EE 547 – Applied and Cloud Computing

Project Report

Jen-Ting Chang, Chengyu Lu, Caoyi Xue

May 8, 2023

Project Title: Meet Game

1 Summary and Description

Our web application offers a streamlined and intuitive interface that allows users to search for games based on genre, platform and so on. With our web application, users can quickly and easily find games that they love and discover new games that they never knew before. And they can also find users that have similar interests.

2 Architecture

1. Summary: Using Neo4j, a graph database, relational data is stored in a graph structure, as described above. The back-end accesses this data using GraphQL, specifically Apollo GraphQL. On the front-end, the landing page displays a list of several games. Users have the option to sign in or sign up using the button located in the top-left corner. Additionally, users can filter the games' list using the genres list on the left side, the search bar at the top, and the selection list below the search bar. Users can also sort the games using the selection list below the search box. In the end, users can access and utilize the application using the IP provided by AWS EC2. Figure 1

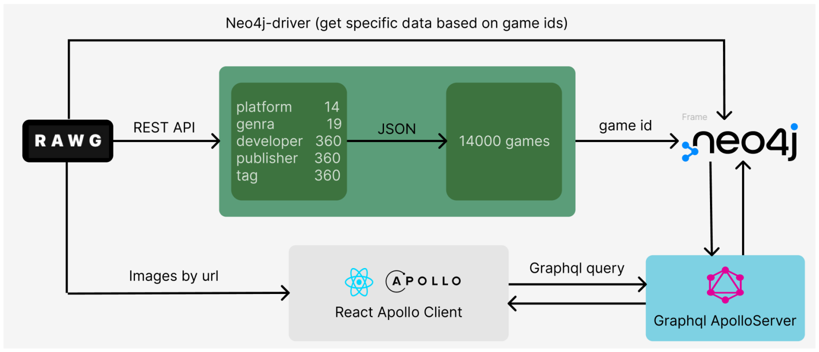

Figure 1: Architecture Diagram

2. Database: Firstly, we request data from RAWG, store the data in Neo4j, and build the relationship among each node.

3. Back-end: GraphQL is used to require data from Neo4j, and then we use cipher to parse the data. In the end, we communicate with the front-end through apollo-server.

4. Front-end: there are two pages and one modal form as below.

   Main Page: Landing page (static with some dynamic content such as number of games). There are several parts on the page. They are items which is used to present games' brief and link to description pages, Genres List which is used to filter items based on genres, Search Bar which is utilized to filter items based on game name, Sign-in & Sign-up button which is for users Sign-in and Sign-up. When users click the sign up button, they will be shown a modal form where they can set their username, password. After registration, users can sign in by typing in their username and password. Figure 2 is an example that show what we have done.

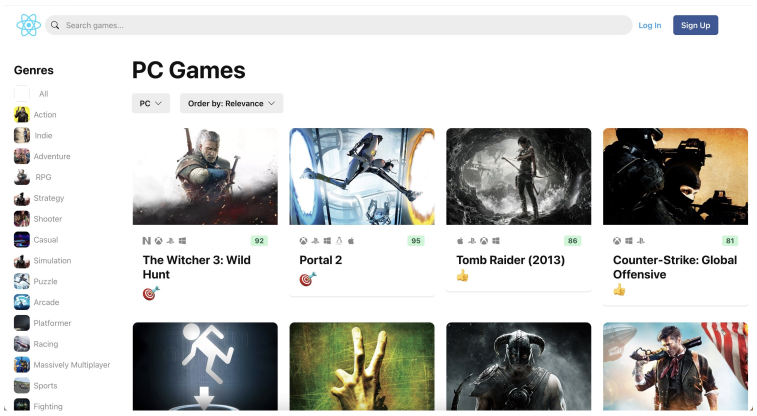

Figure 2: Main Page

   Registration Modal Form: Two modal forms used for creating an account and sign in. We use email as user account name, which is unique identifier. Figure 3 and Figure 4 are examples that show what we have done.

