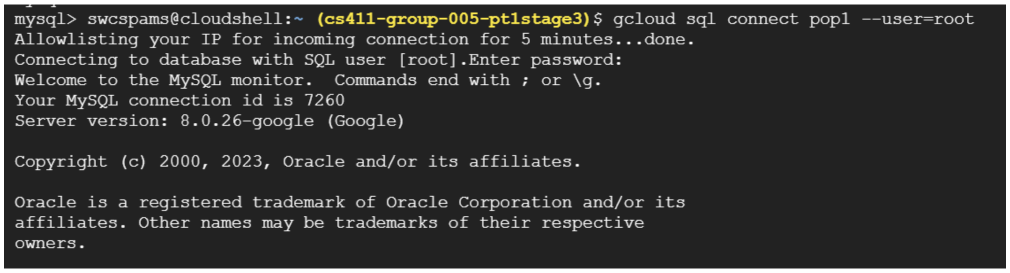
1 GCP Connection



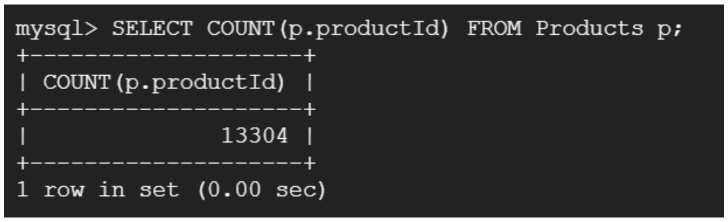
```
mysql> swcspams@cloudshell:~ (cs411-group-005-pt1stage3)$ gcloud sql connect pop1 --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7260
Server version: 8.0.26-google (Google)

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
```

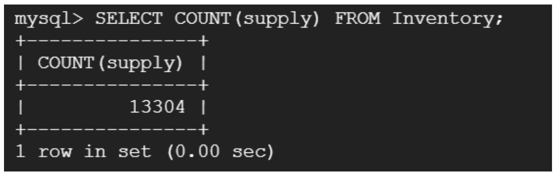
2 DDL Commands

```
CREATE TABLE Products(
    productId INT,
    name VARCHAR(255),
    rating INT,
    requiredAge INT,
    pcRequirements VARCHAR(255),
    description VARCHAR(1000),
    imageLink VARCHAR(500),
    releaseDate DATE,
    PRIMARY KEY(productId, name));
```



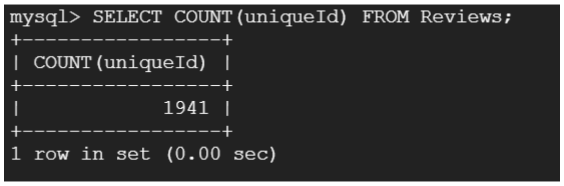
```
mysql> SELECT COUNT(p.productId) FROM Products p;
+--------------------+
| COUNT(p.productId) |
+--------------------+
|              13304 |
+--------------------+
1 row in set (0.00 sec)
```

```
CREATE TABLE Inventory(
    productId INT,
    supply INT,
    price REAL,
    discount REAL,
    PRIMARY KEY(productId),
    FOREIGN KEY productId REFERENCES Products(productId);
```



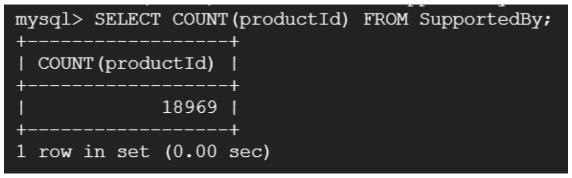
```
mysql> SELECT COUNT(supply) FROM Inventory;
+---------------+
| COUNT(supply) |
+---------------+
|         13304 |
+---------------+
1 row in set (0.00 sec)
```

```
CREATE TABLE Reviews(
    uniqueId INT,
    purchaseId INT,
    rating INT,
    reviewText VARCHAR(2000),
    upvotes INT,
    verifiedPurchase BOOLEAN,
    PRIMARY KEY(uniqueId),
    FOREIGN KEY purchaseId REFERENCES Purchases(purchaseId));
```



