

SteamData®

Database Design, Data Integration and Application Development

Roy Koljonen

September 2024

"SteamData" is not really a registered trademark

Introduction

My Project Goals:

Azure Cloud Database

Implement and maintain a cloud database for storing game data.

Back-End Development

Develop the back-end system for managing and integrating game data with the database.

Front-End Design

Create a web interface for users to search, filter, and view game data.

Database Table Reference Diagram



SQL Commands

```
CREATE TABLE indie_games (  
  AppID INT PRIMARY KEY,  
  name VARCHAR(255) NOT NULL,  
  release_date DATE,  
  price DECIMAL(10, 2),  
  metacritic_score INT,  
  recommendations INT,  
  positive INT,  
  negative INT,  
  estimated_owners VARCHAR(255),  
  average_playtime_forever INT,  
  peak_ccu INT,  
  pct_pos_total DECIMAL(5, 2),  
  num_reviews_total INT  
);  
  
CREATE TABLE tags (  
  tag_id INT PRIMARY KEY AUTO_INCREMENT,  
  tag_name VARCHAR(255) NOT NULL  
);  
  
CREATE TABLE game_tags (  
  AppID INT,  
  tag_id INT,  
  PRIMARY KEY (AppID, tag_id),  
  FOREIGN KEY (AppID) REFERENCES indie_games(AppID) ON DELETE CASCADE,  
  FOREIGN KEY (tag_id) REFERENCES tags(tag_id) ON DELETE CASCADE  
);
```

Many-to-Many Relationships

- **game_tags Table:**
 - Establishes a **many-to-many relationship** between games and tags (same for genres and categories).
 - A single game can have multiple tags, and a single tag can apply to multiple games.
- **Example:**
 - **Game "A"** with `AppID = 1` is tagged as **"Indie"** and **"Horror"**.
 - **"Indie"** has `tag_id = 1`, and **"Horror"** has `tag_id = 2`.
 - The `game_tags` table will contain:
 - `(AppID = 1, tag_id = 1)` for the **"Indie"** tag.
 - `(AppID = 1, tag_id = 2)` for the **"Horror"** tag.
- **Foreign Key Constraints:**

Ensure that each `AppID` in `game_tags` exists in `indie_games`, and each `tag_id` exists in `tags`.
- **Cascade Deletion:**

Automatically removes linked entries in `game_tags` when a game or tag is deleted.

indie_games :

negative TMT													
Results Messages Chart													
	AppID	name	release...	price	metac...	recom...	positive	negative	estimated_...	ave...	peak...	pct_p...	num_...
1	226280	Warp Frontier	2021-09-28	14.99	66	0	40	2	0 - 20000	0	0	93.00	30
2	244930	SNOW - The Ult...	2020-11-11	14.99	0	2224	7786	4303	1000000 - 2000...	294	2	63.00	2224
3	251590	Soul Saga	2020-02-28	14.99	0	0	13	37	0 - 20000	0	0	35.00	37
4	251650	Ray's The Dead	2020-10-22	19.99	0	0	37	23	0 - 20000	0	0	67.00	34
5	251950	WWII Online	2023-07-06	0.00	73	171	1788	1817	200000 - 500000	0	13	49.00	3607
6	257420	Serious Sam 4	2020-09-24	39.99	68	12931	12178	2446	1000000 - 2000...	1220	116	83.00	12969

tags :

negative TMT		
Results Messages		
	tag_id	tag_name
1	132	'Horror'
2	133	'Online Co-Op'
3	134	'Multiplayer'
4	135	'Psychological Horror'
5	136	'Co-op'
6	137	'VR'

game_tags :

Results Messages		
	AppID	tag_id
2359	2930140	132
2360	2930620	132
2361	2933870	132
2362	2938620	132
2363	2940370	132

```

graph TD
    MainForm[Main Form] --> CSVImport[CSV Import Module]
    CSVImport -- Parse Data --> Validation[Validation Module]
    Validation -- Validate and Compare --> Filtering[Filtering Module]
    Filtering -- Filter Data --> Export[Export Module]
    Export -- Export CSV --> DBData[Data View DB Data]
    CSVImport -- Parse Data --> DBData
    Validation -- Validate and Compare --> DBData
    Filtering -- Filter Data --> DBData
    Export -- Export CSV --> DBData
    DBData -- Pull Data --> CSVImport
    DBData -- Pull Data --> Validation
    DBData -- Pull Data --> Filtering
    DBData -- Pull Data --> Export
  