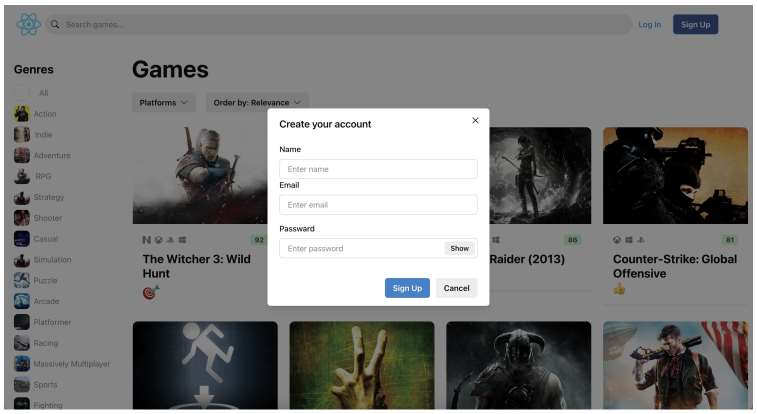

Figure 3: Registration Modal Form

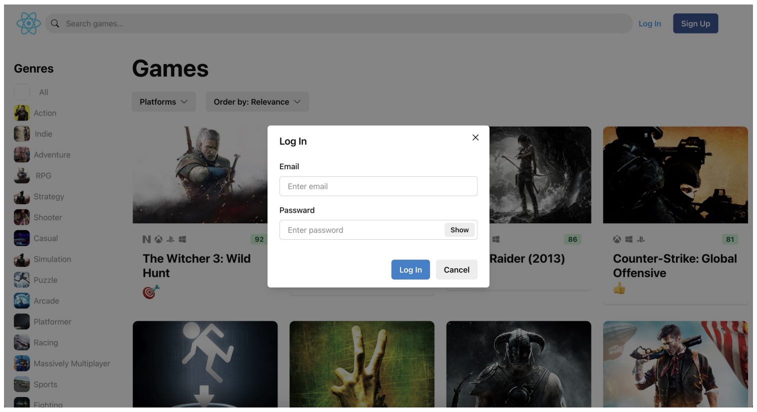

Figure 4: Sign In Modal Form

Description Page: A dynamic page where shows recommended users and games and description details(e.g. poster, description, publisher, scores).
When users click on a game poster on the main page, they will be taken to a description page which displays the poster, a full description, the name of publisher, score, and some tags. Recommendation part contains recommended games and users, which were sorted by the information of games and users.

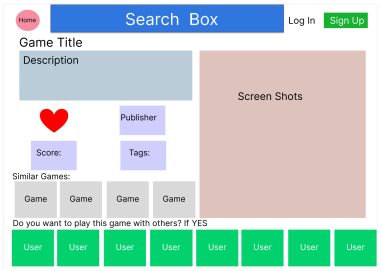

Figure 5: Description Page

5. Figure 6 shows proposed connection among above contents.

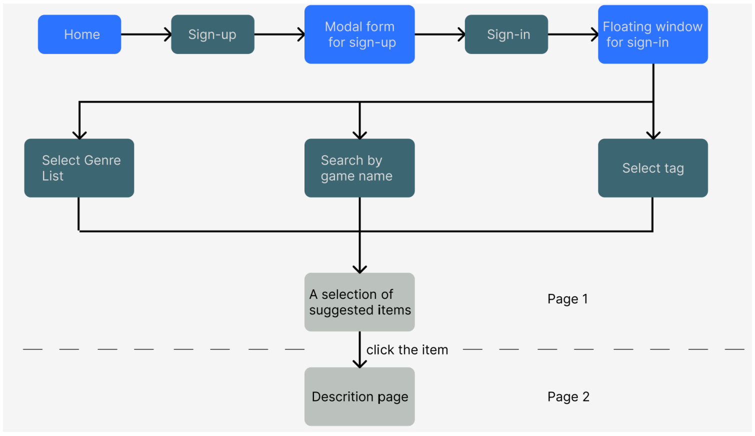

Figure 6: Architecture

3 Implementation

1. Database:

   - Neo4j is used to store user account information, game names, descriptions, genres, tags, and other relevant data. To improve efficiency, we utilize a cloud-based Neo4j database, which allows easy access to the data for the backend when the project is deployed on AWS. Figure 7 shows schema and relationship of this database.

