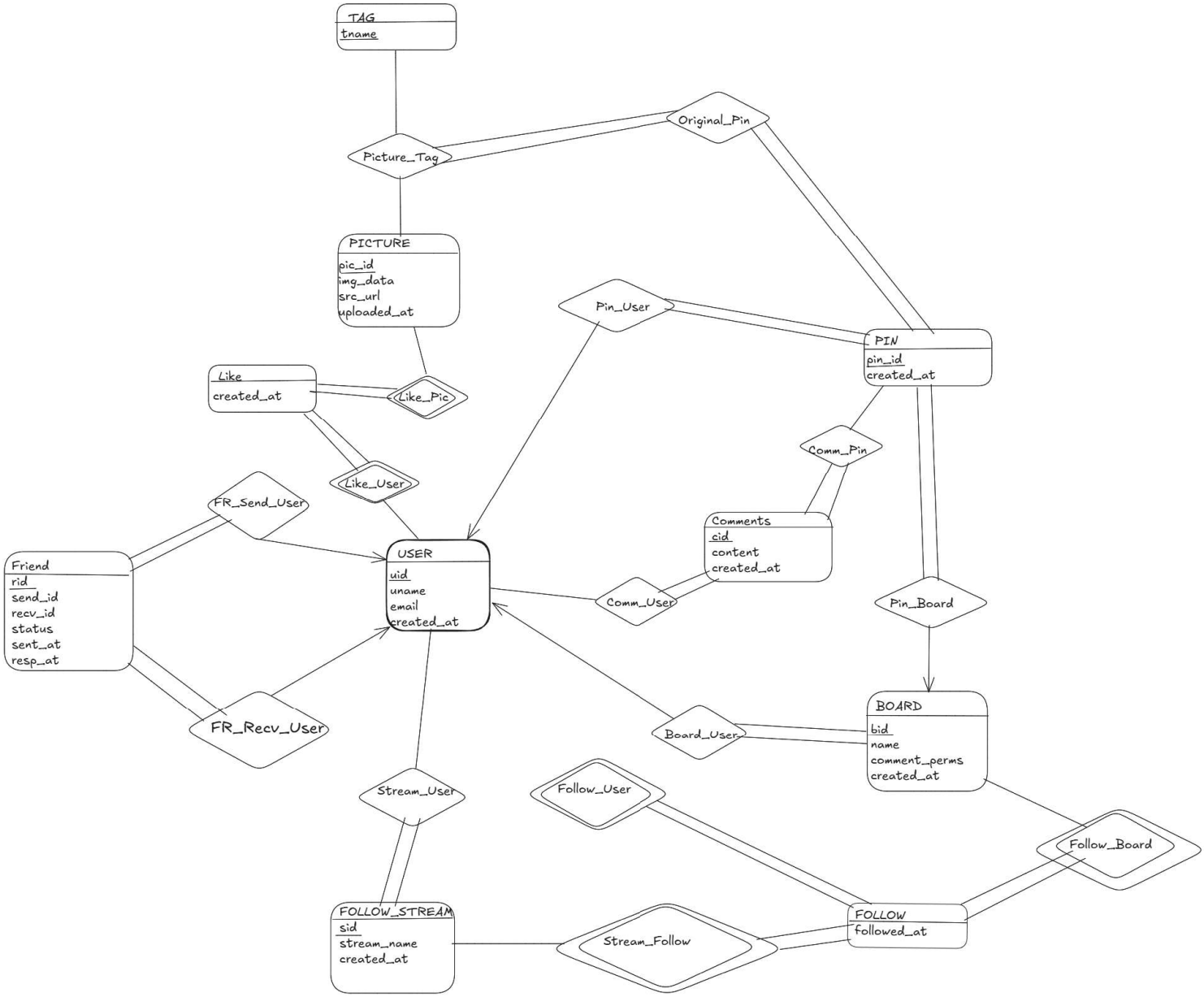
This paper will describe are implementation of the backend and frontend of our Pinterest-like social media website. We began by desiging the backend with an ER diagram. At the center of our ER diagram is the User entitiy. This stores a user's unique id as well as their personal info and account creation time. User has relationship with Board. Board must have total participation in User as every Board needs to be associated with a user. A user can create many boards but a board is only associated with one user. Similarly, the entitiy Pin has total participation in Board. A pin represents an image that the user has pinned to one of their boards. Each pin must be associated with one and only one Board. Normally a board will be associated with many pins, but this is not required. Pin has a relationship with the Picture entity. This entity represents the image that has been pinned to the board. It stores a pic_id, the image, a source url for the image, and an uploaded at timestamp. Pin and Picture participate with each other in both the original_pin and pic_pin relationships. These represent the oringal pin for the uploaded image and a pointer to the original image and associated data for every pin that uses it. When a picture is uplaoded, it can have tags associated with it. This is represented by the Tag entity that stores the tag as text. Tag has a many to many relationship with Picture as one Tag can be associated with many pictures and vice versa. Users can like and comment on pins. Like is a weak entity set that is associated with the user that gave the like and the picture that stores the underlying image. Comments is an entity set with a relationship to the pin that it commented on, not necesarily the original, and the user that made the comment.

