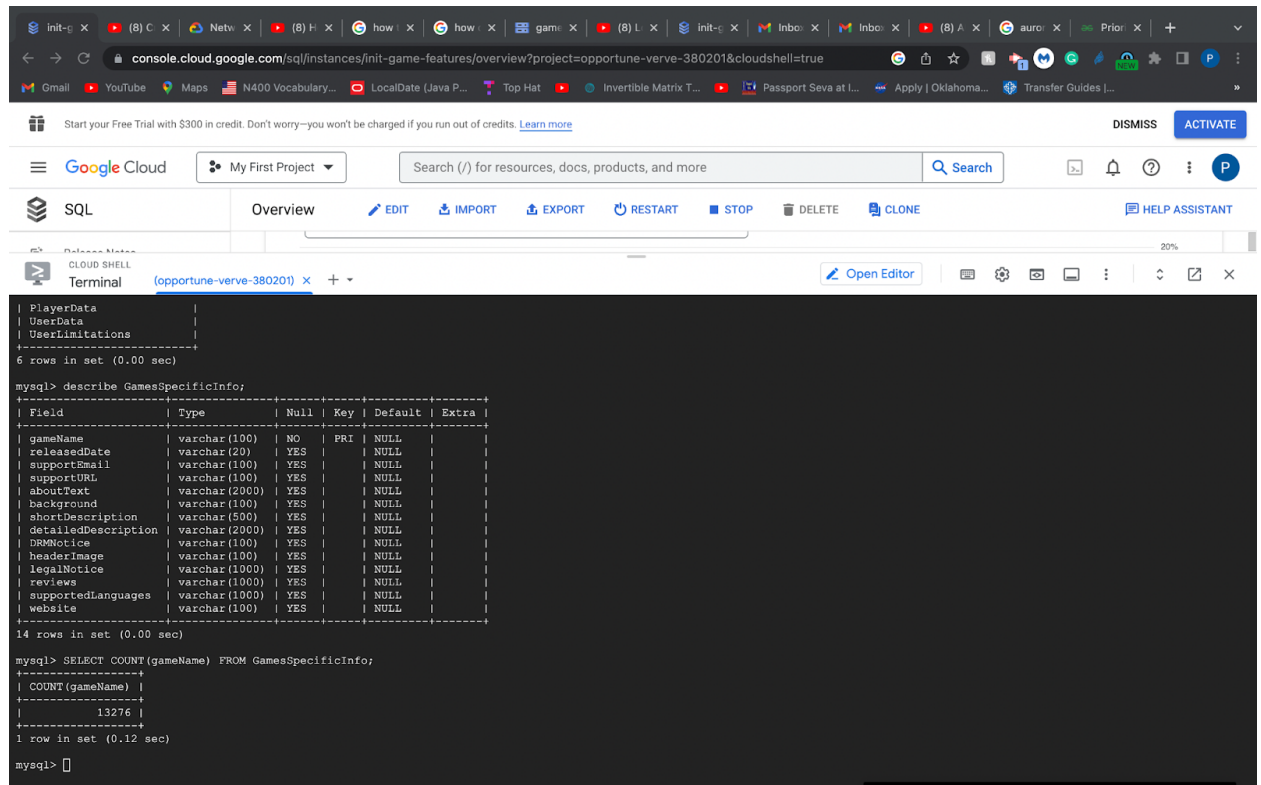


1. Implement at least four main tables (i.e., tables that include core application information, not auxiliary information, such as user profiles and login information). → **DONE**



