ebd

Last edited by **Paulo Miguel de Jesus Coutinho dos Santos Fidalgo** 3 months ago

EBD: Database Specification Component

The vision of this project is to create a social network for food enthusiasts, where they can discover, share, and engage with dining experiences worldwide. This platform will offer in-depth reviews, personalized recommendations, and community interactions, empowering food lovers to make informed dining choices and explore new culinary horizons.

A4: Conceptual Data Model

The goal of this artifact is to provide a structured representation of the project's core entities and relationships that are essential for building the database. Using UML (Unified Modeling Language), we have created a Conceptual Data Model that visually documents all relevant entities, their attributes, and associations within the domain. This model is captured through a UML class diagram, which illustrates the relationships, roles, and multiplicity between entities, serving as a guide for the database design.

1. Class diagram

The UML diagram below illustrates the primary organizational entities of the Raffia platform, including the relationships between them, their attributes and domains, and the multiplicity of these relationships.

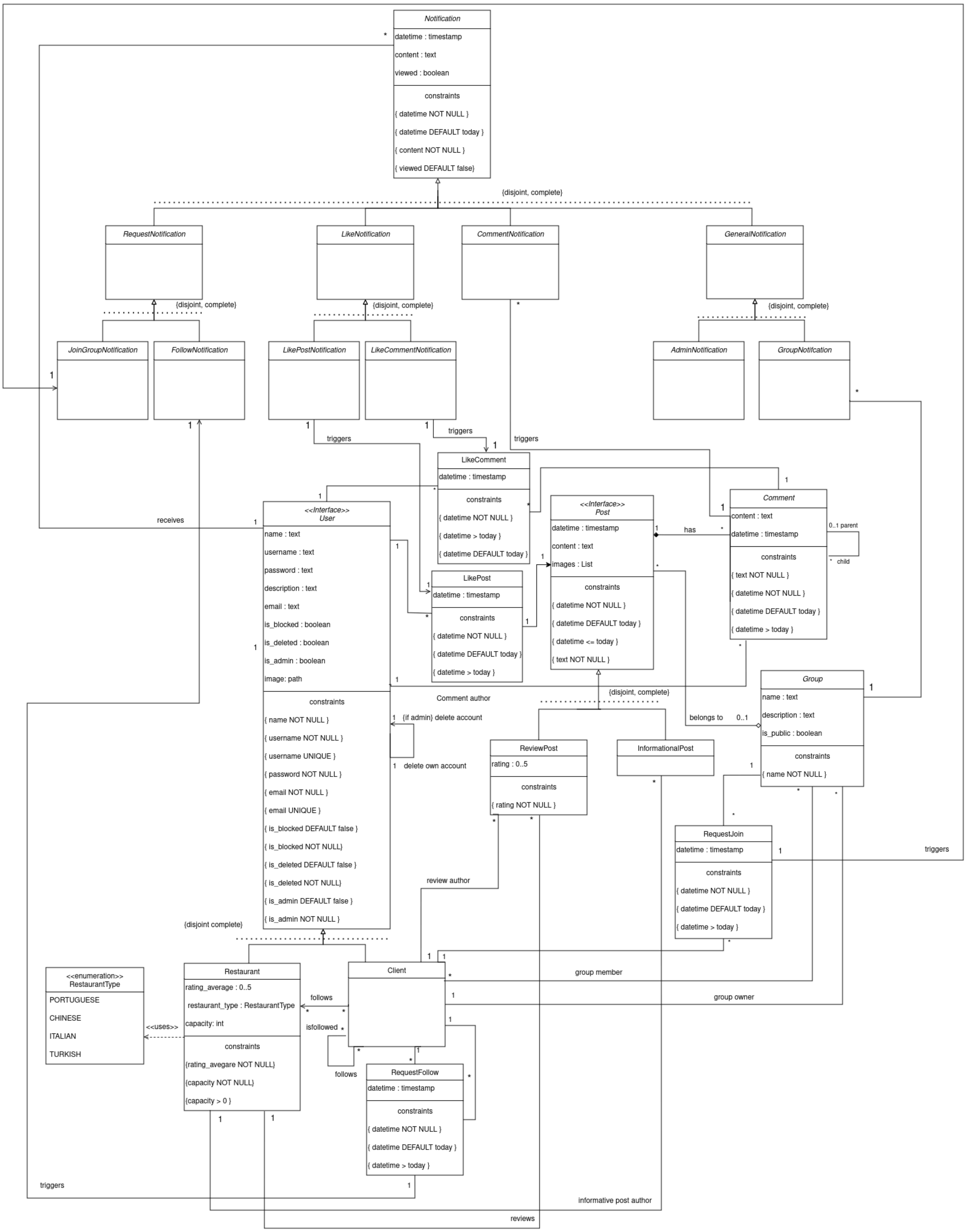

Figure 5: Raffia class diagram

2. Additional Business Rules

Additional business rules that cannot be represented in the UML class diagram are listed in the following table.

ID	Name	Description
BR07	Self follow	Users cannot follow themselves

ID	Name	Description
BR08	Request Join Group	User cannot join a group they already belong to
BR09	Join group	Users cannot request to join a group they are already in
BR10	Like Comment	Users can only like a comment once
BR11	Like Post	Users can only like a post once
BR12	Post on Group	Users can only post to a group they belong to
BR13	Request Follow	Users cannot request to follow other users they already follow
BR14	Follow Clients	Users cannot follow other clients they already follow
BR15	Follow Restaurants	Users cannot follow restaurants they already follow
BR16	Like in Groups	A user cannot like on posts in groups they do not belong to
BR17	Comment in Groups	A user cannot comment on posts in groups they do not belong to
BR18	Group Member	A group owner is also a member of the group
BR19	Delete Group Notification	Delete join group requests after acceptance
BR20	Delete Follow Request	Delete follow requests after acceptance

Table 16: Raffia additional business rules

A5: Relational Schema, validation and schema refinement

This section explains the relational schema created from the conceptual model. It lists attributes, domains, primary and foreign keys, and integrity constraints like unique, default, not null, and check.

1. Relational Schema

Relation reference	Relation Compact Notation
R01	user(id, name **NN**, username **UK NN**, password **NN**, description, email **UK NN**, image **NN**, is_blocked **NN DF** FALSE, is_admin **NN DF** FALSE, is_deleted **NN DF** FALSE)
R02	restaurant(id -> user, rating_average, type **NN**, capacity **NN CK** capacity > 0)
R03	client(id -> user)
R04	post(id, datetime **NN DF** today, content **NN**, images)
R05	review_post(id -> post, rating **NN CK** rating >= 0 AND rating <= 5, client_id -> client, group_id -> group)
R06	informational_post(id -> post, restaurant_id -> restaurant)
R07	comment (id, content **NN**, datetime **NN DF** today, post_id -> post, user_id -> User)
R08	group(id, name **NN**, description, is_public **NN**, owner_id -> client)
R09	notification(id, datetime **NN DF** today, content **NN**, viewed **NN DF** FALSE, user_id -> user)
R10	request_notification(id -> notification)
R11	like_notification(id -> notification)
R12	comment_notification(id -> notification, comment_id -> comment)
R13	general_notification(id -> notification)
R14	join_group_notification(id -> request_notification, client_id -> request_join, group_id -> request_join)
R15	follow_notification(id -> request_notification, requester_client_id -> request_follow, receiver_client_id -> request_follow)

Relation reference	Relation Compact Notation
R16	like_post_notification(id -> like_notification, user_id -> like_post, post_id -> like_post)
R17	like_comment_notification(id -> like_notification, user_id -> like_comment, comment_id -> like_comment)
R18	admin_notification(id -> general_notification)
R19	group_notification(id -> general_Notification, group_id -> group)
R20	like_post(user_id -> user, post_id -> post, datetime **NN DF** today)
R21	like_comment(user_id -> user, comment_id -> comment, datetime **NN DF** today)
R22	follows_restaurant(client_id -> client, restaurant_id -> restaurant)
R23	follows_client(sender_client_id -> client, followed_client_id -> client)
R24	comment_relationship(child -> comment, parent -> comment)
R25	group_member(client_id -> client, group_id -> group)
R26	request_follow(requester_client_id -> client, receiver_client_id -> client, datetime **NN DF** today)
R27	request_join(client_id -> client, group_id -> group, datetime **NN DF** today)

Table 17: Raffia relational schema

Legend:

- **UK** : Unique Key
- **CK** : Check
- **NN** : Not Null
- **DF** : Default

2. Domains

Specification of additional domains.

