BILKENT UNIVERSITY

CS 353

Database Systems

DESIGN REPORT

05 April 2019

gatherTogether

By Group 20

Ahmet Talha Şen

Ana Peçini

Endi Merkuri

Krisela Skënderi

Table of Contents

1. Revised E/R Model 3

1.1 Revised Diagram 3

1.2 Changes Made to Model 4

2.Table Schemas 5

3.Functional Components 16

3.1 Use case Diagram 16

3.2 Functionalities 17

3.3 Scenarios 18

3.4 Data Structures 21

4. User interface design and SQL Statements 22

4.1 Log In/Register 22

4.2 Create an event for a specific group 24

4.3 Make a comment for a particular event 29

4.4 Create a new group 29

4.5 Additional Functionality 32

5. Advanced database components 34

5.1 Views 34

5.2 Triggers 35

5.3 Constraints 35

6. Implementation Plan 35

1. Revised E/R Model

1.1 Revised Diagram

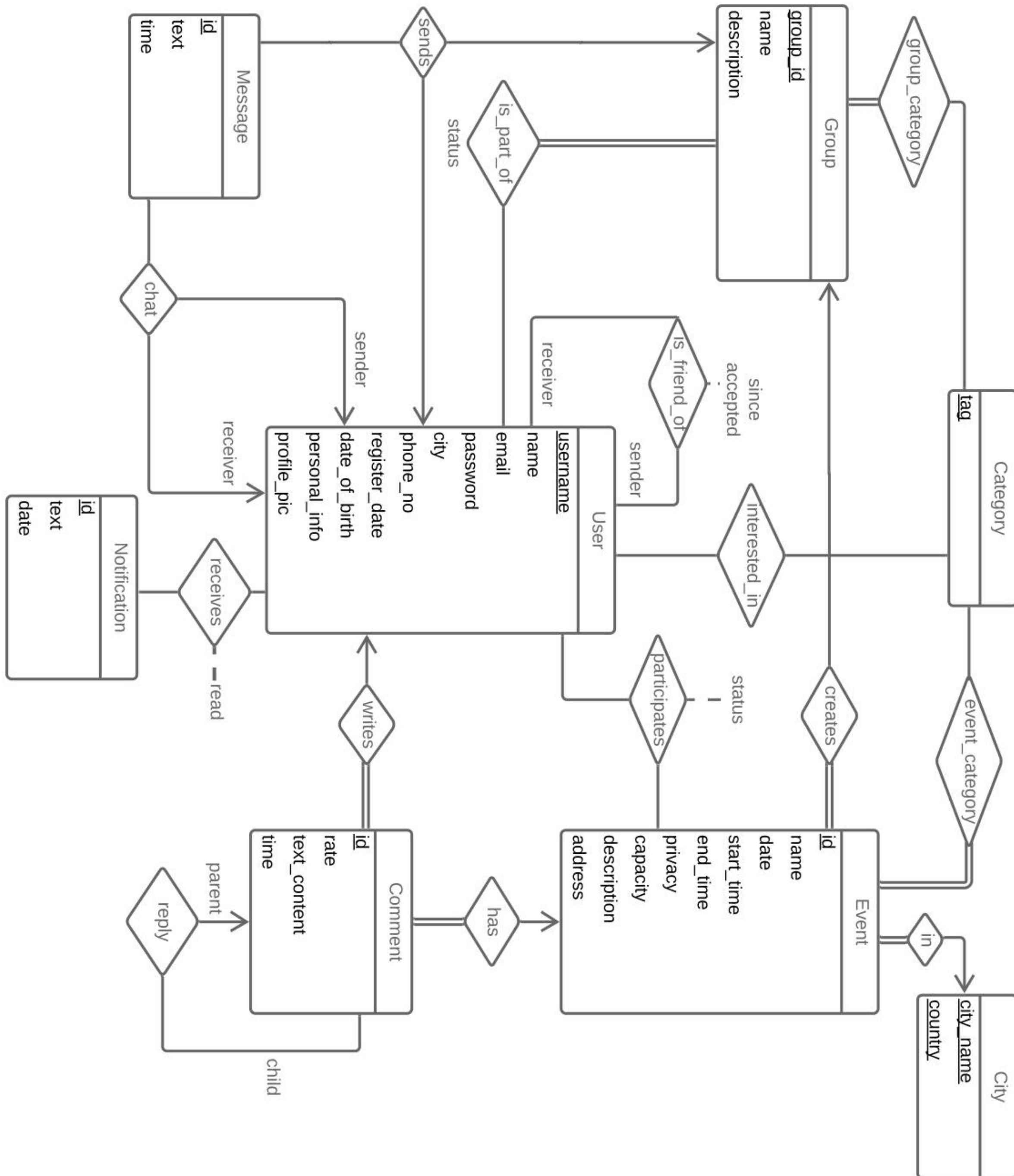

Figure 1 Revised Entity-Relationship Diagram

1.2 Changes made to model

- **"Host"** entity was removed, as well as the aggregation with "Event" entity. Only groups can create events (group admin creates events on behalf of the group).

- A binary relationship **"participates"** with attribute **"status"** was created between "User" and "Event". Status is an enumeration with values: "Going", "Not going", "Interested" and "Banned".

- "is_in" relationship between "Group" and "User" was renamed to **"is_part_of"** and an attribute **"status"** as an enumeration with values: "Creator" ,"Member", "Banned", "Invited", "Requesting".

- **"City"** entity was added. Every "Event" is associated with only one city through **"in"** relationship.

- **"Message"** entity was created with attributes id, text and time. It is related to "Group" and "User" by a ternary relationship **"sends"** to represent group messages. It is also related to "User" with a ternary relation **"chat"** to represent messages between users, where the roles are sender and receiver.

-**"Comment"** entity was created. It is related to "Event" by the binary relationship **"has"** and to "User" by the binary relationship **"writes"**. There is also a binary relationship between two comments named **"reply"** with cardinality one-to-many and roles "parent" and "child".

- An attribute **"since"** was added to "is_friend_of" relationship between users.

- **"Photo"** entity was removed.

- "Notification" entity created and related to "User" by **"receives"** whose cardinality is many-to-many because the same notification may be sent to every member of the group (ex. when an event is created).

2. Table Schemas

