lbaw2356_ebd.md

EBD: Database Specification Component

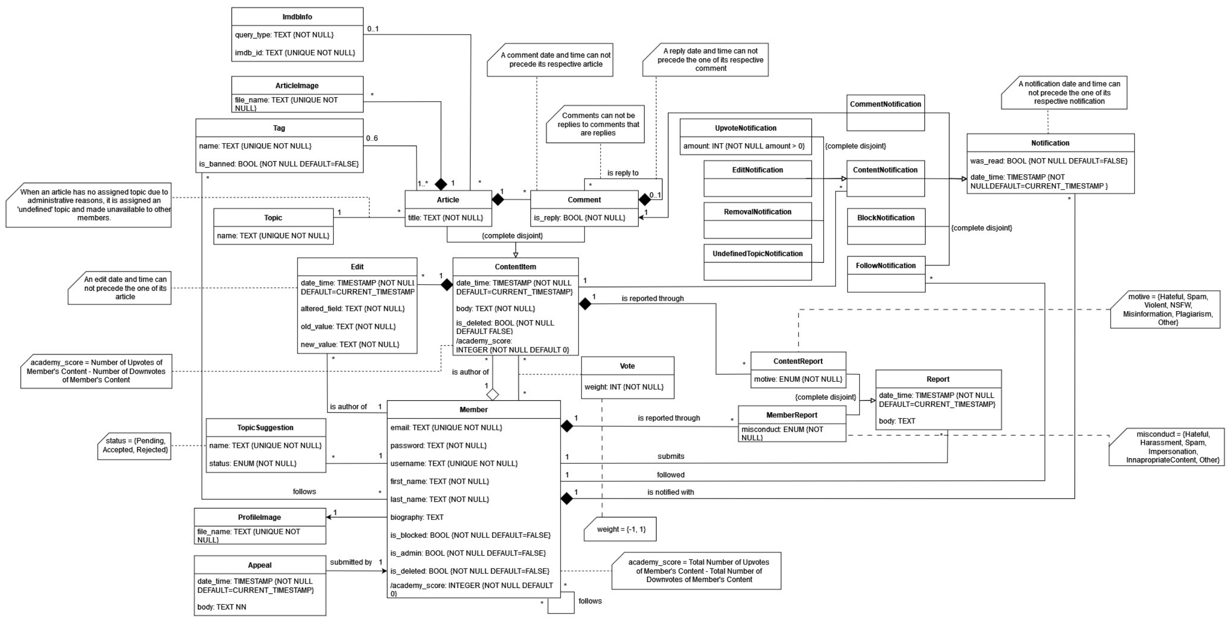
The Popcorn Post aims to be the go-to collaborative news platform where cinema and entertainment enthusiasts actively engage, share, and explore their passion while fostering a trusted and vibrant community.

A4: Conceptual Data Model

The Conceptual Domain Model serves as a comprehensive description of the domain's entities and the relationships between them, all portrayed in a UML class diagram.

The diagram in Figure 1 is a visual representation of the fundamental organizational entities, the relationships between them, the correspondent domains and attributes, and the multiplicity of associations, designed for 'The Popcorn Post' collaborative news website.

1. Class diagram



ImdbInfo
query_type: TEXT {NOT NULL}
imdb_id: TEXT {UNIQUE NOT NULL}
0..1
ArticleImage
file_name: TEXT {UNIQUE NOT NULL}
Tag
name: TEXT {UNIQUE NOT NULL}
is_banned: BOOL {NOT NULL DEFAULT=FALSE}
0..6
1..*
A comment date and time can not precede its respective article
A reply date and time can not precede the one of its respective comment
Comments can not be replies to comments that are replies
is reply to
0..1
When an article has no assigned topic due to administrative reasons, it is assigned an 'undefined' topic and made unavailable to other members.
Topic
name: TEXT {UNIQUE NOT NULL}
Article
title: TEXT {NOT NULL}
Comment
is_reply: BOOL {NOT NULL}
{complete disjoint}
CommentNotification
UpvoteNotification
amount: INT {NOT NULL amount > 0}
EditNotification
RemovalNotification
UndefinedTopicNotification
ContentNotification
BlockNotification
FollowNotification
A notification date and time can not precede the one of its respective notification
Notification
was_read: BOOL {NOT NULL DEFAULT=FALSE}
date_time: TIMESTAMP {NOT NULLDEFAULT=CURRENT_TIMESTAMP }
An edit date and time can not precede the one of its article
Edit
date_time: TIMESTAMP {NOT NULL DEFAULT=CURRENT_TIMESTAMP
altered_field: TEXT {NOT NULL}
old_value: TEXT {NOT NULL}
new_value: TEXT {NOT NULL}
ContentItem
date_time: TIMESTAMP {NOT NULL DEFAULT=CURRENT_TIMESTAMP}
body: TEXT {NOT NULL}
is_deleted: BOOL {NOT NULL DEFAULT FALSE}
/academy_score: INTEGER {NOT NULL DEFAULT 0}
is reported through
motive = {Hateful, Spam, Violent, NSFW, Misinformation, Plagiarism, Other}
academy_score = Number of Upvotes of Member's Content - Number of Downvotes of Member's Content
is author of
Vote
weight: INT {NOT NULL}
ContentReport
motive: ENUM {NOT NULL}
Report
date_time: TIMESTAMP {NOT NULL DEFAULT=CURRENT_TIMESTAMP}
body: TEXT
MemberReport
misconduct: ENUM {NOT NULL}
misconduct = {Hateful, Harassment, Spam, Impersonation, InnapropriateContent, Other}
status = {Pending, Accepted, Rejected}
TopicSuggestion
name: TEXT {UNIQUE NOT NULL}
status: ENUM {NOT NULL}
Member
email: TEXT {UNIQUE NOT NULL}
password: TEXT {NOT NULL}
username: TEXT {UNIQUE NOT NULL}
first_name: TEXT {NOT NULL}
last_name: TEXT {NOT NULL}
biography: TEXT
is_blocked: BOOL {NOT NULL DEFAULT=FALSE}
is_admin: BOOL {NOT NULL DEFAULT=FALSE}
is_deleted: BOOL {NOT NULL DEFAULT=FALSE}
/academy_score: INTEGER {NOT NULL DEFAULT 0}
submits
followed
is notified with
follows
ProfileImage
file_name: TEXT {UNIQUE NOT NULL}
weight = {-1, 1}
Appeal
date_time: TIMESTAMP {NOT NULL DEFAULT=CURRENT_TIMESTAMP}
body: TEXT NN
submitted by
academy_score = Total Number of Upvotes of Member's Content - Total Number of Downvotes of Member's Content

Figure 1: The Popcorn Post's conceptual data model.

2. Additional Business Rules

- **BR01**. A member is not allowed to follow themselves.

- **BR02**. A member is not allowed to report themselves.
- **BR03**. A member is not allowed to report content authored by themselves.
- **BR04**. A new member is assigned a default profile picture.
- **BR05**. Only a blocked member can submit an appeal.

A5: Relational Schema, validation and schema refinement

This artefact is a crucial component derived from the Conceptual Data Model. It comprehensively outlines the structure of each relation, including attributes, domains, primary and foreign keys, and critical integrity rules such as UNIQUE, DEFAULT, NOT NULL, and CHECK constraints. This artefact serves as the foundation for establishing the structural framework and data integrity of the relational database, ensuring that it accurately reflects the conceptual model while adhering to database design best practices.

1. Relational Schema

Relation reference	Relation Compact Notation
R01	member(id, email **UK NN**, password **NN**, username **UK NN**, first_name **NN**, last_name **NN**, biography, is_blocked **NN DF** FALSE, is_admin **NN DF** FALSE, is_deleted **NN DF** FALSE, academy_score **NN DF** 0, profile_image_id -> profile_image **NN DF** 0)
R02	follow_member(follower_id -> member, followed_id -> member)
R03	profile_image(id, file_name **UK NN**)
R04	appeal(id, date_time **NN DF** CURRENT_TIMESTAMP, body **NN**, submitter_id -> member **UK NN**)
R05	topic(id, name **UK NN**)
R06	topic_suggestion(id, name **UK NN**, status **NN CK** status **IN** Statuses, suggester_id -> member **NN**)
R07	article(id-> content_item, title **NN**, topic_id -> topic **NN DF** 0, imdb_info_id -> imdb_info)
R08	imdb_info(id, query_type **NN**, imdb_id **UK NN**)
R09	article_image(id, file_name **UK NN**, article_id -> article **NN**)

Relation reference	Relation Compact Notation
R10	tag(id, name **UK NN**, is_banned **NN DF** FALSE)
R11	tag_article(tag_id -> tag, article_id -> article)
R12	follow_tag(tag_id -> tag, member_id -> member)
R13	comment(id-> content_item, is_reply **NN**, article_id -> article **NN**, reply_id -> comment)
R14	content_item(id, date_time **NN DF** CURRENT_TIMESTAMP, body **NN**, is_deleted **NN DF** FALSE, academy_score **NN DF** 0, author_id -> member **NN**)
R15	edit(id, date_time **NN DF** CURRENT_TIMESTAMP, altered_field **NN**, old_value **NN**, new_value **NN**, content_item_id -> content_item **NN**, author_id -> member **NN**)
R16	vote(member_id -> member, content_item_id -> content_item, weight **NN CK** weight = 1 **OR** weight = -1)
R17	report(id, date_time **NN DF** CURRENT_TIMESTAMP, body, submitter_id -> member **NN**)
R18	content_report(id -> report, motive **NN CK** motive **IN** Motives, content_item_id -> content_item **NN**)
R19	member_report(id -> report, misconduct **NN CK** misconduct **IN** MisconductTypes, member_id -> member **NN**)
R20	notification(id, was_read **NN DF** FALSE, date_time **NN DF** CURRENT_TIMESTAMP, notified_id -> member **NN**)
R21	comment_notification(id -> notification, comment_id -> comment **NN**)
R22	content_notification(id -> notification, content_item_id -> content_item **NN**)
R23	upvote_notification(id -> content_notification, amount **NN CK** amount > 0)
R24	edit_notification(id -> content_notification)
R25	removal_notification(id -> content_notification)
R26	undefined_topic_notification(id -> content_notification)
R27	block_notification(id -> notification)

Relation reference	Relation Compact Notation
R28	follow_notification(<u>id</u> -> notification, follower_id -> member **NN**)

Legend:

- **UK** = UNIQUE KEY
- **NN** = NOT NULL
- **DF** = DEFAULT
- **CK** = CHECK.

2. Domains

Domain	VALUES
Motives	Hateful, Spam, Violent, NSFW, Misinformation, Plagiarism, Other
MisconductTypes	Hateful, Harassment, Spam, Impersonation, InnapropriateContent, Other
Statuses	Pending, Accepted, Rejected

3. Schema validation

TABLE R01	member
Keys	{id}, {email}, {username}
Functional Dependencies:	
FD0101	{id} → {email, password, username, first_name, last_name, biography, is_blocked, is_admin, is_deleted, academy_score, profile_image_id}
FD0102	{email} → {id, password, username, first_name, last_name, biography, is_blocked, is_admin, is_deleted, academy_score, profile_image_id}
FD0103	{username} → {id, email, password, first_name, last_name, biography, is_blocked, is_admin, is_deleted, academy_score, profile_image_id}
NORMAL FORM	BCNF

TABLE R02	follow_member
Keys	{follower_id, followed_id}
Functional Dependencies:	
NORMAL FORM	BCNF

TABLE R03	profile_image
Keys	{id}, {file_name}
Functional Dependencies:	
FD0301	{id} → {file_name}
FD0302	{file_name} → {id}
NORMAL FORM	BCNF

TABLE R04	appeal
Keys	{id}, {submitter_id}
Functional Dependencies:	
FD0401	{id} → {date_time, body, submitter_id}
FD0402	{submitter_id} → {id, date_time, body}
NORMAL FORM	BCNF

TABLE R05	topic
Keys	{id}, {name}
Functional Dependencies:	
FD0501	{id} → {name, proposer_id}
FD0502	{name} → {id, proposer_id}
NORMAL FORM	BCNF

TABLE R06	topic_suggestion
Keys	{id}
Functional Dependencies:	
FD0601	{id} → {name, status, suggester_id}

TABLE R06	topic_suggestion
NORMAL FORM	BCNF

TABLE R07	article
Keys	{id}
Functional Dependencies:	
FD0701	{id} → {title, topic_id, imdb_info_id}
NORMAL FORM	BCNF

TABLE R08	imdb_info
Keys	{id}, {imdb_id}
Functional Dependencies:	
FD0801	{id} → {query_type, imdb_id}
FD0802	{imdb_id} → {id, query_type}
NORMAL FORM	BCNF

TABLE R09	article_image
Keys	{id}, {file_name}
Functional Dependencies:	
FD0901	{id} → {file_name, article_id}
FD0902	{file_name} → {id, article_id}
NORMAL FORM	BCNF

TABLE R10	tag
Keys	{id}, {name}
Functional Dependencies:	
FD1001	{id} → {name, is_banned}
FD1002	{name} → {id, is_banned}
NORMAL FORM	BCNF

TABLE R11	tag_article
Keys	{tag_id, article_id}
Functional Dependencies:	
NORMAL FORM	BCNF

TABLE R12	follow_tag
Keys	{tag_id, member_id}
Functional Dependencies:	
NORMAL FORM	BCNF

TABLE R13	comment
Keys	{id}
Functional Dependencies:	
FD1301	{id} → {is_reply, article_id, reply_id}
NORMAL FORM	BCNF

TABLE R14	content_item
Keys	{id}
Functional Dependencies:	
FD1401	{id} → {date_time, body, is_deleted, academy_score, author_id}
NORMAL FORM	BCNF

TABLE R15	edit
Keys	{id}
Functional Dependencies:	
FD1501	{id} → {date_time, altered_field, old_value, new_value, content_item_id, author_id}
NORMAL FORM	BCNF

TABLE R16	vote
Keys	{member_id, content_item_id}
Functional Dependencies:	
FD1601	{member_id, content_item_id} → {weight}
NORMAL FORM	BCNF

TABLE R17	report
Keys	{id}
Functional Dependencies:	
FD1701	{id} → {date_time, body, submitter_id}
NORMAL FORM	BCNF

TABLE R18	content_report
Keys	{id}
Functional Dependencies:	
FD1801	{id} → {motive, content_item_id}
NORMAL FORM	BCNF

TABLE R19	member_report
Keys	{id}
Functional Dependencies:	
FD1901	{id} → {misconduct, member_id}
NORMAL FORM	BCNF

TABLE R20	notification
Keys	{id}
Functional Dependencies:	
FD2001	{id} → {was_read, date_time, notified_id}
NORMAL FORM	BCNF

TABLE R21	comment_notification
Keys	{id}
Functional Dependencies:	
FD2101	{id} → {comment_id}
NORMAL FORM	BCNF

TABLE R22	content_notification
Keys	{id}
Functional Dependencies:	
FD2201	{id} → {content_item_id}
NORMAL FORM	BCNF

TABLE R23	upvote_notification
Keys	{id}
Functional Dependencies:	
FD2301	{id} → {amount}
NORMAL FORM	BCNF

TABLE R24	edit_notification
Keys	{id}
Functional Dependencies:	
NORMAL FORM	BCNF

TABLE R25	removal_notification
Keys	{id}
Functional Dependencies:	
NORMAL FORM	BCNF

TABLE R26	undefined_topic_notification
Keys	{id}
Functional Dependencies:	

TABLE R26	undefined_topic_notification
NORMAL FORM	BCNF

TABLE R27	block_notification
Keys	{id}
Functional Dependencies:	
NORMAL FORM	BCNF

TABLE R28	follow_notification
Keys	{id}
Functional Dependencies:	
FD2801	{id} → {follower_id}
NORMAL FORM	BCNF

If the dependency X → Y is obvious or trivial, meaning that Y can be completely determined by the attributes in X (Y is a subset of X), it is considered to be in BCNF. Alternatively, if X is a superkey for the entire schema (relation), meaning that X uniquely identifies each row in the table, then the functional dependency X → Y is considered to be in BCNF. Since this is the case for all functional dependencies in the database, all relations are in BCNF.

