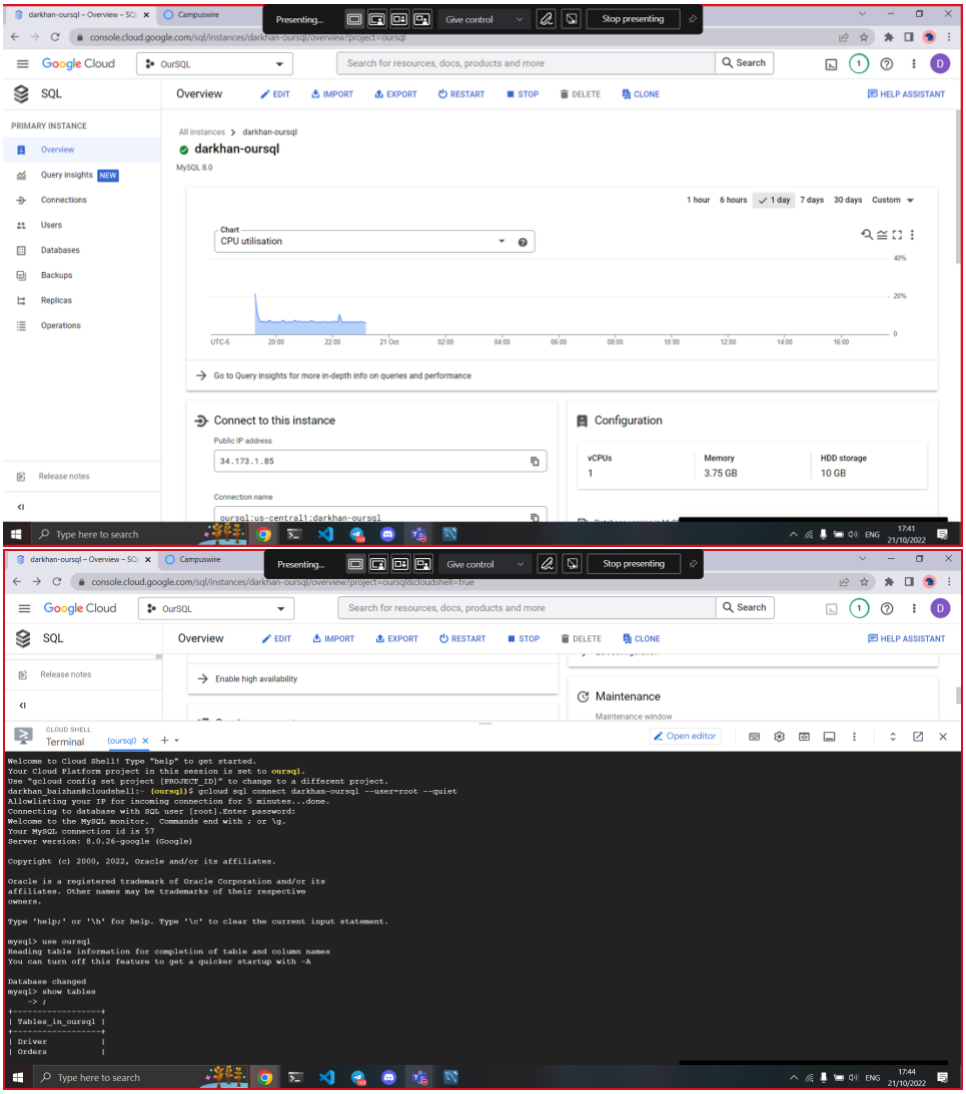
# CS 411 – Team 029 OurSQL
# Stage 3

## 1. DB implementation (screenshot of connection)

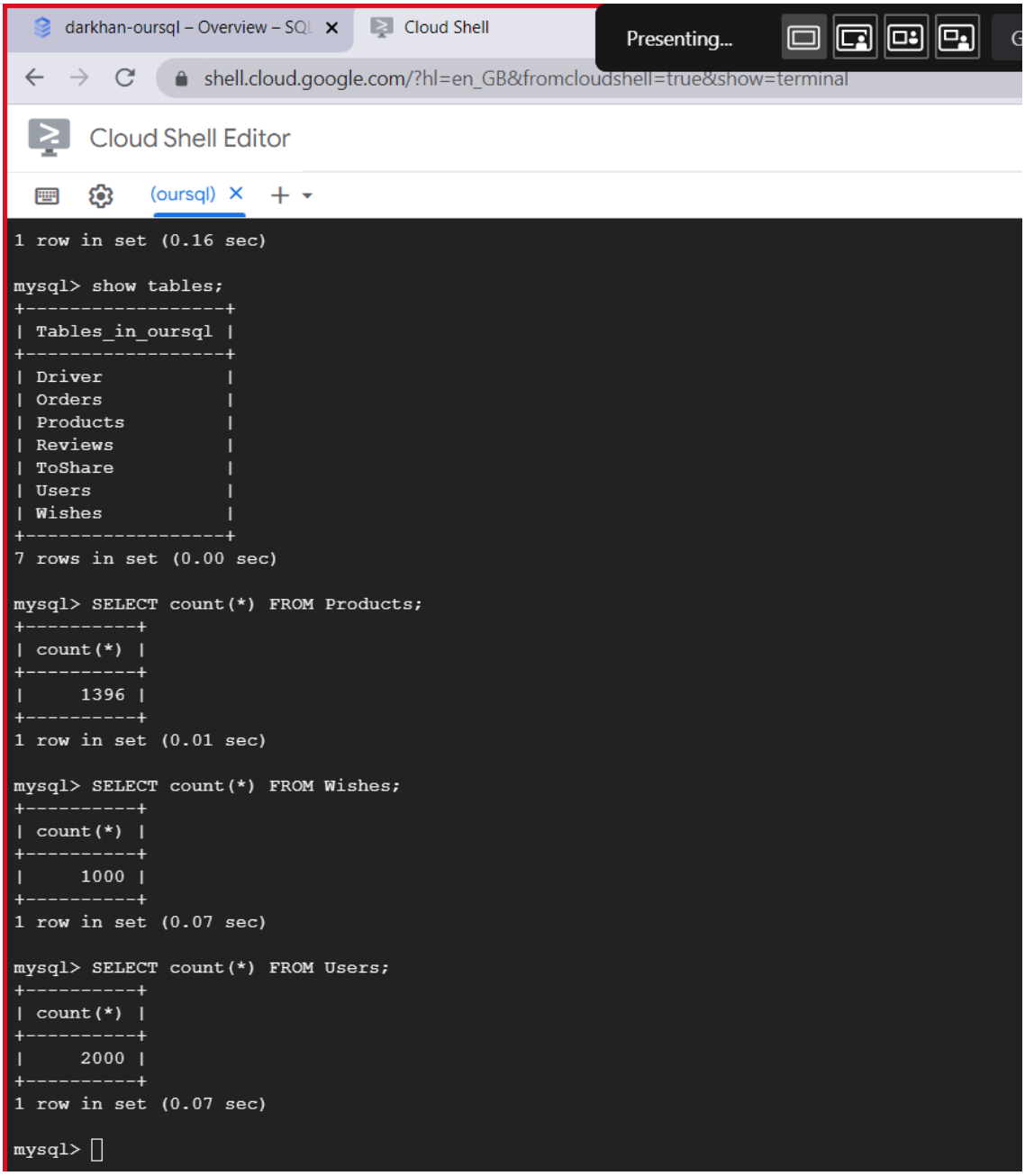


## 2. DDL commands:

```sql
DROP DATABASE IF EXISTS oursql;
CREATE DATABASE oursql;
CREATE TABLE oursql.Products (
  productId int primary key,
  name varchar(255),
  category varchar(225),
  price real,
  quantity int,
  imageURL varchar(200),
  store varchar(225)
  );

CREATE TABLE oursql.Users (
  userId int primary key,
  name varchar(225),
  surname varchar(225),
  email varchar(50),
  password varchar(225) ,
  isDriver int
  );

CREATE TABLE oursql.Driver (
  driverId int primary key,
  availableTime varchar(225),
  FOREIGN KEY(driverId) references Users(userId)
  );
```