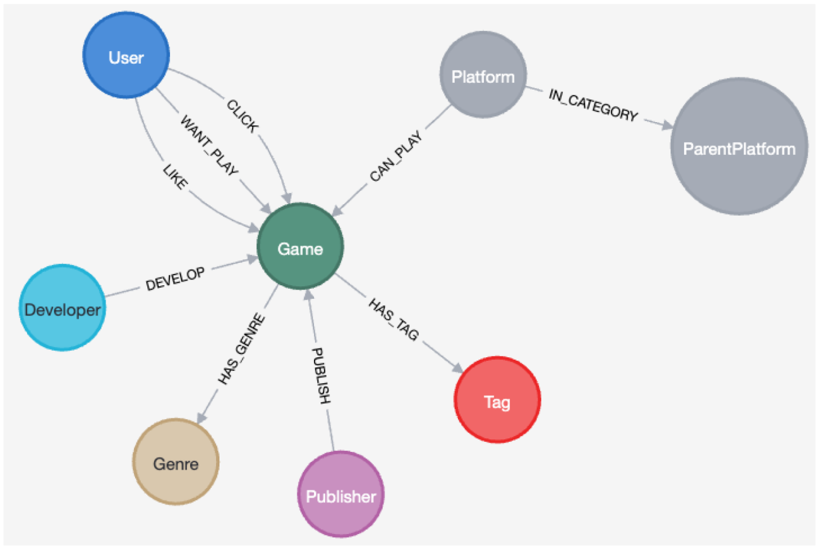

Figure 7: Database Schema and Relationship

This is the database schema. As we can see, user are connected to the game by click, want_play and like, each game have several different attributes like developer, genre, publisher, tag and platform. Different games can be connected by the same attributes. Utilizing this character, we can find similar games and connect different user together through similar interests.

Figure 8 shows game recommendation process using this database.

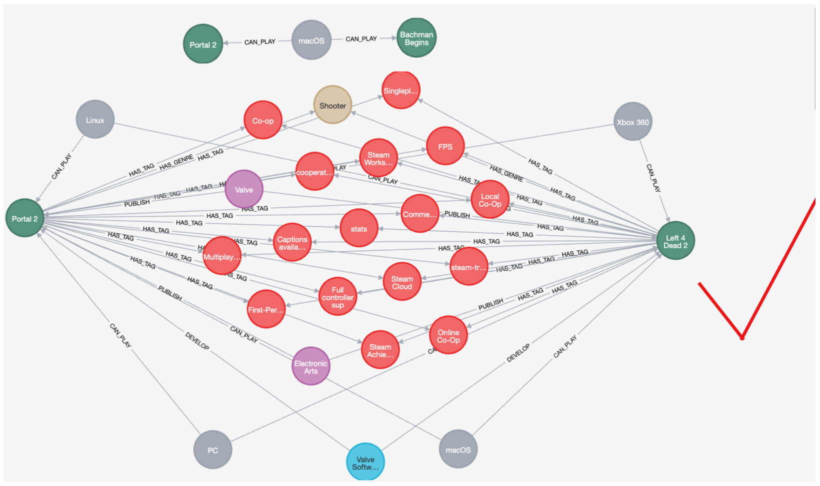

Figure 8: Game Recommendation

For the game recommendation, there are many links between the two games. We use the number of links between two games to sort the games. The more links there are, the more similar the two games are. It's worth noting that we only consider the situation where no other games are in the links.

Figure 9 shows user recommendation step 1 using this database.

