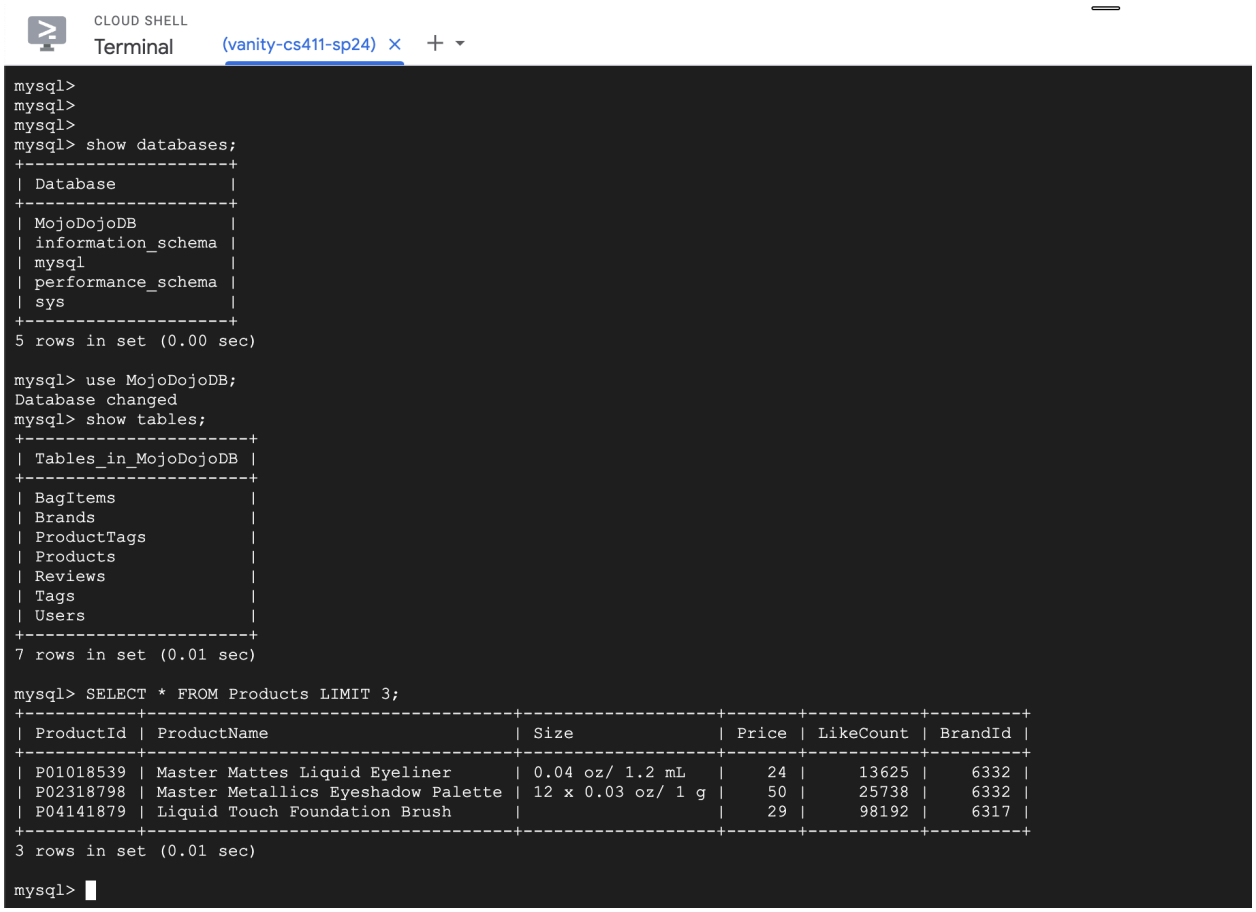
Project Track 1 Stage 3

Database Implementation

GCP Database Tables - “Screenshot of connection (terminal/command-line information)”



DDL commands for tables

```
CREATE TABLE Users(UserId INT PRIMARY KEY, UserName VARCHAR(30) NOT NULL, Password VARCHAR(30) NOT NULL, FirstName VARCHAR(255) NOT NULL, LastName VARCHAR(255) NOT NULL, Email VARCHAR(255) NOT NULL);

CREATE TABLE Brands(BrandId INT PRIMARY KEY, BrandName VARCHAR(255) NOT NULL, BrandRating REAL);

CREATE TABLE Products(ProductId VARCHAR(64) PRIMARY KEY, ProductName VARCHAR(255) NOT NULL, Size VARCHAR(128), Price INT, LikeCount INT, BrandId INT NOT NULL, FOREIGN KEY (BrandId) REFERENCES Brands(BrandId));
```

```
CREATE TABLE Reviews(ReviewId INT PRIMARY KEY, Rating INT, Text
VARCHAR(1000), Title VARCHAR(255), Date DATE, ProductId VARCHAR(64) NOT
NULL, FOREIGN KEY (ProductId) REFERENCES Products(ProductId));

CREATE TABLE Tags(TagId INT PRIMARY KEY, TagName VARCHAR(64) NOT NULL,
Standing INT);

CREATE TABLE BagItems(UserId INT NOT NULL, ProductId VARCHAR(64) NOT
NULL, DateAdded DATE, PRIMARY KEY (UserId, ProductId), FOREIGN KEY
(UserId) REFERENCES Users(UserId), FOREIGN KEY (ProductId) REFERENCES
Products(ProductId));

CREATE TABLE ProductTags(ProductId VARCHAR(64) NOT NULL, TagId INT NOT
NULL, PRIMARY KEY (ProductId, TagId), FOREIGN KEY (ProductId)
REFERENCES Products(ProductId), FOREIGN KEY (TagId) REFERENCES
Tags(TagId));

CREATE TABLE ProductClusters(ProductId VARCHAR(64) NOT NULL, ClusterId
INT NOT NULL, PRIMARY KEY (ProductId, ClusterId), FOREIGN KEY
(ProductId) REFERENCES Products(ProductId));
```

Screenshot of count query for proof of 1000 rows in three different tables

Products

```
mysql> SELECT count(*) FROM Products;
+----------+
| count(*) |
+----------+
|     8495 |
+----------+
1 row in set (0.06 sec)
```

Reviews

```
mysql> SELECT count(*) from Reviews;
+----------+
| count(*) |
+----------+
|    17170 |
+----------+
1 row in set (0.34 sec)
```

Tags

```
mysql> SELECT count(*) FROM Tags;
+----------+
| count(*) |
+----------+
|     4329 |
+----------+
1 row in set (0.00 sec)

mysql>
```

Advanced Queries + Indexing Analysis

Query 1: Most Popular Products (Products found most frequently in user bags that have been viewed in-app, have more than 1,000 likes, and have high review ratings)

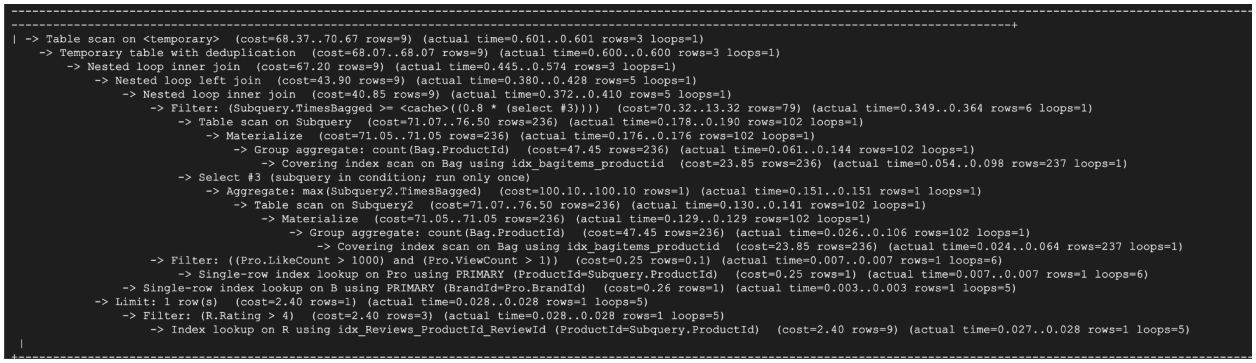
```
SELECT DISTINCT Pro.ProductName, B.BrandName, Subquery.TimesBagged
FROM
(SELECT Bag.ProductId, COUNT(Bag.ProductId) as TimesBagged FROM BagItems Bag
GROUP BY Bag.ProductId) Subquery
LEFT OUTER JOIN Products Pro ON Pro.ProductId = Subquery.ProductId
LEFT OUTER JOIN Brands B ON B.BrandId = Pro.BrandId
LEFT OUTER JOIN Reviews R on R.ProductId = Pro.ProductId

WHERE (Subquery.TimesBagged >= 0.8 * (SELECT MAX(Subquery2.TimesBagged) FROM
(SELECT Bag.ProductId, COUNT(Bag.ProductId) as TimesBagged FROM BagItems Bag
GROUP BY Bag.ProductId) Subquery2)) AND Pro.LikeCount > 1000 AND Pro.ViewCount
> 1 AND R.Rating > 4;
```

