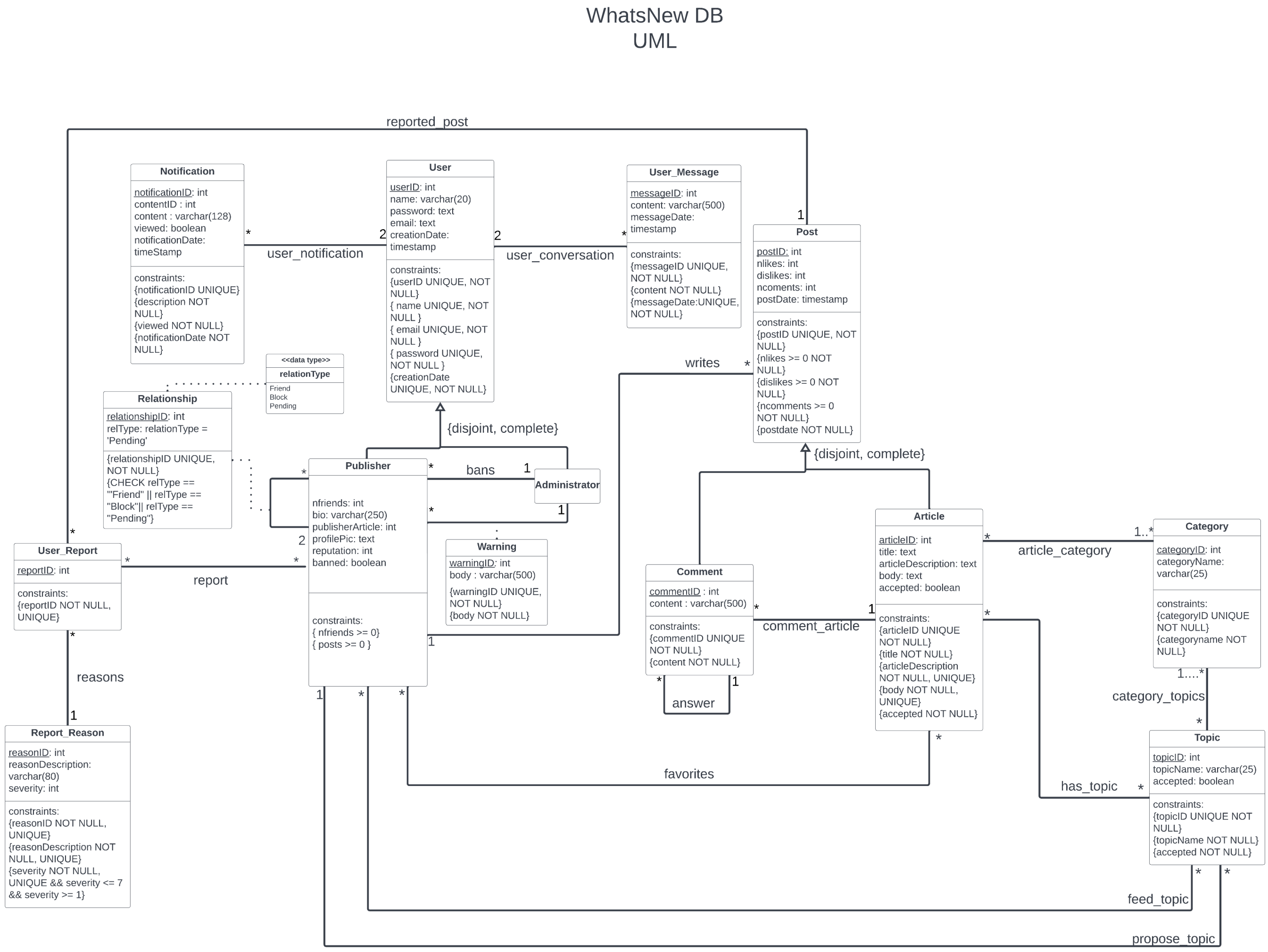
EBD: Database Specification Component

> Intended for any user registered at UP who requires a convenient and easy way of requesting and managing document printings.
> WhatsNew is a collaborative news website that enables the user to submit articles and stay connected with other publishers around the world.
> Unlike the equivalent websites, our product enables an easy and unified access to relevant articles from various topics, promoting an healthy space for debates and sharing information.