Domain Name	Domain Specification
Today	DATE DEFAULT CURRENT_DATE
Type	ENUM ('American', 'Chinese', 'Italian', 'Japanese', 'Mexican', 'Thai', 'Portuguese', 'Burger', 'Pizza', 'Indian', 'French', 'Greek', 'Spanish', 'Korean', 'Vietnamese', 'Lebanese', 'Turkish', 'Brazilian', 'Argentinian', 'Caribbean', 'Mediterranean', 'Moroccan', 'Ethiopian', 'German', 'Russian', 'Cuban', 'Peruvian', 'Filipino', 'Malaysian', 'Indonesian', 'Hawaiian', 'Vegan', 'Vegetarian', 'Seafood', 'Steakhouse', 'BBQ', 'Fast Food', 'Diner', 'Cafe', 'Bakery', 'Dessert', 'Sushi', 'Tapas', 'Middle Eastern', 'Fusion', 'Gluten-Free', 'Organic', 'Farm-to-Table', 'Other')

3. Schema validation

All functional dependencies have been identified, and the normalization of all relational schemas has been completed.

TABLE R01	user
Keys	{ id }, { email }, {username}
Functional Dependencies:	
FD0101	id -> { name, username, password, description, email, image, is_blocked, is_admin, is_deleted }
FD0102	email -> { id, name, username, password, description, image, is_blocked, is_admin, is_deleted }
FD0103	username -> { id, name, email, password, description, image, is_blocked, is_admin, is_deleted }
NORMAL FORM	BCNF

Table 18: user schema validation

TABLE R02	restaurant
Keys	{ id }
Functional Dependencies:	
FD0201	id -> { rating_average, type, capacity }
NORMAL FORM	BCNF

Table 19: restaaurant schema validation

TABLE R03	client
Keys	{ id }
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 20: client schema validation

TABLE R04	post
Keys	{ id }
Functional Dependencies:	
FD0401	id -> { datetime, content, images}
NORMAL FORM	BCNF

Table 21: post schema validation

TABLE R05	review_post
Keys	{ id }
Functional Dependencies:	
FD0501	id -> { rating, client_id, group_id }
NORMAL FORM	BCNF

Table 22: review_post schema validation

TABLE R06	informational_post
Keys	{ id }
Functional Dependencies:	
FD0601	id -> { restaurant_id }
NORMAL FORM	BCNF

Table 23: informational_post schema validation

TABLE R07	comment
Keys	{ id }
Functional Dependencies:	
FD0701	id -> { content, datetime, post_id, user_id }
NORMAL FORM	BCNF

Table 24: comment schema validation

TABLE R08	group
Keys	{ id }
Functional Dependencies:	
FD0801	id -> { name, description, is_public, owner_id }
NORMAL FORM	BCNF

Table 25: group schema validation

TABLE R09	notification
Keys	{ id }
Functional Dependencies:	
FD0901	id -> { datetime, content, viewed, user_id }
NORMAL FORM	BCNF

Table 26: notification schema validation

TABLE R10	request_notification
Keys	{ id }
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 27: request_notification schema validation

TABLE R11	like_notification
Keys	{ id }
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 28: like_notification schema validation

TABLE R12	comment_notification
Keys	{ id }
Functional Dependencies:	
FD01201	id -> { comment_id }
NORMAL FORM	BCNF

Table 29: comment_notification schema validation

TABLE R13	general_notification
Keys	{ id }
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 30: general_notification schema validation

TABLE R14	join_group_notification
Keys	{ id }
Functional Dependencies:	
FD01401	id -> { client_id, group_id }
NORMAL FORM	BCNF

Table 31: join_group_notification schema validation

TABLE R15	follow_notification
Keys	{ id }
Functional Dependencies:	
FD01501	id -> { sender_client_id, receiver_client_id }
NORMAL FORM	BCNF

Table 32: follow_notification schema validation

TABLE R16	like_post_notification
Keys	{ id }
Functional Dependencies:	
FD01601	id -> { user_id, post_id }
NORMAL FORM	BCNF

Table 33: like_post_notification schema validation

TABLE R17	like_comment_notification
Keys	{ id }
Functional Dependencies:	
FD01701	id -> { user_id, comment_id }
NORMAL FORM	BCNF

Table 34: like_comment_notification schema validation

TABLE R18	admin_notification
Keys	{ id }
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 35: admin_notfication schema validation

TABLE R19	group_notification
Keys	{ id }
Functional Dependencies:	
FD01901	id -> { group_id }
NORMAL FORM	BCNF

Table 36: group_notification schema validation

TABLE R20	like_post
Keys	{ user_id, post_id }
Functional Dependencies:	user_id, post_id -> { datetime }
NORMAL FORM	BCNF

Table 37: like_post schema validation

TABLE R21	like_comment
Keys	{ user_id, comment_id }
Functional Dependencies:	
FD02101	user_id, comment_id -> { datetime }
NORMAL FORM	BCNF

Table 38: like_comment schema validation

TABLE R22	follows_restaurant
Keys	{ client_id, restaurant_id }
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 39: follows_restaurant schema validation

TABLE R23	follows_client
Keys	{ sender_client_id, followed_client_id }
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 40: follows_client schema validation

TABLE R24	comment_relationship
Keys	{ child }, {parent}
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 41: comment_relashionship schema validation

TABLE R25	group_member
Keys	{ client_id, group_id }
Functional Dependencies:	none
NORMAL FORM	BCNF

Table 42: group_member schema validation

TABLE R26	request_follow
Keys	{ requester_client_id, receiver_client_id }

TABLE R26	request_follow
Functional Dependencies:	
FD02601	requester_client_id, receiver_client_id -> { datetime }
NORMAL FORM	BCNF

Table 43: request_follow schema validation

TABLE R27	request_join
Keys	{ client_id, group_id }
FD02701	client_id, group_id -> { datetime }
NORMAL FORM	BCNF

Table 44: request_join schema validation

Since all relations are in Boyce-Codd Normal Form (BCNF), the entire relational schema is also in BCNF. Consequently, there is no need for further normalization of the schema.

A6: Indexes, triggers, transactions and database population

This section includes analyzing the database workload, implementing performance indexes and full-text search indicators, using triggers to maintain data integrity, defining transactions for data accuracy, and establishing the database schema and populating it.

1. Database Workload

To create a well-structured database, it is crucial to understand how frequently a table will be accessed and how it is expected to grow. The table below outlines these predictions:

Identifier	Relation Name	Order of Magnitude	Estimated Growth
RS01	user	10 k	10 / day
RS02	restaurant	1 k	1 / day
RS03	client	10 k	10 / day
RS04	post	100 k	1 k / day
RS05	review_post	1 k	10 / day
RS06	informational_post	1 k	10 / day
RS07	comment	10 k	100 / day
RS08	group	1 k	10 / day
RS09	notification	10 k	1 k / day
RS10	request_notification	1 k	10 / day
RS11	like_notification	10 k	1 k / day
RS12	comment_notification	10 k	100 / day
RS13	general_notification	1 k	10 / day
RS14	join_group_notification	1 k	10 / day
RS15	follow_notification	10 k	100 / day
RS16	like_post_notification	10 k	100 / day
RS17	like_comment_notification	1 k	10 / day
RS18	admin_notification	100	10 / day

Identifier	Relation Name	Order of Magnitude	Estimated Growth
RS19	group_notification	1 k	10 / day
RS20	like_post	10 k	100 / day
RS21	like_comment	10 k	100 / day
RS22	follows_restaurant	10 k	100 / day
RS23	follows_client	10 k	100 / day
RS24	comment_relationship	10 k	100 / day
RS25	group_member	1 k	10 / day
RS26	request_follow	10 k	100 / day
RS27	request_join	1 k	10 / day

Table 45: Raffia workload

2. Proposed Indices

We used indexes to improve database performance by enabling quicker location and retrieval of specific rows.

2.1. Performance Indices

Some queries can take a significant amount of time to execute. By implementing performance indexes, we can improve the speed of SELECT queries, but this may lead to longer execution times for INSERT, UPDATE, and DELETE operations.

The tables shown below illustrate the performance indexes that are utilized.