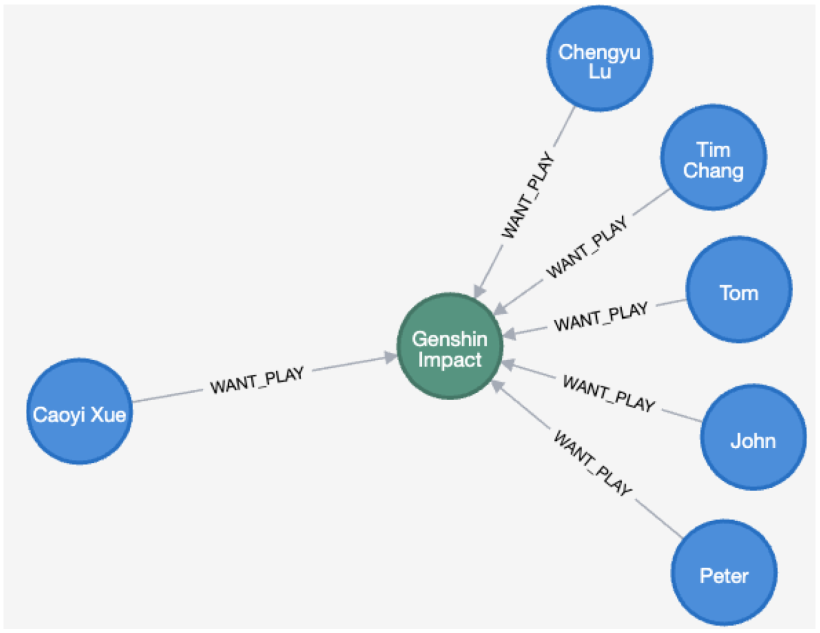

Figure 9: User Recommendation Step 1

For the user recommendation, we find out the users who want play the same game, which means that they all click the want play button. And we would sort them to recommend the users who have higher similarity.

Figure 10 and 11 show user recommendation step 2 using this database.