```

The flowchart illustrates the data import and export process. It begins with the **Main Form**, which leads to the **CSV Import Module**. From there, the process flows through the **Validation Module** (labeled **Parse Data**) and the **Filtering Module** (labeled **Validate and Compare**). The **Filtering Module** then leads to the **Export Module** (labeled **Filter Data**). The **Export Module** leads to **Data View DB Data** (labeled **Export CSV**). There are also direct paths from the **CSV Import Module**, **Validation Module**, **Filtering Module**, and **Export Module** to **Data View DB Data** (labeled **Parse Data**, **Validate and Compare**, **Filter Data**, and **Export CSV** respectively). Finally, **Data View DB Data** leads back to the **CSV Import Module**, **Validation Module**, **Filtering Module**, and **Export Module** (labeled **Pull Data**).

Validation Module

- Ensures there are no duplicate entries.
- Cross-references tags, genres, and categories.

- Filters data before pushing to the database or exporting.
- Supports filtering by name, genre, tag, and date.
- Prepares the final data for upload/export.

- Manages the interactions with the Azure Cloud Database.
- Sends filtered data to the database in batches, optimizing performance.

Validation Module:

- Loops through imported data, validates each game, and updates the view with valid entries.
- Logs progress and statistics (valid, invalid, duplicate).
- Basic validation by checking required fields `AppID`, `Name` and comparing key fields (tags, price, reviews, etc.) with the database.
- If all fields match, it's marked as a duplicate.
- Returns `Valid`, `Invalid` or `Duplicate` status.

```
foreach (var importedGame in importGames.ToList())
{
    var dbGame = dbGames.FirstOrDefault(g => g.AppID == importedGame.AppID);
    var result = ValidateGameData(importedGame, dbGame);
    rowCounter++;

    switch (result)
    {
        case ValidationResult.Valid:
            validGames.Add(importedGame);
            validCounter++;
            break;
        case ValidationResult.Invalid:
            invalidCounter++;
            break;
        case ValidationResult.Duplicate:
            duplicateCounter++;
            break;
    }
}
```

Import CSV	Export CSV	Fetch SteamSpy	Test DB Connectio	Pull DB Data	Validate Game Data	Update DB Data	Batch Push to DB	Purge DB
genres	positive	negative	estimated_owners	average_playtime_fc	peak_ccu	tags	pct_pos	

```
public ValidationResult ValidateGameData(Game importedGame, Game dbGame)
{
    if (importedGame.AppID <= 0 || string.IsNullOrEmpty(importedGame.Name))
    { return ValidationResult.Invalid; }

    if (dbGame == null)
    { return ValidationResult.Valid; }

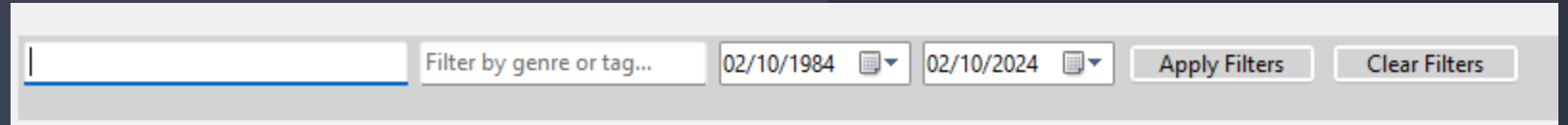
    var sortedImportedTags = importedGame.Tags.OrderBy(t => t).ToList();
    var sortedDbTags = dbGame.Tags.OrderBy(t => t).ToList();
    var sortedImportedGenres = importedGame.Genres.OrderBy(g => g).ToList();
    var sortedDbGenres = dbGame.Genres.OrderBy(g => g).ToList();
    var sortedImportedCategories = importedGame.Categories.OrderBy(c => c).ToList();
    var sortedDbCategories = dbGame.Categories.OrderBy(c => c).ToList();

    if (
        sortedImportedTags.SequenceEqual(sortedDbTags) &&
        sortedImportedGenres.SequenceEqual(sortedDbGenres) &&
        sortedImportedCategories.SequenceEqual(sortedDbCategories) &&
        importedGame.Price == dbGame.Price &&
        importedGame.ReleaseDate == dbGame.ReleaseDate &&
        importedGame.MetacriticScore == dbGame.MetacriticScore &&
        importedGame.Recommendations == dbGame.Recommendations &&
        importedGame.Positive == dbGame.Positive &&
        importedGame.Negative == dbGame.Negative &&
        importedGame.EstimatedOwners == dbGame.EstimatedOwners &&
        importedGame.AveragePlaytime == dbGame.AveragePlaytime &&
        importedGame.PeakCcu == dbGame.PeakCcu &&
        importedGame.PctPosTotal == dbGame.PctPosTotal &&
        importedGame.NumReviews == dbGame.NumReviews)
    { return ValidationResult.Duplicate; }

    return ValidationResult.Valid;
}
```