Users also have the ability to create private follow streams. A user attaches boards to a stream, so that all pins for every board in a stream can be displayed to the user together. The entity Follow represents the connection between a StreamFollow and a Board. This stores a timestamp for when the board was added to the stream. Follow has a many to many relationship with Board, and it has a total participation relationship with StreamFollow as every Follow must be associated with a stream follow. StreamFollow has a many relationship with Follow as a stream can follow many boards. Follow is a weak entity as its primary key attributes existst in Board, bid, and FollowStream, sid. Users can also establish friendships on this website. The FriendRequest relationship stores all data related to a friendship. Each friend request is associated with one pair of users, but a user can be associated with many friend requests. The final relation is the Friend weak entity. When one user accepts a friend request from another user, an entry is added to Friend to represent their internet friendship. Please see an image of the ER diagram below.


TAG
tname
Original_Pin
Picture_Tag
PICTURE
pic_id
img_data
src_url
uploaded_at
Pin_User
PIN
pin_id
created_at
Like
created_at
Like_Pic
Comm_Pin
Like_User
FR_Send_User
Comments
cid
content
created_at
USER
uid
uname
email
created_at
Friend
rid
send_id
recv_id
status
sent_at
resp_at
Comm_User
Pin_Board
BOARD
bid
name
comment_perms
created_at
FR_Recv_User
Board_User
Stream_User
Follow_User
Follow_Board
FOLLOW_STREAM
sid
stream_name
created_at
Stream_Follow
FOLLOW
followed_at

