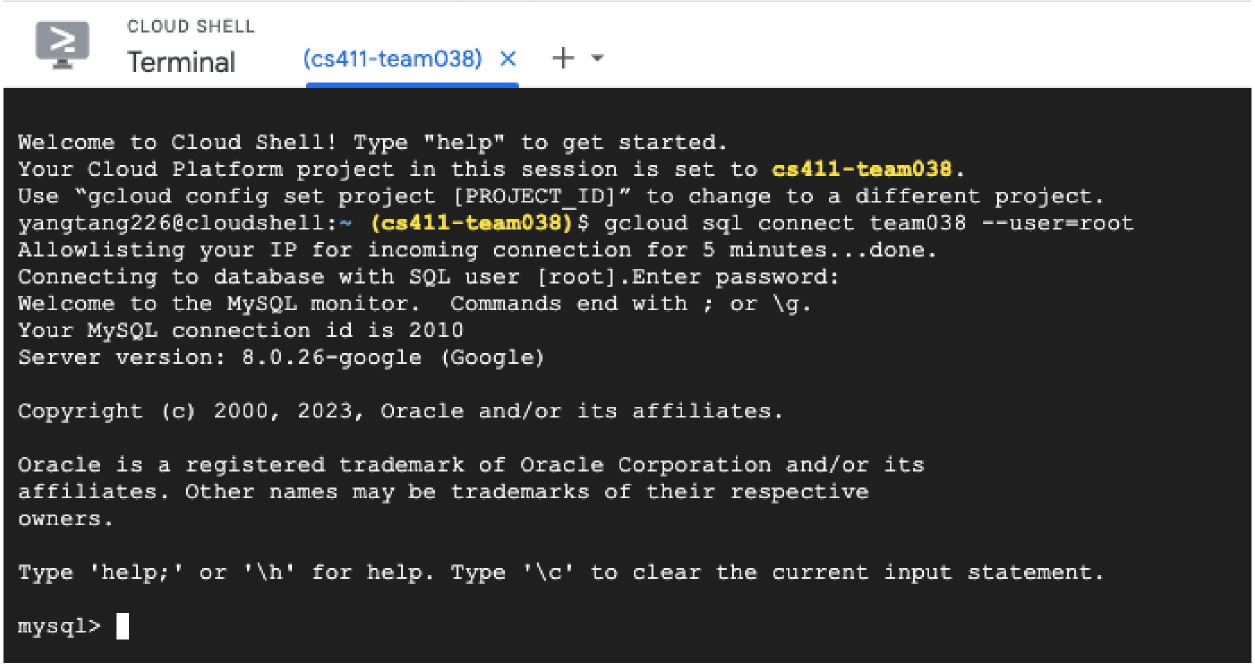
NetIds (Names):
jechang3 (Jacob Chang)
yangt2 (Yang Tang)
eapen2 (Sethu Eapen)
mmasse7 (Marvin Massey)

Stage 3: Database Implementation and Indexing

GCP SQL Instance Connection



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to cs411-team038.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
yangtang226@cloudshell:~ (cs411-team038)$ gcloud sql connect team038 --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 2010
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.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```

Implemented Tables

DDL commands to create tables:

```
CREATE TABLE pending_games (
   request_id INT,
   user_id INT,
   name VARCHAR(255),
   website VARCHAR(255),
   PRIMARY KEY (request_id)
);
```

```
CREATE TABLE genres (
   genre varchar(255),
   LastName varchar(255),
   most_popular_game_id int,
   second_popular_game_id int,
   third_popular_game_id int,
   PRIMARY KEY (genre),
   FOREIGN KEY (most_popular_game_id) REFERENCES games(game_id),
   FOREIGN KEY (second_popular_game_id) REFERENCES games(game_id),
   FOREIGN KEY (third_popular_game_id) REFERENCES games(game_id)
);

CREATE TABLE friends (
   user_id_one INT,
   user_id_two INT,
   PRIMARY KEY (user_id_one, user_id_two)
);


CREATE TABLE owns (
   user_id INT,
   game_id INT,
   play_time INT,
   PRIMARY KEY (user_id, game_id),
   FOREIGN KEY (user_id) REFERENCES users(user_id),
   FOREIGN KEY (game_id) REFERENCES games(game_id)
);

CREATE TABLE compatible (
        game_id INT,
        platform VARCHAR(255)
);

CREATE TABLE games (
        game_id INT NOT NULL auto_increment,
        name VARCHAR(255) NOT NULL,
        genre VARCHAR(255) NOT NULL,
        description VARCHAR(600) NOT NULL,
        release_date DATE NOT NULL,
        image_link VARCHAR(255),
        game_link VARCHAR(255),
        popularity INT NOT NULL,
        PRIMARY KEY (game_id)
);
```

```
CREATE TABLE computers (
	 computer_id INT NOT NULL auto_increment,
	platform VARCHAR(255) NOT NULL,
	user_id INT NOT NULL,
	PRIMARY KEY (computer_id),
	FOREIGN KEY (user_id) references users(user_id)
);

CREATE TABLE users(
	user_id INT NOT NULL AUTO_INCREMENT,
	username VARCHAR(255),
	hashed_password VARCHAR(255),
	email_address VARCHAR(255),
	is_admin BOOLEAN,
	PRIMARY KEY (user_id)
)

CREATE TABLE genres (
	game_id INT NOT NULL,
	genre VARCHAR(255) NOT NULL,
	PRIMARY KEY(game_id, genre),
	FOREIGN KEY (game_id) REFERENCES games(game_id)
);
```

Commands for adding data to tables:

games table:

```
INSERT INTO games (name, description, release_date, image_link, game_link, popularity)
SELECT DISTINCT QueryName AS name, DetailedDescrip AS description, curdate(),
HeaderImage as image_link, Website as game_link, RecommendationCount as popularity
FROM games_features;
```

Inserting data from a CSV file into a table (we used this for many of the tables).

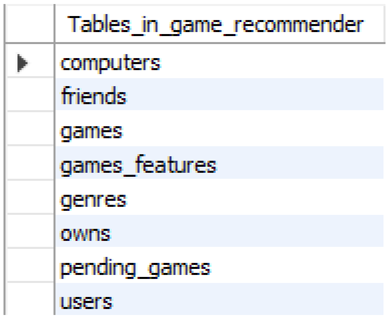
```
LOAD DATA LOCAL INFILE 'C:\\Users\\jacobchang124\\Downloads\\[file_name].csv'
INTO TABLE [table_name]
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

Stored Procedure for populating the genres table:

```
DELIMITER //

CREATE PROCEDURE generate_genre()
BEGIN
  DECLARE game_id_var VARCHAR(255);
  DECLARE game_name_var VARCHAR(255);

  DECLARE exit_loop BOOLEAN DEFAULT FALSE;

  DECLARE custCur CURSOR FOR (SELECT game_id, name FROM games);
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET exit_loop = TRUE;

  OPEN custCur;
    cloop: LOOP
    FETCH custCur INTO game_id_var, game_name_var;
    IF exit_loop THEN
      LEAVE cloop;
    END IF;

    SELECT GenreISAction, GenreIsAdventure, GenreIsCasual, GenreIsStrategy, GenreISRPG,
GenreIsSimulation, GenreIsSports, GenreISRacing
    INTO @isAction, @isAdventure, @isCasual, @isStrategy, @isRPG, @isSimulation,
@isSports, @isRacing
    FROM games_features WHERE QueryName = game_name_var LIMIT 1;

    IF TRIM(@isAction) = 'True' THEN
      INSERT INTO genres VALUES (game_id_var, 'Action');
    END IF;

    IF TRIM(@isAdventure) = 'True' THEN
      INSERT INTO genres VALUES (game_id_var, 'Adventure');
    END IF;

    IF TRIM(@isCasual) = 'True' THEN
      INSERT INTO genres VALUES (game_id_var, 'Casual');
    END IF;

    IF TRIM(@isStrategy) = 'True' THEN
      INSERT INTO genres VALUES (game_id_var, 'Strategy');
    END IF;

    IF TRIM(@isRPG) = 'True' THEN
      INSERT INTO genres VALUES (game_id_var, 'RPG');
    END IF;