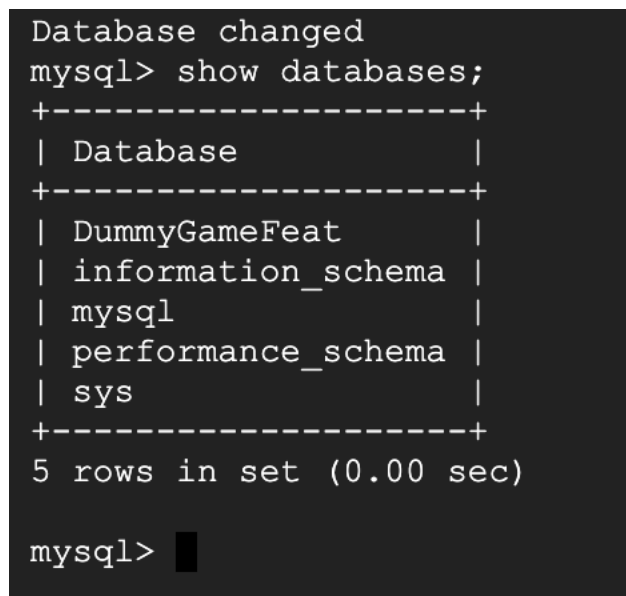
The screenshot shows the Google Cloud SQL console interface. The terminal window displays the following commands and results:

```
mysql> describe GamesSpecificInfo;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| gameId | int(11) | NO | PRI | NULL | |
| gameName | varchar(100) | NO | PRI | NULL | |
| releaseDate | varchar(20) | YES | | NULL | |
| supportEmail | varchar(100) | YES | | NULL | |
| supportURL | varchar(100) | YES | | NULL | |
| aboutText | varchar(2000) | YES | | NULL | |
| background | varchar(100) | YES | | NULL | |
| shortDescription | varchar(500) | YES | | NULL | |
| detailedDescription | varchar(2000) | YES | | NULL | |
| DRMNotice | varchar(100) | YES | | NULL | |
| headerImage | varchar(100) | YES | | NULL | |
| legalNotice | varchar(1000) | YES | | NULL | |
| reviews | varchar(1000) | YES | | NULL | |
| supportedLanguages | varchar(1000) | YES | | NULL | |
| website | varchar(100) | YES | | NULL | |
+-----+-----+-----+-----+-----+-----+
14 rows in set (0.00 sec)

mysql> SELECT COUNT(gameName) FROM GamesSpecificInfo;
+-----+
| COUNT(gameName) |
+-----+
| 13276 |
+-----+
1 row in set (0.12 sec)

mysql>
```

Database created → DummyGameFeat



```
Database changed
mysql> show databases;
+-----+
| Database |
+-----+
| DummyGameFeat |
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
5 rows in set (0.00 sec)

mysql>
```

Tables in our database DummyGameFeat

```
mysql> use DummyGameFeat;
Database changed
mysql> show tables;
+-----+
| Tables_in_DummyGameFeat |
+-----+
| GameCategories           |
| GamesOwned               |
| GamesSpecificInfo        |
| PlayerData               |
| UserData                 |
| UserLimitations          |
+-----+
6 rows in set (0.01 sec)
```

2. In the Database Design markdown or pdf, provide the Data Definition Language (DDL) commands you all used to create each of these tables in the database. Here's the syntax of the CREATE TABLE DDL command:
CREATE TABLE table_name (column1 datatype, column2 datatype, column3 datatype,...); → **DONE** → **uploaded last time on GitHub but copy pasted on the last page for reference.**
3. Insert data into these tables. You should insert at least 1000 rows **each in** three of the tables. Try to use real data, but if you cannot find a good dataset for a particular table, you may use auto-generated data. → **DONE** → **Andrew Zhao working**

Table GamesSpecificInfo:

```
mysql> describe GamesSpecificInfo;
```

Field	Type	Null	Key	Default	Extra
gameName	varchar(100)	NO	PRI	NULL	
releasedDate	varchar(20)	YES		NULL	
supportEmail	varchar(100)	YES		NULL	
supportURL	varchar(100)	YES		NULL	
aboutText	varchar(2000)	YES		NULL	
background	varchar(100)	YES		NULL	
shortDescription	varchar(500)	YES		NULL	
detailedDescription	varchar(2000)	YES		NULL	
DRMNotice	varchar(100)	YES		NULL	
headerImage	varchar(100)	YES		NULL	
legalNotice	varchar(1000)	YES		NULL	
reviews	varchar(1000)	YES		NULL	
supportedLanguages	varchar(1000)	YES		NULL	
website	varchar(100)	YES		NULL	

```
mysql> SELECT COUNT(gameName) FROM GamesSpecificInfo;
+-----+
| COUNT(gameName) |
+-----+
|          13276 |
+-----+
1 row in set (0.12 sec)

mysql> █
```