A6: Indexes, triggers, transactions and database population

This artifact incorporates the database's physical schema, precise index identification, and data integrity support through triggers and user-defined functions. It also covers essential transactions to ensure data integrity amidst concurrent access, specifying isolation levels and justifications. Additionally, it details the database workload and provides a complete creation script, including the SQL for integrity constraints, indexes, triggers, and an insert script for the database population.

1. Database Workload

This section is meant to guide the database design in order to achieve the performance goals. It includes the expected number of tuples for each relation and also the estimated growth.

Relation reference	Relation Name	Order of magnitude	Estimated growth
R01	member	10k (tens of thousands)	10 (tens) / day
R02	follow_member	1M (millions)	100 (hundreds) / day
R03	profile_image	10k (tens of thousands)	10 (tens) / day
R04	appeal	100 (hundreds)	1 (units) / month
R05	topic	10 (tens)	1 / year
R06	topic_suggestion	100 (hundreds)	1 (units) / month
R07	article	100k (hundreds of thousands)	100 (hundreds) / day
R08	imdb_info	100k (hundreds of thousands)	100 (hundreds) / day
R09	article_image	100k (hundreds of thousands)	100 (hundreds) / day
R10	tag	100k (hundreds of thousands)	100 (hundreds) / day
R11	tag_article	1M (millions)	1k (thousands) / day
R12	follow_tag	1M (millions)	100 (hundreds) / day
R13	comment	1M (millions)	1k (thousands) / day
R14	content_item	1M (millions)	1k (thousands) / day
R15	edit	100k (hundreds of thousands)	10 (tens) / day
R16	vote	1B (billions)	10k (tens of thousands) / day

Relation reference	Relation Name	Order of magnitude	Estimated growth
R17	report	10k (tens of thousands)	10 (tens) / day
R18	content_report	10k (tens of thousands)	10 (tens) / day
R19	member_report	100 (hundreds)	1 (units) / day
R20	notification	1B (billions)	1k (thousands) / day
R21	comment_notification	1M (millions)	1k (thousands) / day
R22	content_notification	1B (billions)	100 (hundreds) / day
R23	upvote_notification	1B (billions)	100 (hundreds) / day
R24	edit_notification	100k (hundreds of thousands)	10 (tens) / day
R25	removal_notification	1k (thousands)	1 (units) / week
R26	undefined_topic_notification	1k (thousands)	1 (units) / month
R27	block_notification	100 (hundreds)	1 (units) / month
R28	follow_notification	1M (millions)	100 (hundreds) / day

2. Proposed Indexes

2.1. Performance Indexes

The following tables display the indexes proposed to improve the performance of the identified queries.

Index	IDX01
Relation	content_item
Attribute	date_time

Index	IDX01
Type	B-tree
Cardinality	Medium
Clustering	No
Justification	There are many content items. This index allows the news articles and comments to be searched by date faster. Most queries usually require ordering the content by date thus B-tree type index seems to be the most appropriate. Clustering doesn't seem needed.

SQL Code:

```
DROP INDEX IF EXISTS content_item_date;
CREATE INDEX content_item_date ON content_item USING btree (date_time);
```

Index	IDX02
Relation	content_item
Attribute	academy_score
Type	B-tree
Cardinality	Medium
Clustering	No
Justification	There are many content items. This index allows the news articles and comments to be searched by academy score. Most queries usually require ordering the content by date thus B-tree type index seems to be the most appropriate. Clustering doesn't seem needed.

SQL Code:

```
DROP INDEX IF EXISTS content_item_academy_score;
CREATE INDEX content_item_academy_score ON content_item USING btree
(academy_score);
```

Index	IDX03
Relation	article
Attribute	topic_id

Index	IDX03
Type	Hash
Cardinality	Low
Clustering	No
Justification	There are many news articles. This index allows the news articles to be searched filtered by topic. There is an interest in searching news articles with a certain topic thus Hash type index seems to be the most appropriate. Clustering doesn't seem needed.

SQL Code:

```
DROP INDEX IF EXISTS article_topic;
CREATE INDEX article_topic ON article USING hash (topic_id);
```

2.2. Full-text Search Indexes

Index	IDX04
Relation	content_item
Attribute	body, title (when content_item is an article) / body /when content_item is a comment
Type	GiST
Clustering	No
Justification	To provide full-text search capabilities for querying works based on matching their content in the "body" or "title" (in the case of an article), the chosen index type is GiST. GiST is selected for indexing these specific fields because it is well-suited for dynamic data.

SQL Code:

```
ALTER TABLE content_item
ADD COLUMN tsvectors tsvector;

CREATE FUNCTION content_item_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        IF (NEW.id IN (SELECT id FROM article)) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english',
(SELECT title FROM article WHERE article.id = NEW.id)), 'A') ||
                setweight(to_tsvector('english', NEW.body), 'B')
            );
        ELSE
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.body), 'B')
            );
        END IF;
 END IF;

 IF TG_OP = 'UPDATE' THEN
        IF (NEW.id IN (SELECT id FROM article)) THEN
            IF (NEW.body <> OLD.body) THEN
                NEW.tsvectors = (
                    setweight(to_tsvector('english',
(SELECT title FROM article WHERE article.id = NEW.id)), 'A') ||
                    setweight(to_tsvector('english', NEW.body), 'B')
                );
            END IF;
        ELSE
            IF (NEW.body <> OLD.body) THEN
                NEW.tsvectors = (
                    setweight(to_tsvector('english', NEW.body), 'B')
                );
            END IF;
        END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER content_item_search_update
BEFORE INSERT OR UPDATE ON content_item
FOR EACH ROW
EXECUTE PROCEDURE content_item_search_update();


CREATE FUNCTION article_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        IF (NEW.id IN (SELECT id FROM content_item)) THEN
            UPDATE content_item
            SET tsvectors = (
                setweight(to_tsvector('english',
(SELECT title FROM article WHERE article.id = NEW.id)), 'A') ||
                setweight(to_tsvector('english',
(SELECT body FROM content_item WHERE content_item.id = NEW.id)), 'B')
            )
            WHERE content_item.id = NEW.id;
        END IF;
 END IF;

 IF TG_OP = 'UPDATE' THEN
        IF (NEW.id IN (SELECT id FROM content_item)) THEN
```

```
            IF (NEW.title <> OLD.title) THEN
                UPDATE content_item
                SET tsvectors = (
                    setweight(to_tsvector('english',
(SELECT title FROM article WHERE article.id = NEW.id)), 'A') ||
                    setweight(to_tsvector('english',
(SELECT body FROM content_item WHERE content_item.id = NEW.id)), 'B')
                )
                WHERE content_item.id = NEW.id;
            END IF;
        END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER article_search_update
BEFORE INSERT OR UPDATE ON article
FOR EACH ROW
EXECUTE PROCEDURE article_search_update();

CREATE INDEX content_item_search ON content_item USING GIST (tsvectors);
```

Index	IDX05
Relation	member
Attribute	username, first_name, last_name
Type	GIN
Clustering	No
Justification	To provide full-text search capabilities for querying works based on matching their content in the "username", "first_name" or "last_name", the chosen index type is GIN. GIN is selected for indexing these specific fields because it is well-suited for static data.

SQL Code:

```
ALTER TABLE member
ADD COLUMN tsvectors tsvector;

CREATE FUNCTION member_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.username), 'A') ||
            setweight(to_tsvector('english', NEW.first_name), 'B') ||
            setweight(to_tsvector('english', NEW.last_name), 'B')
```

```
          );
   END IF;

   IF TG_OP = 'UPDATE' THEN
          IF (NEW.username <> OLD.username) THEN
              NEW.tsvectors = (
                  setweight(to_tsvector('english', NEW.username), 'A') ||
                  setweight(to_tsvector('english', NEW.first_name), 'B') ||
                  setweight(to_tsvector('english', NEW.last_name), 'B')
              );
          END IF;
   END IF;
   RETURN NEW;
  END $$

  LANGUAGE plpgsql;

  CREATE TRIGGER member_search_update
  BEFORE INSERT OR UPDATE ON member
  FOR EACH ROW
  EXECUTE PROCEDURE member_search_update();

  CREATE INDEX member_search ON member USING GIN (tsvectors);
```

3. Triggers

The following tables outline the triggers and custom functions created to ensure the integrity of the system's data.

Trigger	TRIGGER01
Description	Anonymizes user data and ensures data integrity when a user account is deleted.

SQL Code:

```
DROP FUNCTION IF EXISTS data_anonymization CASCADE;
DROP TRIGGER IF EXISTS data_anonymization ON member CASCADE;

CREATE FUNCTION data_anonymization()
RETURNS TRIGGER AS
$BODY$
BEGIN
    UPDATE member SET
        email = '',
        username = '',
        password = '',
        first_name = 'Deleted',
        last_name = 'User',
        biography = '',
        profile_image_id = 0,
        is_deleted = TRUE
    WHERE id = OLD.id;
    DELETE FROM profile_image WHERE id != 0 AND id = OLD.profile_image_id;
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER data_anonymization
    BEFORE UPDATE OF is_deleted ON member
    FOR EACH ROW
    WHEN (OLD.is_deleted = false AND NEW.is_deleted = true)
    EXECUTE PROCEDURE data_anonymization();
```

Trigger	TRIGGER02
Description	Handles the removal of votes and comments when an article is deleted.

SQL Code:

```
DROP FUNCTION IF EXISTS delete_content_item CASCADE;
DROP TRIGGER IF EXISTS delete_content_item ON content_item CASCADE;

CREATE FUNCTION delete_content_item()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (OLD.id IN (SELECT id FROM article)) THEN
        UPDATE content_item AS ct
        SET is_deleted = TRUE
        FROM comment AS c
        WHERE c.article_id = OLD.id AND c.id = ct.id AND c.is_reply = FALSE;

    ELSIF (OLD.id IN (SELECT id FROM comment)) THEN
        IF ((SELECT is_reply FROM comment WHERE comment.id = OLD.id) = FALSE)
 THEN
            UPDATE content_item AS ct
            SET is_deleted = TRUE
            FROM comment AS c
            WHERE c.reply_id = OLD.id AND c.id = ct.id AND c.is_reply = TRUE;
        END IF;
    END IF;

    UPDATE member
    SET academy_score = member.academy_score - OLD.academy_score
    WHERE member.id = OLD.author_id;
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER delete_content_item
    BEFORE UPDATE OF is_deleted ON content_item
    FOR EACH ROW
    WHEN (NEW.is_deleted = TRUE AND OLD.is_deleted = FALSE)
    EXECUTE PROCEDURE delete_content_item();
```

Trigger	TRIGGER03
Description	Handles data cleanup for permanently deleted articles.

SQL Code:

```
DROP FUNCTION IF EXISTS before_permanent_delete_article CASCADE;
DROP TRIGGER IF EXISTS before_permanent_delete_article ON article CASCADE;

CREATE FUNCTION before_permanent_delete_article()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM vote
    WHERE vote.content_item_id = OLD.id;

    DELETE FROM comment
    WHERE comment.article_id = OLD.id AND comment.is_reply = FALSE;

    DELETE FROM article_image
    WHERE article_image.article_id = OLD.id;
    RETURN OLD;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER before_permanent_delete_article
    BEFORE DELETE ON article
    FOR EACH ROW
    EXECUTE PROCEDURE before_permanent_delete_article();
```

Trigger	TRIGGER04
Description	Ensures the complete removal of data related to permanently deleted articles by eliminating the corresponding content items from the database.

SQL Code:

```
DROP FUNCTION IF EXISTS after_permanent_delete_article CASCADE;
DROP TRIGGER IF EXISTS after_permanent_delete_article ON article CASCADE;

CREATE FUNCTION after_permanent_delete_article()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM content_item
    WHERE content_item.id = OLD.id;
    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER after_permanent_delete_article
    AFTER DELETE ON article
    FOR EACH ROW
    EXECUTE PROCEDURE after_permanent_delete_article();
```

Trigger	TRIGGER05
Description	Maintains data integrity by removing related votes and replies when a comment is deleted.

SQL Code:

```
DROP FUNCTION IF EXISTS before_delete_comment();
DROP TRIGGER IF EXISTS before_delete_comment ON comment CASCADE;

CREATE FUNCTION before_delete_comment()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM vote
    WHERE vote.content_item_id = OLD.id;

    IF (OLD.is_reply = FALSE) THEN
        DELETE FROM comment
        WHERE comment.reply_id = OLD.id AND comment.is_reply = TRUE;
    END IF;

    RETURN OLD;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER before_delete_comment
    BEFORE DELETE ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE before_delete_comment();
```

Trigger	TRIGGER06
Description	Ensures the removal of associated content items after a comment has been deleted.

SQL Code:

```
DROP FUNCTION IF EXISTS after_delete_comment();
DROP TRIGGER IF EXISTS after_delete_comment ON comment CASCADE;

CREATE FUNCTION after_delete_comment()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM content_item
    WHERE content_item.id = OLD.id;
    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER after_delete_comment
    AFTER DELETE ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE after_delete_comment();
```

Trigger	TRIGGER07
Description	Handles data removal when a tag is banned.

