VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Database System (Lab) (CO2013)

Assigment 2 Submission

SimpleFacebook

Instructor:	Lê Đức Hoàng Nam	
Student(s):	Cao Anh Khôi	2352600
	Nguyễn Bùi Quang	2352969
	Lê Ngọc Hùng Dũng	2452224
	Nguyễn Hồ Khanh	2452487
	Nguyễn Mạnh Quốc Khánh	2352525

Contents

1 DATABASE IMPLEMENTATION

This section describes the database design used for the social platform implemented in this assignment. It documents the schema and constraints that model domain entities and relationships, and provides seed data used for development and testing.

1.1 Database Schema & Constraints

This subsection lists the tables, columns, keys, enumerations, and foreign-key constraints used to enforce integrity and map the application's domain model to a relational schema.

1.1.1 Identity & Access Management

The following content defines the core identity and access management tables: user accounts, profiles, roles, and the mapping of roles to users. These structures support authentication, basic authorization, and user metadata.

1. Table `users`

```
CREATE TABLE `users` (
  `user_id` bigint NOT NULL AUTO_INCREMENT,
  `email` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `phone_number` varchar(20) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `password_hash` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL DEFAULT (now()),
  `last_login` datetime DEFAULT NULL,
  `is_active` tinyint(1) NOT NULL DEFAULT '1',
  PRIMARY KEY (`user_id`),
  UNIQUE KEY `ix_users_email` (`email`),
  UNIQUE KEY `phone_number` (`phone_number`),
  KEY `ix_users_user_id` (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

2. Table `profiles`

```
CREATE TABLE `profiles` (
  `profile_id` bigint NOT NULL AUTO_INCREMENT,
  `user_id` bigint NOT NULL,
  `first_name` varchar(100) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `last_name` varchar(100) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `bio` text COLLATE utf8mb4_unicode_ci,
  `date_of_birth` date DEFAULT NULL,
  `gender` enum('MALE','FEMALE','OTHER') COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `profile_picture_url` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `cover_photo_url` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`profile_id`),
  UNIQUE KEY `ix_profiles_user_id` (`user_id`),
  KEY `ix_profiles_profile_id` (`profile_id`),
  CONSTRAINT `profiles_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

3. Table `roles`

```
CREATE TABLE `roles` (
  `role_id` bigint NOT NULL AUTO_INCREMENT,
  `role_name` varchar(50) COLLATE utf8mb4_unicode_ci NOT NULL,
  `description` text COLLATE utf8mb4_unicode_ci,
  PRIMARY KEY (`role_id`),
  UNIQUE KEY `role_name` (`role_name`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

4. Table `user_roles`

```sql
CREATE TABLE `user_roles` (
  `user_id` bigint NOT NULL,
  `role_id` bigint NOT NULL,
  PRIMARY KEY (`user_id`,`role_id`),
  KEY `role_id` (`role_id`),
  CONSTRAINT `user_roles_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`user_id`),
  CONSTRAINT `user_roles_ibfk_2` FOREIGN KEY (`role_id`) REFERENCES `roles` (`role_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.1.2 Social Graph

This following content models user-to-user relationships such as friendships and their states (pending, accepted, blocked). It captures the social connections used by privacy and feed algorithms.

1. Table `friendships`

```sql
CREATE TABLE `friendships` (
  `user_one_id` bigint NOT NULL,
  `user_two_id` bigint NOT NULL,
  `status` enum('PENDING','ACCEPTED','BLOCKED') COLLATE utf8mb4_unicode_ci NOT NULL,
  `action_user_id` bigint NOT NULL,
  `created_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`user_one_id`,`user_two_id`),
  KEY `user_two_id` (`user_two_id`),
  KEY `action_user_id` (`action_user_id`),
  CONSTRAINT `friendships_ibfk_1` FOREIGN KEY (`user_one_id`) REFERENCES `users` (`user_id`),
  CONSTRAINT `friendships_ibfk_2` FOREIGN KEY (`user_two_id`) REFERENCES `users` (`user_id`),
  CONSTRAINT `friendships_ibfk_3` FOREIGN KEY (`action_user_id`) REFERENCES `users` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.1.3 Content Engine

This following content contains tables for posts, file attachments, and mappings that determine where content appears (user timelines, pages, groups). It supports original posts and shares.

1. Table `posts`

```sql
CREATE TABLE `posts` (
  `post_id` bigint NOT NULL AUTO_INCREMENT,
  `author_id` bigint NOT NULL,
  `author_type` enum('USER','PAGE') COLLATE utf8mb4_unicode_ci NOT NULL,
  `total_comments` int DEFAULT 0,
  `text_content` text COLLATE utf8mb4_unicode_ci,
  `privacy_setting` enum('PUBLIC','FRIENDS','ONLY_ME') COLLATE utf8mb4_unicode_ci
    NOT NULL DEFAULT 'FRIENDS',
  `created_at` datetime NOT NULL DEFAULT (now()),
  `updated_at` datetime DEFAULT NULL,
  `post_type` enum('ORIGINAL','SHARE') COLLATE utf8mb4_unicode_ci DEFAULT 'ORIGINAL'
    ,
  `parent_post_id` bigint DEFAULT NULL,
  PRIMARY KEY (`post_id`),
  KEY `parent_post_id` (`parent_post_id`),
  CONSTRAINT `posts_ibfk_1` FOREIGN KEY (`parent_post_id`) REFERENCES `posts` (`
    post_id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