Table: UserData

```
mysql> describe UserData;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| playerId  | int           | NO   | PRI | NULL    |      |
| password  | varchar(100)  | YES  |     | NULL    |      |
| userName  | varchar(100)  | YES  |     | NULL    |      |
| index_    | int           | YES  |     | NULL    |      |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT COUNT(playerId) FROM UserData;
+-----+
| COUNT(playerId) |
+-----+
|          1000 |
+-----+
1 row in set (0.05 sec)
```

```
mysql> SELECT COUNT(playerId) FROM UserData;
+-----+
| COUNT(playerId) |
+-----+
|          1000 |
+-----+
1 row in set (0.05 sec)

mysql> describe GamesOwned;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| index_         | int           | NO   | PRI | NULL    |      |
| gameName       | varchar(100)  | YES  |     | NULL    |      |
| played        | tinyint(1)    | YES  |     | NULL    |      |
| hoursPlayed    | int           | YES  |     | NULL    |      |
| playerId       | int           | YES  |     | NULL    |      |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.01 sec)

mysql> SELECT COUNT(hoursPlayed) FROM GamesOwned;
+-----+
| COUNT(hoursPlayed) |
+-----+
|          32793 |
+-----+
1 row in set (0.01 sec)
```

4. As a group, develop two advanced SQL queries related to the project that are different from one another. The two advance queries are expected to be part of your final application. The queries should **each** involve **at least two** of the following SQL concepts:
- Join of multiple relations
 - Set operations
 - Aggregation via GROUP BY
 - Subqueries

→ **Query 1: Finding the most popular games based on Critics // popular games based on played**

Complicated query:

```
SELECT g.gameName, SUM(p.playersTotal) AS totalPlayers
FROM GamesSpecificInfo g
NATURAL JOIN PlayerData p
NATURAL JOIN GameCategories c
WHERE p.metaCritic > 75 AND p.playersTotal >= 4400
GROUP BY g.gameName
ORDER BY totalPlayers DESC
LIMIT 15;
```

→ **Query 2: Complex Query- Friend finder**

```
SELECT ud.userName, SUM(go.hoursPlayed) AS totalHoursPlayed
FROM UserData AS ud
INNER JOIN GamesOwned AS go ON ud.playerId = go.playerId
WHERE go.playerId IN (
    SELECT uda.playerId
    FROM UserData AS uda
    INNER JOIN GamesOwned AS goa ON uda.playerId = goa.playerId
    WHERE goa.gameName LIKE 'Alien Swarm'
)
GROUP BY ud.userName
ORDER BY totalHoursPlayed DESC
```

5. Execute your advanced SQL queries and provide a screenshot of the top 15 rows of each query result (you can use the LIMIT clause to select the top 15 rows). If your output is less than 15 rows, say that in your output.

Output:

```
mysql>
mysql>
mysql> SELECT g.gameName, SUM(p.playersTotal) AS totalPlayers
-> FROM GamesSpecificInfo g
-> NATURAL JOIN PlayerData p
-> NATURAL JOIN GameCategories c
-> WHERE p.metaCritic > 75 AND p.playersTotal >= 4400
-> GROUP BY g.gameName
-> ORDER BY totalPlayers DESC
-> LIMIT 15;
```

