loops=1)
                    -> Index lookup on c using idx_comments_product_id (ProductId=p.ProductId) (cost=0.31 rows=1) (actual time=0.002..0.002 rows=0 loops=118)



After using price as index

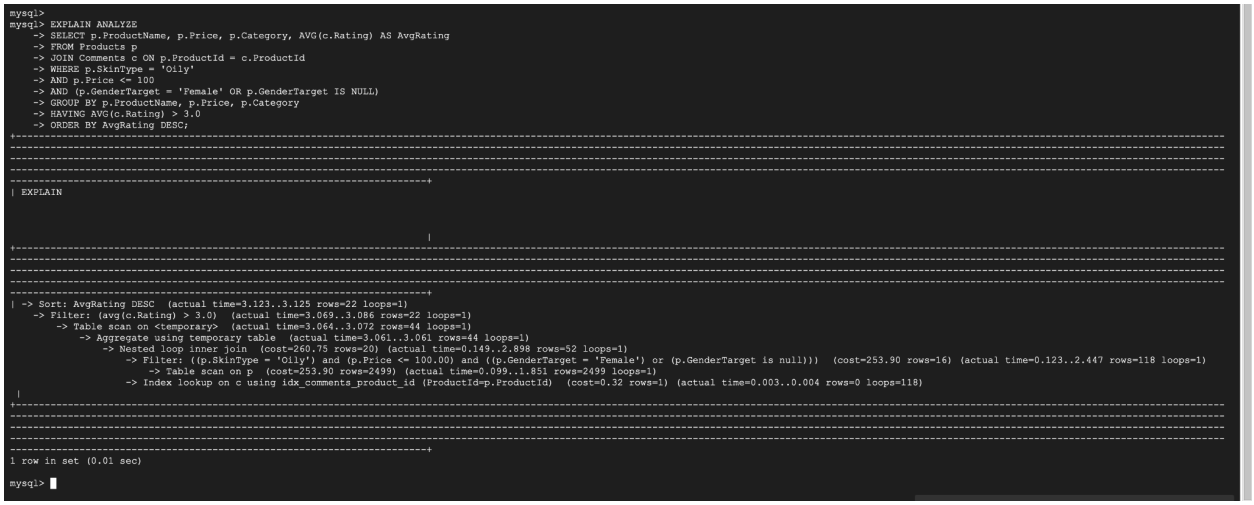
| -> Sort: AvgRating DESC (actual time=3.394..3.398 rows=22 loops=1)
    -> Filter: (avg(c.Rating) > 3.0) (actual time=3.331..3.357 rows=22 loops=1)
        -> Table scan on <temporary> (actual time=3.324..3.336 rows=44 loops=1)
            -> Aggregate using temporary table (actual time=3.321..3.321 rows=44 loops=1)
                -> Nested loop inner join (cost=267.07 rows=38) (actual time=0.149..3.109 rows=52 loops=1)
                    -> Filter: ((p.SkinType = 'Oily') and (p.Price <= 100.00) and ((p.GenderTarget = 'Female') or (p.GenderTarget is null))) (cost=253.90 rows=30) (actual time=0.127..2.556 rows=118 loops=1)
                        -> Table scan on p (cost=253.90 rows=2499) (actual time=0.101..1.890 rows=2499 loops=1)
                    -> Index lookup on c using idx_comments_product_id (ProductId=p.ProductId) (cost=0.31 rows=1) (actual time=0.004..0.004 rows=0 loops=118)


```
Database changed
mysql> CREATE INDEX idx_products_price ON Products(Price);
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> ANALYZE TABLE Products;
+------------------+---------+----------+----------+
| Table            | Op      | Msg_type | Msg_text |
+------------------+---------+----------+----------+
| beauty.Products  | analyze | status   | OK       |
+------------------+---------+----------+----------+
1 row in set (0.03 sec)

mysql> EXPLAIN ANALYZE
    -> SELECT p.ProductName, p.Price, p.Category, AVG(c.Rating) AS AvgRating
    -> FROM Products p
    -> JOIN Comments c ON p.ProductId = c.ProductId
    -> WHERE p.SkinType = 'Oily'
    ->   AND p.Price <= 100
    ->   AND (p.GenderTarget = 'Female' OR p.GenderTarget IS NULL)
    -> GROUP BY p.ProductName, p.Price, p.Category
    -> HAVING AVG(c.Rating) > 3.0
    -> ORDER BY AvgRating DESC;
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------------------------------------------------+
| EXPLAIN
                                                                        |
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------------------------------------------------+
| -> Sort: AvgRating DESC  (actual time=3.394..3.398 rows=22 loops=1)
    -> Filter: (avg(c.Rating) > 3.0)  (actual time=3.331..3.357 rows=22 loops=1)
        -> Table scan on <temporary>  (actual time=3.324..3.336 rows=44 loops=1)
            -> Aggregate using temporary table  (actual time=3.321..3.321 rows=44 loops=1)