SQL Code:

```
DROP FUNCTION IF EXISTS ban_tag();
DROP TRIGGER IF EXISTS ban_tag ON tag CASCADE;

CREATE FUNCTION ban_tag()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM tag_article
    WHERE tag_article.tag_id = OLD.id;
    DELETE FROM follow_tag
    WHERE follow_tag.tag_id = OLD.id;
    RETURN NEW;
END
```

```
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER ban_tag
    BEFORE UPDATE OF is_banned ON tag
    FOR EACH ROW
    WHEN (OLD.is_banned = false AND NEW.is_banned = true)
    EXECUTE PROCEDURE ban_tag();
```

Trigger	TRIGGER08
Description	Handles notifications when content is edited by someone other than the author.

SQL Code:

```
DROP FUNCTION IF EXISTS edit_content_notification();
DROP TRIGGER IF EXISTS edit_content_notification ON edit CASCADE;

CREATE FUNCTION edit_content_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN
    IF (NEW.author_id = (SELECT author_id FROM content_item
WHERE content_item.id = NEW.content_item_id)) THEN
        RETURN NULL;
    END IF;
    INSERT INTO notification (notified_id)
    VALUES ((SELECT author_id FROM content_item WHERE content_item.id =
NEW.content_item_id))
    RETURNING id INTO notification_id;
    INSERT INTO content_notification (id, content_item_id)
    VALUES (notification_id, NEW.content_item_id);
    INSERT INTO edit_notification (id)
    VALUES (notification_id);
    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER edit_content_notification
        AFTER INSERT ON edit
        FOR EACH ROW
        WHEN (NEW.altered_field != 'is_deleted')
    EXECUTE FUNCTION edit_content_notification();
```

Trigger	TRIGGER09
Description	Ensures that when a topic is removed, any associated articles have their topic reference set to 0.

SQL Code:

```
DROP FUNCTION IF EXISTS remove_topic();
DROP TRIGGER IF EXISTS remove_topic ON topic CASCADE;

CREATE FUNCTION remove_topic()
RETURNS TRIGGER AS
$BODY$
BEGIN
    UPDATE article
    SET topic_id = 0
    WHERE topic_id = OLD.id;
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER remove_topic
    BEFORE DELETE ON topic
    FOR EACH ROW
    EXECUTE PROCEDURE remove_topic();
```

Trigger	TRIGGER10
Description	Generates notifications when an article's topic changes from something else to undefined.

SQL Code:

```
DROP FUNCTION IF EXISTS generate_undefined_topic_notification(author_id INT,
 article_id INT);
DROP FUNCTION IF EXISTS notify_undefined_topic CASCADE;
DROP TRIGGER IF EXISTS notify_undefined_topic ON article CASCADE;

CREATE FUNCTION generate_undefined_topic_notification(author_id INT,
article_id INT)
RETURNS VOID AS
$BODY$
DECLARE
    notification_id INTEGER;
BEGIN
    INSERT INTO notification (notified_id)
    VALUES (author_id)
    RETURNING id INTO notification_id;
    INSERT INTO content_notification (id, content_item_id)
    VALUES (notification_id, article_id);
    INSERT INTO undefined_topic_notification (id)
    VALUES (notification_id);
    RETURN;
END
$BODY$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION notify_undefined_topic()
RETURNS TRIGGER AS
$BODY$
DECLARE
    author_id INTEGER;
BEGIN
    SELECT content_item.author_id, content_item.id
    FROM content_item
    WHERE content_item.id = OLD.id
    INTO author_id;
    EXECUTE generate_undefined_topic_notification(author_id, OLD.id);
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER notify_undefined_topic
    AFTER UPDATE OF topic_id ON article
    FOR EACH ROW
    WHEN (OLD.topic_id != 0 AND NEW.topic_id = 0)
    EXECUTE FUNCTION notify_undefined_topic();
```

Trigger	TRIGGER11
Description	Prevents a member from voting on their own content.

SQL Code:

```
DROP FUNCTION IF EXISTS vote_for_own_content CASCADE;
DROP TRIGGER IF EXISTS vote_for_own_content ON vote CASCADE;

CREATE FUNCTION vote_for_own_content()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (NEW.member_id = (SELECT author_id FROM content_item
WHERE content_item.id = NEW.content_item_id)) THEN
        RAISE EXCEPTION 'Members cannot vote on their own content';
    END IF;
    RETURN NEW;
```

```
END
$BODY$
LANGUAGE plpgsql;
CREATE TRIGGER vote_for_own_content
    BEFORE INSERT ON vote
    FOR EACH ROW
    EXECUTE FUNCTION vote_for_own_content();
```

Trigger	TRIGGER12
Description	Prohibits a news article from having more than 6 tags

SQL Code:

```
DROP FUNCTION IF EXISTS news_article_tags CASCADE;
DROP TRIGGER IF EXISTS news_article_tags ON tag_article CASCADE;

CREATE FUNCTION news_article_tags()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF ((6 < (SELECT COUNT(*) FROM tag_article
WHERE tag_article.article_id = NEW.article_id))) THEN
        RAISE EXCEPTION 'Cannot add more than 6 tags to a news article';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER news_article_tags
    BEFORE INSERT ON tag_article
    FOR EACH ROW
    EXECUTE PROCEDURE news_article_tags();
```

Trigger	TRIGGER13
Description	Prohibits a member from appealing more than once or if they are not blocked

SQL Code:

```
DROP FUNCTION IF EXISTS blocked_user_appeal CASCADE;
DROP TRIGGER IF EXISTS blocked_user_appeal ON appeal CASCADE;

CREATE FUNCTION blocked_user_appeal()
RETURNS TRIGGER AS
```

```
$BODY$
BEGIN
    IF ((SELECT COUNT(*) FROM appeal WHERE appeal.submitter_id =
NEW.submitter_id) > 0) THEN
        RAISE EXCEPTION 'Cannot appeal more than one time';
    END IF;

    IF ((SELECT is_blocked FROM member WHERE member.id =
NEW.submitter_id) = false) THEN
        RAISE EXCEPTION 'Cannot appeal if not blocked';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER blocked_user_appeal
    BEFORE INSERT ON appeal
    FOR EACH ROW
    EXECUTE PROCEDURE blocked_user_appeal();
```

Trigger	TRIGGER14
Description	Automatically creates a topic when a topic suggestion is accepted

SQL Code:

```
DROP FUNCTION IF EXISTS topic_control CASCADE;
DROP TRIGGER IF EXISTS topic_control ON topic_suggestion CASCADE;

CREATE FUNCTION topic_control()
RETURNS TRIGGER AS
$BODY$
BEGIN
    INSERT INTO TOPIC (name)
    VALUES (NEW.name);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER topic_control
    AFTER UPDATE OF status ON topic_suggestion
    FOR EACH ROW
    WHEN (OLD.status = 'Pending' AND NEW.status = 'Accepted')
    EXECUTE PROCEDURE topic_control();
```

Trigger	TRIGGER15
Description	Validates that a comment cannot precede the article it is commenting on

SQL Code:

```
DROP FUNCTION IF EXISTS comment_date_validation CASCADE;
DROP TRIGGER IF EXISTS comment_date_validation ON comment CASCADE;

CREATE FUNCTION comment_date_validation()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF ((SELECT date_time FROM content_item WHERE content_item.id = NEW.id) <
(SELECT date_time FROM content_item WHERE content_item.id = NEW.article_id))
THEN
        RAISE EXCEPTION 'Comment date cannot precede article date';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER comment_date_validation
    BEFORE INSERT ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE comment_date_validation();
```

Trigger	TRIGGER16
Description	Validates that a reply cannot precede the comment it is replying to

SQL Code:

```
DROP FUNCTION IF EXISTS comment_reply_validation CASCADE;
DROP TRIGGER IF EXISTS comment_reply_validation ON comment CASCADE;

CREATE FUNCTION comment_reply_validation()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (
        NEW.is_reply = TRUE
        AND
        (SELECT date_time FROM content_item WHERE content_item.id = NEW.reply_id)
        >
        (SELECT date_time FROM content_item WHERE content_item.id = NEW.id)
    ) THEN
        RAISE EXCEPTION 'Reply date cannot precede comment date';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER comment_reply_validation
    BEFORE INSERT ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE comment_reply_validation();
```

Trigger	TRIGGER17
Description	Prohibits a member from replying to a reply

SQL Code:

```
DROP FUNCTION IF EXISTS reply_to_comment_validation CASCADE;
DROP TRIGGER IF EXISTS reply_to_comment_validation ON comment CASCADE;

CREATE FUNCTION reply_to_comment_validation()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (
        NEW.is_reply = TRUE
        AND
        (SELECT is_reply FROM comment WHERE comment.id = NEW.reply_id) = TRUE
    ) THEN
        RAISE EXCEPTION 'Cannot reply to a reply';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER reply_to_comment_validation
    BEFORE INSERT ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE reply_to_comment_validation();
```

Trigger	TRIGGER18
Description	Prohibits a member from reporting themselves

SQL Code:

```sql
DROP FUNCTION IF EXISTS report_self CASCADE;
DROP TRIGGER IF EXISTS report_self ON member_report CASCADE;

CREATE FUNCTION report_self()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF ((SELECT submitter_id FROM report WHERE report.id = NEW.id) = 
NEW.member_id) THEN
        RAISE EXCEPTION 'A member is not allowed to report themselves';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER report_self
    BEFORE INSERT ON member_report
    FOR EACH ROW
    EXECUTE PROCEDURE report_self();
```

Trigger	TRIGGER19
Description	Prohibits a member from reporting their own content

SQL Code:

```sql
DROP FUNCTION IF EXISTS report_self_content CASCADE;
DROP TRIGGER IF EXISTS report_self_content ON content_report CASCADE;

CREATE FUNCTION report_self_content()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF ((SELECT submitter_id FROM report WHERE report.id = NEW.id) = 
(SELECT author_id FROM content_item WHERE content_item.id = NEW.content_item_id))
 THEN
        RAISE EXCEPTION 'A member is not allowed to report their own content';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER report_self_content
    BEFORE INSERT ON content_report
    FOR EACH ROW
    EXECUTE PROCEDURE report_self_content();
```

Trigger	TRIGGER20
Description	Generate a block notification when a member is blocked

SQL Code:

```
DROP FUNCTION IF EXISTS generate_block_notification CASCADE;
DROP TRIGGER IF EXISTS generate_block_notification ON member CASCADE;

CREATE FUNCTION generate_block_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN
    INSERT INTO notification (notified_id)
    VALUES (NEW.id)
    RETURNING id INTO notification_id;

    INSERT INTO block_notification (id)
    VALUES (notification_id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER generate_block_notification
    AFTER UPDATE OF is_blocked ON member
    FOR EACH ROW
    WHEN (OLD.is_blocked = false AND NEW.is_blocked = true)
    EXECUTE PROCEDURE generate_block_notification();
```

Trigger	TRIGGER21
Description	Generate a follow notification when a member follows another member

SQL Code:

```
DROP FUNCTION IF EXISTS generate_follow_notification CASCADE;
DROP TRIGGER IF EXISTS generate_follow_notification ON follow_member CASCADE;

CREATE FUNCTION generate_follow_notification()
```

```
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN
    INSERT INTO notification (notified_id)
    VALUES (NEW.followed_id)
    RETURNING id INTO notification_id;

    INSERT INTO follow_notification (id, follower_id)
    VALUES (notification_id, NEW.follower_id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER generate_follow_notification
    AFTER INSERT ON follow_member
    FOR EACH ROW
    EXECUTE PROCEDURE generate_follow_notification();
```

Trigger	TRIGGER22
Description	Generate a comment notification when a comment is added to a content item

SQL Code:

```
DROP FUNCTION IF EXISTS generate_comment_notification CASCADE;
DROP TRIGGER IF EXISTS generate_comment_notification ON comment CASCADE;

CREATE FUNCTION generate_comment_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN
    INSERT INTO notification (notified_id)
    VALUES ((SELECT author_id FROM content_item WHERE content_item.id = NEW.id))
    RETURNING id INTO notification_id;

    INSERT INTO comment_notification (id, comment_id)
    VALUES (notification_id, NEW.id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;
```

```
CREATE TRIGGER generate_comment_notification
    AFTER INSERT ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE generate_comment_notification();
```

Trigger	TRIGGER23
Description	Updates content item academy score when a vote is added, or removed, or edited

SQL Code:

```
DROP FUNCTION IF EXISTS update_content_item_academy_score CASCADE;
DROP TRIGGER IF EXISTS update_content_item_academy_score ON vote CASCADE;

CREATE FUNCTION update_content_item_academy_score()
RETURNS TRIGGER AS
$BODY$
DECLARE
    total_score INTEGER;
    vote_print vote;
    count INTEGER
BEGIN
    IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE') THEN
        SELECT SUM(weight) INTO total_score
        FROM vote
        WHERE vote.content_item_id = NEW.content_item_id;
    ELSIF (TG_OP = 'DELETE') THEN
        SELECT SUM(weight) INTO total_score
        FROM vote
        WHERE content_item_id = OLD.content_item_id;
    END IF;

    UPDATE content_item
    SET academy_score = COALESCE(total_score, 0)
    WHERE content_item.id =
COALESCE(NEW.content_item_id, OLD.content_item_id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER update_content_item_academy_score
    AFTER INSERT OR DELETE OR UPDATE ON vote
    FOR EACH ROW
    EXECUTE PROCEDURE update_content_item_academy_score();
```

Trigger	TRIGGER24
Description	Updates member academy score when one of his content items' votes changes

SQL Code:

```
DROP FUNCTION IF EXISTS update_member_academy_score CASCADE;
DROP TRIGGER IF EXISTS update_member_academy_score ON content_item CASCADE;

CREATE FUNCTION update_member_academy_score()
RETURNS TRIGGER AS
$BODY$
DECLARE
    total_score INT;
BEGIN
    SELECT SUM(academy_score) INTO total_score
    FROM content_item
    WHERE author_id = NEW.author_id AND is_deleted = FALSE;

    UPDATE member
    SET academy_score = total_score
    WHERE member.id = NEW.author_id;

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER update_member_academy_score
    AFTER UPDATE OF academy_score ON content_item
    FOR EACH ROW
    EXECUTE PROCEDURE update_member_academy_score();
```

Trigger	TRIGGER25
Description	Automatically creates a notification to the author of a content item when it is removed

