ebd

Last edited by **Rafael Ângelo dos Reis Campeão** 6 months ago

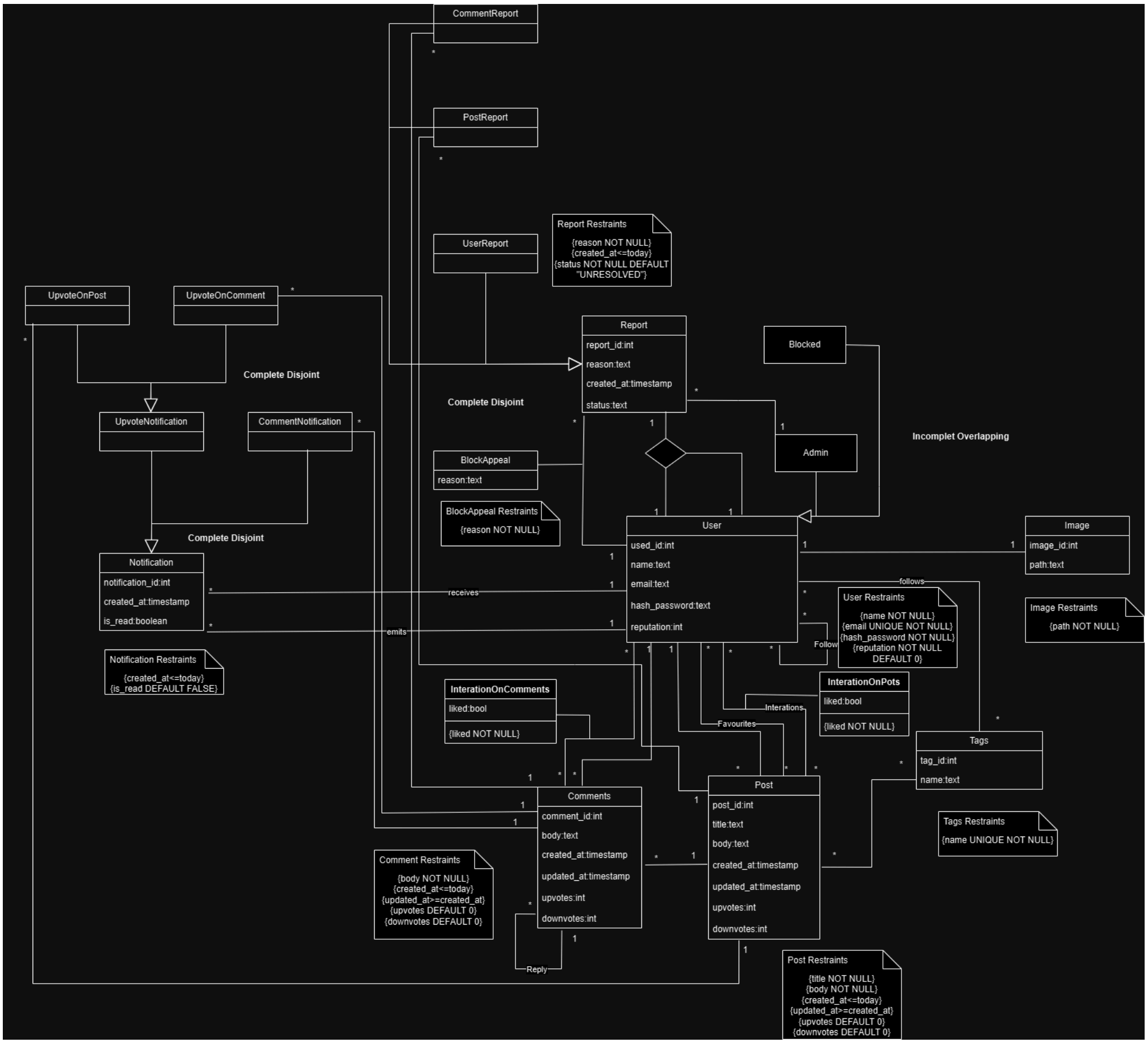
EBD: Database Specification Component

> The Bulletin is a community-driven news platform that empowers users to submit, vote on, and discuss news stories. It offers personalized feeds based on user interests, enabling people to engage with relevant content while shaping its visibility through voting. With a reputation system that rewards quality contributions and moderation tools to maintain platform integrity, The Bulletin fosters a respectful and collaborative environment. Accessible on all devices, it aims to create a democratic space where users actively participate in curating and discussing the news.

A4: Conceptual Data Model

> This section contains the description of the entities and relationships that exist to The Bulletin project and its database specification.

1. Class diagram



> The following UML diagram presents the main organizational entities, the relationships between them, attributes and their domains and the multiplicity of relationships for The Bulletin platform.

2. Additional Business Rules

Identifier	Name	Description
BR09	Report Limitations	A user who already reported another user cannot do it again until the original report is resolved
BR10	Assigning Admin To Report	When a report is created the admin who is assigned is the one with the least amount of reports assigned
BR11	Notification Creation	When a post or a comment receives a like or a comment, a notification is created

A5: Relational Schema, validation and schema refinement

> This section presents the relational schema derived from analyzing the conceptual data model. It includes each relational schema along with its attributes, domains, primary keys, foreign keys, and other essential integrity constraints, such as unique, default, not null, and check rules.

1. Relational Schema

Relation Reference	Relation Compact Notation
R01	user(user_id **PK**, username **NN**, email **UK NN**, hash_password **NN,** reputation **NN DF 0**)
R02	admin(admin_id → user **PK**)
R03	post(post_id **PK**, title **NN**, body **NN**, created_at **NN CK created_at <= today**, updated_at **NN CK updated_at <= today**, upvotes **NN DF 0,** downvotes **NN DF 0**, owner_id → users **NN**)
R04	comment(comment_id **PK**, content **NN**, created_at **NN CK created_at <= today**, updated_at **NN CK updated_at <= today,** upvotes **NN DF 0,** downvotes **NN DF 0**, post_id → post **NN**, reply_to → comment, owner_id → user **NN**)
R05	interaction_post(user_id → user, post_id → post, liked **NN**)
R06	interaction_comment(user_id → user, comment_id → comment, liked **NN**)
R07	tag(tag_id **PK**, name **UK NN**)
R08	follow_user(follower_id → user, followed_id → user)
R09	follow_tag(follower_id → user, tag_id → tag)
R10	comment_notification(notification_id **PK**, is_read **DF FALSE**, created_at **NN CK created_at <= today,** emmiter_id → user, recipient_id → user **NN**, comment_id → comment)
R11	upvote_on_post_notification(notification_id **PK**, is_read **DF FALSE**, created_at **NN CK created_at <= today,** emmiter_id → user, receiver_id → user **NN**, post_id → post)
R12	upvote_on_comment_notification(notification_id **PK**, is_read **DF FALSE**, created_at **NN CK created_at <= today,** emmiter_id → user, receiver_id → user **NN**, liked_comment → comment)
R13	report(report_id **PK**, reason **NN**, created_at **NN CK created_at <= today**, status **IN report_status NN DF "pending",** reporter_id → user **NN**, reported_user → user **NN**, assigned_admin → admin)
R14	user_report(id → report **PK**)
R15	comment_report(id → report, reported_comment → comment)
R16	post_report(id → report, reported_post → post)
R17	blocked(blocked_id → user **PK**)
R18	block_appeal(report_id → report **PK**, user_id → user **NN**, reason **NN**)
R19	image(image_id **PK**, user_id → user **NN**, path **NN**)
R20	favorite_post(user_id → user **NN**, post_id → post **NN**)
R21	post_tag(post_id → post **NN**, tag_id → tag **NN**)