2. Table `post_locations`

```sql
CREATE TABLE `post_locations` (
  `post_id` bigint NOT NULL,
  `location_id` bigint NOT NULL,
```

```
  `location_type` enum('USER_TIMELINE','GROUP', 'PAGE_TIMELINE') COLLATE
    utf8mb4_unicode_ci NOT NULL,
  PRIMARY KEY (`post_id`,`location_id`,`location_type`),
  CONSTRAINT `post_locations_ibfk_1` FOREIGN KEY (`post_id`) REFERENCES `posts` (`
    post_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

3. Table `files`

```
CREATE TABLE `files` (
  `file_id` bigint NOT NULL AUTO_INCREMENT,
  `uploader_user_id` bigint NOT NULL,
  `file_name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `file_type` varchar(100) COLLATE utf8mb4_unicode_ci NOT NULL,
  `file_size` int NOT NULL,
  `file_url` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `thumbnail_url` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `created_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`file_id`),
  KEY `uploader_user_id` (`uploader_user_id`),
  CONSTRAINT `files_ibfk_1` FOREIGN KEY (`uploader_user_id`) REFERENCES `users` (`
    user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

4. Table `post_files`

```
CREATE TABLE `post_files` (
  `post_id` bigint NOT NULL,
  `file_id` bigint NOT NULL,
  `display_order` int NOT NULL DEFAULT '0',
  PRIMARY KEY (`post_id`,`file_id`),
  KEY `file_id` (`file_id`),
  CONSTRAINT `post_files_ibfk_1` FOREIGN KEY (`post_id`) REFERENCES `posts` (`
    post_id`),
  CONSTRAINT `post_files_ibfk_2` FOREIGN KEY (`file_id`) REFERENCES `files` (`
    file_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.1.4 Interactions

This following content covers interaction models: comments and reactions, including nested comments and reaction types that users can apply to posts, comments, and files.

1. Table `comments`

```
CREATE TABLE `comments` (
  `comment_id` bigint NOT NULL AUTO_INCREMENT,
  `commenter_user_id` bigint NOT NULL,
  `commentable_id` bigint NOT NULL,
  `commentable_type` enum('POST','FILE') COLLATE utf8mb4_unicode_ci NOT NULL,
  `parent_comment_id` bigint DEFAULT NULL,
  `text_content` text COLLATE utf8mb4_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`comment_id`),
  KEY `commenter_user_id` (`commenter_user_id`),
  KEY `parent_comment_id` (`parent_comment_id`),
  CONSTRAINT `comments_ibfk_1` FOREIGN KEY (`commenter_user_id`) REFERENCES `users`
    (`user_id`),
  CONSTRAINT `comments_ibfk_2` FOREIGN KEY (`parent_comment_id`) REFERENCES `
    comments` (`comment_id`)
) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

2. Table `reactions`

```
CREATE TABLE `reactions` (
  `reactor_user_id` bigint NOT NULL,
  `reactable_id` bigint NOT NULL,
```

```
  `reactable_type` enum('POST','COMMENT', 'FILE') COLLATE utf8mb4_unicode_ci NOT
    NULL,
  `reaction_type` enum('LIKE','LOVE','HAHA','SAD','ANGRY') COLLATE
    utf8mb4_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`reactor_user_id`,`reactable_id`,`reactable_type`),
  CONSTRAINT `reactions_ibfk_1` FOREIGN KEY (`reactor_user_id`) REFERENCES `users`
    (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.1.5 Groups

The following content defines the group/community structures, membership state, rules, and membership question/answer support for private or moderated groups.

1. Table `groups`

```
CREATE TABLE `groups` (
  `group_id` bigint NOT NULL AUTO_INCREMENT,
  `creator_user_id` bigint NOT NULL,
  `group_name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `description` text COLLATE utf8mb4_unicode_ci,
  `cover_photo_url` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `privacy_type` enum('PUBLIC','PRIVATE') COLLATE utf8mb4_unicode_ci NOT NULL,
  `is_visible` tinyint(1) NOT NULL DEFAULT '1',
  `created_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`group_id`),
  KEY `creator_user_id` (`creator_user_id`),
  CONSTRAINT `groups_ibfk_1` FOREIGN KEY (`creator_user_id`) REFERENCES `users` (`
    user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

2. Table `group_memberships`

```
CREATE TABLE `group_memberships` (
  `user_id` bigint NOT NULL,
  `group_id` bigint NOT NULL,
  `role` enum('ADMIN','MODERATOR','MEMBER') COLLATE utf8mb4_unicode_ci NOT NULL
    DEFAULT 'MEMBER',
  `status` enum('JOINED','PENDING','BANNED','INVITED') COLLATE utf8mb4_unicode_ci
    NOT NULL,
  `joined_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`user_id`,`group_id`),
  KEY `group_id` (`group_id`),
  CONSTRAINT `group_memberships_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users`
    (`user_id`),
  CONSTRAINT `group_memberships_ibfk_2` FOREIGN KEY (`group_id`) REFERENCES `groups`
     (`group_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

3. Table `group_rules`

```
CREATE TABLE `group_rules` (
  `rule_id` bigint NOT NULL AUTO_INCREMENT,
  `group_id` bigint NOT NULL,
  `title` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `details` text COLLATE utf8mb4_unicode_ci,
  `display_order` int NOT NULL DEFAULT '0',
  PRIMARY KEY (`rule_id`),
  KEY `group_id` (`group_id`),
  CONSTRAINT `group_rules_ibfk_1` FOREIGN KEY (`group_id`) REFERENCES `groups` (`
    group_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

4. Table `membership_questions`

```sql
CREATE TABLE `membership_questions` (
  `question_id` bigint NOT NULL AUTO_INCREMENT,
  `group_id` bigint NOT NULL,
  `question_text` text COLLATE utf8mb4_unicode_ci NOT NULL,
  `is_required` tinyint(1) NOT NULL DEFAULT '0',
  PRIMARY KEY (`question_id`),
  KEY `group_id` (`group_id`),
  CONSTRAINT `membership_questions_ibfk_1` FOREIGN KEY (`group_id`) REFERENCES `
    groups` (`group_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

5. Table `membership_answers`

