CIS550 Final Report - SteamDB

Jingxuan Bao
Qihang Dai
Peihan Li
Jingru Wang

Project Introduction:

Steam is the leading gaming PC platform. Over its many years online, Steam has amassed a plethora of reviews for its games. These reviews represent an excellent opportunity to break down the satisfaction and dissatisfaction factors around games and genres, as well as sentiment over time. We want to provide a platform that helps people see the reviews of the games on steam and choose the games suiting them.

This project is a game store website that allows users to login in with an account in our database and browse games. The main page of the website displays top rating and review games filtered by tags and will enable users to search for games using the search bar. The game details page shows information about the game, with a similar game recommendation list. We also designed an algorithm to recommend games and potential friends based on the user's past behavior, this information is displayed on the profile page.

Technology:

Our project is based on the following tech stack:
1. React as the front-end framework. We also use React-Router to manage the routing of the website. We use vanilla CSS to style the website.
2. For data preprocessing, we use Python to process the data and store them in a MySQL database.
3. For the backend, we use Node.js to build the server and use Express to handle the routing of the server.
4. For the database, we use MySQL to store the data. We choose AWS RDS as the database service.

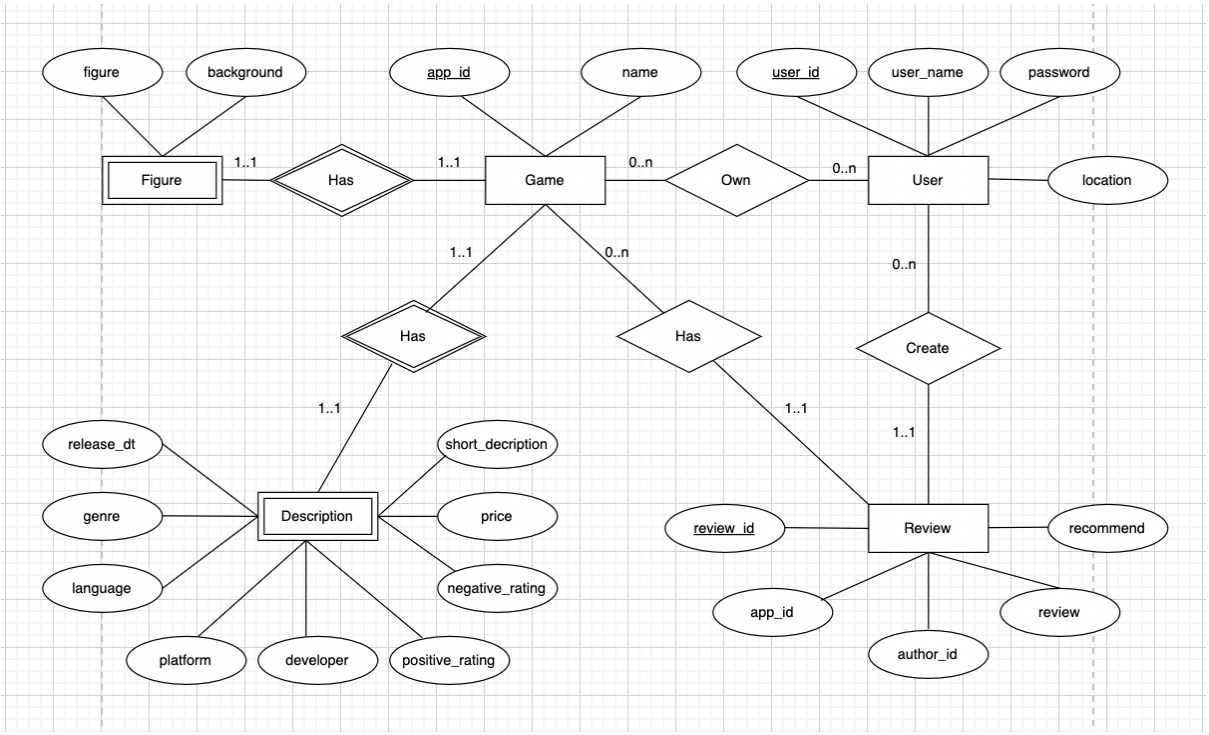
Data and Preprocess:

The first dataset we used was from Kaggle. It contains 27075 games and 18 features. The features include the game name, developer, publisher, release date, price, tags, and so on.
Link: https://www.kaggle.com/datasets/nikdavis/steam-store-games
Another dataset we used to support the game review content and user information, which is a dataset that contains more than one million review contents from different users.
Link: https://www.kaggle.com/datasets/najzeko/steam-reviews-2021

Before we import data to our database, one important step is preprocessing data. We use Pandas and Numpy library to do data wrangling and cleaning. As our dataset is quite large, we can simply drop the record which has null values. Then we drop the duplicates when it

violates our primary key constraints. Finally, we formalize the type of each attribute across different tables. As there is no user table, we extract our user data from the review table.