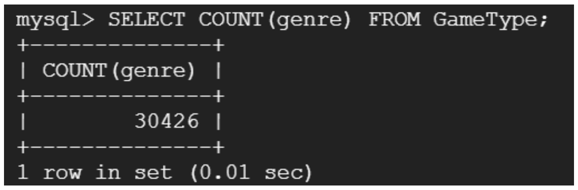
```
mysql> SELECT COUNT(uniqueId) FROM Reviews;
+-----------------+
| COUNT(uniqueId) |
+-----------------+
|            1941 |
+-----------------+
1 row in set (0.00 sec)
```

```
CREATE TABLE SupportedBy(
    platform VARCHAR(255),
    productID INT,
    PRIMARY KEY(platform, productId),
    FOREIGN KEY platform REFERENCES Platform(platform),
    FOREIGN KEY productId REFERENCES Products(productId));
```



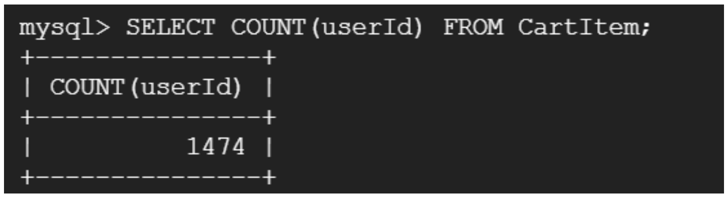
```
mysql> SELECT COUNT(productId) FROM SupportedBy;
+------------------+
| COUNT(productId) |
+------------------+
|            18969 |
+------------------+
1 row in set (0.00 sec)
```

```
CREATE TABLE GameType(
    genre VARCHAR(255),
    productID INT,
    PRIMARY KEY(genre, productId),
    FOREIGN KEY genre REFERENCES Genre(genre),
    FOREIGN KEY productId REFERENCES Products(productId));
```



```
mysql> SELECT COUNT(genre) FROM GameType;
+--------------+
| COUNT(genre) |
+--------------+
|        30426 |
+--------------+
1 row in set (0.01 sec)
```

```
CREATE TABLE CartItem(
    itemNumber INT,
    userId INT
    productId INT,
    count INT,
    PRIMARY KEY(itemNumber, userId),
    FOREIGN KEY productId REFERENCES Products(productId),
    FOREIGN KEY userId REFERENCES Customers(customerId));
```



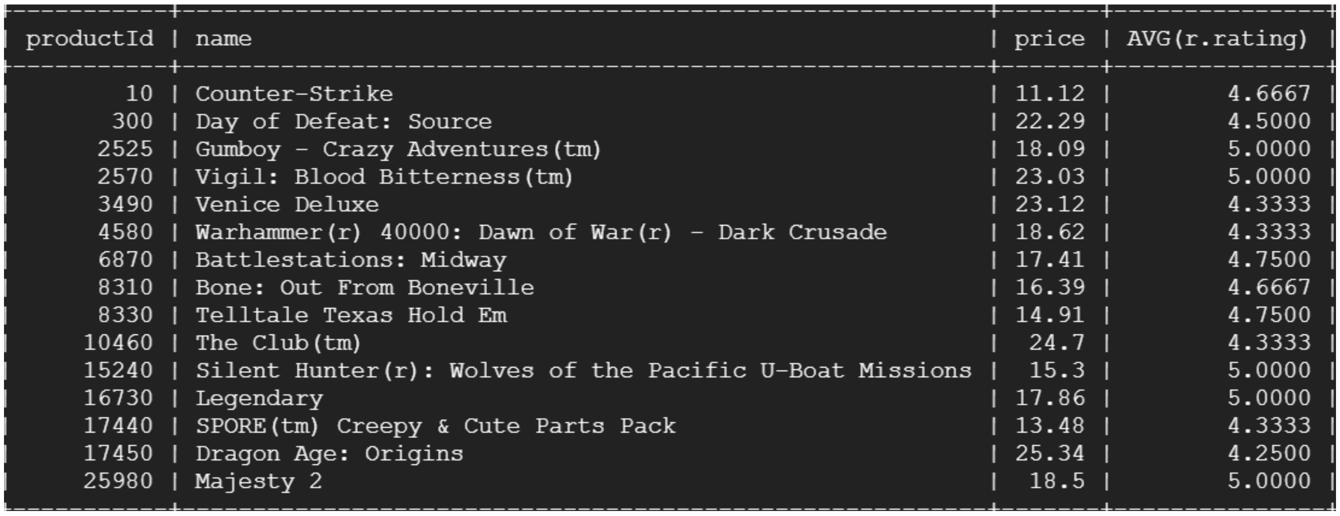
```
mysql> SELECT COUNT(userId) FROM CartItem;
+---------------+
| COUNT(userId) |
+---------------+
|          1474 |
+---------------+
```