```sql
CREATE TABLE `membership_answers` (
  `user_id` bigint NOT NULL,
  `group_id` bigint NOT NULL,
  `question_id` bigint NOT NULL,
  `answer_text` text COLLATE utf8mb4_unicode_ci NOT NULL,
  PRIMARY KEY (`user_id`,`group_id`,`question_id`),
  KEY `group_id` (`group_id`),
  KEY `question_id` (`question_id`),
  CONSTRAINT `membership_answers_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users`
    (`user_id`),
  CONSTRAINT `membership_answers_ibfk_2` FOREIGN KEY (`group_id`) REFERENCES `groups
    ` (`group_id`),
  CONSTRAINT `membership_answers_ibfk_3` FOREIGN KEY (`question_id`) REFERENCES `
    membership_questions` (`question_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.1.6 Pages

The following content covers public pages (brand/organization pages), their roles, and followers.

1. Table `pages`

```sql
CREATE TABLE `pages` (
  `page_id` bigint NOT NULL AUTO_INCREMENT,
  `page_name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `username` varchar(100) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `category` varchar(100) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `description` text COLLATE utf8mb4_unicode_ci,
  `contact_info` json DEFAULT NULL,
  `created_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`page_id`),
  UNIQUE KEY `username` (`username`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

2. Table `page_roles`

```sql
CREATE TABLE `page_roles` (
  `user_id` bigint NOT NULL,
  `page_id` bigint NOT NULL,
  `role` enum('ADMIN','EDITOR','MODERATOR','ANALYST') COLLATE utf8mb4_unicode_ci NOT
     NULL,
  PRIMARY KEY (`user_id`,`page_id`),
  KEY `page_id` (`page_id`),
  CONSTRAINT `page_roles_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`
    user_id`),
  CONSTRAINT `page_roles_ibfk_2` FOREIGN KEY (`page_id`) REFERENCES `pages` (`
    page_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

3. Table `page_follows`

```sql
CREATE TABLE `page_follows` (
  `user_id` bigint NOT NULL,
  `page_id` bigint NOT NULL,
```

```
  `followed_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`user_id`,`page_id`),
  KEY `page_id` (`page_id`),
  CONSTRAINT `page_follows_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`user_id`),
  CONSTRAINT `page_follows_ibfk_2` FOREIGN KEY (`page_id`) REFERENCES `pages` (`page_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.1.7 Events

The following content models events, how they are published to timelines or groups, and participant RSVPs.

1. Table `events`

```
CREATE TABLE `events` (
  `event_id` bigint NOT NULL AUTO_INCREMENT,
  `host_id` bigint NOT NULL,
  `host_type` enum('USER','PAGE') COLLATE utf8mb4_unicode_ci NOT NULL,
  `event_name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `description` text COLLATE utf8mb4_unicode_ci,
  `start_time` datetime NOT NULL,
  `end_time` datetime DEFAULT NULL,
  `location_text` text COLLATE utf8mb4_unicode_ci,
  `privacy_setting` enum('PUBLIC','PRIVATE','FRIENDS') COLLATE utf8mb4_unicode_ci NOT NULL,
  PRIMARY KEY (`event_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

2. Table `event_publications`

```
CREATE TABLE `event_publications` (
  `publication_id` bigint NOT NULL AUTO_INCREMENT,
  `event_id` bigint NOT NULL,
  `publisher_id` bigint NOT NULL,
  `publisher_type` enum('USER','PAGE') COLLATE utf8mb4_unicode_ci NOT NULL,
  `location_id` bigint NOT NULL,
  `location_type` enum('USER_TIMELINE','GROUP','PAGE_TIMELINE') COLLATE utf8mb4_unicode_ci NOT NULL,
  PRIMARY KEY (`publication_id`),
  KEY `event_id` (`event_id`),
  CONSTRAINT `event_publications_ibfk_1` FOREIGN KEY (`event_id`) REFERENCES `events` (`event_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

3. Table `event_participants`

```
CREATE TABLE `event_participants` (
  `event_id` bigint NOT NULL,
  `user_id` bigint NOT NULL,
  `rsvp_status` enum('GOING','INTERESTED','CANT_GO') COLLATE utf8mb4_unicode_ci NOT NULL,
  `updated_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`event_id`,`user_id`),
  KEY `user_id` (`user_id`),
  CONSTRAINT `event_participants_ibfk_1` FOREIGN KEY (`event_id`) REFERENCES `events` (`event_id`),
  CONSTRAINT `event_participants_ibfk_2` FOREIGN KEY (`user_id`) REFERENCES `users` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.1.8 Moderation

The following content describes the reporting and moderation structures: report records, reasons, and administrative actions to enforce community rules.