SQL Code:

```
DROP TRIGGER IF EXISTS generate_removal_notification ON edit CASCADE;
DROP FUNCTION IF EXISTS generate_removal_notification();

CREATE FUNCTION generate_removal_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
```

```
    notification_id INT;
BEGIN

    IF (NEW.author_id = (SELECT author_id FROM content_item
WHERE content_item.id = NEW.content_item_id)) THEN
        RETURN NULL;
    END IF;

    INSERT INTO notification (notified_id)
    VALUES ((SELECT author_id FROM content_item WHERE content_item.id =
NEW.content_item_id))
    RETURNING id INTO notification_id;

    INSERT INTO content_notification (id, content_item_id)
    VALUES (notification_id, NEW.content_item_id);

    INSERT INTO removal_notification (id)
    VALUES (notification_id);

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER generate_removal_notification
    AFTER INSERT ON edit
    FOR EACH ROW
    WHEN (NEW.altered_field = 'is_deleted' AND NEW.old_value = 'FALSE'
AND NEW.new_value = 'TRUE')
    EXECUTE PROCEDURE generate_removal_notification();
CREATE FUNCTION generate_removal_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN

    IF (NEW.author_id = (SELECT author_id FROM content_item
WHERE content_item.id = NEW.content_item_id)) THEN
        RETURN NULL;
    END IF;

    INSERT INTO notification (notified_id)
    VALUES ((SELECT author_id FROM content_item WHERE
content_item.id = NEW.content_item_id))
    RETURNING id INTO notification_id;

    INSERT INTO content_notification (id, content_item_id)
    VALUES (notification_id, NEW.content_item_id);

    INSERT INTO removal_notification (id)
    VALUES (notification_id);
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER generate_removal_notification
    AFTER INSERT ON edit
    FOR EACH ROW
    WHEN (NEW.altered_field = 'is_deleted' AND NEW.old_value = 'FALSE'
AND NEW.new_value = 'TRUE')
    EXECUTE PROCEDURE generate_removal_notification();
```

Trigger	TRIGGER26
Description	Enforces lowercase use on all emails

SQL Code:

```
CREATE FUNCTION email_lowercase()
RETURNS TRIGGER AS
$BODY$
BEGIN
    NEW.email = LOWER(NEW.email);
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

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

Trigger	TRIGGER27
Description	Enforces lowercase use on all tags

SQL Code:

```
CREATE FUNCTION tag_lowercase()
RETURNS TRIGGER AS
$BODY$
BEGIN
    NEW.name = LOWER(NEW.name);
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;
```

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

Trigger	TRIGGER28
Description	Enhances the search functionality for content items by updating tsvector columns based on their content

SQL Code:

```
DROP FUNCTION IF EXISTS content_item_search_update() CASCADE;
DROP TRIGGER IF EXISTS content_item_search_update ON content_item CASCADE;

ALTER TABLE content_item
ADD COLUMN tsvectors tsvector;

CREATE FUNCTION content_item_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        IF (NEW.id IN (SELECT id FROM article)) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', (SELECT title FROM article
WHERE article.id = NEW.id)), 'A') ||
                setweight(to_tsvector('english', NEW.body), 'B')
            );
        ELSE
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.body), 'B')
            );
        END IF;
 END IF;

 IF TG_OP = 'UPDATE' THEN
        IF (NEW.id IN (SELECT id FROM article)) THEN
            IF (NEW.body <> OLD.body) THEN
                NEW.tsvectors = (
                    setweight(to_tsvector('english', (SELECT title FROM article
WHERE article.id = NEW.id)), 'A') ||
                    setweight(to_tsvector('english', NEW.body), 'B')
                );
            END IF;
        ELSE
            IF (NEW.body <> OLD.body) THEN
                NEW.tsvectors = (
                    setweight(to_tsvector('english', NEW.body), 'B')
                );
            END IF;
        END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

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

Trigger	TRIGGER29
Description	Enhances the search functionality for articles, updating related content_item tsvectors based on article changes

SQL Code:

```
DROP FUNCTION IF EXISTS article_search_update() CASCADE;
DROP TRIGGER IF EXISTS article_search_update ON article CASCADE;

CREATE FUNCTION article_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        IF (NEW.id IN (SELECT id FROM content_item)) THEN
            UPDATE content_item
            SET tsvectors = (
                setweight(to_tsvector('english', (SELECT title FROM article
WHERE article.id = NEW.id)), 'A') ||
                setweight(to_tsvector('english', (SELECT body FROM content_item
WHERE content_item.id = NEW.id)), 'B')
            )
            WHERE content_item.id = NEW.id;
        END IF;
 END IF;

 IF TG_OP = 'UPDATE' THEN
        IF (NEW.id IN (SELECT id FROM content_item)) THEN
            IF (NEW.title <> OLD.title) THEN
                UPDATE content_item
                SET tsvectors = (
                    setweight(to_tsvector('english', (SELECT title FROM article
WHERE article.id = NEW.id)), 'A') ||
                    setweight(to_tsvector('english', (SELECT body
FROM content_item WHERE content_item.id = NEW.id)), 'B')
                )
                WHERE content_item.id = NEW.id;
            END IF;
        END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

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

Trigger	TRIGGER30
Description	Enhances the search functionality for members, updating tsvectors to include username, first name, and last name

SQL Code:

```
DROP FUNCTION IF EXISTS member_search_update() CASCADE;
DROP TRIGGER IF EXISTS member_search_update ON member CASCADE;

ALTER TABLE member
ADD COLUMN tsvectors tsvector;

CREATE FUNCTION member_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.username), 'A') ||
            setweight(to_tsvector('english', NEW.first_name), 'B') ||
            setweight(to_tsvector('english', NEW.last_name), 'B')
        );
 END IF;

 IF TG_OP = 'UPDATE' THEN
        IF (NEW.username <> OLD.username) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.username), 'A') ||
                setweight(to_tsvector('english', NEW.first_name), 'B') ||
                setweight(to_tsvector('english', NEW.last_name), 'B')
            );
        END IF;
 END IF;
 RETURN NEW;
END $$

LANGUAGE plpgsql;

CREATE TRIGGER member_search_update
BEFORE INSERT OR UPDATE ON member
```

```
FOR EACH ROW
EXECUTE PROCEDURE member_search_update();
```

4. Transactions

TR01	Create Article
Justification	To ensure consistent data integrity for article creation, this transaction uses the Repeatable Read isolation level. This choice guarantees that once an article is being created, no concurrent transactions can modify or read inconsistent data from the content_item and article tables. By doing so, it protects against potential disruptions that might occur due to concurrent transactions inserting data in these tables, ensuring the integrity of newly created articles. A rollback is initiated in case of any errors, preserving data consistency.
Isolation level	Repeatable Read

SQL Code:

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO content_item (body, author_id)
        VALUES ($body, $author_id);

INSERT INTO article (id, title, topic_id, imdb_info_id)
        VALUES (currval('content_item_id_seq'), $title,
$topic_id, $imdb_info_id);

DO
$$
DECLARE
    tag INT;
    tags_array INT[] := ARRAY $tags;
BEGIN
    FOREACH tag IN ARRAY tags_array
    LOOP
        INSERT INTO tag_article(article_id, tag_id)
        VALUES (currval('content_item_id_seq'), tag);
    END LOOP;
END
$$
LANGUAGE plpgsql;
```

```
COMMIT;
```

TR02	Create Comment
Justification	For creating comments, maintaining data consistency is critical. Hence, this transaction adopts the Repeatable Read isolation level. It ensures that, during comment creation, no concurrent transactions can disrupt the insertion of comments into the content_item and comment tables. This choice safeguards against inconsistencies that might arise if other transactions modify the data concurrently. In case of errors, a rollback is initiated, preventing the storage of inconsistent comments.
Isolation level	Repeatable Read

SQL Code:

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO content_item (body, author_id)
        VALUES ($body, $author_id);

INSERT INTO comment (id, is_reply, article_id, reply_id)
        VALUES (currval('content_item_id_seq'), $is_reply,
$article_id, $reply_id);

COMMIT;
```

TR03	Create Upvote notification
Justification	This transaction, responsible for generating upvote notifications, relies on the Repeatable Read isolation level to guarantee consistent data integrity. By using this isolation level, it prevents concurrent transactions from potentially interfering with updates in the notification and content_notification tables, which might be caused by inserts in the upvote_notification table. The chosen isolation level ensures that upvote notifications are stored without inconsistencies. In case of errors, a rollback is performed, maintaining data integrity.
Isolation level	Repeatable Read

SQL Code:

```sql
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO notification (notified_id)
        VALUES ($notified_id);

INSERT INTO content_notification (id, content_item_id)
        VALUES (currval('notification_id_seq'), $content_item_id);

INSERT INTO upvote_notification (id, amount)
        VALUES (currval('notification_id_seq'), $amount);

COMMIT;
```

TR04	Create Content Report
Justification	The creation of content reports is crucial to maintaining data integrity. This transaction utilizes the Repeatable Read isolation level to prevent concurrent transactions from interfering with updates in the report and content_report tables, which might be triggered by inserts in the content_report table. This approach safeguards against storing inconsistent content reports. In case of errors, a rollback is initiated to ensure data consistency.
Isolation level	Repeatable Read

SQL Code:

```sql
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO report (body, submitter_id)
                VALUES ($body, $submitter_id);

INSERT INTO content_report (id, motive, content_item_id)
        VALUES (currval('report_id_seq'), $motive, $content_item_id);

COMMIT;
```

TR05	Create Member Report
Justification	Creating member reports requires a consistent and reliable approach. This transaction employs the Repeatable Read isolation level to avoid potential disruptions caused by concurrent transactions that might lead to updates in the report and member_report tables due to inserts in the member_report table. The chosen isolation level protects against storing inconsistent member reports. In case of errors, a rollback is performed, ensuring data integrity.
Isolation level	Repeatable Read

SQL Code:

```sql
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO report (body, submitter_id)
        VALUES ($body, $submitter_id);

INSERT INTO member_report (id, misconduct, member_id)
        VALUES (currval('report_id_seq'), $misconduct, $member_id);

COMMIT;
```

TR06	View News Item Comments
Justification	This read-only transaction, responsible for viewing news item comments, utilizes the Repeatable Read isolation level to guarantee data consistency. By preventing concurrent transactions from modifying data, it ensures that the data retrieved by two SELECT statements remains consistent. This is important because new comments may be added to the comment table during the transaction, potentially leading to "Phantom Reads." While the transaction is read-only, it still requires the chosen isolation level to prevent data inconsistencies.
Isolation level	Repeatable Read READ ONLY

SQL Code:

```sql
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ ONLY;

SELECT COUNT(*) FROM comment WHERE article_id = $article_id;

SELECT comment.*, member.first_name, member.last_name,
profile_image.file_name, content_item.* FROM comment
    INNER JOIN content_item ON comment.id = content_item.id
    INNER JOIN member ON content_item.author_id = member.id
    INNER JOIN profile_image ON member.profile_image_id = profile_image.id
    WHERE article_id = $article_id AND content_item.is_deleted = false
    ORDER BY content_item.date_time DESC;

COMMIT;
```

TR07	View News Information
Justification	Ensuring data consistency while viewing news information is crucial. This transaction employs the Repeatable Read isolation level to prevent other transactions from modifying the data being read. By doing so, it guarantees that the data retrieved by the SELECT statements remains consistent, even though it's a read-only operation. This is essential for maintaining accurate and up-to-date news information.
Isolation level	Repeatable Read READ ONLY

SQL Code:

```sql
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ ONLY;

SELECT article.id, article.title, content_item.body, topic.name,
member.first_name, member.last_name, content_item.date_time,
content_item.academy_score, imdb_info.query_type, imdb_info.imdb_id
FROM article
    INNER JOIN topic ON article.topic_id = topic.id
    INNER JOIN imdb_info ON article.imdb_info_id = imdb_info.id
    INNER JOIN content_item ON article.id = content_item.id
    INNER JOIN member ON content_item.author_id = member.id
    WHERE article.id = $article_id AND content_item.is_deleted = false;

SELECT article_image.id, article_image.file_name FROM article_image
    WHERE article_image.article_id = $article_id;
```

```
SELECT tag.id, tag.name FROM tag
    INNER JOIN tag_article ON tag.id = tag_article.tag_id
    WHERE tag_article.article_id = $article_id;

COMMIT;
```

TR08	View related news
Justification	Viewing related news requires consistency in the data retrieved. Therefore, this transaction uses the Repeatable Read isolation level to ensure that the data retrieved by the SELECT statements is consistent. This prevents other transactions from modifying the data that is being read. The chosen isolation level is essential to maintain data integrity, even though the transaction is read-only.
Isolation level	Repeatable Read READ ONLY

SQL Code:

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ ONLY;

SELECT article.id, article.title, content_item.date_time,
content_item.academy_score FROM article
    INNER JOIN content_item ON article.id = content_item.id
    INNER JOIN topic ON article.topic_id = topic.id
    WHERE article.topic_id = $topic_id AND article.id != $article_id
AND content_item.is_deleted = false
    ORDER BY content_item.date_time DESC;

SELECT article.id, article.title, content_item.date_time,
content_item.academy_score FROM article
    INNER JOIN content_item ON article.id = content_item.id
    INNER JOIN topic ON article.topic_id = topic.id
    INNER JOIN tag_article ON article.id = tag_article.article_id
    INNER JOIN tag ON tag_article.tag_id = tag.id
    WHERE tag.id = ANY($tags) AND article.id != $article_id
AND content_item.is_deleted = false
    ORDER BY content_item.date_time DESC;

COMMIT;
```

TR09	View All Notifications
Justification	When viewing all notifications, data consistency is critical. To ensure that the data retrieved by the SELECT statements remains

TR09	View All Notifications
	consistent, this transaction uses the Repeatable Read isolation level, even though it is read-only. By employing this isolation level, it prevents other transactions from modifying the data being read. This is essential to maintain the accuracy and reliability of the displayed notifications.
Isolation level	Repeatable Read READ ONLY

SQL Code:

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ ONLY;

SELECT COUNT(*) FROM notification WHERE notified_id = $member_id
AND was_read = false;

SELECT notification.id, notification.date_time, comment_notification.comment_id
FROM notification
    INNER JOIN comment_notification ON notification.id = comment_notification.id
    WHERE notification.notified_id = $member_id
AND notification.was_read = false;

SELECT notification.id, notification.date_time,
content_notification.content_item_id, upvote_notification.amount
FROM notification
    INNER JOIN content_notification ON notification.id =
content_notification.id
    INNER JOIN upvote_notification ON notification.id =
upvote_notification.id
    WHERE notification.notified_id = $member_id
AND notification.was_read = false;

SELECT notification.id, notification.date_time,
content_notification.content_item_id
FROM notification
    INNER JOIN content_notification ON notification.id =
content_notification.id
    INNER JOIN edit_notification ON notification.id =
edit_notification.id
    WHERE notification.notified_id = $member_id
AND notification.was_read = false;