gameName	totalPlayers
Dota 2	90687580
Team Fortress 2	37878812
Counter-Strike: Global Offensive	25150372
Left 4 Dead 2	13583400
Counter-Strike: Source	11472993
The Elder Scrolls V: Skyrim	10903558
Sid Meier's Civilization V	9150595
Counter-Strike	9140731
Terraria	7764044
Portal 2	7282849
Portal	6864247
PAYDAY 2	6745338
Borderlands 2	6263964
War Thunder	6213387
Grand Theft Auto V	5756584

```
15 rows in set (1.04 sec)
```

```
mysql> SELECT ud.userName, SUM(go.hoursPlayed) AS totalHoursPlayed
-> FROM UserData AS ud
-> INNER JOIN GamesOwned AS go ON ud.playerId = go.playerId
-> WHERE go.playerId IN (
->   SELECT uda.playerId
->   FROM UserData AS uda
->   INNER JOIN GamesOwned AS goa ON uda.playerId = goa.playerId
->   WHERE goa.gameName LIKE
->   "Alien Swarm"
-> ) GROUP BY ud.userName
-> ORDER BY totalHoursPlayed DESC;
+-----+-----+
| userName | totalHoursPlayed |
+-----+-----+
| rswynelh | 4205 |
| rrotherforthhh | 3195 |
| raugar52 | 2804 |
| mnorthino6 | 2772 |
| mcleetoneg | 2382 |
| bespinojo | 1317 |
| umanshawb1 | 1163 |
| haisman91 | 618 |
| whuelinip | 561 |
| tbrokenbrowh7 | 542 |
| dtofanoe2 | 314 |
| tdowngateem | 17 |
+-----+-----+
12 rows in set (0.03 sec)

mysql>
```

Output was less than 12 rows

Indexing: As a team, for each advanced query:

Indexing analysis on first query: Finding popular games

Exploration 1: Indexing primary keys

1. Original performance of the query without any indexing

```
| -> Sort: totalPlayers DESC (actual time=13.744..13.810 rows=880 loops=1)
-> Table scan on <temporary> (actual time=0.021..0.099 rows=880 loops=1)
-> Aggregate using temporary table (actual time=13.227..13.356 rows=880 loops=1)
-> Nested loop inner join (cost=1006.26 rows=469) (actual time=1.299..12.095 rows=880 loops=1)
-> Nested loop inner join (cost=834.38 rows=469) (actual time=1.286..9.114 rows=880 loops=1)
-> Filter: (p.playersTotal >= 4400) (cost=422.36 rows=469) (actual time=1.239..4.006 rows=880 loops=1)
-> Index range scan on p using another, with index condition: (p.metaCritic > 75) (cost=422.36 rows=938) (actual time=1.237..3.904 rows=938 loops=1)
-> Single-row index lookup on g using PRIMARY (gameName=p.gameName) (cost=0.78 rows=1) (actual time=0.006..0.006 rows=1 loops=880)
-> Single-row index lookup on c using PRIMARY (gameName=p.gameName) (cost=0.27 rows=1) (actual time=0.003..0.003 rows=1 loops=880)
```

2. Indexing on primary key result

```
| -> Sort: totalPlayers DESC (actual time=10.156..10.219 rows=880 loops=1)
-> Table scan on <temporary> (actual time=0.003..0.079 rows=880 loops=1)
-> Aggregate using temporary table (actual time=9.547..9.675 rows=880 loops=1)
-> Nested loop inner join (cost=758.39 rows=469) (actual time=0.557..8.472 rows=880 loops=1)
-> Nested loop inner join (cost=586.51 rows=469) (actual time=0.548..6.582 rows=880 loops=1)
-> Filter: (p.playersTotal >= 4400) (cost=422.36 rows=469) (actual time=0.529..2.469 rows=880 loops=1)
-> Index range scan on p using another, with index condition: (p.metaCritic > 75) (cost=422.36 rows=938) (actual time=0.528..2.371 rows=938 loops=1)
-> Single-row index lookup on g using PRIMARY (gameName=p.gameName) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=880)
-> Single-row index lookup on c using PRIMARY (gameName=p.gameName) (cost=0.27 rows=1) (actual time=0.002..0.002 rows=1 loops=880)
```

3. We chose to index the primary key b/c we constantly access it and primary keys are technically already indexed so we inferred that indexing a primary key should make
4. After running the explain analyze and comparing the times and costs between different scans, and nested loops here's what we learned. Sorting the players in a descending order took less time but it still had to look at all 880 rows. The most obvious change was the cost of inner join which reduced by 1/3 and the single row index lookup on gamesSpecificInfo reduced b/c that's where the primary keys were indexed. Our assumption of putting indexes on primary keys won't change the performance was incorrect.

