The Chinese University of Hong Kong, Shenzhen

Video Platform Advertising Database System

CSC3170 Database System

Project Report

May 8, 2024

Introduction

In the digital era, video platforms have emerged as powerful content dissemination and advertising tools. These platforms attract millions of users daily, offering an unparalleled opportunity for brands to reach targeted audiences. However, the effectiveness of ad recommendations on these platforms is contingent upon the sophistication of their underlying database systems.

This project aims to design a database for a video platform such as Bilibili that will enable the platform to make informed decisions about its advertisement recommendation algorithm. The database stores and manages data related to user behavior, content consumption patterns, and advertisement performance metrics. By leveraging this data, the platform can better understand its users' preferences and interests, which helps it deliver more relevant and engaging advertisements and thus increase the profits from ads.

This report will cover our work to meet the project's requirements, including relational schema and ER diagram, sample SQL queries for data mining and analysis, as well as for daily operation, data hashing/indexing, and web design.

Assumption

There are some presumptions in our data design:

1. All data are assumed to be collected within one month to keep consistent with actual cases.
2. Attributes like *finishing_rate*, *like_rate*, and *appropriate_degree* are limited between 0 and 1.
3. For the Tags, Ads, and Ad-providers entity, the data is selected and created based on reality to be the most representative. For the Users and Videos entity, the data is generated by the random number generator within a reasonable range.

Design and implementation

Relational schema

There are five entities: Users, Videos, Tags, Ads, AdProviders.

For the Users entity, we can see that each account has a *user_id* stored on the platform, performing as a primary key. There is also some basic information about users, such as *gender*, *age*, and *location*, which would be used as a yardstick for recommending the ads whenever users are new to the platform and don't have preferred tags. The attribute *time_daily_online* provides a good measurement of the user activity levels. Correspondingly, ad recommendations will focus more on active users.

For the Ads entity, the *total_click* attribute counts the volume of clicks gained from a single ad. *fee_rate* is a discrete distributed number fit for ads released with different quantities. The more ad is released, the lower fee rate they'll pay. In particular, the attribute *Profited* measures whether the ad profit has exceeded a threshold that can be considered successful. It's a bool value used for data mining. *appropriate_degree* describes the degree of similarity between tags appointed by ad providers and ad content. It's worth noticing that a foreign key *AP_id* is referenced to AdProviders.

Tags is used to specify the general type of video content. It contains two attributes: *tag_id* and *tag_name.*

AdProviders stands for providers of ads. Except for basic information, such as AP_id and AP_name, it also has an attribute quantity, meaning the total number of ads the ad providers place on the platform.

It is worth noting that all of our schemas are in BCNF because every determinant is superkey for their schemas for all non-trivial functional dependencies. This way, we reduce data redundancy and inconsistency and increase update efficiency.

ER diagram

Our ER diagram is shown below.

The users who are video uploaders have a Publishment (*user_id*, *video_id*) relationship with video, where users can publish multiple videos, and a video can have several united uploaders.

Likes (*user_id*, *video_id*) are the process that contributes to the user's preference for tags. The users who watch videos can click the Like buttons below videos, and a user can click Likes for many videos, and a video can get Likes from multiple users.

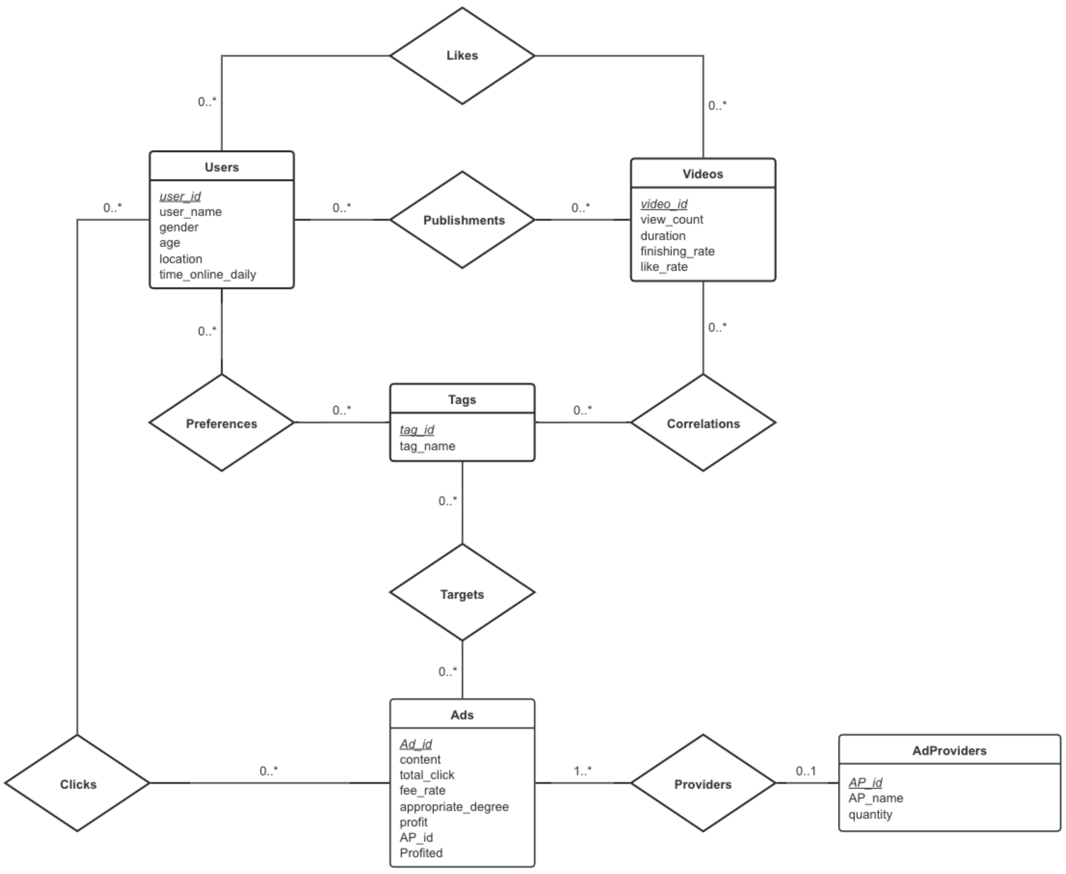
In the Users-Tags relationship Preferences (*user_id*, *tag_id*), the platform can assign users several tags according to their browsing histories. Multiple tags can be assigned to a group of users, so it's a many-to-many relationship.

