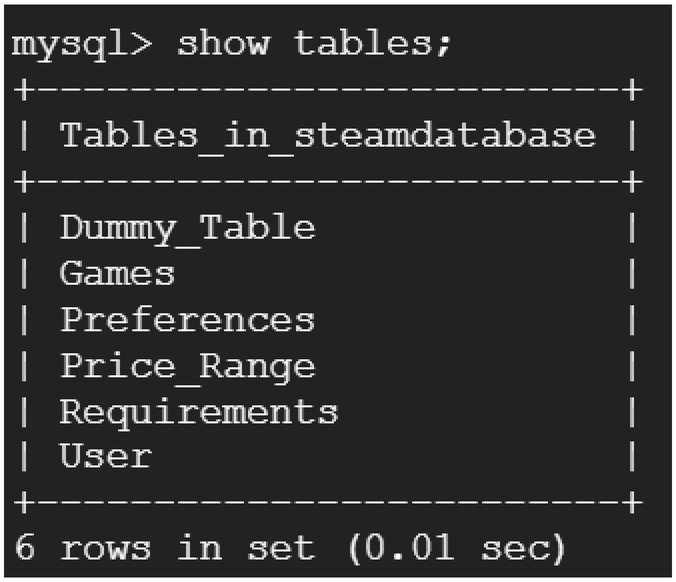
DDL Commands to create our tables:

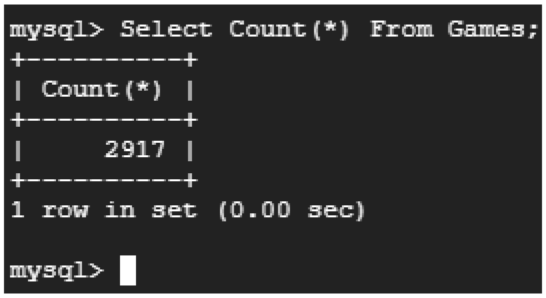
```
CREATE TABLE IF NOT EXISTS steamdatabase.Games (
  games_Id INT NOT NULL,
  game_Name VARCHAR(255) NULL,
  description VARCHAR(512) NULL,
  release_Date VARCHAR(255) NULL,
  languages VARCHAR(255) NULL,
  grade INT NOT NULL,
  PRIMARY KEY (games_Id));

CREATE TABLE IF NOT EXISTS steamdatabase.Requirements (
  game_Id INT NOT NULL,
  pc_Req_Min VARCHAR(255) NULL,
  pc_Req_Text VARCHAR(255) NULL,
  PRIMARY KEY (game_Id),
  INDEX game_Id_idx (game_Id ASC) VISIBLE);

CREATE TABLE IF NOT EXISTS steamdb.Price_Range (
  grade INT NOT NULL AUTO_INCREMENT,
  initial_Price REAL NULL,
  final_Price REAL NULL,
  PRIMARY KEY (grade));

CREATE TABLE IF NOT EXISTS `steamdatabase`.`User` (
  `user_Id` INT NOT NULL AUTO_INCREMENT,
  `email` VARCHAR(255) NULL,
  `phone` INT NULL,
  `password` VARCHAR(255) NULL,
  PRIMARY KEY (`user_Id`));

CREATE TABLE IF NOT EXISTS `steamdatabase`.`Preferences` (
`user_Id` INT NOT NULL,
`game_Id` INT NOT NULL,
`grade` INT NOT NULL,
PRIMARY KEY (`user_Id`),
CONSTRAINT `preferences.grade`
  FOREIGN KEY (`grade`)
  REFERENCES `steamdatabase`.`Price_Range` (`grade`)
  ON DELETE RESTRICT
  ON UPDATE NO ACTION,
CONSTRAINT `preferences.user_Id`
  FOREIGN KEY (`user_Id`)
  REFERENCES `steamdatabase`.`User` (`user_Id`)
  ON DELETE CASCADE
  ON UPDATE CASCADE,
CONSTRAINT `preferences.game_Id`
  FOREIGN KEY (`game_Id`)
  REFERENCES `steamdatabase`.`Games` (`games_Id`)
  ON DELETE CASCADE
  ON UPDATE RESTRICT);
```