In this section we provide the details on the tables we are going to use in our project. To connect to JDBC, at the end of each table definition we will include " ENGINE = InnoDB; " during the implementation (not included now for brevity). Additionally, in case the application will be rerun, the existing tables will also be dropped before creation.

2.1 User

- **Relational Model:**

  user(<u>username</u>, name, email, password, city, phone_no, date_of_birth, register_date, personal_info, profile_pic)

- **Functional Dependencies:**

  username → name, email, password, city, phone_no, date_of_birth, register_date, personal_info, profile_pic

  email → username

- **Candidate Keys:**

  {username}, {email}

- **Normal Form:**

  BCNF

- **Table Definition**:

```
CREATE TABLE user (
        username VARCHAR(50) PRIMARY KEY,
        name VARCHAR(50) NOT NULL,
        email VARCHAR(50) NOT NULL UNIQUE,
        password VARCHAR(50) NOT NULL,
        city VARCHAR(50),
        phone_no CHAR(10),
        date_of_birth DATE,
        register_date DATE DEFAULT CURRENT_DATE,
        personal_info TEXT,
        profile_pic TEXT,
        CHECK(email LIKE '_%@__%._%')    );
```

2.2 Group

- **Relational Model:**

  group(<u>id</u>, name, description)

- **Functional Dependencies:**

  id → name, description

  name → id

- **Candidate Keys:**

  {id}, {name}

- **Normal Form:**

  BCNF

- **Table Definition**:

```
CREATE TABLE group (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE,
    description TEXT
);
```

2.3 Event

- **Relational Model:**

  event(id, name, date, start_time, end_time, privacy, capacity, group_id, description, address, city_name, country)

- **Functional Dependencies:**

  id → name, date, start_time, end_time, privacy, capacity, group_id, description,

  address, city_name, country

- **Candidate Keys:**

  {id}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE event (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    date DATE NOT NULL,
    start_time TIMESTAMP NOT NULL,
    end_time TIMESTAMP NOT NULL,
    privacy BOOLEAN DEFAULT FALSE,
    capacity INT NOT NULL,
    group_id INT NOT NULL,
    description TEXT,
    address TEXT NOT NULL,
    city_name VARCHAR(30) NOT NULL,
    country VARCHAR(30) NOT NULL,
    FOREIGN KEY (group_id) REFERENCES group(id) ON DELETE CASCADE,
    FOREIGN KEY (city_name, country) REFERENCES city   );
```

2.4 Category

- **Relational Model:**

  category(<u>tag</u>)

- **Functional Dependencies:**

  tag $\rightarrow$ tag

- **Candidate Keys:**

  {tag}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE category (
```

tag VARCHAR(20) PRIMARY KEY);