The Correlations (*video_id*, *tag_id*) between Tags and Videos shows the assignment of tags on each video based on their content. A video can have multiple tags, while a tag can be assigned to a group of videos.

Users and Ads have a relationship called Clicks, which shows the number of hits the ads get in a certain period. A user can click on many ads, and an ad can get clicks from multiple users.

The relationship Target (*tag_id*, *ad_id*) describes the process that identifies the target *tag_id* of the ad, which the ad provider provides. Since an ad can be recommended to multiple tags and a tag can have attracted several ads, Target is many-to-many.

Finally, ad providers can provide the platform with many ads, but one ad can only be associated with a single ad provider, so the relationship Providers (*ad_id, ap_id*) is many-to-one.



Database implementation

We construct our database in MySQL. This database, called "myDatabase", stores the above data. Those data are generated by simulating some reasonable data from the platform with several constraints. The detailed code is in the additional zip file.

Sample SQL queries for daily operations

We implement some SQL queries for daily operations, including profile updates and searching for specific information.

User targeting:

Suppose the staff wants to find all the target users of a specific Ad (here, we use the ad with *ad_id* = 30020 as an example), which means the user's preference should overlap with the Ad's target. Then we have an SQL query like:

```
select distinct Users.user_id
from Users, Preferences
where Users.user_id = Preferences.user_id and Preferences.tag_id in
(select tag_id
from Targets, Ads
where Ads.Ad_id = Targets.Ad_id and Ads.Ad_id = 30020);
```

Profile updating:

Suppose the platform wants to adjust the fee rate of a specific ad provider:

```
UPDATE ads
SET fee_rate = "A"
where ap_id = 40001;
```

Due to some reason, one of the ad providers decide not to place ads on the platform:

```
DELETE from ads
where ap_id = 40001;
```

Due to the change in the ad content, one of the ad providers requires the platform to change the target:

```
UPDATE targets
SET tag_id = 42
where Ad_id = 30001 and tag_id = 38;
```

The original tags are not enough since there are some new trends, so the platform needs to add some new tags:

```
INSERT into Tags(tag_id, tag_name)
VALUES
(51, "Cat memes"),
(52, "Dance challenges");
```

Sample SQL queries for data analysis and mining

Moreover, we raise some SQL queries for data mining and analysis.

If we want to find the average daily online time for users of different genders, there is:

```
select gender, avg(time_online_daily) as 'average_time_online_daily'
from Users
group by gender;
```

Suppose we want to find the names of the top 10 Ad Providers with the highest cumulative clicks among the given Ads. We may write:

```
select AdProviders.AP_name, sum(Ads.total_click) as 'cumulative clicks'
from Ads, AdProviders
where Ads.AP_id = AdProviders.AP_id
group by AdProviders.AP_name
order by sum(Ads.total_click) desc
limit 10
```

Suppose we want to find the most popular video (with the highest number of likes) relating to each tag, so we have the following query. The result can help identify the popular video content and tags and facilitate labeling ads. We write:

```
select Tags.tag_name, Videos.video_id as most_liked_video_id, max_like_per_tag.max_like as 'like'
from Tags, Videos, Correlations, (
    select Correlations.tag_id as tag_id, max(Videos.view_count*Videos.like_rate) as max_like
    from Videos, Correlations
    where Correlations.video_id = Videos.video_id
    group by Correlations.tag_id) as max_like_per_tag
where Tags.tag_id = Correlations.tag_id and Videos.video_id = Correlations.video_id and
    Tags.tag_id = max_like_per_tag.tag_id and Videos.view_count*Videos.like_rate = max_like_per_tag.max_like;
```

To find the names of the top 10 users who receive the highest cumulative likes among their published videos:

```
select Users.user_name, sum(Videos.view_count*Videos.like_rate) as total_likes
from Users, Publishments, Videos
where Users.user_id = Publishments.user_id and Videos.video_id = Publishments.video_id
group by Users.user_name
order by sum(Videos.view_count*Videos.like_rate) desc
limit 10
```

The result can help the platform find and cooperate with popular video producers for advertisement.

If we aim to find the target population click rate (the percentage of target users who click the Ad) for each Ad in descending order on the rate, we can use the following queries:

```
drop view if exists Target_users;
create view Target_users as
    select distinct Preferences.user_id, Targets.Ad_id
    from Preferences, Targets
    where Preferences.tag_id = Targets.tag_id;

drop view if exists Target_counts;
create view Target_counts as
    select Target_users.Ad_id, count(*) as 'target_count'
    from Target_users
    group by Target_users.Ad_id;

drop view if exists Target_click_counts;
create view Target_click_counts as
    select Target_users.Ad_id, count(*) as 'target_click_count'
    from Clicks, Target_users
    where Clicks.user_id = Target_users.user_id and Clicks.Ad_id = Target_users.Ad_id
    group by Target_users.Ad_id;

select Target_counts.Ad_id, Target_click_counts.target_click_count/Target_counts.target_count as 'target_user_click_rate'
from Target_counts, Target_click_counts
where Target_counts.Ad_id = Target_click_counts.Ad_id
order by target_user_click_rate desc
```

In this way, we can figure out the preferences of ads for the target population. Note that we should drop the view if it exists some other early defined view to avoid nominal collision.

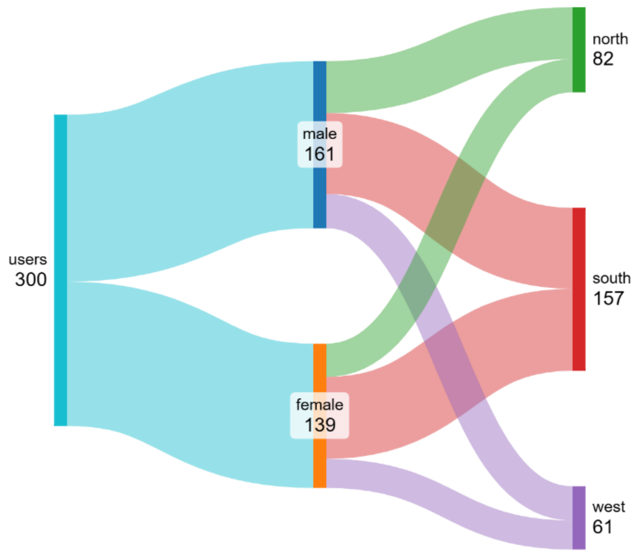
Suppose we want to find the tag associated with the population contributing the most clicks for each ad. We can have queries below. The result gives a better understanding of the decision to design the ad recommendation.

```
drop view if exists Ad_tag_cnts;
create view Ad_tag_cnts as
    select Clicks.Ad_id, Preferences.tag_id, count(*) as 'cnt'
    from Clicks, Preferences
    where Clicks.user_id = Preferences.user_id
    group by Clicks.Ad_id, Preferences.tag_id;

select Ad_tag_cnts.Ad_id, Ad_tag_cnts.tag_id
from Ad_tag_cnts
where (Ad_tag_cnts.Ad_id, Ad_tag_cnts.cnt) in (
    select Ad_tag_cnts.Ad_id, max(Ad_tag_cnts.cnt)
    from Ad_tag_cnts
    group by Ad_tag_cnts.Ad_id)
order by Ad_tag_cnts.Ad_id;
```