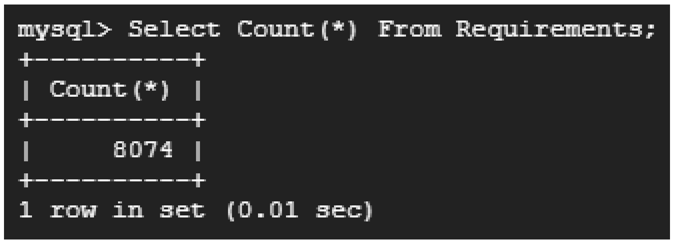
```
mysql> show tables;
+-------------------------+
| Tables_in_steamdatabase |
+-------------------------+
| Dummy_Table             |
| Games                   |
| Preferences             |
| Price_Range             |
| Requirements            |
| User                    |
+-------------------------+
6 rows in set (0.01 sec)
```

ignore Dummy_Table

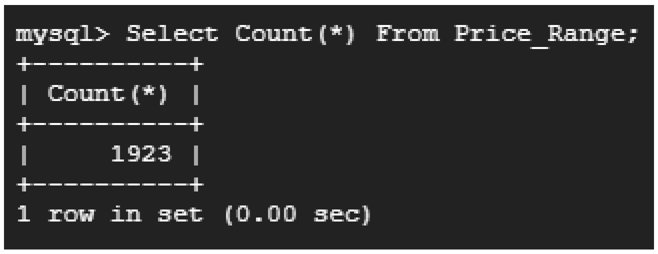
Three tables with 1000 rows each:



```
mysql> Select Count(*) From Games;
+----------+
| Count(*) |
+----------+
|     2917 |
+----------+
1 row in set (0.00 sec)

mysql> 
```



```
mysql> Select Count(*) From Requirements;
+----------+
| Count(*) |
+----------+
|     8074 |
+----------+
1 row in set (0.01 sec)
```

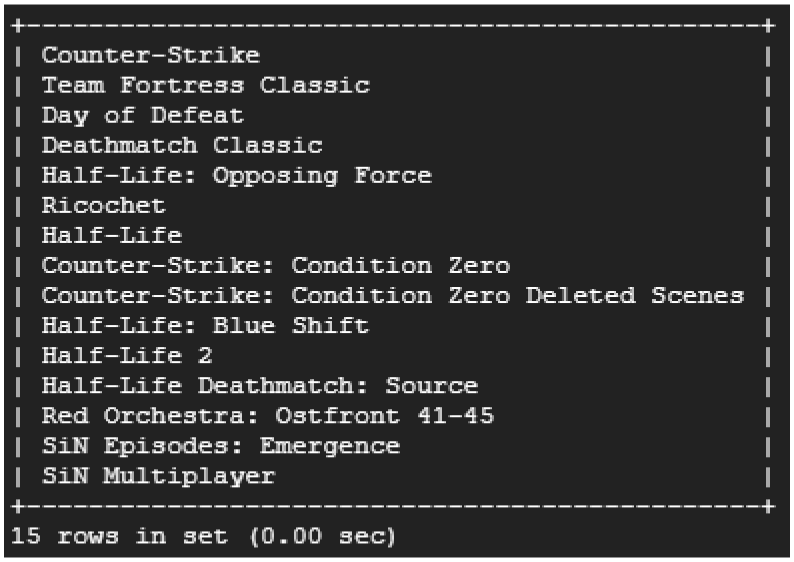


```
mysql> Select Count(*) From Price_Range;
+----------+
| Count(*) |
+----------+
|     1923 |
+----------+
1 row in set (0.00 sec)
```

Advanced SQL Command 1:

Retrieve the names of games that have requirements mentioned and are within a certain price range (e.g., below $20).