Index	IDX01
Relation	review_post
Attribute	client_id
Type	Hash
Cardinality	Medium
Clustering	No
Justification	The 'review_post' table is accessed with a very high frequency for building the feed. Filtering can be done using hash (exact match) as it does not require ordering. While clustering on the userId column could theoretically improve performance—since the most common access pattern involves retrieving posts by a specific user (e.g., when viewing a user's profile)—the high update frequency of this table makes clustering impractical.
SQL code	`CREATE INDEX idx_client_review ON review_post USING hash(client_id);`

Table 46: clientid index

Index	IDX02
Relation	notification
Attribute	user_id
Type	Hash
Cardinality	Medium
Clustering	No
Justification	The 'notification' table is accessed with a very high frequency to get each user's notifications. Filtering can be done using hash (exact match) as it does not require ordering. The best candidate index for clustering would be the 'userId' attribute, since the search that is most commonly done is for a given user's notifications. However, due to its high update frequency, clustering is not viable.
SQL code	`CREATE INDEX idx_receiver_notification ON notification USING hash(user_id);`

Table 46: userId index

Index	IDX03
Relation	restaurant
Attribute	type
Type	B-Tree
Cardinality	Low
Clustering	Yes
Justification	The 'restaurant' table is frequently queried to filter restaurants by their type (an enum indicating the type of cuisine they serve). A clustered index on the type attribute could improve performance by organizing rows with the same type together on disk. Although a hash index would be more appropriate for exact matches, clustering requires the use of a B-tree index.
SQL code	`CREATE INDEX idx_type_restaurant ON restaurant(type); CLUSTER restaurant USING idx_type_restaurant;`

Table 47: type index

2.2. Full-text Search Indices

To improve text search efficiency, we created Full-Text Search indexes on the tables and attributes that are most likely to be queried frequently. Details of these indexes are provided in the following tables:

Index	IDX04
Relation	user
Attribute	name, username, description
Type	GIN
Clustering	No
Justification	To enable full-text search capabilities for finding users by matching names, usernames or profile descriptions a GIN index is used. This index type is chosen because the fields being indexed are expected to remain relatively static.
SQL code	

```
ALTER TABLE "user" ADD COLUMN tsvectors TSVECTOR;

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

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.name <> OLD.name OR NEW.username <> OLD.username OR NEW.description <> OLD.description) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.name), 'A') ||
                setweight(to_tsvector('english', NEW.username), 'B') ||
                setweight(to_tsvector('english', NEW.description), 'C')
            );
        END IF;
    END IF;

    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER user_search_update
```

```
BEFORE INSERT OR UPDATE ON "user"
FOR EACH ROW
EXECUTE PROCEDURE user_search_update();

CREATE INDEX search_user ON "user" USING GIN (tsvectors);
```

Table 48: search name, username and description index

Index	IDX05
Relation	post
Attribute	content
Type	GIN
Clustering	No
Justification	The `post` table is frequently queried to search for specific content within posts. To enable efficient full-text search capabilities, a GIN index is used on the `content` attribute. This index type is chosen because it allows for fast searching and retrieval of text data, and the `content` field is not expected to change frequently.
SQL code	

```
ALTER TABLE post ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = setweight(to_tsvector('english', NEW.content), 'A');
    END IF;

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.content <> OLD.content) THEN
            NEW.tsvectors = setweight(to_tsvector('english', NEW.content), 'A');
        END IF;
    END IF;

    RETURN NEW;
END $$
LANGUAGE plpgsql;

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

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

Table 49: search content in post index

Index	IDX06
Relation	comment
Attribute	content
Type	GIN
Clustering	No
Justification	The `comment` table is frequently queried to search for specific content within comments. To enable efficient full-text search capabilities, a GIN index is used on the `content` attribute. This index type is chosen because it allows for fast searching and retrieval of text data, and the `content` field is not expected to change frequently.

Index	IDX06
SQL code	

```
ALTER TABLE comment ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION comment_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = setweight(to_tsvector('english', NEW.content), 'A');
    END IF;

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.content <> OLD.content) THEN
            NEW.tsvectors = setweight(to_tsvector('english', NEW.content), 'A');
        END IF;
    END IF;

    RETURN NEW;
END $$
LANGUAGE plpgsql;

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

CREATE INDEX search_comment ON comment USING GIN (tsvectors);
```

Table 50: search content in comment index

Index	IDX07
Relation	group
Attribute	name, description
Type	GIN
Clustering	No
Justification	The `group` table is frequently queried to search for specific groups by their name or description. To enable efficient full-text search capabilities, a GIN index is used on the `name` and `description` attributes. This index type is chosen because it allows for fast searching and retrieval of text data, and these fields are not expected to change frequently.
SQL code	

```
ALTER TABLE "group" ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION group_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.name), 'A') ||
            setweight(to_tsvector('english', NEW.description), 'B')
        );
    END IF;

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.name <> OLD.name OR NEW.description <> OLD.description) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.name), 'A') ||
                setweight(to_tsvector('english', NEW.description), 'B')
            );
        END IF;
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER group_search_update
BEFORE INSERT OR UPDATE ON "group"
FOR EACH ROW
EXECUTE PROCEDURE group_search_update();

CREATE INDEX search_group ON "group" USING GIN (tsvectors);
```

Table 51: Search group by name and description index

3. Triggers

To implement integrity rules that cannot be enforced more simply, we identify and describe the necessary triggers by outlining the event, condition, and activation code.

Trigger	TRIGGER01
Description	A user can only like a comment once (business rule BR10)
SQL code	

```
CREATE FUNCTION verify_like_comment()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT 1 FROM like_comment
               WHERE user_id = NEW.user_id AND comment_id = NEW.comment_id) THEN
        RAISE EXCEPTION 'Users can only like a comment once';
    END IF;

    RETURN NEW;
END;
$$
LANGUAGE plpgsql;

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

Table 52: Verify like comment trigger

Trigger	TRIGGER02
Description	A user can only like a post once (business rule BR11)
SQL code	

```
CREATE FUNCTION verify_like_post()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM like_post WHERE NEW.user_id = user_id AND NEW.post_id = post_id) THEN
    RAISE EXCEPTION 'Users can only like a post once';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_like_post
BEFORE INSERT OR UPDATE ON like_post
```

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

Table 53: Verify like post trigger

Trigger	TRIGGER03
Description	A user cannot request to join a group they already in (business rule BR09)
SQL code	

```
CREATE FUNCTION verify_group_request()
RETURNS TRIGGER AS
$$
BEGIN
        IF EXISTS (SELECT * FROM request_join WHERE NEW.client_id = client_id AND NEW.group_id = group_id) THEN
        RAISE EXCEPTION 'Users cannot request to join group they already belong to';
        END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_group_request
BEFORE INSERT OR UPDATE ON request_join
FOR EACH ROW
EXECUTE PROCEDURE verify_group_request();
```

Table 54: Request join group trigger

Trigger	TRIGGER04
Description	A user cannot join a group they already belong to (business rule BR04)
SQL code	

```
CREATE FUNCTION verify_group_entry()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM group_member WHERE NEW.client_id = client_id AND NEW.group_id = group_id) THEN
    RAISE EXCEPTION 'Users cannot join a group they already belong to';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_group_entry
BEFORE INSERT OR UPDATE ON group_member
FOR EACH ROW
EXECUTE PROCEDURE verify_group_entry();
```

Table 55: Join group trigger

Trigger	TRIGGER05
Description	Users cannot request to follow other users they already follow (business rule BR13)
SQL code	

```
CREATE FUNCTION verify_follow_client_request()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM request_follow WHERE NEW.requester_client_id = requester_client_id AND NEW.receiver_client_
    RAISE EXCEPTION 'Users cannot request to follow other users they already follow';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_follow_client_request
BEFORE INSERT OR UPDATE ON request_follow
FOR EACH ROW
EXECUTE PROCEDURE verify_follow_client_request();
```

Table 56: Request follow trigger

Trigger	TRIGGER06
Description	Users cannot follow other users they already follow (business rule BR14)
SQL code	

```
CREATE FUNCTION verify_follow_client()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM follows_client WHERE NEW.sender_client_id = sender_client_id AND NEW.followed_client_id =
    RAISE EXCEPTION 'Users cannot follow other clients they already follow';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_follow_client
BEFORE INSERT OR UPDATE ON follows_client
FOR EACH ROW
EXECUTE PROCEDURE verify_follow_client();
```

Table 57: Follow other users trigger

Trigger	TRIGGER07
Description	Users cannot follow restaurants they already follow (business rule BR015)
SQL code	

```
CREATE FUNCTION verify_follow_restaurant()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM follows_restaurant WHERE NEW.client_id = client_id AND NEW.restaurant_id = restaurant_id)
    RAISE EXCEPTION 'Users cannot follow restaurants they already follow';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_follow_restaurant
BEFORE INSERT OR UPDATE ON follows_restaurant
FOR EACH ROW
EXECUTE PROCEDURE verify_follow_restaurant();
```

Table 58: Follow other restaurants trigger

Trigger	TRIGGER08
Description	A user cannot follow themselves (business rule BR07)

Trigger	TRIGGER08
SQL code	

```
CREATE FUNCTION verify_self_following()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM request_follow WHERE NEW.requester_client_id = requester_client_id AND NEW.receiver_client_
    RAISE EXCEPTION 'Users cannot request to follow themselfs';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_self_following
BEFORE INSERT OR UPDATE ON request_follow
FOR EACH ROW
EXECUTE PROCEDURE verify_self_following();
```

