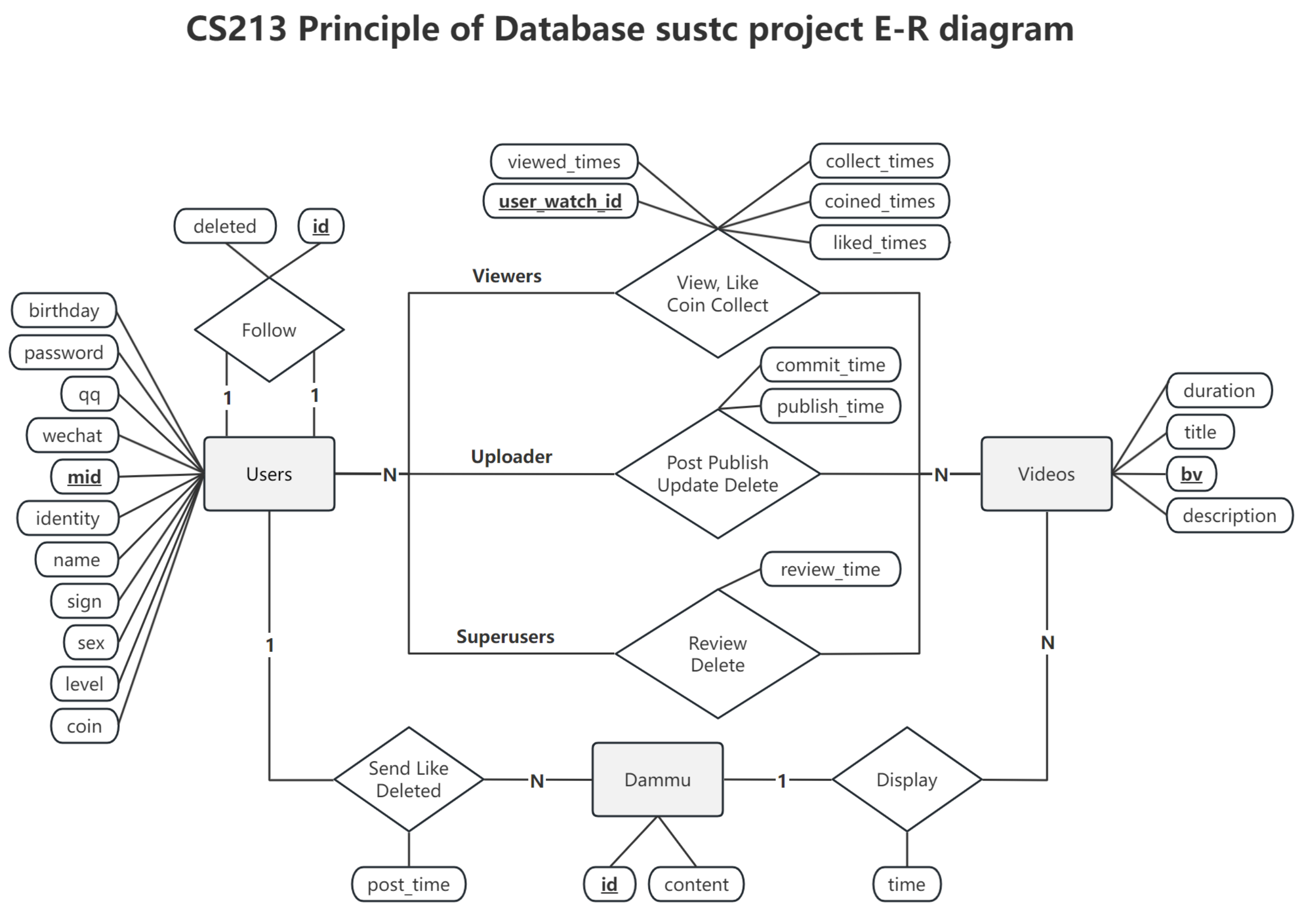
Report for CS213 Principle of Database Systems Project