Filter Module:

- Retrieves user inputs for name, genre/tag, and date range filters.
- Constructs a dynamic filter list based on the entered criteria:
 - **Name filter** checks for partial matches in the `Name` field.
 - **Genre/Tag filter** checks for matches in both the `Tags` and `Genres` fields.
 - **Date filter** ensures the `release_date` falls within the selected range.
- Joins the filter conditions with "AND" to form the final query.
- Applies the filter to the `DataView` (`gameView.RowFilter`) to display matching results in the `DataGridView`.
- Updates the row count label with the number of visible rows.
- Refreshes the `importGames` list with the filtered results.



The screenshot shows a filter module interface. It includes a text input field for a name, a dropdown menu for filtering by genre or tag, and two date pickers for a date range (02/10/1984 to 02/10/2024). There are 'Apply Filters' and 'Clear Filters' buttons.

```
private void ApplyFilters()
{
    string nameFilter = txtNameFilter.Text.ToLower();
    string genreTagFilter = txtGenreTagFilter.Text.ToLower();
    DateTime startDate = startDatePicker.Value.Date;
    DateTime endDate = endDatePicker.Value.Date;

    var filterList = new List<string>();

    if (!string.IsNullOrEmpty(nameFilter))
    { filterList.Add($"(Name LIKE '{nameFilter}%'"); }

    if (!string.IsNullOrEmpty(genreTagFilter))
    { filterList.Add(
        $"(Tags LIKE '{genreTagFilter}%' +
        $"OR Genres LIKE '{genreTagFilter}%'"); }

    filterList.Add(
        $"(release_date >= #{startDate:yyyy-MM-dd}# +
        $"AND release_date <= #{endDate:yyyy-MM-dd}#)");

    gameView.RowFilter = string.Join(" AND ", filterList);

    int visibleRowCount = gameView.Count;
    rowCountLabel.Text = $"Imported row count: {visibleRowCount}";

    UpdateImportGamesList();
}
```



```

var insertGameValues = new StringBuilder();
foreach (var game in games)
{
    string name = game.Name.Replace("'", "");
    insertGameValues.Append($"{game.AppID},
    {game.Name},

    // All other values

    {game.NumReviews} ");
}

if (insertGameValues.Length > 0)
{ insertGameValues.Length--; }

string indieGameInsertQuery = $"
INSERT INTO indie_games (
    AppID,
    name,

    // All other values

    num_reviews_total
) VALUES
{insertGameValues.ToString()}
ON DUPLICATE KEY UPDATE
    name = VALUES(name),

    // All other values

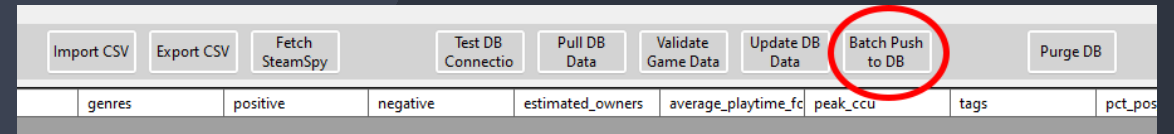
    num_reviews_total = VALUES(num_reviews_total);";

using (var cmd = new MySqlCommand(indieGameInsertQuery, connection, transaction))
{ await cmd.ExecuteNonQueryAsync(); }

await InsertCategoriesBatchAsync(games, connection, transaction);
await InsertGenresBatchAsync(games, connection, transaction);
await InsertTagsBatchAsync(games, connection, transaction);

await transaction.CommitAsync();
}
...