Table 11: The Bulletin Relational Schema

2. Domains

Domain Name	Domain Specification
today	DATE DEFAULT CURRENT_DATE
activity_status	ENUM('active', 'flagged', 'deleted')
report_status	ENUM('resolved', 'pending', 'dismissed')

Table 12: The Bulletin domains

3. Schema Validation

TABLE R01	user
Keys	{ user_id }
Functional Dependencies:	
FD0101	user_id → { username, email, hash_password, reputation }
FD0102	email → { user_id, username, hash_password, reputation }
Normal Form	BCNF

TABLE R02	admin
Keys	{ admin_id }
Functional Dependencies:	
FD0201	admin_id → { user_id }
Normal Form	BCNF

TABLE R03	post
Keys	{ post_id }
Functional Dependencies:	
FD0301	post_id → { title, body, created_at, updated_at, upvotes, downvotes, owner_id }
Normal Form	BCNF

TABLE R04	comment
Keys	{ comment_id }
Functional Dependencies:	
FD0401	comment_id → { content, created_at, updated_at, upvotes, downvotes, post_id, reply_to, owner_id }
Normal Form	BCNF

TABLE R05	interaction_post
Keys	{ user_id, post_id }
Functional Dependencies:	
FD0501	{ user_id, post_id } → { liked }
Normal Form	BCNF

TABLE R06	interaction_comment
Keys	{ user_id, comment_id }
Functional Dependencies:	
FD0601	{ user_id, comment_id } → { liked }
Normal Form	BCNF

TABLE R07	tag
Keys	{ tag_id }
Functional Dependencies:	
FD0701	tag_id → { name }
Normal Form	BCNF

TABLE R08	follow_user
Keys	{ follower_id, followed_id }
Functional Dependencies:	none
Normal Form	BCNF

TABLE R09	follow_tag
Keys	{ follower_id, tag_id }
Functional Dependencies:	none
Normal Form	BCNF

TABLE R10	comment_notification
Keys	{ notification_id }
Functional Dependencies:	
FD1001	notification_id → { is_read, created_at, emmiter_id, recipient_id, comment_id }
Normal Form	BCNF

TABLE R11	upvote_on_post_notification
Keys	{ notification_id }
Functional Dependencies:	
FD1101	notification_id → { is_read, created_at, emmiter_id, receiver_id, post_id }
Normal Form	BCNF

TABLE R12	upvote_on_comment_notification
Keys	{ notification_id }
Functional Dependencies:	
FD1201	notification_id → { is_read, created_at, emmiter_id, receiver_id, liked_comment }
Normal Form	BCNF

TABLE R13	report
Keys	{ report_id }
Functional Dependencies:	
FD1301	report_id → { reason, created_at, status, reporter_id, reported_user, assigned_admin }
Normal Form	BCNF

TABLE R14	user_report
Keys	{ id }
Functional Dependencies:	none
Normal Form	BCNF

TABLE R15	comment_report
Keys	{ id, reported_comment }
Functional Dependencies:	none
Normal Form	BCNF

TABLE R16	post_report
Keys	{ id, reported_post }
Functional Dependencies:	none
Normal Form	BCNF

TABLE R17	blocked
Keys	{ blocked_id }
Functional Dependencies:	none
Normal Form	BCNF

TABLE R18	block_appeal
Keys	{ report_id }
Functional Dependencies:	
FD1801	report_id → { user_id, reason }
Normal Form	BCNF

TABLE R19	image
Keys	{ image_id }
Functional Dependencies:	
FD1901	image_id → { user_id, path }
Normal Form	BCNF

TABLE R20	favorite_post
Keys	{ user_id, post_id }

TABLE R20	favorite_post
Functional Dependencies:	none
Normal Form	BCNF

TABLE R21	post_tag
Keys	{ post_id, tag_id }
Functional Dependencies:	none
Normal Form	BCNF

Because all relations are in the Boyce–Codd Normal Form (BCNF), the relational schema is also in the BCNF and, therefore, the schema does not need to be further normalized.

A6: Indexes, triggers, transactions and database population

> The A6 artifact includes the PostgreSQL code, representing the database's physical schema and its data population. It covers data integrity support through triggers, the identification and characterization of indexes, and the definition of user-defined database functions.
> Additionally, it outlines the transactions necessary to maintain data accuracy following any database access or modifications, along with an explanation of the required isolation level.

1. Database Workload

> To design a well-structured database, it is essential to understand a table's growth and how frequently it will be accessed. The table below outlines these projections:

Relation reference	Relation Name	Order of magnitude	Estimated growth
R01	user	10 k (tens of thousands)	10 (tens) / day
R02	admin	10	1 (units) / month
R03	post	10 k	10 / day
R04	comment	100 k (hundreds of thousands)	100 (hundreds) / day
R05	upvote_on_post	1 M (millions)	1 k (thousands) / day
R06	upvote_on_comment	1 M	1 k / day
R07	tag	10	1 / month
R08	follow_user	100 k	1 k / day
R09	follow_tag	10 k	100 / day
R10	comment_notification	1 M	1 k / day
R11	post_notification	1 M	1 k / day
R12	report	1 k	10 / day
R13	user_report	1 k	1 / day
R14	comment_report	1 k	1 / day
R15	post_report	1 k	1 / day
R16	blocked	100	1 / day
R17	block_appeal	100	10 / month
R18	image	10 k	10 / day
R19	favorite_post	100 k	100 / day

2. Proposed Indices

2.1. Preformance Indices

Indices proposed to improve performance of the identified queries.

Index	IDX01: Comment Notification Date
Relation	CommentNotification
Attribute	created_at
Type	b-tree
Cardinality	high
Clustering	yes
Justification	Due to high cardinality and frequent insertion of new notifications, clustering enhances efficiency by keeping new entries grouped sequentially while also keeping everything ordered chronologically, optimizing the retrieval of recent notifications

SQL code

```
CREATE INDEX comment_notification_date ON CommentNotification USING btree (created_at);
CLUSTER CommentNotification USING comment_notification_date;
```

Index	IDX02:Tag-based Index for Post_tags
Relation	Post_Tags
Attribute	tag_id
Type	B-Tree
Cardinality	medium
Clustering	yes
Justification	This index groups entries by tag_id, optimizing joins between Post_tags and Posts when filtering by tag. Clustering allows related tags to be stored contiguously, improving performance for tag-based queries by reducing the need to scan widely dispersed rows.

SQL code

```
CREATE INDEX post_tags_tag ON Post_tags USING btree (tag_id);
CLUSTER Post_tags USING post_tags_tag;
```

Index	IDX03:Index for Faster Comment Loading on Post Pages
Relation	Comments
Attribute	post_id
Type	B-tree
Cardinality	High
Clustering	no
Justification	This index optimizes queries that retrieve comments for a specific post, which is critical for loading comments on post pages with high cardinality, the B-tree index enables efficient access to all comments for a given post_id without scanning the entire table.