1. Table `reports`

```
CREATE TABLE `reports` (
  `report_id` bigint NOT NULL AUTO_INCREMENT,
  `reporter_user_id` bigint NOT NULL,
  `reportable_id` bigint NOT NULL,
  `reportable_type` enum('POST','COMMENT','USER','PAGE','GROUP') COLLATE
    utf8mb4_unicode_ci NOT NULL,
  `reason_id` bigint NOT NULL,
  `status` enum('PENDING','REVIEWED','ACTION_TAKEN','DISMISSED') COLLATE
    utf8mb4_unicode_ci NOT NULL DEFAULT 'PENDING',
  `created_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`report_id`),
  KEY `reporter_user_id` (`reporter_user_id`),
  KEY `reason_id` (`reason_id`),
  CONSTRAINT `reports_ibfk_1` FOREIGN KEY (`reporter_user_id`) REFERENCES `users` (`
    user_id`),
  CONSTRAINT `reports_ibfk_2` FOREIGN KEY (`reason_id`) REFERENCES `report_reasons`
    (`reason_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

2. Table `report_reasons`

```
CREATE TABLE `report_reasons` (
  `reason_id` bigint NOT NULL AUTO_INCREMENT,
  `title` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `description` text COLLATE utf8mb4_unicode_ci,
  PRIMARY KEY (`reason_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

3. Table `report_actions`

```
CREATE TABLE `report_actions` (
  `action_id` bigint NOT NULL AUTO_INCREMENT,
  `report_id` bigint NOT NULL,
  `reviewer_admin_id` bigint NOT NULL,
  `action_taken` enum('DELETE_CONTENT','BAN_USER','WARN_USER','DISMISS_REPORT')
    COLLATE utf8mb4_unicode_ci NOT NULL,
  `notes` text COLLATE utf8mb4_unicode_ci,
  `action_at` datetime NOT NULL DEFAULT (now()),
  PRIMARY KEY (`action_id`),
  KEY `report_id` (`report_id`),
  KEY `reviewer_admin_id` (`reviewer_admin_id`),
  CONSTRAINT `report_actions_ibfk_1` FOREIGN KEY (`report_id`) REFERENCES `reports`
    (`report_id`),
  CONSTRAINT `report_actions_ibfk_2` FOREIGN KEY (`reviewer_admin_id`) REFERENCES `
    users` (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

1.2 Data Seeding

The following content provides sample seed data for development and testing. The inserts populate core identities, social connections, communities, content, interactions, event data, and moderation examples.

1.2.1 Core Identities (Users, Profiles, Roles)

Seed data for users, profiles, and role assignments to create a minimal set of accounts and permissions for testing.

```
-- 1. Create Users (5 rows)
INSERT INTO users (email, phone_number, password_hash, is_active) VALUES
('admin@system.com', '0900000001', 'hashed_secret_1', 1),
('john.doe@gmail.com', '0900000002', 'hashed_secret_2', 1),
('jane.smith@yahoo.com', '0900000003', 'hashed_secret_3', 1),
('mike.ross@law.com', '0900000004', 'hashed_secret_4', 1),
('rachel.zane@law.com', '0900000005', 'hashed_secret_5', 1);

```

```sql
-- 2. Create Profiles (5 rows - 1:1 with Users)
INSERT INTO profiles (user_id, first_name, last_name, gender, date_of_birth, bio) VALUES
(1, 'System', 'Administrator', 'OTHER', '2000-01-01', 'Maintaining the platform.'),
(2, 'John', 'Doe', 'MALE', '1995-05-15', 'Travel enthusiast and photographer.'),
(3, 'Jane', 'Smith', 'FEMALE', '1998-08-20', 'Coffee addict. Tech lover.'),
(4, 'Mike', 'Ross', 'MALE', '1992-11-10', 'Memory wizard.'),
(5, 'Rachel', 'Zane', 'FEMALE', '1994-04-25', 'Paralegal turned lawyer.');

-- 3. Create Roles (User & System Roles)
INSERT INTO roles (role_name, description) VALUES
('SYSTEM_ADMIN', 'Full access to system settings'),
('MODERATOR', 'Can review reports and ban users'),
('USER', 'Regular user');

-- 4. Assign Roles to Users
INSERT INTO user_roles (user_id, role_id) VALUES
(1, 1), -- Admin is System Admin
(1, 2), -- Admin is also Moderator
(2, 3), -- John is Standard user
(3, 3), -- Jane is Standard user
(4, 3); -- Mike is Standard user
```

1.2.2 Social Graph (Friendships)

Seed examples showing friendship requests, accepted connections, and blocked relationships.

```sql
-- Create Friendships (5 rows)
-- Note: Logic user_one_id < user_two_id should be maintained if enforcing unique pairs
INSERT INTO friendships (user_one_id, user_two_id, status, action_user_id) VALUES
(2, 3, 'ACCEPTED', 3), -- John & Jane are friends (Jane accepted)
(2, 4, 'PENDING', 2), -- John sent request to Mike
(4, 5, 'ACCEPTED', 5), -- Mike & Rachel are friends
(3, 5, 'BLOCKED', 3), -- Jane blocked Rachel
(2, 5, 'ACCEPTED', 5); -- John & Rachel are friends
```

1.2.3 Communities (Groups & Pages)

Seed data for groups, pages, group memberships, and page follows to demonstrate community functionality.

```sql
-- 1. Create Groups (5 rows)
INSERT INTO `groups` (creator_user_id, group_name, privacy_type, description) VALUES
(2, 'Photography Lovers', 'PUBLIC', 'Share your best shots!'),
(3, 'Tech Talk VN', 'PUBLIC', 'Discussing latest tech trends.'),
(4, 'Legal Advice', 'PRIVATE', 'Confidential legal discussions.'),
(2, 'Cat Memes', 'PUBLIC', 'Funny cats only.'),
(5, 'Cooking 101', 'PUBLIC', 'Learn to cook from scratch.');

-- 2. Create Pages (5 rows)
INSERT INTO pages (page_name, username, category) VALUES
('HCMUT Official', 'hcmut_edu', 'Education'),
('Starbucks Vietnam', 'starbucks_vn', 'Food & Beverage'),
('Nike Store', 'nike_official', 'Retail'),
('Marvel Studios', 'marvel', 'Entertainment'),
('VTV24', 'vtv24_news', 'Media');

-- 3. Group Memberships (5 rows)
INSERT INTO group_memberships (user_id, group_id, role, status) VALUES
(2, 1, 'ADMIN', 'JOINED'), -- John created Photo group
(3, 1, 'MEMBER', 'JOINED'), -- Jane joined Photo group
(4, 3, 'ADMIN', 'JOINED'), -- Mike created Legal group
(5, 3, 'MEMBER', 'PENDING'), -- Rachel requested to join Legal group
(3, 2, 'MODERATOR', 'JOINED'); -- Jane mods Tech group

-- 4. Page Follows (5 rows)
INSERT INTO page_follows (user_id, page_id) VALUES
(2, 1), (3, 1), (4, 1), -- Everyone follows HCMUT
(2, 4), -- John follows Marvel
(5, 2); -- Rachel follows Starbucks
```

1.2.4 Content Engine (Posts, Locations)

Sample posts and location mappings to populate timelines, pages, and groups for testing feeds.