3 Queries and Indexing

Query 1: Selecting product ID, name, price, and average rating for games that are in stock and have a rating above 4 stars (JOINs and aggregation)

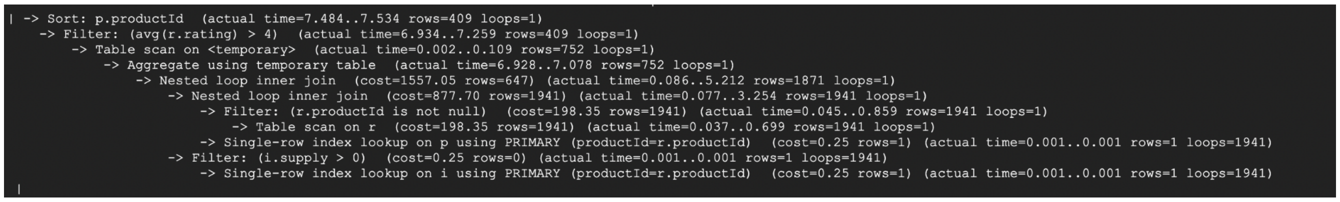
```
SELECT p.productId, p.name, i.price, AVG(r.rating) AS Average_Rating
FROM Products p NATURAL JOIN Inventory i JOIN Reviews r
     ON (p.productId = r.productId)
WHERE i.supply > 0
GROUP BY p.productId
HAVING AVG(r.rating) > 4
ORDER BY p.productId;
```

Query 1 Results:



```
+-----------+---------------------------------------------------------+-------+---------------+
| productId | name                                                    | price | AVG(r.rating) |
+-----------+---------------------------------------------------------+-------+---------------+
|        10 | Counter-Strike                                          | 11.12 |        4.6667 |
|       300 | Day of Defeat: Source                                   | 22.29 |        4.5000 |
|      2525 | Gumboy - Crazy Adventures(tm)                           | 18.09 |        5.0000 |
|      2570 | Vigil: Blood Bitterness(tm)                             | 23.03 |        5.0000 |
|      3490 | Venice Deluxe                                           | 23.12 |        4.3333 |
|      4580 | Warhammer(r) 40000: Dawn of War(r) - Dark Crusade       | 18.62 |        4.3333 |
|      6870 | Battlestations: Midway                                  | 17.41 |        4.7500 |
|      8310 | Bone: Out From Boneville                                | 16.39 |        4.6667 |
|      8330 | Telltale Texas Hold Em                                  | 14.91 |        4.7500 |
|     10460 | The Club(tm)                                            |  24.7 |        4.3333 |
|     15240 | Silent Hunter(r): Wolves of the Pacific U-Boat Missions |  15.3 |        5.0000 |
|     16730 | Legendary                                               | 17.86 |        5.0000 |
|     17440 | SPORE(tm) Creepy & Cute Parts Pack                      | 13.48 |        4.3333 |
|     17450 | Dragon Age: Origins                                     | 25.34 |        4.2500 |
|     25980 | Majesty 2                                               |  18.5 |        5.0000 |
+-----------+---------------------------------------------------------+-------+---------------+
```