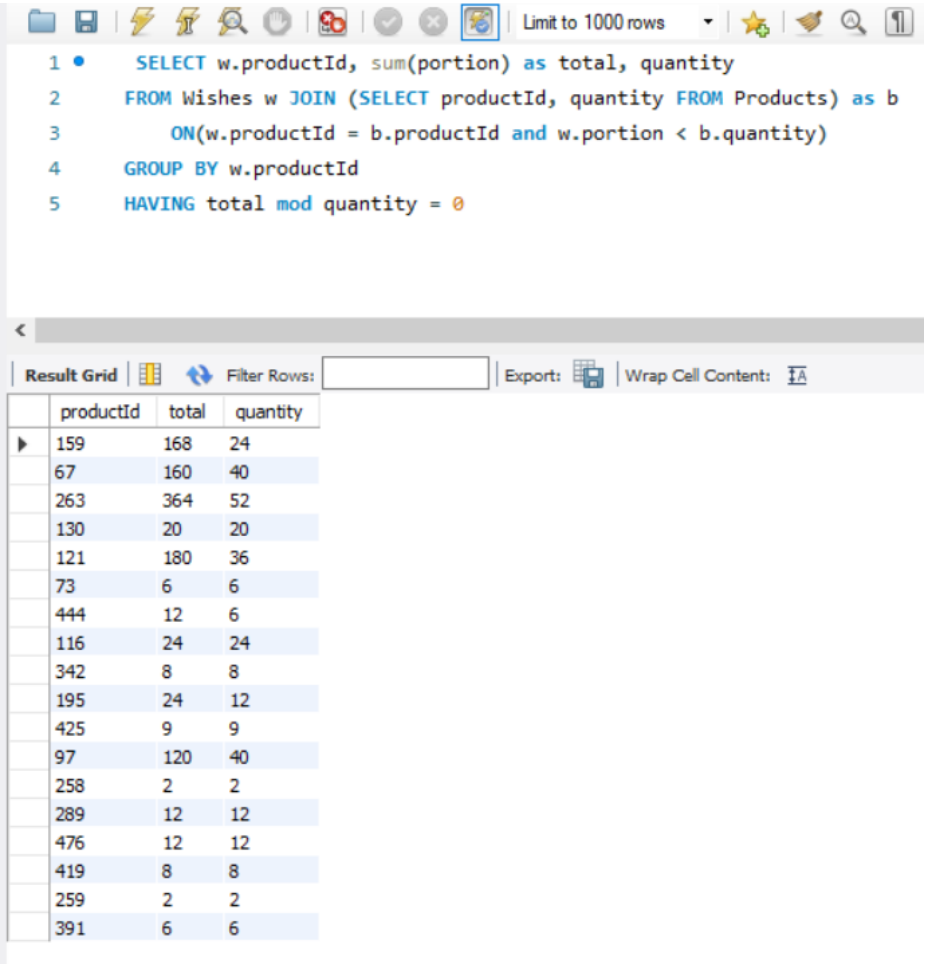
```sql
CREATE TABLE oursql.Orders (
  orderId int primary key,
  driverId int,
  orderDate date,
  store varchar(225),
  fullfiled int,
  FOREIGN KEY(driverId) references Users(userId) );

CREATE TABLE oursql.Wishes (
  wishId int primary key,
  productId int,
  userId int,
  portion real,
  status int,
  FOREIGN KEY(productId) references Products(productId),
  FOREIGN KEY(userId) references Users(userId));

CREATE TABLE oursql.ToShare(
  wishId int,
  orderId int,
  pickedUp int,
  FOREIGN KEY(wishId) references Wishes(wishId),
  FOREIGN KEY(orderId) references Orders(orderId));
CREATE TABLE oursql.Reviews(
  reviewId int primary key,
  userId int,
  productId int,
  rating int,
  comments varchar(1000),
  FOREIGN KEY(userId) references Users(userId),
  FOREIGN KEY(productId) references Products(productId));
```

## 3. Inserting data (screenshots with count)



## 4. Two Advanced SQL queries + screenshots of results

**Query 1** – this query shows the list of products that can be already formed into order. Recalling that the products from wishlists will be put into order as soon they sum to the full package quantity.



```
1  SELECT w.productId, sum(portion) as total, quantity
2  FROM Wishes w JOIN (SELECT productId, quantity FROM Products) as b
3      ON(w.productId = b.productId and w.portion < b.quantity)
4  GROUP BY w.productId
5  HAVING total mod quantity = 0
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| productId | total | quantity |
|---|---|---|
| 159 | 168 | 24 |
| 67 | 160 | 40 |
| 263 | 364 | 52 |
| 130 | 20 | 20 |
| 121 | 180 | 36 |
| 73 | 6 | 6 |
| 444 | 12 | 6 |
| 116 | 24 | 24 |
| 342 | 8 | 8 |
| 195 | 24 | 12 |
| 425 | 9 | 9 |
| 97 | 120 | 40 |
| 258 | 2 | 2 |
| 289 | 12 | 12 |
| 476 | 12 | 12 |
| 419 | 8 | 8 |
| 259 | 2 | 2 |
| 391 | 6 | 6 |

**Query 2** – this query gives us the list of categories and corresponding number of purchases(*sum(freq)*) and the revenue from that category. This query will be used to identify which most demanding and most profitable categories.