SQL code

```
CREATE INDEX comments_post ON Comments(post_id);
```

2.2. Full-text Search Indices

> The system being developed must provide full-text search features supported by PostgreSQL. Thus, it is necessary to specify the fields where full-text search will be available and the associated setup, namely all necessary configurations, indexes definitions and other relevant details.

Index	IDX04: Full-Text Search Index for Posts (Title and Body)
Relation	Posts
Attribute	title, body
Type	GIN
Clustering	no
Justification	Full-text search on posts requires efficient retrieval of rows based on keywords in title and/or content, so we think the best option is a tsvector column with a GIN index to provide fast search performance.

SQL code

```
ALTER TABLE Posts
ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.title), 'A') ||
            setweight(to_tsvector('english', NEW.body), 'B')
        );
    ELSIF TG_OP = 'UPDATE' THEN
        IF (NEW.title <> OLD.title OR NEW.body <> OLD.body) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.title), 'A') ||
                setweight(to_tsvector('english', NEW.body), 'B')
            );
        END IF;
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER post_search_update
BEFORE INSERT OR UPDATE ON Posts
FOR EACH ROW
EXECUTE PROCEDURE post_search_update();

CREATE INDEX post_content ON Posts USING GIN (tsvectors);
```

Index	IDX05: Full-Text Search Index for Usernames
Relation	User
Attribute	username
Type	GIN
Clustering	no
Justification	Users want to search for usernames with partial matches or similar names, which can be helpful in cases where users may not remember full or exact usernames, so we think a index on a tsvector column containing the username provides efficient full-text search capabilities for this field.

SQL code

```
ALTER TABLE User
ADD COLUMN tsvectors TSVECTOR;
```

```
CREATE FUNCTION user_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' OR (TG_OP = 'UPDATE' AND NEW.username <> OLD.username) THEN
        NEW.tsvectors = to_tsvector('english', NEW.username);
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER user_search_update
BEFORE INSERT OR UPDATE ON User
FOR EACH ROW
EXECUTE PROCEDURE user_search_update();

CREATE INDEX user_username ON User USING GIN (tsvectors);
```

Index	IDX06: Full-Text Search Index for Comments (body)
Relation	comment
Attribute	body
Type	GIN
Clustering	no
Justification	For full-text search on comments, this index allows efficient retrieval of comments based on keywords and since comments are often shorter than posts, weighting is not necessary so a simple GIN index provides fast searching across large volumes of comments

SQL code

```
ALTER TABLE Comments
ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION comment_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = to_tsvector('english', NEW.body);
    ELSIF TG_OP = 'UPDATE' THEN
        IF (NEW.body <> OLD.body) THEN
            NEW.tsvectors = to_tsvector('english', NEW.body);
        END IF;
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER comment_search_update
BEFORE INSERT OR UPDATE ON Comments
FOR EACH ROW
EXECUTE PROCEDURE comment_search_update();

CREATE INDEX comment_content ON Comments USING GIN (tsvectors);
```

3. Triggers

Trigger	TRIGGER01
Description	Prevents users from submitting multiple pending reports for the same reported user, ensuring a clean review process (BR09)

SQL code

```
DROP FUNCTION IF EXISTS verify_report;
CREATE FUNCTION verify_report() RETURNS TRIGGER AS
$BODY$
```

```
    BEGIN
        IF EXISTS (SELECT * FROM Report WHERE NEW.reporter_id = reporter_id AND NEW.reported_user = reported_user AND re
        THEN RAISE EXCEPTION 'Report already done and waiting for resolotion';
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER verify_report
    BEFORE INSERT ON Report
    FOR EACH ROW
    EXECUTE PROCEDURE verify_report();
```

Trigger	TRIGGER02
Description	Ensures that users cannot interact (e.g., like) with their own posts (BR04)

SQL code

```
DROP FUNCTION IF EXISTS no_likes_on_own_post;
CREATE FUNCTION no_likes_on_own_post() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF (SELECT ownerId FROM Posts WHERE post_id = NEW.postId) = NEW.userId
        THEN RAISE EXCEPTION 'Users can not interact with their own posts';
        END IF;

        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER no_likes_on_own_post
    BEFORE INSERT ON InterationPosts
    FOR EACH ROW
    EXECUTE PROCEDURE no_likes_on_own_post();
```

Trigger	TRIGGER03
Description	Ensures that users cannot interact (e.g., like) with their own comments (BR04)

SQL code

```
DROP FUNCTION IF EXISTS no_likes_on_own_comment;
CREATE FUNCTION no_likes_on_own_comment() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF (SELECT ownerId FROM Comments WHERE comment_id = NEW.comment_id) = NEW.userId
        THEN RAISE EXCEPTION 'Users can not interact with their own comments';
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER no_likes_on_own_comment
    BEFORE INSERT ON InterationComments
    FOR EACH ROW
    EXECUTE PROCEDURE no_likes_on_own_comment();
```

Trigger	TRIGGER04
Description	Sends a notification to the post owner when someone interacts with their post, aligning with the interaction rules (BR11)

SQL code

```
DROP FUNCTION IF EXISTS post_like_notification;
CREATE FUNCTION post_like_notification() RETURNS TRIGGER AS
$BODY$
    BEGIN
        INSERT INTO UpvoteOnPostNotification(emitter,receiver,post) VALUES (NEW.userId,(SELECT ownerId FROM Posts WHERE
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER post_like_notification
    AFTER INSERT ON InterationPosts
    FOR EACH ROW
    EXECUTE PROCEDURE post_like_notification();
```

Trigger	TRIGGER05
Description	Sends a notification to the comment owner when someone interacts with their comment, following the interaction guidelines (BR11)

SQL code

```
DROP FUNCTION IF EXISTS comment_like_notification;
CREATE FUNCTION comment_like_notification() RETURNS TRIGGER AS
$BODY$
    BEGIN
        INSERT INTO UpvoteOnCommentNotification(emitter,receiver,liked_comment) VALUES (NEW.userId,(SELECT ownerId FROM
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER comment_like_notification
    AFTER INSERT ON InterationComments
    FOR EACH ROW
    EXECUTE PROCEDURE comment_like_notification();
```

Trigger	TRIGGER06
Description	Notifies the owner of a post or comment when they receive a reply, encouraging interaction between different users (BR11)

SQL code

```
DROP FUNCTION IF EXISTS comment_notification;
CREATE FUNCTION comment_notification() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF NEW.reply_to IS NOT NULL
        THEN INSERT INTO CommentNotification(emitter,receiver,comment) VALUES (NEW.ownerID,(SELECT ownerId FROM Comments
        ELSE INSERT INTO CommentNotification(emitter,receiver,comment) VALUES (NEW.ownerID,(SELECT ownerId FROM Posts WH
                END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER comment_notification
    AFTER INSERT ON Comments
    FOR EACH ROW
    EXECUTE PROCEDURE comment_notification();
```

Trigger	TRIGGER07
Description	Prevents deletion of posts that already have comments or interactions (BR03)

