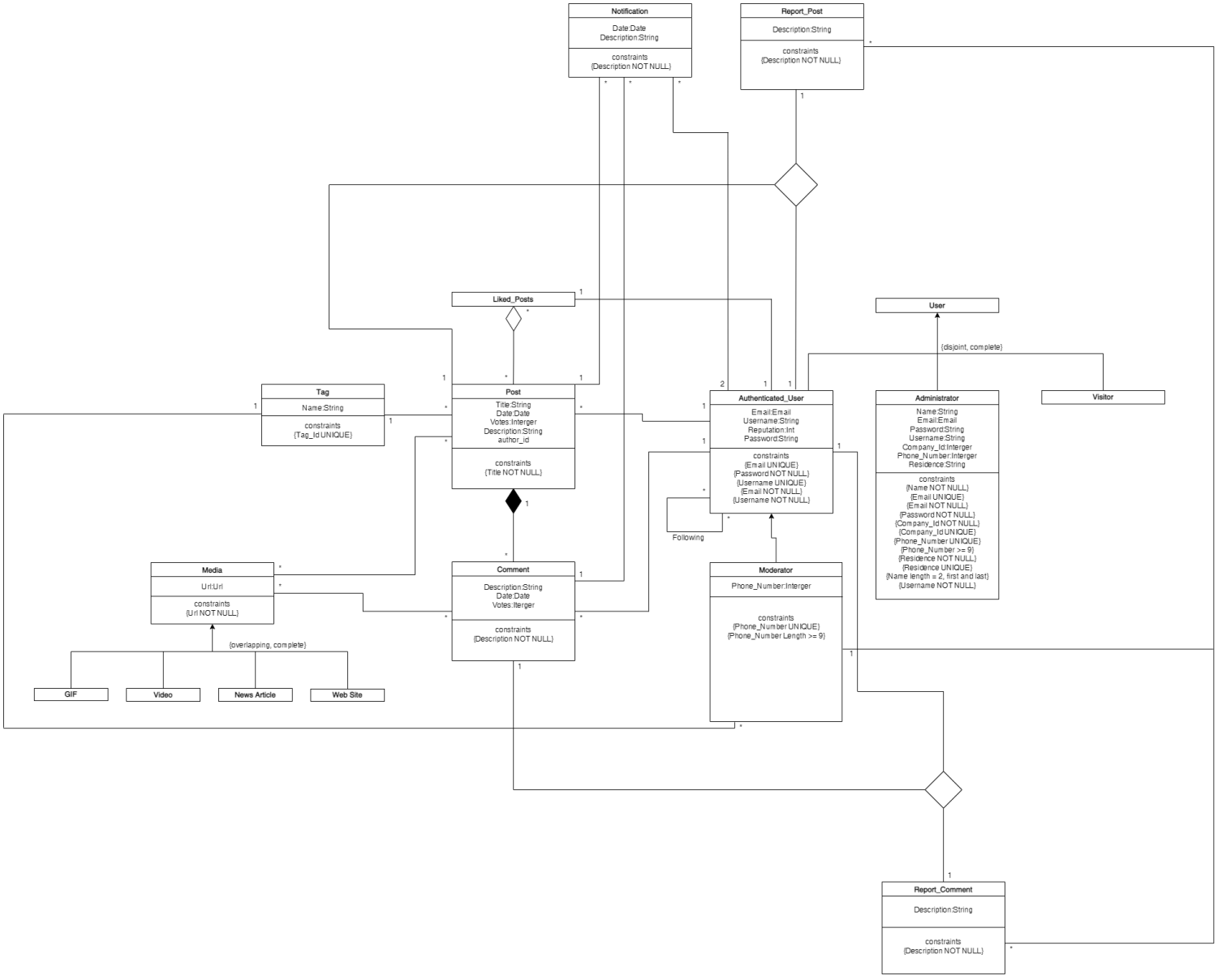
EBD: Database Specification Component

A4: Conceptual Data Model

The Conceptual Data Model helps us map out the relationships present and classes needed in the creation of our database.
For this we will use UML, and start to include the classes mentioned in previous artifacts.

4.1 Class Diagram



4.2 Business Rules

- BR01. A user cannot upvote and downvote the same post.
- BR02. A user can only upvote a post once.
- BR03. A user can only downvote a post once.
- BR04. A user can only upvote each comment once.
- BR05. A user can only downvote each comment once.