                -> Nested loop inner join  (cost=267.07 rows=38) (actual time=0.149..3.109 rows=52 loops=1)
                    -> Filter: ((p.SkinType = 'Oily') and (p.Price <= 100.00) and ((p.GenderTarget = 'Female') or (p.GenderTarget is null)))  (cost=253.90 rows=30) (actual time=0.127..2.556 rows=118 loops=1)
                        -> Table scan on p  (cost=253.90 rows=2499) (actual time=0.101..1.890 rows=2499 loops=1)
                    -> Index lookup on c using idx_comments_product_id (ProductId=p.ProductId)  (cost=0.31 rows=1) (actual time=0.004..0.004 rows=0 loops=118)
 |
+--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

After using category as index

| -> Sort: AvgRating DESC (actual time=3.123..3.125 rows=22 loops=1)
    -> Filter: (avg(c.Rating) > 3.0) (actual time=3.069..3.086 rows=22 loops=1)
        -> Table scan on <temporary> (actual time=3.064..3.072 rows=44 loops=1)
            -> Aggregate using temporary table (actual time=3.061..3.061 rows=44 loops=1)
                -> Nested loop inner join (cost=260.75 rows=20) (actual time=0.149..2.898 rows=52 loops=1)
                    -> Filter: ((p.SkinType = 'Oily') and (p.Price <= 100.00) and ((p.GenderTarget = 'Female') or (p.GenderTarget is null))) (cost=253.90 rows=16) (actual time=0.123..2.447 rows=118 loops=1)
                        -> Table scan on p (cost=253.90 rows=2499) (actual time=0.099..1.851 rows=2499 loops=1)

-> Index lookup on c using idx_comments_product_id (ProductId=p.ProductId) (cost=0.32 rows=1) (actual time=0.003..0.004 rows=0 loops=118)



Index Analysis:

To explore the tradeoffs of adding different indices to optimize Query 1, we tested indexing on various attributes, specifically Date, Price, and Category, and used the EXPLAIN ANALYZE command to measure the impacts on runtime and cost. Without additional indices, the query had a baseline execution time of approximately 2.173 ms and a cost of 260.75. Adding an index on Date slightly reduced runtime to 2.039 ms but increased the cost to 267.07, suggesting limited benefit since Date was not used in primary filtering. Adding an index on Price significantly increased runtime to 3.398 ms and kept the cost at 267.07, as the wide range of Price values offered little advantage in reducing the search space. Similarly, indexing Category—used only in grouping—did not improve performance and raised the execution time to 3.125 ms while maintaining a cost of 260.75. This analysis shows that indexing attributes not directly involved in joins or filtering adds overhead without reducing costs or improving runtime. Thus, we selected the original design without additional indices, as it achieved the lowest runtime and cost, demonstrating the importance of aligning indexes with the query's filtering and join requirements.