```
1 •  SELECT category, sum(freq), ROUND(sum(price*freq),2) as revenue
2    FROM Products NATURAL JOIN (SELECT productId, count(wishId) as freq
3                                FROM Wishes
4                                GROUP BY productId) as p
5    GROUP BY category
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| category | sum(freq) | revenue |
|---|---|---|
| Candy | 30 | 559.55 |
| Snacks | 285 | 3833.39 |
| Beverages | 416 | 6423.33 |
| Pantry | 269 | 3065.72 |

## 5. 3 indexing methods (+1 default indexing), screenshots of results + analysis

**For Query 1:**

### A) Default Indexing

Wishes table:

| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type |
|---|---|---|---|---|---|---|---|---|---|---|
| Wishes | 0 | PRIMARY | 1 | wishId | A | 0 | NULL | NULL | | BTREE |
| Wishes | 1 | productId | 1 | productId | A | 0 | NULL | NULL | YES | BTREE |
| Wishes | 1 | userId | 1 | userId | A | 0 | NULL | NULL | YES | BTREE |

Products Table

| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type |
|---|---|---|---|---|---|---|---|---|---|---|
| Products | 0 | PRIMARY | 1 | productId | A | 1376 | NULL | NULL | | BTREE |

Here we can see that due to the usage of primary and foreign keys, our tables already have indexing applied to those fields.

```
| -> Filter: ((total % b.quantity) = 0)  (actual time=1.998..2.039 rows=18 loops=1)
    -> Table scan on <temporary>  (actual time=0.000..0.009 rows=194 loops=1)
        -> Aggregate using temporary table  (actual time=1.995..2.014 rows=194 loops=1)
            -> Nested loop inner join  (cost=451.25 rows=333) (actual time=0.056..1.710 rows=887 loops=1)
                -> Filter: (w.productId is not null)  (cost=101.25 rows=1000) (actual time=0.041..0.345 rows=1000 loops=1)
                    -> Table scan on w  (cost=101.25 rows=1000) (actual time=0.040..0.276 rows=1000 loops=1)
                -> Filter: (w.portion < b.quantity)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=1000)
                    -> Single-row index lookup on b using PRIMARY (productId=w.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
|
```

### B) With indexing Products.quantity

After CREATE INDEX quantity_idx ON Products(quantity)

| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type |
|---|---|---|---|---|---|---|---|---|---|---|
| Products | 0 | PRIMARY | 1 | productId | A | 1376 | NULL | NULL | | BTREE |
| Products | 1 | quantity_idx | 1 | quantity | A | 57 | NULL | NULL | YES | BTREE |

```
| -> Filter: ((total % b.quantity) = 0)  (actual time=2.010..2.051 rows=18 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.009 rows=194 loops=1)
        -> Aggregate using temporary table  (actual time=2.006..2.026 rows=194 loops=1)
            -> Nested loop inner join  (cost=451.25 rows=333) (actual time=0.040..1.720 rows=887 loops=1)
                -> Filter: (w.productId is not null)  (cost=101.25 rows=1000) (actual time=0.028..0.347 rows=1000 loops=1)
                    -> Table scan on w  (cost=101.25 rows=1000) (actual time=0.027..0.272 rows=1000 loops=1)
                -> Filter: (w.portion < b.quantity)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=1000)
                    -> Single-row index lookup on b using PRIMARY (productId=w.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
|
```

### C) With indexing Wishes.portion

Then DROP INDEX quantity_idx on Products;