SELECT notification.id, notification.date_time,
content_notification.content_item_id
FROM notification
    INNER JOIN content_notification ON notification.id =
content_notification.id
    INNER JOIN removal_notification ON notification.id =
removal_notification.id
    WHERE notification.notified_id = $member_id AND
notification.was_read = false;

SELECT notification.id, notification.date_time,
content_notification.content_item_id
FROM notification
    INNER JOIN content_notification ON notification.id =
content_notification.id
    INNER JOIN undefined_topic_notification ON notification.id =
undefined_topic_notification.id
    WHERE notification.notified_id = $member_id
AND notification.was_read = false;

SELECT notification.id, notification.date_time FROM notification
    INNER JOIN block_notification ON notification.id =
block_notification.id
    WHERE notification.notified_id = $member_id
AND notification.was_read = false;

SELECT notification.id, notification.date_time,
follow_notification.follower_id
FROM notification
    INNER JOIN follow_notification ON notification.id =
follow_notification.id
    WHERE notification.notified_id = $member_id
AND notification.was_read = false;

COMMIT;
```

TR10	View Reports
Justification	Ensuring data consistency while viewing reports is of utmost importance. This transaction uses the Repeatable Read isolation level to guarantee that the data retrieved by the SELECT statements remains consistent. By employing this isolation level, it prevents other transactions from modifying the data being read, ensuring that reports are displayed accurately. Despite being a read-only operation, data integrity must be maintained.
Isolation level	Repeatable Read READ ONLY

SQL Code:

```sql
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ ONLY;

SELECT COUNT(*) FROM content_report;

SELECT content_report.id, content_report.motive,
content_report.content_item_id,
report.date_time, report.body, member.id, member.username, member.first_name,
member.last_name FROM content_report
    INNER JOIN report ON content_report.id = report.id
    INNER JOIN member ON report.submitter_id = member.id;

SELECT COUNT(*) FROM member_report;

SELECT member_report.id, member_report.misconduct, member_report.member_id,
report.date_time, report.body, member.id, member.username, member.first_name,
member.last_name FROM member_report
    INNER JOIN report ON member_report.id = report.id
    INNER JOIN member ON report.submitter_id = member.id;

COMMIT;
```

Annex A. SQL Code

In this section, we provide the essential SQL scripts for creating and populating the database used in the EBD (E-commerce Business Database) component of our project. The SQL scripts are presented as separate elements to facilitate database management. The creation script encompasses the code required to construct and reconstruct the database schema. The population script includes a substantial number of test data tuples with realistic values for each field within the database.

Both scripts can be found in our group's Git repository, which is accessible via the following links:

- create.sql
- populate.sql

These scripts serve as a foundational resource for database management and testing throughout the development and deployment phases of our project.

A.1. Database schema

```sql
DROP SCHEMA IF EXISTS lbaw2356 CASCADE;
CREATE SCHEMA lbaw2356;
SET search_path TO lbaw2356;
```

```sql
-----------------------------------------
-- Drop Tables and Types
-----------------------------------------

DROP TABLE IF EXISTS follow_notification CASCADE;
DROP TABLE IF EXISTS block_notification CASCADE;
DROP TABLE IF EXISTS undefined_topic_notification CASCADE;
DROP TABLE IF EXISTS removal_notification CASCADE;
DROP TABLE IF EXISTS edit_notification CASCADE;
DROP TABLE IF EXISTS upvote_notification CASCADE;
DROP TABLE IF EXISTS content_notification CASCADE;
DROP TABLE IF EXISTS comment_notification CASCADE;
DROP TABLE IF EXISTS notification CASCADE;
DROP TABLE IF EXISTS member_report CASCADE;
DROP TABLE IF EXISTS content_report CASCADE;
DROP TABLE IF EXISTS report CASCADE;
DROP TABLE IF EXISTS vote CASCADE;
DROP TABLE IF EXISTS edit CASCADE;
DROP TABLE IF EXISTS comment CASCADE;
DROP TABLE IF EXISTS follow_tag CASCADE;
DROP TABLE IF EXISTS tag_article CASCADE;
DROP TABLE IF EXISTS tag CASCADE;
DROP TABLE IF EXISTS article CASCADE;
DROP TABLE IF EXISTS content_item CASCADE;
DROP TABLE IF EXISTS topic_suggestion CASCADE;
DROP TABLE IF EXISTS topic CASCADE;
DROP TABLE IF EXISTS follow_member CASCADE;
DROP TABLE IF EXISTS appeal CASCADE;
DROP TABLE IF EXISTS member CASCADE;
DROP TABLE IF EXISTS profile_image CASCADE;
DROP TABLE IF EXISTS imdb_info CASCADE;

DROP TYPE IF EXISTS motives;
DROP TYPE IF EXISTS misconduct_types;
DROP TYPE IF EXISTS statuses;



-----------------------------------------
-- Drop Triggers and Functions
-----------------------------------------

DROP TRIGGER IF EXISTS data_anonymization ON member CASCADE;
DROP FUNCTION IF EXISTS data_anonymization();
DROP TRIGGER IF EXISTS delete_content_item ON content_item CASCADE;
DROP FUNCTION IF EXISTS delete_content_item();
DROP TRIGGER IF EXISTS before_permanent_delete_article ON article CASCADE;
DROP FUNCTION IF EXISTS before_permanent_delete_article();
DROP TRIGGER IF EXISTS after_permanent_delete_article ON article CASCADE;
DROP FUNCTION IF EXISTS after_permanent_delete_article();
DROP TRIGGER IF EXISTS before_delete_comment ON comment CASCADE;
DROP FUNCTION IF EXISTS before_delete_comment();
DROP TRIGGER IF EXISTS after_delete_comment ON comment CASCADE;
DROP FUNCTION IF EXISTS after_delete_comment();
DROP TRIGGER IF EXISTS ban_tag ON tag CASCADE;
DROP FUNCTION IF EXISTS ban_tag();
DROP TRIGGER IF EXISTS edit_content_notification ON edit CASCADE;
DROP FUNCTION IF EXISTS edit_content_notification();
DROP TRIGGER IF EXISTS remove_topic ON topic CASCADE;
DROP FUNCTION IF EXISTS remove_topic();
DROP TRIGGER IF EXISTS notify_undefined_topic ON article CASCADE;
DROP FUNCTION IF EXISTS notify_undefined_topic();
DROP FUNCTION IF EXISTS generate_undefined_topic_notification(author_id INT,
article_id INT);
DROP TRIGGER IF EXISTS vote_for_own_content ON vote CASCADE;
DROP FUNCTION IF EXISTS vote_for_own_content();
DROP TRIGGER IF EXISTS news_article_tags ON tag_article CASCADE;
DROP FUNCTION IF EXISTS news_article_tags();
DROP TRIGGER IF EXISTS blocked_user_appeal ON appeal CASCADE;
DROP FUNCTION IF EXISTS blocked_user_appeal();
DROP TRIGGER IF EXISTS topic_control ON topic_suggestion CASCADE;
DROP FUNCTION IF EXISTS topic_control();
DROP TRIGGER IF EXISTS comment_date_validation ON comment CASCADE;
DROP FUNCTION IF EXISTS comment_date_validation();
DROP TRIGGER IF EXISTS comment_reply_validation ON comment CASCADE;
DROP FUNCTION IF EXISTS comment_reply_validation();
DROP TRIGGER IF EXISTS reply_to_comment_validation ON comment CASCADE;
DROP FUNCTION IF EXISTS reply_to_comment_validation();
DROP TRIGGER IF EXISTS report_self ON member_report CASCADE;
DROP FUNCTION IF EXISTS report_self();
DROP TRIGGER IF EXISTS report_self_content ON content_report CASCADE;
DROP FUNCTION IF EXISTS report_self_content();
DROP TRIGGER IF EXISTS generate_block_notification ON member CASCADE;
DROP FUNCTION IF EXISTS generate_block_notification();
DROP TRIGGER IF EXISTS generate_follow_notification ON follow_member CASCADE;
DROP FUNCTION IF EXISTS generate_follow_notification();
DROP TRIGGER IF EXISTS generate_comment_notification ON comment CASCADE;
DROP FUNCTION IF EXISTS generate_comment_notification();
DROP TRIGGER IF EXISTS update_content_item_academy_score ON vote CASCADE;
DROP FUNCTION IF EXISTS update_content_item_academy_score();
DROP TRIGGER IF EXISTS update_member_academy_score ON content_item CASCADE;
DROP FUNCTION IF EXISTS update_member_academy_score();
DROP TRIGGER IF EXISTS generate_removal_notification ON edit CASCADE;
DROP FUNCTION IF EXISTS generate_removal_notification();
DROP FUNCTION IF EXISTS content_item_search_update() CASCADE;
DROP TRIGGER IF EXISTS content_item_search_update ON content_item CASCADE;
DROP FUNCTION IF EXISTS article_search_update() CASCADE;
DROP TRIGGER IF EXISTS article_search_update ON article CASCADE;
DROP FUNCTION IF EXISTS member_search_update() CASCADE;
DROP TRIGGER IF EXISTS member_search_update ON member CASCADE;
DROP FUNCTION IF EXISTS email_lowercase() CASCADE;
DROP TRIGGER IF EXISTS email_lowercase ON member CASCADE;
DROP FUNCTION IF EXISTS tag_lowercase() CASCADE;
DROP TRIGGER IF EXISTS tag_lowercase ON tag CASCADE;

-----------------------------------------
-- Drop Indexes
```

```sql
-----------------------------------------

DROP INDEX IF EXISTS content_item_date;
DROP INDEX IF EXISTS content_item_academy_score;
DROP INDEX IF EXISTS article_topic;
DROP INDEX IF EXISTS content_item_search;
DROP INDEX IF EXISTS member_search;


-----------------------------------------
-- Types
-----------------------------------------

CREATE TYPE motives AS ENUM ('Hateful', 'Spam', 'Violent', 'NSFW',
'Misinformation', 'Plagiarism', 'Other');
CREATE TYPE misconduct_types AS ENUM ('Hateful', 'Harassment', 'Spam',
'Impersonation', 'InnapropriateContent', 'Other');
CREATE TYPE statuses AS ENUM ('Pending', 'Accepted', 'Rejected');


-----------------------------------------
-- Tables
-----------------------------------------

CREATE TABLE imdb_info (
    id               SERIAL,
    query_type       VARCHAR(255)    NOT NULL,
    imdb_info_id          VARCHAR(255)    UNIQUE NOT NULL,
    CONSTRAINT imdb_info_pk PRIMARY KEY (id),
    CONSTRAINT imdb_info_imdb_info_id_unique UNIQUE (imdb_info_id)
);

CREATE TABLE profile_image (
    id               SERIAL,
    file_name        VARCHAR(255)    UNIQUE NOT NULL,
    CONSTRAINT profile_image_pk PRIMARY KEY (id),
    CONSTRAINT profile_image_file_name_unique UNIQUE (file_name)
);

CREATE TABLE member (
    id               SERIAL,
    email            VARCHAR(255)    UNIQUE NOT NULL,
    password         VARCHAR(255)    NOT NULL,
    username         VARCHAR(255)    UNIQUE NOT NULL,
    first_name       VARCHAR(255)    NOT NULL,
    last_name        VARCHAR(255)    NOT NULL,
    biography        TEXT,
    is_blocked       BOOLEAN         NOT NULL DEFAULT FALSE,
    is_admin         BOOLEAN         NOT NULL DEFAULT FALSE,
    is_deleted       BOOLEAN         NOT NULL DEFAULT FALSE,
    academy_score    INTEGER         NOT NULL DEFAULT 0,
    profile_image_id INTEGER        NOT NULL DEFAULT 0,
    CONSTRAINT member_pk PRIMARY KEY (id),
    CONSTRAINT member_email_unique UNIQUE (email),
    CONSTRAINT member_username_unique UNIQUE (username),
    CONSTRAINT member_profile_image_fk FOREIGN KEY (profile_image_id)
REFERENCES profile_image(id)
);

CREATE TABLE appeal (
    id                SERIAL,
    date_time         TIMESTAMP       NOT NULL DEFAULT CURRENT_TIMESTAMP,
    body              TEXT,
    submitter_id      INTEGER         NOT NULL,
    CONSTRAINT appeal_pk PRIMARY KEY (id),
    CONSTRAINT appeal_submitter_fk FOREIGN KEY (submitter_id)
REFERENCES member(id)
);

CREATE TABLE follow_member (
    follower_id       INTEGER         NOT NULL,
    followed_id       INTEGER         NOT NULL,
    CONSTRAINT follow_pk PRIMARY KEY (follower_id, followed_id),
    CONSTRAINT follow_follower_fk FOREIGN KEY (follower_id)
REFERENCES member(id)
ON DELETE CASCADE,
    CONSTRAINT follow_followed_fk FOREIGN KEY (followed_id)
REFERENCES member(id)
ON DELETE CASCADE,
    CONSTRAINT follow_check CHECK (follower_id != followed_id)
);

CREATE TABLE topic (
    id                SERIAL,
    name              VARCHAR(255)    UNIQUE NOT NULL,
    CONSTRAINT topic_pk PRIMARY KEY (id),
    CONSTRAINT topic_name_unique UNIQUE (name)
);

CREATE TABLE topic_suggestion (
    id                SERIAL,
    name              VARCHAR(255)    UNIQUE NOT NULL,
    status            statuses        NOT NULL DEFAULT 'Pending',
    suggester_id       INTEGER         NOT NULL,
    CONSTRAINT topic_suggestion_pk PRIMARY KEY (id),
    CONSTRAINT topic_suggestion_name_unique UNIQUE (name),
    CONSTRAINT topic_suggester_fk FOREIGN KEY (suggester_id)
REFERENCES member(id)
);

CREATE TABLE content_item (
    id                SERIAL,
    date_time         TIMESTAMP       NOT NULL DEFAULT CURRENT_TIMESTAMP,
    body              TEXT            NOT NULL,
    is_deleted        BOOLEAN         NOT NULL DEFAULT FALSE,
    academy_score     INTEGER         NOT NULL DEFAULT 0,
    author_id       INTEGER         NOT NULL,
    CONSTRAINT content_item_pk PRIMARY KEY (id),
    CONSTRAINT content_item_author_fk FOREIGN KEY (author_id)
REFERENCES member(id)
);

CREATE TABLE article (
    id              INTEGER,
    title           VARCHAR(255)    NOT NULL,
    topic_id        INTEGER         NOT NULL,
    imdb_info_id    INTEGER,
    CONSTRAINT article_pk PRIMARY KEY (id),
    CONSTRAINT article_content_item_fk FOREIGN KEY (id)
REFERENCES content_item(id)
ON DELETE CASCADE,
    CONSTRAINT article_topic_fk FOREIGN KEY (topic_id) REFERENCES topic(id),
    CONSTRAINT article_imdb_info_fk FOREIGN KEY (imdb_info_id)
REFERENCES imdb_info(id)
);