    IF TRIM(@isSimulation) = 'True' THEN
      INSERT INTO genres VALUES (game_id_var, 'Simulation');
    END IF;

    IF TRIM(@isSports) = 'True' THEN
      INSERT INTO genres VALUES (game_id_var, 'Sports');
    END IF;

    IF TRIM(@isRacing) = 'True' THEN
      INSERT INTO genres VALUES (game_id_var, 'Racing');
    END IF;

    END LOOP cloop;
    CLOSE custCur;

END;

DELIMITER ;

CALL generate_genre();
```

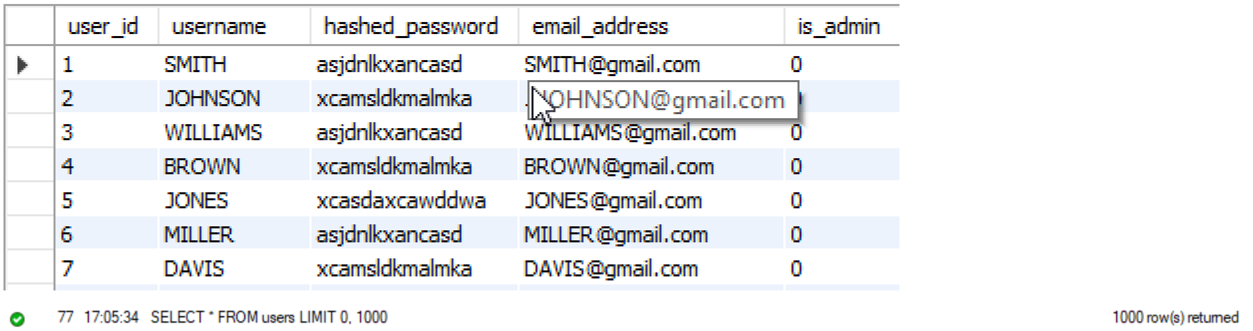
Screenshots of the data having been inserted into the tables:

Database layout (all the tables available):



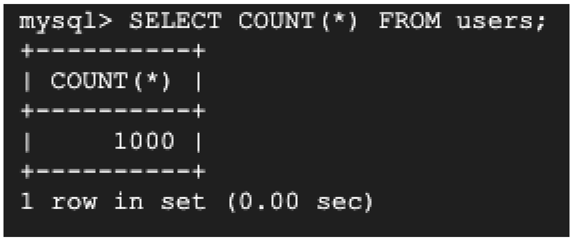
Tables_in_game_recommender
computers
friends
games
games_features
genres
owns
pending_games
users

Users table:



	user_id	username	hashed_password	email_address	is_admin
▶	1	SMITH	asjdnlkxancasd	SMITH@gmail.com	0
	2	JOHNSON	xcamsldkmalmka	JOHNSON@gmail.com	0
	3	WILLIAMS	asjdnlkxancasd	WILLIAMS@gmail.com	0
	4	BROWN	xcamsldkmalmka	BROWN@gmail.com	0
	5	JONES	xcasdaxcawddwa	JONES@gmail.com	0
	6	MILLER	asjdnlkxancasd	MILLER@gmail.com	0
	7	DAVIS	xcamsldkmalmka	DAVIS@gmail.com	0

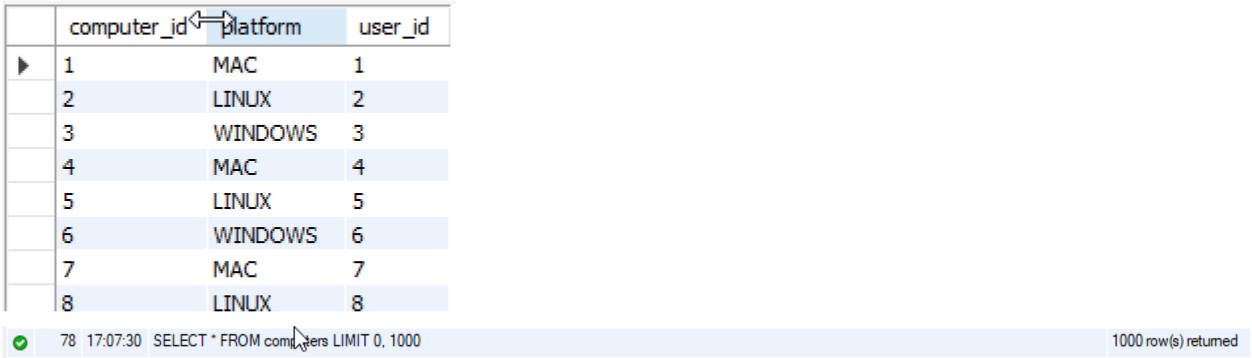
```
77  17:05:34  SELECT * FROM users LIMIT 0, 1000                    1000 row(s) returned
```



```
mysql> SELECT COUNT(*) FROM users;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)
```

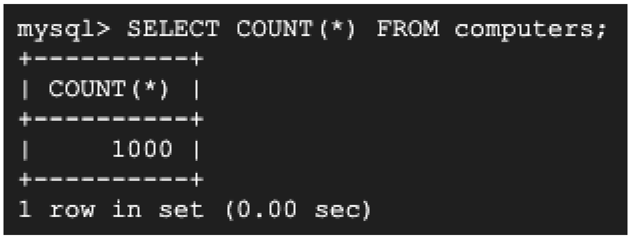
There are 1000 rows in users table.

Computers table:



	computer_id	platform	user_id
▶	1	MAC	1
	2	LINUX	2
	3	WINDOWS	3
	4	MAC	4
	5	LINUX	5
	6	WINDOWS	6
	7	MAC	7
	8	LINUX	8

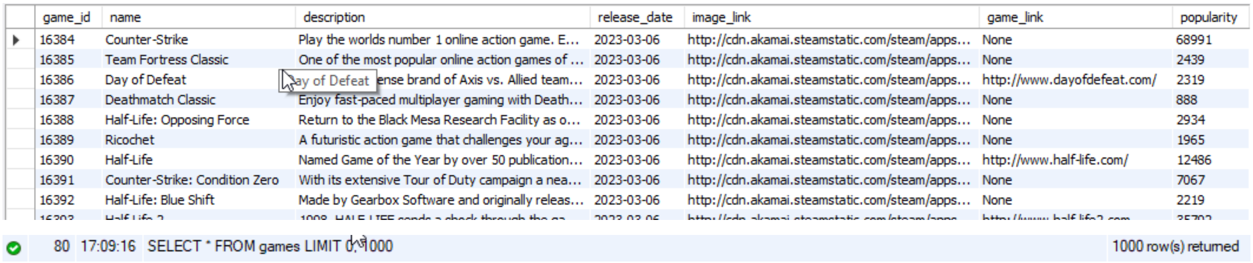
```
78  17:07:30  SELECT * FROM computers LIMIT 0, 1000                1000 row(s) returned
```