After designing our ER diagram, we created a schema for our database, as seen on the next page. The primary and foreign keys were chosen to reflect the entities and relationships of our ER diagram. We also added the requirement that many of these keys not have null values. For the Pin and Picture entities, we had an issue with a foreign key cycle between Pin and Picture due to every Pin being associated with a picture and every picture having an attribute for the original pin that uploaded it. After discussing this cycle, we decided that these attributes made the most logical sense in their current locations. We got around this cycle by allowing a picture and pin to have a null picid. When a user uploads a new image, we insert a new row into Pin with a null pic id. We then use the new pinid to insert a row into picture, and use the resulting new picid to update the new pin. A user cannot observe any of these implementation details, so the functionality of the database is preserved. Most relationships have a timestamp attribute, the main purpose of these timestamps is data analysis and perhaps additional info for users in the future such as a frienship aniversay reminder.

```
CREATE TABLE "User" (
  uname VARCHAR(50) PRIMARY KEY,
  personal_name VARCHAR(50) NOT NULL,
  passwd VARCHAR(50) NOT NULL,
  email VARCHAR(100) NOT NULL UNIQUE,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE Board (
  bid SERIAL PRIMARY KEY,
  uname VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  name VARCHAR(100) NOT NULL,
  comment_perms VARCHAR(20) NOT NULL DEFAULT 'public'
  CHECK (comment_perms IN ('public', 'friends')),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE Pin (
  pinid SERIAL PRIMARY KEY,
  uname VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  bid INTEGER NOT NULL REFERENCES Board(bid) ON DELETE CAS-
```

```
CADE,
  picid INTEGER, – Added via ALTER later
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE Picture (
  picid SERIAL PRIMARY KEY,
  org_pinid INTEGER REFERENCES Pin(pinid) ON DELETE CASCADE DE-
FAULT ,
  img_data BYTEA NOT NULL, – Stores the actual image
  src_url VARCHAR(255), – Page where the image was found
  uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

ALTER TABLE Pin ADD CONSTRAINT fk_pins_pictures
  FOREIGN KEY (picid) REFERENCES Picture(picid) ON DELETE CASCADE;

CREATE TABLE Tag (
  tname VARCHAR(50) PRIMARY KEY
);

CREATE TABLE PictureTag (
  picid INTEGER NOT NULL REFERENCES Picture(picid) ON DELETE CAS-
CADE,
  tname VARCHAR(50) NOT NULL REFERENCES Tag(tname) ON DELETE
CASCADE,
  PRIMARY KEY (picid, tname)
);

CREATE TABLE "Like" (
  uname VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  picid INTEGER NOT NULL REFERENCES Picture(picid) ON DELETE CAS-
CADE,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (uname, org_pinid)
);

CREATE TABLE "Comment" (
  cid SERIAL PRIMARY KEY,
```

```
  uname VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  pinid INTEGER NOT NULL REFERENCES Pin(pinid) ON DELETE CAS-
CADE,
  content TEXT NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE FriendRequest (
  rid SERIAL PRIMARY KEY,
  sender_uname VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  receiver_uname VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  status VARCHAR(20) NOT NULL DEFAULT 'pending'
  CHECK (status IN ('pending', 'accepted', 'rejected')),
  sent_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  CHECK (sender_uname != receiver_uname)
);
– Prevent duplicate pending requests
CREATE UNIQUE INDEX unique_pending_request