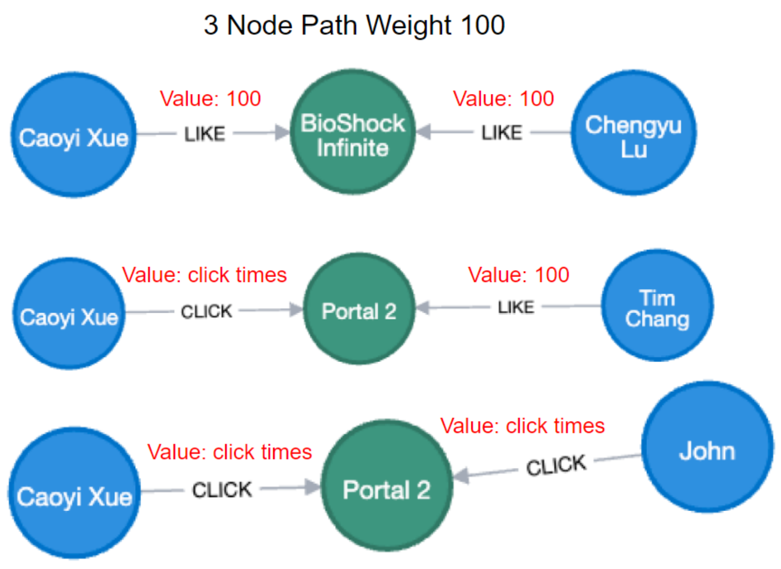

Figure 10: User Recommendation Step 2

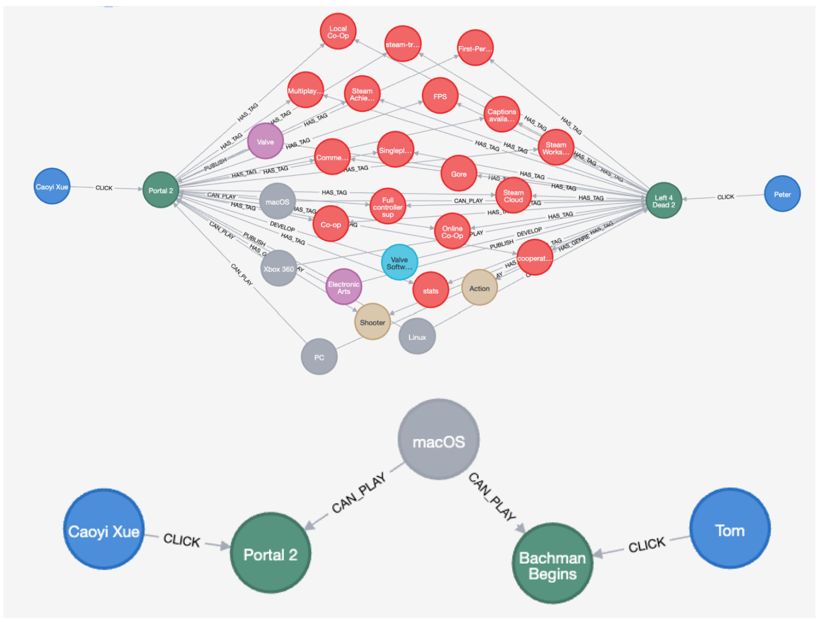

Figure 11: User Recommendation Step 2

For the users we obtained in step 1, if they are all connected by some games, each game we would add value 100 to the relationship line of the user who like the game and add click times to the relationship line of the user who click the game. And For the paths between two users, if the number of nodes equal to 3 we would add the weight P of the user pair by 100. If the number of nodes greater than 3, we would add the weight P of the user pair by link number. After calculate the sum of value R and weight P, we would multiply them and Use the product as the final score to sort players.

- RAWG: provides a comprehensive and reliable reference database for building a world-class gaming platform.

  Our project obtains game data using REST API of RAWG. There are 14 platforms 19 genres and 360 for developers, publishers and tags. We put the data into the neo4j database. And finally we can use it to provide required information for React web APP. Figure 7 is the architecture diagram.
  In the database preparation, we send requires to RAWG using REST API to collect game ids based on attributes such as platform, genre, developer and so on. According to game ids, the database require full information of the game using neo4j-driver and APOC library.

- Dependencies:
  - "axios": to require data from RAWG via REST API
  - "neo4j-driver": connect to neo4j database

2. Back-end:

- Graphql Apollo server: used to prepare query schema and resolvers for client to use.
- Graphql API: we use graphql directive to implement cypher query. We also take advantage of neo4j graphql library, if we define the type, it would automatically generate the mutation and query method. We have two important graphql type. One is user, the other is game. We use exclude directive to avoid game mutation.
- Cypher: used to parse data passed from neo4j.

```
type Game @exclude(operations: [CREATE, UPDATE, DELETE]) {
  background_image: String
  description_raw: String!
  developers: [Developer!]! @relationship(type: "DEVELOP", direction: IN)
  genres: [Genre!]! @relationship(type: "HAS_GENRE", direction: OUT)
  tags: [Tag!]! @relationship(type: "HAS_TAG", direction: OUT)
  id: String!
  metacritic: Float
  name: String!
  platforms: [Platform!]! @relationship(type: "CAN_PLAY", direction: IN)
  parent_platforms: [ParentPlatform!]!
    @cypher(
      statement: """
      MATCH (this)<-[:CAN_PLAY]-(:Platform)-[:IN_CATEGORY]->(pp:
          ParentPlatform)
      RETURN DISTINCT pp
      """
    )
  publishers: [Publisher!]! @relationship(type: "PUBLISH", direction: IN)
  rating: Float!
  rating_top: Float!
  released: Date
  screen_shots: [String]!
  slug: String!
  similar(first: Int = 6): [Game]
    @cypher(
      statement: """
      MATCH (this)-[:DEVELOP|:CAN_PLAY|:PUBLISH|:HAS_GENRE|:HAS_TAG]-(
          overlap)-[:DEVELOP|:CAN_PLAY|:PUBLISH|:HAS_GENRE|:HAS_TAG]-(res:
          Game)
      WITH res, COUNT(*) AS score
      RETURN res ORDER BY score DESC LIMIT $first
      """
    )
  users_click: [User!]!
    @relationship(type: "CLICK", direction: IN, properties: "
        ClickProperties")
  users_like: [User!]!
    @relationship(type: "LIKE", direction: IN, properties: "
        LikeProperties")
  users_want_play: [User!]! @relationship(type: "WANT_PLAY", direction:
      IN)
}
```