Query 2: Most Popular Products by Hit Rate and Reviews

```
SELECT p.ProductName, p.Category, p.Price, p.NumberOfReviews, COUNT(c.CommentId) AS TotalReviews
FROM Products p
LEFT JOIN Comments c ON p.ProductId = c.ProductId
GROUP BY p.ProductName, p.Category, p.Price, p.NumberOfReviews
```

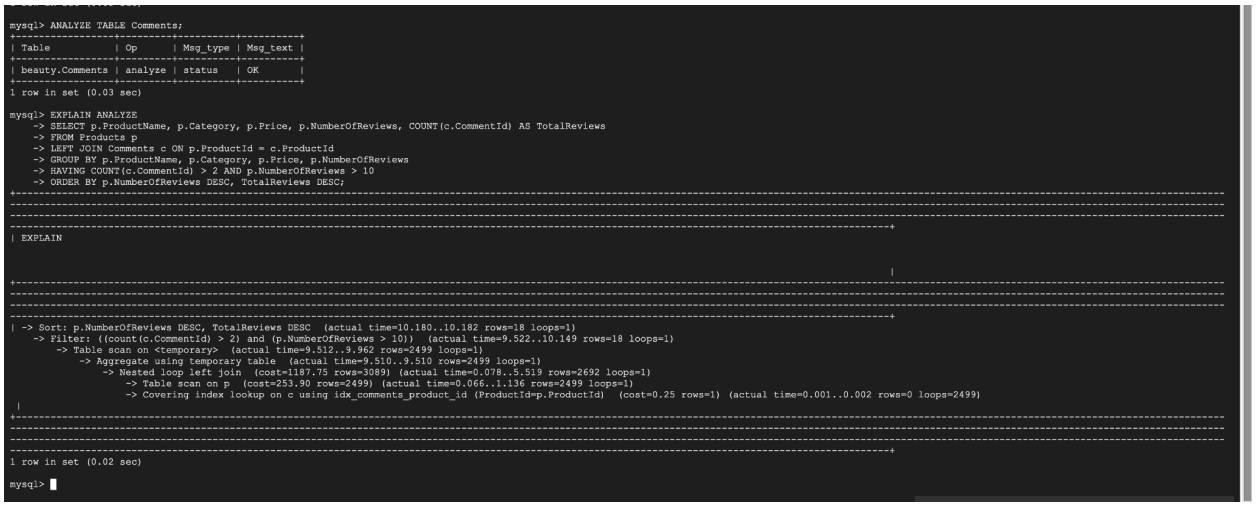
HAVING COUNT(c.CommentId) > 2 AND p.NumberOfReviews > 10
ORDER BY p.NumberOfReviews DESC, TotalReviews DESC;

Original:

| -> Sort: p.NumberOfReviews DESC, TotalReviews DESC (actual time=10.121..10.123 rows=18 loops=1)
    -> Filter: ((count(c.CommentId) > 2) and (p.NumberOfReviews > 10)) (actual time=9.571..10.095 rows=18 loops=1)
        -> Table scan on <temporary> (actual time=9.560..9.936 rows=2499 loops=1)
            -> Aggregate using temporary table (actual time=9.557..9.557 rows=2499 loops=1)
                -> Nested loop left join (cost=1187.75 rows=3089) (actual time=0.080..5.451 rows=2692 loops=1)
                    -> Table scan on p (cost=253.90 rows=2499) (actual time=0.066..1.125 rows=2499 loops=1)
                    -> Covering index lookup on c using idx_comments_product_id (ProductId=p.ProductId) (cost=0.25 rows=1) (actual time=0.001..0.002 rows=0 loops=2499)

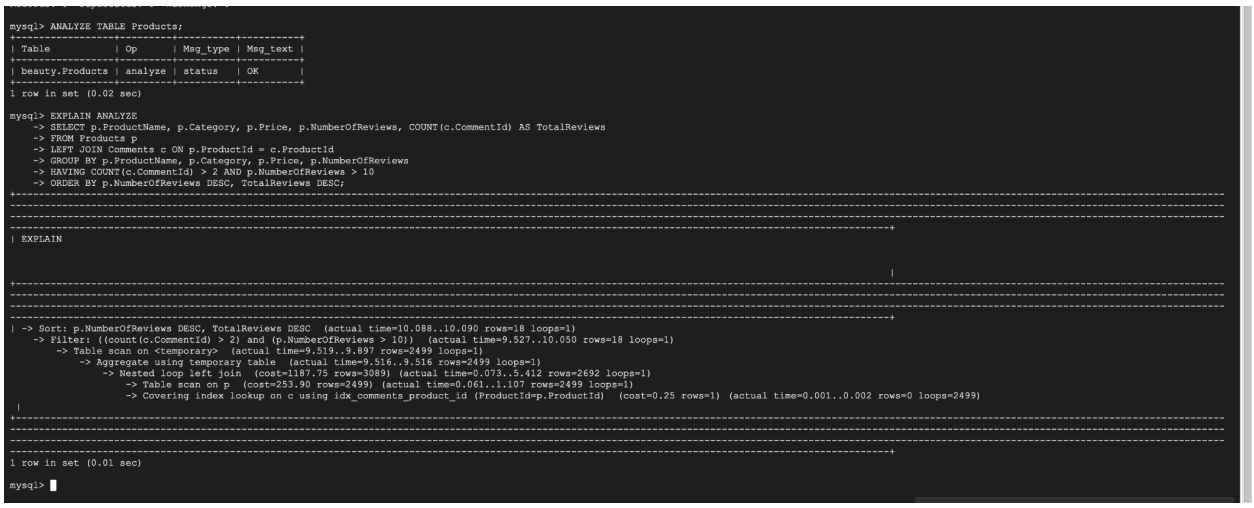
After using date as index

| -> Sort: p.NumberOfReviews DESC, TotalReviews DESC (actual time=10.180..10.182 rows=18 loops=1)
    -> Filter: ((count(c.CommentId) > 2) and (p.NumberOfReviews > 10)) (actual time=9.522..10.149 rows=18 loops=1)
        -> Table scan on <temporary> (actual time=9.512..9.962 rows=2499 loops=1)
            -> Aggregate using temporary table (actual time=9.510..9.510 rows=2499 loops=1)
                -> Nested loop left join (cost=1187.75 rows=3089) (actual time=0.078..5.519 rows=2692 loops=1)
                    -> Table scan on p (cost=253.90 rows=2499) (actual time=0.066..1.136 rows=2499 loops=1)
                    -> Covering index lookup on c using idx_comments_product_id (ProductId=p.ProductId) (cost=0.25 rows=1) (actual time=0.001..0.002 rows=0 loops=2499)



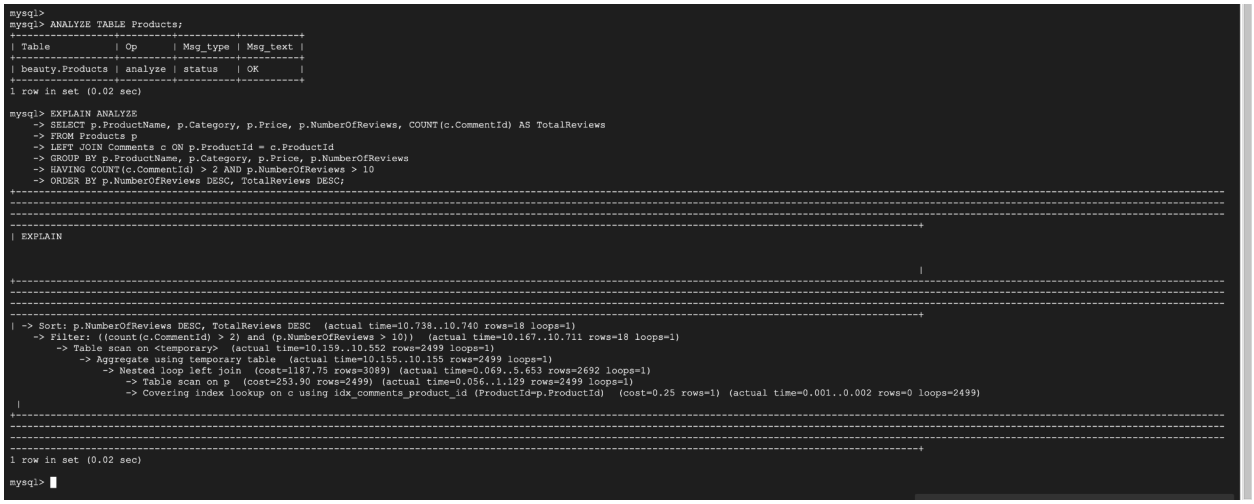
After using price as index

| -> Sort: p.NumberOfReviews DESC, TotalReviews DESC (actual time=10.088..10.090 rows=18 loops=1)
    -> Filter: ((count(c.CommentId) > 2) and (p.NumberOfReviews > 10)) (actual time=9.527..10.050 rows=18 loops=1)
        -> Table scan on <temporary> (actual time=9.519..9.897 rows=2499 loops=1)
            -> Aggregate using temporary table (actual time=9.516..9.516 rows=2499 loops=1)
