Stage 4: Final Report

Team number: 33
Team name: TT
Team member:
Supawit Sutthiboriban (supawit3)
Yu-Ting Cheng (ytcheng4)
Yu-Chien Lin (yuchien4)
Han-Chih Chang (hanchih2)

Changes in Project

Compared to what we mentioned in our proposal, we have accomplished:

- In the **Login** page, users can log in using their email and password. We also implemented a mechanism to notify users if they enter an incorrect email or password.

- In the **Register** page, users can sign up for an account. We also implemented a mechanism to prevent users from registering with the same email address.

- In the **Closet** page, users can add, update, and delete clothes, add and delete tags, search for clothing items by name, and filter clothing items by tags and categories.

- In the **Calendar** page, users can add, update, and delete a wearing history. They can even post today's outfit to the Posts page with a description, allowing others to vote on it.

- In the **Today's Outfit** page, we integrated a current weather API so users can see the current weather. Additionally, we added a Color Frequency Chart, enabling users to identify which colors they wore most frequently over a selected time period, helping them understand their dressing preferences. Based on today's color usage category, users can ask the system to recommend suitable outfits. The recommended outfits are ranked, with the most suitable ones appearing at the top. Our recommendation system considers whether the user has worn the clothing in the past two weeks, the current weather, the color the user wants to wear, and the occasion they are dressing for.

- In the **Posts** page, users can see what other users are wearing today, vote on their outfits, and showcase their own looks for the day.

Overall, we have delivered a much more complete and robust implementation than what was initially outlined in our proposal.

Discussion of Achievements and Usefulness

1. **Achievements**:

    - The application successfully serves as a comprehensive outfit calendar, combining a closet manager, outfit recording, and recommendation system.
    - Its features go beyond basic closet management by integrating **real-time weather**, **color frequency analysis**, and **event-based recommendations**, which help users save time deciding what to wear.
    - The **Posts page** enhances community engagement, allowing users to share and vote on outfits, making the app not only functional but also social.
    - By leveraging the **Color Frequency Chart**, users gain insights into their dressing habits, which promotes self-awareness and better decision-making in styling.

2. **Failed to achieve**:

    - We successfully delivered all proposed features, with no failures or omissions in achieving our goals.

Added and Removed Functionalities

1. **Added Functionalities**:

    - **Login/Registration Validations**: Mechanisms for handling incorrect email/password combinations and duplicate email prevention.
    - **Posts Page**: A new feature for users to post, vote, and showcase their outfits with other users.
    - **Color Frequency Chart**: A tool to help users analyze their color usage history and dressing preferences.
    - **Advanced Recommendations**: The system considers past wear frequency, weather, desired color, and event type to generate ranked outfit suggestions.

2. **Removed Functionalities**:

    - We did not remove any major functionalities explicitly, as all features from the proposal were either delivered or enhanced.

Reasons: Additions were made to enhance user experience and offer features that aligned with user needs as the project evolved.

Changes to Schema or Data Source

Schema

We added some new tables and changed attributes in the schema (marked in blue)

Posts(
    PostId: INT,
    Description: VARCHAR(255)
)

Votes(
    Date: DATE,
    UserId: INT [FK to Users.UserId],
    VoteNum: INT,
)

VoteHistory(
    Date: DATE,
    UserId: INT [FK to Users.UserId],
    PostId: INT [FK to Posts.PostId],
)

Users(
    UserId: INT,
    FirstName: VARCHAR(255),
    LastName: VARCHAR(255),
    PhoneNumber: REAL,
    Email: VARCHAR(255),
    Password: VARCHAR(255),
    Votes: INT
)

Clothes(
    ClothId: INT,
    UserId: INT [FK to Users.UserId],
    ClothName: VARCHAR(255),
    Category: VARCHAR(255),
    Subcategory: VARCHAR(255),
    Color: VARCHAR(255),
    Usage: VARCHAR(255),
    Image: VARCHAR(255),
    TemperatureLevel: INT [FK to Temperature.TemperatureLevel],

~~LatestWearTwoWeeks: DATE [FK to WearingHistory.Date]~~
)

FavoriteGroups(
FavoriteId: INT,
UserId: INT [FK to Users.UserId],
GroupName: VARCHAR(255)
)

Include(
FavoriteId: INT [FK to FavoriteGroups.FavoriteId],
ClothId: INT [FK to Clothes.ClothId]
)
Note: this table comes from the relationship between FavoriteGroups entity and Clothes entity.

