Advanced SQL Query 1:

The following SQL Query grabs the first 15 games that are of the genre Action OR Indie and combines them into one result. The result is the game name and true or false for whether or not the game is Action OR Indie:

```
SELECT MyGames.GameName, Genre.GenreIsAction as Action, Genre.GenreIsIndie as Indie
FROM MyGames
JOIN Genre ON MyGames.GameGenreId = Genre.GenreId
WHERE Genre.GenreIsAction LIKE 'TRUE'
UNION
SELECT MyGames.GameName, Genre.GenreIsAction as Action, Genre.GenreIsIndie as Indie
FROM MyGames
JOIN Genre ON MyGames.GameGenreId = Genre.GenreId
WHERE Genre.GenreIsIndie LIKE 'TRUE'
```

GameName	Action	Indie
Half-Life: Blue Shift	TRUE	FALSE
Half-Life 2: Episode Two	TRUE	FALSE
Red Orchestra: Ostfront 41-45	TRUE	FALSE
Dark Messiah of Might & Magic	TRUE	FALSE
QUAKE II Mission Pack: The Reckoning	TRUE	FALSE
Call of Duty(r) 2	TRUE	FALSE
Call of Duty(r) 4: Modern Warfare(r)	TRUE	FALSE
Spear of Destiny	TRUE	FALSE
Act of War: High Treason	TRUE	FALSE
Call of Duty(r): Modern Warfare(r) 2	TRUE	FALSE
Shadowgrounds Survivor	TRUE	FALSE
Grand Theft Auto: San Andreas	TRUE	FALSE
Tom Clancys Splinter Cell Chaos Theory(r)	TRUE	FALSE
Pirates Vikings and Knights II	TRUE	TRUE
Watchmen: The End is Nigh	TRUE	FALSE
Aliens: Colonial Marines Collection	TRUE	FALSE

Before Indexing:

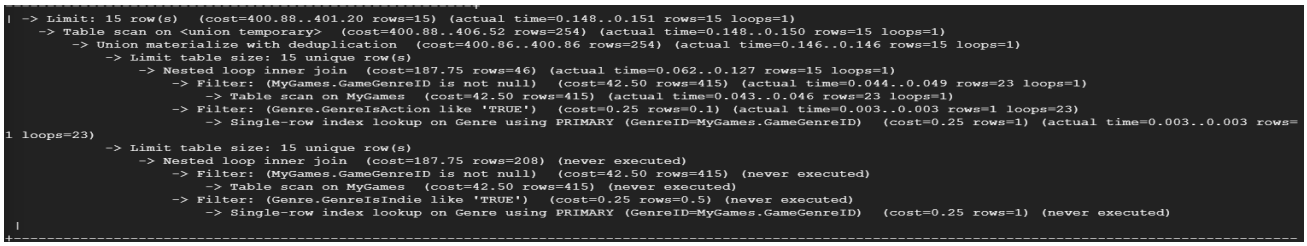
The default indexing method had a cost of approximately 848 over 15 rows and had a real time result of .405 seconds. This method has no indexing heuristics added onto it and for the following indexing designs we aim to have a lower cost and actual time performance.

```
| -> Limit: 15 row(s)  (cost=848.18..848.73 rows=15) (actual time=0.405..0.408 rows=15 loops=1)
    -> Table scan on <union temporary>  (cost=848.18..851.79 rows=92) (actual time=0.404..0.407 rows=15 loops=1)
        -> Union materialize with deduplication  (cost=848.14..848.14 rows=92) (actual time=0.403..0.403 rows=15 loops=1)
            -> Limit table size: 15 unique row(s)
                -> Nested loop inner join  (cost=419.46 rows=46) (actual time=0.085..0.380 rows=15 loops=1)
                    -> Filter: (MyGames.GameGenreID is not null)  (cost=42.50 rows=415) (actual time=0.064..0.070 rows=23 loops=1)
                        -> Table scan on MyGames  (cost=42.50 rows=415) (actual time=0.063..0.067 rows=23 loops=1)
                    -> Filter: (Genre.GenreIsAction like 'TRUE')  (cost=0.81 rows=0.1) (actual time=0.013..0.013 rows=1 loops=23)
                        -> Single-row index lookup on Genre using PRIMARY (GenreID=MyGames.GameGenreID)  (cost=0.81 rows=1) (actual time=0.013..0.013 rows=
1 loops=23)
            -> Limit table size: 15 unique row(s)
                -> Nested loop inner join  (cost=419.46 rows=46) (never executed)
                    -> Filter: (MyGames.GameGenreID is not null)  (cost=42.50 rows=415) (never executed)
                        -> Table scan on MyGames  (cost=42.50 rows=415) (never executed)
                    -> Filter: (Genre.GenreIsIndie like 'TRUE')  (cost=0.81 rows=0.1) (never executed)
                        -> Single-row index lookup on Genre using PRIMARY (GenreID=MyGames.GameGenreID)  (cost=0.81 rows=1) (never executed)
 |
```

After Genre indexing:

Indexing improves the speed of data retrieval based on the columns specified and allows the DBMS to “index” or look up data quicker.
Since this advanced SQL query constantly checks Genre’s “GenreIsAction” and “GenreIsIndie” columns, I made an index involving these two columns.