```
-- 1. Create Posts (5 rows)
-- Note: No file attachments in this seed data to avoid invalid URLs
INSERT INTO posts (author_id, author_type, text_content, privacy_setting, post_type,
    parent_post_id) VALUES
(2, 'USER', 'What a beautiful sunset today!', 'PUBLIC', 'ORIGINAL', NULL), -- Post 1 (
    User)
(1, 'PAGE', 'Enrollment for Fall 2025 starts now!', 'PUBLIC', 'ORIGINAL', NULL), -- Post
    2 (Page)
(3, 'USER', 'Anyone know a good Python tutorial?', 'FRIENDS', 'ORIGINAL', NULL), -- Post
    3 (User)
(4, 'USER', 'Sharing this amazing view!', 'PUBLIC', 'SHARE', 1), -- Post 4 (Share Post 1)
(2, 'USER', 'Just saw the funniest cat video ever!', 'PUBLIC', 'ORIGINAL', NULL); -- Post
     5 (User)

-- 2. Post Locations (Mapping posts to where they appear)
INSERT INTO post_locations (post_id, location_id, location_type) VALUES
(1, 2, 'USER_TIMELINE'), -- Post 1 on John's timeline
(2, 1, 'PAGE_TIMELINE'), -- Post 2 on HCMUT Page
(3, 2, 'GROUP'), -- Post 3 in "Tech Talk VN" Group (Group ID 2)
(4, 4, 'USER_TIMELINE'), -- Post 4 on Mike's timeline
(5, 2, 'USER_TIMELINE'); -- Post 5 on John's timeline
```

1.2.5 Interactions (Comments, Reactions)

Seed examples of comments and reactions to simulate engagement and nested replies.

```
-- 1. Comments (5 rows)
INSERT INTO comments (commenter_user_id, commentable_id, commentable_type, text_content,
    parent_comment_id) VALUES
(3, 1, 'POST', 'Wow amazing sunset!', NULL), -- Jane comments on Post 1
(4, 1, 'POST', 'Nice shot bro.', NULL), -- Mike comments on Post 1
(2, 1, 'POST', 'Thanks guys!', 1), -- John replies to Jane (Comment 1)
(5, 2, 'POST', 'When is the enrollment deadline?', NULL), -- Rachel comments on Post 2
(2, 3, 'POST', 'Check out FreeCodeCamp!', NULL); -- John comments on Post 3

-- 2. Reactions (5 rows)
INSERT INTO reactions (reactor_user_id, reactable_id, reactable_type, reaction_type)
    VALUES
(3, 1, 'POST', 'LOVE'), -- Jane loved Post 1
(4, 1, 'POST', 'LIKE'), -- Mike liked Post 1
(5, 1, 'COMMENT', 'HAHA'), -- Rachel reacted to Comment 1
(2, 2, 'POST', 'LIKE'), -- John liked Post 2
(3, 5, 'POST', 'HAHA'); -- Jane laughed at Post 5
```

1.2.6 Events & Reports

Sample events, RSVPs, report reasons, and reports to illustrate event workflows and moderation scenarios.

```
-- 1. Events (5 rows)
INSERT INTO events (host_id, host_type, event_name, start_time, privacy_setting) VALUES
(2, 'USER', 'John Birthday Party', '2025-12-20 18:00:00', 'FRIENDS'),
(1, 'PAGE', 'HCMUT Open Day', '2025-11-15 08:00:00', 'PUBLIC'),
(3, 'USER', 'Coding Workshop', '2025-10-30 14:00:00', 'PUBLIC'),
(4, 'USER', 'Legal Webinar', '2025-11-05 20:00:00', 'PRIVATE'),
(5, 'USER', 'Cooking Class', '2025-12-01 10:00:00', 'PUBLIC');

-- 2. Event Participants (RSVP)
INSERT INTO event_participants (event_id, user_id, rsvp_status) VALUES
(1, 3, 'GOING'), -- Jane going to John's party
(1, 4, 'CANT_GO'), -- Mike can't go
(2, 2, 'INTERESTED'), -- John interested in Open Day
(2, 3, 'GOING'), -- Jane going to Open Day
(3, 2, 'GOING'); -- John going to Workshop

-- 3. Report Reasons (Metadata)
```

```sql
INSERT INTO report_reasons (title, description) VALUES
('Spam', 'Unwanted or repetitive content'),
('Hate Speech', 'Violent or discriminatory language'),
('False Information', 'Fake news or misleading info'),
('Harassment', 'Bullying or threatening behavior'),
('Nudity', 'Inappropriate sexual content');