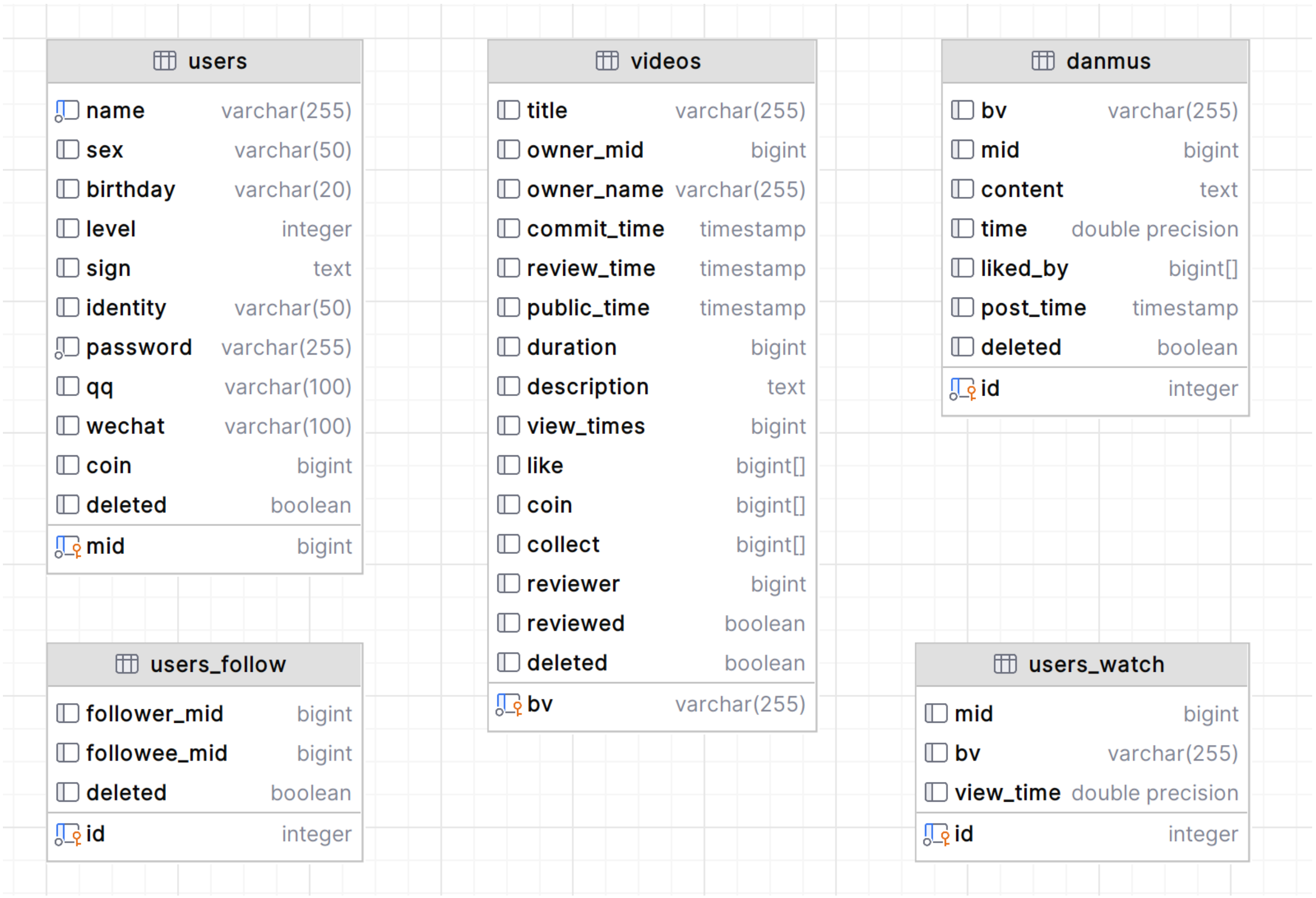
```
Personal: information
Name:  黄朗初
ID: 12213009
```