After CREATE INDEX portion_idx ON Wishes(portion);

| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type |
|---|---|---|---|---|---|---|---|---|---|---|
| Wishes | 0 | PRIMARY | 1 | wishId | A | 0 | NULL | NULL | | BTREE |
| Wishes | 1 | productId | 1 | productId | A | 0 | NULL | NULL | YES | BTREE |
| Wishes | 1 | userId | 1 | userId | A | 0 | NULL | NULL | YES | BTREE |
| Wishes | 1 | portion_idx | 1 | portion | A | 34 | NULL | NULL | YES | BTREE |

```
| -> Filter: ((total % b.quantity) = 0)  (actual time=2.021..2.062 rows=18 loops=1)
    -> Table scan on <temporary>  (actual time=0.000..0.009 rows=194 loops=1)
        -> Aggregate using temporary table  (actual time=2.017..2.037 rows=194 loops=1)
            -> Nested loop inner join  (cost=451.25 rows=333) (actual time=0.054..1.736 rows=887 loops=1)
                -> Filter: (w.productId is not null)  (cost=101.25 rows=1000) (actual time=0.040..0.362 rows=1000 loops=1)
                    -> Table scan on w  (cost=101.25 rows=1000) (actual time=0.038..0.290 rows=1000 loops=1)
                -> Filter: (w.portion < b.quantity)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=1000)
                    -> Single-row index lookup on b using PRIMARY (productId=w.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
|
```

### D) With Indexing both Wishes.portion and Products.quantity

Now try together: CREATE INDEX quantity_idx ON Products(quantity) AND CREATE INDEX portion_idx ON Wishes(portion)

```
| -> Filter: ((total % b.quantity) = 0)  (actual time=2.032..2.081 rows=18 loops=1)
    -> Table scan on <temporary>  (actual time=0.000..0.009 rows=194 loops=1)
        -> Aggregate using temporary table  (actual time=2.028..2.048 rows=194 loops=1)
            -> Nested loop inner join  (cost=451.25 rows=333) (actual time=0.074..1.749 rows=887 loops=1)
                -> Filter: (w.productId is not null)  (cost=101.25 rows=1000) (actual time=0.052..0.360 rows=1000 loops=1)
                    -> Table scan on w  (cost=101.25 rows=1000) (actual time=0.051..0.287 rows=1000 loops=1)
                -> Filter: (w.portion < b.quantity)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=1000)
                    -> Single-row index lookup on b using PRIMARY (productId=w.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
|
```

### Analysis of using Indexing with Query 1

Because our query mostly utilizes the primary and foreign keys (which are already indexed), there are few fields we can try indexing on. We tried applying indexing on other two attributes (Products(quantity), Wishes(portion)) which appear in our query. From the performance evaluations we can observe no improvements. This is an expected result. It is because we do not just search for some specific quantity and portion, but do comparison and sum operations, where indexing does not make any improvements. Therefore, for this query no additional indexing is needed.

## For Query 2:

### A) Default indexing

Default indexing of the tables Products and Wishes is the same as shown in A) for Query 1.

```
| -> Table scan on <temporary>  (actual time=0.000..0.001 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=1.039..1.040 rows=4 loops=1)
        -> Nested loop inner join  (cost=465.00 rows=1000) (actual time=0.365..0.807 rows=262 loops=1)
            -> Filter: (p.productId is not null)  (cost=301.06..115.00 rows=1000) (actual time=0.351..0.398 rows=262 loops=1)
                -> Table scan on p  (cost=0.01..15.00 rows=1000) (actual time=0.001..0.014 rows=262 loops=1)
                    -> Materialize  (cost=301.26..316.25 rows=1000) (actual time=0.350..0.379 rows=262 loops=1)
                        -> Group aggregate: count(Wishes.wishId)  (cost=201.25 rows=1000) (actual time=0.031..0.307 rows=262 loops=1)
                            -> Index scan on Wishes using productId  (cost=101.25 rows=1000) (actual time=0.027..0.220 rows=1000 loops=1)
            -> Single-row index lookup on Products using PRIMARY (productId=p.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=262)
|
```