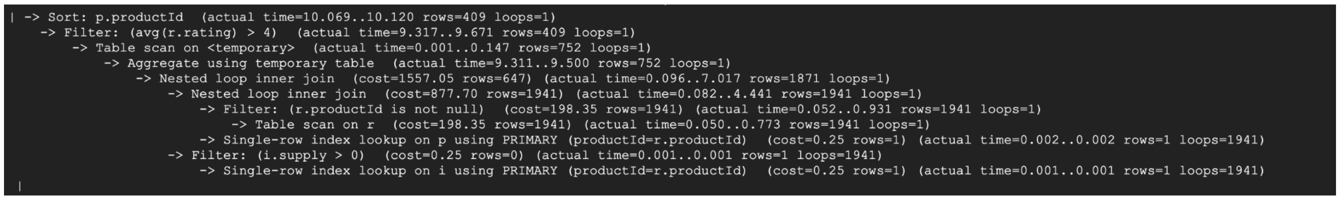
Query 1 Indexing:



```
| -> Sort: p.productId  (actual time=7.484..7.534 rows=409 loops=1)
    -> Filter: (avg(r.rating) > 4)  (actual time=6.934..7.259 rows=409 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.109 rows=752 loops=1)
            -> Aggregate using temporary table  (actual time=6.928..7.078 rows=752 loops=1)
                -> Nested loop inner join  (cost=1557.05 rows=647) (actual time=0.086..5.212 rows=1871 loops=1)
                    -> Nested loop inner join  (cost=877.70 rows=1941) (actual time=0.077..3.254 rows=1941 loops=1)
                        -> Filter: (r.productId is not null)  (cost=198.35 rows=1941) (actual time=0.045..0.859 rows=1941 loops=1)
                            -> Table scan on r  (cost=198.35 rows=1941) (actual time=0.037..0.699 rows=1941 loops=1)
                        -> Single-row index lookup on p using PRIMARY (productId=r.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1941)
                    -> Filter: (i.supply > 0)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=1941)
                        -> Single-row index lookup on i using PRIMARY (productId=r.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1941)
 |
```

Index Design 1 (index on ratings in Reviews table):

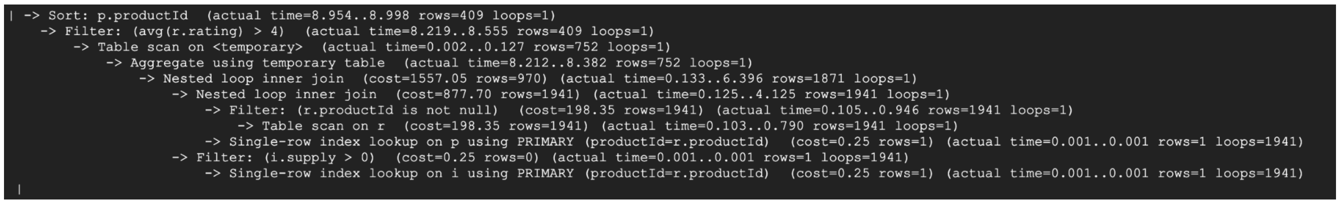
```
     CREATE INDEX highest_rating ON Reviews(rating ASC);
     DROP INDEX highest_rating ON Reviews;
```



```
| -> Sort: p.productId  (actual time=10.069..10.120 rows=409 loops=1)
    -> Filter: (avg(r.rating) > 4)  (actual time=9.317..9.671 rows=409 loops=1)
        -> Table scan on <temporary>  (actual time=0.001..0.147 rows=752 loops=1)
            -> Aggregate using temporary table  (actual time=9.311..9.500 rows=752 loops=1)
                -> Nested loop inner join  (cost=1557.05 rows=647) (actual time=0.096..7.017 rows=1871 loops=1)
                    -> Nested loop inner join  (cost=877.70 rows=1941) (actual time=0.082..4.441 rows=1941 loops=1)
                        -> Filter: (r.productId is not null)  (cost=198.35 rows=1941) (actual time=0.052..0.931 rows=1941 loops=1)
                            -> Table scan on r  (cost=198.35 rows=1941) (actual time=0.050..0.773 rows=1941 loops=1)
                        -> Single-row index lookup on p using PRIMARY (productId=r.productId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1941)
                    -> Filter: (i.supply > 0)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=1941)
                        -> Single-row index lookup on i using PRIMARY (productId=r.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1941)
 |
```

Index Design 2 (index on supply in Inventory table):