SQL code

```
DROP FUNCTION IF EXISTS delete_post_check;
CREATE FUNCTION delete_post_check() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF EXISTS (SELECT * FROM InterationPosts WHERE postId=OLD.post_id) OR EXISTS (SELECT * FROM Comments WHERE post
        THEN RAISE EXCEPTION 'Posts that already has either comments or interations can not be deleted';
        END IF;
        RETURN OLD;
    END
$BODY$
language plpgsql;

CREATE TRIGGER delete_post_check
    BEFORE DELETE ON Posts
    FOR EACH ROW
    EXECUTE PROCEDURE delete_post_check();
```

Trigger	TRIGGER08
Description	Prevents deletion of comments that already have replies or interactions (BR03)

SQL code

```
DROP FUNCTION IF EXISTS delete_comment_check;
CREATE FUNCTION delete_comment_check() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF EXISTS (SELECT * FROM InterationComments WHERE comment_id=OLD.comment_id) OR EXISTS (SELECT * FROM Comments W
        THEN RAISE EXCEPTION 'Comments that already have either replys or interations can not be deleted';
        END IF;
        RETURN OLD;
    END
$BODY$
language plpgsql;

CREATE TRIGGER delete_comment_check
    BEFORE DELETE ON Comments
    FOR EACH ROW
    EXECUTE PROCEDURE delete_comment_check();
```

Trigger	TRIGGER09
Description	Updates the reputation of the user who interacted with a post based on upvotes or downvotes (BR02)

SQL code

```
DROP FUNCTION IF EXISTS update_reputation_posts;
CREATE FUNCTION update_reputation_posts() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF NEW.liked THEN UPDATE Posts SET upvotes = upvotes + 1 WHERE post_id = NEW.postId; UPDATE Users SET reputatio
        ELSE UPDATE Posts SET downvotes = downvotes + 1 WHERE post_id = NEW.postId; UPDATE Users SET reputation = reput
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER update_reputation_posts
```

```
    AFTER INSERT ON InterationPosts
    FOR EACH ROW
    EXECUTE PROCEDURE update_reputation_posts();
```

Trigger	TRIGGER10
Description	Updates the reputation of the user who interacted with a comment based on upvotes or downvotes (BR02)

SQL code

```
DROP FUNCTION IF EXISTS update_reputation_comments;
CREATE FUNCTION update_reputation_comments() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF NEW.liked THEN UPDATE Comments SET upvotes = upvotes + 1 WHERE comment_id = NEW.comment_id; UPDATE Users SET
        ELSE UPDATE Comments SET downvotes = downvotes + 1 WHERE comment_id = NEW.comment_id; UPDATE Users SET reputati
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER update_reputation_comments
    AFTER INSERT ON InterationComments
    FOR EACH ROW
    EXECUTE PROCEDURE update_reputation_comments();
```

Trigger	TRIGGER11
Description	Adjusts the user's reputation when a like or dislike is removed from a post (BR02)

SQL code

```
DROP FUNCTION IF EXISTS update_reputation_posts_removed_like;
CREATE FUNCTION update_reputation_posts_removed_like() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF OLD.liked THEN UPDATE Posts SET upvotes = upvotes - 1 WHERE post_id = OLD.postId; UPDATE Users SET reputatio
        ELSE UPDATE Post SET downvotes = downvotes - 1 WHERE post_id = OLD.postId; UPDATE Users SET reputation = reputa
        END IF;
        RETURN OLD;
    END
$BODY$
language plpgsql;

CREATE TRIGGER update_reputation_posts_removed_like
    BEFORE DELETE ON InterationPosts
    FOR EACH ROW
    EXECUTE PROCEDURE update_reputation_posts_removed_like();
```

Trigger	TRIGGER12
Description	Adjusts the user's reputation when a like or dislike is removed from a comment (BR02)

SQL code

```
DROP FUNCTION IF EXISTS update_reputation_comments_removed_like;
CREATE FUNCTION update_reputation_comments_removed_like() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF OLD.liked THEN UPDATE Comments SET upvotes = upvotes - 1 WHERE comment_id = OLD.comment_id; UPDATE Users SET
        ELSE UPDATE Comments SET downvotes = downvotes - 1 WHERE comment_id = OLD.comment_id; UPDATE Users SET reputati
        END IF;
        RETURN OLD;
    END
```

```
$BODY$
language plpgsql;

CREATE TRIGGER update_reputation_comments_removed_like
    BEFORE DELETE ON InterationComments
    FOR EACH ROW
    EXECUTE PROCEDURE update_reputation_comments_removed_like();
```

Trigger	TRIGGER13
Description	Checks that a report is no longer pending before allowing a block appeal to be created (BR06)

SQL code

```
DROP FUNCTION IF EXISTS block_appeal_verify;
CREATE FUNCTION block_appeal_verify() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF EXISTS(SELECT * FROM Report WHERE report_id=NEW.report_id AND report_status='pending')
        THEN RAISE EXCEPTION 'Report still pending wait until resulotion';
        END IF;
        IF EXISTS(SELECT * FROM Report WHERE report_id=NEW.report_id AND (report_status='resolved' OR report_status='dis
        THEN UPDATE Report SET report_status='pending' WHERE report_id=NEW.report_id;
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER block_appeal_verify
    BEFORE INSERT ON Block_appeal
    FOR EACH ROW
    EXECUTE PROCEDURE block_appeal_verify();
```

Trigger	TRIGGER14
Description	Assigns a report to an admin with the fewest assigned cases, balancing review workload (BR10)

SQL code

```
DROP FUNCTION IF EXISTS give_admin_to_report;
CREATE FUNCTION give_admin_to_report() RETURNS TRIGGER AS
$BODY$
    BEGIN

                UPDATE Report SET assigned_admin = (SELECT admin_id FROM Admins WHERE admin_id = (SELECT assigned_admin
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER give_admin_to_report
    AFTER INSERT ON Report
    FOR EACH ROW
    EXECUTE PROCEDURE give_admin_to_report();
```

4. Transactions

Transaction	TRAN01
Description	Vote on a News Post

Transaction	TRAN01
Justification	This transaction enables users to vote on a news post, either by adding a new vote or updating an existing one. It also updates the author's reputation based on the vote type and sends a notification to the author. The READ COMMITTED isolation level is used to allow concurrent read operations while ensuring only the latest committed vote state is read and applied. This prevents issues like dirty reads, where an uncommitted vote could affect the author's reputation.
Isolation level	READ COMMITTED

SQL code

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Add or update vote on post
INSERT INTO InterationPosts (userId, postId, liked)
VALUES (:userId, :postId, :userId)
ON CONFLICT (userId, postId) DO UPDATE
SET liked = :liked;

-- Update the reputation of the author
UPDATE User
SET reputation = reputation + (CASE WHEN :liked = 'up' THEN 1 ELSE -1 END)
WHERE id = (SELECT ownerId FROM Posts WHERE post_id = :post_id);

-- Create a notification for the author of the post about the vote
INSERT INTO UpvoteOnPostNotification (notfId, is_read, created_at, emitter, receiver, post)
VALUES (:notification_id, FALSE, CURRENT_TIMESTAMP, :userId,
        (SELECT ownerId FROM Posts WHERE post_id = :postId), :postId);