Table 59: Self follow trigger

Trigger	TRIGGER09
Description	A user can only post to a group they belong to. (business rule BR12)
SQL code	

```
CREATE FUNCTION verify_group_membership()
RETURNS TRIGGER AS $$
BEGIN
    IF NOT EXISTS (
    SELECT 1
    FROM group_member
    WHERE client_id = NEW.client_id AND group_id = NEW.group_id
    ) THEN
    RAISE EXCEPTION 'User can only post to a group they belong to';
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER verify_group_membership
BEFORE INSERT ON review_post
FOR EACH ROW
EXECUTE PROCEDURE verify_group_membership();
```

Table 60: Post on group trigger

Trigger	TRIGGER010
Description	A user cannot like on posts in groups they do not belong to (business rule BR16)
SQL code	

```
CREATE FUNCTION check_group_membership_like()
RETURNS TRIGGER AS $$
DECLARE
    group_member_count INTEGER;
BEGIN
    SELECT COUNT(*)
    INTO group_member_count
    FROM group_member
    WHERE client_id = NEW.user_id
        AND group_id = (SELECT group_id FROM post WHERE id = NEW.post_id);

    IF group_member_count = 0 THEN
        RAISE EXCEPTION 'User is not a member of the group and cannot like this post';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_group_membership_like
BEFORE INSERT ON like_post
FOR EACH ROW
EXECUTE FUNCTION check_group_membership_like();
```

Table 62: Group post like restrictions trigger

Trigger	TRIGGER011
Description	A user comment on posts in groups they do not belong to ((business rule BR17))
SQL code	

```
CREATE FUNCTION check_group_membership_comment()
RETURNS TRIGGER AS $$
DECLARE
    group_member_count INTEGER;
BEGIN
    SELECT COUNT(*)
    INTO group_member_count
    FROM group_member
    WHERE client_id = NEW.user_id
        AND group_id = (SELECT group_id FROM post WHERE id = NEW.post_id);

    IF group_member_count = 0 THEN
        RAISE EXCEPTION 'User is not a member of the group and cannot comment on this post';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_group_membership_comment
BEFORE INSERT ON comment
FOR EACH ROW
EXECUTE FUNCTION check_group_membership_comment();
```

Table 63: Group post comment restrictions trigger

Trigger	TRIGGER012
Description	A group owner is also a member of the group. (business rule BR18)
SQL code	

```
CREATE FUNCTION add_group_owner_as_member() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO group_member (client_id, group_id)
    VALUES (NEW.owner_id, NEW.id);
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER add_group_owner_as_member
AFTER INSERT ON "group"
```

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

Table 64: Group owner trigger

Trigger	TRIGGER013
Description	Delete join group requests after acceptance (business rule BR19)
SQL code	

```
CREATE FUNCTION delete_join_requests_after_acceptance()
RETURNS TRIGGER AS
$$
BEGIN
    DELETE FROM request_join
    WHERE client_id = NEW.client_id AND group_id = NEW.group_id;
    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER delete_join_requests_after_acceptance
AFTER INSERT ON group_member
FOR EACH ROW
EXECUTE PROCEDURE delete_join_requests_after_acceptance();
```

Table 65: Delete request join after acceptance trigger

Trigger	TRIGGER014
Description	Delete follow requests after acceptance (business rule BR20)
SQL code	

```
CREATE FUNCTION delete_follow_requests_after_acceptance()
RETURNS TRIGGER AS $$
BEGIN
DELETE FROM request_follow
WHERE requester_client_id = NEW.sender_client_id AND receiver_client_id = NEW.followed_client_id;
RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER delete_follow_requests_after_acceptance
AFTER INSERT ON follows_client
FOR EACH ROW
EXECUTE PROCEDURE delete_follow_requests_after_acceptance();
```

Table 66: Delete follow request after acceptance trigger

Trigger	TRIGGER015
Description	Create a follow request notification after an insertion on table request_follow
SQL code	

```
CREATE FUNCTION create_follow_notification() RETURNS TRIGGER AS $$
DECLARE
    sender_name TEXT;
    notification_id INTEGER;
BEGIN
    SELECT name INTO sender_name
    FROM "user"
    WHERE id = NEW.requester_client_id;
    INSERT INTO notification (content, viewed, user_id)
    VALUES (sender_name || ' has sent you a follow request', FALSE, NEW.receiver_client_id);

    SELECT currval('notification_id_seq') INTO notification_id;

    INSERT INTO request_notification (id)
    VALUES (notification_id);

    INSERT INTO follow_notification (id, sender_client_id, receiver_client_id)
    VALUES (notification_id, NEW.requester_client_id, NEW.receiver_client_id);

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_request_follow
AFTER INSERT ON request_follow
FOR EACH ROW
EXECUTE PROCEDURE create_follow_notification();
```

Table 67: Create follow request after acceptance trigger

Trigger	TRIGGER016
Description	Create a join group request notification after an insertion on table request_join
SQL code	

```
CREATE FUNCTION create_join_group_notification() RETURNS TRIGGER AS $$
DECLARE
    requester_name TEXT;
    notification_id INTEGER;
BEGIN
    SELECT name INTO requester_name
    FROM "user"
    WHERE id = NEW.client_id;

    INSERT INTO notification (content, viewed, user_id)
    VALUES (requester_name || ' has requested to join your group', FALSE, NEW.group_id);

    SELECT currval('notification_id_seq') INTO notification_id;

    INSERT INTO request_notification (id)
    VALUES (notification_id);

    INSERT INTO join_group_notification (id, client_id, group_id)
    VALUES (notification_id, NEW.client_id, NEW.group_id);

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_request_join
AFTER INSERT ON request_join
FOR EACH ROW
EXECUTE PROCEDURE create_join_group_notification();
```

Table 68: Create a join group notification trigger

Trigger	TRIGGER017
Description	Create a like post notification after an insertion on table like_post

Trigger	TRIGGER017
SQL code	

```
CREATE FUNCTION create_like_post_notification() RETURNS TRIGGER AS $$
DECLARE
    liker_name TEXT;
    notification_id INTEGER;
    post_owner_id INTEGER;
BEGIN
    SELECT name INTO liker_name
    FROM "user"
    WHERE id = NEW.user_id;

    IF EXISTS (SELECT 1 FROM review_post WHERE id = NEW.post_id) THEN
        SELECT client_id INTO post_owner_id
        FROM review_post
        WHERE id = NEW.post_id;
    ELSE
        SELECT client_id INTO post_owner_id
        FROM informational_post
        WHERE id = NEW.post_id;
    END IF;

    INSERT INTO notification (content, viewed, user_id)
    VALUES (liker_name || ' liked your post', FALSE, post_owner_id);

    SELECT currval('notification_id_seq') INTO notification_id;

    INSERT INTO like_notification (id)
    VALUES (notification_id);

    INSERT INTO like_post_notification (id, user_id, post_id)
    VALUES (notification_id, NEW.user_id, NEW.post_id);

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_like_post
AFTER INSERT ON like_post
FOR EACH ROW
EXECUTE PROCEDURE create_like_post_notification();
```

Table 69: Create a like notification trigger

Trigger	TRIGGER018
Description	Create a like comment notification after an insertion on table like_comment
SQL code	

```
CREATE FUNCTION create_like_comment_notification() RETURNS TRIGGER AS $$
DECLARE
    liker_name TEXT;
    notification_id INTEGER;
BEGIN
    SELECT name INTO liker_name
    FROM "user"
    WHERE id = NEW.user_id;

    INSERT INTO notification (content, viewed, user_id)
    VALUES (liker_name || ' liked your comment', FALSE, (SELECT user_id FROM comment WHERE id = NEW.comment_id));

    SELECT currval('notification_id_seq') INTO notification_id;
    INSERT INTO like_notification (id)
    VALUES (notification_id);

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

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_like_comment
AFTER INSERT ON like_comment
FOR EACH ROW
EXECUTE PROCEDURE create_like_comment_notification();
```

Table 70: Create a like comment notification trigger

Trigger	TRIGGER019
Description	Send a group notification to every group member after a post has been added to the group
SQL code	

```
CREATE FUNCTION create_group_notification_after_post() RETURNS TRIGGER AS $$
DECLARE
    group_name TEXT;
    member_id INTEGER;
    notification_id INTEGER;
BEGIN
    SELECT name INTO group_name
    FROM "group"
    WHERE id = NEW.group_id;

    FOR member_id IN
        SELECT client_id
        FROM group_member
        WHERE group_id = NEW.group_id
    LOOP
        INSERT INTO notification (content, viewed, user_id)
        VALUES ('A post has been made in the group ' || group_name, FALSE, member_id);

        SELECT currval('notification_id_seq') INTO notification_id;

        INSERT INTO general_notification (id)
        VALUES (notification_id);

        INSERT INTO group_notification (id, group_id)
        VALUES (notification_id, NEW.group_id);
    END LOOP;

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_review_post
AFTER INSERT ON review_post
FOR EACH ROW
EXECUTE PROCEDURE create_group_notification_after_post();
```