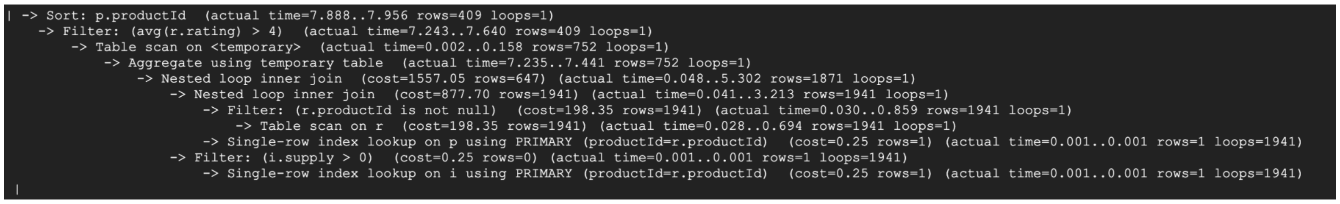
```
	CREATE INDEX in_supply ON Inventory(supply DESC);
	DROP INDEX in_supply ON Inventory;
```



```
| -> Sort: p.productId  (actual time=8.954..8.998 rows=409 loops=1)
    -> Filter: (avg(r.rating) > 4)  (actual time=8.219..8.555 rows=409 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.127 rows=752 loops=1)
            -> Aggregate using temporary table  (actual time=8.212..8.382 rows=752 loops=1)
                -> Nested loop inner join  (cost=1557.05 rows=970) (actual time=0.133..6.396 rows=1871 loops=1)
                    -> Nested loop inner join  (cost=877.70 rows=1941) (actual time=0.125..4.125 rows=1941 loops=1)
                        -> Filter: (r.productId is not null)  (cost=198.35 rows=1941) (actual time=0.105..0.946 rows=1941 loops=1)
                            -> Table scan on r  (cost=198.35 rows=1941) (actual time=0.103..0.790 rows=1941 loops=1)
                        -> Single-row index lookup on p using PRIMARY (productId=r.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1941)
                    -> Filter: (i.supply > 0)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=1941)
                        -> Single-row index lookup on i using PRIMARY (productId=r.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1941)
 |
```

Index Design 3 (index on productId in Products table):

```
	CREATE INDEX product_id_group ON Products(productId);
	DROP INDEX product_id_group ON Products;
```



```
| -> Sort: p.productId  (actual time=7.888..7.956 rows=409 loops=1)
    -> Filter: (avg(r.rating) > 4)  (actual time=7.243..7.640 rows=409 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.158 rows=752 loops=1)
            -> Aggregate using temporary table  (actual time=7.235..7.441 rows=752 loops=1)
                -> Nested loop inner join  (cost=1557.05 rows=647) (actual time=0.048..5.302 rows=1871 loops=1)
                    -> Nested loop inner join  (cost=877.70 rows=1941) (actual time=0.041..3.213 rows=1941 loops=1)
                        -> Filter: (r.productId is not null)  (cost=198.35 rows=1941) (actual time=0.030..0.859 rows=1941 loops=1)
                            -> Table scan on r  (cost=198.35 rows=1941) (actual time=0.028..0.694 rows=1941 loops=1)
                        -> Single-row index lookup on p using PRIMARY (productId=r.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1941)
                    -> Filter: (i.supply > 0)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=1941)
                        -> Single-row index lookup on i using PRIMARY (productId=r.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1941)
 |
```