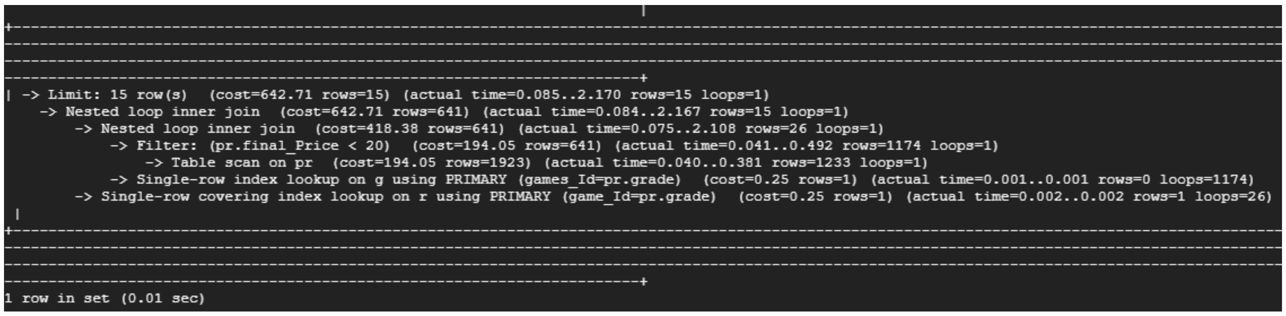
```
SELECT g.game_Name
FROM steamdatabase.Games AS g
JOIN steamdatabase.Requirements AS r ON g.games_Id = r.game_Id
WHERE g.games_Id IN (
   SELECT pr.grade
   FROM steamdatabase.Price_Range AS pr
   WHERE pr.final_Price < 20
)
LIMIT 15;
```



```
+-----------------------------------------------+
| Counter-Strike                                |
| Team Fortress Classic                         |
| Day of Defeat                                 |
| Deathmatch Classic                            |
| Half-Life: Opposing Force                     |
| Ricochet                                      |
| Half-Life                                     |
| Counter-Strike: Condition Zero                |
| Counter-Strike: Condition Zero Deleted Scenes |
| Half-Life: Blue Shift                         |
| Half-Life 2                                   |
| Half-Life Deathmatch: Source                  |
| Red Orchestra: Ostfront 41-45                 |
| SiN Episodes: Emergence                       |
| SiN Multiplayer                               |
+-----------------------------------------------+
15 rows in set (0.00 sec)
```

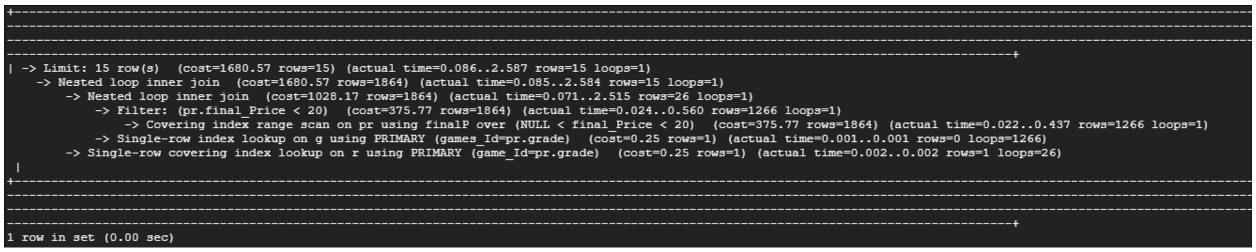
Indexing for 1st advanced subquery:

Original Cost:



```
| -> Limit: 15 row(s)  (cost=642.71 rows=15) (actual time=0.085..2.170 rows=15 loops=1)
    -> Nested loop inner join  (cost=642.71 rows=641) (actual time=0.084..2.167 rows=15 loops=1)
        -> Nested loop inner join  (cost=418.38 rows=641) (actual time=0.075..2.108 rows=26 loops=1)
            -> Filter: (pr.final_Price < 20)  (cost=194.05 rows=641) (actual time=0.041..0.492 rows=1174 loops=1)
                -> Table scan on pr  (cost=194.05 rows=1923) (actual time=0.040..0.381 rows=1233 loops=1)
            -> Single-row index lookup on g using PRIMARY (games_Id=pr.grade)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1174)
        -> Single-row covering index lookup on r using PRIMARY (game_Id=pr.grade)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=26)
 |

1 row in set (0.01 sec)
```