Table 71: Create a group post notification trigger

Trigger	TRIGGER020
Description	Send a group notification to every group member after a user joins the group

Trigger	TRIGGER020
SQL code	

```
CREATE FUNCTION create_group_notification_after_join() RETURNS TRIGGER AS $$
DECLARE
    group_name TEXT;
    member_id INTEGER;
    notification_id INTEGER;
BEGIN
    SELECT name INTO group_name
    FROM "group"
    WHERE id = NEW.group_id;

    FOR member_id IN
        SELECT client_id
        FROM group_member
        WHERE group_id = NEW.group_id
    LOOP
        INSERT INTO notification (content, viewed, user_id)
        VALUES ('A new member has joined the group ' || group_name, FALSE, member_id);

        SELECT currval('notification_id_seq') INTO notification_id;

        INSERT INTO general_notification (id)
        VALUES (notification_id);

        INSERT INTO group_notification (id, group_id)
        VALUES (notification_id, NEW.group_id);
    END LOOP;

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_groupmember
AFTER INSERT ON group_member
FOR EACH ROW
EXECUTE PROCEDURE create_group_notification_after_join();
```

Table 72: Create a new group member notification trigger

Trigger	TRIGGER020
Description	Send a group notification to every group member after a user leaves the group
SQL code	

```
CREATE FUNCTION create_group_notification_after_leave() RETURNS TRIGGER AS $$
DECLARE
    group_name TEXT;
    member_id INTEGER;
    notification_id INTEGER;
BEGIN
    SELECT name INTO group_name
    FROM "group"
    WHERE id = OLD.group_id;

    FOR member_id IN
        SELECT client_id
        FROM group_member
        WHERE group_id = OLD.group_id
    LOOP
        INSERT INTO notification (content, viewed, user_id)
        VALUES ('A member has left the group ' || group_name, FALSE, member_id);
        SELECT currval('notification_id_seq') INTO notification_id;

        INSERT INTO general_notification (id)
        VALUES (notification_id);

        INSERT INTO group_notification (id, group_id)
        VALUES (notification_id, OLD.group_id);
    END LOOP;

    RETURN OLD;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_delete_groupmember
AFTER DELETE ON group_member
FOR EACH ROW
EXECUTE PROCEDURE create_group_notification_after_leave();
```

Table 73: After delete group me notification trigger

Trigger	TRIGGER021
Description	Send an admin notification after a user is blocked
SQL code	

```
CREATE FUNCTION create_admin_notification_after_block() RETURNS TRIGGER AS $$
DECLARE
    notification_id INTEGER;
BEGIN
    INSERT INTO notification (content, viewed, user_id)
    VALUES ('You have been blocked', FALSE, NEW.id);

    SELECT currval('notification_id_seq') INTO notification_id;

    INSERT INTO general_notification (id)
    VALUES (notification_id);

    INSERT INTO admin_notification (id)
    VALUES (notification_id);

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_block_user
AFTER UPDATE OF is_blocked ON "user"
FOR EACH ROW
WHEN (NEW.is_blocked = TRUE AND OLD.is_blocked = FALSE)
EXECUTE PROCEDURE create_admin_notification_after_block();
```

Table 74: After a user has been blocked notification trigger

4. Transactions

The following transactions are used to maintain data integrity during multiple operations.

Transaction	TRAN01
Description	Insert a new review and update the restaurant's rating average
Justification	Ensures that the insertion of a new review and the update of the restaurant's rating average are treated as a single atomic operation. If two reviews are inserted simultaneously, a racing condition may happen when selecting the average rating of the restaurant.This prevents Phantom Reads where the information retrieved in the SELECT statements could differ due to concurrent insertions.

Transaction	TRAN01
Isolation level	SERIALIZABLE
SQL code	

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

INSERT INTO review_post (id, rating, client_id, restaurant_id)
VALUES (:post_id, :rating, :client_id, :restaurant_id);

UPDATE restaurant
SET rating_average = (
    SELECT AVG(rating)
    FROM review_post
    WHERE restaurant_id = :restaurant_id
)
WHERE id = :restaurant_id;

COMMIT;
```

Table 75: Update restaurant average rating transaction

Transaction: Delete User

Transaction	TRAN02
Description	Allows and authenticated user to "delete" their account or an admin to delete a user account.
Justification	Uses SERIALIZABLE to ensure atomicity and consistency when deleting the user account and related records
Isolation level	SERIALIZABLE
SQL code	

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

DELETE FROM notification
WHERE user_id = :userid;

DELETE FROM request_follow
WHERE requester_client_id = :userid OR receiver_client_id = :userid;

DELETE FROM request_join
WHERE client_id = :userid;

DELETE FROM groupmember
WHERE client_id = :userid;

DELETE FROM followsclient
WHERE clientid1 = :userid OR clientid2 = :userid;

DELETE FROM followsrestaurant
WHERE clientid = :userid;

UPDATE "user"
SET name = 'Anonymous user',
    username = 'anon' || id,
    email = 'anon' || id || '@example.com',
    password = 'deleted',
    image = 'default_image.jpg'
```

```
WHERE id = :userid;

COMMIT;
```

Table 76: Delete user transaction

Transaction	**TRAN03**
Description	Creating a new follow notification
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of a follow notification fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to notification_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO notification (content, viewed, user_id)
VALUES ($content, $viewed, $user_id);

SELECT currval('notification_id_seq') INTO notification_id;

INSERT INTO request_notification (id)
VALUES (notification_id);

INSERT INTO follow_notification (id, sender_client_id, receiver_client_id)
VALUES (notification_id, $sender_client_id, $receiver_client_id);

COMMIT;
```

Table 77: Creating a new follow notification transaction

Transaction	**TRAN04**
Description	Creating a new join group notification
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of a join group notification fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to notification_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO notification (content, viewed, user_id)
VALUES ($content, $viewed, $user_id);

SELECT currval('notification_id_seq') INTO notification_id;

INSERT INTO request_notification (id)
VALUES (notification_id);

INSERT INTO join_group_notification (id, client_id, group_id)
VALUES (notification_id, $client_id, $group_id);
```

```
COMMIT;
```

Table 78: Creating a new join group notification transaction

Transaction	**TRAN05**
Description	Creating a new like post notification
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of a like post notification fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to notification_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO notification (content, viewed, user_id)
VALUES ($content, $viewed, $user_id);

SELECT currval('notification_id_seq') INTO notification_id;

INSERT INTO like_notification (id)
VALUES (notification_id);

INSERT INTO like_post_notification (id, user_id, post_id)
VALUES (notification_id, $user_id, $post_id);

COMMIT;
```

Table 79: Creating a new like post notification transaction

Transaction	**TRAN06**
Description	Creating a new like comment notification
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of a like comment notification fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to notification_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO notification (content, viewed, user_id)
VALUES ($content, $viewed, $user_id);

SELECT currval('notification_id_seq') INTO notification_id;

INSERT INTO like_notification (id)
VALUES (notification_id);

INSERT INTO like_comment_notification (id, user_id, comment_id)
VALUES (notification_id, $user_id, $comment_id);
```

```
COMMIT;
```

Table 80: Creating a new like comment notification transaction

Transaction	**TRAN07**
Description	Creating a new admin notification
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of an admin notification fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to notification_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO notification (content, viewed, user_id)
VALUES ($content, $viewed, $user_id);

SELECT currval('notification_id_seq') INTO notification_id;

INSERT INTO general_notification (id)
VALUES (notification_id);

INSERT INTO admin_notification (id)
VALUES (notification_id);

COMMIT;
```

Table 81: Creating a new admin notification transaction

Transaction	**TRAN08**
Description	Creating a new group notification
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of a group notification fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to notification_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO notification (content, viewed, user_id)
VALUES ('A member has left the group ' || $group_name, FALSE, $member_id);

SELECT currval('notification_id_seq') INTO notification_id;

INSERT INTO general_notification (id)
VALUES (notification_id);

INSERT INTO group_notification (id, group_id)
VALUES (notification_id, $group_id);
```

```
COMMIT;
```

Table 82: Creating a new like group notification transaction

Transaction	**TRAN09**
Description	Registering a new restaurant
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of a restaurant fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to user_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO "user" (name, username, password, email, image)
VALUES ($name, $username, $password, $email, $image);

INSERT INTO restaurant (id, rating_average, type, capacity)
VALUES (currval('user_id_seq'), $rating_average, $type, $capacity);

COMMIT;
```

Table 83: Registering a new restaurant transaction

Transaction	**TRAN10**
Description	Registering a new client
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of a client fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to user_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO "user" (name, username, password, email, image)
VALUES ($name, $username, $password, $email, $image);

INSERT INTO client (id)
VALUES (currval('user_id_seq'));

COMMIT;
```