```



SQL Bulk Insert Logic

- **Bulk Query Construction:**

A `StringBuilder` concatenates values for all games in the `games` list, forming a bulk `INSERT` query.

- **Optimized SQL Execution:**

Executes the bulk query in one SQL command. The `ON DUPLICATE KEY UPDATE` clause updates records if they already exist (matching `AppID`).

- **Transaction & Error Handling:**

Wraps the entire process in one transaction to ensure all operations either succeed or are rolled back on error.

- **Inserting Related Data:**

After games, categories, genres, and tags are also inserted into their tables in batches for efficiency.

Insert Tags (and Genres & Categories):

- Loops through each game and its tags.
- For each tag, retrieves or inserts the tag ID using `GetOrCreateTagAsync`.
- Builds a bulk insert query to add tags in batches.
- Executes the query to insert all game-tag pairs at once.
- Avoids inserting duplicate entries using `ON DUPLICATE KEY UPDATE`.

```
private async Task InsertTagsBatchAsync(
    List<Game> games, MySqlConnection connection, MySqlTransaction transaction)
{
    var tagValues = new StringBuilder();
    foreach (var game in games)
    {
        foreach (var tag in game.Tags)
        {
            int tagId = await GetOrCreateTagAsync(tag, connection, transaction);
            tagValues.Append($"({game.AppID}, {tagId}),");
        }
    }
    if (tagValues.Length > 0)
    { tagValues.Length--; }

    string gameTagInsertQuery = $"
        INSERT INTO game_tags (AppID, tag_id) VALUES
        {tagValues.ToString()}
        ON DUPLICATE KEY UPDATE tag_id = VALUES(tag_id);";

    using (var cmd = new MySqlCommand(gameTagInsertQuery, connection, transaction))
    { await cmd.ExecuteNonQuery(); }
}
```

Get or Insert Tag:

- Retrieves the tag ID from the database or inserts a new tag if it doesn't exist.
- Uses a `tagCache` dictionary to store tag IDs, minimizing repeated database queries.
- Steps:
 - First, check if the tag exists in the cache.
 - If not, execute a `SELECT` query to find the tag in the `tags` table.
 - If the tag is found, return its ID.
 - If not, insert the tag into the database and retrieve its ID using `LastInsertedId`.
 - Add the tag and its ID to the cache for future lookups.

```
private async Task<int> GetOrCreateTagAsync(
    string tag, MySqlConnection connection, MySqlTransaction transaction)
{
    if (tagCache.TryGetValue(tag, out int tagId))
    { return tagId; }
    string selectQuery = "SELECT tag_id FROM tags WHERE tag_name = @TagName";
    using (var cmd = new MySqlCommand(selectQuery, connection, transaction))
    {
        cmd.Parameters.AddWithValue("@TagName", tag);
        var result = await cmd.ExecuteScalarAsync();
        if (result != null)
        { tagId = Convert.ToInt32(result); }
        else
        {
            string insertQuery = "INSERT INTO tags (tag_name) VALUES (@TagName)";
            using (var insertCmd = new MySqlCommand(insertQuery, connection, transaction))
            {
                insertCmd.Parameters.AddWithValue("@TagName", tag);
                await insertCmd.ExecuteNonQueryAsync();
                tagId = (int)insertCmd.LastInsertedId;
            }
        }
    }
    tagCache[tag] = tagId;
    return tagId;
}
```

Front-End Web Design

- Web Interface:
 - Simple design allowing users to search, filter, and view game data.
- Data Presentation:
 - Data is presented in tables with dynamic filtering options.
 - Individual game details for insights into specific titles.