CREATE TABLE article_image (
    id              SERIAL,
    file_name       VARCHAR(255)    UNIQUE NOT NULL,
    article_id      INTEGER         NOT NULL,
    CONSTRAINT article_image_pk PRIMARY KEY (id),
    CONSTRAINT article_image_file_name_unique UNIQUE (file_name),
    CONSTRAINT article_image_article_fk FOREIGN KEY (article_id)
REFERENCES article(id)
);

CREATE TABLE tag (
    id              SERIAL,
    name            VARCHAR(255)    UNIQUE NOT NULL,
    is_banned       BOOLEAN         NOT NULL DEFAULT FALSE,
    CONSTRAINT tag_pk PRIMARY KEY (id),
    CONSTRAINT tag_name_unique UNIQUE (name)
);

CREATE TABLE tag_article (
    tag_id          INTEGER         NOT NULL,
    article_id      INTEGER         NOT NULL,
    CONSTRAINT tag_article_pk PRIMARY KEY (tag_id, article_id),
    CONSTRAINT tag_article_tag_fk FOREIGN KEY (tag_id) REFERENCES tag(id)
ON DELETE CASCADE,
    CONSTRAINT tag_article_article_fk FOREIGN KEY (article_id)
REFERENCES article(id)
ON DELETE CASCADE
);

CREATE TABLE follow_tag (
    tag_id     INTEGER         NOT NULL,
    member_id     INTEGER         NOT NULL,
    CONSTRAINT follow_tag_pk PRIMARY KEY (tag_id, member_id),
    CONSTRAINT follow_tag_tag_fk FOREIGN KEY (tag_id) REFERENCES tag(id)
ON DELETE CASCADE,
    CONSTRAINT follow_tag_member_fk FOREIGN KEY (member_id)
REFERENCES member(id) ON DELETE CASCADE
);

CREATE TABLE comment (
    id                INTEGER,
    is_reply          BOOLEAN         NOT NULL,
    article_id        INTEGER         NOT NULL,
    reply_id          INTEGER,
    CONSTRAINT comment_pk PRIMARY KEY (id),
    CONSTRAINT comment_content_item_fk FOREIGN KEY (id)
REFERENCES content_item(id) ON DELETE CASCADE,
    CONSTRAINT comment_article_fk FOREIGN KEY (article_id)
REFERENCES article(id),
    CONSTRAINT comment_reply_fk FOREIGN KEY (reply_id)
REFERENCES comment(id),
    CONSTRAINT comment_reply_check CHECK (is_reply = TRUE
AND reply_id IS NOT NULL
OR is_reply = FALSE AND reply_id IS NULL)
);

CREATE TABLE edit (
    id                SERIAL,
    altered_field     VARCHAR(255)    NOT NULL,
    old_value         TEXT            NOT NULL,
    new_value         TEXT            NOT NULL,
    content_item_id INTEGER           NOT NULL,
    author_id         INTEGER         NOT NULL,
    CONSTRAINT edit_pk PRIMARY KEY (id),
    CONSTRAINT edit_content_item_fk FOREIGN KEY (content_item_id) REFERENCES
content_item(id) ON DELETE CASCADE,
    CONSTRAINT edit_author_fk FOREIGN KEY (author_id) REFERENCES member(id)
);

CREATE TABLE vote (
    member_id         INTEGER         NOT NULL,
    content_item_id INTEGER           NOT NULL,
    weight            INTEGER         NOT NULL
CHECK (weight = 1 OR weight = -1),
    CONSTRAINT vote_pk PRIMARY KEY (member_id, content_item_id),
    CONSTRAINT vote_member_fk FOREIGN KEY (member_id) REFERENCES member(id),
    CONSTRAINT vote_content_item_fk FOREIGN KEY (content_item_id) REFERENCES
content_item(id)
);

CREATE TABLE report (
    id                SERIAL,
    date_time         TIMESTAMP       NOT NULL DEFAULT CURRENT_TIMESTAMP,
    body              TEXT,
    submitter_id      INTEGER         NOT NULL,
    CONSTRAINT report_pk PRIMARY KEY (id),
```

```sql
    CONSTRAINT report_submitter_fk FOREIGN KEY (submitter_id)
REFERENCES member(id)
);

CREATE TABLE content_report (
    id              INTEGER,
    motive          motives         NOT NULL,
    content_item_id INTEGER         NOT NULL,
    CONSTRAINT content_report_pk PRIMARY KEY (id),
    CONSTRAINT content_report_report_fk FOREIGN KEY (id)
REFERENCES report(id)
ON DELETE CASCADE,
    CONSTRAINT content_report_content_item_fk
FOREIGN KEY (content_item_id) REFERENCES content_item(id)
);

CREATE TABLE member_report (
    id              INTEGER,
    misconduct      misconduct_types NOT NULL,
    member_id       INTEGER         NOT NULL,
    CONSTRAINT member_report_pk PRIMARY KEY (id),
    CONSTRAINT member_report_report_fk FOREIGN KEY (id)
REFERENCES report(id) ON DELETE CASCADE,
    CONSTRAINT member_report_member_fk FOREIGN KEY (member_id)
REFERENCES member(id)
);

CREATE TABLE notification (
    id              SERIAL,
    was_read        BOOLEAN         NOT NULL DEFAULT FALSE,
    date_time       TIMESTAMP       NOT NULL DEFAULT CURRENT_TIMESTAMP,
    notified_id     INTEGER         NOT NULL,
    CONSTRAINT notification_pk PRIMARY KEY (id),
    CONSTRAINT notification_notified_fk FOREIGN KEY (notified_id)
REFERENCES member(id) ON DELETE CASCADE
);

CREATE TABLE comment_notification (
    id              INTEGER,
    comment_id      INTEGER         NOT NULL,
    CONSTRAINT comment_notification_pk PRIMARY KEY (id),
    CONSTRAINT comment_notification_notification_fk FOREIGN KEY (id)
REFERENCES notification(id) ON DELETE CASCADE,
    CONSTRAINT comment_notification_comment_fk FOREIGN KEY (comment_id)
REFERENCES comment(id) ON DELETE CASCADE
);

CREATE TABLE content_notification (
    id              INTEGER,
    content_item_id INTEGER         NOT NULL,
    CONSTRAINT content_notification_pk PRIMARY KEY (id),
    CONSTRAINT content_notification_notification_fk FOREIGN KEY (id)
REFERENCES notification(id) ON DELETE CASCADE,
    CONSTRAINT content_notification_content_item_fk
FOREIGN KEY (content_item_id)
REFERENCES content_item(id) ON DELETE CASCADE
);

CREATE TABLE upvote_notification (
    id               INTEGER,
    amount           INTEGER         NOT NULL,
    CONSTRAINT upvote_notification_pk PRIMARY KEY (id),
    CONSTRAINT upvote_notification_content_notification_fk FOREIGN KEY (id)
REFERENCES content_notification(id) ON DELETE CASCADE,
    CONSTRAINT upvote_notification_amount_check CHECK (amount > 0)
);

CREATE TABLE edit_notification (
    id               INTEGER,
    CONSTRAINT edit_notification_pk PRIMARY KEY (id),
    CONSTRAINT edit_notification_content_notification_fk FOREIGN KEY (id)
REFERENCES content_notification(id) ON DELETE CASCADE
);

CREATE TABLE removal_notification (
    id               INTEGER,
    CONSTRAINT removal_notification_pk PRIMARY KEY (id),
    CONSTRAINT removal_notification_content_notification_fk FOREIGN KEY (id)
REFERENCES content_notification(id) ON DELETE CASCADE
);

CREATE TABLE undefined_topic_notification (
    id               INTEGER,
    CONSTRAINT undefined_topic_notification_pk PRIMARY KEY (id),
    CONSTRAINT undefined_topic_notification_content_notification_fk
FOREIGN KEY (id)
REFERENCES content_notification(id) ON DELETE CASCADE
);

CREATE TABLE block_notification (
    id               INTEGER,
    CONSTRAINT block_notification_pk PRIMARY KEY (id),
    CONSTRAINT block_notification_notification_fk FOREIGN KEY (id)
REFERENCES notification(id) ON DELETE CASCADE
);

CREATE TABLE follow_notification (
    id               INTEGER,
    follower_id      INTEGER         NOT NULL,
    CONSTRAINT follow_notification_pk PRIMARY KEY (id),
    CONSTRAINT follow_notification_notification_fk FOREIGN KEY (id)
REFERENCES notification(id) ON DELETE CASCADE,
    CONSTRAINT follow_notification_follower_fk
FOREIGN KEY (follower_id) REFERENCES member(id)
);
```

```
-----------------------------------------
-- Triggers and Functions
-----------------------------------------

CREATE FUNCTION data_anonymization()
RETURNS TRIGGER AS
$BODY$
BEGIN
    UPDATE member SET
        email = '',
        username = '',
        password = '',
        first_name = 'Deleted',
        last_name = 'User',
        biography = '',
        profile_image_id = 0,
        is_deleted = TRUE
    WHERE id = OLD.id;

    DELETE FROM profile_image WHERE id != 0 AND id = OLD.profile_image_id;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER data_anonymization
    BEFORE UPDATE OF is_deleted ON member
    FOR EACH ROW
    WHEN (OLD.is_deleted = false AND NEW.is_deleted = true)
    EXECUTE PROCEDURE data_anonymization();


CREATE FUNCTION delete_content_item()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (OLD.id IN (SELECT id FROM article)) THEN
        UPDATE content_item AS ct
        SET is_deleted = TRUE
        FROM comment AS c
        WHERE c.article_id = OLD.id AND c.id = ct.id AND
c.is_reply = FALSE;
    ELSIF (OLD.id IN (SELECT id FROM comment)) THEN
        IF ((SELECT is_reply FROM comment WHERE comment.id = OLD.id) = FALSE)
THEN
            UPDATE content_item AS ct
            SET is_deleted = TRUE
            FROM comment AS c
            WHERE c.reply_id = OLD.id AND c.id = ct.id AND c.is_reply = TRUE;
        END IF;
    END IF;
    UPDATE member
    SET academy_score = member.academy_score - OLD.academy_score
    WHERE member.id = OLD.author_id;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER delete_content_item
    BEFORE UPDATE OF is_deleted ON content_item
    FOR EACH ROW
    WHEN (NEW.is_deleted = TRUE AND OLD.is_deleted = FALSE)
    EXECUTE PROCEDURE delete_content_item();

CREATE FUNCTION before_permanent_delete_article()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM vote
    WHERE vote.content_item_id = OLD.id;

    DELETE FROM comment
    WHERE comment.article_id = OLD.id AND comment.is_reply = FALSE;

    DELETE FROM article_image
    WHERE article_image.article_id = OLD.id;

    RETURN OLD;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER before_permanent_delete_article
    BEFORE DELETE ON article
    FOR EACH ROW
    EXECUTE PROCEDURE before_permanent_delete_article();


CREATE FUNCTION after_permanent_delete_article()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM content_item
    WHERE content_item.id = OLD.id;

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER after_permanent_delete_article
```

```sql
    AFTER DELETE ON article
    FOR EACH ROW
    EXECUTE PROCEDURE after_permanent_delete_article();


CREATE FUNCTION before_delete_comment()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM vote
    WHERE vote.content_item_id = OLD.id;

    IF (OLD.is_reply = FALSE) THEN
        DELETE FROM comment
        WHERE comment.reply_id = OLD.id AND comment.is_reply = TRUE;
    END IF;

    RETURN OLD;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER before_delete_comment
    BEFORE DELETE ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE before_delete_comment();


CREATE FUNCTION after_delete_comment()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM content_item
    WHERE content_item.id = OLD.id;

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER after_delete_comment
    AFTER DELETE ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE after_delete_comment();


CREATE FUNCTION ban_tag()
RETURNS TRIGGER AS
$BODY$
BEGIN
    DELETE FROM tag_article
    WHERE tag_article.tag_id = OLD.id;
    DELETE FROM follow_tag
    WHERE follow_tag.tag_id = OLD.id;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER ban_tag
    BEFORE UPDATE OF is_banned ON tag
    FOR EACH ROW
    WHEN (OLD.is_banned = false AND NEW.is_banned = true)
    EXECUTE PROCEDURE ban_tag();


CREATE FUNCTION edit_content_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN
    IF (NEW.author_id = (SELECT author_id FROM content_item
WHERE content_item.id = NEW.content_item_id)) THEN
        RETURN NULL;
    END IF;

    INSERT INTO notification (notified_id)
    VALUES ((SELECT author_id FROM content_item
WHERE content_item.id = NEW.content_item_id))
    RETURNING id INTO notification_id;

    INSERT INTO content_notification (id, content_item_id)
    VALUES (notification_id, NEW.content_item_id);

    INSERT INTO edit_notification (id)
    VALUES (notification_id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER edit_content_notification
    AFTER INSERT ON edit
    FOR EACH ROW
    WHEN (NEW.altered_field != 'is_deleted')
    EXECUTE FUNCTION edit_content_notification();


CREATE FUNCTION remove_topic()
RETURNS TRIGGER AS
$BODY$
BEGIN
```

```
    UPDATE article
    SET topic_id = 0
    WHERE topic_id = OLD.id;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER remove_topic
    BEFORE DELETE ON topic
    FOR EACH ROW
    EXECUTE PROCEDURE remove_topic();


CREATE FUNCTION generate_undefined_topic_notification(author_id INT,
article_id INT)
RETURNS VOID AS
$BODY$
DECLARE
    notification_id INTEGER;
BEGIN
    INSERT INTO notification (notified_id)
    VALUES (author_id)
    RETURNING id INTO notification_id;

    INSERT INTO content_notification (id, content_item_id)
    VALUES (notification_id, article_id);

    INSERT INTO undefined_topic_notification (id)
    VALUES (notification_id);

    RETURN;
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION notify_undefined_topic()
RETURNS TRIGGER AS
$BODY$
DECLARE
    author_id INTEGER;
BEGIN
    SELECT content_item.author_id, content_item.id
    FROM content_item
    WHERE content_item.id = OLD.id
    INTO author_id;

    EXECUTE generate_undefined_topic_notification(author_id, OLD.id);

    RETURN NEW;
END
$BODY$
```

```
LANGUAGE plpgsql;


CREATE TRIGGER notify_undefined_topic
    AFTER UPDATE OF topic_id ON article
    FOR EACH ROW
    WHEN (OLD.topic_id != 0 AND NEW.topic_id = 0)
    EXECUTE FUNCTION notify_undefined_topic();