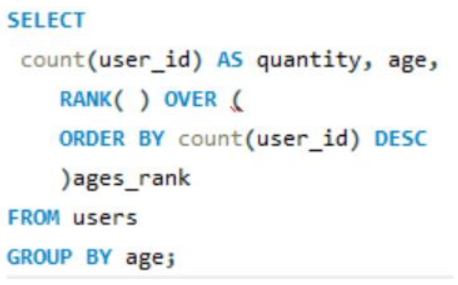
This query summarizes the users' constitutions based on their genders and locations with the help of ROLLUP (). The result can be used to analyze the user profile of our platform.

```
SELECT
    gender, location, count(user_id) as user_quantity
FROM
     users
GROUP BY
gender, location WITH ROLLUP;
```

Further, we specify the distribution of the result as a Sankey map shown below:

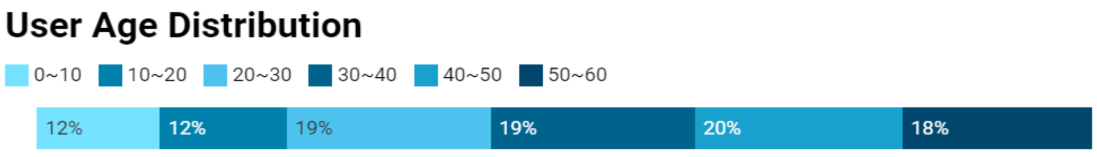


We obtain that the users have an almost equal distribution of genders. Besides, there are more users from the southern provinces of China.

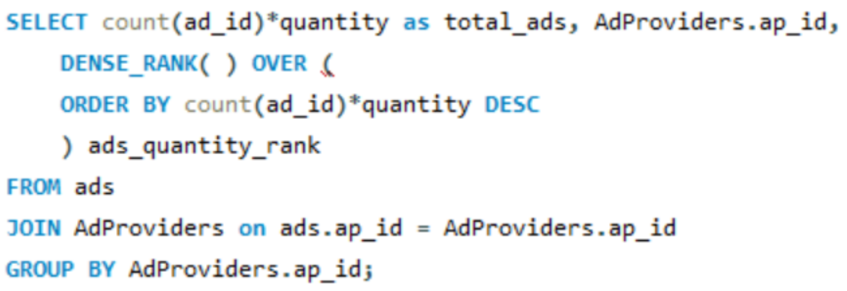


```
SELECT
 count(user_id) AS quantity, age,
    RANK( ) OVER (
    ORDER BY count(user_id) DESC
    )ages_rank
FROM users
GROUP BY age;
```

The above query shows the number of users per age and ranks the amount with the help of RANK (). Consequently, we provide the figure of distribution:

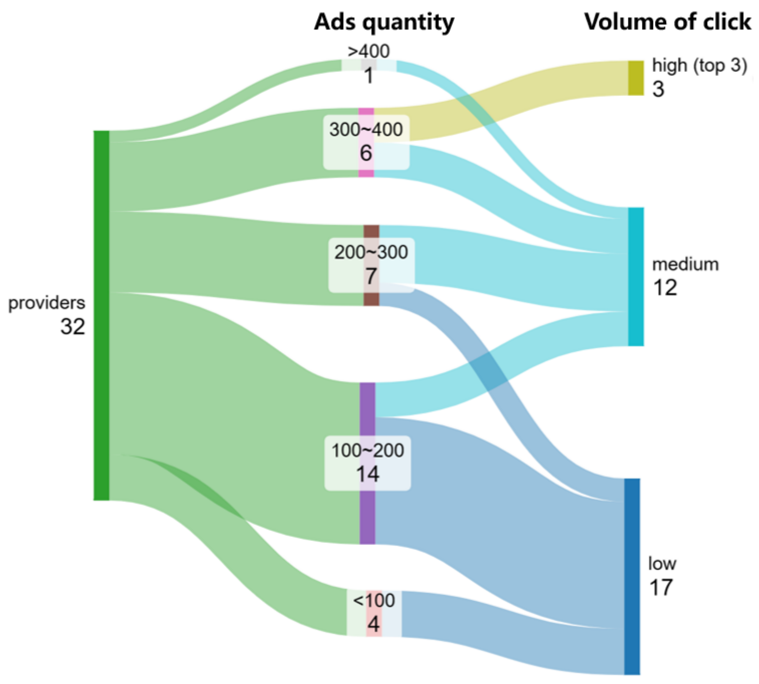


This summarized figure shows that most users' ages are intensively distributed between 20 and 50.



```
SELECT count(ad_id)*quantity as total_ads, AdProviders.ap_id,
    DENSE_RANK( ) OVER (
    ORDER BY count(ad_id)*quantity DESC
    ) ads_quantity_rank
FROM ads
JOIN AdProviders on ads.ap_id = AdProviders.ap_id
GROUP BY AdProviders.ap_id;
```

This query describes the total quantity of ads each ad provider places and ranks the amount with the help of DENSE_RANK (). The picture shows the result:



We find that:

1. The ad providers with a high volume of clicks are Mihoyo, Nongfu Spring, and Meituan.
2. Placing more ads does not lead to more clicks.

As for data mining, we chose the decision tree model using the Gini Index Criterion to train our data set and let it predict what ad content a new user tends to click based on their tag, age, gender, and location. Our accuracy of the prediction part is not so satisfying because of the limited size of the dataset.

Data hashing/indexing

Database indexing is a structural optimization for accelerating data retrieval, akin to a table of contents in a book that allows for quick navigation without a full read-through. Here, we focus on a dataset consisting of video and tag associations, analyzing the impact of indexing on the *tag_id* field to enhance query performance.

Under analysis, the dataset "Correlation" comprises records linking video_id to tag_id, where each record associates a video with a specific tag. For this analysis, a subset containing multiple instances of *tag_id* was examined.