-- 4. Reports (5 rows)
INSERT INTO reports (reporter_user_id, reportable_id, reportable_type, reason_id, status)
    VALUES
(3, 5, 'POST', 1, 'PENDING'), -- Jane reported Post 5 for Spam
(4, 1, 'COMMENT', 2, 'REVIEWED'), -- Mike reported Comment 1 for Hate Speech
(2, 5, 'USER', 4, 'ACTION_TAKEN'), -- John reported User 5 for Harassment
(5, 3, 'GROUP', 3, 'DISMISSED'), -- Rachel reported Group 3
(2, 2, 'PAGE', 1, 'PENDING'); -- John reported Page 2
```

2 ADVANCED SQL PROGRAMMING

This section demonstrates advanced database programming techniques used in the project, including stored procedures, triggers, reporting procedures, and user-defined functions for encapsulating business logic.

2.1 Stored Procedures for CRUD Operations

The following part provides stored procedures that implement validated Create, Update, and Delete operations for critical tables, demonstrating input checks and referential integrity enforcement.

We selected the `users` table to implement Stored Procedures for INSERT, UPDATE, and DELETE operations. This table requires strict validation for data integrity (unique emails, password strength) and referential integrity (cannot delete users who own critical system assets like Groups or Pages).

2.1.1 Procedure 1: Create New User (INSERT)

This procedure validates input data (email format, password length, duplicate checks) before creating a new account.

```
DELIMITER //

CREATE PROCEDURE sp_create_user(
    IN p_email VARCHAR(255),
    IN p_phone_number VARCHAR(20),
    IN p_password_raw VARCHAR(255)
)
BEGIN
    -- 1. Validate: Email cannot be empty
    IF p_email IS NULL OR p_email = '' THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Email cannot be empty.';
    END IF;

    -- 2. Validate: Email format (Must contain '@')
    IF p_email NOT LIKE '%@%' THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Invalid email format (missing @).';
    END IF;

    -- 3. Validate: Password length (Must be >= 6 chars)
    IF CHAR_LENGTH(p_password_raw) < 6 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Password must be at least 6 characters long.';
    END IF;

    -- 4. Validate: Check Duplicate Email
    IF EXISTS (SELECT 1 FROM users WHERE email = p_email) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: This email is already registered.';
    END IF;

    -- 5. Validate: Check Duplicate Phone (if provided)
    IF p_phone_number IS NOT NULL AND EXISTS (SELECT 1 FROM users WHERE phone_number =
    p_phone_number) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: This phone number is already in use.';
    END IF;

    -- 6. Execution: Insert new user
    INSERT INTO users (email, phone_number, password_hash, is_active)
    VALUES (p_email, p_phone_number, p_password_raw, 1);

    -- Optional: Return the ID of the new user
    SELECT LAST_INSERT_ID() as new_user_id;
END //

DELIMITER ;
```

2.1.2 Procedure 2: Update User Status (UPDATE)

This procedure allows updating a user's phone number and active status. It ensures that the new phone number does not conflict with other users.

```
DELIMITER //

CREATE PROCEDURE sp_update_user(
    IN p_user_id BIGINT,
    IN p_new_phone VARCHAR(20),
    IN p_is_active BOOLEAN
)
BEGIN
    -- 1. Validate: Check if User exists
    IF NOT EXISTS (SELECT 1 FROM users WHERE user_id = p_user_id) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: User ID does not exist.';
    END IF;

    -- 2. Validate: Check Duplicate Phone
    -- (Check if the new phone exists in the table BUT belongs to a DIFFERENT user)
    IF p_new_phone IS NOT NULL AND EXISTS (
        SELECT 1 FROM users
        WHERE phone_number = p_new_phone AND user_id != p_user_id
    ) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: This phone number is already taken by another user.';
    END IF;

    -- 3. Execution: Update user
    UPDATE users
    SET phone_number = p_new_phone,
        is_active = p_is_active
    WHERE user_id = p_user_id;
END //

DELIMITER ;
```

2.1.3 Procedure 3: Delete User (DELETE)

Purpose: To remove spam accounts or users who requested data deletion.
Constraint: We prevent deletion if the user is a "Creator" of a Group or "Admin" of a Page. Deleting such users would leave these communities "orphaned" (without an owner).

```
DELIMITER //

CREATE PROCEDURE sp_delete_user(
    IN p_user_id BIGINT
)
BEGIN
    -- 1. Validate: Check if User exists
    IF NOT EXISTS (SELECT 1 FROM `users` WHERE user_id = p_user_id) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: User ID not found.';
    END IF;

    -- 2. Constraint Check: Cannot delete if user is a Group Creator
    -- Note: Backticks used around `groups` as it is a reserved keyword in MySQL 8.0
    IF EXISTS (SELECT 1 FROM `groups` WHERE creator_user_id = p_user_id) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Cannot delete user. They are the creator of one or
    more Groups. Please transfer ownership first.';
    END IF;

    -- 3. Constraint Check: Cannot delete if user is the ONLY Admin of a Page
    IF EXISTS (
        SELECT 1 FROM `page_roles` pr
        WHERE pr.user_id = p_user_id AND pr.role = 'ADMIN'
        AND (SELECT COUNT(*) FROM `page_roles` pr2 WHERE pr2.page_id = pr.page_id AND pr2
    .role = 'ADMIN') = 1
```

```
    ) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Cannot delete user. They are the sole Admin of a Page.
    ';
    END IF;

    -- 4. Execution: Delete user
    DELETE FROM `users` WHERE user_id = p_user_id;
END //

DELIMITER ;
```

2.1.4 Testing Evidence

Below are the commands used to verify the procedures:

```
-- 1. Test Insert Success
CALL sp_create_user('newstudent@hcmut.edu.vn', '0999888777', 'securePass123');

-- 2. Test Insert Fail (Duplicate Email)
CALL sp_create_user('admin@system.com', '0111222333', '123456');
-- Result: Error Code: 1644. Error: This email is already registered.