2.5 Participates

- **Relational Model:**

  participates (<u>event_id, username</u>, status)

- **Functional Dependencies:**

  event_id, username → status

- **Candidate Keys:**

  {event_id, username}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE participates (
        event_id INT,
        username VARCHAR(50),
        status stat2 DEFAULT 'not going',
        PRIMARY KEY (event_id, username),
        FOREIGN KEY (event_id) REFERENCES event(id) ON DELETE CASCADE,
        FOREIGN KEY (username) REFERENCES user (username) ON DELETE CASCADE);
```

2.6 Receives

- **Relational Model:**

  receives (<u>notification_id, username</u>, read)

- **Functional Dependencies:**

  notification_id, username → read

- **Candidate Keys:**

  {notification_id, username}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE receives (
        notification_id INT,
        username VARCHAR(50),
```

```
        read BOOLEAN DEFAULT FALSE,
        PRIMARY KEY (notification_id, username),
        FOREIGN KEY (notification_id) REFERENCES notification(id),
        FOREIGN KEY (username) REFERENCES user(username) ON DELETE CASCADE
);
```

2.7 Notification

- **Relational Model:**

  notification (id, text, date)

- **Functional Dependencies:**

  id → text, date

- **Candidate Keys:**

  {id}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE notification (
        id SERIAL PRIMARY KEY,
        text TEXT,
        date TIMESTAMP DEFAULT NOW()
);
```

2.8 City

- **Relational Model:**

  city (city_name, country)

- **Functional Dependencies**

  city_name, country → city_name, country

- **Candidate Keys**

  {city_name, country }

- **Normal Form**

  BCNF

- **Table Definition**

```
CREATE TABLE city (
        city_name VARCHAR(30),
        country VARCHAR(30),
PRIMARY KEY (city_name, country)
);
```

2.9 is_friend_of

- **Relational Model:**

  is_friend_of (sender, receiver, accepted, since)

- **Functional Dependencies:**

  sender, receiver → accepted, since

- **Candidate Keys:**

  {sender, receiver}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE is_friend_of (
        sender VARCHAR(50),
        receiver VARCHAR(50),
        accepted BOOLEAN DEFAULT FALSE,
        since TIMESTAMP NOT NULL,
        PRIMARY KEY (sender, receiver),
        FOREIGN KEY (sender) REFERENCES user(username) ON DELETE CASCADE,
        FOREIGN KEY (receiver) REFERENCES user(username) ON DELETE CASCADE
);
```

2.10 is_part_of

- **Relational Model:**

  is_part_of (username, group_id, status)

- **Functional Dependencies:**

  username, group_id→ status

- **Candidate Keys:**

{username, group_id}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TYPE stat AS ENUM ('admin', 'member', 'invited', 'requested', 'banned');
CREATE TABLE is_part_of (
        username VARCHAR(50),
        group_id INT,
        status stat NOT NULL,
        PRIMARY KEY (username, group_id),
        FOREIGN KEY (username) REFERENCES user(username) ON DELETE CASCADE,
        FOREIGN KEY (group_id) REFERENCES group(id) ON DELETE CASCADE   );
```

2.11 event_category

- **Relational Model:**

  event_category (event_id, tag)

- **Functional Dependencies:**

  event_id, tag → event_id, tag

- **Candidate Keys:**

  {event_id, tag}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE event_category (
        event_id INT,
        tag VARCHAR(20),
        PRIMARY KEY (event_id, tag),
        FOREIGN KEY (event_id) REFERENCES event(id) ON DELETE CASCADE,
        FOREIGN KEY (tag) REFERENCES category   );
```

2.12 interested_in

- **Relational Model:**

  interested_in (<u>username, tag</u>)

- **Functional Dependencies:**

  username, tag → username, tag

- **Candidate Keys:**

  {username, tag}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE interested_in (
    username VARCHAR(50),
    tag VARCHAR(20),
    PRIMARY KEY (username, tag),
    FOREIGN KEY (username) REFERENCES user(username) ON DELETE CASCADE,
    FOREIGN KEY (tag) REFERENCES category
);
```

2.13 group_category

- **Relational Model:**

  group_category (<u>group_id, tag</u>)

- **Functional Dependencies:**

  group_id, tag → group_id, tag

- **Candidate Keys:**

  {group_id, tag}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE group_category (
    group_id INT,
    tag VARCHAR(20),
    PRIMARY KEY (group_id, tag),
```