WearingHistory(
Date: DATE,
UserId: INT [FK to Users.UserId],
Cloth1: INT [FK to Clothes.ClothId],
Cloth2: INT [FK to Clothes.ClothId],
Cloth3: INT [FK to Clothes.ClothId],
Cloth4: INT [FK to Clothes.ClothId],
Cloth5: INT [FK to Clothes.ClothId],
PostId: INT [FK to Posts.PostId]
)

Temperature(
TemperatureLevel: INT,
TemperatureMin: INT,
TemperatureMax: INT
)

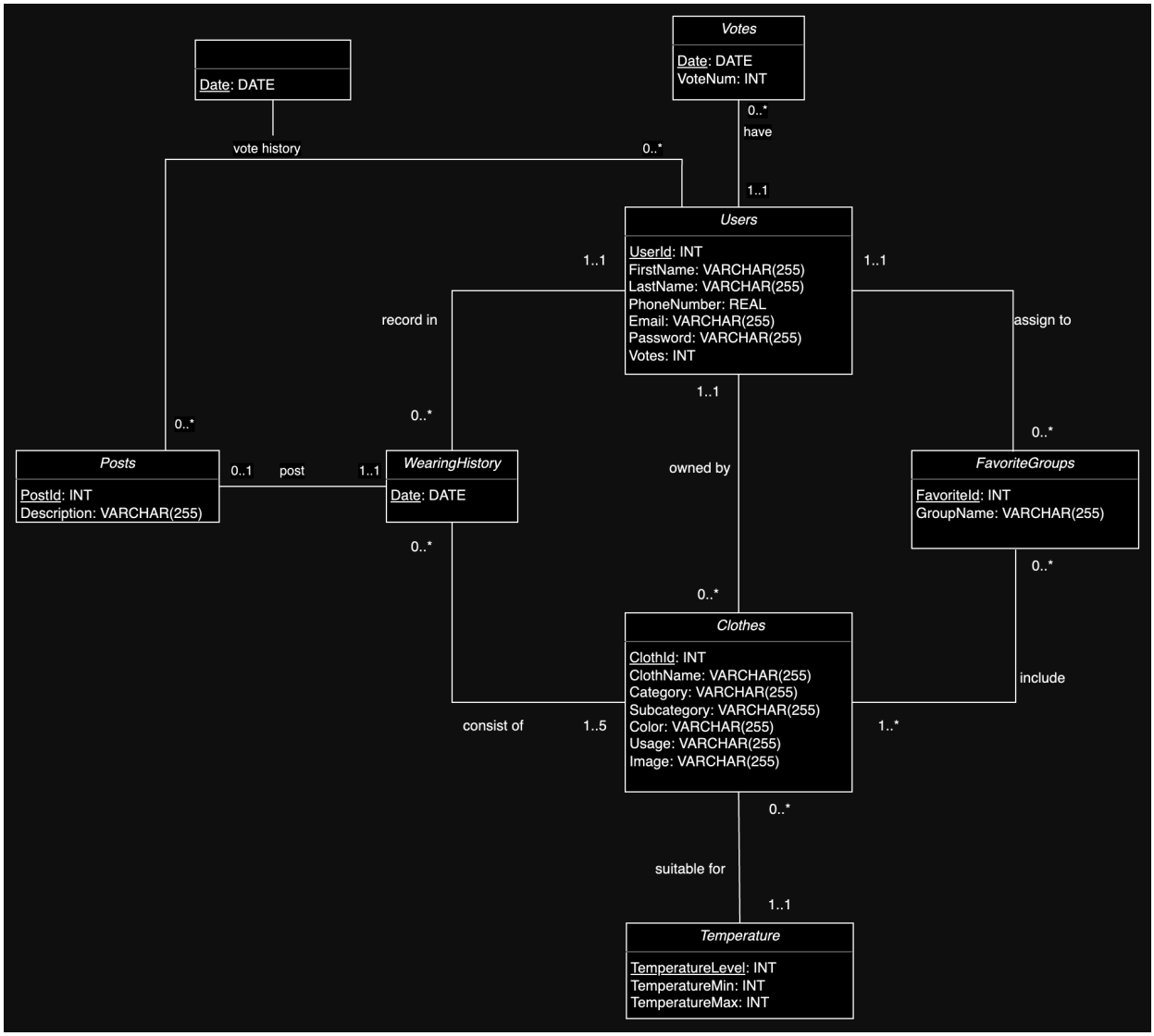
Data Source
We are using the Fashion Product Images Dataset that we mentioned in our previous proposal. Additionally, since our application is customized, each user should upload their own clothes. Therefore, all team members have registered an account and uploaded our own clothes, recording our outfits in our calendars.