Table 84: Registering a new client transaction

Transaction	**TRAN11**
Description	Registering a new client

Transaction	TRAN11
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of an informational post fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to post_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO post (datetime, content, images, group_id)
VALUES (NOW(), $content, $images, $group_id);

INSERT INTO informational_post (id, restaurant_id)
VALUES (currval('post_id_seq'), $restaurant_id);

COMMIT;
```

Table 85: Creating a new informational post transaction

Transaction	TRAN12
Description	Creating a new review post
Justification	To ensure data consistency, it is essential to utilize a transaction, allowing the entire operation to execute without errors. If any error arises, a ROLLBACK will be triggered (for instance, if the insertion of a review post fails). The isolation level is set to REPEATABLE READ to prevent potential updates caused by concurrent transactions (an update to post_id_seq may happen), which could lead to the storage of inconsistent data.
Isolation level	REPEATABLE READ
SQL code	

```
BEGIN;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO post (datetime, content, images, group_id)
VALUES (NOW(), $content, $images, $group_id);

INSERT INTO review_post (id, rating, client_id, restaurant_id)
VALUES (currval('post_id_seq'), $rating, $client_id, $restaurant_id);

COMMIT;
```

Table 86: Creating a new review post transaction

Annex A. SQL Code

> The database scripts are included in this annex to the EBD component.
>
> The database creation script and the population script should be presented as separate elements. The creation script includes the code necessary to build (and rebuild) the database. The population script includes an amount of tuples suitable for testing and with plausible values for the fields of the database.
>
> The complete code of each script must be included in the group's git repository and links added here.

For more information, you can check the View SQL Directory for the SQL files related to the project.

A.1. Database schema

```
DROP SCHEMA IF EXISTS raffia CASCADE;

CREATE SCHEMA raffia;

SET search_path TO raffia;

CREATE TYPE restaurant_type AS ENUM ('American', 'Chinese', 'Italian', 'Japanese', 'Mexican', 'Thai', 'Portuguese', 'Bu


CREATE TABLE "user" (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    username TEXT UNIQUE NOT NULL,
    password TEXT NOT NULL,
    description TEXT,
    email TEXT UNIQUE NOT NULL,
    image TEXT NOT NULL,
    is_blocked BOOLEAN NOT NULL DEFAULT FALSE,
    is_admin BOOLEAN NOT NULL DEFAULT FALSE,
    is_deleted BOOLEAN NOT NULL DEFAULT FALSE
);

CREATE TABLE restaurant (
    id INTEGER PRIMARY KEY REFERENCES "user"(id) ON UPDATE CASCADE ON DELETE CASCADE,
    rating_average FLOAT,
    type restaurant_type NOT NULL,
    capacity INTEGER NOT NULL CHECK (capacity > 0)
);

CREATE TABLE client (
    id INTEGER PRIMARY KEY REFERENCES "user"(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE "group" (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    description TEXT,
    is_public BOOLEAN NOT NULL,
    owner_id INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE post (
    id SERIAL PRIMARY KEY,
    datetime TIMESTAMP NOT NULL DEFAULT NOW(),
    content TEXT NOT NULL,
    images TEXT[]
);

CREATE TABLE review_post (
    id INTEGER PRIMARY KEY REFERENCES post(id) ON UPDATE CASCADE ON DELETE CASCADE,
    rating INTEGER NOT NULL CHECK (rating >= 0 AND rating <= 5),
    client_id  INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE,
    group_id INTEGER REFERENCES "group"(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE informational_post  (
    id INTEGER PRIMARY KEY REFERENCES post(id) ON UPDATE CASCADE ON DELETE CASCADE,
    restaurant_id INTEGER REFERENCES restaurant(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE comment (
    id SERIAL PRIMARY KEY,
    content TEXT NOT NULL,
    datetime TIMESTAMP NOT NULL DEFAULT NOW(),
    post_id INTEGER REFERENCES post(id) ON UPDATE CASCADE ON DELETE CASCADE,
    user_id INTEGER REFERENCES "user"(id) ON UPDATE CASCADE
);

CREATE TABLE notification (
    id SERIAL PRIMARY KEY,
    datetime TIMESTAMP NOT NULL DEFAULT NOW(),
    content TEXT NOT NULL,
    viewed BOOLEAN NOT NULL DEFAULT FALSE,
    user_id INTEGER REFERENCES "user"(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE request_notification  (
    id INTEGER PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE like_notification  (
    id INTEGER PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE comment_notification (
    id INTEGER PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE ON DELETE CASCADE,
    comment_id  INTEGER REFERENCES comment(id)
);

CREATE TABLE general_notification (
    id INTEGER PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE request_join (
    datetime TIMESTAMP NOT NULL DEFAULT NOW(),
    client_id  INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE,
    group_id  INTEGER REFERENCES "group"(id) ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (client_id, group_id )
);

CREATE TABLE join_group_notification  (
    id INTEGER PRIMARY KEY REFERENCES request_notification(id) ON UPDATE CASCADE ON DELETE CASCADE,
    client_id  INTEGER,
    group_id INTEGER,
    FOREIGN KEY (client_id, group_id ) REFERENCES request_join(client_id, group_id ) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE request_follow  (
    datetime TIMESTAMP NOT NULL DEFAULT NOW(),
    requester_client_id  INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE,
    receiver_client_id  INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (requester_client_id, receiver_client_id )
);

CREATE TABLE follow_notification (
    id INTEGER PRIMARY KEY REFERENCES request_notification(id) ON UPDATE CASCADE ON DELETE CASCADE,
    sender_client_id  INTEGER,
    receiver_client_id  INTEGER,
    FOREIGN KEY (sender_client_id, receiver_client_id) REFERENCES request_follow(requester_client_id, receiver_client_i
);

CREATE TABLE like_post  (
    datetime TIMESTAMP NOT NULL DEFAULT NOW(),
    user_id INTEGER REFERENCES "user"(id) ON UPDATE CASCADE ON DELETE CASCADE,
    post_id  INTEGER REFERENCES post(id) ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (user_id, post_id)
);

CREATE TABLE like_post_notification  (
    id INTEGER PRIMARY KEY REFERENCES like_notification(id) ON UPDATE CASCADE ON DELETE CASCADE,
    user_id  INTEGER,
    post_id  INTEGER,
    FOREIGN KEY (user_id, post_id) REFERENCES like_post(user_id , post_id ) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE like_comment  (
    datetime TIMESTAMP NOT NULL DEFAULT NOW(),
    user_id  INTEGER REFERENCES "user"(id) ON UPDATE CASCADE ON DELETE CASCADE,
    comment_id  INTEGER REFERENCES comment(id) ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (user_id , comment_id )
);

CREATE TABLE like_comment_notification  (
    id INTEGER PRIMARY KEY REFERENCES like_notification(id) ON UPDATE CASCADE ON DELETE CASCADE,
    user_id  INTEGER,
    comment_id  INTEGER,
    FOREIGN KEY (user_id, comment_id ) REFERENCES like_comment(user_id, comment_id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE admin_notification  (
    id INTEGER PRIMARY KEY REFERENCES general_notification(id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE group_notification  (
    id INTEGER PRIMARY KEY REFERENCES general_notification(id) ON UPDATE CASCADE ON DELETE CASCADE,
    group_id  INTEGER REFERENCES "group"(id)
);

CREATE TABLE follows_restaurant  (
    client_id  INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE,
    restaurant_id  INTEGER REFERENCES restaurant(id) ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (client_id, restaurant_id )
);

CREATE TABLE follows_client  (
    sender_client_id  INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE,
    followed_client_id  INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (sender_client_id , followed_client_id)
);

CREATE TABLE comment_relationship  (
    child INTEGER REFERENCES comment(id) ON UPDATE CASCADE ON DELETE CASCADE,
    parent INTEGER REFERENCES comment(id) ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (child, parent)
);

CREATE TABLE group_member  (
    client_id  INTEGER REFERENCES client(id) ON UPDATE CASCADE ON DELETE CASCADE,
    group_id  INTEGER REFERENCES "group"(id) ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (client_id, group_id )
);

CREATE INDEX notified_user_notification ON notification USING btree (user_id);
CLUSTER notification USING notified_user_notification;

CREATE INDEX idx_client_review ON review_post USING hash(client_id);
CREATE INDEX idx_receiver_notification ON notification USING hash(user_id);
CREATE INDEX idx_type_restaurant ON restaurant(type);
CLUSTER restaurant USING idx_type_restaurant;

ALTER TABLE "user" ADD COLUMN tsvectors TSVECTOR;

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

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.name <> OLD.name OR NEW.username <> OLD.username OR NEW.description <> OLD.description) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.name), 'A') ||
                setweight(to_tsvector('english', NEW.username), 'B') ||
                setweight(to_tsvector('english', NEW.description), 'C')
            );
        END IF;
    END IF;