Indie Game Database

Filter for indie games released after the year 2020

AppID:

Name:

Category:

--Select Category--



Genre:

--Select Genre--



Tag:

--Select Tag--



Search

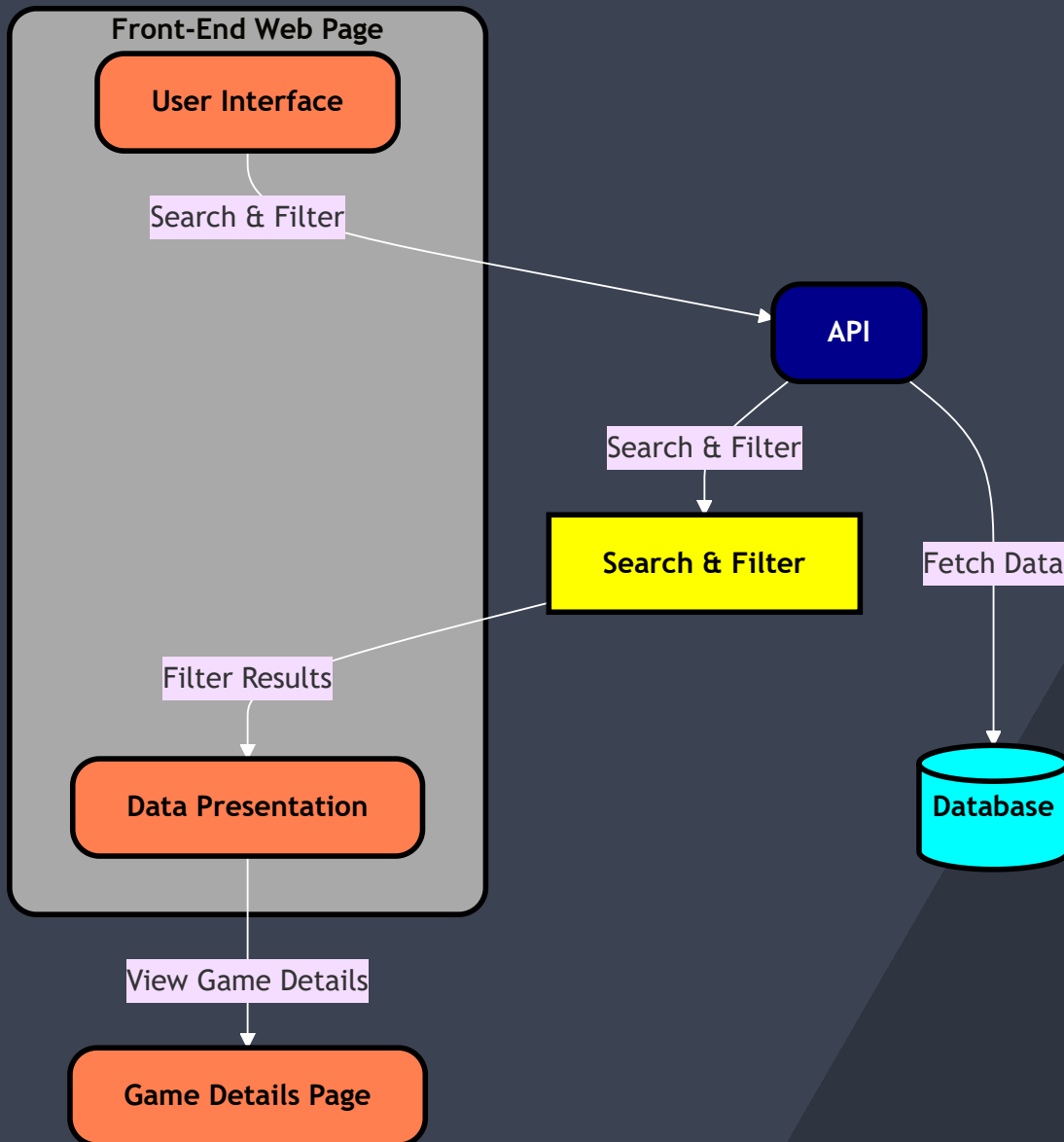
Fetch First 10 Games

Warp Frontier

Categories: Commentary available / Family Sharing / Single-player / Steam Achievements / Steam Cloud / Steam Trading Cards