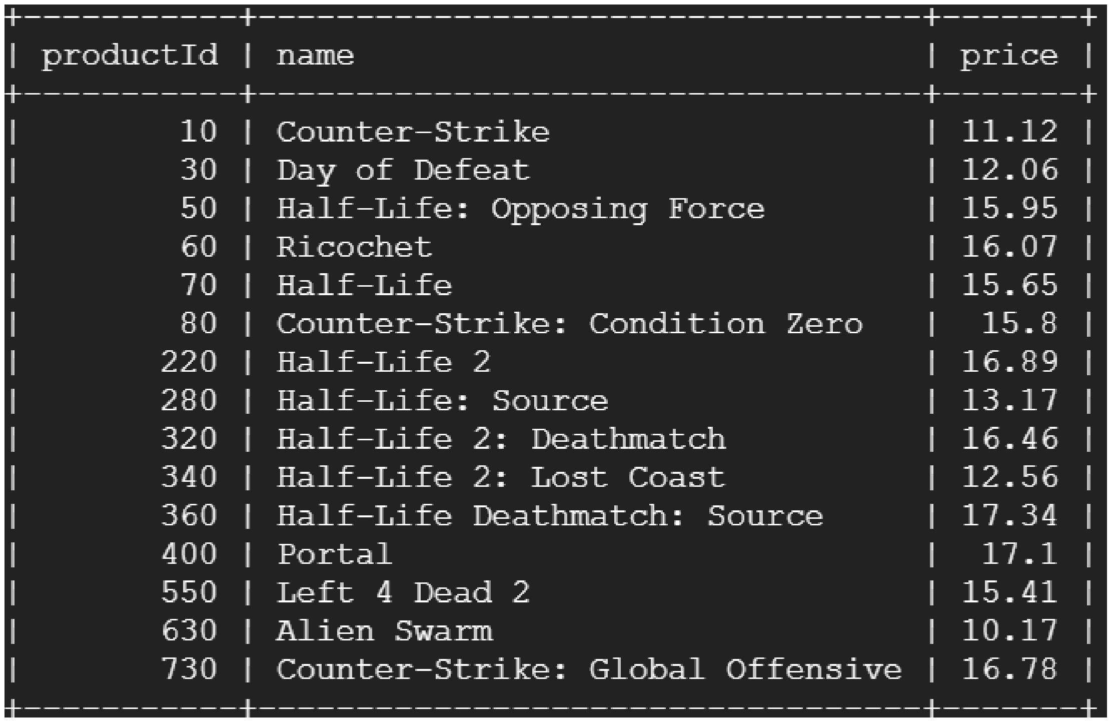
Justification:

We choose Index Design 1 to be indexing on ratings in the Reviews table because there are many reviews with the same ratings value, so entries would be able to share blocks. Additionally, ratings is a very important attribute that we will be using in a lot of our queries, so we wanted to see if it is a viable indexing scheme. We chose Index Design 2 to be indexing on supply in the Inventory table for similar reasons. The reason we decided on this one was after running the first indexing design, we realized that having both a “GROUP BY” and index would be inefficient. We chose to sort it on the basis that the search algorithm would first look at those in supply only, so the field of checking that particular condition would be only until a value that was not in supply was found. Therefore the query would only have to look for and group by ratings. Our query did improve runtime, but we found a better indexing design. We chose Index Design 3 to be indexing on productId in the Products table because productId is the key that we are grouping by in this query, so we thought that the query may run faster with indexing by productId.

Query 2: Selecting games that have prices below the average price (JOINs, subqueries, and aggregation)

```
SELECT p.productId, p.name, i.price
FROM Products p JOIN Inventory i ON (p.productId = i.productId)
WHERE i.price <= (SELECT AVG(price) FROM Inventory i)
ORDER BY p.productId;
```

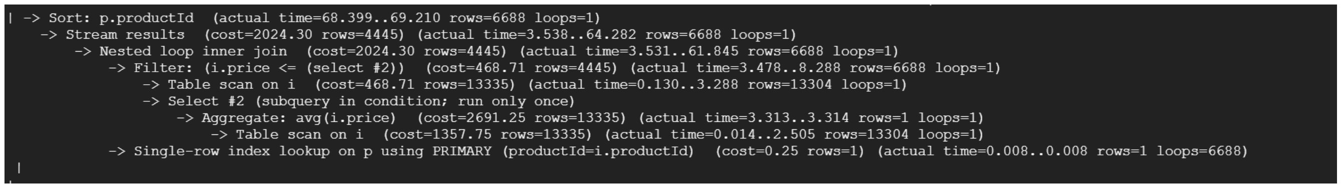
Query 2 Results:



```
+-----------+------------------------------------+-------+
| productId | name                               | price |
+-----------+------------------------------------+-------+
|        10 | Counter-Strike                     | 11.12 |
|        30 | Day of Defeat                      | 12.06 |
|        50 | Half-Life: Opposing Force          | 15.95 |
|        60 | Ricochet                           | 16.07 |
|        70 | Half-Life                          | 15.65 |
|        80 | Counter-Strike: Condition Zero     |  15.8 |
|       220 | Half-Life 2                        | 16.89 |
|       280 | Half-Life: Source                  | 13.17 |
|       320 | Half-Life 2: Deathmatch            | 16.46 |
|       340 | Half-Life 2: Lost Coast            | 12.56 |
|       360 | Half-Life Deathmatch: Source       | 17.34 |
|       400 | Portal                             |  17.1 |
|       550 | Left 4 Dead 2                      | 15.41 |
|       630 | Alien Swarm                        | 10.17 |
|       730 | Counter-Strike: Global Offensive   | 16.78 |
+-----------+------------------------------------+-------+
```