    RETURN NEW;
END $$
LANGUAGE plpgsql;

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

CREATE INDEX search_user ON "user" USING GIN (tsvectors);

ALTER TABLE post ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = setweight(to_tsvector('english', NEW.content), 'A');
    END IF;

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.content <> OLD.content) THEN
            NEW.tsvectors = setweight(to_tsvector('english', NEW.content), 'A');
        END IF;
    END IF;

    RETURN NEW;
END $$
LANGUAGE plpgsql;

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

CREATE INDEX search_post ON post USING GIN (tsvectors);

ALTER TABLE comment ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION comment_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = setweight(to_tsvector('english', NEW.content), 'A');
    END IF;

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.content <> OLD.content) THEN
            NEW.tsvectors = setweight(to_tsvector('english', NEW.content), 'A');
        END IF;
    END IF;

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

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

CREATE INDEX search_comment ON comment USING GIN (tsvectors);


ALTER TABLE "group" ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION group_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
            setweight(to_tsvector('english', NEW.name), 'A') ||
            setweight(to_tsvector('english', NEW.description), 'B')
        );
    END IF;

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.name <> OLD.name OR NEW.description <> OLD.description) THEN
            NEW.tsvectors = (
                setweight(to_tsvector('english', NEW.name), 'A') ||
                setweight(to_tsvector('english', NEW.description), 'B')
            );
        END IF;
    END IF;

    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER group_search_update
BEFORE INSERT OR UPDATE ON "group"
FOR EACH ROW
EXECUTE PROCEDURE group_search_update();

CREATE INDEX search_group ON "group" USING GIN (tsvectors);

CREATE FUNCTION verify_like_comment()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT 1 FROM like_comment
               WHERE user_id = NEW.user_id AND comment_id = NEW.comment_id) THEN
        RAISE EXCEPTION 'Users can only like a comment once';
    END IF;

    RETURN NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER verify_like_comment
BEFORE INSERT OR UPDATE ON like_comment
FOR EACH ROW
EXECUTE PROCEDURE verify_like_comment();

CREATE FUNCTION verify_like_post()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM like_post WHERE NEW.user_id = user_id AND NEW.post_id = post_id) THEN
    RAISE EXCEPTION 'Users can only like a post once';
    END IF;
    RETURN NEW;
END
```

```
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_like_post
BEFORE INSERT OR UPDATE ON like_post
FOR EACH ROW
EXECUTE PROCEDURE verify_like_post();

CREATE FUNCTION verify_group_request()
RETURNS TRIGGER AS
$$
BEGIN
        IF EXISTS (SELECT * FROM request_join WHERE NEW.client_id = client_id AND NEW.group_id = group_id) THEN
        RAISE EXCEPTION 'Users cannot request to join group they already belong to';
        END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_group_request
BEFORE INSERT OR UPDATE ON request_join
FOR EACH ROW
EXECUTE PROCEDURE verify_group_request();

CREATE FUNCTION verify_group_entry()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM group_member WHERE NEW.client_id = client_id AND NEW.group_id = group_id) THEN
    RAISE EXCEPTION 'Users cannot join a group they already belong to';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_group_entry
BEFORE INSERT OR UPDATE ON group_member
FOR EACH ROW
EXECUTE PROCEDURE verify_group_entry();

CREATE FUNCTION verify_follow_client_request()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM request_follow WHERE NEW.requester_client_id = requester_client_id AND NEW.receiver_client_
    RAISE EXCEPTION 'Users cannot request to follow other users they already follow';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_follow_client_request
BEFORE INSERT OR UPDATE ON request_follow
FOR EACH ROW
EXECUTE PROCEDURE verify_follow_client_request();


CREATE FUNCTION verify_follow_client()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM follows_client WHERE NEW.sender_client_id = sender_client_id AND NEW.followed_client_id = 
    RAISE EXCEPTION 'Users cannot follow other clients they already follow';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
```

```sql
CREATE TRIGGER verify_follow_client
BEFORE INSERT OR UPDATE ON follows_client
FOR EACH ROW
EXECUTE PROCEDURE verify_follow_client();


CREATE FUNCTION verify_follow_restaurant()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM follows_restaurant WHERE NEW.client_id = client_id AND NEW.restaurant_id = restaurant_id) T
    RAISE EXCEPTION 'Users cannot follow restaurants they already follow';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_follow_restaurant
BEFORE INSERT OR UPDATE ON follows_restaurant
FOR EACH ROW
EXECUTE PROCEDURE verify_follow_restaurant();

CREATE FUNCTION verify_self_following()
RETURNS TRIGGER AS
$$
BEGIN
    IF EXISTS (SELECT * FROM request_follow WHERE NEW.requester_client_id = requester_client_id AND NEW.receiver_client_
    RAISE EXCEPTION 'Users cannot request to follow themselfs';
    END IF;
    RETURN NEW;
END
$$
LANGUAGE plpgsql;
CREATE TRIGGER verify_self_following
BEFORE INSERT OR UPDATE ON request_follow
FOR EACH ROW
EXECUTE PROCEDURE verify_self_following();

CREATE FUNCTION verify_group_membership()
RETURNS TRIGGER AS $$
BEGIN
    IF NOT EXISTS (
    SELECT 1
    FROM group_member
    WHERE client_id = NEW.client_id AND group_id = NEW.group_id
    ) THEN
    RAISE EXCEPTION 'User can only post to a group they belong to';
    END IF;
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER verify_group_membership
BEFORE INSERT ON review_post
FOR EACH ROW
EXECUTE PROCEDURE verify_group_membership();



CREATE FUNCTION check_group_membership_like()
RETURNS TRIGGER AS $$
DECLARE
    group_member_count INTEGER;
BEGIN
    SELECT COUNT(*)
    INTO group_member_count
    FROM group_member
    WHERE client_id = NEW.user_id
        AND group_id = (SELECT group_id FROM post WHERE id = NEW.post_id);

    IF group_member_count = 0 THEN
        RAISE EXCEPTION 'User is not a member of the group and cannot like this post';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_group_membership_like
BEFORE INSERT ON like_post
FOR EACH ROW
EXECUTE FUNCTION check_group_membership_like();


CREATE FUNCTION check_group_membership_comment()
RETURNS TRIGGER AS $$
DECLARE
    group_member_count INTEGER;
BEGIN
    SELECT COUNT(*)
    INTO group_member_count
    FROM group_member
    WHERE client_id = NEW.user_id
        AND group_id = (SELECT group_id FROM post WHERE id = NEW.post_id);

    IF group_member_count = 0 THEN
        RAISE EXCEPTION 'User is not a member of the group and cannot comment on this post';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_group_membership_comment
BEFORE INSERT ON comment
FOR EACH ROW
EXECUTE FUNCTION check_group_membership_comment();


CREATE FUNCTION add_group_owner_as_member() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO group_member (client_id, group_id)
    VALUES (NEW.owner_id, NEW.id);
    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER add_group_owner_as_member
AFTER INSERT ON "group"
FOR EACH ROW
EXECUTE PROCEDURE add_group_owner_as_member();



CREATE FUNCTION delete_join_requests_after_acceptance()
RETURNS TRIGGER AS
$$
BEGIN
    DELETE FROM request_join
    WHERE client_id = NEW.client_id AND group_id = NEW.group_id;
    RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER delete_join_requests_after_acceptance
AFTER INSERT ON group_member
FOR EACH ROW
EXECUTE PROCEDURE delete_join_requests_after_acceptance();


CREATE FUNCTION delete_follow_requests_after_acceptance()
RETURNS TRIGGER AS $$
BEGIN
DELETE FROM request_follow
WHERE requester_client_id = NEW.sender_client_id AND receiver_client_id = NEW.followed_client_id;
RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER delete_follow_requests_after_acceptance
AFTER INSERT ON follows_client
FOR EACH ROW
EXECUTE PROCEDURE delete_follow_requests_after_acceptance();

CREATE FUNCTION create_follow_notification() RETURNS TRIGGER AS $$
DECLARE
    sender_name TEXT;
    notification_id INTEGER;
BEGIN
    SELECT name INTO sender_name
    FROM "user"
    WHERE id = NEW.requester_client_id;

    INSERT INTO notification (content, viewed, user_id)
    VALUES (sender_name || ' has sent you a follow request', FALSE, NEW.receiver_client_id);

    SELECT currval('notification_id_seq') INTO notification_id;

    INSERT INTO request_notification (id)
    VALUES (notification_id);

    INSERT INTO follow_notification (id, sender_client_id, receiver_client_id)
    VALUES (notification_id, NEW.requester_client_id, NEW.receiver_client_id);