Output: Only 3 Most Popular Products in User bags (this table grows with user activity)

```
+-----------------------------------------------------------+---------------------+-------------+
| ProductName                                               | BrandName           | TimesBagged |
+-----------------------------------------------------------+---------------------+-------------+
| The Camellia Oil 2-in-1 Makeup Remover & Cleanser         | Tatcha              |          20 |
| Mini Superfood Antioxidant Cleanser                       | Youth To The People |          20 |
| Mini Revitalizing Supreme+ Youth Power Creme Moisturizer | Estée Lauder        |          19 |
+-----------------------------------------------------------+---------------------+-------------+
3 rows in set (0.01 sec)
```

EXPLAIN ANALYZE Performance before *new* indexes:

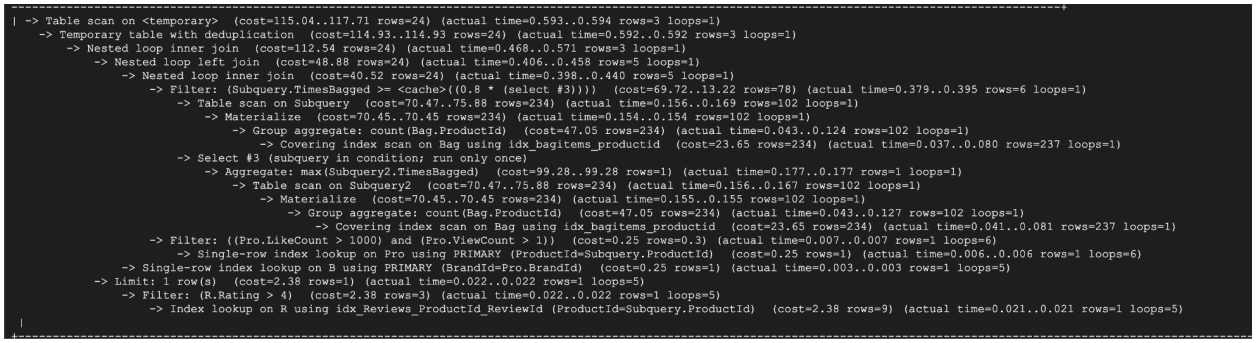


```
-> Table scan on <temporary>  (cost=68.37..70.67 rows=9) (actual time=0.601..0.601 rows=3 loops=1)
    -> Temporary table with deduplication  (cost=68.07..68.07 rows=9) (actual time=0.600..0.600 rows=3 loops=1)
        -> Nested loop inner join  (cost=67.20 rows=9) (actual time=0.445..0.574 rows=3 loops=1)
            -> Nested loop left join  (cost=43.90 rows=9) (actual time=0.380..0.428 rows=5 loops=1)
                -> Nested loop inner join  (cost=40.85 rows=9) (actual time=0.372..0.410 rows=5 loops=1)
                    -> Filter: (Subquery.TimesBagged >= <cache>((0.8 * (select #3))))  (cost=70.32..13.32 rows=79) (actual time=0.349..0.364 rows=6 loops=1)
                        -> Table scan on Subquery  (cost=71.07..76.50 rows=236) (actual time=0.178..0.190 rows=102 loops=1)
                            -> Materialize  (cost=71.05..71.05 rows=236) (actual time=0.176..0.176 rows=102 loops=1)
                                -> Group aggregate: count(Bag.ProductId)  (cost=47.45 rows=236) (actual time=0.061..0.144 rows=102 loops=1)
                                    -> Covering index scan on Bag using idx_bagitems_productid  (cost=23.85 rows=236) (actual time=0.054..0.098 rows=237 loops=1)
                    -> Select #3 (subquery in condition; run only once)
                        -> Aggregate: max(Subquery2.TimesBagged)  (cost=100.10..100.10 rows=1) (actual time=0.151..0.151 rows=1 loops=1)
                            -> Table scan on Subquery2  (cost=71.07..76.50 rows=236) (actual time=0.130..0.141 rows=102 loops=1)
                                -> Materialize  (cost=71.05..71.05 rows=236) (actual time=0.129..0.129 rows=102 loops=1)
                                    -> Group aggregate: count(Bag.ProductId)  (cost=47.45 rows=236) (actual time=0.026..0.106 rows=102 loops=1)
                                        -> Covering index scan on Bag using idx_bagitems_productid  (cost=23.85 rows=236) (actual time=0.024..0.064 rows=237 loops=1)
                -> Filter: ((Pro.LikeCount > 1000) and (Pro.ViewCount > 1))  (cost=0.25 rows=0.1) (actual time=0.007..0.007 rows=1 loops=6)
                    -> Single-row index lookup on Pro using PRIMARY (ProductId=Subquery.ProductId)  (cost=0.25 rows=1) (actual time=0.007..0.007 rows=1 loops=6)
            -> Single-row index lookup on B using PRIMARY (BrandId=Pro.BrandId)  (cost=0.26 rows=1) (actual time=0.003..0.003 rows=1 loops=5)
        -> Limit: 1 row(s)  (cost=2.40 rows=1) (actual time=0.028..0.028 rows=1 loops=5)
            -> Filter: (R.Rating > 4)  (cost=2.40 rows=3) (actual time=0.028..0.028 rows=1 loops=5)
                -> Index lookup on R using idx_Reviews_ProductId_ReviewId (ProductId=Subquery.ProductId)  (cost=2.40 rows=9) (actual time=0.027..0.028 rows=1 loops=5)
```

Total Cost: 70.67 + 68.07 + 67.20 + 43.90 + 40.85 + 70.32 + 76.50 + 71.05 + 47.45 + 23.85 + 100.10 + 76.50 +71.05 + 47.45 + 23.85 + 0.25 + 0.25 + 0.26 + 2.40 + 2.40 + 2.40 = **906.77**

Indexing Design 1

- New Index:
  `CREATE INDEX idx_Products_LikeCount ON Products(LikeCount);`
- EXPLAIN ANALYZE screenshot of this indexing design