Database:



In this steam DB database, we have three main(strong) entities: User, Game, and Review and two weak entities: Game_Description and Game_Figure. The relationship between these three strong entities is shown in the following ER diagram: A user can have many reviews, and a review belongs to a user. Game has many reviews, and a review belongs to a game. To optimize the performance of the query, we also have a own Game table that stores game_id and user_id relationships.

As for the two weak entities: Game_Description and Game_Figure that are associated with Game. Game_Description stores the description of the game, and Game_Figure stores the figure of the game. We use the game_id as the foreign key to associate these two tables with Game.

The database is in BCNF and 3NF. There is no transitive dependency in our database. For each relation in our model(X -> A), X is always the key of a table. For example, the user_id can determine other attributes of User and there is no other functional dependency in User. We can show that each relation scheme in our database is in BCNF. Then we can prove our database is in BCNF. Because BCNF is strictly stronger than 3NF, our database is in 3NF as well.

Website Architecture:

Login Page:
Our login page provides a login form for users to log into our platform based on their unique userID. The login page will send a request to check whether the user exists, the userID logged will be kept in our platform for further game/friend recommendation.

Main Page:
Main page is the page users are directed to after log in. On the top it shows 10 random high rating games that the user may be interested in. Below are two lists of games, one shows the top 10 games that have the highest rating, and the other shows the top 10 games that have the most reviews. The user can click on the game to jump into the game detail page. Also there is a search bar which provides users to search for games by name, and a filter bar which provides users to filter games by tags, release date, and price.

Game Page:
The game page provides game details for users. Including game name, developer, publisher, release date, price, tags, and game short description. There is also a list of review content of the game. Besides these, the game page could recommend similar games for the user, based on this game's type, review, and rating information.

Profile Page:
Our profile page will provide users with the games they own. The games that are owned by the user will be displayed automatically one by one. Also, our profile page will recommend games to users based on the information of this user including the game played by the user, the living country of the user, the game commented by the user and the genres liked by the user etc. If the user clicked the picture of the game, it would jump into the corresponding game page. In the profile page, our website will also provide a list of friends the user may be interested in. The friend recommendation takes the location of two users, the common game played by the two users into consideration.

Queries and evaluation:
There are around 20 queries distributed among our 4 pages(main page, profile page, game page, and login page) to make each page function well. Among all these queries, there are 2 complex queries that will run over the 20s before optimization. These complex queries include recommending games to users on the profile page, recommending friends to users on the profile page, recommending similar games on the game page, and sorting games by a particular genre. Even if there are two game recommendation queries on our website, the recommendation algorithms are based on completely different information. So the games recommended by the queries rarely overlap.

When optimizing these queries, there are several principles that we should stick to.

1. Select specific fields rather than select *. Select attributes can reduce the optionality of our middle table, then reduce the buffers it requires.
2. Use where to filter data as early as possible.

3. Use join instead of subqueries. Join could be optimized in many systems as we discussed in the class. Sometimes the system could find a way to optimize our join especially when there are a lot of available buffers.
4. Use inner join instead of cartesian product. The result of the cartesian product could be enormous. So we should use inner join to avoid this situation.
5. Do not use DISTINCT when the uniqueness of each record is not necessary.
6. Create indexes for some attributes.
7. Use exists instead of count

Firstly, let us look at a simple example to show the most important steps of our optimization.

```
SELECT DISTINCT(app_id)
FROM REVIEW r, USER u
WHERE r.author_id = u.user_id;
```

```
SELECT app_id
FROM REVIEW r JOIN USER u ON r.author_id = u.user_id
GROUP BY app_id;
```

The two queries above do the same thing, but the difference between them is that the first one uses cartesian product while the second one uses an inner join. As our Review table has more than 1,000,000 records, the execution time of the two queries is quite different. The first query runs over 10 minutes while the second one runs about 30 seconds. With this fact, we can improve our query a lot. We followed the above rules to optimize our complex and big SQL query.