A5: Relation Schema, Validation and Schema Refinement

This artifact details the relation schema gathered through our UML diagram. The relation schema maps out the attributes, domains, primary keys, foreign keys and integrity rules such as: NO NULL, DEFAULT, CHECK and UNIQUE.

5.1 Relational Schema

Below is the table containing the relations schemas we will be using on our system.

R01	Authenticated_User(user_id -> user, email **UK NN**, username **NN UK**, reputation, password **NN)**
R02	Administrator(name **NN**, email **UK NN**, password **NN**, username **NN**, company_id **UK NN**, phone_number **UK NN**, residence **UK NN**)
R03	Moderator(authenticated_user_id -> user, phone_number **UK NN**)
R04	Post(post_id, title **NN**, date **DF**, votes, description, type **IN** Type, tag_id -> tag, author_id ->user)
R05	Comment(comment_id, post_id -> post, description, date **DF**, votes, type **IN** Type)
R06	Tag(tag_id, name **UK**)
R07	Report_Post(report_post_id, post_id -> post, user_id -> user , description **NN**)

R08	Report_Comment(report_comment_id, comment_id -> comment, user_id -> user, description **NN**)
R09	Liked_Posts(user_id -> user, post_id -> post, down_or_upvote **NN**)
R10	Media(url **UK NN**, post_id -> post, comment_id -> comment)
R11	User(user_id)
R12	Visitor(visitor_id -> user_id)
R13	Following(authenticated_user_id1 -> user, authenticated_user_id2 -> user)
R14	Notification(authenticated_user_id1 -> user, authenticated_user_id2 -> user, comment_id -> comment, post_id -> post, date **DF**, description **NN**)

Legend:

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

5.2 Domains

Table of additional domains.

Today	DATE DEFAULT CURRENT DATE
Type	ENUM(None, ‘GIF’, ‘VIDEO’, ‘NEWS ARTICLE’, ‘WEB SITE’)

5.3 Schema Validation

Table R01 (Authenticated_User)	
Keys: {email},{username},{user_id}	
Functional Dependencies	
FD0101	{email}→{username, password}
FD0102	{username}→{reputation}
FD0103	{user_id}→{email, username, reputation, password}
Normal Form	BCNF

Table R02 (Administrator)	
Keys: {email}, {company_id}, {phone_number}, {residence}	
Functional Dependencies	
FD0201	{email}→{password, username}
FD0202	{company_id}→{name, email, password, username, company_id, phone_number, residence}
FD0203	{phone_number}→{name, email, password, username, company_id, phone_number, residence}
Normal Form	BCNF

Table R04 (Post)	
Keys: {post_id}, {tag_id}	
Functional Dependencies	
FD0301	{post_id}→{title, date, votes, description, type, tag_id, author_id}
Normal Form	BCNF

Table R05 (Comment)	
Keys: {comment_id}, {post_id}	
Functional Dependencies	
FD0501	{comment_id}→{post_id, description, date, votes, type}
Normal Form	BCNF

Table R06 (Tag)	
Keys: {tag_id}	
Functional Dependencies	
FD0601	{tag_id}→{name}
Normal Form	BCNF

Table R07 (Report_Post)	
Keys: {report_post_id}, {post_id},{user_id}	
Functional Dependencies	
FD0701	{report_post_id}→{post_id, user_id, description}
Normal Form	BCNF

Table R08 (Report_Comment)	
Keys: {report_comment_id}, {post_id},{user_id}	
Functional Dependencies	
FD0801	{report_comment_id}→{comment_id, user_id, description}
Normal Form	BCNF

Annex A. SQL Code

The base sql used in this project is included as an annex of the EBD component. The creation script and population script should be included separately.