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_request_follow
AFTER INSERT ON request_follow
FOR EACH ROW
EXECUTE PROCEDURE create_follow_notification();

CREATE FUNCTION create_join_group_notification() RETURNS TRIGGER AS $$
DECLARE
    requester_name TEXT;
    notification_id INTEGER;
BEGIN
    SELECT name INTO requester_name
    FROM "user"
    WHERE id = NEW.client_id;

    INSERT INTO notification (content, viewed, user_id)
    VALUES (requester_name || ' has requested to join your group', FALSE, NEW.group_id);

    SELECT currval('notification_id_seq') INTO notification_id;

    INSERT INTO request_notification (id)
    VALUES (notification_id);

    INSERT INTO join_group_notification (id, client_id, group_id)
    VALUES (notification_id, NEW.client_id, NEW.group_id);

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_request_join
AFTER INSERT ON request_join
FOR EACH ROW
EXECUTE PROCEDURE create_join_group_notification();


CREATE FUNCTION create_like_comment_notification() RETURNS TRIGGER AS $$
DECLARE
    liker_name TEXT;
    notification_id INTEGER;
BEGIN
    SELECT name INTO liker_name
    FROM "user"
    WHERE id = NEW.user_id;

    INSERT INTO notification (content, viewed, user_id)
    VALUES (liker_name || ' liked your comment', FALSE, (SELECT user_id FROM comment WHERE id = NEW.comment_id));

    SELECT currval('notification_id_seq') INTO notification_id;

    INSERT INTO like_notification (id)
    VALUES (notification_id);

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

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_like_comment
AFTER INSERT ON like_comment
FOR EACH ROW
EXECUTE PROCEDURE create_like_comment_notification();

CREATE FUNCTION create_like_post_notification() RETURNS TRIGGER AS $$
DECLARE
    liker_name TEXT;
    notification_id INTEGER;
    post_owner_id INTEGER;
BEGIN
    SELECT name INTO liker_name
    FROM "user"
    WHERE id = NEW.user_id;

    IF EXISTS (SELECT 1 FROM review_post WHERE id = NEW.post_id) THEN
        SELECT client_id INTO post_owner_id
        FROM review_post
        WHERE id = NEW.post_id;
    ELSE
        SELECT client_id INTO post_owner_id
        FROM informational_post
        WHERE id = NEW.post_id;
    END IF;

    INSERT INTO notification (content, viewed, user_id)
    VALUES (liker_name || ' liked your post', FALSE, post_owner_id);

    SELECT currval('notification_id_seq') INTO notification_id;

    INSERT INTO like_notification (id)
    VALUES (notification_id);

    INSERT INTO like_post_notification (id, user_id, post_id)
    VALUES (notification_id, NEW.user_id, NEW.post_id);

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_like_post
AFTER INSERT ON like_post
FOR EACH ROW
EXECUTE PROCEDURE create_like_post_notification();


CREATE FUNCTION create_group_notification_after_post() RETURNS TRIGGER AS $$
DECLARE
    group_name TEXT;
    member_id INTEGER;
    notification_id INTEGER;
BEGIN
    SELECT name INTO group_name
    FROM "group"
    WHERE id = NEW.group_id;

    FOR member_id IN
        SELECT client_id
        FROM group_member
        WHERE group_id = NEW.group_id
    LOOP
        INSERT INTO notification (content, viewed, user_id)
        VALUES ('A post has been made in the group ' || group_name, FALSE, member_id);

        SELECT currval('notification_id_seq') INTO notification_id;

        INSERT INTO general_notification (id)
        VALUES (notification_id);

        INSERT INTO group_notification (id, group_id)
        VALUES (notification_id, NEW.group_id);
    END LOOP;

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_review_post
AFTER INSERT ON review_post
FOR EACH ROW
EXECUTE PROCEDURE create_group_notification_after_post();


CREATE FUNCTION create_group_notification_after_join() RETURNS TRIGGER AS $$
DECLARE
    group_name TEXT;
    member_id INTEGER;
    notification_id INTEGER;
BEGIN
    SELECT name INTO group_name
    FROM "group"
    WHERE id = NEW.group_id;

    FOR member_id IN
        SELECT client_id
        FROM group_member
        WHERE group_id = NEW.group_id
    LOOP
```

```
        INSERT INTO notification (content, viewed, user_id)
        VALUES ('A new member has joined the group ' || group_name, FALSE, member_id);

        SELECT currval('notification_id_seq') INTO notification_id;

        INSERT INTO general_notification (id)
        VALUES (notification_id);

        INSERT INTO group_notification (id, group_id)
        VALUES (notification_id, NEW.group_id);
    END LOOP;

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_groupmember
AFTER INSERT ON group_member
FOR EACH ROW
EXECUTE PROCEDURE create_group_notification_after_join();


CREATE FUNCTION create_group_notification_after_leave() RETURNS TRIGGER AS $$
DECLARE
    group_name TEXT;
    member_id INTEGER;
    notification_id INTEGER;
BEGIN
    SELECT name INTO group_name
    FROM "group"
    WHERE id = OLD.group_id;

    FOR member_id IN
        SELECT client_id
        FROM group_member
        WHERE group_id = OLD.group_id
    LOOP
        INSERT INTO notification (content, viewed, user_id)
        VALUES ('A member has left the group ' || group_name, FALSE, member_id);

        SELECT currval('notification_id_seq') INTO notification_id;

        INSERT INTO general_notification (id)
        VALUES (notification_id);

        INSERT INTO group_notification (id, group_id)
        VALUES (notification_id, OLD.group_id);
    END LOOP;

    RETURN OLD;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_delete_groupmember
AFTER DELETE ON group_member
FOR EACH ROW
EXECUTE PROCEDURE create_group_notification_after_leave();
```

A.2. Database population

> Only a sample of the database population script may be included here, e.g. the first 10 lines. The full script must be available in the repository.

```
INSERT INTO "user" (name, username, password, description, email, image, is_blocked, is_admin)
VALUES
```

```
    ('Daniel Teixeira', 'dteixeira', 'teixeira10', 'Food lover', 'dteixeira@example.com', 'dteixeira.jpg', FALSE, FALSE
    ('Alice Smith', 'alice123', 'password1', 'Loves cooking', 'alice@example.com', 'alice.jpg', FALSE, FALSE),
    ('Bob Brown', 'bobbie', 'password2', 'Food blogger', 'bob@example.com', 'bob.jpg', FALSE, FALSE),
    ('Charlie Chef', 'chefcharlie', 'password3', 'Master chef', 'charlie@example.com', 'charlie.jpg', FALSE, TRUE),
    ('Daniel Delgado', 'daniel_foodie', 'password4', 'Food lover and critic', 'daniel@example.com', 'daniel.jpg', FALSE
    ('Emily Evans', 'emily_eats', 'password5', 'Healthy eating advocate', 'emily@example.com', 'emily.jpg', FALSE, FALS
    ('Fiona French', 'fiona_fries', 'password6', 'Blogger specializing in street food', 'fiona@example.com', 'fiona.jpg
    ('George Gourmet', 'george_gourmet', 'password7', 'Fine dining enthusiast', 'george@example.com', 'george.jpg', FAL
    ('Hannah Hernandez', 'hannah_harvest', 'password8', 'Vegan chef and recipe developer', 'hannah@example.com', 'hanna
    ('Isaac Irons', 'isaac_international', 'password9', 'Traveling chef exploring international cuisines', 'isaac@examp
    ('Jasmine Jones', 'jasmine_jazzy', 'password10', 'Enthusiast of Mediterranean flavors', 'jasmine@example.com', 'jas
    ('Kevin Kim', 'kimchi_king', 'password11', 'Korean BBQ expert', 'kevin@example.com', 'kevin.jpg', FALSE, FALSE),
    ('Linda Liu', 'linda_luvs_food', 'password12', 'Passionate about Asian cuisine', 'linda@example.com', 'linda.jpg',
    ('Mark Mendoza', 'mark_meat', 'password13', 'Steakhouse owner and meat specialist', 'mark@example.com', 'mark.jpg',
    ('wagner','wagner','wagner','admin','wagner@gmail.com','admin.png',FALSE,TRUE),
    ('nelson','nelson','nelson','admin','nelson@gmail.com','admin.png',FALSE,TRUE),
    ('sara','sara','sara','admin','sara@gmail.com','admin.png',FALSE,TRUE),
    ('paulo','paulo','paulo','admin','paulo@gmail.com','admin.png',FALSE,TRUE);
```

Revision history

1. Review and update the user stories and their priorities.

Editor: Nelson Neto

GROUP2432, 3/11/2024

- Paulo Fidalgo, up201806603@fe.up.pt
- Wagner Pedrosa, up201908556@fe.up.pt
- Sara Azevedo, up202006902@fe.up.pt (Editor)
- Nelson Neto, up202108117@fe.up.pt