Profile page:

Game recommend in profile page:

```
WITH location_game(app_id) AS(
   SELECT app_id
   FROM  OWN_GAME o JOIN (
       SELECT user_id FROM USER t1 WHERE EXISTS (
           SELECT * FROM USER t2
           WHERE t2.user_id = '%${userid}%'
           AND t1.location = t2.location
           ) ) u ON o.user_id = u.user_id
),top_genre(genre, cnt) AS (SELECT genre, COUNT(*) AS cnt
                            FROM location_game l JOIN DESCRIPTION d ON
l.app_id = d.app_id

                        GROUP BY genre
                            ORDER BY cnt DESC
                            LIMIT 100),
```

```
   year_top100(app_id) AS (
       SELECT app_id
       FROM(
           SELECT app_id, ROW_NUMBER() over (partition by release_dt
order by positive_ratings) AS rnk
           FROM DESCRIPTION
           )t
       WHERE rnk <= 100
   ),
   candidate_game(app_id) AS (
       SELECT d.app_id AS app_id
       FROM DESCRIPTION d JOIN top_genre t ON d.genre = t.genre
       JOIN year_top100 y ON d.app_id = y.app_id
       WHERE d.app_id NOT IN (SELECT app_id FROM OWN_GAME WHERE user_id
= '%${userid}%')
   )
SELECT c.app_id AS app_id, COUNT(*) AS cnt
FROM candidate_game c JOIN REVIEW r ON c.app_id = r.app_id
GROUP BY c.app_id
ORDER BY cnt DESC
LIMIT 10;
```

This game recommendation query is based on the user's interest, the game ratings as well as the potential friend of this user. Firstly, by studying the games owned by users from different countries, we see that users' favorite game genres vary a lot among different locations. Suppose we want to recommend games to user A. In this query, we filter the users from the same location as A. Then we get the app_ids owned by people from this location and find the top genres liked by these people. Also, we want to recommend some games with high positive ratings. So we choose the yearly top 100 games with the highest positive ratings. Then we find the intersection of the two parts as our candidates. Finally, we order the candidate game by the number of reviews and display the top 10 games.

Initially, this query could run over 20s. After pushing down the where statement and using inner join, the query executes less than 1s.

Friend recommend in profile page:

```
WITH GAMELIST AS
(SELECT G.app_id
FROM OWN_GAME O
JOIN GAME G on G.app_id = O.app_id
WHERE user_id = '%${userid}%'
),  FRIENDLIST AS (
  SELECT O.user_id, COUNT(G.app_id) AS game_num
  FROM OWN_GAME O
```

```
  JOIN GAMELIST G
  GROUP BY O.user_id
  HAVING O.user_id <> '%${userid}%'
  ORDER BY game_num desc
  LIMIT 10
),  FRIENDGAMELIST AS (
  SELECT app_id
  FROM OWN_GAME O
  JOIN FRIENDLIST F ON F.user_id = O.user_id
),  FRIENDLIST2 AS (
  SELECT O.user_id, COUNT(G.app_id) AS game_num
  FROM OWN_GAME O
  JOIN FRIENDGAMELIST G ON O.app_id = G.app_id
  GROUP BY O.user_id
  HAVING O.user_id NOT IN (SELECT user_id FROM FRIENDLIST)
  AND O.user_id <> '%${userid}%'
  ORDER BY game_num desc
  LIMIT 10
),  UNIONFRIEND AS (
  SELECT user_id
  FROM FRIENDLIST
  UNION
  SELECT user_id
  FROM FRIENDLIST2
)
(SELECT * FROM UNIONFRIEND);
```

This complex SQL query recommends potential friends for the logged user, this algorithm finds friends based on the number of own games in common with the user. We believe that if you share a higher number of common games, there should be a higher percentage that you could be friends on our platform. We first picked the top 10 friends, and then employed another level of BFS to find the “friends of friends”. After the union of the friend list and “friends of friends” list, the algorithm return the friend list we recommended for the user.

Initially, this query could run about 21.673s in our 10 times DataGrip query running tests. We optimized this SQL base on the strategy we pointed out before. Like we used “Where” to filter the User own game data in the beginning, this operation saved about 17% running time in total. And also “Subquey” and “Cartesian Product” was replaced by “inner join”. Additionally, we observed the user_id and app_id in the Own_Game table were repeatedly used in the query, so we created index on these two attributes.