A.1 Database Schema

```
-- Types

CREATE TYPE med AS ENUM ('NONE', 'GIF', 'VIDEO', 'NEWS ARTICLE', 'WEB
SITE');

-- Tables

-- Table named users instead of user because user is a reserved word in
PostgreSQL.
CREATE TABLE users (
    id SERIAL PRIMARY KEY
);

CREATE TABLE authenticated_users (
    email TEXT NOT NULL UNIQUE,
    username TEXT NOT NULL UNIQUE,
    reputation TEXT,
    password TEXT NOT NULL,
    user_id INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE administrator (
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    username TEXT NOT NULL,
    company_id SERIAL NOT NULL UNIQUE,
    phone_number VARCHAR NOT NULL UNIQUE,
    residence TEXT NOT NULL UNIQUE
);
```

```
CREATE TABLE moderator (
    phone_number VARCHAR NOT NULL UNIQUE,
    authenticated_user_id INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE tag (
    tag_id SERIAL PRIMARY KEY,
    name TEXT UNIQUE
);
```

```sql
CREATE TABLE post (
    post_id SERIAL PRIMARY KEY,
    author_id INT REFERENCES users(id) ON UPDATE CASCADE,
    title TEXT NOT NULL,
    date TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
    votes INT,
    description TEXT,
    tag_id INT REFERENCES tag(tag_id) ON UPDATE CASCADE,
    TYPE med
);

CREATE TABLE comment (
    comment_id SERIAL PRIMARY KEY,
    description TEXT,
    date TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
    votes INT,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    TYPE med
);

CREATE TABLE report_post (
    report_post_id SERIAL,
    description TEXT NOT NULL,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    user_id INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE report_comment (
    report_comment_id SERIAL,
    description TEXT NOT NULL,
    comment_id INT REFERENCES comment(comment_id) ON UPDATE CASCADE,
    user_id INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE liked_posts (
    user_id INT REFERENCES users(id) ON UPDATE CASCADE,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    down_or_upvote BIT NOT NULL
);
```

```sql
CREATE TABLE media (
    url VARCHAR UNIQUE NOT NULL,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    comment_id INT REFERENCES comment(comment_id) ON UPDATE CASCADE
);

CREATE TABLE visitor (
    visitor_id INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE following (
    authenticated_user_id1 INT REFERENCES users(id) ON UPDATE CASCADE,
    authenticated_user_id2 INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE notification (
    authenticated_user_id1 INT REFERENCES users(id) ON UPDATE CASCADE,
    authenticated_user_id2 INT REFERENCES users(id) ON UPDATE CASCADE,
    comment_id INT REFERENCES comment(comment_id) ON UPDATE CASCADE,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    date TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
    description TEXT NOT NULL
);
```

A.2 Database Population

The database population script will be included in A6.

A6: Indexes, triggers, transactions and database population

This artifact helps us improve the performance of our application by allowing us to add indexes to search for specific rows much faster and efficiently. We also have an easier time UPDATE and DELETE them. We also implement transactions and their isolation level and current accesses. Triggers are also implemented to automate and assure certain aspects. And finally, we added a Database Workload that estimates our web application's growth.

6.1 Database Workload

To develop a worthwhile database we should keep in mind our expectations of our application's growth. This workload estimates the number of tuples and estimated growth.

Relation	**Relation Name**	**Order of Magnitude**	**Estimated Growth**
R01	Authenticated_User	10k (ten thousand)	10 (ten) /day
R02	Administrator	100	5 / month
R03	Moderator	500	20 / month
R04	Post	100k (one hundred thousand)	20 / day
R05	Comment	1m	100 / day
R06	Tag	20	5 / month
R07	Report_Post	1k	2 / day
R08	Report_Comment	10k	10 / day
R09	Liked_Posts	100	2 / day
R10	Media	10k	2 / day
R11	User	50k	50 / day
R12	Visitor	30k	30 / day
R13	Following	1k	5 / day
R14	Notification	100k	20 / day

6.2 Proposed Indexes

Indexes are used to improve the performance of the database by allowing the retrieval of rows and information in a more timely manner. It also speeds up query joins and commands like UPDATE and DELETE.

6.2.1. Performance indexes

Performance indexes are essential to improve performance and there can only be three, so it is important to implement the most important ones.

Index	IDX01
Index relation	comment
Index attribute	comment_id
Index type	Hash
Cardinality	High
Clustering	No
Justification	The table ‘comment’ is very extensive and several queries are made to this table. As we are filtering by exact match, a hash makes the most sense as we are not using any clustering.
SQL Code	
CREATE INDEX comments_index **ON comment USING** hash (comment_id);	

Index	IDX02
Index relation	post
Index attribute	post_id
Index type	B-tree
Cardinality	High
Clustering	yes
Justification	The table ‘post’ is very large and very frequently accessed. As we are applying clustering the index type that suits the table best is B-tree. Update frequency should be low so it is a prime candidate for clustering.