Exploration 2: Indexing PlayerData(metaCritic)

1. Original performance of the query without any indexing

```
| -> Sort: totalPlayers DESC (actual time=13.744..13.810 rows=880 loops=1)
    -> Table scan on <temporary> (actual time=0.021..0.099 rows=880 loops=1)
        -> Aggregate using temporary table (actual time=13.227..13.356 rows=880 loops=1)
            -> Nested loop inner join (cost=1006.26 rows=469) (actual time=1.299..12.095 rows=880 loops=1)
                -> Nested loop inner join (cost=834.38 rows=469) (actual time=1.286..9.114 rows=880 loops=1)
                    -> Filter: (p.playersTotal >= 4400) (cost=422.36 rows=469) (actual time=1.239..4.006 rows=880 loops=1)
                        -> Index range scan on p using another, with index condition: (p.metaCritic > 75) (cost=422.36 rows=938) (actual time=1.237..3.904 rows=938 loops=1)
                            -> Single-row index lookup on g using PRIMARY (gameName=p.gameName) (cost=0.78 rows=1) (actual time=0.006..0.006 rows=1 loops=880)
                                -> Single-row index lookup on c using PRIMARY (gameName=p.gameName) (cost=0.27 rows=1) (actual time=0.003..0.003 rows=1 loops=880)
```

2. Indexing on PlayerData(metaCritic) result

```
| -> Sort: totalPlayers DESC (actual time=9.561..9.643 rows=880 loops=1)
    -> Table scan on <temporary> (actual time=0.003..0.075 rows=880 loops=1)
        -> Aggregate using temporary table (actual time=9.068..9.192 rows=880 loops=1)
            -> Nested loop inner join (cost=758.39 rows=469) (actual time=0.261..7.886 rows=880 loops=1)
                -> Nested loop inner join (cost=586.51 rows=469) (actual time=0.255..6.272 rows=880 loops=1)
                    -> Filter: (p.playersTotal >= 4400) (cost=422.36 rows=469) (actual time=0.242..2.112 rows=880 loops=1)
                        -> Index range scan on p using another, with index condition: (p.metaCritic > 75) (cost=422.36 rows=938) (actual time=0.241..2.010 rows=938 loops=1)
                            -> Single-row index lookup on g using PRIMARY (gameName=p.gameName) (cost=0.25 rows=1) (actual time=0.004..0.005 rows=1 loops=880)
                                -> Single-row index lookup on c using PRIMARY (gameName=p.gameName) (cost=0.27 rows=1) (actual time=0.002..0.002 rows=1 loops=880)
```

3. We chose to index metaCritic of playerData b/c it was suggested in the video to index components that are accessed often. After looking more into indexing analysis, the general rule of thumb is to index on joins and where clauses.
4. After running the analysis on GCP, we figured out the time for sorting in descending order reduced from 13.7 to 9.56. In addition, the cost to read for the first time where we use metaCritic reduced as well from 1.23 to 0.24 seconds as expected. The lookup cost using primary key reduced from 0.78 to 0.25 which is consistent with previous indexing.

Exploration 3: Indexing PlayerData(PlayersTotal)

1. Original performance of the query without any indexing


```

-> Sort: totalPlayers DESC (actual time=13.744..13.810 rows=880 loops=1)
-> Table scan on <temporary> (actual time=0.021..0.099 rows=880 loops=1)
-> Aggregate using temporary table (actual time=13.227..13.356 rows=880 loops=1)
-> Nested loop inner join (cost=1006.26 rows=469) (actual time=1.299..12.095 rows=880 loops=1)
-> Nested loop inner join (cost=834.38 rows=469) (actual time=1.286..9.114 rows=880 loops=1)
-> Filter: (p.playersTotal >= 4400) (cost=422.36 rows=469) (actual time=1.239..4.006 rows=880 loops=1)
-> Index range scan on p using another, with index condition: (p.metaCritic > 75) (cost=422.36 rows=938) (actual time=1.237..3.904 rows=938 loops=1)
-> Single-row index lookup on g using PRIMARY (gameName=p.gameName) (cost=0.78 rows=1) (actual time=0.006..0.006 rows=1 loops=880)
-> Single-row index lookup on c using PRIMARY (gameName=p.gameName) (cost=0.27 rows=1) (actual time=0.003..0.003 rows=1 loops=880)