1. Database design

1.1 E-R diagram



1.2 Database diagram



users	
name	varchar(255)
sex	varchar(50)
birthday	varchar(20)
level	integer
sign	text
identity	varchar(50)
password	varchar(255)
qq	varchar(100)
wechat	varchar(100)
coin	bigint
deleted	boolean
mid	bigint

videos	
title	varchar(255)
owner_mid	bigint
owner_name	varchar(255)
commit_time	timestamp
review_time	timestamp
public_time	timestamp
duration	bigint
description	text
view_times	bigint
like	bigint[]
coin	bigint[]
collect	bigint[]
reviewer	bigint
reviewed	boolean
deleted	boolean
bv	varchar(255)

danmus	
bv	varchar(255)
mid	bigint
content	text
time	double precision
liked_by	bigint[]
post_time	timestamp
deleted	boolean
id	integer

users_follow	
follower_mid	bigint
followee_mid	bigint
deleted	boolean
id	integer

users_watch	
mid	bigint
bv	varchar(255)
view_time	double precision
id	integer

1.3 Database Schema Description

`users` Table

- `mid`: **Primary key**. A unique bigint identifier for the user.
- `name`: Stores the full name of the user, a string up to 255 characters.
- `sex`: Stores the sex of the user, a string up to 50 characters. Contains (男， 女，未知).
- `birthday`: Stores the user's birthday, likely in '?月 ? 日' format, up to 20 characters.
- `level`: Stores the user's level as an integer, indicating their rank or status.
- `sign`: Stores additional textual information about the user.
- `identity`: Stores a unique identifier for the user. Contains ("USER", "SUPERUSER", "UNKNOWN").
- `password`: Stores the user's password, a hashed string up to 255 characters.
- `qq`: Stores the user's QQ messenger ID, a string up to 100 characters.
- `wechat`: Stores the user's WeChat ID, a string up to 100 characters.
- `coin`: Stores the number of virtual coins or currency the user has, as a bigint.
- `deleted`: Stores a boolean value indicating whether the user's account is deleted.

`users_follow` Table

- `id`: **Primary key**. Auto generated integer identifier for the follow relationship.
- `follower_mid`: Bigint identifier for the follower in a user relationship.
- `followee_mid`: Bigint identifier for the followee in a user relationship.
- `deleted`: Boolean indicating if the follow relationship is deleted.

`videos` Table

- `bv`: **Primary key**. a string up to 255 characters.
- `title`: Stores the title of the video, a string up to 255 characters.
- `owner_mid`: Bigint identifier of the user who owns the video.
- `owner_name`: Name of the user who owns the video, a string up to 255 characters.
- `commit_time`: Timestamp of when the video was committed/created.
- `review_time`: Timestamp of when the video was reviewed.
- `public_time`: Timestamp of when the video was made public.
- `duration`: Length of the video as a bigint.
- `description`: Text description of the video content.
- `view_times`: Stores the number of times the video has been viewed, as a bigint.
- `like`: Array of bigints indicating users who liked the video.
- `coin`: Array of bigints indicating users who coined the video.
- `collect`: Array of bigints indicating users who collect the video.
- `reviewer`: Bigint identifier for the user who reviewed the video.
- `reviewed`: Boolean indicating if the video has been reviewed.
- `deleted`: Boolean indicating if the video has been deleted.

`dannmus` Table

- `id`: **Primary key**. Integer identifier for the dannmu.
- `bv`: Unique identifier string for the related video, up to 255 characters.
- `mid`: Bigint identifier for the user.
- `content`: The main text content of the dannmu.
- `time`: Double precision timestamp for when the dannmu was posted in video.
- `liked_by`: Array of bigints indicating which users liked the dannmu.
- `post_time`: Timestamp of when the dannmu was posted.
- `deleted`: Boolean indicating if the dannmu has been deleted.

`users_watch` Table

- `id`: **Primary key**. Integer identifier for the watch record.
- `mid`: Bigint identifier for the user.
- `bv`: Unique identifier string for the video, up to 255 characters.
- `view_time`: Double precision timestamp for when the video was watched.