A4: Conceptual Data Model

> The presented UML below shows the organisation and structures of the various objects that make part of WhatsNew's system, as well as its attributes, domains, relations between them and their multiplicity. WhatsNew' Constraints and Business Rules are not represented in the UML but in a text box below the diagram.

1. Class Diagram



BR constraints:

{A user can post a comment in each 30 seconds.}

{A user cannot vote and/or comment. They can only vote and publish comments on comments in his/her post made by other users.}

{When a user account is deleted, all their posts will be maintained. However, it is not shared any personal information about them.}

{When a user posts an article or a comment, they can only delete it if it does not have any votes or comments.}

{When a user account is deleted, all their posts will be maintained. However, it is not shared any personal information about them.}

Fig.1 WhatsNewDB UML

2. Additional Business Rules

BR constraints (represented in the UML Diagram):

-A user can post a comment in each 30 seconds. -A user cannot vote and/or comment. They can only vote and publish comments on comments in his/her post made by other users -When a user account is deleted, all their posts will be maintained. However, it is not shared any personal information about them. -When a user posts an article or a comment, they can only delete it if it does not have any votes or comments. -When a user account is deleted, all their posts will be maintained. However, it is not shared any personal information about them.

A5: Relation Schema, validation and schema refinement

> We used the relation compact notation in order to map the classes, attributes and multiplicities of our UML in our relation schema.
> The schema validation is shown below and we also identified the functional dependencies. It was necessary to make any decomposition since they were already in BCNF.

1. Relation Schema

Relation Number	
R01	user(**userID**, name, password, email, creationDate)
R02	publisher(publisherID, userData->user, nfriends, bio, publisherArticle, reputation, profilePic, banned, bannedBy->administrator)
R03	administrator(**adminID**, **adminData**->user)
R04	notifications(**notificationID**, receiverID->user, senderID->user, notificationDescription, viewed, notificationDate)
R05	user_report(**reportID**, reported-> publisher, reporter-> publisher, reasonID -> report_reason report_reason)
R06	user_message(**messageID**, senderID->publisher, receiverID->publisher, content, messageDate)
R07	post(**postID**, userID->publisher, nLikes, nDislikes, nComments, postDate)
R08	comment(**commentID**, parentID-> comment, postID->post, articleID->article, content)
R09	article(**articleID**, postID->post, title, articleDescription, body, accepted)
R10	article_category(**artcatID**, articleID->article, categoryID->category)
R11	category(**categoryID**, categoryName)
R12	topic(**topicID**, publisherID->publisherID, topicName, accepted)
R13	category_topics(**cattopID**, topicID -> topic, categoryID -> category)
R14	relationship(**relationshipID**, publisher1ID-> publisher, publisher2ID-> publisher, relType)
R15	feed_topic(**feedtopID**, publisherID-> publisher, topicID-> topic)
R16	favorites(**favoriteID**, publisherID -> publisher, postID ->article)
R17	warning(**warningID**, publisherID-> publisher, adminID-> admin, body)
R18	has_topic(**hastopID**, topicID->topic, articleID->article)
R19	report_reason(**reasonID**, reasonDescription, severity)

Note: all words in **bold** are NOT NULL (NN) UNIQUE KEYS (UK)

2. Domains

Domain	Column 2
post.nlikes	INT >= 0
post.ndislikes	INT >= 0
post.ncomments	INT >= 0
report_reasons.severity	INT >= 1 && <= 7
user.name	VARCHAR(20)
user.nfriends	INT >= 0
publisher.bio	VARCHAR(250)
publisher.publisherPosts	INT >= 0
publisher.reputation	INT = post.nlikes - post.ndislikes
report_reasons.reasonDescription	VARCHAR(80)
user_message.content	VARCHAR(500)
notifications.content	VARCHAR(128)
relationship.relationType	relationType == '"Friend" OR relationType == "Block" OR relationType == "Pending"

Note: We use "OR" instead of || because, in markdown, the | icon is considered "end of row", so the rest of the row would not appear.