SQL Code
CREATE INDEX posts_index **ON post USING** btree (post_id); **CLUSTER** post **USING** posts_index;

Index	IDX03
Index relation	following
Index attribute	authenticated_user_id1
Index type	Hash
Cardinality	Medium
Clustering	No
Justification	'following' is a small table with low update frequency. We are going to apply a hash without clustering as an index.
SQL Code	
CREATE INDEX follows_index **ON following USING** hash (authenticated_user_id1);	

6.2.2. Full-text Search indexes

Full-text search indexes are useful for keyword based search in our database. Records can be attributed a weight and value.

Index	IDX11
Index relation	post
Index attribute	title, description
Index type	GIN
Clustering	No
Justification	It is imperative to provide full-text search features to look for posts based on matching titles. Gin is the Index type we choose as the index attributes are not expected to change a lot.
SQL Code	

```
-- Add column to post to store computed ts_vectors.
ALTER TABLE post
ADD COLUMN tsvectors TSVECTOR;


-- Create a function to automatically update ts_vectors.
CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
            NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.title), 'A') ||
            setweight(to_tsvector('english', NEW.description), 'B')
            );
 END IF;
 IF TG_OP = 'UPDATE' THEN
            IF (NEW.title <> OLD.title OR NEW.description <> OLD.description) THEN
            NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.title), 'A') ||
            setweight(to_tsvector('english', NEW.description), 'B')
            );
            END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;


-- Create a trigger before insert or update on post.
CREATE TRIGGER post_search_update
```

```
BEFORE INSERT OR UPDATE ON post

FOR EACH ROW

EXECUTE PROCEDURE post_search_update();


-- Finally, create a GIN index for ts_vectors.

CREATE INDEX search_idx ON post USING GIN (tsvectors);
```

6.3 Triggers

Triggers are user defined functions used to automate tasks tailor made for the database. Normally used to enforce Business Rules.

Trigger	TRIGGER01
Description	A user can only upvote/downvote a post once and cannot upvote and downvote the same post.
SQL Code	

```
CREATE FUNCTION new_liked_posts() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF EXISTS (SELECT * FROM liked_posts WHERE user_id = NEW.user_id AND
post_id = NEW.post_id) THEN
            RAISE EXCEPTION 'You cannot upvote/downvote twice the same post and
you cannot upvote and downvote the same post.';
       END IF;
       RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

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

Trigger	TRIGGER02
Description	A user can only follow another user once.

SQL Code

```
CREATE FUNCTION new_follow() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF EXISTS (SELECT * FROM following WHERE authenticated_user_id1 =
NEW.authenticated_user_id1 AND authenticated_user_id2 =
NEW.authenticated_user_id2) THEN
        RAISE EXCEPTION 'You cannot follow twice the same user.';
      END IF;
      RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

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

Trigger	TRIGGER03
Description	A user can't follow himself.

SQL Code

```
CREATE FUNCTION follow_himself() RETURNS TRIGGER AS
$BODY$
BEGIN
      IF (NEW.authenticated_user_id1 = NEW.authenticated_user_id2) THEN
        RAISE EXCEPTION 'You cannot follow yourself.';
      END IF;
      RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER follow_himself
BEFORE INSERT OR UPDATE ON following
FOR EACH ROW
EXECUTE PROCEDURE follow_himself();
EXECUTE PROCEDURE new_liked_posts();
```

6.4 Transactions

Transactions are used to ensure the data suffers no losses.

Transaction	TRAN01
Description	Insert a new post
Justification	To maintain the consistency of the information when posting a post, it is necessary to use a transaction to make sure the code executes with no problem. If the execution fails a ROLLBACK will be issued. This transaction's isolation level is Repeatable Read to ensure no data is lost.
Isolation level	Repeatable Read
SQL Code	

```
BEGIN TRANSACTION;

-- Insert post
INSERT INTO post (title, description, date, votes, tag, author_id)
 VALUES ($title, $description, $date, $votes, $tag, $author_id);

END TRANSACTION;
```

Transaction	TRAN02
Description	Insert a comment
Justification	When posting a comment we have to be sure no data is lost so a transaction is needed. It is very similar to the transaction used to insert a post. The isolation should also be Repeatable Read for similar reasons of conserving data.
Isolation level	Repeatable Read
SQL Code	

```
BEGIN TRANSACTION;

-- Insert comment
INSERT INTO comment (comment_id, description, date, votes, post_id)
 VALUES ($comment_id, $description, $date, $votes, $post_id);


END TRANSACTION;
```

Annex A. Complete SQL Code

The first version of Annex A (created with A5) is now updated with the new elements created in the development of A6, specifically: indices, triggers and user defined functions, transactions, and the population script.