                -> Nested loop left join (cost=1187.75 rows=3089) (actual time=0.073..5.412 rows=2692 loops=1)
                    -> Table scan on p (cost=253.90 rows=2499) (actual time=0.061..1.107 rows=2499 loops=1)
                    -> Covering index lookup on c using idx_comments_product_id (ProductId=p.ProductId) (cost=0.25 rows=1) (actual time=0.001..0.002 rows=0 loops=2499)



After using category as index

| -> Sort: p.NumberOfReviews DESC, TotalReviews DESC (actual time=10.738..10.740 rows=18 loops=1)
    -> Filter: ((count(c.CommentId) > 2) and (p.NumberOfReviews > 10)) (actual time=10.167..10.711 rows=18 loops=1)
        -> Table scan on <temporary> (actual time=10.159..10.552 rows=2499 loops=1)
            -> Aggregate using temporary table (actual time=10.155..10.155 rows=2499 loops=1)
                -> Nested loop left join (cost=1187.75 rows=3089) (actual time=0.069..5.653 rows=2692 loops=1)
                    -> Table scan on p (cost=253.90 rows=2499) (actual time=0.056..1.129 rows=2499 loops=1)
                    -> Covering index lookup on c using idx_comments_product_id (ProductId=p.ProductId) (cost=0.25 rows=1) (actual time=0.001..0.002 rows=0 loops=2499)



Index Analysis:

To evaluate the impact of different indexing configurations on Query 2, which identifies popular products by hit rate and reviews, we tested indices on attributes Date, Price, and Category while monitoring runtime and cost with EXPLAIN ANALYZE. In the original query without additional indices, the execution time for sorting was 10.121 ms, with a total runtime of 10.123 ms and a cost of 1187.75. Adding an index on Date slightly increased execution time to 10.180 ms, with an overall runtime of 10.182 ms and the same cost, indicating minimal impact since Date is not involved in the filter or sorting conditions. Introducing an index on Price achieved a minor runtime reduction, with the sorting step at 10.088 ms and the total execution time at 10.090 ms, though cost remained unchanged at 1187.75, showing marginal gains likely due to Price not being critical for filtering or grouping in this query. Lastly, indexing Category resulted in a runtime increase to 10.738 ms and a total execution time of 10.740 ms, with cost remaining at 1187.75, suggesting the added index did not optimize performance but instead added overhead, as Category is only used in grouping. From this analysis, we concluded that the original query without added indices was most efficient, as additional indices did not meaningfully reduce cost or runtime. This underscores the importance of indexing only those columns central to filtering and joins to avoid unnecessary indexing overhead.