RESULTS:
The cost dropped from 848 to 400 after creating this index. Moreover, the actual time reduced from .405 to .148 seconds which is significant enough for users to notice. Potentially this change in performance is due to the fact that this query involves a UNION operation, so on the second SELECT, the engine can quickly find results from the first SELECT statement. Therefore, we will stick with this implementation for this query and potentially more UNION queries in the future due to the cost being halved.



ADVANCED QUERY 2:

```
SELECT
   G.MetacriticScore,
   D.DeveloperWebsite,
   COUNT(*) AS NumberOfGames
FROM
   MyGames AS G
JOIN
   Developers AS D ON G.GameDeveloperID = D.DeveloperID
WHERE
   D.DeveloperWebsite <> 'None'
GROUP BY
   G.MetacriticScore,
   D.DeveloperWebsite
ORDER BY
   NumberOfGames DESC
LIMIT 15;
```

```
    -> LIMIT 15;
+-----------------+-------------------------------------------+---------------+
| MetacriticScore | DeveloperWebsite                          | NumberOfGames |
+-----------------+-------------------------------------------+---------------+
|               0 | http://frogwares.com/                     |             7 |
|               0 | http://store.steampowered.com/app/901660/ |             5 |
|               0 | http://www.dawnofwar.com                  |             4 |
|               0 | http://store.steampowered.com/app/901663/ |             4 |
|               0 | http://gsc-game.com/                      |             3 |
|               0 | http://www.totalwar.com                   |             3 |
|               0 | http://www.race-game.org/                 |             3 |
|               0 | http://www.lucasarts.com/                 |             3 |
|               0 | http://www.runaway-thegame.com/           |             2 |
|               0 | http://www.BioShockGame.com               |             2 |
|               0 | http://www.totalwar.com/                  |             2 |
|               0 | http://www.rebellion.co.uk/               |             2 |
|               0 | http://www.shankgame.com                  |             2 |
|               0 | http://www.lucasarts.com                  |             2 |
|               0 | http://www.callofduty.com/                |             2 |
+-----------------+-------------------------------------------+---------------+
15 rows in set (0.00 sec)
```

Explain Analyze:

1)

```
mysql> EXPLAIN SELECT     G.MetacriticScore,     D.DeveloperWebsite,     COUNT(*) AS NumberOfGames FROM     MyGames AS G JOIN     Developers AS D ON G.GameDeveloperID = D.DeveloperID WHERE
 D.DeveloperWebsite <> 'None' GROUP BY     G.MetacriticScore,     D.DeveloperWebsite ORDER BY     NumberOfGames DESC LIMIT 15;
+----+-------------+-------+------------+--------+-----------------+---------+---------+-------------------------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type   | possible_keys   | key     | key_len | ref                     | rows | filtered | Extra                                        |
+----+-------------+-------+------------+--------+-----------------+---------+---------+-------------------------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | G     | NULL       | ALL    | GameDeveloperID | NULL    | NULL    | NULL                    |  415 |   100.00 | Using where; Using temporary; Using filesort |
|  1 | SIMPLE      | D     | NULL       | eq_ref | PRIMARY         | PRIMARY | 4       | games.G.GameDeveloperID |    1 |    90.00 | Using where                                  |
+----+-------------+-------+------------+--------+-----------------+---------+---------+-------------------------+------+----------+----------------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

```
----------------------------------+
| -> Limit: 15 row(s)  (actual time=1.271..1.273 rows=15 loops=1)
    -> Sort: NumberOfGames DESC, limit input to 15 row(s) per chunk  (actual time=1.270..1.271 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=1.191..1.226 rows=231 loops=1)
            -> Aggregate using temporary table  (actual time=1.190..1.190 rows=231 loops=1)
                -> Nested loop inner join  (cost=483.17 rows=373) (actual time=0.088..0.820 rows=271 loops=1)
                    -> Filter: (G.GameDeveloperID is not null)  (cost=42.50 rows=415) (actual time=0.062..0.223 rows=415 loops=1)
                        -> Table scan on G  (cost=42.50 rows=415) (actual time=0.057..0.186 rows=415 loops=1)
                    -> Filter: (D.DeveloperWebsite <> 'None')  (cost=0.96 rows=1) (actual time=0.001..0.001 rows=1 loops=415)
                        -> Single-row index lookup on D using PRIMARY (DeveloperID=G.GameDeveloperID)  (cost=0.96 rows=1) (actual time=0.001..0.001 rows=1 loops=415)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

2)

Based on the output, we are considering adding an index on G.GameDeveloperID. This is despite that there's a possible_keys suggestion, it's not being used.