FOREIGN KEY (group_id) REFERENCES group(id) ON DELETE CASCADE,

FOREIGN KEY (tag) REFERENCES category);

2.14 sends

- **Relational Model:**

  sends (message_id, sender, group_id)

- **Functional Dependencies:**

  message_id → sender, group_id

- **Candidate Keys:**

  {message_id}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE sends (
    message_id INT PRIMARY KEY,
    sender VARCHAR(50) NOT NULL,
    group_id INT NOT NULL,
    FOREIGN KEY (group_id) REFERENCES group(id),
    FOREIGN KEY (message_id) REFERENCES message(id),
    FOREIGN KEY (sender) REFERENCES user(username) );
```

2.15 chat

- **Relational Model:**

  chat (message_id, sender, receiver)

- **Functional Dependencies:**

  message_id → sender, receiver

- **Candidate Keys:**

  {message_id}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE chat (
```

```
    message_id INT PRIMARY KEY,
    sender VARCHAR(50) NOT NULL,
    receiver VARCHAR(50) NOT NULL,
    FOREIGN KEY (message_id) REFERENCES message(id),
    FOREIGN KEY (sender) REFERENCES user(username),
    FOREIGN KEY (receiver) REFERENCES user(username)
);
```

2.16 message

- Relational Model:

  message (<u>id</u>, text, time)

- Functional Dependencies

  id → text, time

- Candidate Keys

  {id}

- Normal Form

  BCNF

- Table Definition:

```
CREATE TABLE message (
    id SERIAL PRIMARY KEY,
    text TEXT NOT NULL,
    time TIMESTAMP DEFAULT NOW()
);
```

2.17 reply

- Relational Model:

  reply (<u>child</u>, parent)

- Functional Dependencies:

  child → parent

- Candidate Keys:

  {child}

- Normal Form:

  BCNF

- **Table Definition:**

```
CREATE TABLE reply (
    child INT PRIMARY KEY,
    parent INT,
    FOREIGN KEY (parent) REFERENCES comment(id) ON DELETE CASCADE,
    FOREIGN KEY (child) REFERENCES comment(id) ON DELETE CASCADE
);
```

2.18 comment

- **Relational Model:**

  comment (id, rate, text_content, time, event_id, username)

- **Functional Dependencies:**

  id → rate, text_content, time, event_id, username

- **Candidate Keys:**

  {id}

- **Normal Form:**

  BCNF

- **Table Definition:**

```
CREATE TABLE comment (
        id SERIAL PRIMARY KEY,
        rate INT,
        text_content TEXT NOT NULL,
        time TIMESTAMP DEFAULT NOW(),
        event_id INT,
        username VARCHAR(50),
        FOREIGN KEY (event_id) REFERENCES event(id) ON DELETE CASCADE,
        FOREIGN KEY (username) REFERENCES "user"(username) ON DELETE SET NULL,
        CHECK(rate >= 0 AND rate <= 5)     );
```

3. Functional Components

3.1 Use Case Diagram

Figure 2 Use Case Diagram for gatherTogether system

3.2 Functionalities

3.2.1 User Functionalities

- Users should be able to search for events by location, date and categories.
- Users must be able to request membership to groups.
- Users must decide whether or not to accept a group invitation.
- Users can participate in many groups at the same time.
- Users should be able to create groups (become group admins).

3.2.2 Group Member Functionalities

- Group members should be able to access information about groups' past events.
- Group members can decide whether or not to participate in a group event.
- Group members can post comments (and also a rate) about an event after attending it, as well as reply to all comments and replies.
- Group members can exchange messages in the group chat.
- Group members should be able to leave the group.

3.2.3 Group Admin Functionalities

- Group admins can manage group properties such as name and categories.
- Group admins should be able to create events on behalf of the group.
- Group admins can manage group members: handle requests for membership; send invitations to users by searching through location and tags/categories and remove/ban users from the group or from the events of the group.
- Group admins should be able to send notifications to group members about new events.

3.3 Scenarios

Below are the detailed scenarios for the main functionalities of our system.