```

2. Indexing on PlayerData(playersTotal) result

```

-> Sort: totalPlayers DESC (actual time=10.188..10.251 rows=880 loops=1)
-> Table scan on <temporary> (actual time=0.003..0.078 rows=880 loops=1)
-> Aggregate using temporary table (actual time=9.617..9.743 rows=880 loops=1)
-> Nested loop inner join (cost=1006.26 rows=469) (actual time=0.427..8.487 rows=880 loops=1)
-> Nested loop inner join (cost=834.38 rows=469) (actual time=0.419..5.600 rows=880 loops=1)
-> Filter: (p.playersTotal >= 4400) (cost=422.36 rows=469) (actual time=0.403..2.304 rows=880 loops=1)
-> Index range scan on p using another, with index condition: (p.metaCritic > 75) (cost=422.36 rows=938) (actual time=0.401..2.186 rows=938 loops=1)
-> Single-row index lookup on g using PRIMARY (gameName=p.gameName) (cost=0.78 rows=1) (actual time=0.005..0.005 rows=1 loops=880)
-> Single-row index lookup on c using PRIMARY (gameName=p.gameName) (cost=0.27 rows=1) (actual time=0.002..0.002 rows=1 loops=880)

```

3. We chose to create indices on playersTotal b/c we are constantly checking the condition for playersTotal when doing the check for totalPlayers. As stated before, it is good to have indices on values or rows that will be used for joins and are in where conditions.
4. After running the analysis on GCP, we found that filtering for playersTotal went from 1.239 seconds to 0.239 s. In addition, the time for joining also reduced.

Indexing analysis on second query: Finding friends

Exploration 1: Indexing primary keys

1. Original performance of the query without any indexing

```
-----+
| -> Sort: totalHoursPlayed DESC (actual time=108.271..108.272 rows=12 loops=1)
|   -> Table scan on <temporary> (actual time=0.003..0.005 rows=12 loops=1)
|     -> Aggregate using temporary table (actual time=108.216..108.219 rows=12 loops=1)
|       -> Hash semijoin (goa.playerId = go.playerId) (cost=1260114.12 rows=116046) (actual time=85.377..107.104 rows=1448 loops=1)
|         -> Nested loop inner join (cost=24690.60 rows=30662) (actual time=0.141..22.146 rows=2707 loops=1)
|           -> Nested loop inner join (cost=13958.90 rows=30662) (actual time=0.136..21.124 rows=2707 loops=1)
|             -> Filter: (go.playerId is not null) (cost=3227.20 rows=30662) (actual time=0.044..12.170 rows=32793 loops=1)
|               -> Table scan on go (cost=3227.20 rows=30662) (actual time=0.042..9.796 rows=32793 loops=1)
|                 -> Single-row index lookup on ud using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=0 loops=32793)
|                 -> Single-row index lookup on uda using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=2707)
|             -> Hash
|               -> Filter: (goa.gameName like 'Alien Swarm') (cost=2369.35 rows=341) (actual time=2.302..84.283 rows=104 loops=1)
|                 -> Table scan on goa (cost=2369.35 rows=30662) (actual time=2.295..80.723 rows=32793 loops=1)
```