Query 2 Indexing:

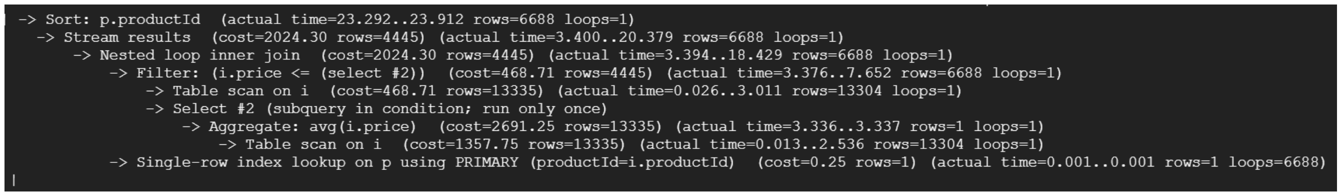
Default:



```
| -> Sort: p.productId  (actual time=68.399..69.210 rows=6688 loops=1)
    -> Stream results  (cost=2024.30 rows=4445) (actual time=3.538..64.282 rows=6688 loops=1)
        -> Nested loop inner join  (cost=2024.30 rows=4445) (actual time=3.531..61.845 rows=6688 loops=1)
            -> Filter: (i.price <= (select #2))  (cost=468.71 rows=4445) (actual time=3.478..8.288 rows=6688 loops=1)
                -> Table scan on i  (cost=468.71 rows=13335) (actual time=0.130..3.288 rows=13304 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Aggregate: avg(i.price)  (cost=2691.25 rows=13335) (actual time=3.313..3.314 rows=1 loops=1)
                        -> Table scan on i  (cost=1357.75 rows=13335) (actual time=0.014..2.505 rows=13304 loops=1)
            -> Single-row index lookup on p using PRIMARY (productId=i.productId)  (cost=0.25 rows=1) (actual time=0.008..0.008 rows=1 loops=6688)
 |
```

Index Design 1 (index by productId in Products table):

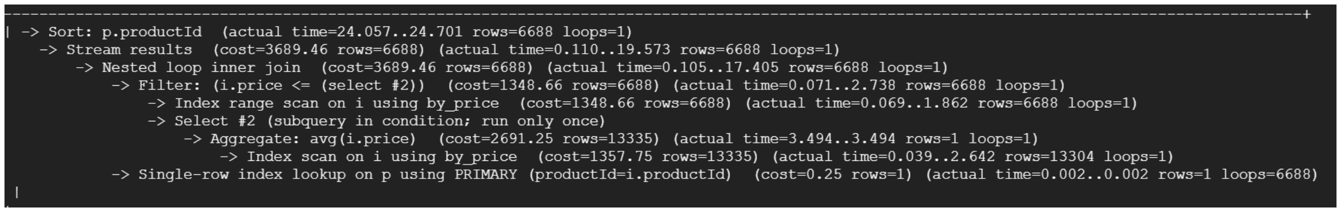
```
      CREATE INDEX product_id_group ON Products(productId);
      DROP INDEX product_id_group ON Products;
```



```
| -> Sort: p.productId  (actual time=23.292..23.912 rows=6688 loops=1)
    -> Stream results  (cost=2024.30 rows=4445) (actual time=3.400..20.379 rows=6688 loops=1)
        -> Nested loop inner join  (cost=2024.30 rows=4445) (actual time=3.394..18.429 rows=6688 loops=1)
            -> Filter: (i.price <= (select #2))  (cost=468.71 rows=4445) (actual time=3.376..7.652 rows=6688 loops=1)
                -> Table scan on i  (cost=468.71 rows=13335) (actual time=0.026..3.011 rows=13304 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Aggregate: avg(i.price)  (cost=2691.25 rows=13335) (actual time=3.336..3.337 rows=1 loops=1)
                        -> Table scan on i  (cost=1357.75 rows=13335) (actual time=0.013..2.536 rows=13304 loops=1)
            -> Single-row index lookup on p using PRIMARY (productId=i.productId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=6688)
 |
```