Log In	
Participating actors:	User
Pre-condition:	User has opened the GatherTogether webpage.

Post-condition:	User is logged in to his account and has access to all the functionalities of the system.
Flow of events:	1. User clicks on Log In button. 2. System prompts User to enter log in information. 3. User provides username and password. 4. System checks the information. 4.1 If information is not correct, steps 2-4 are repeated. 5. User is logged in.
Alternative Flow of events:	1. If User is not registered in the system and does not have an account, User presses Register Button. 2. System prompts user to enter a username and password and confirm the password. 2.1 If username already exists, step 2 is repeated. 3. User is logged in.

Create an event for a particular group	
Participating actors:	Admin
Pre-condition:	1. Event information is already discussed in the group chat and admin is creating the event on behalf of the group. 2. Admin is logged in the system.
Post-condition:	1. Event is created and a notification is sent to each member of the group. 2. The status of each member is initialized to “Not going”.
Flow of events:	1. Admin presses Create New Event from the page of the group to whom the event will belong. 2. System prompts Admin to enter event name, address, capacity, categories, private/public and an optional description. 3. Admin confirms the creation of event. 4. System displays a message to indicate whether event was successfully created. 4.1 If not successful, steps 2-4 are repeated. 5. System notifies group members if event was created.

Make a comment for a particular event	
Participating actors:	Group member (referred below as User).
Pre-condition:	User found the event he wants to comment on in the group page.
Post-condition:	1. The comment is displayed just below the event information. 2. Other users are able to reply to the comment.
Flow of events:	1. User opens the Event page.

	2. User writes the comment in the box that is displayed. 2.1 User can additionally choose a rate for the event (stars). 3. User submits comments. 4. System displays comment as the top comment (most recent one) for the moment, just below the event information. 5. User can similarly comment/reply on other comments for the event, by clicking on the reply button below a comment. 5.1 System notifies the writer of the original comment about the reply.

Create a group	
Participating actors:	User
Pre-condition:	1. User is logged in the system.
Post-condition:	1. Group is created with a unique group id. 2. Creator of the group is automatically an admin for that group and has all the admin privileges.
Flow of events:	1. User presses Create New Group in his main page. 2. System prompts User to enter group name, categories and an optional description. 3. User enters information. 4. User confirms the creation of the group. 5. System displays a message to indicate whether group was successfully created. 5.1 If not successful, steps 2-4 are repeated. 6. User automatically becomes an admin for the group.

Send messages in the group chat	
Participating actors:	Group member (referred below as User)
Pre-condition:	User has opened the group page.
Post-condition:	The message is displayed in the group chat together with the username of the sender and the time.
Flow of events:	1. User opens the page of the group in which he wants to send the message. 2. User types the message he would like to send in the chat box. 3. User presses send message.

Send a message to another user	
Participating actors:	User
Pre-condition:	User has opened the profile page of the user he would like to message/recipient.
Post-condition:	The messages are stored for both the sender and the user and can be accessed from their profile pages.
Flow of events:	1. User presses on the message icon in the other user's profile page. 2. System displays a pop-up. 3. User writes the message and chooses the Send option. 4. System notifies the recipient about the new message.

3.4 Data Structures

We will be using 5 types of domain types for the schemas in our database: Numeric type, Text Type, Boolean Type, Enumerated Type and Date type:

- Numeric types **(INT)** will be used for attributes such as id, phone number and rate.

Note: For ids the keyword SERIAL is used for autogeneration.

- Text types **(VARCHAR)** will be used for names, addresses (city, country, street), email, text, description.

- **BOOLEAN** type will be used for flag/boolean attributes such as accepted in "is_friend_of" relationship to show the outcome of a request.

- **ENUMERATED (ENUM)** type will be used to indicate the 'status' attribute of the "is_part_of" relationship between "User" and "Group". It will take values from ('admin', 'member', 'invited', 'requested', 'banned').

Note: In PostgreSQL enums are created as Types using CREATE TYPE command.

- Built-in types for dates: **DATE** type will be used for birthday, creation date, registration date; and **TIMESTAMP** type for event date, message time and comment times.