COMMIT;
```

Transaction	TRAN02
Description	Vote on a Comment
Justification	This transaction allows users to vote on a comment by adding a new vote or updating an existing one. It updates the author's reputation based on the vote type and notifies the author. The READ COMMITTED isolation level is applied to allow concurrent reads while ensuring only committed vote data is used, avoiding dirty reads that could impact the reputation calculation.
Isolation level	READ COMMITTED

SQL code

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Add or update vote on comment
INSERT INTO InterationComments (userId, comment_id, liked)
VALUES (:userId, :comment_id, :userId)
ON CONFLICT (userId, comment_id) DO UPDATE
SET liked = :liked;

-- Update the reputation of the author
UPDATE User
SET reputation = reputation + (CASE WHEN :liked = 'up' THEN 1 ELSE -1 END)
WHERE id = (SELECT ownerId FROM Comments WHERE comment_id = :comment_id);

-- Create a notification for the author of the comment about the vote
INSERT INTO UpvoteOnCommentNotification (notfId, is_read, created_at, emitter, receiver, liked_comment)
VALUES (:notification_id, FALSE, CURRENT_TIMESTAMP, :userId,
        (SELECT ownerId FROM Comments WHERE comment_id = :comment_id), :comment_id);

COMMIT;
```

Transaction	TRAN03
Description	Delete a User
Justification	This transaction completely removes a user from the system, including all associated records such as votes, posts, comments, and follower relationships. The SERIALIZABLE isolation level helps to prevent any concurrent access or modifications to the user's data during the deletion process, ensuring data integrity and avoiding orphaned references or inconsistent states across related tables.
Isolation level	SERIALIZABLE

SQL code

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Delete votes by the user on posts and comments
DELETE FROM InterationPosts WHERE userId = :user_id;
DELETE FROM InterationComments WHERE userId = :user_id;

-- Update the user's posts and comments so that they now belong to the anonymous user
UPDATE Posts SET ownerId = 1 WHERE ownerId = :user_id;
UPDATE Comments SET ownerId = 1 WHERE ownerId = :user_id;

-- Remove follower relationships
DELETE FROM follwed_users WHERE userId1 = :user_id OR userId2 = :user_id;

-- Delete the user record
DELETE FROM User WHERE user_id = :user_id;

COMMIT;
```

Transaction	TRAN04
Description	Report a Comment
Justification	This transaction allows a user to report a comment, initiating a process for moderation. It includes inserting the report details and creating a reference to the specific comment. SERIALIZABLE isolation ensures that the report is unique and prevents duplicate reports on the same comment. Additionally, it prevents other users from reporting the same comment simultaneously with an identical status.
Isolation level	SERIALIZABLE

SQL code

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Insert a new report
INSERT INTO Report (reporter_id, reason, status, created_at)
VALUES (:reporter_id, :reason, 'pending', CURRENT_TIMESTAMP)
RETURNING report_id INTO :new_report_id;

-- Link the report to the comment
INSERT INTO CommentReport (rID, reported_comment)
VALUES (:new_report_id, :comment_id);

COMMIT;
```

Transaction	TRAN05
Description	Create a Comment

Transaction	TRAN05
Justification	This transaction handles the insertion of a new comment on a post and, if applicable, notifies the post's author about the new comment. The READ COMMITTED isolation level ensures that only committed data is used when fetching the author's ID, which prevents dirty reads while allowing concurrent comment insertions across different posts.
Isolation level	READ COMMITTED

SQL code

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Insert the new comment
INSERT INTO Comments (comment_id, body, created_at, updated_at, upvotes, downvotes, post, reply_to, ownerId)
VALUES (:comment_id, :body, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP, 0, 0, :post_id, :reply_to_id, :owner_id)
RETURNING comment_id INTO :new_comment_id;

-- Notify the post author about the new comment
INSERT INTO CommentNotification (notfId, is_read, created_at, emitter, receiver, comment)
VALUES (:notification_id, FALSE, CURRENT_TIMESTAMP, :owner_id,
        (SELECT ownerId FROM Posts WHERE post_id = :post_id), :new_comment_id);

COMMIT;
```

Transaction	TRAN06
Description	Report a Post
Justification	his transaction handles the reporting of a post by inserting a new report and linking it to the reported post. The SERIALIZABLE isolation level ensures the uniqueness of reports and prevents duplicate reports on the same post during concurrent operations.
Isolation level	SERIALIZABLE

SQL code

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Insert a new report
INSERT INTO Report (report_id, reason, created_at, status, reporter_id, reported_user, assigned_admin)
VALUES (:report_id, :reason, CURRENT_TIMESTAMP, 'pending', :reporter_id,
        (SELECT ownerId FROM Posts WHERE post_id = :post_id), :assigned_admin_id)
RETURNING report_id INTO :new_report_id;

-- Link the report to the post
INSERT INTO PostReport (rID, reported_post)
VALUES (:new_report_id, :post_id);

COMMIT;
```

Transaction	TRAN07
Description	Report a User
Justification	This transaction allows users to report another user, creating a record in the Report table and linking it to the reported user. The SERIALIZABLE isolation level prevents duplicate reports on the same user with identical statuses.
Isolation level	READ COMMITTED

SQL code

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Insert a new report
INSERT INTO Report (report_id, reason, created_at, status, reporter_id, reported_user, assigned_admin)
VALUES (:report_id, :reason, CURRENT_TIMESTAMP, 'pending', :reporter_id, :reported_user_id, :assigned_admin_id);

-- Link the report to the user
INSERT INTO UserReport (rID)
VALUES (:report_id);

COMMIT;
```

Annex A. SQL Code

> This annex contains the database scripts (creation and population), the complete code of each script can be accessed here schema.sql and populate.sql.

A.1. Database schema

```
CREATE SCHEMA IF NOT EXISTS lbaw24104;
DROP TYPE IF EXISTS activity_status;
DROP TYPE IF EXISTS report_status;
CREATE TYPE activity_status AS ENUM('active', 'flagged', 'deleted');
CREATE TYPE report_status AS ENUM('resolved', 'pending', 'dismissed');

DROP TABLE IF EXISTS Users CASCADE;
DROP TABLE IF EXISTS Admins CASCADE;
DROP TABLE IF EXISTS Blocked CASCADE;
DROP TABLE IF EXISTS Images CASCADE;
DROP TABLE IF EXISTS Tag CASCADE;
DROP TABLE IF EXISTS Posts CASCADE;
DROP TABLE IF EXISTS Comments CASCADE;
DROP TABLE IF EXISTS followed_tags CASCADE;
DROP TABLE IF EXISTS follwed_users CASCADE;
DROP TABLE IF EXISTS Post_tags CASCADE;
DROP TABLE IF EXISTS InterationComments CASCADE;
DROP TABLE IF EXISTS InterationPosts CASCADE;
DROP TABLE IF EXISTS favorite_posts CASCADE;
DROP TABLE IF EXISTS Report CASCADE;
DROP TABLE IF EXISTS UserReport CASCADE;
DROP TABLE IF EXISTS CommentReport CASCADE;
DROP TABLE IF EXISTS PostReport CASCADE;
DROP TABLE IF EXISTS Block_appeal CASCADE;
DROP TABLE IF EXISTS UpvoteOnPostNotification CASCADE;
DROP TABLE IF EXISTS UpvoteOnCommentNotification CASCADE;
DROP TABLE IF EXISTS CommentNotification CASCADE;