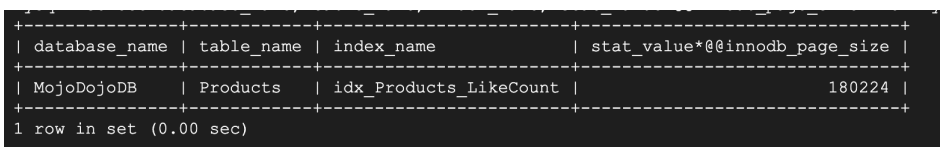


```
-> Table scan on <temporary>  (cost=115.04..117.71 rows=24) (actual time=0.593..0.594 rows=3 loops=1)
    -> Temporary table with deduplication  (cost=114.93..114.93 rows=24) (actual time=0.592..0.592 rows=3 loops=1)
        -> Nested loop inner join  (cost=112.54 rows=24) (actual time=0.468..0.571 rows=3 loops=1)
            -> Nested loop left join  (cost=48.88 rows=24) (actual time=0.406..0.458 rows=5 loops=1)
                -> Nested loop inner join  (cost=40.52 rows=24) (actual time=0.398..0.440 rows=5 loops=1)
                    -> Filter: (Subquery.TimesBagged >= <cache>((0.8 * (select #3))))  (cost=69.72..13.22 rows=78) (actual time=0.379..0.395 rows=6 loops=1)
                        -> Table scan on Subquery  (cost=70.47..75.88 rows=234) (actual time=0.156..0.169 rows=102 loops=1)
                            -> Materialize  (cost=70.45..70.45 rows=234) (actual time=0.154..0.154 rows=102 loops=1)
                                -> Group aggregate: count(Bag.ProductId)  (cost=47.05 rows=234) (actual time=0.043..0.124 rows=102 loops=1)
                                    -> Covering index scan on Bag using idx_bagitems_productid  (cost=23.65 rows=234) (actual time=0.037..0.080 rows=237 loops=1)
                    -> Select #3 (subquery in condition; run only once)
                        -> Aggregate: max(Subquery2.TimesBagged)  (cost=99.28..99.28 rows=1) (actual time=0.177..0.177 rows=1 loops=1)
                            -> Table scan on Subquery2  (cost=70.47..75.88 rows=234) (actual time=0.156..0.167 rows=102 loops=1)
                                -> Materialize  (cost=70.45..70.45 rows=234) (actual time=0.155..0.155 rows=102 loops=1)
                                    -> Group aggregate: count(Bag.ProductId)  (cost=47.05 rows=234) (actual time=0.043..0.127 rows=102 loops=1)
                                        -> Covering index scan on Bag using idx_bagitems_productid  (cost=23.65 rows=234) (actual time=0.041..0.081 rows=237 loops=1)
                -> Filter: ((Pro.LikeCount > 1000) and (Pro.ViewCount > 1))  (cost=0.25 rows=0.3) (actual time=0.007..0.007 rows=1 loops=6)
                    -> Single-row index lookup on Pro using PRIMARY (ProductId=Subquery.ProductId)  (cost=0.25 rows=1) (actual time=0.006..0.006 rows=1 loops=6)
            -> Single-row index lookup on B using PRIMARY (BrandId=Pro.BrandId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=5)
        -> Limit: 1 row(s)  (cost=2.38 rows=1) (actual time=0.022..0.022 rows=1 loops=5)
            -> Filter: (R.Rating > 4)  (cost=2.38 rows=3) (actual time=0.022..0.022 rows=1 loops=5)
                -> Index lookup on R using idx_Reviews_ProductId_ReviewId (ProductId=Subquery.ProductId)  (cost=2.38 rows=9) (actual time=0.021..0.021 rows=1 loops=5)
```

- Total Cost: 117.71 + 114.93 + 112.54 +48.88 + 40.52 + 69.7 + 75.88 + 70.45 + 47.05 + 23.6 + 99.28 + 75.88 + 70.45 + 47.05 + 23.65 + 0.25 + 0.25 + 0.25 + 2.38 + 2.38 + 2.38 = **1045.46**

Pros/Performance Gains: No performance gains.

The LikeCount attribute is only used in one WHERE clause, and so ideally the index should improve filtering performance. However, in this case, LikeCount index is creating performance degradations, perhaps because it is quite large. Other reasons for degradations explained below



```
+---------------+------------+------------------------+-------------------------------+
| database_name | table_name | index_name             | stat_value*@@innodb_page_size |
+---------------+------------+------------------------+-------------------------------+
| MojoDojoDB    | Products   | idx_Products_LikeCount |                        180224 |
+---------------+------------+------------------------+-------------------------------+
1 row in set (0.00 sec)
```

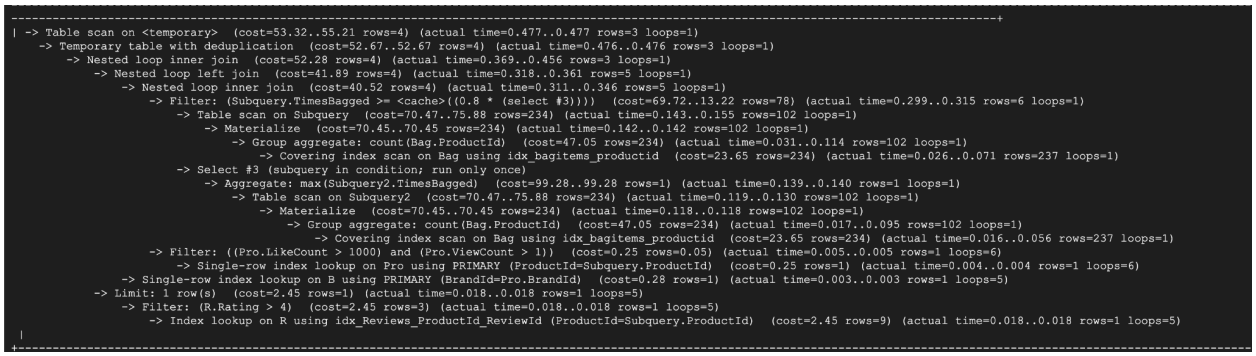
Cons/Degradations: Degradations present. Adding this extra index will take up more storage space in the BTREE, and add an extra layer of complexity to queries. Since we already have indexes for all the primary keys, it might not benefit performance significantly to add another index for filtering LikeCount.

Additionally, adding an index can interfere with the MySQL engine developing an efficient query optimization plan. It is possible that because we added this new index, MySQL had to work around it to develop a less efficient query optimization plan, for example, a plan with more locking.

Indexing Design 2

- New Index:
  `CREATE INDEX idx_Products_ViewCount ON Products(ViewCount);`

- EXPLAIN ANALYZE screenshot of this indexing design



```
-> Table scan on <temporary>  (cost=53.32..55.21 rows=4) (actual time=0.477..0.477 rows=3 loops=1)
    -> Temporary table with deduplication  (cost=52.67..52.67 rows=4) (actual time=0.476..0.476 rows=3 loops=1)
        -> Nested loop inner join  (cost=52.28 rows=4) (actual time=0.369..0.456 rows=3 loops=1)
            -> Nested loop left join  (cost=41.89 rows=4) (actual time=0.318..0.361 rows=5 loops=1)
                -> Nested loop inner join  (cost=40.52 rows=4) (actual time=0.311..0.346 rows=5 loops=1)
                    -> Filter: (Subquery.TimesBagged >= <cache>((0.8 * (select #3))))  (cost=69.72..13.22 rows=78) (actual time=0.299..0.315 rows=6 loops=1)
                        -> Table scan on Subquery  (cost=70.47..75.88 rows=234) (actual time=0.143..0.155 rows=102 loops=1)
                            -> Materialize  (cost=70.45..70.45 rows=234) (actual time=0.142..0.142 rows=102 loops=1)
                                -> Group aggregate: count(Bag.ProductId)  (cost=47.05 rows=234) (actual time=0.031..0.114 rows=102 loops=1)
                                    -> Covering index scan on Bag using idx_bagitems_productid  (cost=23.65 rows=234) (actual time=0.026..0.071 rows=237 loops=1)
                        -> Select #3 (subquery in condition; run only once)
                            -> Aggregate: max(Subquery2.TimesBagged)  (cost=99.28..99.28 rows=1) (actual time=0.139..0.140 rows=1 loops=1)
                                -> Table scan on Subquery2  (cost=70.47..75.88 rows=234) (actual time=0.119..0.130 rows=102 loops=1)
                                    -> Materialize  (cost=70.45..70.45 rows=234) (actual time=0.118..0.118 rows=102 loops=1)
                                        -> Group aggregate: count(Bag.ProductId)  (cost=47.05 rows=234) (actual time=0.017..0.095 rows=102 loops=1)
                                            -> Covering index scan on Bag using idx_bagitems_productid  (cost=23.65 rows=234) (actual time=0.016..0.056 rows=237 loops=1)
                    -> Filter: ((Pro.LikeCount > 1000) and (Pro.ViewCount > 1))  (cost=0.25 rows=0.05) (actual time=0.005..0.005 rows=1 loops=6)
                        -> Single-row index lookup on Pro using PRIMARY (ProductId=Subquery.ProductId)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=6)
                -> Single-row index lookup on B using PRIMARY (BrandId=Pro.BrandId)  (cost=0.28 rows=1) (actual time=0.003..0.003 rows=1 loops=5)
            -> Limit: 1 row(s)  (cost=2.45 rows=1) (actual time=0.018..0.018 rows=1 loops=5)
                -> Filter: (R.Rating > 4)  (cost=2.45 rows=3) (actual time=0.018..0.018 rows=1 loops=5)
                    -> Index lookup on R using idx_Reviews_ProductId_ReviewId (ProductId=Subquery.ProductId)  (cost=2.45 rows=9) (actual time=0.018..0.018 rows=1 loops=5)
```