on FriendRequest(sender_uname, receiver_uname)
WHERE status = 'pending';

CREATE TABLE Friend (
  uname1 VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  uname2 VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  became_friend_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (uname1, uname2),
  CHECK (uname1 ¡ uname2) – Enforce consistent ordering
);

CREATE TABLE FollowStream (
  sid SERIAL PRIMARY KEY,
  uname VARCHAR(50) NOT NULL REFERENCES "User"(uname)
  ON DELETE CASCADE,
  stream_name VARCHAR(100) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

);

CREATE TABLE Follow (
  sid INTEGER NOT NULL REFERENCES FollowStream(sid) ON DELETE CASCADE,
  bid INTEGER NOT NULL REFERENCES Board(bid) ON DELETE CASCADE,
  followed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (sid, bid)
);

We next checked if this database schema is in Boyce-Codd normal form. Please see the following proof.

BCNF Check:
User:
$F = \{uid \to (uname, email, created\_at)$
$uname \to (uid, email, created\_at)$
$email \to (uid, uname, created\_at)\}$
$F = F^+$
For each member of F^+, the left-hand side is a superkey.
$\therefore$ this table is in BCNF.
Board:
$bid \to (uid, name, comment\_perms, created\_at)$
Pin:
$F = \{pin\_id \to (created\_at)$
$(uid, name) \to (bid, comment\_perms, created\_at)\}$
$F = F^+$
For each member of F^+, the left-hand side is a superkey.
$\therefore$ this table is in BCNF.
Picture:
$F = \{pic\_id \to (org\_pid, img\_data, src\_url, uploaded\_at)\}$
$F = F^+$
For each member of F^+, the left-hand side is a superkey.
$\therefore$ this table is in BCNF.
Tag:
$F = \{(tname, pic\_id) \to (tname, pic\_id)\}$
$F = F^+$
For each member of F^+, the dependency is trivial.
$\therefore$ this table is in BCNF.

Like:
$F = \{(uid, org\_pid) \rightarrow created\_at\}$
$F = F^+$
For each member of F^+, the left-hand side is a superkey.
$\therefore$ this table is in BCNF.
Comment:
$F = \{cid \rightarrow (uid, pid, content, created\_at)\}$
$F = F^+$
For each member of F^+, the left-hand side is a superkey.
$\therefore$ this table is in BCNF.
Friend:
$F = \{rid \rightarrow (send\_id, recv\_id, status, sent\_at, resp\_at)$
$(send\_id, recv\_id, sent\_at) \rightarrow (rid, status, resp\_at)$
$\}$
$F = F^+$
Due to the contraint that a user cannot send a friend request to another user if they already have a friend request pending, it is not possible for two friend requests to have the same sender, reciever, and send time. Consequently, these three attributes form a superkey. For each member of F^+, the left-hand side is a superkey.
$\therefore$ this table is in BCNF.
Follow_Stream:
$F = \{$
$sid \rightarrow (uid, stream\_name, created\_at)$
$(uid, stream\_name) \rightarrow (sid, created\_at)$
$\}$
$F = F^+$
For each member of F^+, the left-hand side is a superkey.
$\therefore$ this table is in BCNF.
Follow:
$F = \{(uid, bid) \rightarrow (followed\_at)\}$
$F = F^+$
For each member of F^+, the left-hand side is a superkey.
$\therefore$ this table is in BCNF.
Each table in this database schema is in BCNF.
$\therefore$ This database schema is in BCNF.■

Query Set: %s represents input data fromt the user.
Signing up, Creating Boards, and Pinning
Sign Up
INSERT INTO User (uname, passwd, email)

We implemented our web app using Django. All sql queries were written in a views.py file. Including all queries here would exceed the 15-20 page requirement. Instead we include some of the more interesting queries with photos of their results on the webapp.\\
Note: \%s represents input data from the user.

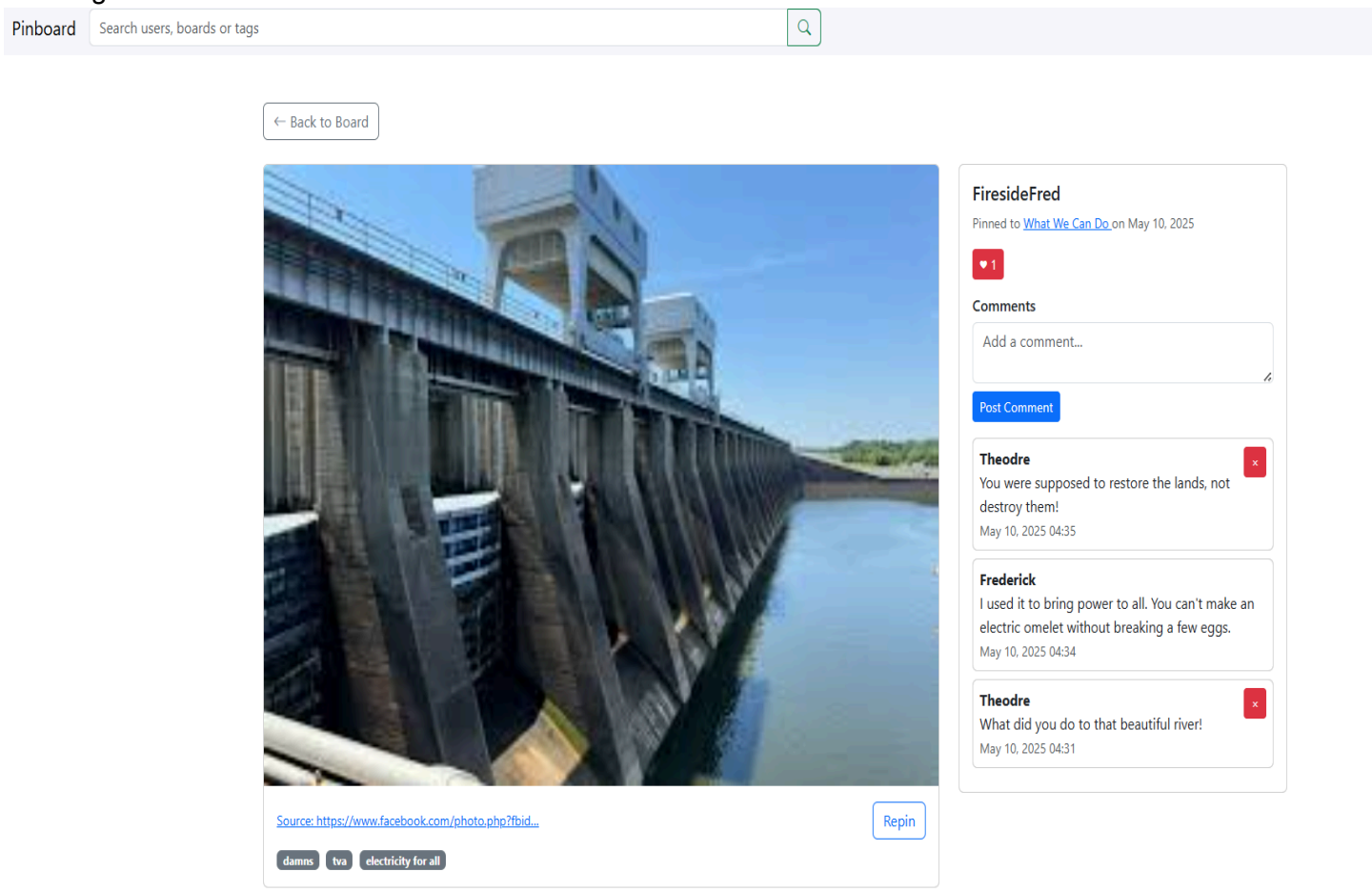
Pin View:\\