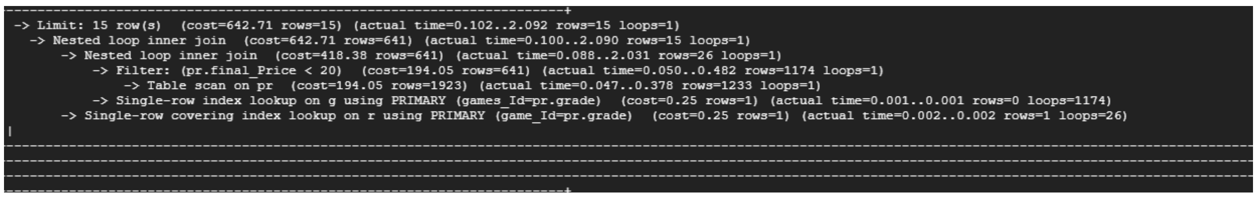
Index 1: CREATE INDEX finalP ON steamdatabase.Price_Range (final_Price);



```
| -> Limit: 15 row(s)  (cost=1680.57 rows=15) (actual time=0.086..2.587 rows=15 loops=1)
    -> Nested loop inner join  (cost=1680.57 rows=1864) (actual time=0.085..2.584 rows=15 loops=1)
        -> Nested loop inner join  (cost=1028.17 rows=1864) (actual time=0.071..2.515 rows=26 loops=1)
            -> Filter: (pr.final_Price < 20)  (cost=375.77 rows=1864) (actual time=0.024..0.560 rows=1266 loops=1)
                -> Covering index range scan on pr using finalP over (NULL < final_Price < 20)  (cost=375.77 rows=1864) (actual time=0.022..0.437 rows=1266 loops=1)
            -> Single-row index lookup on g using PRIMARY (games_Id=pr.grade)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1266)
        -> Single-row covering index lookup on r using PRIMARY (game_Id=pr.grade)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=26)
 |

1 row in set (0.00 sec)
```

In an attempt to optimize our subquery, we tried adding an index to our Final_Price attribute. After creating the index and re-running the command, the cost tripled from ~600 to ~1800. This showed that our original query was optimized enough that adding in an index only increased the cost of it.

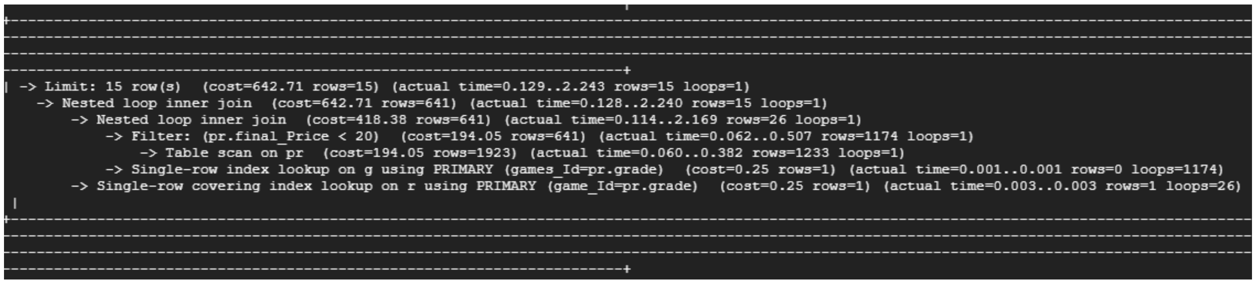
Index 2: CREATE INDEX idx_grade ON steamdatabase.Price_Range (grade);



```
 -> Limit: 15 row(s)  (cost=642.71 rows=15) (actual time=0.102..2.092 rows=15 loops=1)
   -> Nested loop inner join  (cost=642.71 rows=641) (actual time=0.100..2.090 rows=15 loops=1)
       -> Nested loop inner join  (cost=418.38 rows=641) (actual time=0.088..2.031 rows=26 loops=1)
           -> Filter: (pr.final_Price < 20)  (cost=194.05 rows=641) (actual time=0.050..0.482 rows=1174 loops=1)
               -> Table scan on pr  (cost=194.05 rows=1923) (actual time=0.047..0.378 rows=1233 loops=1)
           -> Single-row index lookup on g using PRIMARY (games_Id=pr.grade)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1174)
       -> Single-row covering index lookup on r using PRIMARY (game_Id=pr.grade)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=26)
|
```

Our second index was created for our grade attribute in our Price_Range table. This was selected within our subquery but creating an index for this did not decrease the cost of the query but it also did not increase either. Our time for the query did decrease though after creating the index, it went from 2.170 to 2.092.