After the optimization, the SQL query implements less than 1s on average, and could also be immediately rendered in the profile front end page after being redirected.

Main Page:

game sort on genre:

```
        With game_description AS(
            SELECT *
            FROM DESCRIPTION
            WHERE genre like '%${genre}%'),
        game_review_count as(
            SELECT GAME.app_id, name, count(DISTINCT review_id) as
review_count
            FROM GAME, REVIEW
            WHERE GAME.app_id = REVIEW.app_id AND GAME.app_id IN (SELECT
app_id FROM game_description)
            GROUP BY app_id),
        top_games AS(
            SELECT game_description.app_id, name, short_description,
release_dt, language,
            platform, developer, genre, positive_ratings,
negative_ratings, price, average_playtime, review_count
            FROM game_description, game_review_count
            WHERE game_description.app_id = game_review_count.app_id
            ORDER BY positive_ratings DESC, review_count DESC
            LIMIT 50),
        game_picture AS(
            SELECT app_id, figure, background
            FROM FIGURE
            WHERE app_id IN (SELECT app_id FROM top_games))
        SELECT top_games.app_id, top_games.name, game_picture.figure,
game_picture.background,
        top_games.short_description, top_games.release_dt,
top_games.language, top_games.platform,
        top_games.developer, top_games.genre,
top_games.positive_ratings, top_games.negative_ratings,
        top_games.price, top_games.average_playtime,
top_games.review_count
        FROM top_games, game_picture
        WHERE top_games.app_id = game_picture.app_id
        ORDER BY positive_ratings DESC, review_count DESC;
```

The most complex SQL query in our mainpage asks the user to pass a parameter genre, sorted by rating and review_count. On the main page when a user selects a tag tab, this query would ask the database to return the best games in the selected genre that are ordered by rating first and then ordered by a number of reviews. This query can run around 3 seconds on average before optimization. However, without proper management of Join and subQuery, it can run up to 12 seconds.

Game page:

similar game recommend:

```
        SELECT distinct D.app_id,
                D.genre,
                D.short_description,
                D.positive_ratings,
                D.negative_ratings,
                F.figure,
                D.positive_ratings / (D.positive_ratings +
D.negative_ratings) as positive_ratings_percentage
        FROM DESCRIPTION D
            join FIGURE F
            on F.app_id = D.app_id
        where D.genre like concat('%', (select genre from DESCRIPTION
where app_id = '${gameid}'), '%')
        and D.average_playtime > (select average_playtime from
DESCRIPTION where app_id = '${gameid}') - 100
        and D.average_playtime > (select average_playtime from
DESCRIPTION where app_id = '${gameid}') + 100
        order by positive_ratings_percentage desc
        limit 10
```

The most complex SQL query in the game page is to implement a similar game recommendation function. In addition to recommending games that users may like on the profile page, we also design a recommendation algorithm to find similar games with that particular game on the game page. This recommendation was designed based on the particular game genre, game play time, and game rating. This query can run around 1.78 seconds on average after optimization. The whole optimization included filter target attributes at the start of the query.

Technical challenges and Future Exploration:

The first challenge we meet is in the front end: our team uses two different styles of React, one is class based and the other is functional(Hook) based. We need to combine these two styles together to make the project work. There are times when the component did not update when the state changes or fails to render the correct data. We need to debug the code to find whether the problem is in the front end or back end, or related to the SQL query.

Another one of our challenges came from the data part because we do not have a pre well designed user dataset. However, based on our ER diagram, User is an important part of our relation model. So we extract user data from Review to create the OwnGame table. According to the schema of Review, each review has an author_id and app_id. We extract

all the author_id and app_id and use DISTINCT to eliminate duplicates, then we have a new table OwnGame consisting of two attributes(app_id, user_id).

Besides the challenges we overcame successfully, we initially want to design a page for user registration. This would let visitors register their own account and could receive the recommended information exactly based on their real behaviors. But in the actual implementation of this idea, we encountered some difficulties. For example, this operation means that we need to provide registered users with additional options to purchase games and follow other users. And this means a much higher workload, and how to safely store the user's password will be another challenge. Therefore, we decided that this problem and the improvement of the above-mentioned recommendation algorithms will be the direction we are willing to explore further in the future.

References:

[1] Martin Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems
[2] C.J. Date, Database Design and Relational Theory: Normal Forms and All That Jazz
[3] Jan L. Harrington, Relational Database Design and Implementation: Clearly Explained