```
mysql> SELECT COUNT(*) FROM computers;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)
```

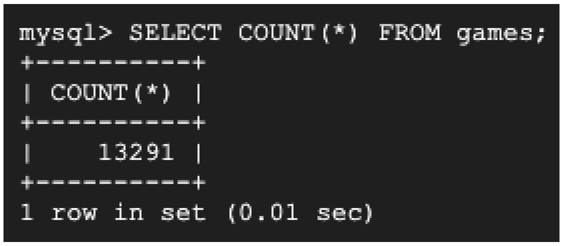
There are 1000 rows in computers table.

Games:



	game_id	name	description	release_date	image_link	game_link	popularity
▶	16384	Counter-Strike	Play the worlds number 1 online action game. E…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	None	68991
	16385	Team Fortress Classic	One of the most popular online action games of …	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	None	2439
	16386	Day of Defeat	Day of Defeat ense brand of Axis vs. Allied team…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	http://www.dayofdefeat.com/	2319
	16387	Deathmatch Classic	Enjoy fast-paced multiplayer gaming with Death…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	None	888
	16388	Half-Life: Opposing Force	Return to the Black Mesa Research Facility as o…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	None	2934
	16389	Ricochet	A futuristic action game that challenges your ag…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	None	1965
	16390	Half-Life	Named Game of the Year by over 50 publication…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	http://www.half-life.com/	12486
	16391	Counter-Strike: Condition Zero	With its extensive Tour of Duty campaign a nea…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	None	7067
	16392	Half-Life: Blue Shift	Made by Gearbox Software and originally releas…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	None	2219
	16393	Half-Life 2	1998. HALF-LIFE sends a shock through the ga…	2023-03-06	http://cdn.akamai.steamstatic.com/steam/apps…	http://www.half-life2.com	35792

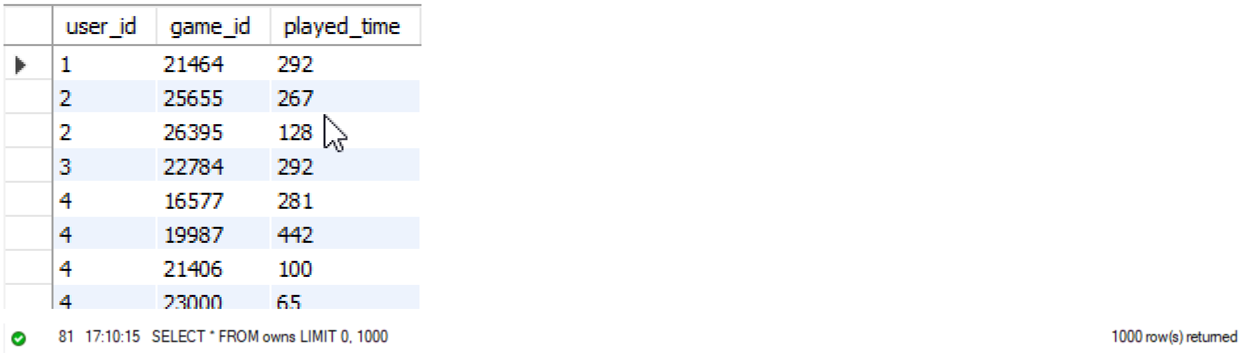
```
80  17:09:16  SELECT * FROM games LIMIT 0, 1000                    1000 row(s) returned
```



```
mysql> SELECT COUNT(*) FROM games;
+----------+
| COUNT(*) |
+----------+
|    13291 |
+----------+
1 row in set (0.01 sec)
```

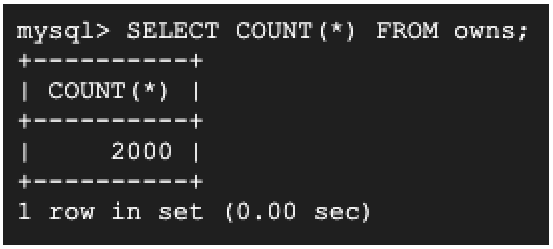
There are 13291 rows in games table.

Owns:



user_id	game_id	played_time
1	21464	292
2	25655	267
2	26395	128
3	22784	292
4	16577	281
4	19987	442
4	21406	100
4	23000	65

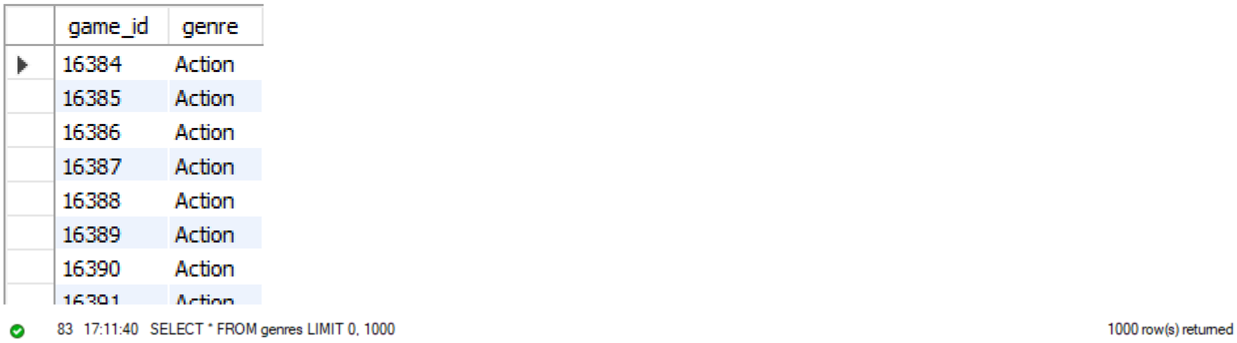
```
81  17:10:15  SELECT * FROM owns LIMIT 0, 1000          1000 row(s) returned
```



```
mysql> SELECT COUNT(*) FROM owns;
+----------+
| COUNT(*) |
+----------+
|     2000 |
+----------+
1 row in set (0.00 sec)
```

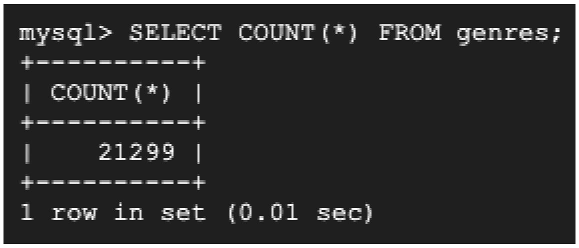
There are 2000 rows in owns table.

Genres:



game_id	genre
16384	Action
16385	Action
16386	Action
16387	Action
16388	Action
16389	Action
16390	Action
16391	Action

```
83  17:11:40  SELECT * FROM genres LIMIT 0, 1000          1000 row(s) returned
```



```
mysql> SELECT COUNT(*) FROM genres;
+----------+
| COUNT(*) |
+----------+
|    21299 |
+----------+
1 row in set (0.01 sec)
```

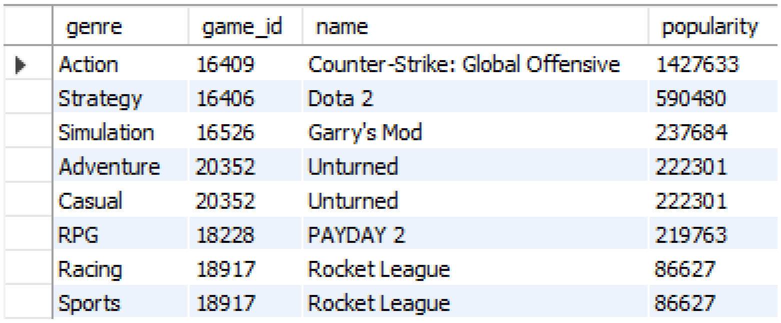
There are 21299 rows in genres table.

Advanced Queries

Ranking the most popular genres (JOIN and GROUP BY and Subqueries):

```
SELECT gs.genre, g1.game_id, g1.name, g1.popularity
FROM games as g1 JOIN genres as gs ON (g1.game_id = gs.game_id)
WHERE (gs.genre, g1.popularity) IN (SELECT g.genre, MAX(ga.popularity)
                FROM games as ga JOIN genres as g ON (ga.game_id = g.game_id)
                GROUP BY g.genre
)
ORDER BY g1.popularity DESC
```

Image of full output:



genre	game_id	name	popularity
Action	16409	Counter-Strike: Global Offensive	1427633
Strategy	16406	Dota 2	590480
Simulation	16526	Garry's Mod	237684
Adventure	20352	Unturned	222301
Casual	20352	Unturned	222301
RPG	18228	PAYDAY 2	219763
Racing	18917	Rocket League	86627
Sports	18917	Rocket League	86627

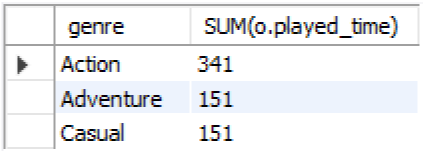
Since there are only 8 genres of all games, thus we have 8 rows returned.
We will use this query when recommending games to users. Once we find out their favorite genre then we can recommend them the most popular game within that genre.

Finding the play time per genre per selected user (JOIN and GROUP BY):