Genres: [Adventure](#) / [Indie](#)

Tags: [2D](#) / [Adventure](#) / [Choices Matter](#) / [Conversation](#) / [Dark Humor](#) / [Detective](#) / [Drama](#) / [Emotional](#) / [Futuristic](#) / [Hand-drawn](#) / [Indie](#) /



Workflow:

- **User Interface:** Users search/filter games on the front-end.
- **Search & Filter:** Filters are passed to the backend, processed by the API.
- **API:** Retrieves data from the database based on filter conditions.
- **Database:** Stores and fetches game data (genres, tags, etc.).
- **Data Presentation:** Filtered results are displayed on the UI.
- **Game Details Page:** Users can view detailed information for a selected game.

HTML & Styles:

- **Search fields** for AppID, Name, Category, Genre, and Tags are arranged in a **grid layout**.
- The **search button** triggers the `search()` function, sending user input values for filtering.
- Designed with **responsive CSS grid**, adjusting the layout to fit various screen sizes.

```
<body>
  <h1>Indie Game Database</h1>
  <p>Filter for indie games released after the year 2020</p>

  <div class="search-grid">
    <div class="grid-item">
      <label for="appid">AppID:</label>
      <input type="number" id="appid" placeholder="Enter AppID" />
    </div>
  ...
  <div class="search-button">
    <button onclick="search()">Search</button>
  </div>
  ...
```

```
.search-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
  gap: 20px;
  margin-bottom: 20px;
}

button {
  padding: 10px 20px;
  margin: 5px 0;
  background-color: #2a475e;
  color: #e6e7e8;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  transition: background-color 0.3s;
}
```

`search()` Function:

- **Collects input values** from various fields like AppID, Name, Category, Genre, and Tag.
- Creates **query parameters** dynamically based on the fields filled by the user.
- Makes a **GET request** using `fetch()` to send the parameters to the backend.

```
function search() {
  const appId = document.getElementById('appid').value;
  const name = document.getElementById('name').value;
  const category = document.getElementById('category').value;
  const genre = document.getElementById('genre').value;
  const tag = document.getElementById('tag').value;

  const queryParams = new URLSearchParams();

  if (appId) queryParams.append('appid', appId);
  if (name) queryParams.append('name', name);
  if (category) queryParams.append('category', category);
  if (genre) queryParams.append('genre', genre);
  if (tag) queryParams.append('tag', tag);

  fetch(`/api/games?${queryParams.toString()}`)
    .then((response) => response.json())
    .then((data) => displayGames(data))
    .catch((error) => console.error('Error:', error));
}
```

Game Search API

- `api/games` route handles search requests for games based the query parameters.
- The API handles the many-to-many junction tables with `searchGames()`.
- If filters are provided, it calls `searchGames`.
- `searchGames()` builds a dynamic SQL query based on filters and joins related tables.
- Executes the query and returns matching results.

```
app.get('/api/games', (req, res) => {
  const { limit = 10, name, category, genre, tag } = req.query;

  if (name || category || genre || tag) {
    searchGames({ name, category, genre, tag }, (err, games) => {
      if (err) return res.status(500).json({ error: err.message });
      res.json(games);
    });
  } else {
    fetchAllGames(parseInt(limit), (err, games) => {
      if (err) return res.status(500).json({ error: err.message });
      res.json(games);
    });
  }
});
```

```
function searchGames(filters, callback) {
  const { name, category, genre, tag } = filters;
  let query =
    `SELECT ig.appid, ig.name, ig.release_date, ig.price,
      GROUP_CONCAT(DISTINCT c.category_name ORDER BY c.category_name) AS categories,
      GROUP_CONCAT(DISTINCT g.genre_name ORDER BY g.genre_name) AS genres,
      GROUP_CONCAT(DISTINCT t.tag_name ORDER BY t.tag_name) AS tags
    FROM indie_games ig
    LEFT JOIN game_categories gc ON ig.appid = gc.appid
    LEFT JOIN categories c ON gc.category_id = c.category_id
    LEFT JOIN game_genres gg ON ig.appid = gg.appid
    LEFT JOIN genres g ON gg.genre_id = g.genre_id
    LEFT JOIN game_tags gt ON ig.appid = gt.appid
    LEFT JOIN tags t ON gt.tag_id = t.tag_id
    WHERE 1=1`;
  const queryParams = [];

  if (name) {
    query += ` AND ig.name LIKE ?`;
    queryParams.push(`%${name}%`);
  }
  if (category) {
    query += ` AND c.category_name LIKE ?`;
    queryParams.push(`%${category}%`);
  }
  if (genre) {
    query += ` AND g.genre_name LIKE ?`;
    queryParams.push(`%${genre}%`);
  }
  if (tag) {
    query += ` AND t.tag_name LIKE ?`;
    queryParams.push(`%${tag}%`);
  }

  query += ` GROUP BY ig.appid`;

  connection.query(query, queryParams, callback);
}
```