1.3 Design ideas

When I was designing the database, I was thinking about the following **questions**:

- Should I use a foreign key to connect my tables?
- Should the N-N relationship be in a separate table?
- Why I should add a delete tag instead of deleting the record directly?
- How to generate the `bv` and `mid` for registering and posting, ensuring the uniqueness and the efficiency?

After looking for some information, I have the following **solutions**:

- Alibaba Java rules contains that the foreign key should NOT be used in database and the relation should be handled in the application layer. To avoid the deadlock problem and reduce the constrain for developing, I removed all the foreign keys in my database.
- Both separated table and in-table ways to handle N-N relationship are OK. But for complex and detailed data, the separated table can be more efficient and ordered. So I only extract the `follow` and `view` relationship into separated tables.`
- For data security, we should not delete the record directly. Instead, we should add a `deleted` tag to mark the record as deleted. And for the continuous growth of index, which increase the performance of the database, we should use a delete tag to mark delete the record.
- Here are the algorithms I used to generate the primary key:
  - `mid`:
  - `bv`：`String bv = UUID.randomUUID().toString().replace("-", "");`. The bv is generated by UUID. It's a 32-bit string. And it's unique. So it can be used as the primary key.
  - `id`: `id SERIAL PRIMARY KEY`.Generate by postgresql automatically. It's not mandatory to add the serial id. But it can be used to improve the performance of the database. And it's also convenient for us to get the latest record.

2. Basic API Specification

2.1 DatabaseService

`importData`

1. **User Data Import**

- Inserts user data into `users` table.
- Inserts following relationships into `users_follow` table.
- Uses batch processing for efficiency, executing batches every 5000 users and 100 followings.
- Provides progress updates on the console.
- Catches `SQLException` and may roll back the transaction.

1. **Video Data Import**

- Inserts video data into `videos` table.
- Inserts view records into `users_watch` table.
- Handles arrays for `coin`, `like`, and `collect` fields using `conn.createArrayOf`.
- Executes batches every 10 videos and 2000 views.
- Provides progress updates on the console.
- Catches `SQLException`, rolls back the transaction, and throws a new exception with a message.

1. **Danmu Data Import**

- Inserts danmu records into `danmus` table.
- Handles arrays for `liked_by` field.
- Executes batches every 100 danmus.
- Provides progress updates on the console.
- Catches `SQLException`, rolls back the transaction, and throws a new exception with a message.

`truncate`

```sql
DO $$
DECLARE r RECORD;
BEGIN
FOR r IN (SELECT tablename FROM pg_tables WHERE schemaname = 'public')
LOOP
EXECUTE 'TRUNCATE TABLE ' || quote_ident(r.tablename) || ' CASCADE;';
END LOOP;
END $$;
```

This script is designed to truncate all tables within the 'public' schema in a PostgreSQL database. It uses an anonymous code block (`DO $$ ... $$;`) and dynamic SQL within a loop.

2.2 UserService

`long register(RegisterUserReq req)`

1. **Initialization:**
   - Retrieves the highest `mid` from `users` table on the first call.
   - Sets `maxMid` to this value plus one.
2. **Validation:**
   - Checks if the user already exists and for non-null, non-empty `name`, `password`, and `sex`.
   - Validates `birthday` format if provided.
3. **User Creation:**
   - Increments `maxMid` within a synchronized block for a unique ID.
   - Inserts new user details into the `users` table via SQL.
4. **Outcome:**
   - Returns the new `mid` on success.
   - On failure, returns an error code or throws an exception.

`boolean deleteAccount(AuthInfo auth, long mid)`

1. **Authentication Validation:**
   - Checks if the provided `AuthInfo` object is valid by calling `isAuthValid`.
   - Retrieves the deletion status of the account associated with the provided `mid`.
2. **Authorization Check:**
   - Ensures that the requester has the right to delete the account.
   - For normal users (`id == 1`), they can only delete their own account.

- For administrators (`id == 2`), they must not attempt to delete an already deleted account or an account not associated with their `AuthInfo`.

3. **SQL Execution:**

   - Prepares an SQL statement to set the `deleted` flag to `true` for the given `mid`.
   - Executes the update and checks if any rows were affected.