2. Indexing on the the playerId from UserData result:

```
-----+
| -> Sort: totalHoursPlayed DESC (actual time=36.178..36.179 rows=12 loops=1)
|   -> Table scan on <temporary> (actual time=0.002..0.005 rows=12 loops=1)
|     -> Aggregate using temporary table (actual time=36.151..36.155 rows=12 loops=1)
|       -> Hash semijoin (goa.playerId = go.playerId) (cost=1251095.22 rows=116046) (actual time=12.972..34.922 rows=1448 loops=1)
|         -> Nested loop inner join (cost=24569.85 rows=30662) (actual time=0.140..22.350 rows=2707 loops=1)
|           -> Nested loop inner join (cost=13838.15 rows=30662) (actual time=0.135..21.351 rows=2707 loops=1)
|             -> Filter: (go.playerId is not null) (cost=3106.45 rows=30662) (actual time=0.041..12.547 rows=32793 loops=1)
|               -> Table scan on go (cost=3106.45 rows=30662) (actual time=0.039..9.951 rows=32793 loops=1)
|                 -> Single-row index lookup on ud using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=0 loops=32793)
|                 -> Single-row index lookup on uda using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=2707)
|             -> Hash
|               -> Filter: (goa.gameName like 'Alien Swarm') (cost=2270.25 rows=341) (actual time=0.067..12.066 rows=104 loops=1)
|                 -> Table scan on goa (cost=2270.25 rows=30662) (actual time=0.063..9.190 rows=32793 loops=1)
```

3. We decided to index playerId from UserData b/c it is used to join the two tables. In addition, it is also used in where clause to see if the particular playerId exists within the given database.
4. After running it on GCP we found, that joining cost reduced from 1260114 to 1251095 which is 10,000 less than previous cost. In addition, the table scan cost reduced by 100.

Exploration 2: Indexing GamesOwned(hoursPlayed)

1. Original performance of the query without any indexing

```
-----+
| -> Sort: totalHoursPlayed DESC (actual time=108.271..108.272 rows=12 loops=1)
|   -> Table scan on <temporary> (actual time=0.003..0.005 rows=12 loops=1)
|     -> Aggregate using temporary table (actual time=108.216..108.219 rows=12 loops=1)
|       -> Hash semijoin (goa.playerId = go.playerId) (cost=1260114.12 rows=116046) (actual time=85.377..107.104 rows=1448 loops=1)
|         -> Nested loop inner join (cost=24690.60 rows=30662) (actual time=0.141..22.146 rows=2707 loops=1)
|           -> Nested loop inner join (cost=13958.90 rows=30662) (actual time=0.136..21.124 rows=2707 loops=1)
|             -> Filter: (go.playerId is not null) (cost=3227.20 rows=30662) (actual time=0.044..12.170 rows=32793 loops=1)
|               -> Table scan on go (cost=3227.20 rows=30662) (actual time=0.042..9.796 rows=32793 loops=1)
|                 -> Single-row index lookup on ud using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=0 loops=32793)
|                 -> Single-row index lookup on uda using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=2707)
|             -> Hash
|               -> Filter: (goa.gameName like 'Alien Swarm') (cost=2369.35 rows=341) (actual time=2.302..84.283 rows=104 loops=1)
|                 -> Table scan on goa (cost=2369.35 rows=30662) (actual time=2.295..80.723 rows=32793 loops=1)
```

2. Indexing on the GamesOwned(hoursPlayed)

```
-----+
| -> Sort: totalHoursPlayed DESC (actual time=34.963..34.964 rows=12 loops=1)
|   -> Table scan on <temporary> (actual time=0.002..0.005 rows=12 loops=1)
|     -> Aggregate using temporary table (actual time=34.935..34.939 rows=12 loops=1)
|       -> Hash semijoin (goa.playerId = go.playerId) (cost=1251095.22 rows=116046) (actual time=12.889..33.770 rows=1448 loops=1)
|         -> Nested loop inner join (cost=24569.85 rows=30662) (actual time=0.174..21.286 rows=2707 loops=1)
|           -> Filter: (go.playerId is not null) (cost=3106.45 rows=30662) (actual time=0.045..11.755 rows=32793 loops=1)
|             -> Table scan on go (cost=3106.45 rows=30662) (actual time=0.043..9.485 rows=32793 loops=1)
|               -> Single-row index lookup on ud using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=0 loops=32793)
|               -> Single-row index lookup on uda using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=2707)
|             -> Hash
|               -> Filter: (goa.gameName like 'Alien Swarm') (cost=2270.25 rows=341) (actual time=0.054..11.972 rows=104 loops=1)
|                 -> Table scan on goa (cost=2270.25 rows=30662) (actual time=0.049..9.081 rows=32793 loops=1)
|
|
```

3. We chose to index on GamesOwned(hoursPlayed) b/c we are performing a sum on the hoursPlayed and we are also sorting (order by) the output based on the newly calculated totalHoursPlayed but I am not too sure how much impact that has.
4. After running it on GCP we found, that the cost for ordering the totalHoursPlayed reduced from 108.7 to 34.9 which is nearly 3 times less than the original cost without any indexing.