```
SELECT g.genre, SUM(o.played_time)
```

```
FROM owns as o JOIN genres as g ON (o.game_id = g.game_id)
WHERE o.user_id = 144
GROUP BY g.genre
ORDER BY SUM(o.played_time) DESC
```

Image of full output for user 144:



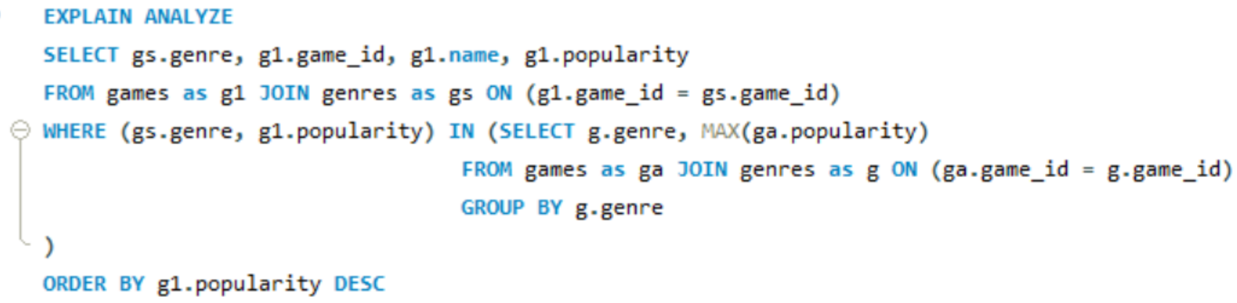
genre	SUM(o.played_time)
Action	341
Adventure	151
Casual	151

Since in our database the user with id 144 only plays 3 genres of game, thus we have 3 rows returned.
This query will be used for users of our platform with usable game data so that we can find the genre which they like the most.

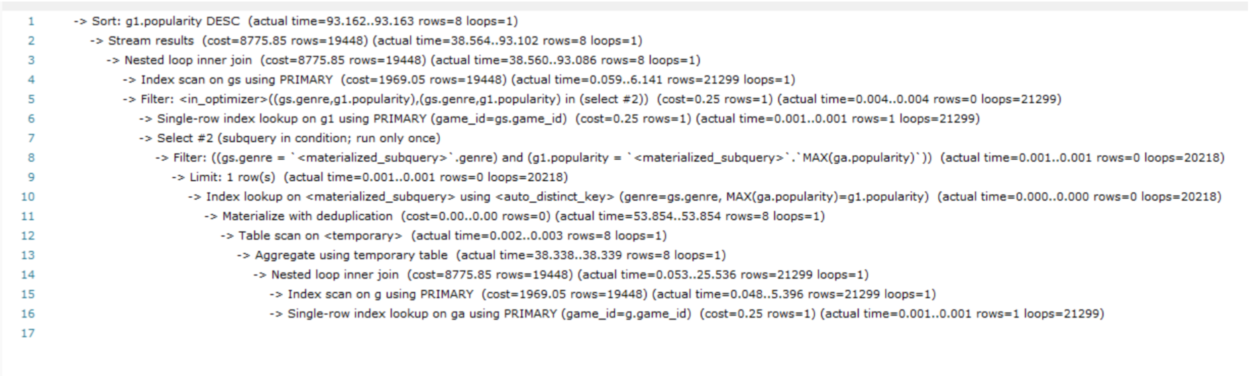
Indexing

FIRST QUERY

Before adding any popularity (on table games) index:

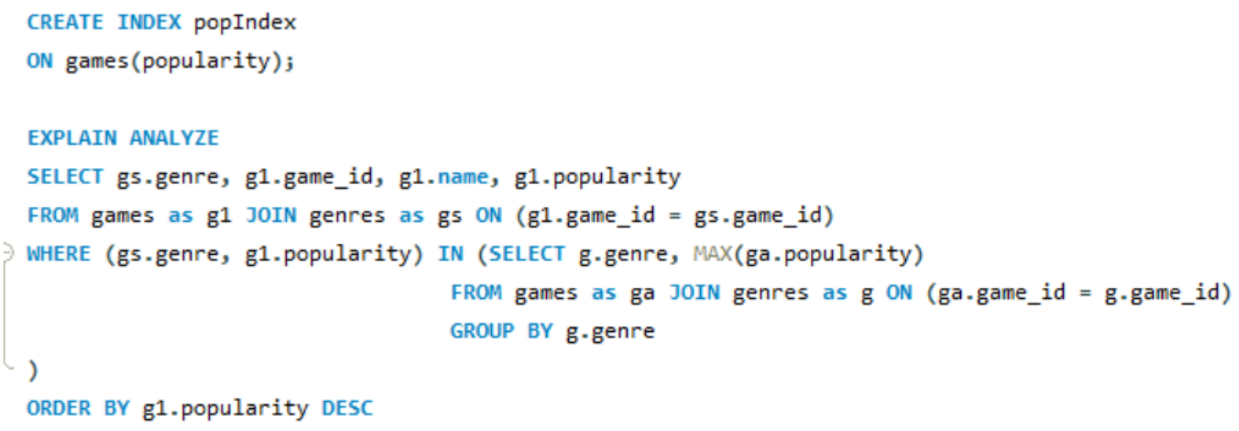


```
EXPLAIN ANALYZE
SELECT gs.genre, g1.game_id, g1.name, g1.popularity
FROM games as g1 JOIN genres as gs ON (g1.game_id = gs.game_id)
WHERE (gs.genre, g1.popularity) IN (SELECT g.genre, MAX(ga.popularity)
                                    FROM games as ga JOIN genres as g ON (ga.game_id = g.game_id)
                                    GROUP BY g.genre
)
ORDER BY g1.popularity DESC
```

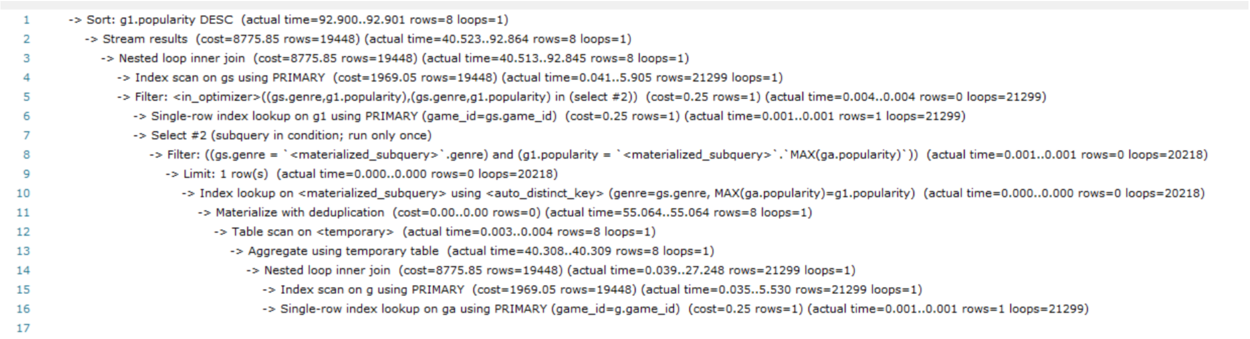