Query 3: Compare Average Price of Products by Category Using Set Operators

```
SELECT p.Category, AVG(p.Price) AS AvgPrice, b.BrandName
FROM Products p
JOIN Brands b ON p.BrandId = b.BrandId
GROUP BY p.Category, b.BrandName
HAVING p.Category IN ('Highlighter', 'Blush', 'Powder', 'Lip Gloss', 'CC Cream', 'Eye Shadow');
```

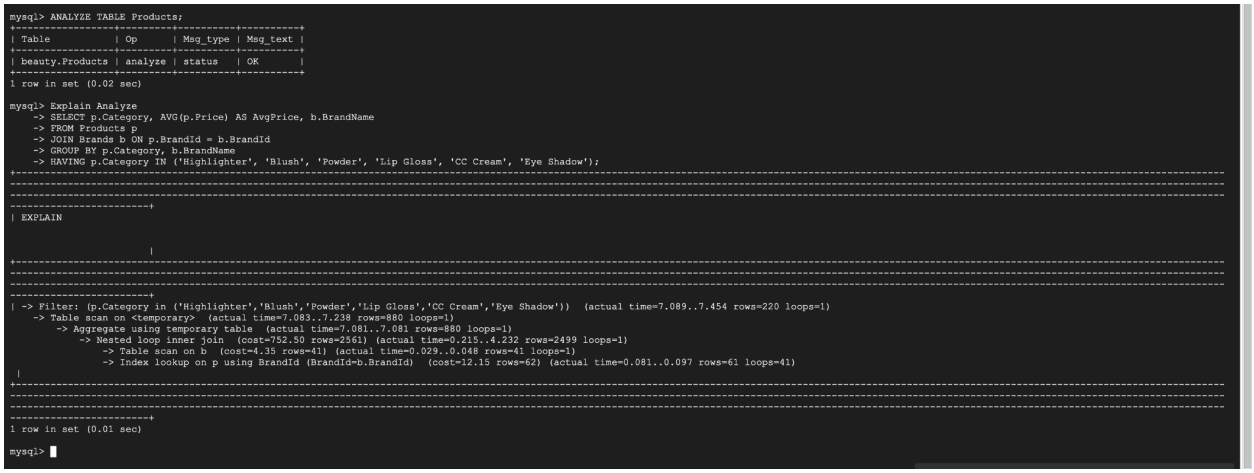
Original:

| -> Filter: (p.Category in ('Highlighter','Blush','Powder','Lip Gloss','CC Cream','Eye Shadow')) (actual time=6.675..7.054 rows=220 loops=1)

-> Table scan on <temporary> (actual time=6.669..6.819 rows=880 loops=1)
    -> Aggregate using temporary table (actual time=6.667..6.667 rows=880 loops=1)
        -> Nested loop inner join (cost=752.50 rows=2561) (actual time=0.198..3.941 rows=2499 loops=1)
            -> Table scan on b (cost=4.35 rows=41) (actual time=0.043..0.059 rows=41 loops=1)
            -> Index lookup on p using BrandId (BrandId=b.BrandId) (cost=12.15 rows=62) (actual time=0.079..0.090 rows=61 loops=41)

After using price as index

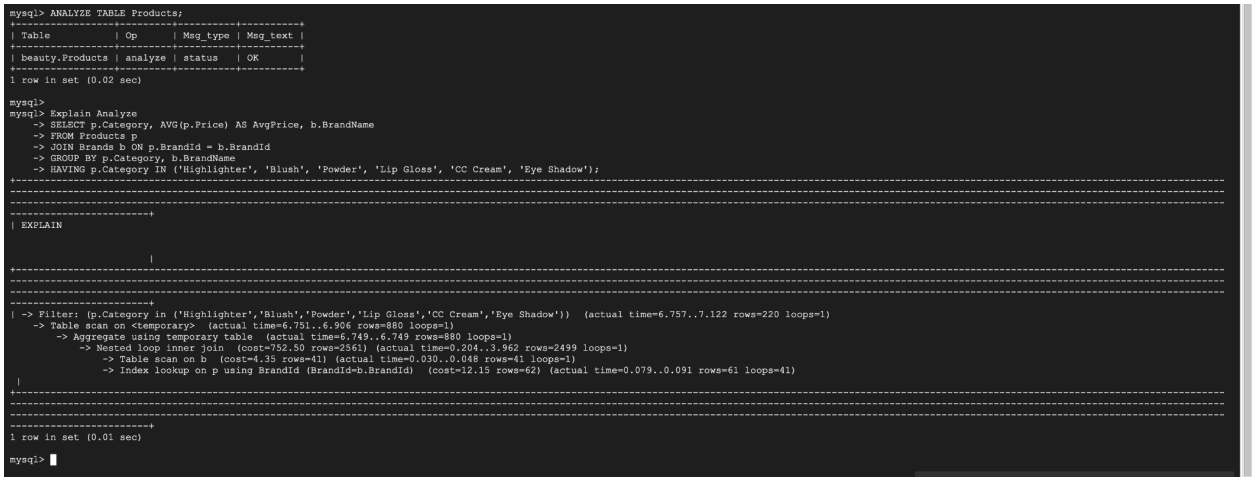
| -> Filter: (p.Category in ('Highlighter','Blush','Powder','Lip Gloss','CC Cream','Eye Shadow')) (actual time=7.089..7.454 rows=220 loops=1)
    -> Table scan on <temporary> (actual time=7.083..7.238 rows=880 loops=1)
        -> Aggregate using temporary table (actual time=7.081..7.081 rows=880 loops=1)
            -> Nested loop inner join (cost=752.50 rows=2561) (actual time=0.215..4.232 rows=2499 loops=1)
                -> Table scan on b (cost=4.35 rows=41) (actual time=0.029..0.048 rows=41 loops=1)
                -> Index lookup on p using BrandId (BrandId=b.BrandId) (cost=12.15 rows=62) (actual time=0.081..0.097 rows=61 loops=41)



After using category as index

| -> Filter: (p.Category in ('Highlighter','Blush','Powder','Lip Gloss','CC Cream','Eye Shadow')) (actual time=6.757..7.122 rows=220 loops=1)
    -> Table scan on <temporary> (actual time=6.751..6.906 rows=880 loops=1)
        -> Aggregate using temporary table (actual time=6.749..6.749 rows=880 loops=1)
            -> Nested loop inner join (cost=752.50 rows=2561) (actual time=0.204..3.962 rows=2499 loops=1)
                -> Table scan on b (cost=4.35 rows=41) (actual time=0.030..0.048 rows=41 loops=1)