Exploration 3: Indexing gamesOwned(gameName)

1. Original performance of the query without any indexing

```
-----+
| -> Sort: totalHoursPlayed DESC (actual time=108.271..108.272 rows=12 loops=1)
|   -> Table scan on <temporary> (actual time=0.003..0.005 rows=12 loops=1)
|     -> Aggregate using temporary table (actual time=108.216..108.219 rows=12 loops=1)
|       -> Hash semijoin (goa.playerId = go.playerId) (cost=1260114.12 rows=116046) (actual time=85.377..107.104 rows=1448 loops=1)
|         -> Nested loop inner join (cost=24690.60 rows=30662) (actual time=0.141..22.146 rows=2707 loops=1)
|           -> Nested loop inner join (cost=13958.90 rows=30662) (actual time=0.136..21.124 rows=2707 loops=1)
|             -> Filter: (go.playerId is not null) (cost=3227.20 rows=30662) (actual time=0.044..12.170 rows=32793 loops=1)
|               -> Table scan on go (cost=3227.20 rows=30662) (actual time=0.042..9.796 rows=32793 loops=1)
|                 -> Single-row index lookup on ud using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=0 loops=32793)
|                 -> Single-row index lookup on uda using PRIMARY (playerId=go.playerId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=2707)
|             -> Hash
|               -> Filter: (goa.gameName like 'Alien Swarm') (cost=2369.35 rows=341) (actual time=2.302..84.283 rows=104 loops=1)
|                 -> Table scan on goa (cost=2369.35 rows=30662) (actual time=2.295..80.723 rows=32793 loops=1)
|
|
```

2. Indexing on the gamesOwned(gameName)

```
-----+
| -> Sort: totalHoursPlayed DESC (actual time=15.050..15.051 rows=12 loops=1)
|   -> Table scan on <temporary> (actual time=0.002..0.004 rows=12 loops=1)
|     -> Aggregate using temporary table (actual time=15.023..15.026 rows=12 loops=1)
|       -> Remove duplicate (ud, uda, go) rows using temporary table (weedout) (cost=319054.78 rows=318885) (actual time=1.228..13.921 rows=1448 loops=1)
|         -> Inner hash join (goa.playerId = goa.playerId) (cost=319054.78 rows=318885) (actual time=1.221..13.347 rows=1448 loops=1)
|           -> Table scan on go (cost=3.43 rows=30662) (actual time=0.044..9.107 rows=32793 loops=1)
|             -> Hash
|               -> Nested loop inner join (cost=119.86 rows=104) (actual time=0.420..0.616 rows=12 loops=1)
|                 -> Nested loop inner join (cost=83.46 rows=104) (actual time=0.417..0.595 rows=12 loops=1)
|                   -> Filter: (goa.playerId is not null) (cost=47.06 rows=104) (actual time=0.396..0.437 rows=104 loops=1)
|                     -> Index range scan on goa using i, with index condition: (goa.gameName like 'Alien Swarm') (cost=47.06 rows=104) (actual time=0.394..0.428 rows=104 loops=1)
|                   -> Single-row index lookup on ud using PRIMARY (playerId=goa.playerId) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=104)
|                   -> Single-row index lookup on uda using PRIMARY (playerId=goa.playerId) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=12)
|
|
```

3. We chose to index the gameName b/c it is used in a where clause and its constantly compared against the user input which in our case is "Alien Swarm"
4. After running it on GCP we found, that the cost of checking the user input gameName to existing gameNames went from 2369.35 to 47 and the actual time to read first row only went from 2.3 to 0.394 which is what we had expected.