To demonstrate the effectiveness of indexing, we simulated two query scenarios: one without an index, seeking all video IDs associated with a specific tag ID (*tag_id* = 10), and another with an index on the *tag_id* field.

Without indexing, the query operation involved a full scan of the dataset composed of *video_id* and *tag_id*. With 469 records in the table, the database had to inspect each record to locate those with *tag_id* = 10, necessitating a review of all 469 entries.

Conversely, with an index on *tag_id*, the database could directly access the records associated with *tag_id* = 10, bypassing the need to examine other entries in the dataset. This dataset contained 15 records with *tag_id* = 10. Therefore, with indexing, only these 15 records were inspected.

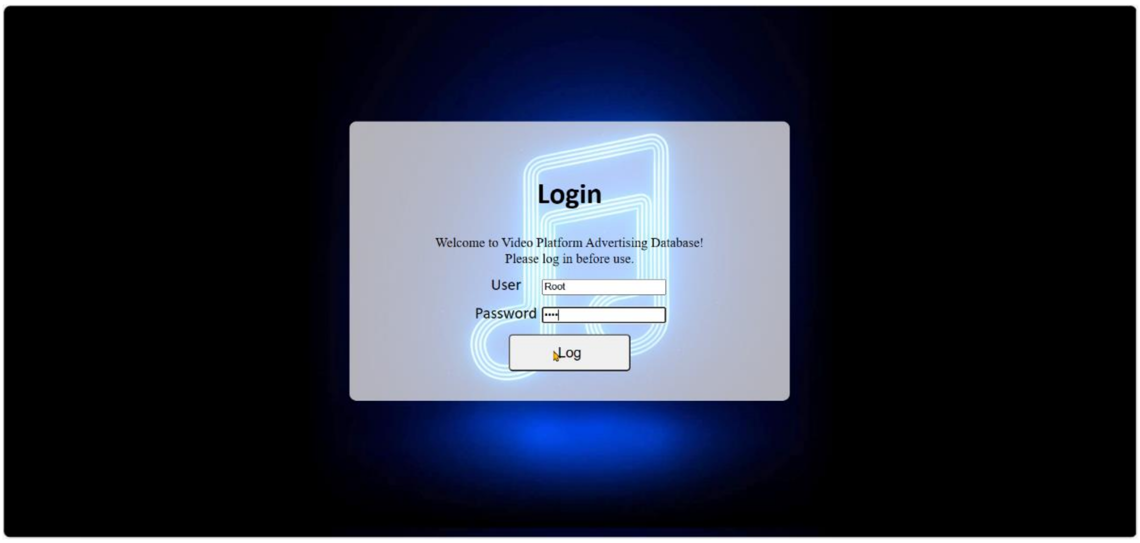
Web design

A simple website has been designed and implemented so the staff can manage our database system.

Web function introduction

The guidelines for this website are shown below.

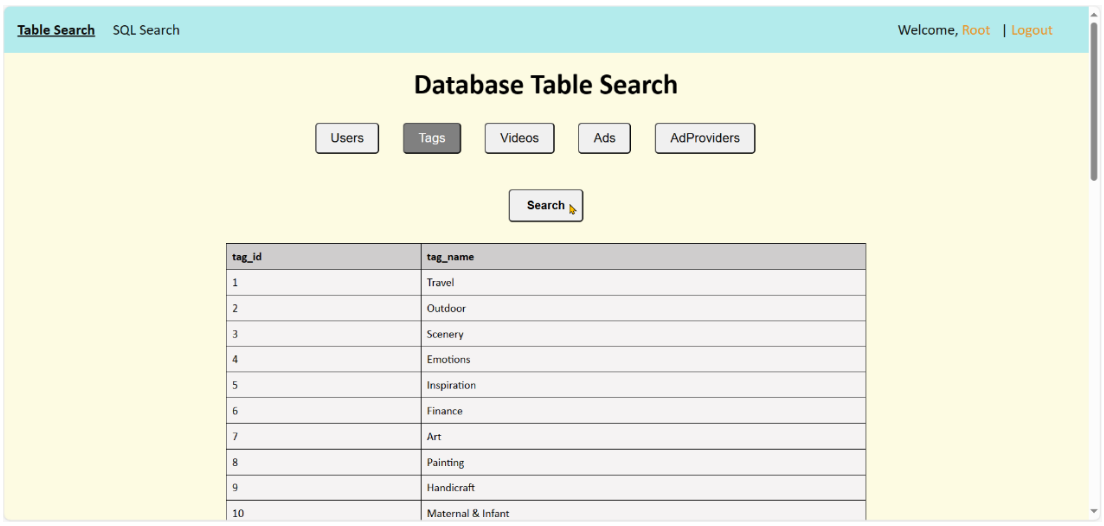
Users can type in their username and password on the home page to log in to the system.



Welcome to Video Platform Advertising Database!
Please log in before use.

Root

After logging in, users are on the Table search page. On this page, five buttons stand for five tables. Users can click on one of those buttons and click "Search", then the table they want is shown below. If the required tables don't exist, "No results found" will be returned.



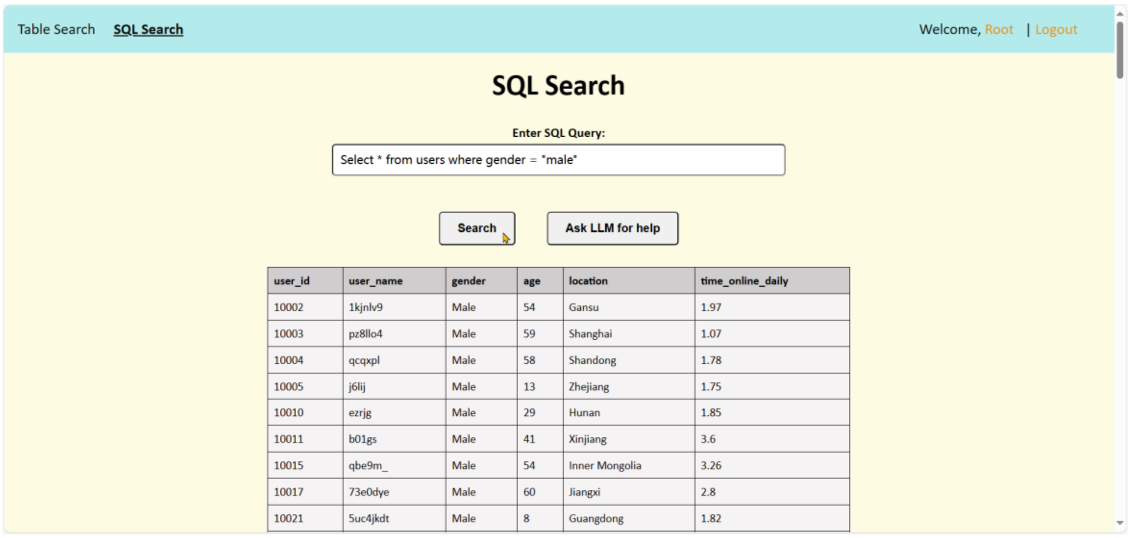
Database Table Search

tag_id	tag_name
1	Travel
2	Outdoor
3	Scenery
4	Emotions
5	Inspiration
6	Finance
7	Art
8	Painting
9	Handicraft
10	Maternal & Infant