Index 3: CREATE INDEX idx_games_id ON steamdatabase.Games (games_Id);



```
| -> Limit: 15 row(s)  (cost=642.71 rows=15) (actual time=0.129..2.243 rows=15 loops=1)
    -> Nested loop inner join  (cost=642.71 rows=641) (actual time=0.128..2.240 rows=15 loops=1)
        -> Nested loop inner join  (cost=418.38 rows=641) (actual time=0.114..2.169 rows=26 loops=1)
            -> Filter: (pr.final_Price < 20)  (cost=194.05 rows=641) (actual time=0.062..0.507 rows=1174 loops=1)
                -> Table scan on pr  (cost=194.05 rows=1923) (actual time=0.060..0.382 rows=1233 loops=1)
            -> Single-row index lookup on g using PRIMARY (games_Id=pr.grade)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1174)
        -> Single-row covering index lookup on r using PRIMARY (game_Id=pr.grade)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=26)
 |
```

Our third index is created for games_Id in our Games table. This index also did not decrease or increase the cost of the query. This index also increased the time for the query from 2.170 to 2.243.

Advance SQL Command 2:

Find the names and descriptions of games that are more expensive than the average final price of all games. This will help users discover premium games that are priced above average.

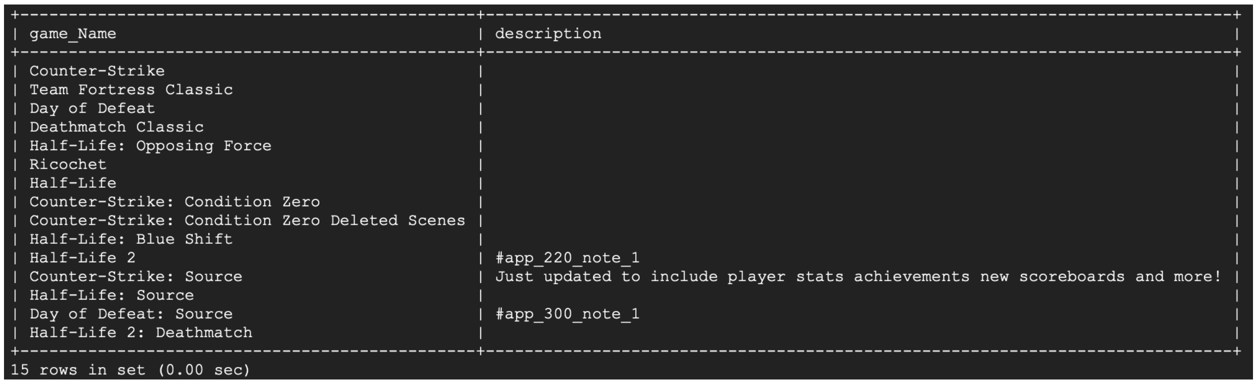
SELECT g.game_Name, g.description

FROM steamdatabase.Games AS g

JOIN steamdatabase.Price_Range AS pr ON g.grade = pr.grade

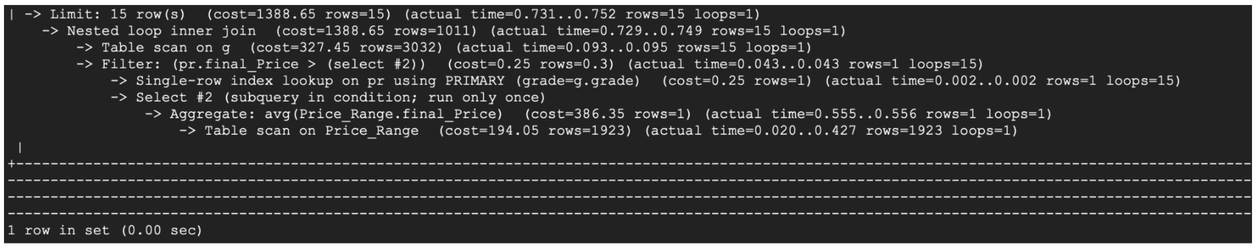
WHERE pr.final_Price > (SELECT AVG(final_Price) FROM steamdatabase.Price_Range)

LIMIT 15;



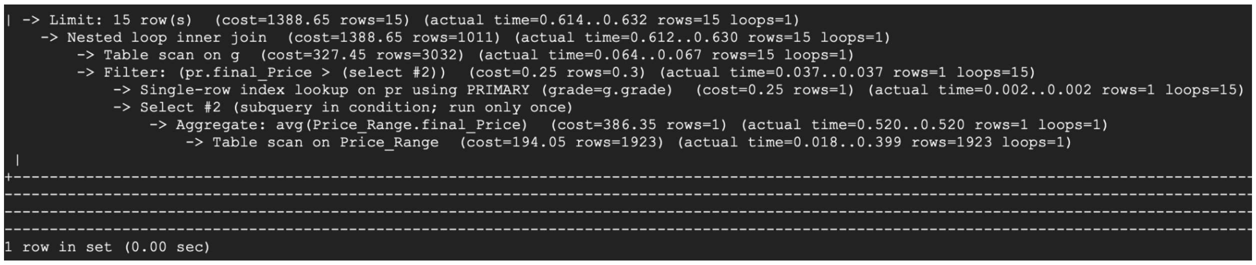
```
+-----------------------------------------------+------------------------------------------------------------------------------+
| game_Name                                     | description                                                                  |
+-----------------------------------------------+------------------------------------------------------------------------------+
| Counter-Strike                                |                                                                              |
| Team Fortress Classic                         |                                                                              |
| Day of Defeat                                 |                                                                              |
| Deathmatch Classic                            |                                                                              |