```
-> Sort: g1.popularity DESC  (actual time=93.162..93.163 rows=8 loops=1)
    -> Stream results  (cost=8775.85 rows=19448) (actual time=38.564..93.102 rows=8 loops=1)
        -> Nested loop inner join  (cost=8775.85 rows=19448) (actual time=38.560..93.086 rows=8 loops=1)
            -> Index scan on gs using PRIMARY  (cost=1969.05 rows=19448) (actual time=0.059..6.141 rows=21299 loops=1)
            -> Filter: <in_optimizer>((gs.genre,g1.popularity),(gs.genre,g1.popularity) in (select #2))  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=0 loops=21299)
                -> Single-row index lookup on g1 using PRIMARY (game_id=gs.game_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=21299)
                -> Select #2 (subquery in condition; run only once)
                    -> Filter: ((gs.genre = `<materialized_subquery>`.genre) and (g1.popularity = `<materialized_subquery>`.`MAX(ga.popularity)`))  (actual time=0.001..0.001 rows=0 loops=20218)
                        -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=20218)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (genre=gs.genre, MAX(ga.popularity)=g1.popularity)  (actual time=0.000..0.000 rows=0 loops=20218)
                                -> Materialize with deduplication  (cost=0.00..0.00 rows=0) (actual time=53.854..53.854 rows=8 loops=1)
                                    -> Table scan on <temporary>  (actual time=0.002..0.003 rows=8 loops=1)
                                        -> Aggregate using temporary table  (actual time=38.338..38.339 rows=8 loops=1)
                                            -> Nested loop inner join  (cost=8775.85 rows=19448) (actual time=0.053..25.536 rows=21299 loops=1)
                                                -> Index scan on g using PRIMARY  (cost=1969.05 rows=19448) (actual time=0.048..5.396 rows=21299 loops=1)
                                                -> Single-row index lookup on ga using PRIMARY (game_id=g.game_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=21299)
```

After adding popularity index on table games:



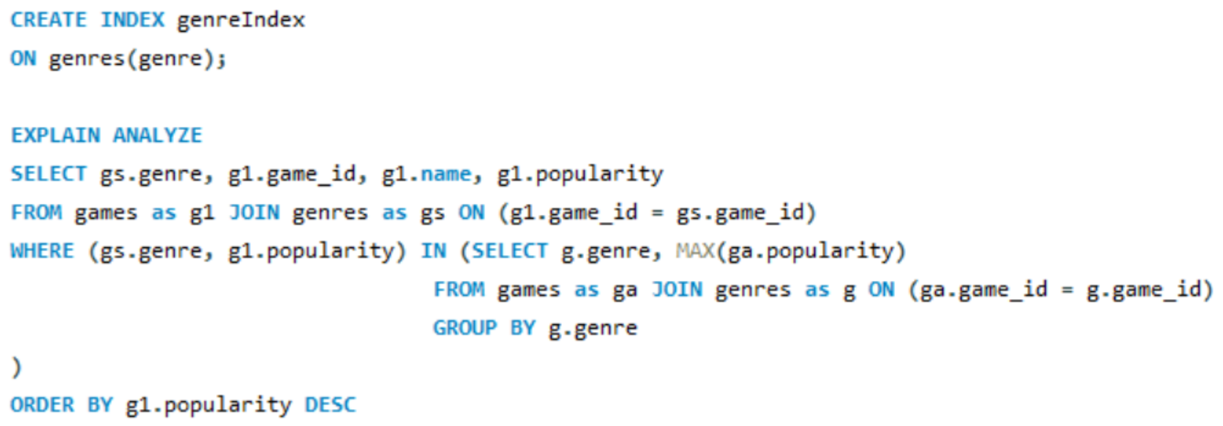
```sql
CREATE INDEX popIndex
ON games(popularity);

EXPLAIN ANALYZE
SELECT gs.genre, g1.game_id, g1.name, g1.popularity
FROM games as g1 JOIN genres as gs ON (g1.game_id = gs.game_id)
WHERE (gs.genre, g1.popularity) IN (SELECT g.genre, MAX(ga.popularity)
                                    FROM games as ga JOIN genres as g ON (ga.game_id = g.game_id)
                                    GROUP BY g.genre
)
ORDER BY g1.popularity DESC
```



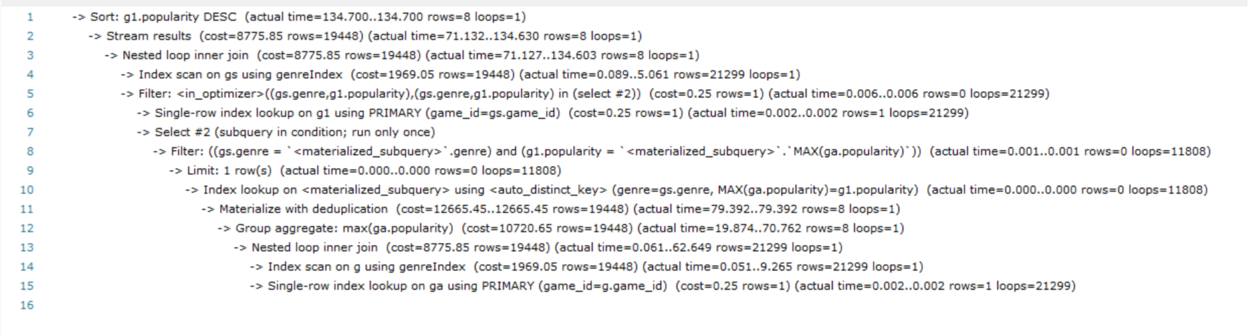
```
-> Sort: g1.popularity DESC  (actual time=92.900..92.901 rows=8 loops=1)
    -> Stream results  (cost=8775.85 rows=19448) (actual time=40.523..92.864 rows=8 loops=1)
        -> Nested loop inner join  (cost=8775.85 rows=19448) (actual time=40.513..92.845 rows=8 loops=1)
            -> Index scan on gs using PRIMARY  (cost=1969.05 rows=19448) (actual time=0.041..5.905 rows=21299 loops=1)
            -> Filter: <in_optimizer>((gs.genre,g1.popularity),(gs.genre,g1.popularity) in (select #2))  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=0 loops=21299)
                -> Single-row index lookup on g1 using PRIMARY (game_id=gs.game_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=21299)
                -> Select #2 (subquery in condition; run only once)
                    -> Filter: ((gs.genre = `<materialized_subquery>`.genre) and (g1.popularity = `<materialized_subquery>`.`MAX(ga.popularity)`))  (actual time=0.001..0.001 rows=0 loops=20218)
                        -> Limit: 1 row(s)  (actual time=0.000..0.000 rows=0 loops=20218)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (genre=gs.genre, MAX(ga.popularity)=g1.popularity)  (actual time=0.000..0.000 rows=0 loops=20218)
                                -> Materialize with deduplication  (cost=0.00..0.00 rows=0) (actual time=55.064..55.064 rows=8 loops=1)
                                    -> Table scan on <temporary>  (actual time=0.003..0.004 rows=8 loops=1)
                                        -> Aggregate using temporary table  (actual time=40.308..40.309 rows=8 loops=1)
                                            -> Nested loop inner join  (cost=8775.85 rows=19448) (actual time=0.039..27.248 rows=21299 loops=1)
                                                -> Index scan on g using PRIMARY  (cost=1969.05 rows=19448) (actual time=0.035..5.530 rows=21299 loops=1)
                                                -> Single-row index lookup on ga using PRIMARY (game_id=g.game_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=21299)
```

Then we deleted the popularity index, and added genre index on table genres.

After adding genre index on table genres:



```sql
CREATE INDEX genreIndex
ON genres(genre);

EXPLAIN ANALYZE
SELECT gs.genre, g1.game_id, g1.name, g1.popularity
FROM games as g1 JOIN genres as gs ON (g1.game_id = gs.game_id)
WHERE (gs.genre, g1.popularity) IN (SELECT g.genre, MAX(ga.popularity)
                                    FROM games as ga JOIN genres as g ON (ga.game_id = g.game_id)
                                    GROUP BY g.genre
)
ORDER BY g1.popularity DESC
```

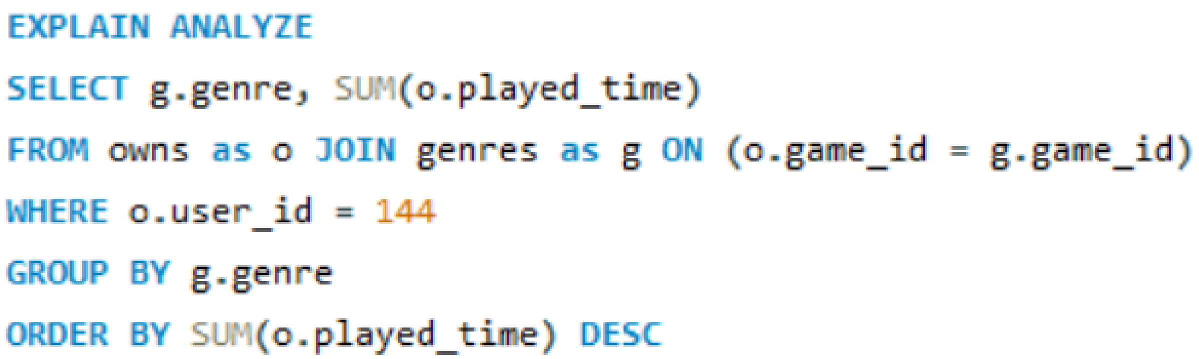