Changes to ER Diagram and Table Implementations

Compared to the table structure reported in Stage 3, we added the Votes, Posts tables, and Have (between Votes and Users), Post (between WearingHistory and Posts), and VoteHistory (between Votes and Posts) relationships in the final stage. This expansion was driven by the implementation of the Posts page, allowing users to see what others are wearing today, vote on their outfits, and showcase their own looks.

We also added Users.Votes to track the total votes of a user and WearingHistory.PostId to reference the posted outfit. We removed Clothes.LatestWearTwoWeeks because we can get the information by searching in the WearingHistory table.

The rest of the ER diagram remained unchanged.



Detailed Feature

1. **Trigger**

TRIGGER user_insert
When a new user registers, our application automatically adds a default clothing item to their virtual closet, providing an example of how to use the application.

```
DELIMITER $$

CREATE TRIGGER user_insert
AFTER INSERT ON Users
FOR EACH ROW
BEGIN
    INSERT INTO Clothes (UserId, ClothName, Category, Subcategory, Color, Usages, Image,
TemperatureLevel)
    VALUES (NEW.UserId, 'White T-shirt', 'Topwear', 'Tshirts', 'White', 'Casual',

'http://assets.myntassets.com/v1/images/style/properties/57eeffafae5ef5e4a3a5812faa7b2fd3
_images.jpg', '2');

END$$

DELIMITER ;
```

```
mysql> DELIMITER $$
mysql>
mysql> CREATE TRIGGER user_insert
    -> AFTER INSERT ON Users
    -> FOR EACH ROW
    -> BEGIN
    ->     INSERT INTO Clothes (UserId, ClothName, Category, Subcategory, Color, Usages, Image, TemperatureLevel)
    ->     VALUES (
    ->         NEW.UserId,
    ->         'White T-shirt',
    ->         'Topwear',
    ->         'Tshirts',
    ->         'White',
    ->         'Casual',
    ->         'http://assets.myntassets.com/v1/images/style/properties/57eeffafae5ef5e4a3a5812faa7b2fd3_images.jpg',
    ->         '2'
    ->     );
    -> END$$
Query OK, 0 rows affected (0.02 sec)

mysql>
mysql> DELIMITER ;
mysql>
```

2. **Stored Procedure**

GetRecommendedClothes

The GetRecommendedClothes stored procedure is designed to recommend clothing items to users based on personalized preferences and context. It calculates a score for each item by evaluating multiple criteria, such as matching the preferred color, usage, and category, suitability for the current temperature, and whether the item has not been worn within 14 days (using a subquery on the WearingHistory table). The procedure