Also, we can add an Index on G.MetacriticScore: Since we're grouping and ordering by MetacriticScore, an index on this column could help with sorting and group-by operations, potentially avoiding the Using filesort operation which can be costly.

3)

```
| -> Limit: 15 row(s)  (actual time=1.163..1.165 rows=15 loops=1)
    -> Sort: NumberOfGames DESC, limit input to 15 row(s) per chunk  (actual time=1.163..1.164 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=1.089..1.123 rows=231 loops=1)
            -> Aggregate using temporary table  (actual time=1.087..1.087 rows=231 loops=1)
                -> Nested loop inner join  (cost=483.17 rows=373) (actual time=0.061..0.754 rows=271 loops=1)
                    -> Filter: (G.GameDeveloperID is not null)  (cost=42.50 rows=415) (actual time=0.041..0.178 rows=415 loops=1)
                        -> Covering index scan on G using idx_meta_developer  (cost=42.50 rows=415) (actual time=0.040..0.144 rows=415 loops=1)
                    -> Filter: (D.DeveloperWebsite <> 'None')  (cost=0.96 rows=1) (actual time=0.001..0.001 rows=1 loops=415)
                        -> Single-row index lookup on D using PRIMARY (DeveloperID=G.GameDeveloperID)  (cost=0.96 rows=1) (actual time=0.001..0.001 rows=1 loops=415)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

After adding these indexes we can see that our cost doesn't decrease or increase. We think that because our query is very straightforward and only uses a simple group by function to group metacritic scores. Also what helped out our cost could be that there are game developers who have been assigned multiple same metacritic scores which shows that it's simpler for the query to quickly find the groups.

Relational Schema:
MyGames(GameID, Name, ReleaseDate, MetacriticScore, Price, AgeRequirment)
Genre(GenreID, Description)
Platform(PlatformID, Name)
Developer(DeveloperId, Name)
User(UserId, UserName, Password)

DDL Commands:
MyGames

```
CREATE TABLE MyGames
(
   GameID INT NOT NULL,
   GameName varchar(255),
   ReleaseDate varchar(255),
   MetacriticScore INT,
   PRICE FLOAT,
   Age INT,
   GameGenreID INT,
   GameDeveloperID INT,
   GamePlatformID INT,
   PRIMARY KEY (GameID),
   FOREIGN KEY GameGenreID REFERENCES Genre(GenreID),
   FOREIGN KEY GameDeveloperID REFERENCES Developer(DeveloperID),
   FOREIGN KEY GamePlatformID REFERENCES Genre(PlatformID),
);
```

Genre

```
CREATE TABLE Genre (
   GenreID INT PRIMARY KEY,
   GenreIsNonGame VARCHAR(255),
   GenreIsIndie VARCHAR(255),
   GenreIsAction VARCHAR(255),
   GenreIsAdventure VARCHAR(255),
   GenreIsCasual VARCHAR(255),
   GenreIsStrategy VARCHAR(255),
   GenreIsRPG VARCHAR(255),
   GenreIsSimulation VARCHAR(255),
```

```
    GenreIsEarlyAccess VARCHAR(255),
    GenreIsFreeToPlay VARCHAR(255),
    GenreIsSports VARCHAR(255),
    GenreIsRacing VARCHAR(255),
    GenreIsMassivelyMultiplayer VARCHAR(255)
);
```

Platform

```
CREATE TABLE Platform
(
    PlatformID INT NOT NULL,
    Name VARCHAR(255),
    PRIMARY KEY (PlatformID),
   );
```

Developer