A.1 Database schema

```sql
-- Drop old schema

DROP TABLE IF EXISTS users CASCADE;
DROP TABLE IF EXISTS authenticated_users CASCADE;
DROP TABLE IF EXISTS administrator CASCADE;
DROP TABLE IF EXISTS moderator CASCADE;
DROP TABLE IF EXISTS post CASCADE;
DROP TABLE IF EXISTS comment CASCADE;
DROP TABLE IF EXISTS tag CASCADE;
DROP TABLE IF EXISTS report_post CASCADE;
DROP TABLE IF EXISTS report_comment CASCADE;
DROP TABLE IF EXISTS liked_posts CASCADE;
DROP TABLE IF EXISTS media CASCADE;
DROP TABLE IF EXISTS visitor CASCADE;
DROP TABLE IF EXISTS following CASCADE;
DROP TABLE IF EXISTS notification CASCADE;

DROP TYPE IF EXISTS med;

DROP TRIGGER IF EXISTS post_search_update ON post;
DROP TRIGGER IF EXISTS new_liked_posts ON liked_posts;
DROP TRIGGER IF EXISTS new_follow ON following;
DROP TRIGGER IF EXISTS follow_himself ON following;
```

```sql
DROP FUNCTION IF EXISTS post_search_update;
DROP FUNCTION IF EXISTS new_liked_posts;
DROP FUNCTION IF EXISTS new_follow;
DROP FUNCTION IF EXISTS follow_himself;

-- Types

CREATE TYPE med AS ENUM ('NONE', 'GIF', 'VIDEO', 'NEWS ARTICLE', 'WEB
SITE');

-- Tables

-- Table named users instead of user because user is a reserved word in
PostgreSQL.
CREATE TABLE users (
    id SERIAL PRIMARY KEY
);

CREATE TABLE authenticated_users (
    email TEXT NOT NULL UNIQUE,
    username TEXT NOT NULL UNIQUE,
    reputation TEXT,
    password TEXT NOT NULL,
    user_id INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE administrator (
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    username TEXT NOT NULL,
    company_id SERIAL NOT NULL UNIQUE,
    phone_number VARCHAR NOT NULL UNIQUE,
    residence TEXT NOT NULL UNIQUE
);

CREATE TABLE moderator (
    phone_number VARCHAR NOT NULL UNIQUE,
    authenticated_user_id INT REFERENCES users(id) ON UPDATE CASCADE
);
```

```sql
CREATE TABLE tag (
    tag_id SERIAL PRIMARY KEY,
    name TEXT UNIQUE
);

CREATE TABLE post (
    post_id SERIAL PRIMARY KEY,
    author_id INT REFERENCES users(id) ON UPDATE CASCADE,
    title TEXT NOT NULL,
    date TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
    votes INT,
    description TEXT,
    tag_id INT REFERENCES tag(tag_id) ON UPDATE CASCADE,
    TYPE med
);
```

```sql
CREATE TABLE comment (
    comment_id SERIAL PRIMARY KEY,
    description TEXT,
    date TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
    votes INT,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    TYPE med
);

CREATE TABLE report_post (
    report_post_id SERIAL,
    description TEXT NOT NULL,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    user_id INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE report_comment (
    report_comment_id SERIAL,
    description TEXT NOT NULL,
    comment_id INT REFERENCES comment(comment_id) ON UPDATE CASCADE,
    user_id INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE liked_posts (
    user_id INT REFERENCES users(id) ON UPDATE CASCADE,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    down_or_upvote BIT NOT NULL
);

CREATE TABLE media (
    url VARCHAR UNIQUE NOT NULL,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    comment_id INT REFERENCES comment(comment_id) ON UPDATE CASCADE
);

CREATE TABLE visitor (
    visitor_id INT REFERENCES users(id) ON UPDATE CASCADE
);
```

```sql
CREATE TABLE following (
    authenticated_user_id1 INT REFERENCES users(id) ON UPDATE CASCADE,
    authenticated_user_id2 INT REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE notification (
    authenticated_user_id1 INT REFERENCES users(id) ON UPDATE CASCADE,
    authenticated_user_id2 INT REFERENCES users(id) ON UPDATE CASCADE,
    comment_id INT REFERENCES comment(comment_id) ON UPDATE CASCADE,
    post_id INT REFERENCES post(post_id) ON UPDATE CASCADE,
    date TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
    description TEXT NOT NULL
);
```

```sql
-- Indexes

CREATE INDEX comments_index ON comment USING hash (comment_id);

CREATE INDEX posts_index ON post USING btree (post_id);
CLUSTER post USING posts_index;

CREATE INDEX follows_index ON following USING hash
(authenticated_user_id1);

-- FTS Indexes

-- Add column to post to store computed ts_vectors.
ALTER TABLE post
ADD COLUMN tsvectors TSVECTOR;

-- Create a function to automatically update ts_vectors.
CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
        setweight(to_tsvector('english', NEW.title), 'A') ||
        setweight(to_tsvector('english', NEW.description), 'B')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
        IF (NEW.title <> OLD.title OR NEW.description <> OLD.description)
        THEN
        NEW.tsvectors = (
               setweight(to_tsvector('english', NEW.title), 'A') ||
            setweight(to_tsvector('english', NEW.description), 'B')
        );
        END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;
```

```sql
-- Create a trigger before insert or update on post.
CREATE TRIGGER post_search_update
 BEFORE INSERT OR UPDATE ON post
 FOR EACH ROW
 EXECUTE PROCEDURE post_search_update();

-- Finally, create a GIN index for ts_vectors.
CREATE INDEX search_idx ON post USING GIN (tsvectors);
```

```sql
-- Triggers

-- Trigger #1
-- An user cannot upvote/downvote twice the same post and cannot upvote
and downvote the same post.

CREATE FUNCTION new_liked_posts() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF EXISTS (SELECT * FROM liked_posts WHERE user_id = NEW.user_id
        AND post_id = NEW.post_id) THEN
           RAISE EXCEPTION 'You cannot upvote/downvote twice the same post
           and you cannot upvote and downvote the same post.';
        END IF;
        RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER new_liked_posts
BEFORE INSERT OR UPDATE ON liked_posts
FOR EACH ROW
EXECUTE PROCEDURE new_liked_posts();

-- Trigger #2
-- An user can only follow one time another user.

CREATE FUNCTION new_follow() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF EXISTS (SELECT * FROM following WHERE authenticated_user_id1 =
        NEW.authenticated_user_id1 AND authenticated_user_id2 =
        NEW.authenticated_user_id2) THEN
           RAISE EXCEPTION 'You cannot follow twice the same user.';
        END IF;
        RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

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