Next, users can click on the "SQL Search" in the top left corner of the webpage and turn to a new page. Users can type SQL queries on this page to do some searching. For example, a user types in a query:

*" select * from users where gender = 'male' "*

Then the webpage shows the corresponding result:



SQL Search

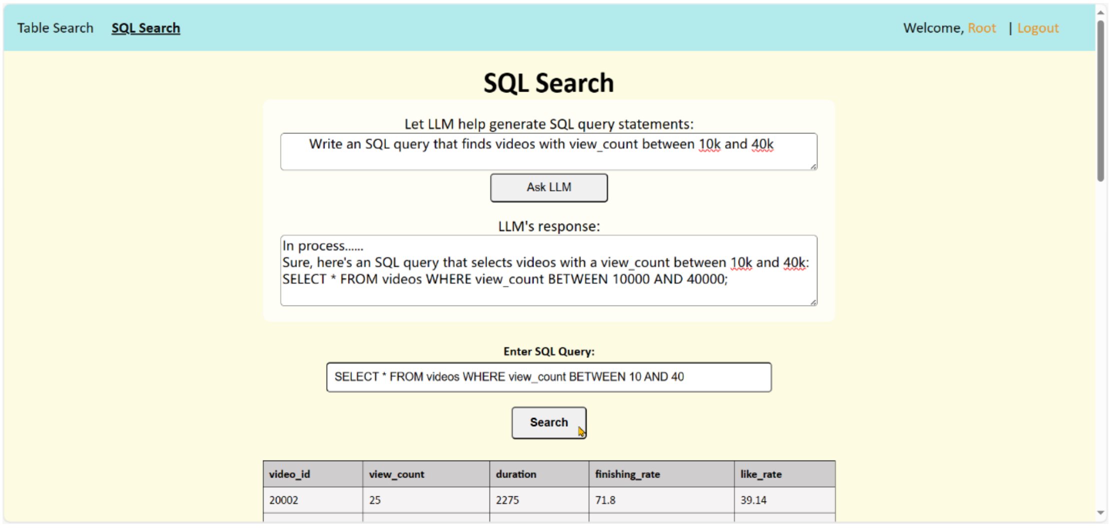
Enter SQL Query:

```
Select * from users where gender = "male"
```

user_id	user_name	gender	age	location	time_online_daily
10002	1kjnlv9	Male	54	Gansu	1.97
10003	pz8llo4	Male	59	Shanghai	1.07
10004	qcqxpl	Male	58	Shandong	1.78
10005	j6lij	Male	13	Zhejiang	1.75
10010	ezrjg	Male	29	Hunan	1.85
10011	b01gs	Male	41	Xinjiang	3.6
10015	qbe9m_	Male	54	Inner Mongolia	3.26
10017	73e0dye	Male	60	Jiangxi	2.8
10021	5uc4jkdt	Male	8	Guangdong	1.82

(This website mainly supports SQL queries as input because it's designed for data analysts and other professionals.)

For people unfamiliar with SQL queries, we also equip the database with the application of Large Language Model (LLM). After clicking "Ask LLM for help", the users enter a new page, where they can enter requirements in natural language in the first blank. The following response of LLM in SQL form will output in the middle blank. These SQL queries can be easily copied down and used for search. An example is shown below:



SQL Search

Let LLM help generate SQL query statements:

Write an SQL query that finds videos with view_count between 10k and 40k

LLM's response:

```
In process......
Sure, here's an SQL query that selects videos with a view_count between 10k and 40k:
SELECT * FROM videos WHERE view_count BETWEEN 10000 AND 40000;
```

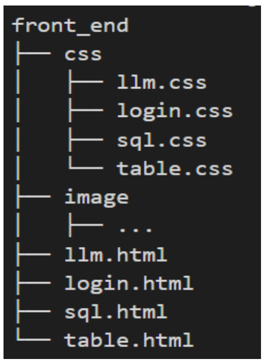
Enter SQL Query:

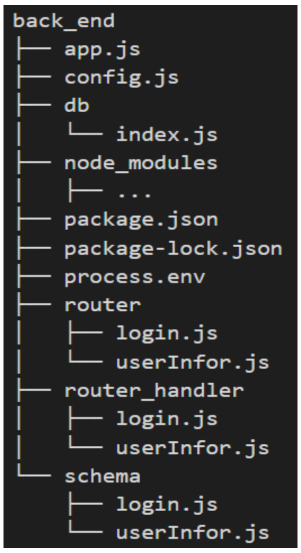
```
SELECT * FROM videos WHERE view_count BETWEEN 10 AND 40
```

video_id	view_count	duration	finishing_rate	like_rate
20002	25	2275	71.8	39.14

Web implementation code introduction

This web page uses the front and back-end separation technology to achieve this. They are implemented in the front_end and back_end folders, respectively. An overview of the structure of these two folders is shown below:


```
front_end
├── css
│   ├── llm.css
│   ├── login.css
│   ├── sql.css
│   └── table.css
├── image
│   ├── ...
├── llm.html
├── login.html
├── sql.html
└── table.html
```