3. Schema Validation

TABLE R01	**user**
keys	{userID}
Functional Dependencies FD0101	{userID} → {name, password, email, creationDate, nfriends}
Normal Form	BCNF

TABLE R02	**publisher**
keys	{publisherID}
Functional Dependencies FD0201	{userID} → {publisherID, name, password, email, creationDate, nfriends, bio, publisherArticle, reputation, profilePic, banned, bannedBy->administrator}
Normal Form	BCNF

TABLE R03	**administrator**
keys	{userID}
Functional Dependencies FD0301	{adminID} → {userID->user, name, password, email, creationDate}

TABLE R03	**administrator**
Normal Form	BCNF

TABLE R04	**notification**
keys	{notificationID}
Functional Dependencies FD0401	{notificationID} → {receiverID->user, senderID->user, notificationDescription, viewed, notificationDate}
Normal Form	BCNF

TABLE R05	**user_report**
keys	{reportID}
Functional Dependencies FD0501	{reportID} →{postID->post, reported-> publisher, reporter-> publisher, reasonID -> report_reason}
Normal Form	BCNF

TABLE R06	**user_message**
keys	{messageID}
Functional Dependencies FD0601	{messageID} → {senderID->publisher, receiverID->publisher content, messageDate}
Normal Form	BCNF

TABLE R07	**post**
keys	{postID}
Functional Dependencies FD0701	{reportID}-> {userID->publisher, nLikes, nDislikes, nComments, postDate}
Normal Form	BCNF

TABLE R08	**comment**
keys	{commentID}
Functional Dependencies FD0801	{commentID} → {postID->article,, parentID->comment, content}
Normal Form	BCNF

TABLE R09	**article**
keys	{articleID}
Functional Dependencies FD0901	{articleID} -> {postID, title, articleDescription, body, accepted}
Normal Form	BCNF

TABLE R10	**article_category**
keys	{artcatID}

TABLE R10	**article_category**
Functional Dependencies FD1001	{artcatID} -> {articleID, categoryID}
Normal Form	BCNF

TABLE R11	**category**
keys	{categoryID}
Functional Dependencies FD1101	{categoryID} -> {categoryName}
Normal Form	BCNF

TABLE R12	**topic**
keys	{topicID}
Functional Dependencies FD1201	{topicID} -> {publisherID, topicName, accepted}
Normal Form	BCNF

TABLE R13	**category_topics**
keys	{cattopID}
Functional Dependencies FD1301	{{cattopID} -> {categoryID, topicID}
Normal Form	BCNF

TABLE R14	**relationship**
keys	{relationshipID}
Functional Dependencies FD1401	{relationshipID} -> {publisher1ID-> publisher, publisher2ID-> publisher, relType}
Normal Form	BCNF

TABLE R15	**feed_topic**
keys	{feedtopID}
Functional Dependencies FD1501	{feedtopID} -> {publisherID-> publisher, topicID-> topic}
Normal Form	BCNF

TABLE R16	**favorites**
keys	{favoriteID}
Functional Dependencies FD1601	{favoriteID} -> {publisherID -> publisher, postID ->article}
Normal Form	BCNF

TABLE R17	**warning**
keys	{warningID}
Functional Dependencies FD1701	warningID} -> {publisherID-> publisher, adminID-> admin, body}

TABLE R17	**warning**
Normal Form	BCNF

TABLE R18	**has_topic**
keys	{hastopID}
Functional Dependencies FD1801	{hastopID} -> {topicID->topic, articleID->article}
Normal Form	BCNF

TABLE R19	**report_reasons**
keys	{reasonID}
Functional Dependencies FD1901	{reasonID} -> {reasonDescription, severity}
Normal Form	BCNF

> **Justification**

We chose the E/R-style mapping to model generalizations because, after some deliberation, we concluded that the application of this model would allow us to prevent some problems and simplify the implementation down the line. Not only does it make the child tables less verbose, it allows us to implement the deletion of personal data with ease – all we need to do is delete the information in the users table. In our particular model, this choice makes the model more flexible and robust, allowing a database modelling more similar to an OOP implementation.

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

A6: Indexes, triggers, transactions and database population

> The database workload shows our estimation of the number of rows in each table, as well as its expected growth.
> We also created 3 indexes in order to improve our DB performance and make more efficient accesses to information, since it has several tables with a huge amount of data.
> Finally, we developed some triggers and transactions so that the integrity of data was maintained.

1. Database workload

> Workload presents our estimate of the number of rows in each table, as well as an estimate of its growth.