| Half-Life: Opposing Force                     |                                                                              |
| Ricochet                                      |                                                                              |
| Half-Life                                     |                                                                              |
| Counter-Strike: Condition Zero                |                                                                              |
| Counter-Strike: Condition Zero Deleted Scenes |                                                                              |
| Half-Life: Blue Shift                         |                                                                              |
| Half-Life 2                                   | #app_220_note_1                                                              |
| Counter-Strike: Source                        | Just updated to include player stats achievements new scoreboards and more! |
| Half-Life: Source                             |                                                                              |
| Day of Defeat: Source                         | #app_300_note_1                                                              |
| Half-Life 2: Deathmatch                       |                                                                              |
+-----------------------------------------------+------------------------------------------------------------------------------+
15 rows in set (0.00 sec)
```

Original Cost:



```
| -> Limit: 15 row(s)  (cost=1388.65 rows=15) (actual time=0.731..0.752 rows=15 loops=1)
    -> Nested loop inner join  (cost=1388.65 rows=1011) (actual time=0.729..0.749 rows=15 loops=1)
        -> Table scan on g  (cost=327.45 rows=3032) (actual time=0.093..0.095 rows=15 loops=1)
        -> Filter: (pr.final_Price > (select #2))  (cost=0.25 rows=0.3) (actual time=0.043..0.043 rows=1 loops=15)
            -> Single-row index lookup on pr using PRIMARY (grade=g.grade)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=15)
            -> Select #2 (subquery in condition; run only once)
                -> Aggregate: avg(Price_Range.final_Price)  (cost=386.35 rows=1) (actual time=0.555..0.556 rows=1 loops=1)
                    -> Table scan on Price_Range  (cost=194.05 rows=1923) (actual time=0.020..0.427 rows=1923 loops=1)
 |
+---------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

Index 1: CREATE INDEX idx_games_grade ON steamdatabase.Games(grade);