```
CREATE TABLE Developer
(
    DeveloperID INT NOT NULL,
    Name VARCHAR(255),
    PRIMARY KEY (DeveloperID)
);
```

Entities:

User Entity:
Attributes: UserID (PK), Username, Email, Password, ProfilePicture, Preferences
Assumptions:

- Each user can have 1 profile
- Each profile corresponds to 1 user

Game Entity:
Attributes: GameID (PK), GenreID(FK), GameName, ReleaseDate, Price, Description
Assumptions:

- Each game can have 1+ associated genres
- Each genre corresponds to 1+ games
- Each game can be compatible with multiple gaming platforms
- Each gaming platform can support multiple games.

Genre Entity:
Attributes: GenreID (PK), GenreName
Assumptions:

- Each genre can be associated with multiple games.
- Each game can belong to multiple genres

Platform Entity:
Attributes: PlatformID (PK), PlatformName
Assumptions:

- Each gaming platform can support multiple games
- Each game can be compatible with multiple gaming platforms

Interaction Entity:
Attributes: InteractionID (PK), UserID (FK), GameID (FK), InteractionType, InteractionDate
Assumptions:

- Each user can have one or more interactions with games
- Each game can have interactions with multiple users
- Each interaction corresponds to one user and one game
- Each interaction has attributes such as InteractionType and InteractionDate

Relationships:

User-Game Relationship:
Relationship Type: Many-to-Many
Assumptions:

- Users can interact with multiple games
- Games can have interactions with multiple users.
- Additional attributes capture the nature of these interactions

Game-Genre Relationship:
Relationship Type: Many-to-Many
Assumptions:

- Games can belong to multiple genres (e.g. A game can be both Action & RPG)
- Genres can be associated with multiple games

Game-Platform Relationship:
Relationship Type: Many-to-Many
Assumptions:

- Games can be compatible with multiple gaming platforms (e.g. Windows, Linux, and Mac)
- Gaming platforms can support multiple games

Game to Developer: Many-to-One
Assumptions:

- Developers can create many games

- Games can only have one developer

Game to SearchResult: Many-to-Many
Assumptions:
- Each game can have multiple searches
- Searches can have multiple results

SearchResult to User: Many-to-One
Assumptions:
- Users can have multiple searches
- Each search is exclusive to one user due to unique searchId

Normalization:

BCNF Normalization:
The schema above has the following functional dependencies for each relation:

User:
UserID -> Username, Password

Game:
GameID -> Name, ReleaseDate, RequiredAge, Price, Rating

Genre:
GenreId -> Description

Platform:
PlatformId -> Name

Developer:
DeveloperId -> Name

SearchResult:
ResultId -> Games

Our relations were defined in straightforward manner such that the functional dependencies can be proven to have a superkey for their respective relations. Computing the attribute closures of each follows:

{UserId}+ = Username, Password
User attributes are: Username, Password
UserId is a superkey of User

{GameId}+ = Name, ReleaseDate, RequiredAge, Price, Rating
Game attributes are: Name, ReleaseDate, RequiredAge, Price, Rating
GameId is a superkey of Game

{DeveloperId}+ = Name
Developer attributes are: Name
DeveloperId is a superkey of Developer

{PlatformId}+ = Name
Platform attributes are: Name
PlatformId is a superkey of Platform

{ResultId}+ = Username, Password
SearchResult attributes are: Username, Password
ResultId is a superkey of SearchResult

The set of relations fits BCNF form since for each relation the non-trivial functional dependencies all have a superkey on their left-hand side.

We chose to use the BCNF form to normalize our database because it was effective in minimizing information loss and avoiding redundancy, which made sense given the nature of our database where we wanted to ensure accuracy. It is also efficient in that it gives us lossless join, although in comparison to 3NF, it does not give us dependency preservation - which matters less in the context of our project where we are giving recommendations.

Relational Schema:
Table-Name(Column1:Domain [PK], Column2:Domain [FK to table.column], Column3:Domain,...)

User (
  UserID:INT[PK]
  Password:VARCHAR(255)
  Username:VARCHAR(255)
)

Game (
  GameID:INT[PK]
  Name:VARCHAR(255)
  ReleaseDate:VARCHAR(255)
  RequiredAge:INT
  GenreID:INT[FK to Genre.GenreId]
  PlatformID:INT[FK to Platform.PlatformId]
  DeveloperId:INT[FK to Developer.DevelopId]

```
    Rating:INT
    Price:Int
)

Genre(
    GenreId(PK)
    Description:VARCHAR(255)
)

Platform(
    PlatformId(PK)
    Description:VARCHAR(255)
)

Developer(
    DeveloperId(PK)
    Description:VARCHAR(255)
)

SearchResult(
    ResultId:INT(PK)
    UserId:INT(FK to User.UserId)
    Games:VARCHAR(255)
)
```

Fixing suggestions from previous stage:
A suggestion from the previous stage was to expand on functionality other than search.

A major feature our project intends to have is to also store a history for each user in order for users to be able to reference previous purchases and queries. The table will be smaller in size compared to other tables such as the massive Game table. This will help the user understand their recent history only since that would be of more importance to them than queries made months ago.

This history table will be produced from the output of a filter procedure we will have whenever a user wants to query our game database. When a user has expressed interest in a game by either filtering by categories such as genre or through a “shopping cart/starred” feature, the games can effectively be stored in history so users can view what they have been interested in the past.

In order for a user to access their history database it would be as simple as getting all records from this table. If the user only wants the most recent history we can then limit the amount of records returned.

The columns of this table will be a list of games and the UserId which will relate to a specific user. The list of games will represent the history for the user. The UserId will ensure that

a specific history can be linked to a user. Therefore, our history table will have a one-to-one relationship with the user table. This way any user will have access to their own history.