-> Index lookup on p using BrandId (BrandId=b.BrandId) (cost=12.15 rows=62) (actual time=0.079..0.091 rows=61 loops=41)



After using category and price as index

| -> Filter: (p.Category in ('Highlighter','Blush','Powder','Lip Gloss','CC Cream','Eye Shadow')) (actual time=7.080..7.520 rows=220 loops=1)
-> Table scan on <temporary> (actual time=7.074..7.282 rows=880 loops=1)
-> Aggregate using temporary table (actual time=7.071..7.071 rows=880 loops=1)
-> Nested loop inner join (cost=752.50 rows=2561) (actual time=0.154..4.078 rows=2499 loops=1)
-> Table scan on b (cost=4.35 rows=41) (actual time=0.039..0.059 rows=41 loops=1)
-> Index lookup on p using BrandId (BrandId=b.BrandId) (cost=12.15 rows=62) (actual time=0.081..0.093 rows=61 loops=41)

Index Analysis:

To optimize Query 3, which compares the average price of products by category using set operators, we tested various indexing strategies on Price and Category attributes, and combinations of both, analyzing their impact on runtime and cost with EXPLAIN ANALYZE. The original query, without additional indices, had a sorting execution time of 6.675 ms and an overall runtime of 7.054 ms with a cost of 752.50. Adding an index on Price increased the runtime to 7.454 ms, while maintaining the cost at 752.50, suggesting that indexing Price alone did not benefit the query as expected since Price is only averaged but not directly filtered or sorted. Indexing Category alone offered a marginal runtime improvement, lowering it slightly to 7.122 ms, with the cost remaining at 752.50; this minor benefit likely resulted from Category being used as a filter in the HAVING clause. When combining both indices on Category and Price, the runtime increased to 7.520 ms, indicating no improvement and slight added overhead due to the extra indexing. Thus, based on these results, the original query configuration without additional indices was the most efficient in terms of both runtime and cost. This analysis emphasizes that adding indexes only on frequently filtered or joined columns aligns better with query optimization needs, avoiding the unnecessary cost associated with unused or minimally effective indices.