Index Design 2 (index on price in Prices table):

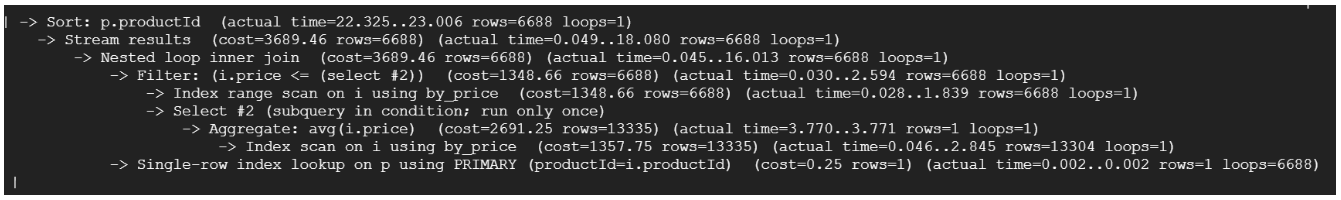
```
    CREATE INDEX by_price ON Inventory(price DESC);
    DROP INDEX by_price ON Inventory;
```



```
----------------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: p.productId  (actual time=24.057..24.701 rows=6688 loops=1)
    -> Stream results  (cost=3689.46 rows=6688) (actual time=0.110..19.573 rows=6688 loops=1)
        -> Nested loop inner join  (cost=3689.46 rows=6688) (actual time=0.105..17.405 rows=6688 loops=1)
            -> Filter: (i.price <= (select #2))  (cost=1348.66 rows=6688) (actual time=0.071..2.738 rows=6688 loops=1)
                -> Index range scan on i using by_price  (cost=1348.66 rows=6688) (actual time=0.069..1.862 rows=6688 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Aggregate: avg(i.price)  (cost=2691.25 rows=13335) (actual time=3.494..3.494 rows=1 loops=1)
                        -> Index scan on i using by_price  (cost=1357.75 rows=13335) (actual time=0.039..2.642 rows=13304 loops=1)
            -> Single-row index lookup on p using PRIMARY (productId=i.productId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=6688)
 |
```

Index Design 3 (index on price in Prices table and by productId in Products table):

```
    CREATE INDEX by_price ON Inventory(price DESC);
    CREATE INDEX product_id_group ON Products(productId);
    DROP INDEX product_id_group ON Products;
    DROP INDEX by_price ON Inventory;
```



```
| -> Sort: p.productId  (actual time=22.325..23.006 rows=6688 loops=1)
    -> Stream results  (cost=3689.46 rows=6688) (actual time=0.049..18.080 rows=6688 loops=1)
        -> Nested loop inner join  (cost=3689.46 rows=6688) (actual time=0.045..16.013 rows=6688 loops=1)
            -> Filter: (i.price <= (select #2))  (cost=1348.66 rows=6688) (actual time=0.030..2.594 rows=6688 loops=1)
                -> Index range scan on i using by_price  (cost=1348.66 rows=6688) (actual time=0.028..1.839 rows=6688 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Aggregate: avg(i.price)  (cost=2691.25 rows=13335) (actual time=3.770..3.771 rows=1 loops=1)
                        -> Index scan on i using by_price  (cost=1357.75 rows=13335) (actual time=0.046..2.845 rows=13304 loops=1)
            -> Single-row index lookup on p using PRIMARY (productId=i.productId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=6688)
 |
```

Justification:
The first index design that we chose, indexing on productId in the Products table, ran slightly faster than the default indexing. It minimized the single-row index lookup for the Products table. However, its nested loop inner join and stream results processes were not much faster than the default indexing. The second index design that we chose, indexing on price in the Prices table, ran significantly faster than both the first design and the default indexing. It significantly reduced the time for the nested loop inner join. The 3rd indexing design was the one we chose to use, because it was the fastest. The third design was fastest because it grouped by the price, which like the second design increased the efficiency of the joining process and the index on productId greatly decreased the time needed to perform the sort on productId. Together, these two indices made the third index design the most efficient way to perform query 2.