```
| -> Limit: 15 row(s)  (cost=1388.65 rows=15) (actual time=0.614..0.632 rows=15 loops=1)
    -> Nested loop inner join  (cost=1388.65 rows=1011) (actual time=0.612..0.630 rows=15 loops=1)
        -> Table scan on g  (cost=327.45 rows=3032) (actual time=0.064..0.067 rows=15 loops=1)
        -> Filter: (pr.final_Price > (select #2))  (cost=0.25 rows=0.3) (actual time=0.037..0.037 rows=1 loops=15)
            -> Single-row index lookup on pr using PRIMARY (grade=g.grade)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=15)
            -> Select #2 (subquery in condition; run only once)
                -> Aggregate: avg(Price_Range.final_Price)  (cost=386.35 rows=1) (actual time=0.520..0.520 rows=1 loops=1)
                    -> Table scan on Price_Range  (cost=194.05 rows=1923) (actual time=0.018..0.399 rows=1923 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

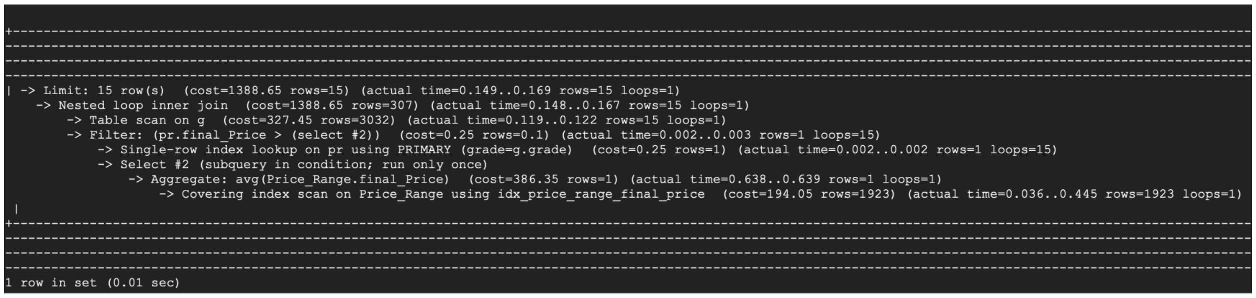
Table Scan on g (Games table):
- Before Index: actual time=0.093..0.095
- After Index: actual time=0.064..0.067

Aggregate Function on Price_Range table:
- Before Index: actual time=0.555..0.556
- After Index: actual time=0.520..0.520

Overall Query Execution Time:
- Before Index: actual time=0.731..0.752
- After Index: actual time=0.614..0.632

Based on the provided EXPLAIN ANALYZE outputs, the index on the grade column of the Games table (idx_games_grade) has improved the query performance. However, it's worth noting that the main bottleneck still appears to be the full table scan on the Price_Range table to compute the average final_Price. An index on the final_Price column of the Price_Range table could further optimize this operation.

Index 2: CREATE INDEX idx_price_range_final_price ON steamdatabase.Price_Range(final_Price);



```
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
| -> Limit: 15 row(s)  (cost=1388.65 rows=15) (actual time=0.149..0.169 rows=15 loops=1)
    -> Nested loop inner join  (cost=1388.65 rows=307) (actual time=0.148..0.167 rows=15 loops=1)
        -> Table scan on g  (cost=327.45 rows=3032) (actual time=0.119..0.122 rows=15 loops=1)
        -> Filter: (pr.final_Price > (select #2))  (cost=0.25 rows=0.1) (actual time=0.002..0.003 rows=1 loops=15)
            -> Single-row index lookup on pr using PRIMARY (grade=g.grade)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=15)
            -> Select #2 (subquery in condition; run only once)
                -> Aggregate: avg(Price_Range.final_Price)  (cost=386.35 rows=1) (actual time=0.638..0.639 rows=1 loops=1)
                    -> Covering index scan on Price_Range using idx_price_range_final_price  (cost=194.05 rows=1923) (actual time=0.036..0.445 rows=1923 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
1 row in set (0.01 sec)
```

Overall Query Execution Time:
- Before Index (on final_Price): actual time=0.614..0.632
- After Index (on final_Price): actual time=0.149..0.169