4. **Outcome:**

   - Returns `true` if the account was successfully marked as deleted.
   - Returns `false` if no rows were affected, indicating that the account could not be found or was already deleted.
   - Throws a `RuntimeException` if there's an `SQLException`, indicating a failure in the delete operation.

`boolean follow(AuthInfo auth, long followeeMid)`

1. **Validation:**

   - Retrieves the requester's `mid` using the provided `AuthInfo`.
   - Prevents self-following by comparing `mid` with `followeeMid`.
   - The commented line suggests a check for the existence of the followee user, which is currently not enforced.
   - Verifies the validity of the `AuthInfo`.

2. **Follow Action:**

   - Prepares an SQL statement to insert a new follow relationship into `users_follow` table.
   - Sets the `follower_mid` to the requester's `mid` and `followee_mid` to the target user's `mid`.

3. **Execution and Outcome:**

   - Executes the insert operation.
   - Returns `true` if the insertion is successful and affects at least one row, indicating a new follow relationship has been created.
   - Returns `false` if no rows were affected, indicating the follow operation did not take place (e.g., due to a duplicate entry).
   - Throws a `RuntimeException` if an `SQLException` occurs, indicating a failure in the follow operation.

`UserInfoResp getUserInfo(long mid)`

1. **User Existence Check:**

   - Initially commented out, there's a check to verify if the user exists using the `isUserExist` method. If the user does not exist, it would return `null`.

2. **Data Retrieval:**

   - A `UserInfoResp` object is created to store the user's information.

- Multiple SQL queries are constructed to retrieve different aspects of user data:
  - Basic user information from the `users` table.
  - Followers and followings from the `users_follow` table.
  - Videos watched from the `users_watch` table.
  - Videos liked and collected from the `videos` table.
  - Videos posted by the user.

3. **Data Processing:**

   - Executes the first query to get the user's basic information.
   - If the user exists, it sets the basic information and then fetches relational data like followers, followings, videos watched, liked, and collected using helper methods.
   - If no records are found, returns `null`.

4. **Helper Methods:**

   - `fetchUserRelations`: Fetches and returns user relationship data (followers or followings) as an array of `long`.
   - `fetchUserWatch`: Fetches and returns the list of videos watched by the user as an array of `String`.
   - `fetchVideoRelations`: Fetches and returns the list of videos liked or collected by the user as an array of `String`.

5. **Error Handling:**

   - Catches `SQLException` and throws a `RuntimeException` with a message indicating failure in retrieving user information.

6. **Return Value:**

   - Returns a populated `UserInfoResp` object if the user exists and data is successfully retrieved.
   - Returns `null` if the user does not exist or no information is available.

2.3 VideoService

`String postVideo(AuthInfo auth, PostVideoReq req)`

1. **Validation**

   - Verifies user authentication using `isAuthValid`. Returns `null` if authentication fails.
   - Checks if the request (`req`) is valid through `isReqValid`. Returns `null` if invalid.

2. ** Duplication Check**

   - Queries the `videos` table to check if a video with the same title by the same user already exists.
   - Returns `null` if a duplicate is found.

3. **Video Insertion**

   - Generates a unique identifier (`bv`) for the video using `UUID.randomUUID()`.
   - Prepares and executes an SQL `INSERT` statement to add the new video to the `videos` table.
   - Includes video details like title, description, duration, commit time, public time, and `bv`.

`boolean deleteVideo(AuthInfo auth, String bv)`

1. **Authentication Validation:**
   - Validates the user's authentication using `isAuthValid`. Returns `false` if authentication fails.
2. **Deletion Permission Check:**
   - Checks if the user has the permission to delete the specified video through `isDeleteValid`. Returns `false` if the deletion is not valid.
3. **Video Deletion:**
   - Prepares and executes an SQL `UPDATE` statement to mark the video as deleted (`deleted = true`) in the `videos` table, using the provided `bv`.