CREATE FUNCTION vote_for_own_content()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (NEW.member_id = (SELECT author_id FROM content_item
WHERE content_item.id = NEW.content_item_id)) THEN
        RAISE EXCEPTION 'Members cannot vote on their own content';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER vote_for_own_content
    BEFORE INSERT ON vote
    FOR EACH ROW
    EXECUTE FUNCTION vote_for_own_content();


CREATE FUNCTION news_article_tags()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF ((6 < (SELECT COUNT(*) FROM tag_article
WHERE tag_article.article_id = NEW.article_id))) THEN
        RAISE EXCEPTION 'Cannot add more than 6 tags to a news article';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER news_article_tags
    BEFORE INSERT ON tag_article
    FOR EACH ROW
    EXECUTE PROCEDURE news_article_tags();


CREATE FUNCTION blocked_user_appeal()
RETURNS TRIGGER AS
$BODY$
```

```
BEGIN
    IF ((SELECT COUNT(*) FROM appeal WHERE appeal.submitter_id =
NEW.submitter_id) > 0) THEN
        RAISE EXCEPTION 'Cannot appeal more than one time';
    END IF;

    IF ((SELECT is_blocked FROM member WHERE member.id =
NEW.submitter_id) = false) THEN
        RAISE EXCEPTION 'Cannot appeal if not blocked';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER blocked_user_appeal
    BEFORE INSERT ON appeal
    FOR EACH ROW
    EXECUTE PROCEDURE blocked_user_appeal();


CREATE FUNCTION topic_control()
RETURNS TRIGGER AS
$BODY$
BEGIN
    INSERT INTO TOPIC (name)
    VALUES (NEW.name);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER topic_control
    AFTER UPDATE OF status ON topic_suggestion
    FOR EACH ROW
    WHEN (OLD.status = 'Pending' AND NEW.status = 'Accepted')
    EXECUTE PROCEDURE topic_control();


CREATE FUNCTION comment_date_validation()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF ((SELECT date_time FROM content_item WHERE content_item.id = NEW.id) <
(SELECT date_time FROM content_item WHERE content_item.id =
NEW.article_id)) THEN
        RAISE EXCEPTION 'Comment date cannot precede article date';
    END IF;

    RETURN NEW;
END
```

```
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER comment_date_validation
    BEFORE INSERT ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE comment_date_validation();


CREATE FUNCTION comment_reply_validation()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (
        NEW.is_reply = TRUE
        AND
        (SELECT date_time FROM content_item WHERE content_item.id =
NEW.reply_id)
        >
        (SELECT date_time FROM content_item WHERE content_item.id = NEW.id)
    ) THEN
        RAISE EXCEPTION 'Reply date cannot precede comment date';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER comment_reply_validation
    BEFORE INSERT ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE comment_reply_validation();


CREATE FUNCTION reply_to_comment_validation()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF (
        NEW.is_reply = TRUE
        AND
        (SELECT is_reply FROM comment WHERE comment.id = NEW.reply_id) =
TRUE) THEN
        RAISE EXCEPTION 'Cannot reply to a reply';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;
```

```
CREATE TRIGGER reply_to_comment_validation
    BEFORE INSERT ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE reply_to_comment_validation();


CREATE FUNCTION report_self()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF ((SELECT submitter_id FROM report WHERE report.id = NEW.id)
= NEW.member_id) THEN
        RAISE EXCEPTION 'A member is not allowed to report themselves';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER report_self
    BEFORE INSERT ON member_report
    FOR EACH ROW
    EXECUTE PROCEDURE report_self();


CREATE FUNCTION report_self_content()
RETURNS TRIGGER AS
$BODY$
BEGIN
    IF ((SELECT submitter_id FROM report WHERE report.id = NEW.id) =
(SELECT author_id FROM content_item WHERE content_item.id =
NEW.content_item_id)) THEN
        RAISE EXCEPTION 'A member is not allowed to report their own content';
    END IF;

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER report_self_content
    BEFORE INSERT ON content_report
    FOR EACH ROW
    EXECUTE PROCEDURE report_self_content();


CREATE FUNCTION generate_block_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN
```

```
    INSERT INTO notification (notified_id)
    VALUES (NEW.id)
    RETURNING id INTO notification_id;

    INSERT INTO block_notification (id)
    VALUES (notification_id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER generate_block_notification
    AFTER UPDATE OF is_blocked ON member
    FOR EACH ROW
    WHEN (OLD.is_blocked = false AND NEW.is_blocked = true)
    EXECUTE PROCEDURE generate_block_notification();


CREATE FUNCTION generate_follow_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN
    INSERT INTO notification (notified_id)
    VALUES (NEW.followed_id)
    RETURNING id INTO notification_id;

    INSERT INTO follow_notification (id, follower_id)
    VALUES (notification_id, NEW.follower_id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER generate_follow_notification
    AFTER INSERT ON follow_member
    FOR EACH ROW
    EXECUTE PROCEDURE generate_follow_notification();


CREATE FUNCTION generate_comment_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN
    INSERT INTO notification (notified_id)
    VALUES ((SELECT author_id FROM content_item WHERE content_item.id =
NEW.article_id))
    RETURNING id INTO notification_id;
    INSERT INTO comment_notification (id, comment_id)
    VALUES (notification_id, NEW.id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER generate_comment_notification
    AFTER INSERT ON comment
    FOR EACH ROW
    EXECUTE PROCEDURE generate_comment_notification();


CREATE FUNCTION update_content_item_academy_score()
RETURNS TRIGGER AS
$BODY$
DECLARE
    total_score INTEGER;
    vote_print vote;
    count INTEGER;
BEGIN
    IF (TG_OP = 'INSERT' OR TG_OP = 'UPDATE') THEN
        SELECT SUM(weight) INTO total_score
        FROM vote
        WHERE vote.content_item_id = NEW.content_item_id;
    ELSIF (TG_OP = 'DELETE') THEN
        SELECT SUM(weight) INTO total_score
        FROM vote
        WHERE content_item_id = OLD.content_item_id;
    END IF;

    UPDATE content_item
    SET academy_score = COALESCE(total_score, 0)
    WHERE content_item.id = COALESCE(NEW.content_item_id,
OLD.content_item_id);

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER update_content_item_academy_score
    AFTER INSERT OR DELETE OR UPDATE ON vote
    FOR EACH ROW
    EXECUTE PROCEDURE update_content_item_academy_score();


CREATE FUNCTION update_member_academy_score()
RETURNS TRIGGER AS
$BODY$
DECLARE
```

```
    total_score INT;
BEGIN
    SELECT SUM(academy_score) INTO total_score
    FROM content_item
    WHERE author_id = NEW.author_id AND is_deleted = FALSE;

    UPDATE member
    SET academy_score = total_score
    WHERE member.id = NEW.author_id;

    RETURN NULL;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER update_member_academy_score
    AFTER UPDATE OF academy_score ON content_item
    FOR EACH ROW
    EXECUTE PROCEDURE update_member_academy_score();

CREATE FUNCTION generate_removal_notification()
RETURNS TRIGGER AS
$BODY$
DECLARE
    notification_id INT;
BEGIN

    IF (NEW.author_id = (SELECT author_id FROM content_item
WHERE content_item.id = NEW.content_item_id)) THEN
        RETURN NULL;
    END IF;

    INSERT INTO notification (notified_id)
    VALUES ((SELECT author_id FROM content_item WHERE content_item.id =
NEW.content_item_id))
    RETURNING id INTO notification_id;

    INSERT INTO content_notification (id, content_item_id)
    VALUES (notification_id, NEW.content_item_id);

    INSERT INTO removal_notification (id)
    VALUES (notification_id);

    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER generate_removal_notification
    AFTER INSERT ON edit
    FOR EACH ROW
    WHEN (NEW.altered_field = 'is_deleted' AND NEW.old_value = 'FALSE'
AND NEW.new_value = 'TRUE')
```

```
    EXECUTE PROCEDURE generate_removal_notification();

CREATE FUNCTION email_lowercase()
RETURNS TRIGGER AS
$BODY$
BEGIN
    NEW.email = LOWER(NEW.email);
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER email_lowercase
    BEFORE INSERT OR UPDATE ON member
    FOR EACH ROW
    EXECUTE PROCEDURE email_lowercase();


CREATE FUNCTION tag_lowercase()
RETURNS TRIGGER AS
$BODY$
BEGIN
    NEW.name = LOWER(NEW.name);
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER tag_lowercase
    BEFORE INSERT OR UPDATE ON tag
    FOR EACH ROW
    EXECUTE PROCEDURE tag_lowercase();


-----------------------------------------
-- Indexes
-----------------------------------------

-- Performance Indexes

CREATE INDEX content_item_date ON content_item USING btree (date_time);
CREATE INDEX content_item_academy_score ON content_item
USING btree (academy_score);
CREATE INDEX article_topic ON article USING hash (topic_id);


-- Full-text Search Indexes

ALTER TABLE content_item
ADD COLUMN tsvectors tsvector;

CREATE FUNCTION content_item_search_update() RETURNS TRIGGER AS $$
BEGIN
```

```
 IF TG_OP = 'INSERT' THEN
        IF (NEW.id IN (SELECT id FROM article)) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english',
(SELECT title FROM article WHERE article.id = NEW.id)), 'A') ||
                setweight(to_tsvector('english', NEW.body), 'B')
            );
        ELSE
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.body), 'B')
            );
        END IF;
 END IF;

 IF TG_OP = 'UPDATE' THEN
        IF (NEW.id IN (SELECT id FROM article)) THEN
            IF (NEW.body <> OLD.body) THEN
                NEW.tsvectors = (
                    setweight(to_tsvector('english',
(SELECT title FROM article WHERE article.id = NEW.id)), 'A') ||
                    setweight(to_tsvector('english', NEW.body), 'B')
                );
            END IF;
        ELSE
            IF (NEW.body <> OLD.body) THEN
                NEW.tsvectors = (
                    setweight(to_tsvector('english', NEW.body), 'B')
                );
            END IF;
        END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER content_item_search_update
BEFORE INSERT OR UPDATE ON content_item
FOR EACH ROW
EXECUTE PROCEDURE content_item_search_update();


CREATE FUNCTION article_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        IF (NEW.id IN (SELECT id FROM content_item)) THEN
            UPDATE content_item
            SET tsvectors = (
                setweight(to_tsvector('english',
(SELECT title FROM article WHERE article.id = NEW.id)), 'A') ||
                setweight(to_tsvector('english',
(SELECT body FROM content_item WHERE content_item.id = NEW.id)), 'B')
            )
            WHERE content_item.id = NEW.id;
        END IF;
 END IF;

 IF TG_OP = 'UPDATE' THEN
        IF (NEW.id IN (SELECT id FROM content_item)) THEN
            IF (NEW.title <> OLD.title) THEN
                UPDATE content_item
                SET tsvectors = (
                    setweight(to_tsvector('english',
(SELECT title FROM article WHERE article.id = NEW.id)), 'A') ||
                    setweight(to_tsvector('english',
(SELECT body FROM content_item WHERE content_item.id = NEW.id)), 'B')
                )
                WHERE content_item.id = NEW.id;
            END IF;
        END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER article_search_update
BEFORE INSERT OR UPDATE ON article
FOR EACH ROW
EXECUTE PROCEDURE article_search_update();


CREATE INDEX content_item_search ON content_item USING GIST (tsvectors);

ALTER TABLE member
ADD COLUMN tsvectors tsvector;

CREATE FUNCTION member_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.username), 'A') ||
            setweight(to_tsvector('english', NEW.first_name), 'B') ||
            setweight(to_tsvector('english', NEW.last_name), 'B')
        );
 END IF;

 IF TG_OP = 'UPDATE' THEN
        IF (NEW.username <> OLD.username) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.username), 'A') ||
                setweight(to_tsvector('english', NEW.first_name), 'B') ||
                setweight(to_tsvector('english', NEW.last_name), 'B')
            );
        END IF;
 END IF;
 RETURN NEW;
END $$
```

```
LANGUAGE plpgsql;

CREATE TRIGGER member_search_update
BEFORE INSERT OR UPDATE ON member
FOR EACH ROW
EXECUTE PROCEDURE member_search_update();

CREATE INDEX member_search ON member USING GIN (tsvectors);
```

A.2. Database population

```
SET search_path TO lbaw2356;

DROP FUNCTION IF EXISTS create_imdb_info(query_type VARCHAR(255),
imdb_info_id VARCHAR(255));
DROP FUNCTION IF EXISTS create_profile_image(file_name VARCHAR(255));
DROP FUNCTION IF EXISTS create_member(email VARCHAR(255),
password VARCHAR(255),
username VARCHAR(255), first_name VARCHAR(255), last_name VARCHAR(255),
biography TEXT);
DROP FUNCTION IF EXISTS create_admin(email VARCHAR(255),
password VARCHAR(255), username VARCHAR(255), first_name VARCHAR(255),
last_name VARCHAR(255), biography TEXT);
DROP FUNCTION IF EXISTS create_appeal(body VARCHAR(255),
submitter_id INTEGER);
DROP FUNCTION IF EXISTS create_follow_member(follower_id INTEGER,
followed_id INTEGER);
DROP FUNCTION IF EXISTS create_topic(name VARCHAR(255));
DROP FUNCTION IF EXISTS create_topic_suggestion(name VARCHAR(255),
status statuses, suggester_id INTEGER);
DROP FUNCTION IF EXISTS create_content_item(body TEXT, author_id INTEGER);
DROP FUNCTION IF EXISTS create_article(title VARCHAR(255), body TEXT,
author_id INTEGER, topic_id INTEGER, imdb_info_id INTEGER);
DROP FUNCTION IF EXISTS create_article_image(file_name VARCHAR(255),
article_id INTEGER);
DROP FUNCTION IF EXISTS create_tag(name VARCHAR(255));
DROP FUNCTION IF EXISTS create_tag_article(tag_id INTEGER,
article_id INTEGER);
DROP FUNCTION IF EXISTS create_follow_tag(tag_id INTEGER,
member_id INTEGER);
DROP FUNCTION IF EXISTS create_comment(body TEXT, author_id INTEGER,
is_reply BOOLEAN, article_id INTEGER, reply_id INTEGER);
DROP FUNCTION IF EXISTS create_edit(altered_field VARCHAR(255),
old_value TEXT, new_value TEXT, content_item_id INTEGER,
author_id INTEGER);
DROP FUNCTION IF EXISTS create_vote(member_id INTEGER,
content_item_id INTEGER, weight INTEGER);
DROP FUNCTION IF EXISTS create_report(body TEXT,
submitter_id INTEGER);
DROP FUNCTION IF EXISTS create_content_report(body TEXT,
```