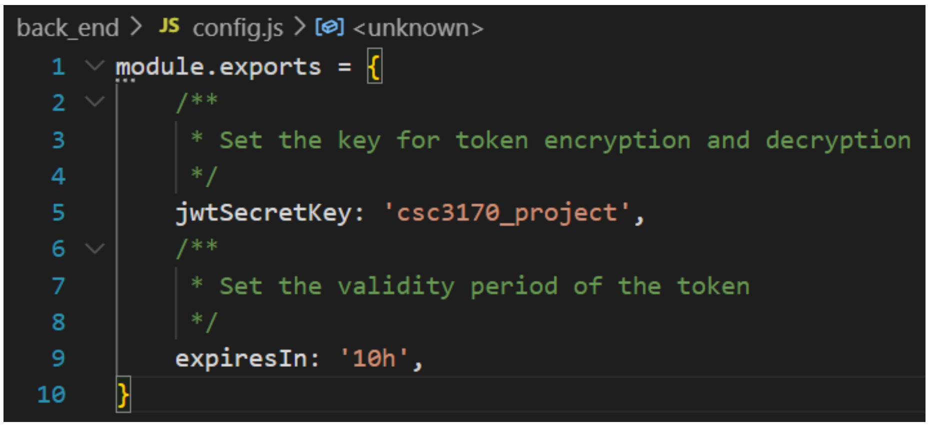

```
back_end
├── app.js
├── config.js
├── db
│   └── index.js
├── node_modules
│   ├── ...
├── package.json
├── package-lock.json
├── process.env
├── router
│   ├── login.js
│   └── userInfor.js
├── router_handler
│   ├── login.js
│   └── userInfor.js
└── schema
    ├── login.js
    └── userInfor.js
```

The implementation of the front end is relatively simple. Four html files correspond to each of the four interfaces on our web page. At the same time, the style implementation of each web page is stored in the corresponding css file.

The following function implementation introductions focus on the back-end implementation method.

Implementation of login and authentication functions

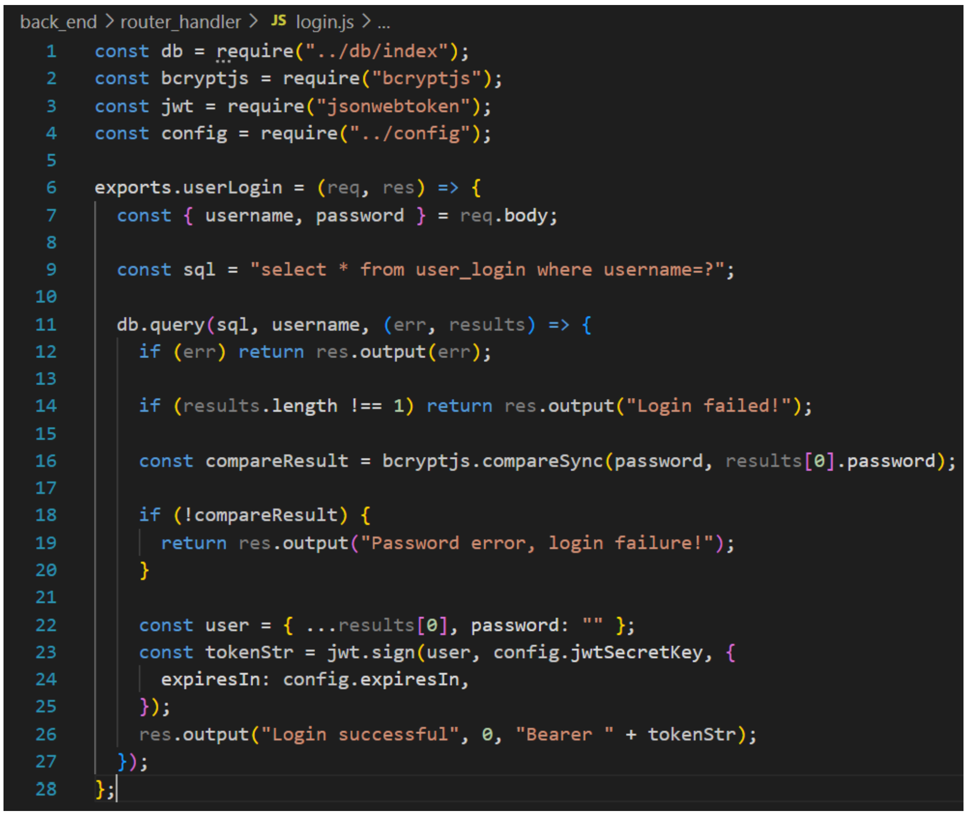
In this implementation, we will use a package *jsonwebtoken*, which generates the token. In the global configuration file config.js shown below, we set the key of the encryption method used to create the token and the token duration. This key is used for both encryption and decryption. Therefore, we have extracted the key and placed it in a separate file for convenience.



```
module.exports = {
    /**
     * Set the key for token encryption and decryption
     */
    jwtSecretKey: 'csc3170_project',
    /**
     * Set the validity period of the token
     */
    expiresIn: '10h',
}
```

The login.js file shown below contains the login handler functions. In the *userLogin* function, we first receive the user name and password from the client and then use the query statement in the database to find if there is a corresponding user name. If no, the login fails. If there is a corresponding user name, we check to see if the password is the same. The password comparison process uses the *compareSync()* method in the *bcryptjs* package. This method takes two parameters: the first is the password sent from the client, and the second is the encrypted password stored in the database, which returns true or false depending on the result. If the result is different, the login fails.

If the passwords are the same, the login succeeds. At the same time, we need to generate the token to return to the client. This step uses the *sign()* method in the *jsonwebtoken* package. It generates tokens based on the user's information and the token's duration. However, the client cannot use the generated token directly, so we also need to add the "Bearer" string in front of the token before it can be returned to the front end for direct use.

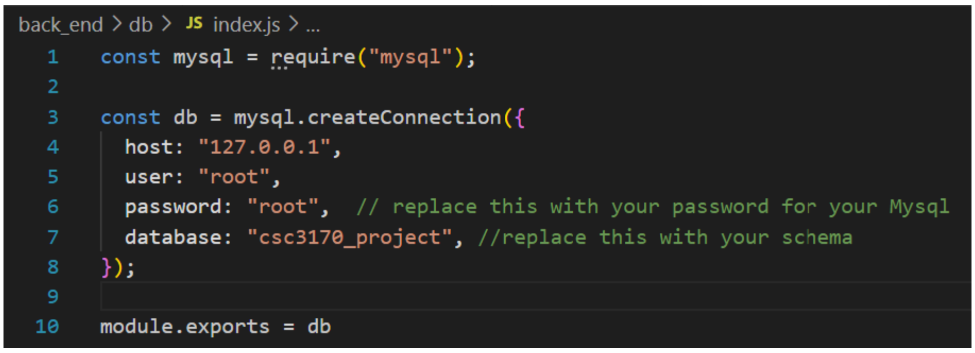


back_end > router_handler > JS login.js > ...

```javascript
const db = require("../db/index");
const bcryptjs = require("bcryptjs");
const jwt = require("jsonwebtoken");
const config = require("../config");

exports.userLogin = (req, res) => {
  const { username, password } = req.body;

  const sql = "select * from user_login where username=?";

  db.query(sql, username, (err, results) => {
    if (err) return res.output(err);

    if (results.length !== 1) return res.output("Login failed!");

    const compareResult = bcryptjs.compareSync(password, results[0].password);

    if (!compareResult) {
      return res.output("Password error, login failure!");
    }

    const user = { ...results[0], password: "" };
    const tokenStr = jwt.sign(user, config.jwtSecretKey, {
      expiresIn: config.expiresIn,
    });
    res.output("Login successful", 0, "Bearer " + tokenStr);
  });
};
```