Query 4: Identify Trending Products Based on Recent Reviews

```
SELECT p.ProductName, p.Category, COUNT(c.CommentId) AS RecentReviews
FROM Products p
JOIN Comments c ON p.ProductId = c.ProductId
WHERE c.Date >= CURDATE() - INTERVAL 365 DAY
GROUP BY p.ProductName, p.Category
HAVING RecentReviews > 3
ORDER BY RecentReviews DESC;
```

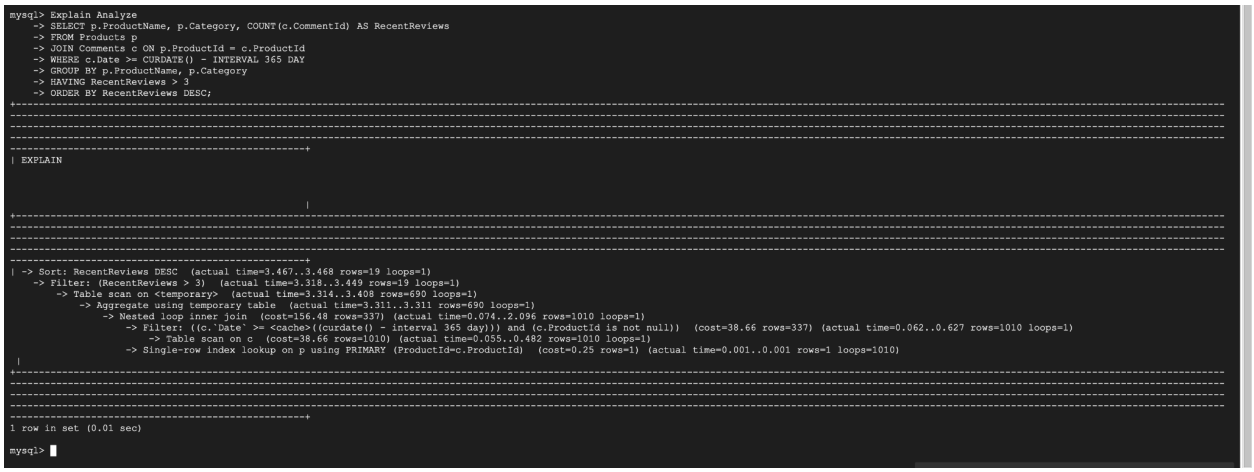
Original:

```
| -> Sort: RecentReviews DESC  (actual time=3.386..3.388 rows=19 loops=1)
    -> Filter: (RecentReviews > 3)  (actual time=3.236..3.367 rows=19 loops=1)
        -> Table scan on <temporary>  (actual time=3.231..3.324 rows=690 loops=1)
            -> Aggregate using temporary table  (actual time=3.229..3.229 rows=690 loops=1)
                -> Nested loop inner join  (cost=156.48 rows=337) (actual time=0.086..2.039
rows=1010 loops=1)
                    -> Filter: ((c.`Date` >= <cache>((curdate() - interval 365 day))) and (c.ProductId is
not null))  (cost=38.66 rows=337) (actual time=0.061..0.571 rows=1010 loops=1)
                        -> Table scan on c  (cost=38.66 rows=1010) (actual time=0.054..0.405
rows=1010 loops=1)
                    -> Single-row index lookup on p using PRIMARY (ProductId=c.ProductId)
(cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1010)|
```

After using price as index

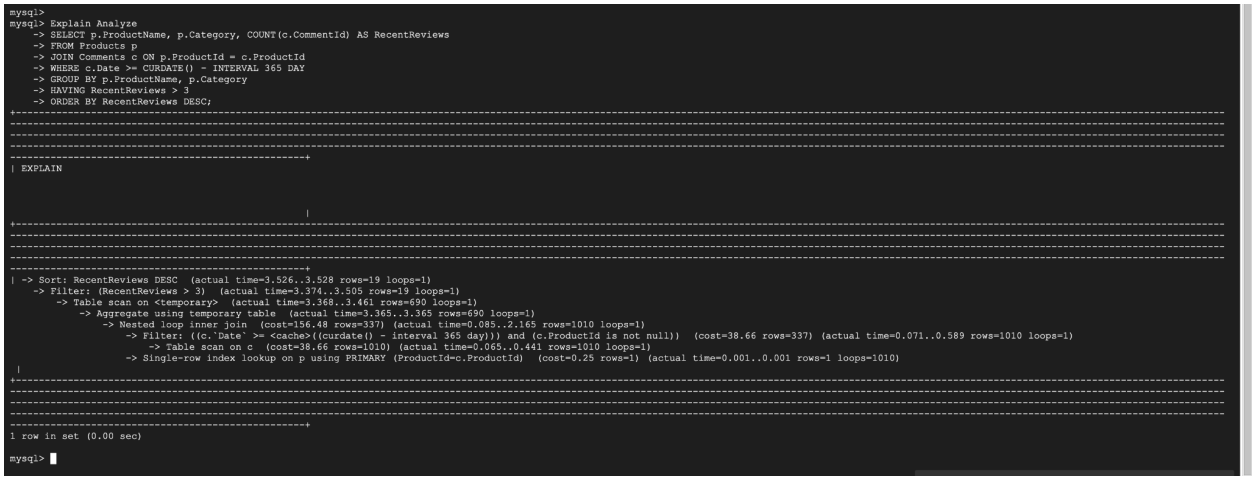
```
| -> Sort: RecentReviews DESC  (actual time=3.467..3.468 rows=19 loops=1)
    -> Filter: (RecentReviews > 3)  (actual time=3.318..3.449 rows=19 loops=1)
        -> Table scan on <temporary>  (actual time=3.314..3.408 rows=690 loops=1)
```

-> Aggregate using temporary table (actual time=3.311..3.311 rows=690 loops=1)
-> Nested loop inner join (cost=156.48 rows=337) (actual time=0.074..2.096 rows=1010 loops=1)
-> Filter: ((c.`Date` >= <cache>((curdate() - interval 365 day))) and (c.ProductId is not null)) (cost=38.66 rows=337) (actual time=0.062..0.627 rows=1010 loops=1)
-> Table scan on c (cost=38.66 rows=1010) (actual time=0.055..0.482 rows=1010 loops=1)
-> Single-row index lookup on p using PRIMARY (ProductId=c.ProductId) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1010)



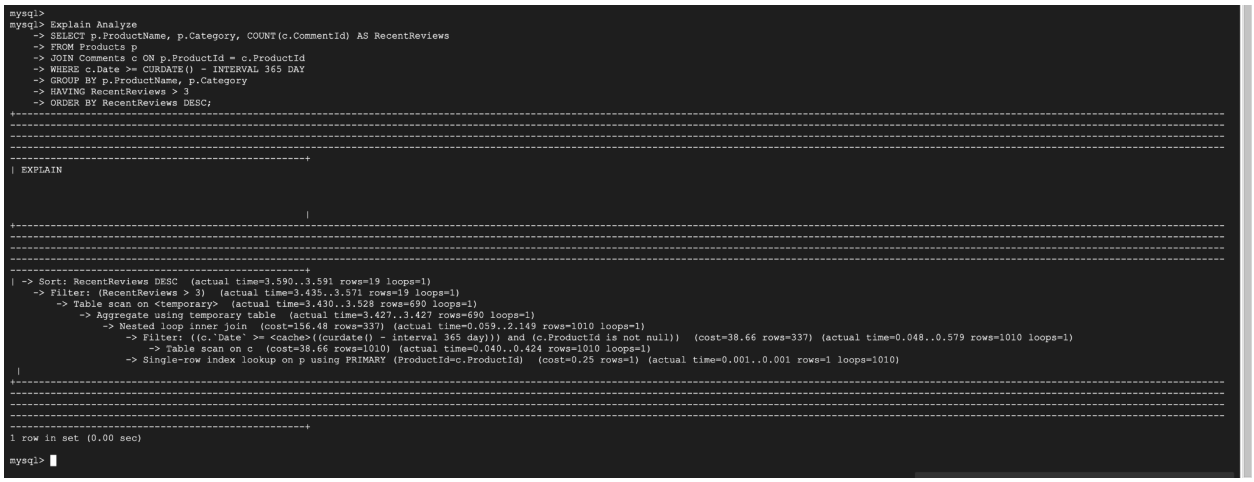
After using category as index

| -> Sort: RecentReviews DESC (actual time=3.526..3.528 rows=19 loops=1)
-> Filter: (RecentReviews > 3) (actual time=3.374..3.505 rows=19 loops=1)
-> Table scan on <temporary> (actual time=3.368..3.461 rows=690 loops=1)
-> Aggregate using temporary table (actual time=3.365..3.365 rows=690 loops=1)
-> Nested loop inner join (cost=156.48 rows=337) (actual time=0.085..2.165 rows=1010 loops=1)