CREATE TABLE Users(
    user_id SERIAL PRIMARY KEY NOT NULL,
    username TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL,
    hash_password TEXT NOT NULL,
    reputation INT DEFAULT 0 NOT NULL
);


CREATE TABLE Admins(
    admin_id INT PRIMARY KEY REFERENCES Users (user_id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

```
CREATE TABLE Blocked(
    blocked_id INT PRIMARY KEY REFERENCES Users (user_id) ON UPDATE CASCADE ON DELETE CASCADE
);


CREATE TABLE Images(
    image_id SERIAL PRIMARY KEY NOT NULL,
    user_id INT REFERENCES Users (user_id) ON UPDATE CASCADE ON DELETE CASCADE,
    path TEXT NOT NULL
);


CREATE TABLE Tag(
    tag_id SERIAL PRIMARY KEY NOT NULL,
    name TEXT UNIQUE NOT NULL
);


CREATE TABLE Posts(
    post_id SERIAL PRIMARY KEY NOT NULL,
    title TEXT NOT NULL,
    body TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL CHECK(created_at<=CURRENT_TIMESTAMP),
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP CHECK(updated_at>=created_at),
    upvotes INT DEFAULT 0 NOT NULL,
    downvotes INT DEFAULT 0 NOT NULL,
    ownerId INT REFERENCES Users (user_id) ON UPDATE CASCADE
);


CREATE TABLE Comments(
    comment_id SERIAL PRIMARY KEY NOT NULL,
    body TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL CHECK(created_at<=CURRENT_TIMESTAMP),
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP CHECK(updated_at>=created_at),
    upvotes INT DEFAULT 0 NOT NULL,
    downvotes INT DEFAULT 0 NOT NULL,
    post INT REFERENCES Posts (post_id) ON UPDATE CASCADE,
    reply_to INT REFERENCES Comments (comment_id) ON UPDATE CASCADE,
    ownerId INT REFERENCES Users (user_id) ON UPDATE CASCADE
);


CREATE TABLE followed_tags(
    tagId INT REFERENCES Tag (tag_id) ON UPDATE CASCADE,
    userId INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    PRIMARY KEY (tagId,userId)
);


CREATE TABLE follwed_users(
    userId1 INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    userId2 INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    PRIMARY KEY (userId1,userId2)
);


CREATE TABLE Post_tags(
    post INT REFERENCES Posts (post_id) ON UPDATE CASCADE,
    tag INT REFERENCES Tag (tag_id) ON UPDATE CASCADE,
    PRIMARY KEY (post,tag)
);

--upvotes / downvotes
CREATE TABLE InterationComments(
    userId INT REFERENCES Users (user_id) ON UPDATE CASCADE,
```

```
    comment_id INT REFERENCES Comments (comment_id) ON UPDATE CASCADE,
    liked BOOLEAN NOT NULL,
    PRIMARY KEY (userId,comment_id)
);


CREATE TABLE InterationPosts(
    userId INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    postId INT REFERENCES Posts (post_id) ON UPDATE CASCADE,
    liked BOOLEAN NOT NULL,
    PRIMARY KEY (userId,postId)
);


CREATE TABLE favorite_posts(
    userId INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    postId INT REFERENCES Posts (post_id) ON UPDATE CASCADE,
    PRIMARY KEY (userId,postId)
);
```

```
CREATE TABLE Report(
    report_id SERIAL PRIMARY KEY NOT NULL,
    reason TEXT NOT NULL,
    created_at TIMESTAMP CHECK(created_at<=CURRENT_TIMESTAMP),
    report_status TEXT NOT NULL DEFAULT 'pending',
    reporter_id INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    reported_user INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    assigned_admin INT REFERENCES Admins (admin_id) ON UPDATE CASCADE
);

CREATE TABLE UserReport(
    rID INT REFERENCES Report (report_id) ON UPDATE CASCADE,
    PRIMARY KEY (rID)

);


CREATE TABLE CommentReport(
    rID INT REFERENCES Report (report_id) ON UPDATE CASCADE,
    reported_comment INT REFERENCES Comments (comment_id) ON UPDATE CASCADE,
    PRIMARY KEY (rID)

);
```

```
CREATE TABLE PostReport(
    rID INT REFERENCES Report (report_id) ON UPDATE CASCADE,
    reported_post INT REFERENCES Posts (post_id) ON UPDATE CASCADE,
    PRIMARY KEY (rID)

);


CREATE TABLE Block_appeal(
    report_id INT REFERENCES Report(report_id) ON UPDATE CASCADE,
    user_id INT REFERENCES Users(user_id) ON UPDATE CASCADE,
    reason TEXT NOT NULL,
    PRIMARY KEY (report_id)
);


CREATE TABLE UpvoteOnPostNotification(
    notfId SERIAL PRIMARY KEY,
    is_read BOOLEAN DEFAULT false NOT NULL,
    created_at TIMESTAMP CHECK(created_at<=CURRENT_TIMESTAMP),
```

```
    emitter INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    receiver INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    post INT REFERENCES Posts (post_id) ON UPDATE CASCADE
);


CREATE TABLE UpvoteOnCommentNotification(
    notfId SERIAL PRIMARY KEY,
    is_read BOOLEAN DEFAULT false NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP CHECK(created_at<=CURRENT_TIMESTAMP),
    emitter INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    receiver INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    liked_comment INT REFERENCES Comments (comment_id) ON UPDATE CASCADE
);


CREATE TABLE CommentNotification(
    notfId SERIAL PRIMARY KEY,
    is_read BOOLEAN DEFAULT false NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP CHECK(created_at<=CURRENT_TIMESTAMP),
    emitter INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    receiver INT REFERENCES Users (user_id) ON UPDATE CASCADE,
    comment INT REFERENCES Comments (comment_id) ON UPDATE CASCADE
);


CREATE INDEX comment_notification_date ON CommentNotification USING btree (created_at);
CLUSTER CommentNotification USING comment_notification_date;



CREATE INDEX post_tags_tag ON Post_tags USING btree (tag);
CLUSTER Post_tags USING post_tags_tag;



CREATE INDEX comments_post ON Comments(post);



ALTER TABLE Posts
ADD COLUMN tsvectors TSVECTOR;

DROP FUNCTION IF EXISTS post_search_update;
CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.title), 'A') ||
            setweight(to_tsvector('english', NEW.body), 'B')
        );
    ELSIF TG_OP = 'UPDATE' THEN
        IF (NEW.title <> OLD.title OR NEW.body <> OLD.body) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.title), 'A') ||
                setweight(to_tsvector('english', NEW.body), 'B')
            );
        END IF;
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER post_search_update
BEFORE INSERT OR UPDATE ON Posts
FOR EACH ROW
EXECUTE PROCEDURE post_search_update();
```

```
CREATE INDEX post_content ON Posts USING GIN (tsvectors);



ALTER TABLE Users
ADD COLUMN tsvectors TSVECTOR;

DROP FUNCTION IF EXISTS user_search_update;
CREATE FUNCTION user_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' OR (TG_OP = 'UPDATE' AND NEW.username <> OLD.username) THEN
        NEW.tsvectors = to_tsvector('english', NEW.username);
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER user_search_update
BEFORE INSERT OR UPDATE ON Users
FOR EACH ROW
EXECUTE PROCEDURE user_search_update();

CREATE INDEX user_username ON Users USING GIN (tsvectors);



ALTER TABLE Comments
ADD COLUMN tsvectors TSVECTOR;

DROP FUNCTION IF EXISTS comment_search_update;
CREATE FUNCTION comment_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = to_tsvector('english', NEW.body);
    ELSIF TG_OP = 'UPDATE' THEN
        IF (NEW.body <> OLD.body) THEN
            NEW.tsvectors = to_tsvector('english', NEW.body);
        END IF;
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER comment_search_update
BEFORE INSERT OR UPDATE ON Comments
FOR EACH ROW
EXECUTE PROCEDURE comment_search_update();

CREATE INDEX comment_content ON Comments USING GIN (tsvectors);

DROP FUNCTION IF EXISTS verify_report;