```
# Get pin details
cursor.execute("""
    select *
    from pin
    where pinid = %s
    """, [pinid])
pin_data = cursor.fetchone()
# Get pic details
cursor.execute("""
    select *
    from picture
    where picid = %s
    """, [pin_data[3]])
pic_data = cursor.fetchone()
# Get board details
cursor.execute("""
    select b.bid, b.name
    from board b join pin p
    on p.bid = b.bid
    where p.pinid = %s
    """, [pinid])
board_data = cursor.fetchone()
#get user info
cursor.execute("""
        select u.uname, u.personal_name
        from "User" u join board b
        on u.uname = b.uname
        where b.bid = %s
        """, [board_data[0]])
user_data = cursor.fetchone()
# Get tags
cursor.execute("""
    select tname
    from picturetag
    where picid = %s
    """, [pic_data[0]])
tags = [row[0] for row in cursor.fetchall()]
# get like count
cursor.execute("""
    select coalesce(count(uname), 0)
    from "Like"
    where picid = %s
    """, [pic_data[0]])
like_count = cursor.fetchone()[0]
if not pin_data:
```

```
    print("Failed to get pindata, returned to dashboard.\n")
    return redirect('dashboard')
# Get comments with ownership info
cursor.execute("""
    SELECT c.cid, c.content, c.created_at,
           u.personal_name, c.uname, b.uname as board_owner
    FROM "Comment" c
    JOIN "User" u ON c.uname = u.uname
    JOIN pin p ON c.pinid = p.pinid
    JOIN board b ON p.bid = b.bid
    WHERE c.pinid = %s
    ORDER BY c.created_at DESC
""", [pinid])
columns = [col[0] for col in cursor.description]
comments = [dict(zip(columns, row)) for row in cursor.fetchall()]
# Check if user has already like this picture
cursor.execute("""
    select exists(
        select *
        from "Like"
        where picid = %s and uname = %s)
    """, [pic_data[0], user_data[0]])
liked = cursor.fetchone()[0]
#get user's boards that don't have the picture
cursor.execute(
    """
    with matching_boards(bid) as
    (
        select distinct b.bid
        from board b join pin p
        on b.bid = p.bid and b.uname = p.uname
        where p.picid = %s
    ),
    other_boards(bid) as
    (
        select bid
        from board
        except
        select bid
        from matching_boards
    )
    select ob.bid, name
    from other_boards ob join board b on ob.bid = b.bid
    where uname = %s
    """, [pic_data[0], current_user]
)
boards = [{'bid' : row[0], 'board_name': row[1]} for row in cursor.fetchall()]
```

These queries in the pin_view function collects all relevant info for viewing a pin. This includes the amount of likes the underlying picture has, comments on the pin, comment permission level of the pin, tags associated with the pin, the user who made the pin, and the board they

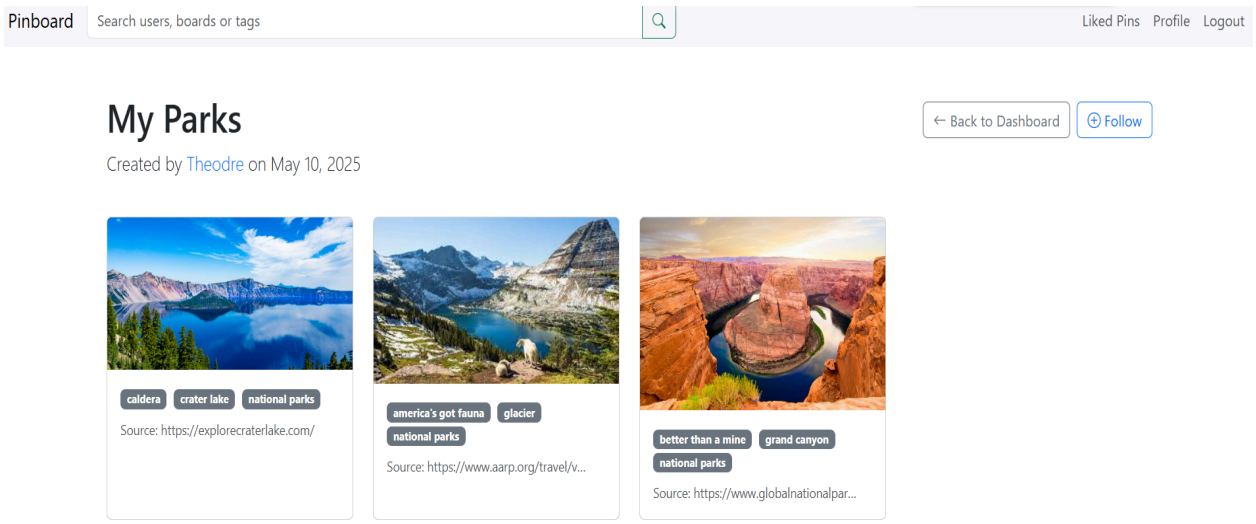
uploaded it to. This info is then fed to the view_pin.html file and displayed to the user as seen in the image below.



The functionalities of pin_view() and view_pin.html form an important part of a large portion of the web app. For example, this query is executed in board_view(),