Relation	**Relation Name**	**Order of magnitude**	**Estimated Growth**
R01	users	10 k (tens of thousands)	100 (hundreds) / day
R02	publisher	10 K	100 / day
R03	administrator	50 (units)	2 / year
R04	notifications	250 K (hundreds of thousands)	30.1 K / day

Relation	Relation Name	Order of magnitude	Estimated Growth
R05	user_report	50 K	100 K / day
R06	user_message	100 K	1 K (thousands) / day
R07	post	240 K	1.1 K / day
R08	comment	200 K	1 K / day
R09	article	40 K	100 / day
R10	article_category	80 K	200 / day
R11	category	10	0 (zero) / day
R12	topic	1 K	10 / day
R13	category_topics	30 K	30 K / day
R14	relationship	2.5 M (millions)	1.5 K / day
R15	feed_topic	50 K	500 / day
R16	favorites	100 K	10 / day
R17	warning	1 K	3 / day
R18	has_topic	60 K	100 / day
R19	report_reason	5	0 / day

2. Indexes

2.1 Performance Indexes

> We selected these 3 indexes to improve the performance of our database, since they belong to tables with a large amount of data, thus making access to them more efficient.

Index	**IDX01**
Index relation	article
Index attribute	postDate
Index type	B-tree
Cardinality	High
Clustering	No
Justification	In order to display the top articles in the feed of our website, a B-tree is the best option the retrieve the most recent and / or the most popular ones, since it maintains the articles sorted. There is no need for clustering since the update frequency can be high.

SQL Code IDX01

```
CREATE INDEX top_article ON post USING btree(postDate);
```

Index	IDX02
Index relation	article_has_topic
Index attribute	postID
Index type	Hash
Cardinality	High
Clustering	No
Justification	To be able to show the news related to the topic, the best option is to use a hash table, since we use the topic id that gives us the correspondence between the elements of the two tables.

SQL Code IDX02

```
CREATE INDEX article_topic ON has_topic USING hash(articleID);
```

2.2 Full-text Search Indexes

> In order to be able to do full-text search, indexes were created based on the match of pre-defined attributes.

Index	IDX03
Index relation	article
Index attribute	title, body
Index type	GIN
Clustering	No
Justification	To provide full-text search features to look for articles based on matching titles or body.

SQL Code IDX03

```

ALTER TABLE article
ADD COLUMN tsvectors TSVECTOR;

```

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

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


CREATE INDEX article_search_idx ON article USING GIN (tsvectors);
```

Index	**IDX04**
Index relation	publisher
Index attribute	nome
Index type	GIN
Clustering	No
Justification	To provide full-text search features to look for publisher based on matching names.

SQL Code IDX04

```


ALTER TABLE publisher
ADD COLUMN tsvectors TSVECTOR;


CREATE OR REPLACE FUNCTION publisher_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.nome), 'A')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
        IF (NEW.nome <> OLD.nome) THEN
          NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.nome), 'A')
          );
        END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;


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


CREATE INDEX publisher_search_idx ON publisher USING GIN (tsvectors);
```

Index	IDX05
Index relation	category
Index attribute	categoryName
Index type	GIN
Clustering	No
Justification	To provide full-text search features to look for categories based on matching names (categoryName).

SQL Code IDX05

```
ALTER TABLE category
ADD COLUMN tsvectors TSVECTOR;

CREATE OR REPLACE FUNCTION category_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.categoryName), 'A')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
        IF (NEW.categoryName <> OLD.categoryName) THEN
          NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.categoryName), 'A')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

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

CREATE INDEX category_search_idx ON category USING GIN (tsvectors);
```

Index	**IDX06**
Index relation	topic
Index attribute	topicName
Index type	GIN
Clustering	No
Justification	To provide full-text search features to look for topics based on matching names (topicName).

SQL Code IDX06

```
ALTER TABLE topic
ADD COLUMN tsvectors TSVECTOR;


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

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


CREATE INDEX topic_search_idx ON topic USING GIN (tsvectors);
```

3. Triggers

> We created these triggers in so that we had more control structures to ensure the DB integrity.

Triggers	TRIGGER01
Description	A user cannot comment an article that themselves have published.

SQL Code TRIGGER01

```
CREATE OR REPLACE FUNCTION not_comment() RETURNS TRIGGER AS
$BODY$

BEGIN
       IF EXISTS (SELECT *
        FROM
        (SELECT publisherId FROM post WHERE NEW.postid = post.postid) AS
comment_userID,
        (SELECT publisherId FROM article INNER JOIN post USING (postID) WHERE
NEW.articleID = article.articleID) AS article_userID
        WHERE comment_userID.publisherID = article_userID.publisherID) THEN
       RAISE EXCEPTION 'A user cannot comment an article that themselves have
published';
       END IF;
       RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

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

Triggers	TRIGGER02
Description	When a user posts an article or a comment, they can only delete it if it does not have any votes or comments.