CREATE FUNCTION verify_report() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF EXISTS (SELECT * FROM Report WHERE NEW.reporter_id = reporter_id AND NEW.reported_user = reported_user AND r
        THEN RAISE EXCEPTION 'Report already done and waiting for resolotion';
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER verify_report
    BEFORE INSERT ON Report
    FOR EACH ROW
    EXECUTE PROCEDURE verify_report();
```

```
DROP FUNCTION IF EXISTS no_likes_on_own_post;
CREATE FUNCTION no_likes_on_own_post() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF (SELECT ownerId FROM Posts WHERE post_id = NEW.postId) = NEW.userId
        THEN RAISE EXCEPTION 'Users can not interact with their own posts';
        END IF;

        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER no_likes_on_own_post
    BEFORE INSERT ON InterationPosts
    FOR EACH ROW
    EXECUTE PROCEDURE no_likes_on_own_post();

DROP FUNCTION IF EXISTS no_likes_on_own_comment;
CREATE FUNCTION no_likes_on_own_comment() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF (SELECT ownerId FROM Comments WHERE comment_id = NEW.comment_id) = NEW.userId
        THEN RAISE EXCEPTION 'Users can not interact with their own comments';
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER no_likes_on_own_comment
    BEFORE INSERT ON InterationComments
    FOR EACH ROW
    EXECUTE PROCEDURE no_likes_on_own_comment();

DROP FUNCTION IF EXISTS post_like_notification;
CREATE FUNCTION post_like_notification() RETURNS TRIGGER AS
$BODY$
    BEGIN
        INSERT INTO UpvoteOnPostNotification(emitter,receiver,post) VALUES (NEW.userId,(SELECT ownerId FROM Posts WHERE
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER post_like_notification
    AFTER INSERT ON InterationPosts
    FOR EACH ROW
    EXECUTE PROCEDURE post_like_notification();

DROP FUNCTION IF EXISTS comment_like_notification;
CREATE FUNCTION comment_like_notification() RETURNS TRIGGER AS
$BODY$
    BEGIN
        INSERT INTO UpvoteOnCommentNotification(emitter,receiver,liked_comment) VALUES (NEW.userId,(SELECT ownerId FROM
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER comment_like_notification
    AFTER INSERT ON InterationComments
    FOR EACH ROW
    EXECUTE PROCEDURE comment_like_notification();

DROP FUNCTION IF EXISTS comment_notification;
```

```
CREATE FUNCTION comment_notification() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF NEW.reply_to IS NOT NULL
        THEN INSERT INTO CommentNotification(emitter,receiver,comment) VALUES (NEW.ownerID,(SELECT ownerId FROM Comments
        ELSE INSERT INTO CommentNotification(emitter,receiver,comment) VALUES (NEW.ownerID,(SELECT ownerId FROM Posts WH
                END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER comment_notification
    AFTER INSERT ON Comments
    FOR EACH ROW
    EXECUTE PROCEDURE comment_notification();

DROP FUNCTION IF EXISTS delete_post_check;
CREATE FUNCTION delete_post_check() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF EXISTS (SELECT * FROM InterationPosts WHERE postId=OLD.post_id) OR EXISTS (SELECT * FROM Comments WHERE post
        THEN RAISE EXCEPTION 'Posts that already has either comments or interations can not be deleted';
        END IF;
        RETURN OLD;
    END
$BODY$
language plpgsql;

CREATE TRIGGER delete_post_check
    BEFORE DELETE ON Posts
    FOR EACH ROW
    EXECUTE PROCEDURE delete_post_check();


DROP FUNCTION IF EXISTS delete_comment_check;
CREATE FUNCTION delete_comment_check() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF EXISTS (SELECT * FROM InterationComments WHERE comment_id=OLD.comment_id) OR EXISTS (SELECT * FROM Comments W
        THEN RAISE EXCEPTION 'Comments that already have either replys or interations can not be deleted';
        END IF;
        RETURN OLD;
    END
$BODY$
language plpgsql;

CREATE TRIGGER delete_comment_check
    BEFORE DELETE ON Comments
    FOR EACH ROW
    EXECUTE PROCEDURE delete_comment_check();



DROP FUNCTION IF EXISTS update_reputation_posts;
CREATE FUNCTION update_reputation_posts() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF NEW.liked THEN UPDATE Posts SET upvotes = upvotes + 1 WHERE post_id = NEW.postId; UPDATE Users SET reputatio
        ELSE UPDATE Posts SET downvotes = downvotes + 1 WHERE post_id = NEW.postId; UPDATE Users SET reputation = reput
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER update_reputation_posts
```

```
    AFTER INSERT ON InterationPosts
    FOR EACH ROW
    EXECUTE PROCEDURE update_reputation_posts();

DROP FUNCTION IF EXISTS update_reputation_comments;
CREATE FUNCTION update_reputation_comments() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF NEW.liked THEN UPDATE Comments SET upvotes = upvotes + 1 WHERE comment_id = NEW.comment_id; UPDATE Users SET
        ELSE UPDATE Comments SET downvotes = downvotes + 1 WHERE comment_id = NEW.comment_id; UPDATE Users SET reputati
        END IF;
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER update_reputation_comments
    AFTER INSERT ON InterationComments
    FOR EACH ROW
    EXECUTE PROCEDURE update_reputation_comments();

DROP FUNCTION IF EXISTS update_reputation_posts_removed_like;
CREATE FUNCTION update_reputation_posts_removed_like() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF OLD.liked THEN UPDATE Posts SET upvotes = upvotes - 1 WHERE post_id = OLD.postId; UPDATE Users SET reputatio
        ELSE UPDATE Post SET downvotes = downvotes - 1 WHERE post_id = OLD.postId; UPDATE Users SET reputation = reputa
        END IF;
        RETURN OLD;
    END
$BODY$
language plpgsql;

CREATE TRIGGER update_reputation_posts_removed_like
    BEFORE DELETE ON InterationPosts
    FOR EACH ROW
    EXECUTE PROCEDURE update_reputation_posts_removed_like();

DROP FUNCTION IF EXISTS update_reputation_comments_removed_like;
CREATE FUNCTION update_reputation_comments_removed_like() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF OLD.liked THEN UPDATE Comments SET upvotes = upvotes - 1 WHERE comment_id = OLD.comment_id; UPDATE Users SET
        ELSE UPDATE Comments SET downvotes = downvotes - 1 WHERE comment_id = OLD.comment_id; UPDATE Users SET reputati
        END IF;
        RETURN OLD;
    END
$BODY$
language plpgsql;

CREATE TRIGGER update_reputation_comments_removed_like
    BEFORE DELETE ON InterationComments
    FOR EACH ROW
    EXECUTE PROCEDURE update_reputation_comments_removed_like();


DROP FUNCTION IF EXISTS block_appeal_verify;
CREATE FUNCTION block_appeal_verify() RETURNS TRIGGER AS
$BODY$
    BEGIN
        IF EXISTS(SELECT * FROM Report WHERE report_id=NEW.report_id AND report_status='pending')
        THEN RAISE EXCEPTION 'Report still pending wait until resulotion';
        END IF;
        IF EXISTS(SELECT * FROM Report WHERE report_id=NEW.report_id AND (report_status='resolved' OR report_status='di
        THEN UPDATE Report SET report_status='pending' WHERE report_id=NEW.report_id;
        END IF;
        RETURN NEW;
```

```
    END
$BODY$
language plpgsql;

CREATE TRIGGER block_appeal_verify
    BEFORE INSERT ON Block_appeal
    FOR EACH ROW
    EXECUTE PROCEDURE block_appeal_verify();


DROP FUNCTION IF EXISTS give_admin_to_report;
CREATE FUNCTION give_admin_to_report() RETURNS TRIGGER AS
$BODY$
    BEGIN

                UPDATE Report SET assigned_admin = (SELECT admin_id FROM Admins WHERE admin_id = (SELECT assigned_admin
        RETURN NEW;
    END
$BODY$
language plpgsql;

CREATE TRIGGER give_admin_to_report
AFTER INSERT ON Report
FOR EACH ROW
EXECUTE PROCEDURE give_admin_to_report();
```