Aggregate Function on Price_Range table:
- Before Index: actual time=0.520..0.520

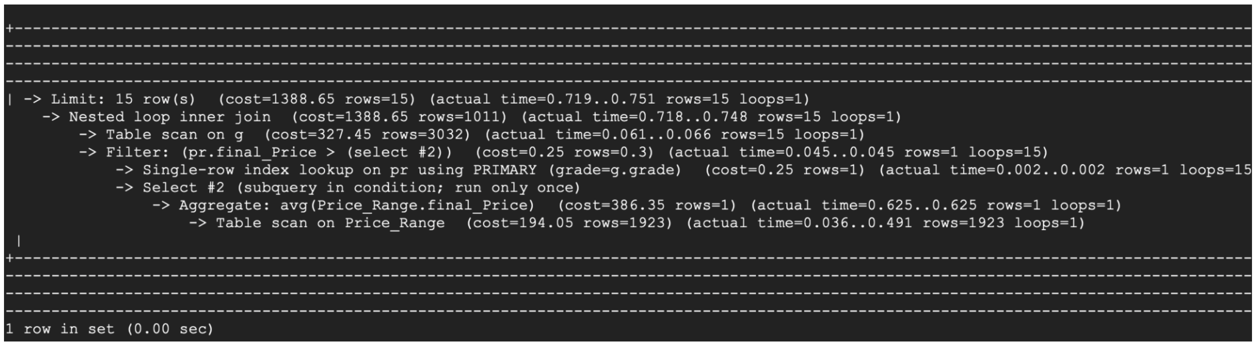
- After Index: actual time=0.638..0.639

Scanning of Price_Range table:

- Before Index: Table scan on Price_Range ... actual time=0.018..0.399
- After Index: Covering index scan on Price_Range using idx_price_range_final_price ... actual time=0.036..0.445

In summary, the addition of the index on the final_Price column of the Price_Range table has significantly optimized the query. The database now leverages the index to efficiently access rows in the Price_Range table, resulting in faster query execution times. This highlights the importance of indexing columns that are frequently used in filtering conditions or join operations.

Index 3: CREATE INDEX idx_price_range_grade ON steamdatabase.Price_Range(grade);



```
+--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
| -> Limit: 15 row(s)  (cost=1388.65 rows=15) (actual time=0.719..0.751 rows=15 loops=1)
    -> Nested loop inner join  (cost=1388.65 rows=1011) (actual time=0.718..0.748 rows=15 loops=1)
        -> Table scan on g  (cost=327.45 rows=3032) (actual time=0.061..0.066 rows=15 loops=1)
        -> Filter: (pr.final_Price > (select #2))  (cost=0.25 rows=0.3) (actual time=0.045..0.045 rows=1 loops=15)
            -> Single-row index lookup on pr using PRIMARY (grade=g.grade)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=15
            -> Select #2 (subquery in condition; run only once)
                -> Aggregate: avg(Price_Range.final_Price)  (cost=386.35 rows=1) (actual time=0.625..0.625 rows=1 loops=1)
                    -> Table scan on Price_Range  (cost=194.05 rows=1923) (actual time=0.036..0.491 rows=1923 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

Overall Query Execution Time:

- Before Index (on grade): actual time=0.614..0.632
- After Index (on grade): actual time=0.719..0.751

Table Scan on g (Games table):

- Before Index: actual time=0.064..0.067
- After Index: actual time=0.061..0.066

Aggregate Function on Price_Range table:

- Before Index: actual time=0.520..0.520
- After Index: actual time=0.625..0.625

Scanning of Price_Range table:

- Before Index: actual time=0.018..0.399
- After Index: actual time=0.036..0.491

In summary, the addition of the index on the grade column of the Price_Range table did not provide significant performance improvements for this particular query. In some cases, the performance was slightly worse. This highlights the fact that not all indexes will lead to better performance, and it's essential to monitor and analyze the actual impact of indexing decisions. In practice, if an index doesn't provide a clear benefit, it might be a candidate for removal to save storage and reduce maintenance overhead.