-- 3. Test Update Fail (Duplicate Phone)
CALL sp_update_user(2, '0900000003', 1);
-- Result: Error Code: 1644. Error: This phone number is already taken by another user.

-- 4. Test Delete Fail (User owns a group)
CALL sp_delete_user(2);
-- Result: Error Code: 1644. Error: Cannot delete user. They are the creator of one or
    more Groups...
```

2.2 Triggers

2.2.1 Business Logic Trigger

Constraint Description: We enforce a logical constraint on nested comments (Replies).

Rule: "If a comment is a reply (has a `parent_comment_id`), it must refer to the same `commentable_id` (Post/File) as its parent comment."

```
DELIMITER //

CREATE TRIGGER trg_check_reply_consistency
BEFORE INSERT ON `comments`
FOR EACH ROW
BEGIN
    DECLARE parent_target_id BIGINT;
    DECLARE parent_target_type VARCHAR(50);

    -- Only check if this is a reply (parent_comment_id is not null)
    IF NEW.parent_comment_id IS NOT NULL THEN
        -- 1. Get target info of the parent comment
        SELECT commentable_id, commentable_type
        INTO parent_target_id, parent_target_type
        FROM `comments`
        WHERE comment_id = NEW.parent_comment_id;

        -- 2. Validate: Must match the new comment's target
        IF parent_target_id <> NEW.commentable_id OR parent_target_type <> NEW.
    commentable_type THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Error: A reply must belong to the same Post/File as its
    parent comment.';
        END IF;
    END IF;
END //

DELIMITER ;
```

Testing Evidence:

```
-- Setup: Post 1 and Post 2 exist. Comment 1 belongs to Post 1.
INSERT INTO `comments` (commenter_user_id, commentable_id, commentable_type, text_content
    )
VALUES (2, 1, 'POST', 'Root comment on Post 1');

-- Test: Invalid Reply (Reply to Comment 1, but trying to attach to Post 2) -> FAIL
INSERT INTO `comments` (commenter_user_id, commentable_id, commentable_type,
    parent_comment_id, text_content)
VALUES (3, 2, 'POST', 1, 'Hacking Attempt');
-- Result: Error Code: 1644. Error: A reply must belong to the same Post/File...
```

2.2.2 Derived Attribute Trigger

Attribute Selection: `total_comments` in table `posts`. This attribute stores the total number of comments for a specific post to optimize read performance.

Trigger 1: Increment on Insert

```
DELIMITER //

CREATE TRIGGER trg_update_comment_count_insert
AFTER INSERT ON `comments`
FOR EACH ROW
BEGIN
    -- Only update if the comment is on a POST
    IF NEW.commentable_type = 'POST' THEN
        UPDATE `posts`
        SET `total_comments` = `total_comments` + 1
        WHERE `post_id` = NEW.commentable_id;
    END IF;
END //

DELIMITER ;
```

Trigger 2: Decrement on Delete

```
DELIMITER //

CREATE TRIGGER trg_update_comment_count_delete
AFTER DELETE ON `comments`
FOR EACH ROW
BEGIN
    IF OLD.commentable_type = 'POST' THEN
        UPDATE `posts`
        SET `total_comments` = GREATEST(0, `total_comments` - 1)
        WHERE `post_id` = OLD.commentable_id;
    END IF;
END //

DELIMITER ;
```

Testing Evidence:

```
-- 1. Insert a new comment on Post 1
INSERT INTO `comments` (commenter_user_id, commentable_id, commentable_type, text_content
    )
VALUES (2, 1, 'POST', 'Testing Trigger');

-- 2. Check count -> Should be 1
SELECT post_id, total_comments FROM posts WHERE post_id = 1;

-- 3. Delete the comment
DELETE FROM `comments` WHERE text_content = 'Testing Trigger';

-- 4. Check count -> Should be 0
SELECT post_id, total_comments FROM posts WHERE post_id = 1;
```

2.3 Reporting Procedures

2.3.1 Procedure 1: Get User's Friend List

Requirement: Join ≥ 2 tables, WHERE, ORDER BY.
Description: This procedure retrieves the list of accepted friends for a specific user, including their profile details. It joins 'users', 'profiles', and 'friendships' tables.

```
DELIMITER //

CREATE PROCEDURE sp_get_user_friends(
    IN p_user_id BIGINT
)
BEGIN
    -- Select friend's ID, Name, and when they became friends
    SELECT
        u.user_id,
        u.email,
        CONCAT(p.first_name, ' ', p.last_name) AS full_name,
        p.profile_picture_url,
        f.created_at AS friendship_date
    FROM users u
    JOIN profiles p ON u.user_id = p.user_id
    JOIN friendships f ON (f.user_one_id = u.user_id OR f.user_two_id = u.user_id)
    WHERE
        -- Logic: Find rows where the input user is involved...
        (f.user_one_id = p_user_id OR f.user_two_id = p_user_id)
        -- ...but we want to display the OTHER person (the friend), not the input user
        AND u.user_id != p_user_id
        -- Only accepted friendships
        AND f.status = 'ACCEPTED'
    ORDER BY full_name ASC; -- Order alphabetically
END //

DELIMITER ;
```

2.3.2 Procedure 2: Get Top Active Users (Statistics)

Requirement: Aggregate function, GROUP BY, HAVING, WHERE, ORDER BY, Join ≥ 2 tables.
Description: This procedure identifies "Active Users" who have published more than a certain number of posts within a specific year. It calculates the total post count for each user.