- Total Cost: 52.67 + 55.21 + 52.28 + 41.89 + 40.52 + 69.72 + 75.88 + 70.45 + 47.05 + 23.65 + 99.28 + 75.88 + 70.45 + 47.05 + 23.65 + 0.25 + 0.25 + 0.28 + 2.45 + 2.45 + 2.45 = **853.76**

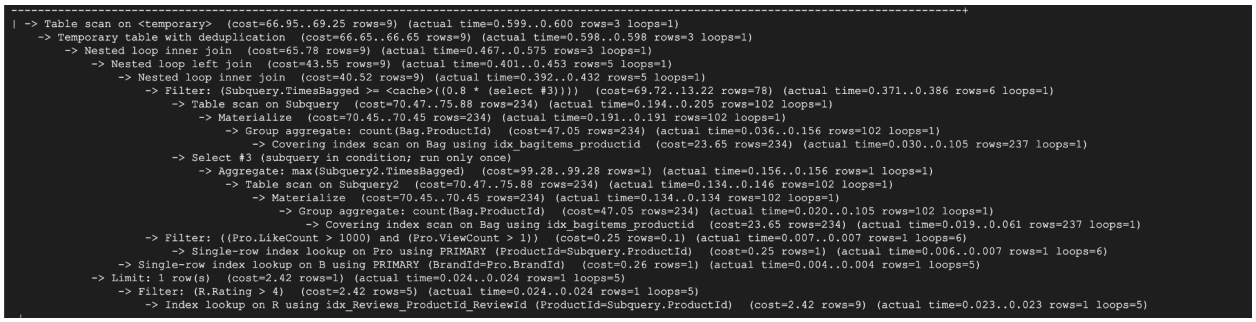
Pros/Performance Gains: Performance gains present! Since the ViewCount index is used in the main WHERE clause, indexing on ViewCount improves the query's filtering performance.

Cons/Degradations: No cons/degradations. Since we already have a ProductId index in the Brands table (Primary key) and in the Products table (foreign key), it might not benefit performance significantly to add an extra index for ViewCount. However in this case, it does improve performance a slight bit.

Indexing Design 3

- New Index: `CREATE INDEX idx_Reviews_Rating ON Reviews(Rating);`

- EXPLAIN ANALYZE screenshot of this indexing design



```
-> Table scan on <temporary>  (cost=66.95..69.25 rows=9) (actual time=0.599..0.600 rows=3 loops=1)
    -> Temporary table with deduplication  (cost=66.65..66.65 rows=9) (actual time=0.598..0.598 rows=3 loops=1)
        -> Nested loop inner join  (cost=65.78 rows=9) (actual time=0.467..0.575 rows=3 loops=1)
            -> Nested loop left join  (cost=43.55 rows=9) (actual time=0.401..0.453 rows=5 loops=1)
                -> Nested loop inner join  (cost=40.52 rows=9) (actual time=0.392..0.432 rows=5 loops=1)
                    -> Filter: (Subquery.TimesBagged >= <cache>((0.8 * (select #3))))  (cost=69.72..13.22 rows=78) (actual time=0.371..0.386 rows=6 loops=1)
                        -> Table scan on Subquery  (cost=70.47..75.88 rows=234) (actual time=0.194..0.205 rows=102 loops=1)
                            -> Materialize  (cost=70.45..70.45 rows=234) (actual time=0.191..0.191 rows=102 loops=1)
                                -> Group aggregate: count(Bag.ProductId)  (cost=47.05 rows=234) (actual time=0.036..0.156 rows=102 loops=1)
                                    -> Covering index scan on Bag using idx_bagitems_productid  (cost=23.65 rows=234) (actual time=0.030..0.105 rows=237 loops=1)
                        -> Select #3 (subquery in condition; run only once)
                            -> Aggregate: max(Subquery2.TimesBagged)  (cost=99.28..99.28 rows=1) (actual time=0.156..0.156 rows=1 loops=1)
                                -> Table scan on Subquery2  (cost=70.47..75.88 rows=234) (actual time=0.134..0.146 rows=102 loops=1)
                                    -> Materialize  (cost=70.45..70.45 rows=234) (actual time=0.134..0.134 rows=102 loops=1)
                                        -> Group aggregate: count(Bag.ProductId)  (cost=47.05 rows=234) (actual time=0.020..0.105 rows=102 loops=1)
                                            -> Covering index scan on Bag using idx_bagitems_productid  (cost=23.65 rows=234) (actual time=0.019..0.061 rows=237 loops=1)
                    -> Filter: ((Pro.LikeCount > 1000) and (Pro.ViewCount > 1))  (cost=0.25 rows=0.1) (actual time=0.007..0.007 rows=1 loops=6)
                        -> Single-row index lookup on Pro using PRIMARY (ProductId=Subquery.ProductId)  (cost=0.25 rows=1) (actual time=0.006..0.007 rows=1 loops=6)
                -> Single-row index lookup on B using PRIMARY (BrandId=Pro.BrandId)  (cost=0.26 rows=1) (actual time=0.004..0.004 rows=1 loops=5)
            -> Limit: 1 row(s)  (cost=2.42 rows=1) (actual time=0.024..0.024 rows=1 loops=5)
                -> Filter: (R.Rating > 4)  (cost=2.42 rows=5) (actual time=0.024..0.024 rows=1 loops=5)
                    -> Index lookup on R using idx_Reviews_ProductId_ReviewId (ProductId=Subquery.ProductId)  (cost=2.42 rows=9) (actual time=0.023..0.023 rows=1 loops=5)
```

- Total Cost: 69.25 + 66.65 + 65.78 + 43.55 + 40.52 + 69.72 + 75.88 + 70.45 + 47.0 + 23.65 + 99.28 + 75.88 + 70.45 + 47.05 + 23.65 +0.25 + 0.25 + 0.26 + 2.42 + 2.42 + 2.42 = **896.78**

- Pros and cons, performance gains and degradations of this indexing design

Pros/Performance Gains: Performance gains present! Since the Rating index is used in the main WHERE clause, indexing on Rating improves the query's filtering performance.

Cons/Degradations: No cons/degradations. Since we already have a ProductId index in the Reviews table (Primary key) and in the Products table (foreign key), it might not benefit performance significantly to add an extra index for Rating. But, because we are rigorously filtering on rating, it improves performance.