```sql
submitter_id INTEGER,
motive motives, content_item_id INTEGER);
DROP FUNCTION IF EXISTS create_member_report(body TEXT,
submitter_id INTEGER,
misconduct misconduct_types, member_id INTEGER);
DROP FUNCTION IF EXISTS create_notification(notified_id INTEGER);
DROP FUNCTION IF EXISTS create_comment_notification(notified_id INTEGER,
comment_id INTEGER);
DROP FUNCTION IF EXISTS create_content_notification(
notified_id INTEGER, content_item_id INTEGER);
DROP FUNCTION IF EXISTS create_upvote_notification(
notified_id INTEGER, content_item_id INTEGER, amount INTEGER);
DROP FUNCTION IF EXISTS create_edit_notification(
notified_id INTEGER, content_item_id INTEGER);
DROP FUNCTION IF EXISTS create_removal_notification(
notified_id INTEGER, content_item_id INTEGER);
DROP FUNCTION IF EXISTS create_undefined_topic_notification(
notified_id INTEGER, content_item_id INTEGER);
DROP FUNCTION IF EXISTS create_block_notification(
notified_id INTEGER);
DROP FUNCTION IF EXISTS create_follow_notification(
notified_id INTEGER, follower_id INTEGER);

CREATE FUNCTION create_imdb_info(query_type VARCHAR(255),
imdb_info_id VARCHAR(255))
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO imdb_info (query_type, imdb_info_id)
    VALUES (query_type, imdb_info_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_profile_image(file_name VARCHAR(255))
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO profile_image (file_name)
    VALUES (file_name);
END
$BODY$
LANGUAGE plpgsql;


CREATE FUNCTION create_member(email VARCHAR(255),
password VARCHAR(255),
username VARCHAR(255), first_name VARCHAR(255),
last_name VARCHAR(255), biography TEXT)
RETURNS INTEGER AS
$BODY$
DECLARE
    member_id INTEGER;
```

```
BEGIN
    INSERT INTO member (email, password, username,
first_name, last_name, biography)
    VALUES (email, password, username, first_name,
last_name, biography)
    RETURNING id INTO member_id;

    RETURN member_id;
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_admin(email VARCHAR(255),
password VARCHAR(255),
username VARCHAR(255), first_name VARCHAR(255),
last_name VARCHAR(255), biography TEXT)
RETURNS VOID AS
$BODY$
DEClARE
    member_id INTEGER;
BEGIN
    member_id := create_member(email, password,
username, first_name,
last_name, biography);
    UPDATE member
    SET is_admin = TRUE
    WHERE id = member_id;
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_appeal(body VARCHAR(255),
submitter_id INTEGER)
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO appeal (body, submitter_id)
    VALUES (body, submitter_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_follow_member(follower_id INTEGER,
followed_id INTEGER)
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO follow_member (follower_id, followed_id)
    VALUES (follower_id, followed_id);
END
$BODY$
LANGUAGE plpgsql;
```

```
CREATE FUNCTION create_topic(name VARCHAR(255))
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO topic (name)
    VALUES (name);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_topic_suggestion(name VARCHAR(255),
status statuses,
suggester_id INTEGER)
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO topic_suggestion (name, status, suggester_id)
    VALUES (name, status, suggester_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_content_item(body TEXT, author_id INTEGER)
RETURNS INTEGER AS
$BODY$
DECLARE
    content_item_id INTEGER;
BEGIN
    INSERT INTO content_item (body, author_id)
    VALUES (body, author_id)
    RETURNING id INTO content_item_id;

    RETURN content_item_id;
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_article(title VARCHAR(255), body TEXT,
author_id INTEGER, topic_id INTEGER, imdb_info_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_content_item(body, author_id);

    INSERT INTO article (id, title, topic_id, imdb_info_id)
    VALUES (id, title, topic_id, imdb_info_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_article_image(file_name VARCHAR(255),
```

```sql
article_id INTEGER)
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO article_image (file_name, article_id)
    VALUES (file_name, article_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_tag(name VARCHAR(255))
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO tag (name)
    VALUES (name);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_tag_article(tag_id INTEGER, article_id INTEGER)
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO tag_article (tag_id, article_id)
    VALUES (tag_id, article_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_follow_tag(tag_id INTEGER, member_id INTEGER)
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO follow_tag (tag_id, member_id)
    VALUES (tag_id, member_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_comment(body TEXT, author_id INTEGER,
is_reply BOOLEAN, article_id INTEGER, reply_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_content_item(body, author_id);
    INSERT INTO comment (id, is_reply, article_id, reply_id)
    VALUES (id, is_reply, article_id, reply_id);
END
$BODY$
LANGUAGE plpgsql;
```

```sql
CREATE FUNCTION create_edit(altered_field VARCHAR(255), old_value TEXT,
new_value TEXT, content_item_id INTEGER, author_id INTEGER)
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO edit (altered_field, old_value, new_value,
content_item_id, author_id)
    VALUES (altered_field, old_value, new_value,
content_item_id, author_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_vote(member_id INTEGER,
content_item_id INTEGER, weight INTEGER)
RETURNS VOID AS
$BODY$
BEGIN
    INSERT INTO vote (member_id, content_item_id, weight)
    VALUES (member_id, content_item_id, weight);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_report(body TEXT, submitter_id INTEGER)
RETURNS INTEGER AS
$BODY$
DECLARE
    report_id INTEGER;
BEGIN
    INSERT INTO report (body, submitter_id)
    VALUES (body, submitter_id)
    RETURNING id INTO report_id;

    RETURN report_id;
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_content_report(body TEXT, submitter_id INTEGER,
motive motives, content_item_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_report(body, submitter_id);
    INSERT INTO content_report (id, motive, content_item_id)
    VALUES (id, motive, content_item_id);
END
$BODY$
LANGUAGE plpgsql;
```

```sql
CREATE FUNCTION create_member_report(body TEXT, submitter_id INTEGER,
misconduct misconduct_types, member_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_report(body, submitter_id);
    INSERT INTO member_report (id, misconduct, member_id)
    VALUES (id, misconduct, member_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_notification(notified_id INTEGER)
RETURNS INTEGER AS
$BODY$
DECLARE
    notification_id INTEGER;
BEGIN
    INSERT INTO notification (notified_id)
    VALUES (notified_id)
    RETURNING id INTO notification_id;

    RETURN notification_id;
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_comment_notification(notified_id INTEGER,
comment_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_notification(comment_id);
    INSERT INTO comment_notification (id, comment_id)
    VALUES (id, comment_id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_content_notification(notified_id INTEGER,
content_item_id INTEGER)
RETURNS INTEGER AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_notification(notified_id);
    INSERT INTO content_notification (id, content_item_id)
    VALUES (id, content_item_id);

    RETURN id;
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_upvote_notification(notified_id INTEGER,
content_item_id INTEGER, amount INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_content_notification(notified_id, content_item_id);
    INSERT INTO upvote_notification (id, amount)
    VALUES (id, amount);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_edit_notification(notified_id INTEGER,
content_item_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_content_notification(notified_id, content_item_id);
    INSERT INTO edit_notification (id)
    VALUES (id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_removal_notification(notified_id INTEGER,
content_item_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_content_notification(notified_id, content_item_id);
    INSERT INTO removal_notification (id)
    VALUES (id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_undefined_topic_notification(notified_id INTEGER,
content_item_id INTEGER)
RETURNS VOID AS
$BODY$
```

```
DECLARE
    id INTEGER;
BEGIN
    id := create_content_notification(notified_id, content_item_id);
    INSERT INTO undefined_topic_notification (id)
    VALUES (id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_block_notification(notified_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_notification(notified_id);
    INSERT INTO block_notification (id)
    VALUES (id);
END
$BODY$
LANGUAGE plpgsql;

CREATE FUNCTION create_follow_notification(notified_id INTEGER,
follower_id INTEGER)
RETURNS VOID AS
$BODY$
DECLARE
    id INTEGER;
BEGIN
    id := create_notification(notified_id);
    INSERT INTO follow_notification (id, follower_id)
    VALUES (id, follower_id);
END
$BODY$
LANGUAGE plpgsql;



SELECT create_imdb_info('movie', 'tt0111161');
SELECT create_imdb_info('movie', 'tt0068646');
SELECT create_imdb_info('name', 'tt0071562');

INSERT INTO profile_image (id, file_name) VALUES (0, 'default.jpg');
SELECT create_profile_image('profile_image_1.jpg');
SELECT create_profile_image('profile_image_2.jpg');
SELECT create_profile_image('profile_image_3.jpg');

SELECT create_member('harvey.specter@popcornpost.com', 'password', 'harveyspecter
SELECT create_member('mike.ross@popcornpost.com', 'password', 'mikeross', 'Mike',
SELECT create_member('donna.paulsen@popcornpost.com', 'password', 'donnapaulsen',
SELECT create_member('louis.litt@popcorpost.com', 'password', 'louislitt', 'Louis
SELECT create_admin('jessica.pearson@popcornpost.com', 'password', 'jessicapearso
SELECT create_member('harold.jakowski@popcornpost.com', 'password', 'haroldjakows

SELECT create_follow_member(1,2);
SELECT create_follow_member(1,3);
SELECT create_follow_member(1,5);
SELECT create_follow_member(2,1);
SELECT create_follow_member(2,3);
SELECT create_follow_member(2,5);
SELECT create_follow_member(3,1);
SELECT create_follow_member(3,2);
SELECT create_follow_member(3,4);
SELECT create_follow_member(3,5);
SELECT create_follow_member(4,1);

INSERT INTO topic (id, name) VALUES (0, 'Undefined');
SELECT create_topic('Review');
SELECT create_topic('Leaks');
SELECT create_topic('Interview');
SELECT create_topic('Analysis');
SELECT create_topic('Announcement');

SELECT create_topic_suggestion('Awards', 'Pending', 1);
SELECT create_topic_suggestion('Rumors', 'Pending', 3);
SELECT create_topic_suggestion('Drama', 'Rejected', 4);
UPDATE topic_suggestion SET status = 'Accepted' WHERE id = 2;

SELECT create_article('The Godfather: A Cinematic Masterpiece that Suits Every Ta
SELECT create_article('Defending the Silver Screen: Hollywood''s Legal Dramas', '
SELECT create_article('Donna Paulsen unveils Donna Paulsen''s show - Suits', 'Don
SELECT create_article('Luis Litt''s Favorite Movie: A Lesson in Tenacity', 'Hey,

SELECT create_article_image('article_image_1.jpg', 1);
SELECT create_article_image('article_image_2.jpg', 1);
SELECT create_article_image('article_image_3.jpg', 1);
SELECT create_article_image('article_image_4.jpg', 3);
SELECT create_article_image('article_image_5.jpg', 3);
SELECT create_article_image('article_image_6.jpg', 4);

SELECT create_tag('godfather');
SELECT create_tag('suits');
SELECT create_tag('donna_paulsen');
SELECT create_tag('basketball');
SELECT create_tag('opera');

SELECT create_tag_article(1, 1);
SELECT create_tag_article(2, 3);
SELECT create_tag_article(3, 3);

SELECT create_follow_tag(1, 1);
SELECT create_follow_tag(1, 2);
SELECT create_follow_tag(2, 1);
SELECT create_follow_tag(2, 2);
SELECT create_follow_tag(2, 3);
```

```sql
SELECT create_follow_tag(2, 4);
SELECT create_follow_tag(2, 5);
SELECT create_follow_tag(3, 3);
SELECT create_follow_tag(4, 1);
SELECT create_follow_tag(5, 3);
SELECT create_follow_tag(5, 4);

SELECT create_comment('I love this movie! It is a masterpiece!', 2, FALSE, 1, NUL
SELECT create_comment('I have seen better...', 4, TRUE, 1, 5);
SELECT create_comment('I have read better articles', 4, FALSE, 1, NULL);
SELECT create_comment('Louis... cut it out.', 3, TRUE, 1, 5);
SELECT create_comment('Fantastic work, I wish more people shared this view.', 5,
SELECT create_comment('I do share this view as well.', 2, TRUE, 2, 9);
SELECT create_comment('Only you to write an article about yourself.', 1, FALSE, 3
SELECT create_comment('Wow... just, wow...', 2, TRUE, 3, 11);
SELECT create_comment('Wonderful read!', 4, FALSE, 3, NULL);

SELECT create_edit('title', 'Donna Paulsen unveils the Donna Paulsen ', 'Donna Pa
UPDATE article SET title = 'Donna Paulsen Uncovers Exclusive Behind-the-Scenes Se

UPDATE article SET topic_id = 0 WHERE id = 3;

SELECT create_edit('body', 'I love this movie! It is a masterpiece!', 'I love thi
UPDATE content_item SET body = 'I love this movie!' WHERE id = 5;

SELECT create_vote(2, 1, 1);
SELECT create_vote(3, 1, 1);
SELECT create_vote(4, 1, -1);
SELECT create_vote(5, 1, -1);
SELECT create_vote(2, 2, 1);
SELECT create_vote(3, 2, 1);
SELECT create_vote(4, 2, -1);
SELECT create_vote(1, 3, 1);
SELECT create_vote(2, 3, 1);
SELECT create_vote(4, 3, 1);
SELECT create_vote(5, 3, 1);
SELECT create_vote(6, 3, 1);
SELECT create_vote(3, 4, 1);
SELECT create_vote(1, 4, -1);
SELECT create_vote(1, 5, 1);
SELECT create_vote(1, 6, -1);
SELECT create_vote(2, 6, -1);
SELECT create_vote(3, 6, -1);
SELECT create_vote(5, 6, -1);
SELECT create_vote(1, 7, -1);
SELECT create_vote(2, 7, -1);
SELECT create_vote(2, 8, 1);
SELECT create_vote(5, 8, 1);
SELECT create_vote(1, 9, 1);
SELECT create_vote(3, 11, 1);
SELECT create_vote(3, 12, 1);
SELECT create_vote(3, 13, 1);
SELECT create_upvote_notification(3, 3, 5);

SELECT create_content_report(NULL, 1, 'Misinformation', 4);

SELECT create_member_report('This member is a disgrace to the legal profession.',

UPDATE content_item SET is_deleted = TRUE WHERE id = 4;
SELECT create_edit('is_deleted', 'FALSE', 'TRUE', 4, 5);

UPDATE member SET is_blocked = TRUE WHERE id = 4;

SELECT create_appeal('I did not do anything wrong, I demand to be unblocked!', 4)
```

Revision history

Changes made to the first submission:

1. Item 1
2. ..

GROUP2356, 11/10/2023

- Marco André Pereira da Costa, up202108821@up.pt (Editor)
- João Filipe Oliveira Ramos, up202108743@up.pt
- Tiago Filipe Castro Viana, up201807126@up.pt
- Diogo Alexandre Figueiredo Gomes, up201905991@up.pt