```
SELECT p.pinid, pic.img_data, pic.src_url,
       array_agg(t.tname) AS tags
FROM pin p
JOIN picture pic ON p.picid = pic.picid
LEFT JOIN picturetag pt ON pic.picid = pt.picid
LEFT JOIN tag t ON pt.tname = t.tname
WHERE p.bid = %s
GROUP BY p.pinid, pic.picid
ORDER BY p.created_at DESC
```

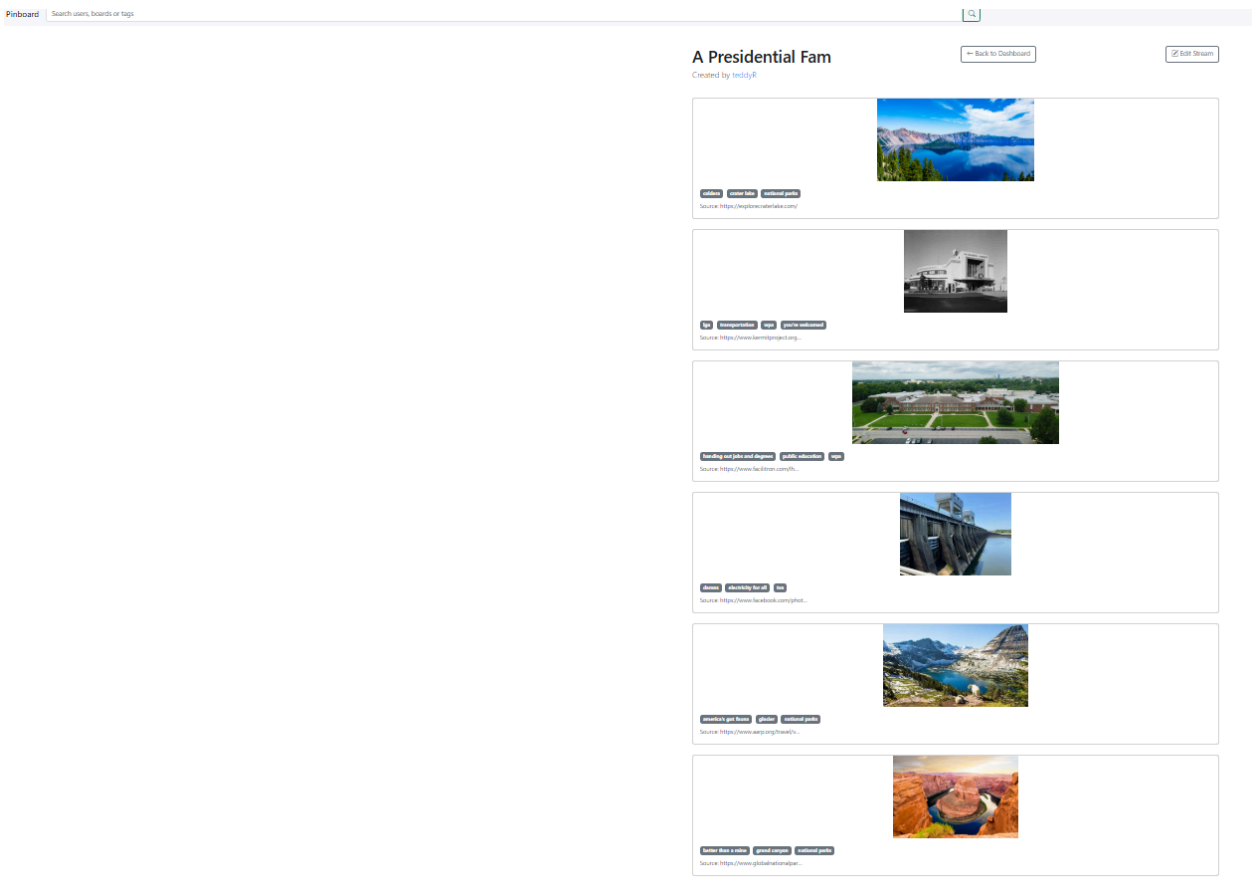
After pulling info related to a board with the above query, all pins associated with that board are sent to the view_board.html file which then uses pin_view as seen above to display each pin. The result can be seen in the following image.



For streams, this query is used.