```
-> Sort: g1.popularity DESC  (actual time=134.700..134.700 rows=8 loops=1)
    -> Stream results  (cost=8775.85 rows=19448) (actual time=71.132..134.630 rows=8 loops=1)
        -> Nested loop inner join  (cost=8775.85 rows=19448) (actual time=71.127..134.603 rows=8 loops=1)
            -> Index scan on gs using genreIndex  (cost=1969.05 rows=19448) (actual time=0.089..5.061 rows=21299 loops=1)
            -> Filter: <in_optimizer>((gs.genre,g1.popularity),(gs.genre,g1.popularity) in (select #2))  (cost=0.25 rows=1) (actual time=0.006..0.006 rows=0 loops=21299)
                -> Single-row index lookup on g1 using PRIMARY (game_id=gs.game_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=21299)
                -> Select #2 (subquery in condition; run only once)
                    -> Filter: ((gs.genre = `<materialized_subquery>`.genre) and (g1.popularity = `<materialized_subquery>`.`MAX(ga.popularity)`))  (actual time=0.001..0.001 rows=0 loops=11808)
                        -> Limit: 1 row(s)  (actual time=0.000..0.000 rows=0 loops=11808)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (genre=gs.genre, MAX(ga.popularity)=g1.popularity)  (actual time=0.000..0.000 rows=0 loops=11808)
                                -> Materialize with deduplication  (cost=12665.45..12665.45 rows=19448) (actual time=79.392..79.392 rows=8 loops=1)
                                    -> Group aggregate: max(ga.popularity)  (cost=10720.65 rows=19448) (actual time=19.874..70.762 rows=8 loops=1)
                                        -> Nested loop inner join  (cost=8775.85 rows=19448) (actual time=0.061..62.649 rows=21299 loops=1)
                                            -> Index scan on g using genreIndex  (cost=1969.05 rows=19448) (actual time=0.051..9.265 rows=21299 loops=1)
                                            -> Single-row index lookup on ga using PRIMARY (game_id=g.game_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=21299)
```

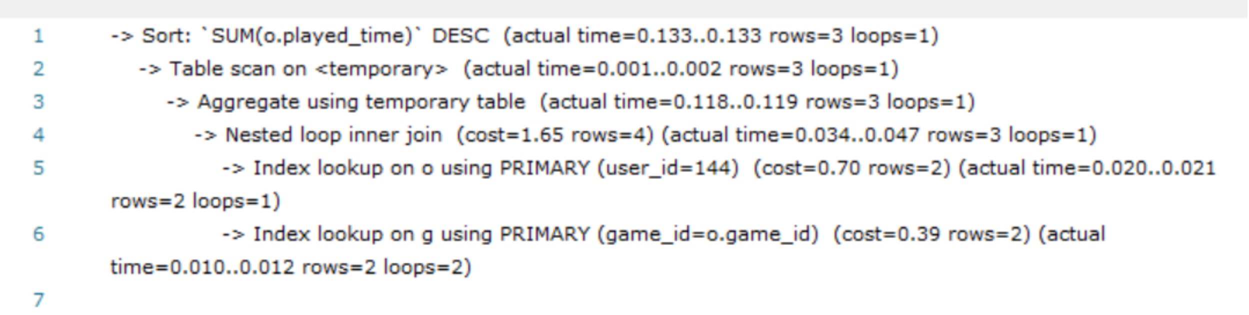
Analysis: After adding the popularity index, the cost did not change compared to the results where no index is added, but the actual time decreases. And after adding the genre index, the cost did not change either, but the actual time increased. The reason behind the increased actual time may result from the fact that popularity is not used in range queries of WHERE clause, which is one of the main advantages of B+ tree index, and the use of index adds overhead under the hood. Therefore, we decided to choose to index on the popularity column on table games.

SECOND QUERY:

Before any indexes:

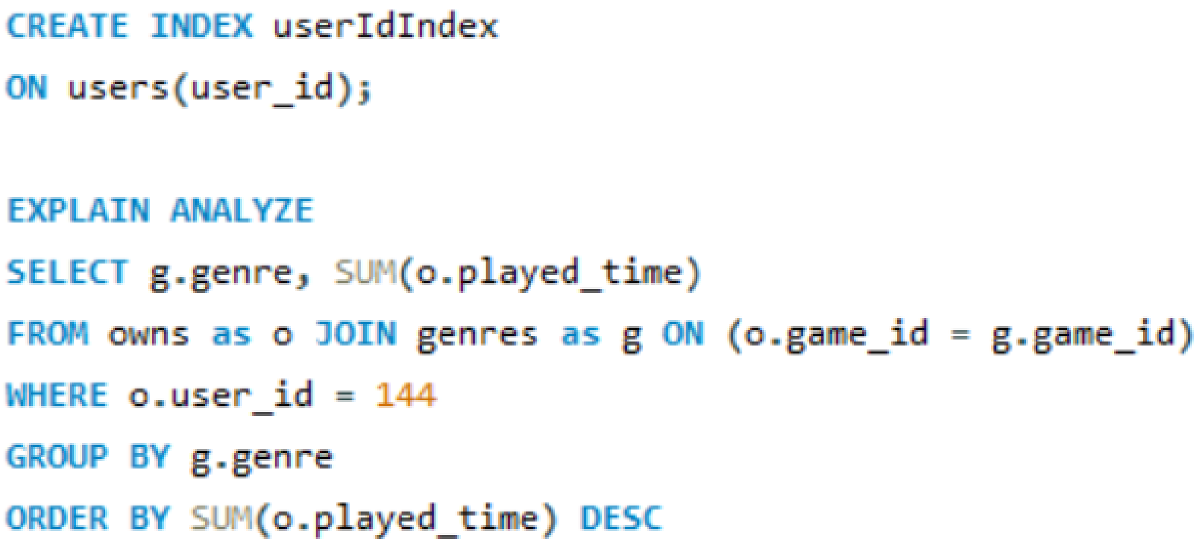


```
EXPLAIN ANALYZE
SELECT g.genre, SUM(o.played_time)
FROM owns as o JOIN genres as g ON (o.game_id = g.game_id)
WHERE o.user_id = 144
GROUP BY g.genre
ORDER BY SUM(o.played_time) DESC
```

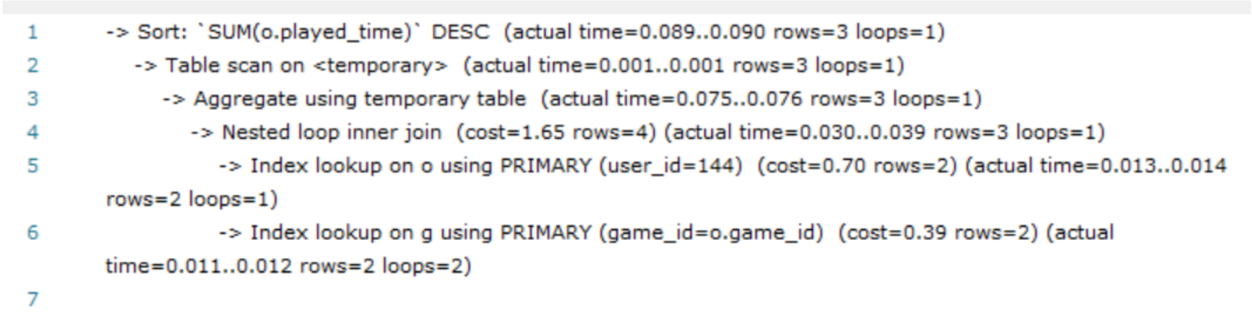