SQL CODE TRIGGER02

```
CREATE OR REPLACE FUNCTION mantain_votes_and_comments() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF EXISTS (SELECT * FROM post WHERE post.postId = OLD.postId AND
            (post.nLikes > 0 OR post.nDislikes > 0 OR post.nComments > 0)) THEN
           RAISE EXCEPTION 'When a user posts an article or a comment, they can only delete it if it does not have any votes or comments.';
        END IF;
        RETURN OLD;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER mantain_votes_and_comments
        BEFORE DELETE ON post
        FOR EACH ROW
        EXECUTE PROCEDURE mantain_votes_and_comments();
```

4. Transactions

> We use the transaction to ensure the integrity of information when multiple operations are being performed.

Transaction	**TRAN01**
Description	When a user account is deleted or banned, the data is maintained.
Justification	As it is specified, a user can delete its account; however, his/her posts will be kept in the database.
Isolation Level	SERIALIZABLE READ ONLY

SQL Code

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

UPDATE publisher SET userID = NULL
    WHERE banned = 'True';


UPDATE post SET publisherID = 1
    WHERE publisherID = NULL;

END TRANSACTION;
```

Annex A. SQL Code

A.1 Database Schema

> Here is the code of the our population schema. The transactions is commented because it has an error we couldn't resolve in time. However, we keep it on the file since it will be used on our website.

```
DROP TABLE IF EXISTS has_topic;
DROP TABLE IF EXISTS warning;
DROP TABLE IF EXISTS favorites;
DROP TABLE IF EXISTS feed_topic;
DROP TABLE IF EXISTS relationship;
DROP TABLE IF EXISTS category_topics;
DROP TABLE IF EXISTS topic;
DROP TABLE IF EXISTS article_category;
DROP TABLE IF EXISTS category;
DROP TABLE IF EXISTS comment;
DROP TABLE IF EXISTS article;
DROP TABLE IF EXISTS user_message;
DROP TABLE IF EXISTS user_report;
DROP TABLE IF EXISTS report_reason;
DROP TABLE IF EXISTS post;
DROP TABLE IF EXISTS notifications;
DROP TABLE IF EXISTS administrator;
DROP TABLE IF EXISTS publisher;
DROP TABLE IF EXISTS users;

DROP INDEX IF EXISTS top_article;
DROP INDEX IF EXISTS article_topic;


DROP TRIGGER IF EXISTS not_comment ON comment;
DROP TRIGGER IF EXISTS category_search_update ON article;
DROP TRIGGER IF EXISTS publisher_search_update ON article;
DROP TRIGGER IF EXISTS article_search_update ON article;
DROP TRIGGER IF EXISTS notify ON user_message;
DROP TRIGGER IF EXISTS wait_before_comment ON write_comment;
DROP TRIGGER IF EXISTS not_comment ON write_comment;

------------
-- TYPES
------------
DROP TYPE IF EXISTS relationType;

CREATE TYPE relationType AS ENUM ('Friend', 'Block', 'Pending');

------------
-- TABLES
------------

CREATE TABLE users(
    userID SERIAL PRIMARY KEY,
    nome VARCHAR(20) NOT NULL,
    email TEXT NOT NULL,
    publisherPassword TEXT NOT NULL,
    creationDate TIMESTAMP NOT NULL
);

CREATE TABLE publisher(
    publisherID SERIAL PRIMARY KEY,
    userID INTEGER REFERENCES users NOT NULL,
    nFriends INTEGER,
    profilePic TEXT,
    bio VARCHAR(250),
    publisherArticles INTEGER,
    reputation INTEGER,
    banned BOOLEAN NOT NULL,

    CONSTRAINT nfriends_positive_ck
    CHECK (nfriends >= 0), CONSTRAINT nposts_ck CHECK (publisherArticles >= 0)

);

create table administrator(
    adminID SERIAL PRIMARY KEY,
    userID INTEGER REFERENCES users NOT NULL
);

CREATE TABLE notifications(
    notificationID SERIAL PRIMARY KEY,
    receiverID INTEGER REFERENCES users NOT NULL,
    senderID INTEGER REFERENCES users NOT NULL,
    contentID INTEGER NOT NULL,
    content VARCHAR(128) NOT NULL,
    viewed BOOLEAN NOT NULL,
    notificationDate TIMESTAMP NOT NULL
);