```
cursor.execute("""
    select p.pinid, pic.img_data, pic.src_url,
           array_agg(t.tname) as tags
    from FollowStream s join Follow f on s.sid = f.sid
    join Pin p on f.bid = p.bid
    join picture pic on p.picid = pic.picid
    left join picturetag pt on pic.picid = pt.picid
    left join tag t on pt.tname = t.tname
    where s.sid = %s
    group by p.pinid, pic.img_data, pic.src_url
    order by p.created_at desc;
""", [sid])
```

The query returns the relevant info for all the boards that are being followed by the stream. The pins are then displayed in reverse chronological order as shown below. The below image is zoomed out to allow the entire followstream to fit in one page.



Views and html files are also used in combination with the database to allow users to search for boards, users, or tags.

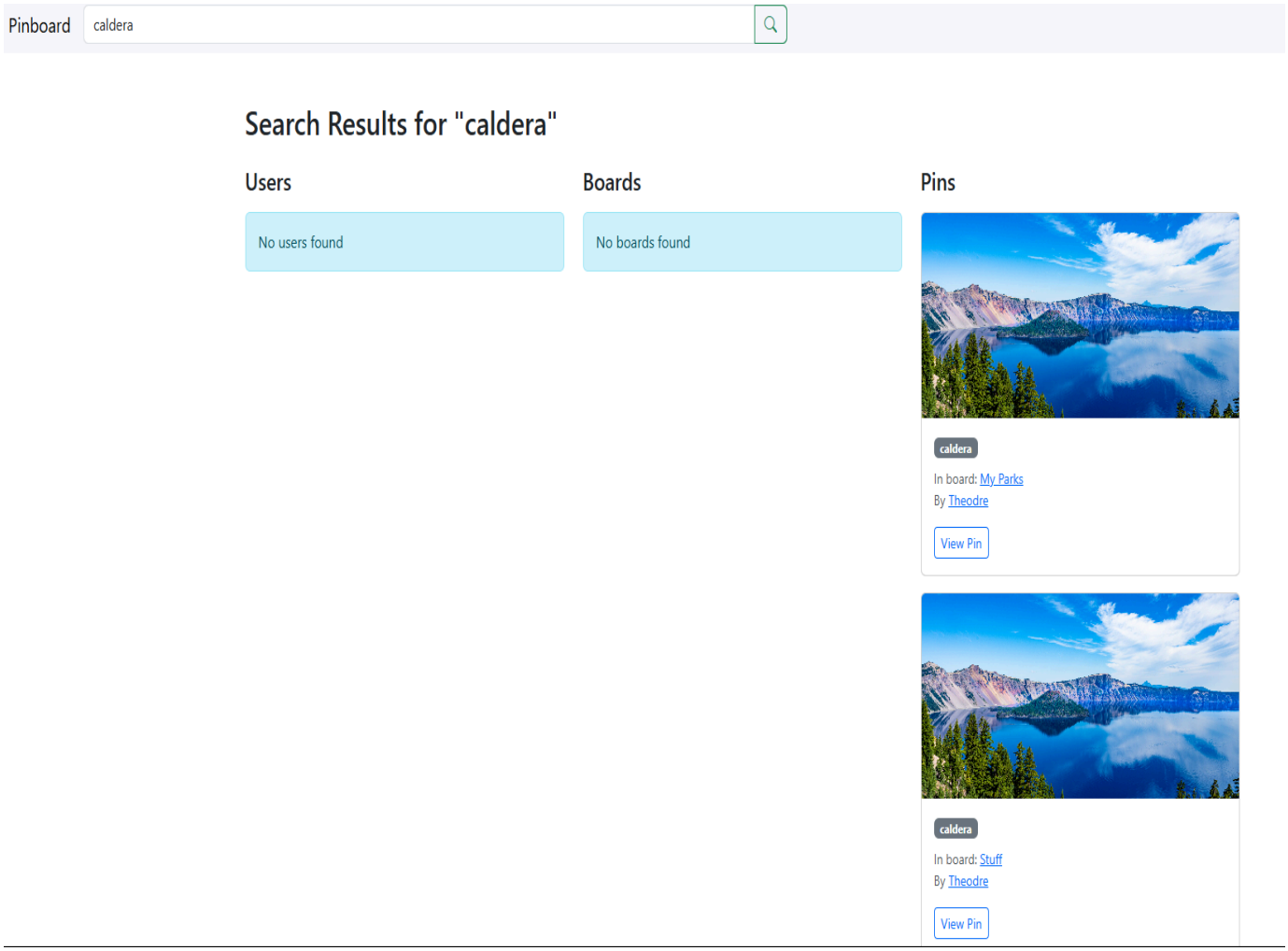
```
# Search Users
cursor.execute("""
    SELECT uname, personal_name, created_at
    FROM "User"
    WHERE uname ILIKE %s OR personal_name ILIKE %s
    LIMIT 10
""", [f'%{query}%', f'%{query}%'])
context['users'] = [
    dict(zip([col[0] for col in cursor.description], row))
    for row in cursor.fetchall()
]

# Search Boards
cursor.execute("""
    SELECT bid, name, created_at, uname
    FROM Board
    WHERE name ILIKE %s
```

```
    ORDER BY created_at DESC
    LIMIT 12
""", [f'%{query}%'])
context['boards'] = [
    dict(zip([col[0] for col in cursor.description], row))
    for row in cursor.fetchall()
]

# Search Pins by Tag
cursor.execute("""
    SELECT DISTINCT p.pinid, pic.src_url, pic.img_data,
           b.bid, b.name AS board_name,
           u.uname, u.personal_name,
           ARRAY_AGG(t.tname) AS tags
    FROM Tag t
    JOIN PictureTag pt ON t.tname = pt.tname
    JOIN Picture pic ON pt.picid = pic.picid
    JOIN Pin p ON pic.picid = p.picid
    JOIN Board b ON p.bid = b.bid
    JOIN "User" u ON b.uname = u.uname
    WHERE t.tname ILIKE %s
    GROUP BY p.pinid, pic.src_url, pic.img_data,
             b.bid, b.name, u.uname, u.personal_name
    LIMIT 12
""", [f'%{query}%'])
```

The above query is used to find any boards, users, or tags that are similar to the search term. The results are then displayed to the user as shown below.



Similar sql queries are used to coordinate all functionalities of the web app, such as user registration, friend requests, repinning images, etc. The rest of this paper describes how users interact with the web app and different components of the front end.

Base View

- Consistent navigation & session management
- Responsive design baseline (Bootstrap 5)
- Central resource loading (CSS/JS)
- Authentication state visibility

Key Features

1. Smart Navigation Bar
    - Adapts to login state:
      Logged Out: Login | Register
      Logged In: Profile | Liked Pins | Logout
    - Persistent search bar with query preservation
2. Content Structure
    - Main content container with top margin (`mt-5`)
    - Semantic HTML foundation
3. User Flow

    For New Users:

    1. See prominent Register button
    2. Access limited functionality

    For Members:

    1. Quick profile access
    2. Manage saved content (Liked Pins)
    3. Maintain search history

Dashboard View

Primary user hub showing:

- Personalized greeting with user stats
- Quick access to content creation (Boards/Streams)
- Overview of recent Boards & Streams
- Navigation to detailed views

Key Features

Dual Content Management

1. Boards (Collections): Focus on commenting permissions
2. Streams (Feeds): Track post counts
3. Unified creation buttons with icon and text

Responsive Card Grid