### B) With Indexing Products.Category

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type |
|---|---|---|---|---|---|---|---|---|---|---|
| Products | 0 | PRIMARY | 1 | productId | A | 1376 | NULL | NULL | | BTREE |
| Products | 1 | category_idx | 1 | category | A | 10 | NULL | NULL | YES | BTREE |

```
| -> Table scan on <temporary>  (actual time=0.000..0.001 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=1.023..1.024 rows=4 loops=1)
        -> Nested loop inner join  (cost=465.00 rows=1000) (actual time=0.340..0.794 rows=262 loops=1)
            -> Filter: (p.productId is not null)  (cost=301.06..115.00 rows=1000) (actual time=0.330..0.376 rows=262 loops=1)
                -> Table scan on p  (cost=0.01..15.00 rows=1000) (actual time=0.001..0.014 rows=262 loops=1)
                    -> Materialize  (cost=301.26..316.25 rows=1000) (actual time=0.329..0.358 rows=262 loops=1)
                        -> Group aggregate: count(Wishes.wishId)  (cost=201.25 rows=1000) (actual time=0.029..0.287 rows=262 loops=1)
                            -> Index scan on Wishes using productId  (cost=101.25 rows=1000) (actual time=0.026..0.200 rows=1000 loops=1)
            -> Single-row index lookup on Products using PRIMARY (productId=p.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=262)
|
```

### C) With Indexing Products.Price

| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type |
|---|---|---|---|---|---|---|---|---|---|---|
| Products | 0 | PRIMARY | 1 | productId | A | 1376 | NULL | NULL | | BTREE |
| Products | 1 | price_idx | 1 | price | A | 572 | NULL | NULL | YES | BTREE |

```
| -> Table scan on <temporary>  (actual time=0.000..0.001 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=1.070..1.070 rows=4 loops=1)
        -> Nested loop inner join  (cost=465.00 rows=1000) (actual time=0.379..0.832 rows=262 loops=1)
            -> Filter: (p.productId is not null)  (cost=301.06..115.00 rows=1000) (actual time=0.365..0.410 rows=262 loops=1)
                -> Table scan on p  (cost=0.01..15.00 rows=1000) (actual time=0.001..0.014 rows=262 loops=1)
                    -> Materialize  (cost=301.26..316.25 rows=1000) (actual time=0.364..0.392 rows=262 loops=1)
                        -> Group aggregate: count(Wishes.wishId)  (cost=201.25 rows=1000) (actual time=0.038..0.320 rows=262 loops=1)
                            -> Index scan on Wishes using productId  (cost=101.25 rows=1000) (actual time=0.035..0.233 rows=1000 loops=1)
            -> Single-row index lookup on Products using PRIMARY (productId=p.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=262)
|
```

### D) With Indexing Products.Price and Products.Category

| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type |
|---|---|---|---|---|---|---|---|---|---|---|
| Products | 0 | PRIMARY | 1 | productId | A | 1376 | NULL | NULL | | BTREE |
| Products | 1 | price_idx | 1 | price | A | 572 | NULL | NULL | YES | BTREE |
| Products | 1 | category_idx | 1 | category | A | 10 | NULL | NULL | YES | BTREE |

```
| -> Table scan on <temporary>  (actual time=0.000..0.001 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=1.143..1.143 rows=4 loops=1)
        -> Nested loop inner join  (cost=465.00 rows=1000) (actual time=0.355..0.871 rows=262 loops=1)
            -> Filter: (p.productId is not null)  (cost=301.06..115.00 rows=1000) (actual time=0.344..0.404 rows=262 loops=1)
                -> Table scan on p  (cost=0.01..15.00 rows=1000) (actual time=0.001..0.016 rows=262 loops=1)
                    -> Materialize  (cost=301.26..316.25 rows=1000) (actual time=0.342..0.373 rows=262 loops=1)
                        -> Group aggregate: count(Wishes.wishId)  (cost=201.25 rows=1000) (actual time=0.030..0.301 rows=262 loops=1)
                            -> Index scan on Wishes using productId  (cost=101.25 rows=1000) (actual time=0.027..0.202 rows=1000 loops=1)
            -> Single-row index lookup on Products using PRIMARY (productId=p.productId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=262)
|
```

### Analysis of using Indexing with Query 2

The only fields used in this query which are not primary or foreign keys are category and store name. We tried indexing each of them separately and then together. Neither of this indexing techniques gave us improvement in performance. Again, since these fields are not used for filtering directly, we still need to go through each of the rows in the data. Therefore, indexing is not helpful. So, it's better not to use any indexing for this query.

Overall, we tried a lot of different queries and indexing methods. It looks like as long as we have optimal normalized database design, additional indexing techniques are not required. They would be helpful only in the case when we are searching for specific field which are not primary keys (eg. Searching for some product by its name).