Final Indexing Choice and Why

Indexing Design #2 is our final indexing choice as it has the lowest cost — 853 as opposed to the original cost of 906 without the index. Because our query includes ViewCount in the WHERE clause, this query will filter thousands of records by ViewCount. Adding an index on this non-primary key attribute will improve the performance of this filtering

---

Query 2: User Bag Overview During Friends Search

```
SELECT UserName, COUNT(ProductId) AS NumProductsInBag
FROM Users LEFT OUTER JOIN BagItems ON Users.UserId = BagItems.UserId
GROUP BY (Users.UserId)
ORDER BY LEAST(LEVENSHTEIN(UserName, *search*), LEVENSHTEIN(FirstName,
*search*), LEVENSHTEIN(LastName, *search*))
LIMIT 15;
```

Output: example query replaces *search* with “kylie”

```
+-----------------+-------------------+
| UserName        | NumProductsInBag  |
+-----------------+-------------------+
| kyliejenner     |                 4 |
| julie           |                 3 |
| karlie          |                 4 |
| ria             |                 4 |
| natalie         |                 4 |
| bellahadid      |                 4 |
| elvis           |                 3 |
| Hyram           |                 4 |
| tati            |                 1 |
| i.love.skincare |                 4 |
| taylor          |                 2 |
| nitya           |                 3 |
| oju             |                 4 |
| marilyn         |                 4 |
| i.love.makeup   |                 4 |
+-----------------+-------------------+
15 rows in set (0.02 sec)
```

EXPLAIN ANALYZE Performance before adding indexes:

```
| -> Limit: 15 row(s)  (actual time=15.981..15.983 rows=15 loops=1)
    -> Sort: least(LEVENSHTEIN(Users.UserName,'kylie'),LEVENSHTEIN(Users.FirstName,'kylie'),LEVENSHTEIN(Users.LastName,'kylie')), limit input to 15 row(s) per chunk  (actual time=15.980..15.981 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=15.957..15.961 rows=21 loops=1)
            -> Aggregate using temporary table  (actual time=15.955..15.955 rows=21 loops=1)
                -> Nested loop left join  (cost=15.52 rows=73) (actual time=0.079..0.286 rows=73 loops=1)
                    -> Table scan on Users  (cost=2.35 rows=21) (actual time=0.040..0.063 rows=21 loops=1)
                    -> Covering index lookup on BagItems using PRIMARY (UserId=Users.UserId)  (cost=0.30 rows=3) (actual time=0.005..0.009 rows=3 loops=21)
|
```

Total Cost: 15.52 + 2.35 + 0.30 = 18.17

Indexing Design 1

- New Index: CREATE INDEX idx_Users_UserName ON Users (UserName);
- EXPLAIN ANALYZE screenshot of this indexing design

```
| -> Limit: 15 row(s)  (actual time=14.969..14.971 rows=15 loops=1)
    -> Sort: least(LEVENSHTEIN(Users.UserName,'kylie'),LEVENSHTEIN(Users.FirstName,'kylie'),LEVENSHTEIN(Users.LastName,'kylie')), limit input to 15 row(s) per chunk  (actual time=14.968..14.969 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=14.936..14.940 rows=21 loops=1)
            -> Aggregate using temporary table  (actual time=14.934..14.934 rows=21 loops=1)
                -> Nested loop left join  (cost=15.52 rows=73) (actual time=0.038..0.196 rows=73 loops=1)
                    -> Table scan on Users  (cost=2.35 rows=21) (actual time=0.025..0.041 rows=21 loops=1)
                    -> Covering index lookup on BagItems using PRIMARY (UserId=Users.UserId)  (cost=0.30 rows=3) (actual time=0.004..0.007 rows=3 loops=21)
|
```

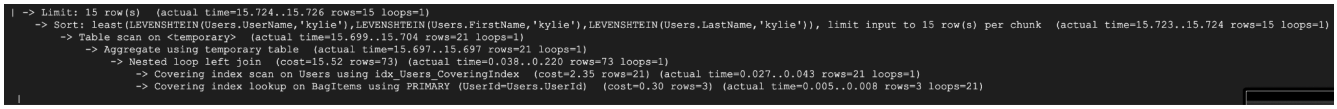
- Total Cost: 15.52 + 2.35 + 0.30 = 18.17
- Pros and cons, performance gains and degradations of this indexing design

We expect no performance improvement or degradation. This is likely due to the fact that Users.UserName is only used in the Levenshtein distance calculation. Due to the primary key

UserId in the User table, and the fact that the UserId is the only attribute being used in the JOIN and GROUP BY clauses, adding an index on the UserName attribute of the Users table will only take up more storage space without improving performance.

Indexing Design 2

- New Index: CREATE INDEX idx_Users_CoveringIndex ON Users (UserName, FirstName, LastName);
- EXPLAIN ANALYZE screenshot of this indexing design

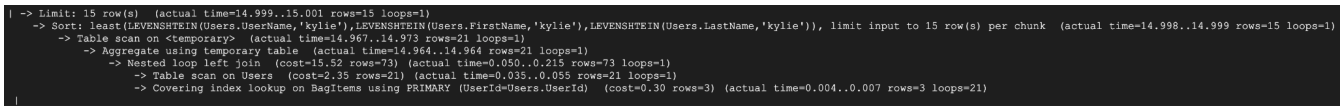


- Total Cost: 15.52 + 2.35 + 0.30 = 18.17
- Pros and cons, performance gains and degradations of this indexing design

We expect no performance improvement or degradation. This is likely due to the fact that Users.UserName, Users.FirstName, and Users.LastName are only used in the Levenshtein distance calculations. Due to the primary key UserId in the User table, and the fact that the UserId is the only attribute being used in the JOIN and GROUP BY clauses, adding an index on the UserName, FirstName, and LastName attributes of the Users table will only take up more storage space without improving performance.

Indexing Design 3

- New Index: CREATE INDEX idx_Users_BagItems_JoinIndex ON BagItems (UserId, ProductId);
- EXPLAIN ANALYZE screenshot of this indexing design



- Total Cost: 15.52 + 2.35 + 0.30 = 18.17
- Pros and cons, performance gains and degradations of this indexing design

We expect no performance improvement or degradation. This is likely due to the fact that BagItems.UserId is used in the JOIN clause but is already a primary key, and BagItems.ProductId is not used in the query. Since ProductId is not used in the query, adding an index on it only takes up more storage space without improving performance.

Final Indexing Choice and Why

Justification: The final index design we chose for Advanced Query #2 is the default index (the primary key UserId in the Users table and the primary/foreign key UserId in the BagItems table). We chose this index because all of the other indexing designs we tried did not impact the cost of the query. This is because this query simply uses the primary key UserId in both the JOIN and GROUP BY clauses, while any other attributes are simply used in the SELECT or ORDER BY clauses, and thus do not have an effect on the cost.

Query 3: Users also bagged

```
SELECT BI.ProductId AS OtherPID,
       P.ProductName,
       AVG(R.Rating) AS AverageRating,
       COUNT(R.ReviewId) AS NumberOfReviews
FROM BagItems BI
    JOIN Products P
        ON BI.ProductId = P.ProductId
    JOIN
    (
        SELECT ProductId,
               MAX(ReviewId) AS LatestReviewId
        FROM Reviews
        GROUP BY ProductId
    ) AS LatestReview
        ON BI.ProductId = LatestReview.ProductId
    JOIN Reviews R
        ON LatestReview.LatestReviewId = R.ReviewId
WHERE BI.ProductId != 'P501265'
      AND BI.UserId IN (
          SELECT UserId FROM BagItems WHERE ProductId = 'P501265')
      AND R.Rating > 1
GROUP BY BI.ProductId,
         P.ProductName
ORDER BY BI.ProductId,
         AverageRating DESC;
```

Output: example query replaces *current-product* with ‘P501265’

OtherPID	ProductName	AverageRating	NumberOfReviews
P392235	The Camellia Oil 2-in-1 Makeup Remover & Cleanser	5.0000	16
P423259	Cicapair Tiger Grass Serum	3.0000	1
P429659	Squalane + Hyaluronic Toning Mist	5.0000	1
P438643	The Balance pH Balancing Gel Cleanser	5.0000	1
P441644	Mini Superfood Antioxidant Cleanser	2.0000	17
P465741	Wild Huckleberry 8-Acid Polishing Peel Mask	5.0000	1
P467615	Cicapair Tiger Grass Sleepair Intensive Mask	5.0000	10
P469087	Flash Nap Instant Revival Priming Eye Gel-Cream With Green Tea + Persian Silk Tree	4.0000	1
P480278	Rapid Radiance Set	4.0000	1
P481084	Mini Revitalizing Supreme+ Youth Power Creme Moisturizer	5.0000	17
P481817	Beauty Elixir Prep, Set, Glow Face Mist	4.0000	1
P505020	The POREfessional Good Cleanup Foaming Cleanser	5.0000	1

12 rows in set (0.02 sec)

EXPLAIN ANALYZE Performance before adding indexes:

Total Cost: 1083.47 + 578.06 + 72.65 + 23.37 + 2.25 + 0.38 + 0.38 + 0.25 + 825.55 + 914.45 + 734.35 + 0.25 + 0.25 = 4235.66

```
| -> Sort: BI.ProductId  (actual time=16.188..16.189 rows=9 loops=1)
    -> Table scan on <temporary>  (actual time=16.167..16.169 rows=9 loops=1)
        -> Aggregate using temporary table  (actual time=16.165..16.165 rows=9 loops=1)
            -> Nested loop inner join  (cost=1041.91 rows=1392) (actual time=15.347..16.085 rows=55 loops=1)
                -> Nested loop inner join  (cost=554.88 rows=1392) (actual time=15.334..15.977 rows=55 loops=1)
                    -> Nested loop inner join  (cost=67.85 rows=131) (actual time=0.033..0.409 rows=146 loops=1)
                        -> Nested loop inner join  (cost=21.95 rows=131) (actual time=0.026..0.161 rows=146 loops=1)
                            -> Covering index lookup on BagItems using idx_bagitems_productid (ProductId='P501265')  (cost=2.14 rows=17) (actual time=0.015..0.022 rows=17 loops=1)
                            -> Filter: (BI.ProductId <> 'P501265')  (cost=0.38 rows=8) (actual time=0.004..0.007 rows=9 loops=17)
                                -> Covering index lookup on BI using PRIMARY (UserId=BagItems.UserId)  (cost=0.38 rows=8) (actual time=0.004..0.006 rows=10 loops=17)
                        -> Single-row index lookup on P using PRIMARY (ProductId=BI.ProductId)  (cost=0.25 rows=1) (actual time=0.001..0.002 rows=1 loops=146)
                    -> Filter: (LatestReview.LatestReviewId is not null)  (cost=826.25..2.66 rows=11) (actual time=0.106..0.106 rows=0 loops=146)
                        -> Index lookup on LatestReview using <auto_key1> (ProductId=BI.ProductId)  (actual time=0.106..0.106 rows=0 loops=146)
                            -> Materialize  (cost=911.95..911.95 rows=1801) (actual time=15.271..15.271 rows=1800 loops=1)
                                -> Covering index skip scan for grouping on Reviews using ProductId  (cost=731.85 rows=1801) (actual time=0.018..12.924 rows=1800 loops=1)
            -> Single-row index lookup on R using PRIMARY (ReviewId=LatestReview.LatestReviewId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=55)
 |
```

Indexing Design 1

- New Index: CREATE INDEX ByRating on Reviews(Rating);
- EXPLAIN ANALYZE screenshot of this indexing design

```
| -> Sort: BI.ProductId, AverageRating DESC  (actual time=16.276..16.277 rows=12 loops=1)
    -> Table scan on <temporary>  (actual time=16.224..16.227 rows=12 loops=1)
        -> Aggregate using temporary table  (actual time=16.221..16.221 rows=12 loops=1)
            -> Nested loop inner join  (cost=1083.47 rows=722) (actual time=15.136..16.109 rows=68 loops=1)
                -> Nested loop inner join  (cost=578.06 rows=1444) (actual time=15.116..15.975 rows=68 loops=1)
                    -> Nested loop inner join  (cost=72.65 rows=141) (actual time=0.076..0.601 rows=184 loops=1)
                        -> Nested loop inner join  (cost=23.37 rows=141) (actual time=0.064..0.233 rows=184 loops=1)
                            -> Covering index lookup on BagItems using idx_bagitems_productid (ProductId='P501265')  (cost=2.25 rows=18) (actual time=0.029..0.037 rows=18 loops=1)
                            -> Filter: (BI.ProductId <> 'P501265')  (cost=0.38 rows=8) (actual time=0.005..0.010 rows=10 loops=18)
                                -> Covering index lookup on BI using PRIMARY (UserId=BagItems.UserId)  (cost=0.38 rows=8) (actual time=0.005..0.008 rows=11 loops=18)
                        -> Single-row index lookup on P using PRIMARY (ProductId=BI.ProductId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=184)
                    -> Filter: (LatestReview.LatestReviewId is not null)  (cost=825.55..2.57 rows=10) (actual time=0.083..0.083 rows=0 loops=184)
                        -> Index lookup on LatestReview using <auto_key1> (ProductId=BI.ProductId)  (actual time=0.083..0.083 rows=0 loops=184)
                            -> Materialize  (cost=914.45..914.45 rows=1801) (actual time=14.990..14.990 rows=1800 loops=1)
                                -> Covering index skip scan for grouping on Reviews using idx_Reviews_ProductId_ReviewId  (cost=734.35 rows=1801) (actual time=0.028..12.590 rows=180
0 loops=1)
                -> Filter: (R.Rating > 1)  (cost=0.25 rows=0.5) (actual time=0.002..0.002 rows=1 loops=68)
                    -> Single-row index lookup on R using PRIMARY (ReviewId=LatestReview.LatestReviewId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=68)
 |
```

- Total Cost: 1083.47 + 578.06 + 72.65 + 23.37 + 2.25 + 0.38 + 0.38 + 0.25 + 825.55 + 914.45 + 734.35 + 0.25 + 0.25 = 4235.66
- Pros and cons, performance gains and degradations of this indexing design: This indexing has potential to speed up this query quite a bit. Although the above cost analysis did not show any improvement, there is a chance that with a large number of reviews, there could be improvement in the cost and time. Since Rating is in the WHERE clause and in the Order By clause, those operations will speed up with this design. This index would improve the filtering efficiency, especially if there are many rows with ratings greater than 1. One con about this query is that it may be less useful as the Reviews table gets modified because it would require a lot of maintenance. Since Ratings is only referenced in a couple of places, it might not be worth implementing this index design that would require consistent maintenance.

Indexing Design 2

- New Index: CREATE INDEX ByName on Products(ProductName);
- EXPLAIN ANALYZE screenshot of this indexing design

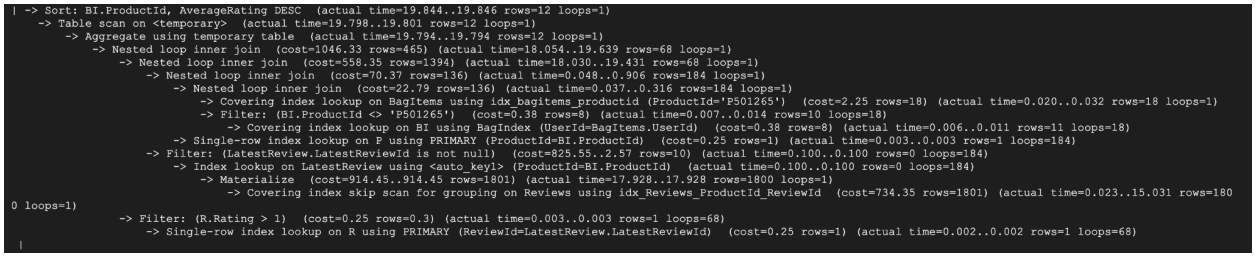
```
| -> Sort: BI.ProductId, AverageRating DESC  (actual time=16.225..16.226 rows=12 loops=1)
    -> Table scan on <temporary>  (actual time=16.187..16.189 rows=12 loops=1)
        -> Aggregate using temporary table  (actual time=16.184..16.184 rows=12 loops=1)
            -> Nested loop inner join  (cost=1083.47 rows=481) (actual time=15.101..16.078 rows=68 loops=1)
                -> Nested loop inner join  (cost=578.06 rows=1444) (actual time=15.078..15.938 rows=68 loops=1)
                    -> Nested loop inner join  (cost=72.65 rows=141) (actual time=0.047..0.563 rows=184 loops=1)
                        -> Nested loop inner join  (cost=23.37 rows=141) (actual time=0.037..0.206 rows=184 loops=1)
                            -> Covering index lookup on BagItems using idx_bagitems_productid (ProductId='P501265')  (cost=2.25 rows=18) (actual time=0.023..0.031 rows=18 loops=1)
                            -> Filter: (BI.ProductId <> 'P501265')  (cost=0.38 rows=8) (actual time=0.004..0.009 rows=10 loops=18)
                                -> Covering index lookup on BI using PRIMARY (UserId=BagItems.UserId)  (cost=0.38 rows=8) (actual time=0.004..0.007 rows=11 loops=18)
                        -> Single-row index lookup on P using PRIMARY (ProductId=BI.ProductId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=184)
                    -> Filter: (LatestReview.LatestReviewId is not null)  (cost=825.55..2.57 rows=10) (actual time=0.083..0.083 rows=0 loops=184)
                        -> Index lookup on LatestReview using <auto_key1> (ProductId=BI.ProductId)  (actual time=0.083..0.083 rows=0 loops=184)
                            -> Materialize  (cost=914.45..914.45 rows=1801) (actual time=14.979..14.979 rows=1800 loops=1)
                                -> Covering index skip scan for grouping on Reviews using idx_Reviews_ProductId_ReviewId  (cost=734.35 rows=1801) (actual time=0.023..12.592 rows=180
0 loops=1)
                -> Filter: (R.Rating > 1)  (cost=0.25 rows=0.3) (actual time=0.002..0.002 rows=1 loops=68)
                    -> Single-row index lookup on R using PRIMARY (ReviewId=LatestReview.LatestReviewId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=68)
 |
```