Implementation of the web query function

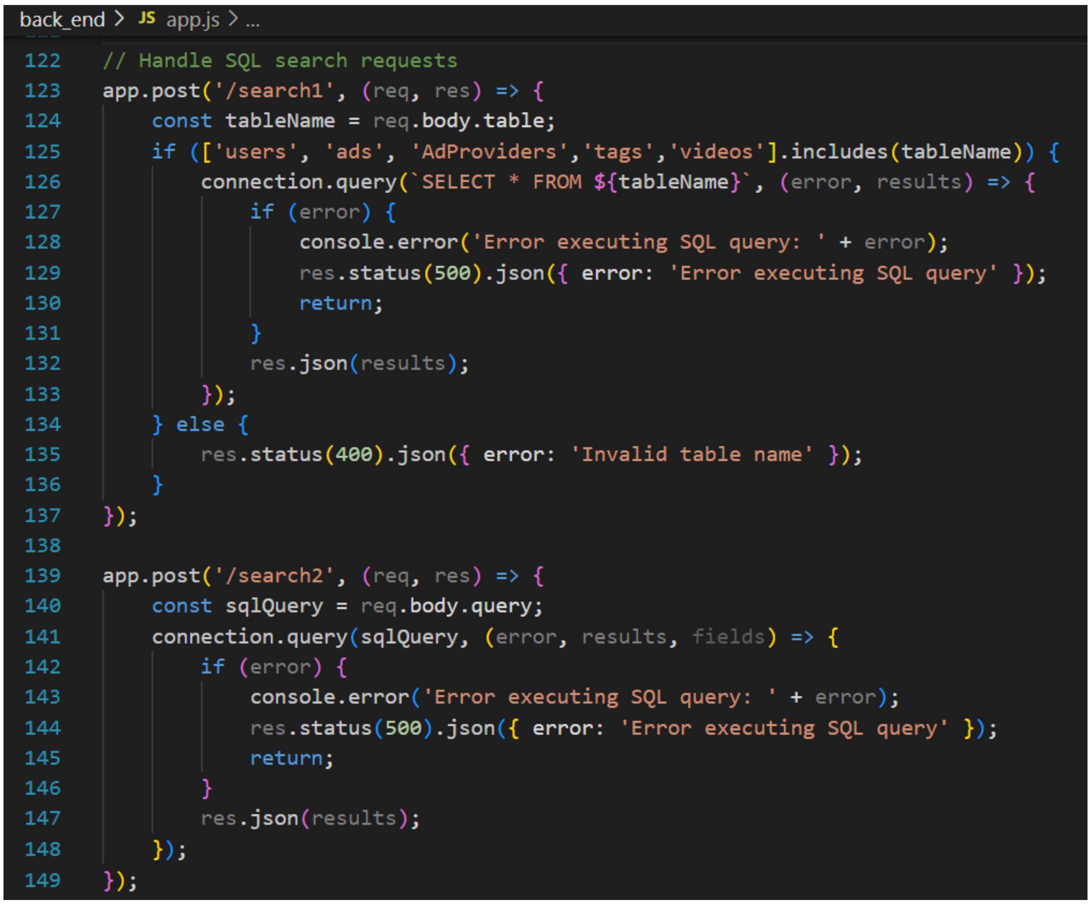
First, we need to connect to the local storage-related database via inde.js below.



back_end > db > JS index.js > ...

```javascript
const mysql = require("mysql");

const db = mysql.createConnection({
  host: "127.0.0.1",
  user: "root",
  password: "root",  // replace this with your password for your Mysql
  database: "csc3170_project", //replace this with your schema
});

module.exports = db
```

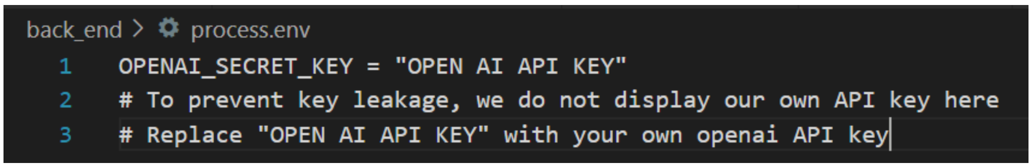
Secondly, the corresponding function of the back-end query is mainly implemented in the *search1()* and *search2()* functions of app.js. We first use *req.body()* to get these functions' field values from the front end. Then, we execute the SQL query statement using *connection.query()* and return the query results to the front end. At the same time, we check for possible errors and set the corresponding handling methods.



```
back_end > JS app.js > ...
// Handle SQL search requests
app.post('/search1', (req, res) => {
    const tableName = req.body.table;
    if (['users', 'ads', 'AdProviders','tags','videos'].includes(tableName)) {
        connection.query(`SELECT * FROM ${tableName}`, (error, results) => {
            if (error) {
                console.error('Error executing SQL query: ' + error);
                res.status(500).json({ error: 'Error executing SQL query' });
                return;
            }
            res.json(results);
        });
    } else {
        res.status(400).json({ error: 'Invalid table name' });
    }
});

app.post('/search2', (req, res) => {
    const sqlQuery = req.body.query;
    connection.query(sqlQuery, (error, results, fields) => {
        if (error) {
            console.error('Error executing SQL query: ' + error);
            res.status(500).json({ error: 'Error executing SQL query' });
            return;
        }
        res.json(results);
    });
});
```

Implementation of large language models accessing

First, we visit the OpenAI website and use the existing account to generate the API key. We will add the *dotenv* package to the project. It helps us create environment variables that we use to store our API keys. Because keys are sensitive information, we store them in the process.env file shown below to prevent them from being made public to everyone.