CREATE TABLE report_reason(
    reportID SERIAL PRIMARY KEY,
    reasonDescripton VARCHAR(80) NOT NULL UNIQUE,
    severity INTEGER,

    CONSTRAINT severity_ck
    CHECK (severity >= 1 AND severity <= 7)
);

CREATE TABLE post(
    postID SERIAL PRIMARY KEY,
    publisherID INTEGER REFERENCES publisher NOT NULL,
    nLikes INTEGER NOT NULL,
    nDislikes INTEGER NOT NULL,
    nComments INTEGER NOT NULL,
    postDate TIMESTAMP NOT NULL,

    CONSTRAINT n_likes_ck
    CHECK (nlikes >= 0),
    CONSTRAINT n_dislikes_ck
    CHECK (nDislikes >=0),

    CONSTRAINT n_comments_ck
    CHECK (nComments >= 0)
);

CREATE TABLE user_report(
    reportID SERIAL PRIMARY KEY,
    postID INTEGER REFERENCES post NOT NULL,
    reported INTEGER REFERENCES publisher NOT NULL,
    reporter INTEGER REFERENCES publisher NOT NULL,
    reasonID INTEGER REFERENCES report_reason NOT NULL
);

CREATE TABLE user_message(
    messageID SERIAL PRIMARY KEY,
    senderID INTEGER REFERENCES users NOT NULL,
    receiverID INTEGER REFERENCES users NOT NULL,
    content VARCHAR(500) NOT NULL,
    messageDate TIMESTAMP NOT NULL
);

CREATE TABLE article(
    articleID SERIAL PRIMARY KEY,
    postID INTEGER REFERENCES post NOT NULL,
    title TEXT NOT NULL,
    articleDescription TEXT NOT NULL UNIQUE,
    body TEXT NOT NULL UNIQUE,
    accepted BOOLEAN NOT NULL
);

CREATE TABLE comment(
    commentID SERIAL PRIMARY KEY,
    postID INTEGER REFERENCES post NOT NULL,
    parentID INTEGER REFERENCES comment,
    articleID INTEGER REFERENCES article NOT NULL,
    content VARCHAR(500)
);

CREATE TABLE category(
    categoryID SERIAL PRIMARY KEY,
    categoryName VARCHAR(25) NOT NULL
);

CREATE TABLE article_category(
    artcatID SERIAL PRIMARY KEY,
    articleID INTEGER REFERENCES article NOT NULL,
    categoryID INTEGER REFERENCES category NOT NULL
);

CREATE TABLE topic(
    topicID SERIAL PRIMARY KEY,
    publisherID INTEGER REFERENCES publisher NOT NULL,
    topicName VARCHAR(25) NOT NULL,
    accepted BOOLEAN NOT NULL
);

CREATE TABLE category_topics(
    cattopID SERIAL PRIMARY KEY,
    topicID INTEGER REFERENCES topic NOT NULL,
    categoryID INTEGER REFERENCES category NOT NULL
);

CREATE TABLE relationship(
    relationshipID SERIAL PRIMARY KEY,
    publisher1ID INTEGER REFERENCES publisher NOT NULL,
    publisher2ID INTEGER REFERENCES publisher NOT NULL,
    relType relationType NOT NULL DEFAULT 'Pending'
);

CREATE TABLE feed_topic(
    feedtopID SERIAL PRIMARY KEY,
    publisherID INTEGER REFERENCES publisher NOT NULL,
    topicID INTEGER REFERENCES topic NOT NULL
);

CREATE TABLE favorites(
    favoriteID SERIAL PRIMARY KEY,
    publisherID INTEGER REFERENCES publisher NOT NULL,
    postID INTEGER REFERENCES post NOT NULL
);

CREATE TABLE warning(
    warningID SERIAL PRIMARY KEY,
    publisherID INTEGER REFERENCES publisher NOT NULL,
    adminID INTEGER REFERENCES administrator NOT NULL,
    body VARCHAR(500) NOT NULL
);

CREATE TABLE has_topic(
    hastopID SERIAL PRIMARY KEY,
    topicID INTEGER REFERENCES topic NOT NULL,
    articleID INTEGER REFERENCES article NOT NULL
);


------------
-- INDEXES
------------

--INDEX 01

CREATE INDEX top_article ON post USING btree(postDate);