- Total Cost: 1083.47 + 578.06 + 72.65 + 23.37 + 2.25 + 0.38 + 0.38 + 0.25 + 825.55 + 914.45 + 734.35 + 0.25 + 0.25 = 4235.66

- Pros and cons, performance gains and degradations of this indexing design: This indexing design did not facilitate any performance gains or degradations. The impact of the index remains neutral after the analysis, with a total cost of 4235.66. Since ProductName is often referenced in the scope of our application, it could be useful for other queries, but this query does not involve ProductName in major operations. The query does order the results using ProductName, which could be enhanced by the index, but this was not proved in the above cost analysis. The index might also improve search performance when searching products by name. The query identifies distinct products based on ProductId, not based on ProductName, so the index does not affect the cost.

Indexing Design 3

- New Index: CREATE INDEX BagIndex ON BagItems(UserId, ProductId);
- EXPLAIN ANALYZE screenshot of this indexing design



```
| -> Sort: BI.ProductId, AverageRating DESC  (actual time=19.844..19.846 rows=12 loops=1)
    -> Table scan on <temporary>  (actual time=19.798..19.801 rows=12 loops=1)
        -> Aggregate using temporary table  (actual time=19.794..19.794 rows=12 loops=1)
            -> Nested loop inner join  (cost=1046.33 rows=465) (actual time=18.054..19.639 rows=68 loops=1)
                -> Nested loop inner join  (cost=558.35 rows=1394) (actual time=18.030..19.431 rows=68 loops=1)
                    -> Nested loop inner join  (cost=70.37 rows=136) (actual time=0.048..0.906 rows=184 loops=1)
                        -> Nested loop inner join  (cost=22.79 rows=136) (actual time=0.037..0.316 rows=184 loops=1)
                            -> Covering index lookup on BagItems using idx_bagitems_productid (ProductId='P501265')  (cost=2.25 rows=18) (actual time=0.020..0.032 rows=18 loops=1)
                            -> Filter: (BI.ProductId <> 'P501265')  (cost=0.38 rows=8) (actual time=0.007..0.014 rows=10 loops=18)
                                -> Covering index lookup on BI using BagIndex (UserId=BagItems.UserId)  (cost=0.38 rows=8) (actual time=0.006..0.011 rows=11 loops=18)
                        -> Single-row index lookup on P using PRIMARY (ProductId=BI.ProductId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=184)
                    -> Filter: (LatestReview.LatestReviewId is not null)  (cost=825.55..2.57 rows=10) (actual time=0.100..0.100 rows=0 loops=184)
                        -> Index lookup on LatestReview using <auto_key1> (ProductId=BI.ProductId)  (actual time=0.100..0.100 rows=0 loops=184)
                            -> Materialize  (cost=914.45..914.45 rows=1801) (actual time=17.928..17.928 rows=1800 loops=1)
                                -> Covering index skip scan for grouping on Reviews using idx_Reviews_ProductId_ReviewId  (cost=734.35 rows=1801) (actual time=0.023..15.031 rows=180
0 loops=1)
                -> Filter: (R.Rating > 1)  (cost=0.25 rows=0.3) (actual time=0.003..0.003 rows=1 loops=68)
                    -> Single-row index lookup on R using PRIMARY (ReviewId=LatestReview.LatestReviewId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=68)
 |
```

- Total Cost: 1046.33+558.35+70.37+22.79+2.25+0.38+0.38+0.25+825.55+914.45+734.35+0.25+0.25=4175.95
- Pros and cons, performance gains and degradations of this indexing design: This query actually caused a performance improvement. Most of the operations showed cost improvements. For example, the “Nested Loop Inner Join” operations all showed decreases in cost. This is because the index allows the database to quickly locate rows in “BagItems” based on the UserId and ProductId columns. However, since this filter condition involves a subquery that retrieves UserId values for a specific ProductId, any changes to the BagItems table that affect UserId or ProductId values may require updates to the index. This could cause slow-down for write operations.

Final Indexing Choice and Why

- Report on the final index design you selected and explain why you chose it, referencing the analysis you performed:

After thorough analysis, I will choose the BagIndex design because it depicted the most significant cost decrease out of the 3 possible options. Under the EXPLAIN ANALYZE command, the BagIndex design costed 59.71 less than the ByName and ByRating index designs. The BagIndex impacted a major JOIN clause, which caused a significant improvement. Since JOIN operations are very performance-intensive, improvement on these shows that the index causes substantial optimization. Although the index would need consistent updates, the other two index designs would also need consistent maintenance, so it is not the right measure

of the effectiveness of the index. Choosing the BagIndex design should be complemented by continuous monitoring and optimization.

Query 4: Aggregations for common products bagged between the application user and the user's bag they are viewing

```
SELECT BI1.UserId AS UserId1,BI2.UserId AS UserId2,
COUNT(BI1.ProductId) AS SharedProductsCount,GROUP_CONCAT(DISTINCT
P.ProductName ORDER BY P.ProductName SEPARATOR ', ') AS
SharedProductNames,AVG(P.Price) AS AveragePriceOfSharedProducts
FROM BagItems BI1
JOIN BagItems BI2 ON BI1.ProductId = BI2.ProductId AND BI1.UserId <
BI2.UserId
JOIN Products P ON BI1.ProductId = P.ProductId
GROUP BY BI1.UserId, BI2.UserId
HAVING COUNT(BI1.ProductId) > 0
LIMIT 15;
```

Output:

```
+---------+---------+--------------------+-------------------------------------------------------------------------------------------------------------------------------------------------+-----------------------------+
| UserId1 | UserId2 | SharedProductsCount | SharedProductNames                                                                                                                              | AveragePriceOfSharedProducts |
+---------+---------+--------------------+-------------------------------------------------------------------------------------------------------------------------------------------------+-----------------------------+
|       1 |       2 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |       3 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |       4 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |       5 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |       6 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |       7 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |       8 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |       9 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |      11 |                  5 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette                                                   |                        36.6000 |
|       1 |      13 |                  5 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette                                                   |                        36.6000 |
|       1 |      16 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |      17 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |      18 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |      19 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
|       1 |      20 |                  6 | Lash Extension Tubing Mascara, Mini Revitalizing Supreme+ Youth Power Creme
ascara, Soft Glam Eyeshadow Palette, The Camellia Oil 2-in-1 Makeup Remover & Cleanser |                        38.8333 |
+---------+---------+--------------------+-------------------------------------------------------------------------------------------------------------------------------------------------+-----------------------------+
```