Games List Display

- `displayGames()` parses and formats the data.
- Generates HTML structure for each game by creating a div element containing the game's title, metadata and a details table.
- The results are inserted into `index.html`:

```
<div id="result">
  <!-- Game results will be inserted here -->
</div>
```

```
function displayGames(games) {
  const resultDiv = document.getElementById('result');
  resultDiv.innerHTML = '';

  if (!games || games.length === 0) {
    resultDiv.innerHTML = '<p>No games found.</p>';
    return;
  }

  const parseAndFormat = (text) => { ...
  const createLinks = (text, type) => { ...

  return text
    .split(',')
    .map((item) => {
      const formattedItem = item.replace(/'/g, '').trim();
      return '<a href="${baseUrl}${encodeURIComponent(formattedItem)}"
        target="_blank" style="color: #008CFF;"> ${formattedItem}</a>';
    }).join(' / ');
  });

  games.forEach((game) => {
    const gameDiv = document.createElement('div');
    gameDiv.className = 'game-result';
    const titleDiv = document.createElement('div');
    titleDiv.className = 'game-title';
    titleDiv.innerHTML =
      '<a href="https://store.steampowered.com/app/${game.appid}"
        target="_blank" style="color: #008CFF; font-size: 1.5em;
        font-weight: bold;"> ${game.name || 'Unknown Name'}</a>';
    gameDiv.appendChild(titleDiv);

    const metaDiv = document.createElement('div');
    metaDiv.className = 'game-meta';
    metaDiv.innerHTML =
      '<div><strong>Categories:</strong> ${parseAndFormat(game.categories)}</div>
      <div><strong>Genres:</strong> ${createLinks(game.genres, 'genre')}</div>
      <div><strong>Tags:</strong> ${createLinks(game.tags, 'tag')}</div>';
    gameDiv.appendChild(metaDiv);
    const detailsTable = document.createElement('table');
    detailsTable.className = 'game-details';
    detailsTable.innerHTML =
      '<tr><td><strong>Release:</strong></td><td>${game.release_date ? new Date(game.release_date)
        .toLocaleDateString() : 'N/A'}</td></tr>
      <tr><td><strong>Price:</strong></td><td>${game.price || 'N/A'}</td></tr>

      // All other columns

      <tr><td><strong># Reviews:</strong></td><td>${game.num_reviews_total || '0'}</td></tr>';

    gameDiv.appendChild(detailsTable);
    resultDiv.appendChild(gameDiv);
  });
}
```


Summary of Future Enhancements

Backend:

- Current app has bugs and needs further debugging.
- Potential future enhancements:
 - Automatic database updates from the Steam API.
 - Optimization of inserts and data retrieval.
 - More robust filtering and data handling modules.

Frontend:

- Potential future features:
 - Sorting options for displayed data.
 - Enhanced filtering for better user experience.
 - Tools for user-driven analysis and sentiment analysis using LLMs.

Thank you

Questions?