--INDEX 02

CREATE INDEX article_topic ON has_topic USING hash(articleID);
```

```
--FULL-TEXT SEARCH INDEX 03

ALTER TABLE article
ADD COLUMN tsvectors TSVECTOR;

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

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


CREATE INDEX article_search_idx ON article USING GIN (tsvectors);

--FULL-TEXT SEARCH INDEX 04


ALTER TABLE publisher
ADD COLUMN tsvectors TSVECTOR;


CREATE OR REPLACE FUNCTION publisher_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.nome), 'A')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.nome <> OLD.nome) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.nome), 'A')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;


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


CREATE INDEX publisher_search_idx ON publisher USING GIN (tsvectors);


--FULL-TEXT SEARCH INDEX 05

ALTER TABLE category
ADD COLUMN tsvectors TSVECTOR;

CREATE OR REPLACE FUNCTION category_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.categoryName), 'A')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.categoryName <> OLD.categoryName) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.categoryName), 'A')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

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

CREATE INDEX category_search_idx ON category USING GIN (tsvectors);

--FULL-TEXT SEARCH INDEX 06


ALTER TABLE topic
ADD COLUMN tsvectors TSVECTOR;


CREATE OR REPLACE FUNCTION topic_search_update() RETURNS TRIGGER AS $$
BEGIN
```

```
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.topicName), 'A')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.topicName <> OLD.topicName) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.topicName, 'A')
           ));
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;


CREATE TRIGGER topic_search_update
 BEFORE INSERT OR UPDATE ON topic
 FOR EACH ROW
 EXECUTE PROCEDURE topic_search_update();


CREATE INDEX topic_search_idx ON topic USING GIN (tsvectors);



----------------
-- TRIGGERS
----------------

-- TRIGGER 01

CREATE OR REPLACE FUNCTION not_comment() RETURNS TRIGGER AS
$BODY$

BEGIN
       IF EXISTS (SELECT *
        FROM
        (SELECT publisherId FROM post WHERE NEW.postid = post.postid) AS
comment_userID,
        (SELECT publisherId FROM article INNER JOIN post USING (postID) WHERE
NEW.articleID = article.articleID) AS article_userID
        WHERE comment_userID.publisherID = article_userID.publisherID) THEN
       RAISE EXCEPTION 'A user cannot comment an article that themselves have
published';
       END IF;
       RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

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


-- TRIGGER 02

CREATE OR REPLACE FUNCTION mantain_votes_and_comments() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF EXISTS (SELECT * FROM post WHERE post.postId = OLD.postId AND
            (post.nLikes > 0 OR post.nDislikes > 0 OR post.nComments > 0)) THEN
           RAISE EXCEPTION 'When a user posts an article or a comment, they can only delete it if it does not have any votes or comments.';
        END IF;
        RETURN OLD;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER mantain_votes_and_comments
        BEFORE DELETE ON post
        FOR EACH ROW
        EXECUTE PROCEDURE mantain_votes_and_comments();

---------------
--TRANSACTION
---------------

BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

UPDATE publisher SET userID = NULL
    WHERE banned = 'True';


UPDATE post SET publisherID = 1
    WHERE publisherID = NULL;

END TRANSACTION;
```

In the transactions section we commented the code we developed, since we were having an error that we couldn't fix. However, we decided to leave it in the report to be able to understand what it will be used for in our system.

A.2 Database Population

> We created some C++ scripts in order to populate all the tables automatically. Here is the link for GitLab for those scripts. https://git.fe.up.pt/lbaw/lbaw2223/lbaw2231/-/blob/main/populate.sql

Revision History

Changes made to the second submission:

1. 10/10/2022- Added content relative of A4.
2. 17/10/2022- Added content relative of A5.
3. 23/10/2022- Added content relative of A6 and database schema and population.
4. 31/10/2022- Reuploaded content relative to A4, A5 and A6.

GROUP2231, 23/10/2022

- Group member 1: Inês Sá Pereira Estêvão Gaspar, up202007210@edu.fe.up.pt (Editor)
- Group member 2: Maria Sofia Brandão Porto Carvalho Gonçalves, up202006927@edu.fe.up.pt (Editor)
- Group member 3: Pedro Fardilha Barbeira, up201303693@edu.fe.up.pt (Editor)
- Group member 4: Pedro Pereira Ferreira, up202004986@edu.fe.up.pt (Editor)