```
-> Sort: `SUM(o.played_time)` DESC  (actual time=0.133..0.133 rows=3 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.002 rows=3 loops=1)
        -> Aggregate using temporary table  (actual time=0.118..0.119 rows=3 loops=1)
            -> Nested loop inner join  (cost=1.65 rows=4) (actual time=0.034..0.047 rows=3 loops=1)
                -> Index lookup on o using PRIMARY (user_id=144)  (cost=0.70 rows=2) (actual time=0.020..0.021 rows=2 loops=1)
                -> Index lookup on g using PRIMARY (game_id=o.game_id)  (cost=0.39 rows=2) (actual time=0.010..0.012 rows=2 loops=2)
```

After adding an index of user_id of the users table:

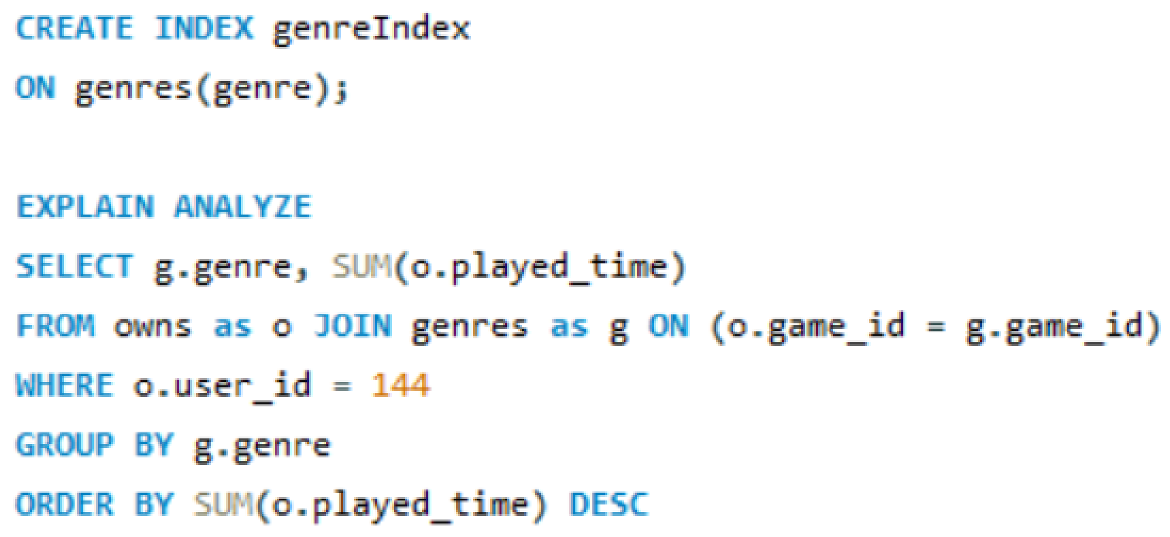


```sql
CREATE INDEX userIdIndex
ON users(user_id);

EXPLAIN ANALYZE
SELECT g.genre, SUM(o.played_time)
FROM owns as o JOIN genres as g ON (o.game_id = g.game_id)
WHERE o.user_id = 144
GROUP BY g.genre
ORDER BY SUM(o.played_time) DESC
```

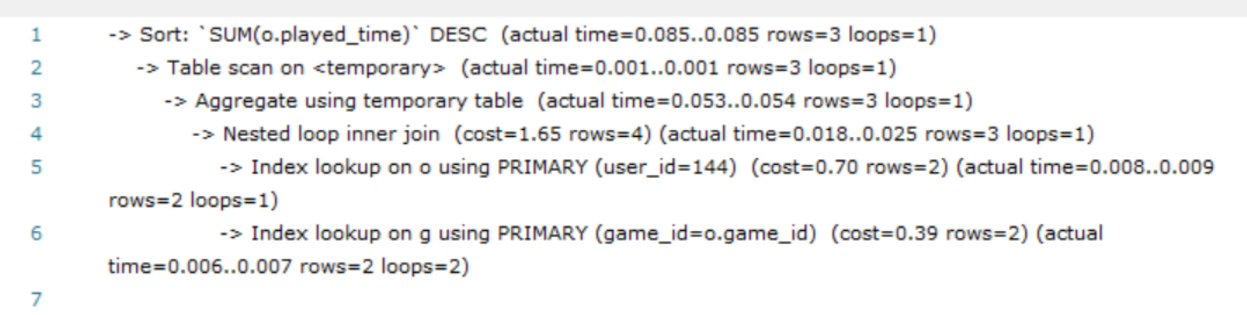


```
-> Sort: `SUM(o.played_time)` DESC  (actual time=0.089..0.090 rows=3 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.001 rows=3 loops=1)
        -> Aggregate using temporary table  (actual time=0.075..0.076 rows=3 loops=1)
            -> Nested loop inner join  (cost=1.65 rows=4) (actual time=0.030..0.039 rows=3 loops=1)
                -> Index lookup on o using PRIMARY (user_id=144)  (cost=0.70 rows=2) (actual time=0.013..0.014 rows=2 loops=1)
                -> Index lookup on g using PRIMARY (game_id=o.game_id)  (cost=0.39 rows=2) (actual time=0.011..0.012 rows=2 loops=2)
```

We now remove the index on user_id of the users table, and add an index for the genre column of the genres table, for a comparison.

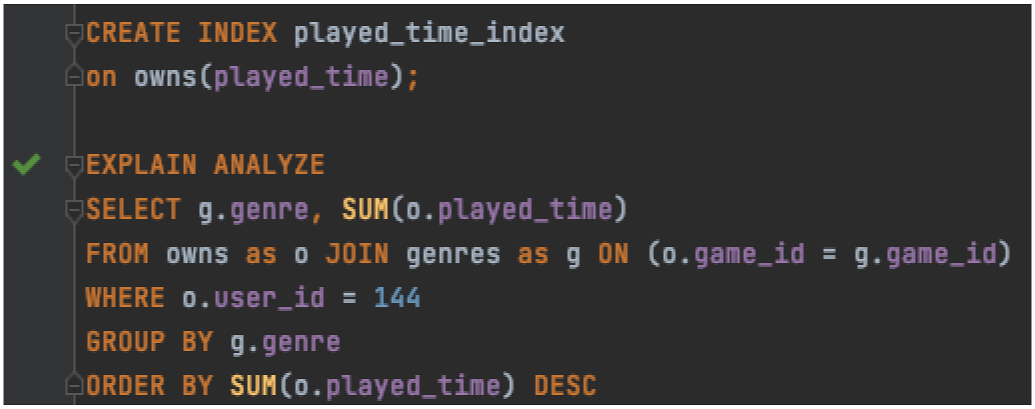


```sql
CREATE INDEX genreIndex
ON genres(genre);

EXPLAIN ANALYZE
SELECT g.genre, SUM(o.played_time)
FROM owns as o JOIN genres as g ON (o.game_id = g.game_id)
WHERE o.user_id = 144
GROUP BY g.genre
ORDER BY SUM(o.played_time) DESC
```

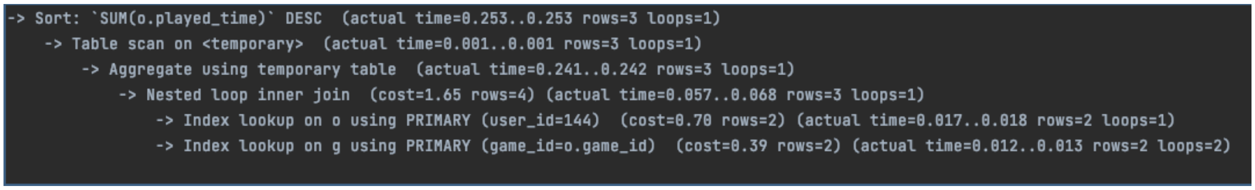


```
-> Sort: `SUM(o.played_time)` DESC  (actual time=0.085..0.085 rows=3 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.001 rows=3 loops=1)
        -> Aggregate using temporary table  (actual time=0.053..0.054 rows=3 loops=1)
            -> Nested loop inner join  (cost=1.65 rows=4) (actual time=0.018..0.025 rows=3 loops=1)
                -> Index lookup on o using PRIMARY (user_id=144)  (cost=0.70 rows=2) (actual time=0.008..0.009 rows=2 loops=1)
                -> Index lookup on g using PRIMARY (game_id=o.game_id)  (cost=0.39 rows=2) (actual time=0.006..0.007 rows=2 loops=2)
```

We now remove the index on genre of genres table, and add an index for the play_time column of the own table, for another comparison.