`boolean updateVideoInfo(AuthInfo auth, String bv, PostVideoReq req)`

1. **Authentication and Request Validation:**
   - Checks user authentication using `isAuthValid`. Returns `false` if authentication fails.
   - Verifies the validity of the update request (`req`) using `isReqValid`. Returns `false` if the request is invalid.
2. **Video Ownership and Content Check:**
   - Queries the `videos` table to verify if the video with the specified `bv` exists and matches the requester's details.
   - Ensures that the requester is the owner of the video.
   - Checks if the new duration, title, and description are different from the existing ones. Returns `false` if there are no changes.

`List<String> searchVideo(AuthInfo auth, String keywords, int pageSize, int pageNum)`

1. **Authentication and Parameter Validation:**
   - Validates user authentication and checks if input parameters (`keywords`, `pageSize`, `pageNum`) are valid.
2. **Search Criteria and Execution:**
   - Searches videos based on keywords in the title, description, or owner's name.
   - Applies different visibility criteria for administrators and regular users.
3. **Result Ranking and Pagination:**
   - Ranks search results based on keyword relevance and view counts.
   - Implements pagination to return results according to `pageSize` and `pageNum`.

`double getAverageViewRate(String bv)`

1. **BV Validation and Video Existence Check:**

    - Validates the `bv` parameter and checks if the corresponding video exists.

2. **Data Retrieval:**
    - Retrieves view count and duration for the specified video.

3. **Average View Rate Computation:**
    - Calculates the average view rate based on total view duration and video metrics.

`Set<Integer> getHotspot(String bv)`

1. **Video Validation:**
    - Validates the `bv` parameter and ensures the video exists.

2. **Data Gathering:**
    - Retrieves danmu (comment) timestamps and video duration.

3. **Hotspot Identification:**
    - Groups danmus into 10-second intervals and finds the most active intervals.
    - Returns these intervals as hotspots in chronological order.

`boolean reviewVideo(AuthInfo auth, String bv)`

1. **User Privilege and Video Existence Check:**
    - Verifies administrative privileges and the existence of the specified video.

2. **Review Eligibility Check:**
    - Checks if the video has already been reviewed.

3. **Review Update:**
    - Marks the video as reviewed, updating its review time and reviewer.
    - Returns `true` for a successful review update.

`boolean coinVideo(AuthInfo auth, String bv)`

- Reference the `reviewVideo` method.

`boolean likeVideo(AuthInfo auth, String bv)`

- Reference the `reviewVideo` method.

`boolean collectVideo(AuthInfo auth, String bv)`

- Reference the `reviewVideo` method.

2.4 DanmuService

`long sendDanmu(AuthInfo auth, String bv, String content, float time)`

1. **User Authentication and Video Validation:**
   - Checks if the user is authenticated and if the specified video exists.
2. **User Watch History Check:**
   - Verifies that the user has previously watched the video.
3. **Danmu Timing Validation:**
   - Ensures the danmu posting time is within the video's duration.
4. **Danmu Insertion:**
   - Inserts the new danmu into the `danmus` table.
   - Returns the generated ID of the newly inserted danmu.

`List<Long> displayDanmu(String bv, float timeStart, float timeEnd, boolean filter)`

1. **Input Validation:**
   - Checks if the `bv` (video ID) is valid and not empty.
   - Validates the `timeStart` and `timeEnd` range, ensuring they are within a valid range and the start is less than the end.
2. **Video Existence and Duration Check:**
   - Verifies if the video exists and if the specified time range is within the video's duration.
3. **Danmu Retrieval:**
   - Retrieves danmu IDs from the `danmus` table for the specified video and time range.
   - Optionally applies a filter to the retrieved danmu records.
4. **Result Compilation:**
   - Returns a list of danmu IDs that match the criteria.

`boolean likeDanmu(AuthInfo auth, long id)`

1. **User Authentication:**
   - Validates the user's authentication status.
2. **Danmu ID Validation:**
   - Checks if the danmu ID is positive and valid.
3. **Video Watch Verification:**
   - Ensures that the user has watched the video associated with the danmu.

4. **Like Status Toggle:**
   - Toggles the like status of the danmu for the authenticated user.
   - Prevents liking if the user has already liked the danmu.
   - Updates the `danmus` table to reflect the new like status.