```sql
-- Trigger #3
-- An user can't follow himself.

CREATE FUNCTION follow_himself() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF (NEW.authenticated_user_id1 = NEW.authenticated_user_id2) THEN
           RAISE EXCEPTION 'You cannot follow yourself.';
        END IF;
        RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

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

A.2 Database population

```sql
INSERT INTO authenticated_users (email, username, reputation, password, user_id)
VALUES ('gharkess0@hc360.com', 'jmethringham0', 33, 'YRGC95', 1);
INSERT INTO authenticated_users (email, username, reputation, password, user_id)
VALUES ('jfladgate1@ameblo.jp', 'cdagg1', 88, 'xPUj1m45N', 2);
INSERT INTO authenticated_users (email, username, reputation, password, user_id)
VALUES ('fohartnedy2@taobao.com', 'ltemple2', 21, 'aqBx2alg7gT', 3);
INSERT INTO authenticated_users (email, username, reputation, password, user_id)
VALUES ('civashkin3@rakuten.co.jp', 'lcoakley3', 26, 'yT92tko', 4);
INSERT INTO authenticated_users (email, username, reputation, password, user_id)
VALUES ('jhawkswell4@sciencedaily.com', 'rhagley4', 27, 'x1pbLsz8', 5);
INSERT INTO authenticated_users (email, username, reputation, password, user_id)
VALUES ('apetricek5@cbsnews.com', 'aracine5', 16, '533V0GHvPGH', 6);

/*removed for brevity*/
```

Revision history

- [01-12-21] Added A4, A5, A6 to the EBD
- [01-12-21] Added revision history to the EBD
- [01-12-21] Added authors to the EBD

Authors

- Dogukan Olgun, up202102204, up202102204@edu.fe.up.pt
- Diogo Moreira, up201804904, up201804904@fc.up.pt (Editor)
- Diogo Gomes, up201805367, up201805367@edu.fc.up.pt
- Eduardo Ramos, up201906732, up201906732@edu.fe.up.pt