joins the Clothes and Temperature tables to ensure recommendations are weather-appropriate and ranks results by their total score.

```
CREATE PROCEDURE GetRecommendedClothes(
    IN p_UserId INT,
    IN p_Category VARCHAR(255),
    IN p_Color VARCHAR(255),
    IN p_Usages VARCHAR(255),
    IN p_CurrentTemperature INT
)
BEGIN
    SELECT c.*,
           CASE
               WHEN c.Color = p_Color THEN 1 ELSE 0
           END +
           CASE
               WHEN c.Usages = p_Usages THEN 1 ELSE 0
           END +
           CASE
                               WHEN  p_CurrentTemperature*1.8+32  >  t.TemperatureMin  AND
p_CurrentTemperature*1.8+32 < t.TemperatureMax THEN 1 ELSE 0
           END +
           CASE
               WHEN NOT EXISTS (
                   SELECT 1
                   FROM WearingHistory wh
                   WHERE (wh.Cloth1 = c.ClothId OR
                          wh.Cloth2 = c.ClothId OR
                          wh.Cloth3 = c.ClothId OR
                          wh.Cloth4 = c.ClothId OR
                          wh.Cloth5 = c.ClothId)
                   AND wh.Date >= DATE_SUB(CURRENT_DATE, INTERVAL 14 DAY)
               ) THEN 1 ELSE 0
           END AS Score
    FROM Clothes c
    NATURAL JOIN Temperature t
    WHERE c.UserId = p_UserId
        AND c.Category = p_Category
    ORDER BY Score DESC;
END;
```

GetColorFrequency

The GetColorFrequency stored procedure calculates the frequency of clothing colors worn by a specific user over a given interval of days. It has two parameters: input_UserId to identify the user and input_IntervalDays to define the time range. The procedure first checks if the interval is valid (greater than 0); otherwise, it returns an

error message. If valid, it joins the WearingHistory and Clothes tables. It filters results by the user ID, the date range, and non-null color values, then aggregates the data using GROUP BY to count occurrences of each color.

```
CREATE PROCEDURE GetColorFrequency(
   IN input_UserId INT,
   IN input_IntervalDays INT
)
BEGIN
   IF input_IntervalDays <= 0 THEN
       SELECT 'Error: The interval days must be greater than 0' AS ErrorMessage;
   ELSE
       SELECT c.Color,
           COUNT(*) AS ColorFrequency
       FROM WearingHistory wh
       JOIN Clothes c
           ON wh.Cloth1 = c.ClothId OR wh.Cloth2 = c.ClothId
           OR wh.Cloth3 = c.ClothId OR wh.Cloth4 = c.ClothId
           OR wh.Cloth5 = c.ClothId
       WHERE wh.UserId = input_UserId
       AND wh.Date BETWEEN DATE_SUB(CURRENT_DATE, INTERVAL input_IntervalDays DAY)
                       AND CURRENT_DATE
       AND c.Color IS NOT NULL
       GROUP BY c.Color
       ORDER BY ColorFrequency DESC;
   END IF;
END;
```

3. Transaction

The transaction level of our program is set to SERIALIZABLE isolation level, which guarantees the highest level of data consistency by ensuring that no two transactions can modify the same rows simultaneously. This prevents issues such as race conditions or inconsistencies in concurrent operations. For example, if two users attempt to vote for the same post at the same time, SERIALIZABLE isolation ensures that one transaction must wait for the other to complete before proceeding. This strict ordering ensures that the Votes count for the post's creator is incremented correctly, maintaining data integrity across the system.

Vote Transaction:

This transaction handles the voting process in a structured and atomic manner to maintain database consistency and ensure accurate vote updates. It begins by decrementing the available vote count for a user in the Votes table, ensuring they do not