-> Filter: ((c.`Date` >= <cache>((curdate() - interval 365 day))) and (c.ProductId is not null)) (cost=38.66 rows=337) (actual time=0.071..0.589 rows=1010 loops=1)
-> Table scan on c (cost=38.66 rows=1010) (actual time=0.065..0.441 rows=1010 loops=1)
-> Single-row index lookup on p using PRIMARY (ProductId=c.ProductId) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1010)



After using category and price as index

| -> Sort: RecentReviews DESC (actual time=3.590..3.591 rows=19 loops=1)
-> Filter: (RecentReviews > 3) (actual time=3.435..3.571 rows=19 loops=1)
-> Table scan on <temporary> (actual time=3.430..3.528 rows=690 loops=1)
-> Aggregate using temporary table (actual time=3.427..3.427 rows=690 loops=1)
-> Nested loop inner join (cost=156.48 rows=337) (actual time=0.059..2.149 rows=1010 loops=1)
-> Filter: ((c.`Date` >= <cache>((curdate() - interval 365 day))) and (c.ProductId is not null)) (cost=38.66 rows=337) (actual time=0.048..0.579 rows=1010 loops=1)
-> Table scan on c (cost=38.66 rows=1010) (actual time=0.040..0.424 rows=1010 loops=1)
-> Single-row index lookup on p using PRIMARY (ProductId=c.ProductId) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1010)



Index Analysis:

To assess the impact of indexing on Query 4, which identifies trending products based on recent reviews, we tested indices on Price, Category, and a combination of both, measuring the

resulting changes in runtime and cost using EXPLAIN ANALYZE. In the original query without added indices, the Sort step on RecentReviews executed in 3.386 ms, with a total runtime of 3.388 ms and a cost of 156.48. Adding an index on Price slightly increased execution time to 3.467 ms, while cost remained unchanged, indicating that indexing Price did not improve performance, as Price is not used for filtering or joining in this query. Indexing Category alone further increased the runtime to 3.528 ms without affecting the cost, reflecting minimal relevance since Category is only used in grouping. Combining indices on both Category and Price led to the highest runtime at 3.591 ms, again with no reduction in cost, likely due to the additional overhead from non-essential indices. Consequently, we concluded that the original query configuration—without additional indices—was the most efficient, as adding indices did not meaningfully reduce runtime or cost. This analysis underscores the importance of focusing indexing on columns that are critical for filtering, sorting, or joining, avoiding unnecessary indexing that may increase overhead without enhancing query performance.