```
<div class="row row-cols-1 row-cols-md-3 row-cols-lg-4 g-4">
```

4. Adapts from 1 to 4 columns based on screen size
5. Consistent card height with `h-100`

User Flow

- Consistent
- Sees personalized greeting with name
- Creates new content via prominent buttons
- Interacts with cards:
    i. View details (Primary action)
    ii. Edit boards (Secondary action)
- Empty states guide to creation

Pinboard Creation View

Enable authenticated users to create new content boards with controlled commenting permissions.

Key Features

Minimalist Form Design

```
<div class="col-md-6">...</div> <!-- Centered container -->
```

- Focused 2-field form (Name + Permissions)
- Mobile-optimized layout (Bootstrap grid)
- Clear primary/secondary action buttons

Permission System

```
<select name="comment_perms">
  <option value="public">Public</option>
  <option value="friends">Friends Only</option>
</select>
```

- Pre-defined privacy levels
- Defaults to public accessibility

User Flow

1. Access from Dashboard CTA
2. Submit required name + permissions
3. Receive instant feedback:
    - Success: Redirect to dashboard
    - Error: Stay on form with message

Pinboard View

Display detailed pin view with social interactions and content management options.

Key Features

Multi-Pane Layout

```
<div class="col-lg-6"> <!-- Image -->
<div class="col-lg-6"> <!-- Details -->
```

- Left pane: Full-size image display
- Right pane: Metadata and interactions
- Responsive stack on mobile

Social Engagement

- Like counter with toggle
- Comment section with ownership indicators
- Repin functionality via modal

Content Management

```
<button data-bs-toggle="modal">Repin</button>
```

- Modal shows available boards
- Visual distinction for already-liked state

User Flow

1. Discover through board/dashboard
2. View image + metadata
3. Engage via:

    - Liking
    - Commenting
    - Repinning
4. Navigate to author's profile

Profile View

Display user profiles with social connections and content overview, enabling friend management and board discovery.

Key Features

Profile Header

1. Personal name & username display
2. Join date formatting ("Joined Jan 2023")
3. Contextual actions:
    - *Own Profile:* Edit button
    - *Others:* Friend request states (Pending/Accepted/Add)

Friend System

```
{% if friend_status == 'accepted' %}...{% endif %}
```

- Visual status indicators
- Incoming request management for profile owner
- Friends grid with avatars

Content Display

- Board cards with permissions badges
- Empty state handling for boards/friends
- Responsive 3-column grid (→1 column on mobile)

User Flow

1. **Profile Owner**
    - Manages incoming friend requests
    - Views friend list & personal boards
    - Edits profile via dedicated button
2. **Other Users**
    - Sends/views friend request status

    - Browses public boards
    - Cannot see private friend list

Search View

Provide unified search across users, boards, and pins with categorized results display.

Key Features

Tri-Pane Results Layout

```
<div class="col-md-4"> <!-- Users -->
<div class="col-md-4"> <!-- Boards -->
<div class="col-md-4"> <!-- Pins -->
```

- Equal-width columns for easy scanning
- Distinct visual treatments per content type
- Collapses to vertical stack on mobile

Smart Search Logic

- **Users:** Name/username partial match
- **Boards:** Name-based search
- **Pins:** Tag-driven results

Visual Hierarchy

- Users: List with join dates
- Boards: Cards with creator info
- Pins: Image previews and tags

Search Algorithms

```
/* Users */ WHERE uname ILIKE %query% OR personal_name ILIKE %query%
/* Boards */ WHERE name ILIKE %query%
/* Pins */ WHERE t.tname ILIKE %query%
```

- Case-insensitive partial matching
- Tag-centric pin discovery

User Flow

1. Enter search term in navbar
2. View categorized results
3. Drill down via:
    - User profiles
    - Board details
    - Pin inspection