exceed their voting limit for the day. Next, it increments the Votes count for the user associated with the post being voted on by utilizing nested subqueries to identify the correct user in the Users table. Finally, it inserts a new record into the VoteHistory table to log the voting action, linking the UserId, PostId, and the current date (Today). The use of subqueries ensures precise targeting of the relevant rows, while the transaction guarantees atomicity and consistency. By committing at the end, the process ensures all changes are saved together or none at all, preventing partial updates.

```
@app.route('/api/vote', methods=['POST', 'OPTIONS'])
def vote():
    Today = request.form.get('Date', None)
    UserId = request.form.get('UserId', None)
    PostId = request.form.get('PostId', None)

    print(f"POST /api/vote: User {UserId} vote for Post {PostId})")
    if Today and UserId and PostId:
        try:
            conn = pool.connection()
            with conn.cursor() as cursor:
                cursor.execute("SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE")
                cursor.execute("START TRANSACTION")
                # check votes you have
                cursor.execute('SELECT VoteNum FROM Votes WHERE Date= %s AND UserId=%s', (Today, UserId))
                votes = cursor.fetchone()
                votes = votes['VoteNum']
                if votes == 0:
                    return jsonify({"message": "Out of votes"})

                sql_remove = """UPDATE Votes
                                SET VoteNum =  VoteNum - 1
                                WHERE Date=%s AND UserId=%s"""
                cursor.execute(sql_remove, (Today, UserId))

                print(PostId)

                # Add one vote to the 'Users' table
                sql_add = """
                    UPDATE Users AS u
                    SET Votes = (
                        SELECT temp.Votes + 1
                        FROM (
                            SELECT u2.Votes
                            FROM Users u2
                            WHERE u2.UserId = (
                                SELECT w.UserId
                                FROM WearingHistory w
                                WHERE w.PostId = %s
                            )
                        ) AS temp
```

```
                )
                WHERE u.UserId = (
                    SELECT w.UserId
                    FROM WearingHistory w
                    WHERE w.PostId = %s
                );
            """

            cursor.execute(sql_add, (PostId, PostId))
            # add vote history
            sql = """INSERT INTO VoteHistory
                    VALUES (%s, %s, %s)"""
            cursor.execute(sql, (Today, UserId, PostId))
            conn.commit()

            return jsonify({'message': 'voted'})
    except Exception as e:
        if conn:
            conn.rollback()
        print('Error:', e)
        return jsonify({'error': str(e)})
    finally:
        if 'conn' in locals():
            conn.close()
```

Unvote Transaction:

This transaction handles the “unvote” process, which is the reverse of the voting process. The steps and queries are conceptually similar to the voting transaction but are flipped to reverse the vote’s effects. The process ensures that all updates are executed together or not at all, maintaining the database’s integrity.

```
@app.route('/api/unvote', methods=['POST', 'OPTIONS'])
def unvote():
   Today = request.form.get('Date', None)
   UserId = request.form.get('UserId', None)
   PostId = request.form.get('PostId', None)

   print(f"POST /api/vote: User {UserId} vote for Post {PostId})")
   if Today and UserId and PostId:
       try:
           conn = pool.connection()
           with conn.cursor() as cursor:
               # cursor.execute("SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE")
               cursor.execute("SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE")
               cursor.execute("START TRANSACTION")
               # add one vote to User
               sql_remove = """UPDATE Votes
                               SET VoteNum = (
                                   SELECT temp.VoteNum + 1
                                   FROM (
```

```
                                SELECT VoteNum
                                FROM Votes
                                WHERE Date=%s AND UserId=%s
                            ) AS temp
                        )
                        WHERE Date=%s AND UserId=%s"""
            cursor.execute(sql_remove, (Today, UserId, Today, UserId))
            sql_remove_vote = """
                UPDATE Users AS u
                SET Votes = (
                    SELECT temp.Votes - 1
                    FROM (
                        SELECT u2.Votes
                        FROM Users u2
                        WHERE u2.UserId = (
                            SELECT w.UserId
                            FROM WearingHistory w
                            WHERE w.PostId = %s
                        )
                    ) AS temp
                )
                WHERE u.UserId = (
                    SELECT w.UserId
                    FROM WearingHistory w
                    WHERE w.PostId = %s
                );
            """
            cursor.execute(sql_remove_vote, (PostId, PostId))

            # Remove vote history
            sql_delete_vote_history = """
                DELETE FROM VoteHistory
                WHERE Date = %s AND UserId = %s AND PostId = %s;
            """
            cursor.execute(sql_delete_vote_history, (Today, UserId, PostId))
            conn.commit()

            return jsonify({'message': 'unvoted'})
    except Exception as e:
        if conn:
            conn.rollback()
        print('Error:', e)
        return jsonify({'error': str(e)})
    finally:
        # Ensure the connection is returned to the pool
        if 'conn' in locals():
            conn.close()
```

4. Constraints

Our database is designed with essential constraints to ensure data integrity and maintain logical relationships between entities. Each table incorporates Primary Keys (PK), such as UserId in the

Users table and ClothId in the Clothes table, to uniquely identify records. For example, the WearingHistory table uses a composite key comprising Date and UserId to ensure that each user's outfit history for a specific date is unique.