Listing 1: schema for Game

```
search(
    searchString: String
    genreId: String
    parentPlatformId: String
    offset: Int = 0
    limit: Int = 12
    order: String
): [Game!]
    @cypher(
      statement: """
      CALL apoc.do.when(
        $searchString IS NULL,
        'MATCH (node:Game) RETURN node',
        'CALL db.index.fulltext.queryNodes(index, searchString) YIELD
            node WITH node RETURN node',
        {searchString: $searchString, index: 'gameNameIndex'}
      ) YIELD value
      WITH DISTINCT value.node AS node
      CALL apoc.do.when(
        $genreId IS NULL,
        'RETURN node',
        'MATCH (node)-[:HAS_GENRE]->(:Genre {id: genreId}) RETURN node',
        {genreId: $genreId, node: node}
      ) YIELD value
      WITH DISTINCT value.node AS node
      CALL apoc.do.when(
        $parentPlatformId IS NULL,
        'RETURN node',
        'MATCH (:ParentPlatform {id: parentPlatformId})<-[:IN_CATEGORY
            ]-(:Platform)-[:CAN_PLAY]->(node) RETURN node',
        {parentPlatformId: $parentPlatformId, node: node}
      ) YIELD value
      WITH DISTINCT value.node AS node
      CALL apoc.do.when(
        $order IS NULL,
        'RETURN node',
        'RETURN node ORDER BY node[order] DESC',
        {order: $order, node: node}
      ) YIELD value
```

```
        WITH value.node AS node
        RETURN node
        SKIP $offset LIMIT $limit
        """
    )
}
```

Listing 2: schema for search

```
type User {
  name: String
  email: String!
  passward: String!
  click: [Game!]!
    @relationship(type: "CLICK", direction: OUT, properties: "
        ClickProperties")
  like: [Game!]!
    @relationship(type: "LIKE", direction: OUT, properties: "
        LikeProperties")
  want_play: [Game!]! @relationship(type: "WANT_PLAY", direction: OUT)
  recommend(gameId: String!, first: Int = 6): [User]
    @cypher(
      statement: """
      MATCH (this)-[:WANT_PLAY]->(g:Game {id: $gameID})<-[:WANT_PLAY]-(
          others:User)

      OPTIONAL MATCH (u)-[r1s:LIKE|CLICK]->(gs:Game)<-[r2s:LIKE|CLICK]-(
          others)
      WITH r1s, r2s, (r1s.value+r2s.value)*100 AS scores1, others
      UNWIND others as other
      WITH other AS others, sum(scores1) AS scores1

      OPTIONAL MATCH (u)-[rel21s:LIKE|CLICK]->(g1s:Game)-[:DEVELOP|
          CAN_PLAY|PUBLISH|HAS_GENRE|HAS_TAG]-(overlaps)-[:DEVELOP|
          CAN_PLAY|PUBLISH|HAS_GENRE|HAS_TAG]-(g2s:Game)<-[rel22s:LIKE|
          CLICK]-(others)
      WITH rel21s, rel22s, count(overlaps)*(rel21s.value+rel22s.value) AS
           scores2, others, scores1
      UNWIND others AS other
      WITH other, sum(scores2) AS scores2, scores1
      WITH other AS others, scores2+scores1 AS scores
      RETURN others ORDER BY scores DESC
      """
    )
}
```

Listing 3: schema for User

- Dependencies:
  - "@neo4j/graphql": This library provides the link between the Neo4j driver and GraphQL server. When executing queries and mutations against the generated schema, the Neo4j GraphQL Library generates a single Cypher query, which is then executed against the database. This approach eliminates the N+1 Problem, improving the efficiency and performance of GraphQL implementations.
  - "@apollo/server": This library utilizes the schema and resolver from the Neo4j GraphQL Library to serve GraphQL queries and mutations to users. Apollo Server 4 is the latest version available.

- "@neo4j/introspector 1.0.3": the library allows for the automatic generation of a GraphQL schema based on an existing Neo4j database.
- "body-parser":"1.20.2": the library is used to parse the body header.
- "cors 2.8.5": the library provides a Connect/Express middleware for handling Cross-Origin Resource Sharing (CORS) in the server.
- "express 4.18.2": the library is a middleware used to construct a server. It attaches Apollo Server to an Express server and utilizes the cors and body-parser libraries.
- "neo4j-driver 5.8.0": the library is used to facilitate communication between the server and Neo4j database.