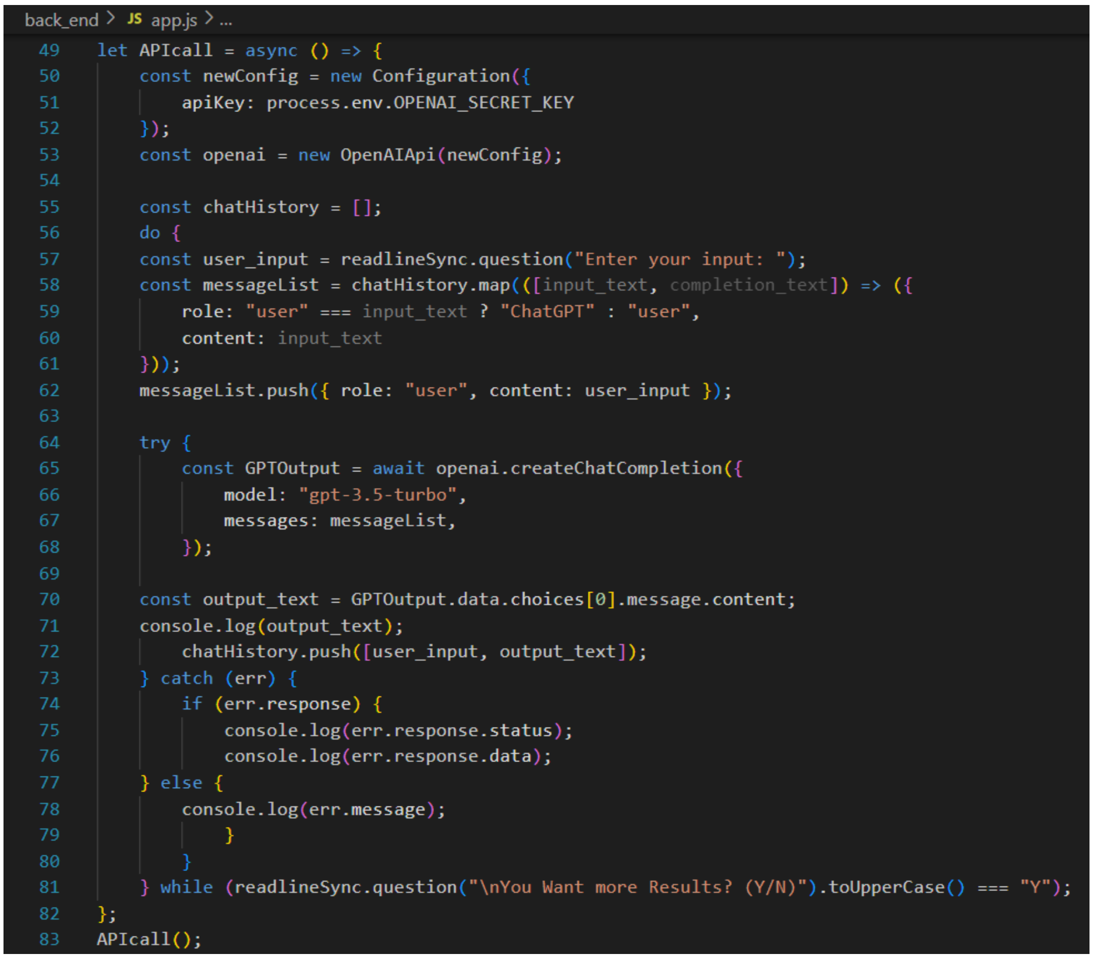


```
back_end > process.env
OPENAI_SECRET_KEY = "OPEN AI API KEY"
# To prevent key leakage, we do not display our own API key here
# Replace "OPEN AI API KEY" with your own openai API key
```

The back-end API access function is implemented in the *APIcall()* function of app.js, as shown below. In this function, we first get the API key from the environment variable in the process.env file. At the same time, we created an *OpenAIApi* class object *openai*. It accepts the configuration object that contains the API key. Using this object, we can make a request to the OpenAI API using our API key.

Next, we create a do-while loop that continuously receives user input and generates corresponding responses until the user wishes to end. Each time the loop completes an iteration, we ask the user if they want to continue the conversation. Then, we use the function *openai.createChatCompletion()* to generate OpenAI API response and use the *model* variables to specify ChatGPT model used to generate the response.

Finally, the response generated through ChatGPT will be extracted and stored in a variable and returned to the front end to be presented to the user through a web interface.



```javascript
let APIcall = async () => {
    const newConfig = new Configuration({
        apiKey: process.env.OPENAI_SECRET_KEY
    });
    const openai = new OpenAIApi(newConfig);

    const chatHistory = [];
    do {
    const user_input = readlineSync.question("Enter your input: ");
    const messageList = chatHistory.map(([input_text, completion_text]) => ({
        role: "user" === input_text ? "ChatGPT" : "user",
        content: input_text
    }));
    messageList.push({ role: "user", content: user_input });

    try {
        const GPTOutput = await openai.createChatCompletion({
            model: "gpt-3.5-turbo",
            messages: messageList,
        });

    const output_text = GPTOutput.data.choices[0].message.content;
    console.log(output_text);
        chatHistory.push([user_input, output_text]);
    } catch (err) {
        if (err.response) {
            console.log(err.response.status);
            console.log(err.response.data);
    } else {
        console.log(err.message);
            }
        }
    } while (readlineSync.question("\nYou Want more Results? (Y/N)").toUpperCase() === "Y");
};
APIcall();
```

Conclusion and self-evaluation

In this project, we designed a database for ad recommendations for a video platform. We created an ER diagram and its corresponding relational schema with normalization, which helps reduce data redundancy and avoid data quality issues. We also do hashing and indexing on our relational schemas to enhance query efficiency and accelerate data retrieval. We designed a website armed with LLM to enable viewing and selecting data in the database using SQL queries, which significantly support our efficient access to data. After that, we wrote some sample SQL queries for daily operation on the database system, as well as data analysis and data mining to ensure the consistency of our database.

Team Member and Division of Labor

Name	Student_ID	Email address	Contribution	100%
李毅然 (Coordinator)	121090291	121090291@link.cuhk.edu.cn	Presentation Sample SQL queries Report(part)	15%
姜耘逸	122090870	122090870@link.cuhk.edu.cn	Presentation Relation schemas	12%
何智涵	121090174	121090174@link.cuhk.edu.cn	Report	12%
张晓宇	121090792	121090792@link.cuhk.edu.cn	Presentation Web design Report(part)	16%
吴优	121090619	121090619@link.cuhk.edu.cn	Data generation	12%
李洪宇	122090259	122090259@link.cuhk.edu.cn	Data generation	12%
张渊铭	122040090	122040090@link.cuhk.edu.cn	Indexing/Hashing of data field	7%
杨路佳宁	121090703	121090703@link.cuhk.edu.cn	Database design Sample SQL queries	14%