Additionally, Foreign Keys (FK) establish relationships between tables, such as Clothes.UserId referencing Users.UserId to ensure that every clothing item belongs to a valid user, and Include.FavoriteId referencing FavoriteGroups.FavoriteId to link clothing items to specific favorite groups. For instance, this ensures that a clothing item cannot be associated with a non-existent group. These constraints maintain data consistency and enforce logical dependencies across the database.

Advanced Database Programs

Our advanced database programs significantly enhance the reliability and efficiency of our application by leveraging transactions and connection pooling. Transactions ensure that critical operations, such as voting or unvoting, are executed atomically, maintaining data consistency and integrity across multiple tables.

Also, connection pooling further complements this by optimizing database access, enabling multiple users to interact with the application simultaneously without causing delays or overloading the database. By reusing established connections, the application minimizes the overhead of establishing new connections, resulting in faster response times and improved scalability. Together, these features ensure that our application performs reliably and efficiently, even as user demand grows.

Technical Challenges and Advice

1. **Yu-Ting Cheng**:
    - **Challenge**: Implementing **CRUD operations for tags** in the closet. The difficulty was ensuring tags dynamically updated across multiple components while maintaining database integrity.
    - **Advice**: Use a state management tool like Redux to synchronize updates across components and design a tagging schema in the database to avoid redundancy.
2. **Yu-Chien Lin**:
    - **Challenge**: Handling the **calendar's date-based outfit filtering and editing**. A significant hurdle was ensuring seamless updates to both the calendar UI and the backend database when users modified or removed items.
    - **Advice**: Use libraries like Calendar.js for front-end calendar functionality and ensure consistent API contracts between the front and back ends.

3. **Supawit Sutthiboriban**:

    - **Challenge**: Developing the **recommendation system**. Designing a weighted scoring algorithm that effectively balanced weather, past wear, and user preferences was challenging due to conflicting requirements.
    - **Advice**: Test and refine scoring weights with sample data iteratively. Using machine learning models could improve future recommendations.

4. **Han-Chih Chang**:

    - **Challenge**: Designing a database schema for **scalability and performance**. Handling large datasets (e.g., Fashion Product Images Dataset) while ensuring quick search and filtering was complex.
    - **Advice**: Optimize queries with proper indexing and consider caching frequently used data, such as category filters or frequently accessed outfits.

Changes from the Original Proposal

1. Added functionalities like the Posts page, Interactive Voting Mechanism, Color Frequency Chart, and validation mechanisms were not part of the initial plan.
2. Delivered a more comprehensive implementation for tag management, which was not explicitly detailed in the proposal.

Future Work

1. **Recommendation System**:

    - Improve by integrating machine learning to provide smarter recommendations based on user behavior and feedback.
    - Add a feature to suggest outfits for multiple events or locations in one day.

2. **Posts Page**:

    - Introduce filters (e.g., event type, weather) to let users explore outfits more effectively.
    - Add a comment section for users to provide feedback on outfits.

3. **Scalability**:

- Optimize the application for larger closets and user bases by enhancing backend performance and improving UI responsiveness.

4. **Mobile App**:

    - Develop a mobile version to provide a more seamless and accessible user experience.

Final Division of Labor and Teamwork

1. **Division of Labor**:

    - **Yu-Ting Cheng**: CRUD operations for clothing items and tags in the closet.
    - **Yu-Chien Lin**: Calendar page and outfit history operations.
    - **Supawit Sutthiboriban**: Recommendation system and weather API integration.
    - **Han-Chih Chang**: Database design and overall backend architecture.
    - **Front-End**: All members contributed to front-end design and integration.

2. **Teamwork Management**:

    - The team maintained effective communication and clear task delegation. Weekly progress meetings and task trackers helped ensure alignment.
    - Collaboration tools like GitHub were used to manage code and track tasks, minimizing duplication of efforts or conflicts.