A.2. Database population

```
insert into Users (username, email, hash_password, reputation) values ('dcarlens0', 'rmaven0@dailymotion.com', '$2a$
insert into Users (username, email, hash_password, reputation) values ('mclarycott1', 'cweld1@infoseek.co.jp', '$2a$
insert into Users (username, email, hash_password, reputation) values ('qmounce2', 'shanne2@businesswire.com', '$2a$
insert into Users (username, email, hash_password, reputation) values ('llyles3', 'lskidmore3@msn.com', '$2a$04$1Gwh
insert into Users (username, email, hash_password, reputation) values ('ehampson4', 'jayers4@weather.com', '$2a$04$3

INSERT INTO Admins (admin_id) VALUES
(1),
(2),
(3),
(4),
(5);

INSERT INTO Tag (name) VALUES
('Technology'),
('Science'),
('Health'),
('Education'),
('Travel'),
('Food');



INSERT INTO Posts (title, body, upvotes, downvotes, ownerId, created_at, updated_at) VALUES
('The Future of AI', 'Exploring how artificial intelligence will change our world...', 120, 10, 1, '2024-01-15 08:00
('Top 10 Healthy Foods', 'A guide to the best foods for a healthy lifestyle...', 85, 2, 2, '2024-01-18 09:15:00', NU
('Travel Tips for 2024', 'Get ready for new destinations and experiences...', 60, 5, 3, '2024-01-20 10:30:00', '2024
('Finance Hacks', 'Simple ways to manage your finances...', 90, 7, 4, '2024-01-22 11:00:00', NULL),
('Digital Learning Trends', 'What you need to know about digital education...', 110, 3, 5, '2024-01-25 14:10:00', NU



INSERT INTO Comments (body, upvotes, downvotes, post, reply_to, ownerId, created_at, updated_at) VALUES
('Interesting perspective!', 15, 2, 1, NULL, 6, '2024-01-15 09:00:00', NULL),
('I disagree with this point...', 8, 10, 2, NULL, 7, '2024-01-18 09:30:00', NULL),
('Well explained, thanks!', 20, 1, 3, NULL, 8, '2024-01-20 10:45:00', '2024-01-25 11:00:00'),
('Great read!', 12, 0, 4, NULL, 9, '2024-01-22 11:00:00', NULL),
('Could you elaborate on this?', 10, 0, 5, NULL, 10, '2024-01-25 13:10:00', NULL);


INSERT INTO InterationPosts (userId, postId, liked) VALUES
(22, 1, true),
(2, 1, false),
(3, 2, true),
(4, 3, true),
(5, 4, false);


INSERT INTO InterationComments (userId, comment_id, liked) VALUES
(22, 1, true),
(2, 2, false),
(3, 3, true),
(4, 4, true),
(5, 5, false),
(6, 6, true),
(7, 7, false);

INSERT INTO followed_tags (tagId, userId) VALUES
(1, 22),
(2, 3),
(3, 4),
(4, 5),
(5, 6),
(6, 8);
```

```
INSERT INTO follwed_users (userId1, userId2) VALUES
(22, 2),
(2, 3),
(3, 4),
(4, 5),
(5, 6),
(6, 7);


INSERT INTO favorite_posts (userId, postId) VALUES
(55, 1),
(22, 3),
(23, 5),
(41, 7),
(15, 9),
(26, 2);



INSERT INTO Post_tags (post, tag) VALUES
(1, 1),
(1, 3),
(2, 2),
(2, 4),
(3, 5),
(4, 1);
```

Revision history

Artifact - A4

> Editor: Rafael Campeão

October 23:

1. Added: First UML diagram

> By:

- João Cordeiro
- Luana Lima
- Miguel Neri
- Rafael Campeão

November 6:

1. Added: Final UML diagram

> By:

- Rafael Campeão

Artifact - A5

> Editor: João Cordeiro

October 16:

1. Added: Relational Schema

> By:

- João Cordeiro

October 23:

1. Added: Domains and Schema Validation
2. Updated: Relational Schema

> By:

- João Cordeiro
- Luana Lima
- Miguel Neri
- Rafael Campeão

November 5:

1. Updated: Relational Schema, Domains and Schema Validation

> By:

- João Cordeiro
- Luana Lima
- Miguel Neri
- Rafael Campeão

Artifact - A6

> Editor: Miguel Neri

November 4:

1. Added: Database Workload

> By:

- Luana Lima
- Miguel Neri

November 5:

1. Added: Proposed Indices
2. Added: Transactions
3. Added: Sql file with the schema for the database

> By:

- João Cordeiro
- Luana Lima
- Miguel Neri
- Rafael Campeão

November 6:

1. Added: Triggers
2. Added: Sql file with the population of the database

> By:

- João Cordeiro
- Luana Lima
- Miguel Neri
- Rafael Campeão

GROUP104, 06/11/2024

- João Cordeiro, up202205682@up.pt
- Luana Lima, up202205682@up.pt (editor)
- Miguel Neri, up202006475@up.pt
- Rafael Campeão, up202207553@up.pt

Comments