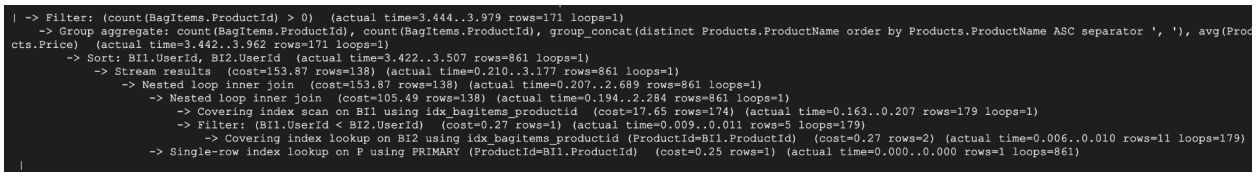
EXPLAIN ANALYZE Performance before adding indexes:

```
-> Limit: 15 row(s)  (actual time=2.527..2.578 rows=15 loops=1)
    -> Filter: (count(BagItems.ProductId) > 0)  (actual time=2.527..2.577 rows=15 loops=1)
        -> Group aggregate: count(BagItems.ProductId), count(BagItems.ProductId), group_concat(distinct Products.ProductName order by Products.ProductName ASC separator ',
roducts.Price)  (actual time=2.526..2.574 rows=15 loops=1)
            -> Sort: BI1.UserId, BI2.UserId  (actual time=2.503..2.512 rows=89 loops=1)
                -> Stream results  (cost=153.87 rows=138) (actual time=0.076..2.280 rows=861 loops=1)
                    -> Nested loop inner join  (cost=153.87 rows=138) (actual time=0.074..1.790 rows=861 loops=1)
                        -> Nested loop inner join  (cost=105.49 rows=138) (actual time=0.064..1.384 rows=861 loops=1)
                            -> Covering index scan on BI1 using ProductId  (cost=17.65 rows=174) (actual time=0.041..0.072 rows=179 loops=1)
                            -> Filter: (BI1.UserId < BI2.UserId)  (cost=0.27 rows=1) (actual time=0.004..0.007 rows=5 loops=179)
                                -> Covering index lookup on BI2 using ProductId (ProductId=BI1.ProductId)  (cost=0.27 rows=2) (actual time=0.002..0.006 rows=11 loops=179)
                        -> Single-row index lookup on P using PRIMARY (ProductId=BI1.ProductId)  (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=861)
 |
```

Total Cost: 153.87+153.87+105.49+17.65+0.27+0.27 + 0.25= 431.67

Indexing Design 1

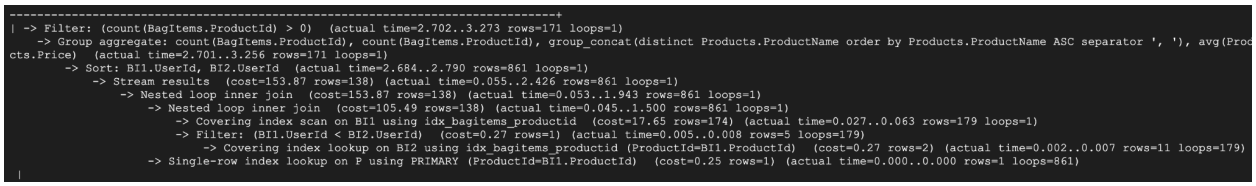
- New Index: CREATE INDEX idx_bagitems_productid ON BagItems(ProductId);
- EXPLAIN ANALYZE screenshot of this indexing design



- Total Cost: 153.87+153.87+105.49+17.65+0.27+0.27 + 0.25= 431.67
- Pros and cons, performance gains and degradations of this indexing design
    - There are no performance improvements or degradations for this index. This is most likely because ProductId is both a primary key and a foreign key to another table, so it is possible that, being a primary key, it already provided sufficient indexing for the query's needs without the index. Being that ProductId is also a foreign key in BagItems, this index is now tied to a foreign key constraint and will not be removed. The indexing design is used in the query's execution plan, and does not have any negative effects on the performance of the query.

Indexing Design 2

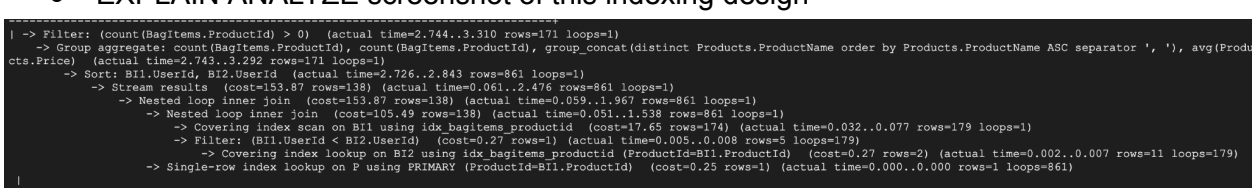
- New Index: CREATE INDEX idx_products_productname ON Products(ProductName);
- EXPLAIN ANALYZE screenshot of this indexing design



- Total Cost:153.87+153.87+105.49+17.65+0.27+0.27 + 0.25= 431.67
- Pros and cons, performance gains and degradations of this indexing design
    - There are no performance improvements or degradations for this index. It can be seen above that the query's execution plan doesn't use the index at all. This could be because the computational cost is heavily dominated by something other than retrieving or sorting by ProductName, so the impact of indexing on ProductName is negligible and the ProductName isn't much of a limiting factor.

Indexing Design 3

- New Index: CREATE INDEX idx_products_productname_price ON Products(ProductName, Price);
- EXPLAIN ANALYZE screenshot of this indexing design



- Total Cost:153.87+153.87+105.49+17.65+0.27+0.27 + 0.25= 431.67

- Pros and cons, performance gains and degradations of this indexing design
    - There are no performance gains and degradations for this indexing design. The query's execution plan doesn't use this indexing design either. Indexing ProductName and Price together did not outperform the current execution plan significantly enough, which could mean the cost of using the index may have outweighed the performance benefits.

Final Indexing Choice and Why

- Report on the final index design you selected and explain why you chose it, referencing the analysis you performed
    - In the final index design, we will be using Indexing Design 1 and not using Indexing Design 2 and 3. Although Indexing Design 1 doesn't change the computational cost, the index cannot be removed without changing the entire database due to a Foreign Key constraint. It is utilized in the query's execution plan, therefore we decided to continue using it. It also does not affect the total cost negatively. We will not be using index designs 2 and 3 because the impact of them is clearly negligible, and using the index designs will not outperform a full table scan when executing the query. This is likely due to another part of the query taking up the majority of the computational cost. Using the GROUP_CONCAT function in this query was likely very costly and time consuming compared to the indexes attempted, however there is little indexing that can be done to lower the cost of the GROUP_CONCAT usage.

Appendix

QUERIES IN PROGRESS/OLD QUERIES

#	Advanced query	Screenshot of max(15, query result) rows of advanced query results
1	**Average Rating** // SELECT "P132239" FROM SELECT ProductId, AVG(Rating), COUNT(Rating) AS NumRatings	On the Products page, to display these stats ⭐⭐⭐⭐⭐ 4.97 avg rating (from 19 reviews)

	FROM Products Pro JOIN Reviews Rev ON Pro.ProductId = Rev.ProductId GROUP BY ProductId;	
4a	**Seeing how many items from each brand a user has in their bag:** SELECT U.UserId, U.FirstName, U.LastName, COUNT(BI.ProductId) AS ProductCount, P.BrandId FROM Users U JOIN BagItems BI ON U.UserId = BI.UserId JOIN Products P ON BI.ProductId = P.ProductId GROUP BY U.UserId, P.BrandId HAVING COUNT(BI.ProductId) > 3 LIMIT 15;	
4b *	**Comparing one user's bag to another user's to see how many items they have similar:** SELECT BI1.UserId AS User1, BI2.UserId AS User2, COUNT(BI1.ProductId) AS SharedProductCount FROM BagItems BI1 JOIN BagItems BI2 ON BI1.ProductId = BI2.ProductId AND BI1.UserId <> BI2.UserId WHERE BI1.UserId = 18 AND BI2.UserId = 6 GROUP BY BI1.UserId, BI2.UserId HAVING COUNT(BI1.ProductId) > 2 LIMIT 15;	