```
DELIMITER //

CREATE PROCEDURE sp_get_active_users(
    IN p_year INT,
    IN p_min_posts INT
)
BEGIN
    SELECT
        u.user_id,
        u.email,
        COUNT(po.post_id) AS total_posts,
        MAX(po.created_at) AS last_post_date
    FROM users u
    JOIN posts po ON u.user_id = po.author_id
    WHERE
        -- Filter by Year (WHERE clause)
        YEAR(po.created_at) = p_year
        AND po.author_type = 'USER' -- Only count user posts, not page posts
    GROUP BY u.user_id, u.email
    HAVING
        -- Filter by Aggregate result (HAVING clause)
        total_posts >= p_min_posts
    ORDER BY total_posts DESC; -- Show most active users first
END //

DELIMITER ;
```

2.3.3 Testing Evidence

Below are the commands used to verify the reporting procedures with the seeded data:

```
-- 1. Test Procedure 1: Get friends of User ID 2 (John Doe)
-- Expectation: Should return Jane (ID 3) and Rachel (ID 5) based on seed data.
CALL sp_get_user_friends(2);

-- 2. Test Procedure 2: Get users who posted at least 1 post in 2025
-- Expectation: Should list users and their post counts, ordered by count descending.
CALL sp_get_active_users(2025, 1);
```

2.4 Functions

2.4.1 Function 1: Calculate User Activity Score

Requirement: Use Cursor, Loop, IF, Input Validation.
Logic: This function calculates a score for a user based on their posting history to identify "Influencers".

- ORIGINAL post: +10 points.
- SHARE post: +5 points.

```
DELIMITER //

CREATE FUNCTION func_calculate_user_activity_score(
    p_user_id BIGINT
)
RETURNS INT
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE v_total_score INT DEFAULT 0;
    DECLARE v_post_type VARCHAR(20);
    DECLARE v_done INT DEFAULT FALSE;

    -- 1. Declare Cursor to iterate through user's posts
    DECLARE cur_posts CURSOR FOR
        SELECT post_type FROM posts WHERE author_id = p_user_id AND author_type = 'USER';

    -- 2. Declare Handler for Cursor (Stop when no rows left)
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = TRUE;

    -- 3. Validate Input: Check if User exists
    IF NOT EXISTS (SELECT 1 FROM users WHERE user_id = p_user_id) THEN
        RETURN -1; -- Return -1 to indicate error/not found
    END IF;

    -- 4. Open Cursor and Loop
    OPEN cur_posts;

    read_loop: LOOP
        FETCH cur_posts INTO v_post_type;

        IF v_done THEN
            LEAVE read_loop;
        END IF;

        -- 5. Calculate Score based on Post Type
        IF v_post_type = 'ORIGINAL' THEN
            SET v_total_score = v_total_score + 10;
        ELSEIF v_post_type = 'SHARE' THEN
            SET v_total_score = v_total_score + 5;
        END IF;

    END LOOP;

    CLOSE cur_posts;

```

```
    RETURN v_total_score;
END //

DELIMITER ;
```

2.4.2 Function 2: Calculate Post Sentiment Score

Requirement: Use Cursor, Loop, IF/CASE, Input Validation.
Logic: This function calculates the "Sentiment" of a post by weighing reactions differently.

- LOVE: +3 points
- LIKE, HAHA: +1 point
- SAD: -1 point
- ANGRY: -2 points

```
DELIMITER //

CREATE FUNCTION func_calculate_post_sentiment(
    p_post_id BIGINT
)
RETURNS INT
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE v_sentiment_score INT DEFAULT 0;
    DECLARE v_reaction_type VARCHAR(20);
    DECLARE v_done INT DEFAULT FALSE;

    -- 1. Declare Cursor for reactions on this post
    DECLARE cur_reactions CURSOR FOR
        SELECT reaction_type FROM reactions
        WHERE reactable_id = p_post_id AND reactable_type = 'POST';

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = TRUE;

    -- 2. Validate Input
    IF NOT EXISTS (SELECT 1 FROM posts WHERE post_id = p_post_id) THEN
        RETURN NULL;
    END IF;

    OPEN cur_reactions;

    reaction_loop: LOOP
        FETCH cur_reactions INTO v_reaction_type;

        IF v_done THEN
            LEAVE reaction_loop;
        END IF;

        -- 3. Weighted Scoring Logic
        CASE v_reaction_type
            WHEN 'LOVE' THEN SET v_sentiment_score = v_sentiment_score + 3;
            WHEN 'LIKE' THEN SET v_sentiment_score = v_sentiment_score + 1;
            WHEN 'HAHA' THEN SET v_sentiment_score = v_sentiment_score + 1;
            WHEN 'SAD' THEN SET v_sentiment_score = v_sentiment_score - 1;
            WHEN 'ANGRY' THEN SET v_sentiment_score = v_sentiment_score - 2;
            ELSE SET v_sentiment_score = v_sentiment_score + 0;
        END CASE;

    END LOOP;

    CLOSE cur_reactions;

    RETURN v_sentiment_score;
END //

DELIMITER ;
```

2.4.3 Testing Evidence

Below are the commands used to verify the functions:

```sql
-- 1. Test Function 1: User Activity
-- User 2 has 2 ORIGINAL posts (in seed data) -> Expect: 20 points
-- User 4 has 1 SHARE post -> Expect: 5 points
SELECT
    user_id,
    email,
    func_calculate_user_activity_score(user_id) AS activity_score
FROM users
WHERE user_id IN (2, 4);

-- 2. Test Function 2: Post Sentiment
-- Post 1 has: 1 LOVE (Jane), 1 LIKE (Mike) -> Score: 3 + 1 = 4
-- Post 5 has: 1 HAHA (Jane) -> Score: 1
SELECT
    post_id,
    text_content,
    func_calculate_post_sentiment(post_id) AS sentiment_score
FROM posts
WHERE post_id IN (1, 5);
```