2.5 RecommenderService

`List<String> recommendNextVideo(String bv)`

1. **Video Existence Check:**
   - Verifies if the provided `bv` corresponds to an existing video.
2. **Recommendation Query Execution:**
   - Executes an SQL query to find videos with shared viewers to the input video.
   - Limits the result to the top 5 videos based on shared viewers count.
3. **Result Processing:**
   - Collects and sorts the recommended video IDs (`bv`).
   - Returns a list of these IDs as recommendations.\

`List<String> generalRecommendations(int pageSize, int pageNum)`

1. **Parameter Validation:**
   - Checks the validity of `pageSize` and `pageNum`.
2. **Recommendation Score Calculation:**
   - Executes an SQL query to calculate a recommendation score based on multiple factors like likes, coins, collections, danmu count, and average finish time.
3. **Pagination and Result Collection:**
   - Applies pagination based on `pageSize` and `pageNum`.
   - Collects and returns the video IDs (`bv`) as general recommendations.

`List<String> recommendVideosForUser(AuthInfo auth, int pageSize, int pageNum)`

1. **Authentication and Parameter Validation:**
   - Validates user authentication and checks `pageSize` and `pageNum`.
2. **Interest-Based Video Selection:**
   - Fetches videos related to the user's interests, focusing on content from followed and following users.
3. **Unwatched Video Filtering:**

   - Filters out videos already watched by the user.

4. **Video Ranking and Pagination:**

   - Ranks videos based on popularity among the user's network and video owner's level.
   - Applies pagination to the sorted list of recommendations.

5. **Result Generation:**

   - If no specific recommendations are found, defaults to general recommendations.
   - Returns a list of recommended video IDs (`bv`).

`List<Long> recommendFriends(AuthInfo auth, int pageSize, int pageNum)`

1. **Authentication and Parameter Validation:**

   - Validates user authentication and checks the validity of `pageSize` and `pageNum`.

2. **Friend Recommendation Query:**

   - Executes an SQL query to identify potential friend recommendations based on mutual video watch history and user interactions.

3. **Data Processing and Pagination:**

   - Processes the query results to identify unique user IDs that align with the friend recommendation criteria.
   - Applies pagination based on `pageSize` and `pageNum`.

4. **Result Generation:**

   - Returns a list of user IDs as friend recommendations.

3. Advanced APIs and Other Requirements

3.1 Speed

1. **Index**

For the frenquent query, I add the index for the table. Which slower the import speed but faster the query speed.

```sql
-- index
CREATE INDEX uploader_index ON videos(owner_mid);
CREATE INDEX danmus_index ON danmus(bv);
CREATE INDEX follower ON users_follow(follower_mid);
CREATE INDEX followee ON users_follow(followee_mid);
CREATE INDEX watch_index ON users_watch(bv);
CREATE INDEX watch_mid on users_watch(mid);
```

Used multi-table queries and added indexes to calculate the average view rate of videos. This improves the efficiency and accuracy of queries and leverages the database optimization features.

2. **Batch insert and import**

3. **Select the minimum number of columns**

3.2 Transaction

1. **Genrate id**

In `register` method, I use the `synchronized` to ensure the uniqueness of the `mid`.

```
    private static long maxMid = -1;
            synchronized (UserServiceImpl.class) {
                midValue = UserServiceImpl.maxMid++;
            }
```

This method is not the best way to generate the id. But it's simple and easy to implement. And it's also thread-safe.

2. **@Async annotation**

Implemented asynchronous queries using the @Async annotation. This improves efficiency and concurrency while reducing thread blocking and waiting times. Implemented transaction management and rollback mechanisms to ensure data consistency and stability. This helps prevent data loss or errors.

3.3 Security

1. use `PreparedStatement` to prevent SQL injection
2. security check the string input before insert into the database
3. use `synchronized` to ensure the uniqueness of the `mid`
4. use `deleted` tag to mark the record as deleted instead of deleting the record directly, which can avoid the deadlock problem and reduce the constrain for developing.