```sql
CREATE INDEX played_time_index
on owns(played_time);

EXPLAIN ANALYZE
SELECT g.genre, SUM(o.played_time)
FROM owns as o JOIN genres as g ON (o.game_id = g.game_id)
WHERE o.user_id = 144
GROUP BY g.genre
ORDER BY SUM(o.played_time) DESC
```



```
-> Sort: `SUM(o.played_time)` DESC  (actual time=0.253..0.253 rows=3 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.001 rows=3 loops=1)
        -> Aggregate using temporary table  (actual time=0.241..0.242 rows=3 loops=1)
            -> Nested loop inner join  (cost=1.65 rows=4) (actual time=0.057..0.068 rows=3 loops=1)
                -> Index lookup on o using PRIMARY (user_id=144)  (cost=0.70 rows=2) (actual time=0.017..0.018 rows=2 loops=1)
                -> Index lookup on g using PRIMARY (game_id=o.game_id)  (cost=0.39 rows=2) (actual time=0.012..0.013 rows=2 loops=2)
```

Analysis: After adding the user_id index, the cost did not change compared to the results where no index is added, but the actual time decreases. And after adding the genre index, the cost did not change either, but the actual time decreased as well, and the actual time is even less than when the user_id index is added. After adding the played_time_index, the did not change either, but the actual time increased compared to the results where no index is added. The reason behind the increased actual time may result from the fact that played_time is not used in range queries of WHERE clause, which is one of the main advantages of B+ tree index, and the use of index adds overhead under the hood. Therefore, we decided to choose the index genre on table genres.

Fixes from Stage 2

Replacing the “Platforms” table with “Computer” because a computer is a real entity which needs to be a table. Also each computer has a platform attribute (which is an essential requirement that a user has a computer which is a supported platform for a specific game). The minimum requirements for specific games comes from the original Steam dataset.

Fixing the games table DDL command:

```
CREATE TABLE games (
        game_id INT NOT NULL auto_increment,
        name VARCHAR(255) NOT NULL,
        genre VARCHAR(255) NOT NULL,
        description VARCHAR(600) NOT NULL,
        release_date DATE NOT NULL,
        image_link VARCHAR(255),
        game_link VARCHAR(255),
        popularity INT NOT NULL,
        PRIMARY KEY (game_id)
);
```

Fixes from Stage 3

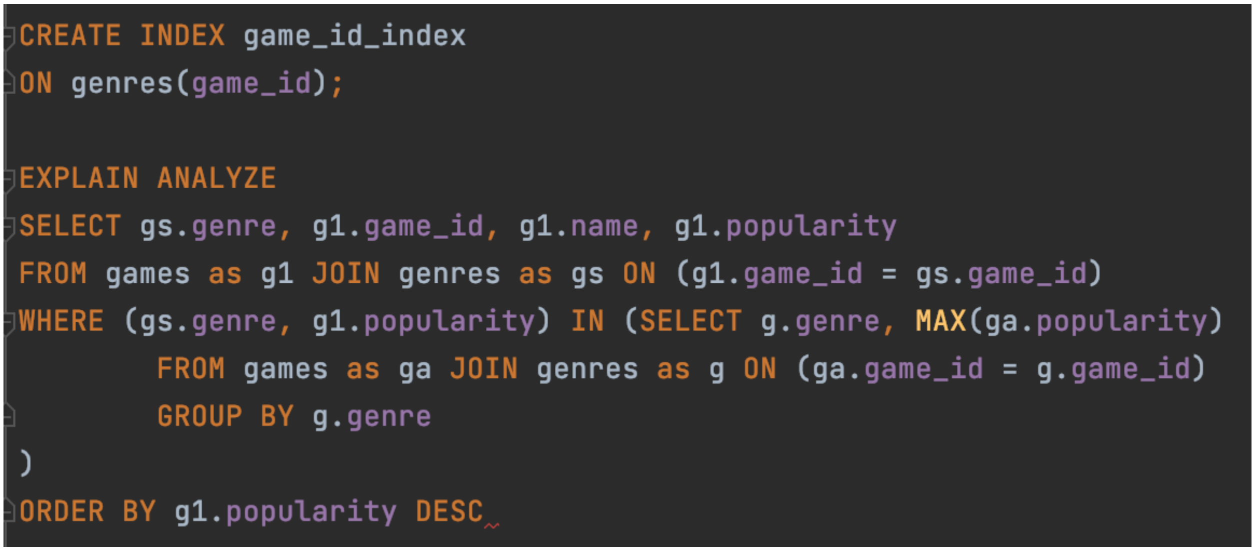
1. Fixing DDL command for table “compatible” (missing primary key)

```
CREATE TABLE compatible (
        game_id INT,
        platform VARCHAR(255),
        PRIMARY KEY (game_id, platform),
        FOREIGN KEY (game_id) REFERENCES games(game_id)
);
```

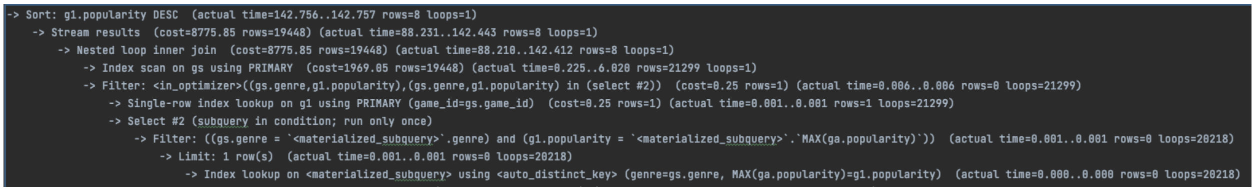
2. Add more index on advanced query 1

After trying on the genre index on genres table, we deleted the genre index, and added game_id index on table genres.

After adding game_id index on table genres:



```
CREATE INDEX game_id_index
ON genres(game_id);

EXPLAIN ANALYZE
SELECT gs.genre, g1.game_id, g1.name, g1.popularity
FROM games as g1 JOIN genres as gs ON (g1.game_id = gs.game_id)
WHERE (gs.genre, g1.popularity) IN (SELECT g.genre, MAX(ga.popularity)
        FROM games as ga JOIN genres as g ON (ga.game_id = g.game_id)
        GROUP BY g.genre
)
ORDER BY g1.popularity DESC
```



```
-> Sort: g1.popularity DESC  (actual time=142.756..142.757 rows=8 loops=1)
    -> Stream results  (cost=8775.85 rows=19448) (actual time=88.231..142.443 rows=8 loops=1)
        -> Nested loop inner join  (cost=8775.85 rows=19448) (actual time=88.210..142.412 rows=8 loops=1)
            -> Index scan on gs using PRIMARY  (cost=1969.05 rows=19448) (actual time=0.225..6.020 rows=21299 loops=1)
            -> Filter: <in_optimizer>((gs.genre,g1.popularity),(gs.genre,g1.popularity) in (select #2))  (cost=0.25 rows=1) (actual time=0.006..0.006 rows=0 loops=21299)
                -> Single-row index lookup on g1 using PRIMARY (game_id=gs.game_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=21299)
                -> Select #2 (subquery in condition; run only once)
                    -> Filter: ((gs.genre = `<materialized_subquery>`.genre) and (g1.popularity = `<materialized_subquery>`.`MAX(ga.popularity)`))  (actual time=0.001..0.001 rows=0 loops=20218)
                        -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=20218)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (genre=gs.genre, MAX(ga.popularity)=g1.popularity)  (actual time=0.000..0.000 rows=0 loops=20218)
```

Analysis: After adding the popularity index, the cost did not change compared to the results where no index is added, but the actual time decreases. And after adding the genre index, the cost did not change either, but the actual time increased. The reason behind the increased actual time may result from the fact that popularity is not used in range queries of WHERE clause, which is one of the main advantages of B+ tree index, and the use of index adds overhead under the hood. In addition, after adding game_id index in table games, the cost did not change either while the actual time increased. Therefore, we decided to choose to index on the popularity column on table games.