3. Front-end:

   - React: The front-end pages are developed to establish user interactions. The GameGrid component is responsible for presenting the game list. Within the GameGrid component, there is a PlatformSelector component that displays icons representing different platforms. The NavBar component acts as the navigation bar and facilitates the updating of game data. It includes the AuthStatus component, which designs a modal form for user sign-in and sign-up. The GenreList component is used to update the games list based on the selected genres. The SortSelector component enables users to update the games list using various sorting methods. Additionally, the Infinite-Scroll-Component is implemented to limit the number of games fetched and displayed, ensuring that only the initial set of games is shown. Showed as figure 2, figure 3, and figure 4
   - Graphql Apollo client: utilized to extract data by Graphql queries from the Apollo server.
   - Dependencies:
     - @apollo/client: It is a library used to manage both local and remote data with GraphQL. It enables fetching, caching, and modifying application data while automatically updating the UI. The library provides ApolloProvider, allowing all React components to connect with the backend using Apollo Client. graphql (16.6.0): This package is used for working with GraphQL.
     - @chakra-ui/react: It is a bootstrap library that provides a set of components for easy styling and customization.
     - @emotion/react (11.10.8): This package enables simple styling in React using Emotion.
     - @emotion/styled (11.10.8): This package provides a way to create styled components using Emotion in React.
     - framer-motion (10.12.7): It is a library used for implementing animations in React.
     - zod (3.21.4): This package is used for form validation, particularly for sign-in and sign-up forms.
     - @hookform/resolvers: It is used to prepare Zod for use with React Hook Form.
     - react-hook-form: This package integrates form validation with React, working in conjunction with Zod.
     - react-infinite-scroll-component: This package allows the game page to scroll infinitely and display all games. It includes Apollo Client cache methodology to fetch and display games gradually instead of sending all games to the client at once. Initially, only the first 12 games are fetched, and then the fetchMore function from

Apollo Client and normalized cache method are used to progressively fetch and show more games to the user.

- zustand: This library is used to maintain multiple states for searching games. It provides a high-level state management solution that makes it easier to debug and implement search logic. The specific states maintained include searchString(String) for game name, genreId (String) for genre type, parentPlatformId (String) for platform, order (String) for sort order, limit (Int) set to 12, and offset (Int) set to 0.
- react-router-dom: This package is used to develop multiple pages in a React application.
- react and react-dom: These are required packages for building React applications and converting JSX to the document object model (DOM).

4 End-user experience

When the user types "http://3.13.47.159:3001" in the URL, they are presented with a beautiful main page. Users have the option to sign up or sign in using a button located in the top-left corner. After signing in, users can view the game list, which displays all the available games. If users wish to view games based on genres, they can select the desired genres from the list located on the left side. Additionally, a selection list is provided under the navigation bar for users to view games based on platforms. Users can also order the games list according to their preference by selecting the desired order criteria from the dropdown menu under the navigation bar.

5 References, Tutorials, Codebases, Documentation, and Libraries

- AWS EC2: AWS EC2 Tutorial
- React: React Tutorial
- MDN registration page: MDN Login
- Apollo Server : Apollo Server
- RAWG API: RAWG API

6 Estimated Compute Needs

AWS EC2 for deploying the project.

7 Team Roles

The following is the rough breakdown of roles and responsibilities for our team:

- Jen-Ting Chang: Front-end and Back-end Debug, AWS deployment, Report.
- Chengyu Lu: Front-end and Back-end Debug, Presentation, Video.
- Caoyi Xue: Database, Front-end, Back-end.

8 Misunderstanding before starting the project

At the beginning, we thought that neo4j is similar to relational database because they both have a lot of relations. Actually, if we want to get some special attributes of game from neo4j, we need to put them into the game node which means it is more like a document in mongodb.
At the beginning, we thought that we should use shortest path to solve the recommendation problem, but the time complexity is too high for us to consider all the multiple node path. So, we only consider the path with limited nodes which naturally has higher similarity.