4. User Interface design and SQL Statements

4.1 Log In and Register

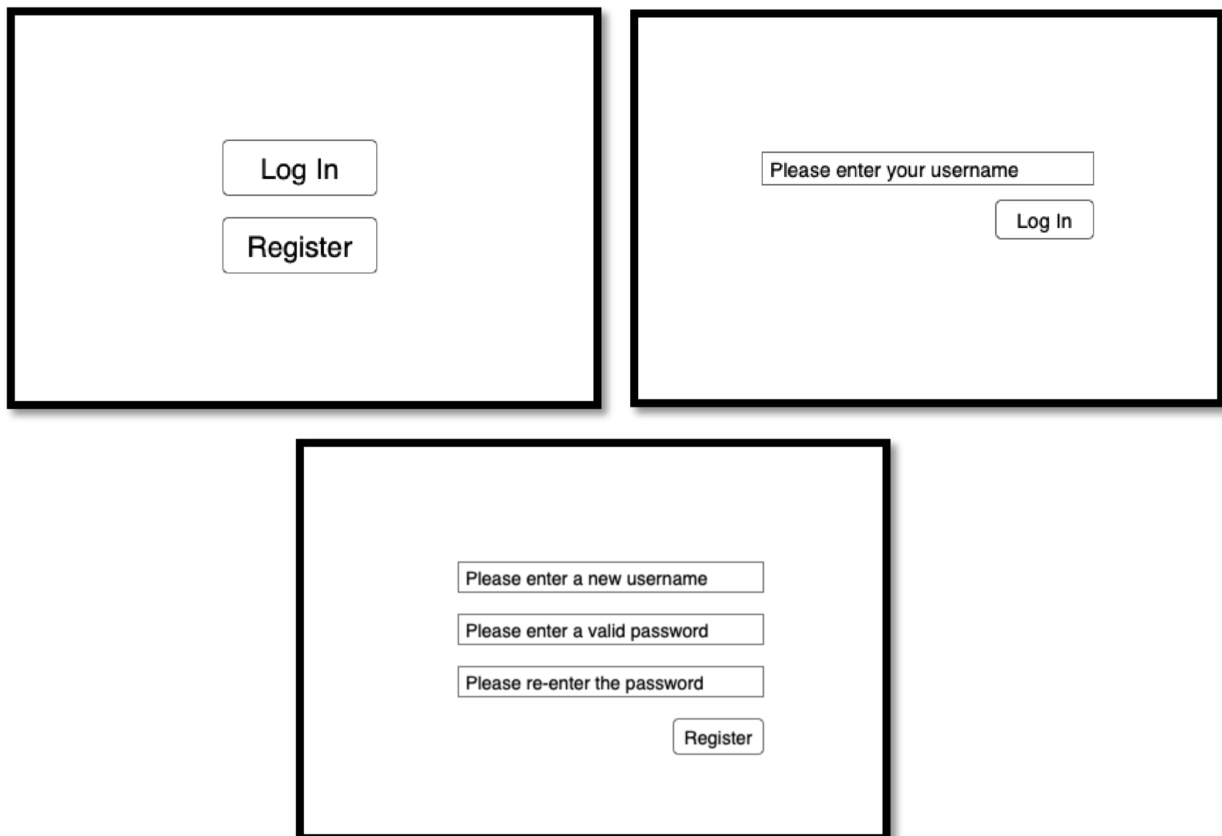

Figure 3. Log In and Register Screens

REGISTER:
Inputs: @username, @password

```sql
INSERT INTO user (username, password)
VALUES (@username, @password);
```

LOG IN:
Inputs: @username, @password

```sql
SELECT username
FROM user U
WHERE U.username = @username AND U.password = @password;
```

Update User information

After creating a new account User can update his profile information at any time in his profile page.

Figure 4. Mockup of user's profile page

Inputs: @username, @name, @email, @city, @date_of_birth, @phone_no,
@personal_info, @profile_pic, @interests (this is an array input handled by JDBC)

```
UPDATE user
SET name = @name, email = @email, city = @city, date_of_birth =
    @date_of_birth, phone_no = @phone_no, profile_pic = @profile_pic
WHERE username = @username;
```

```
-- Updating interested_in table to store the interests of the new user
CREATE TRIGGER add_user_interests
AFTER INSERT ON user
REFERENCING NEW ROW AS nrow
FOR EACH ROW
BEGIN
      FOREACH elem IN ARRAY @interest
      LOOP
            INSERT INTO interested_in
            VALUES (nrow.id, elem)
      END LOOP;
END;
```

4.2 Create an event for a particular group

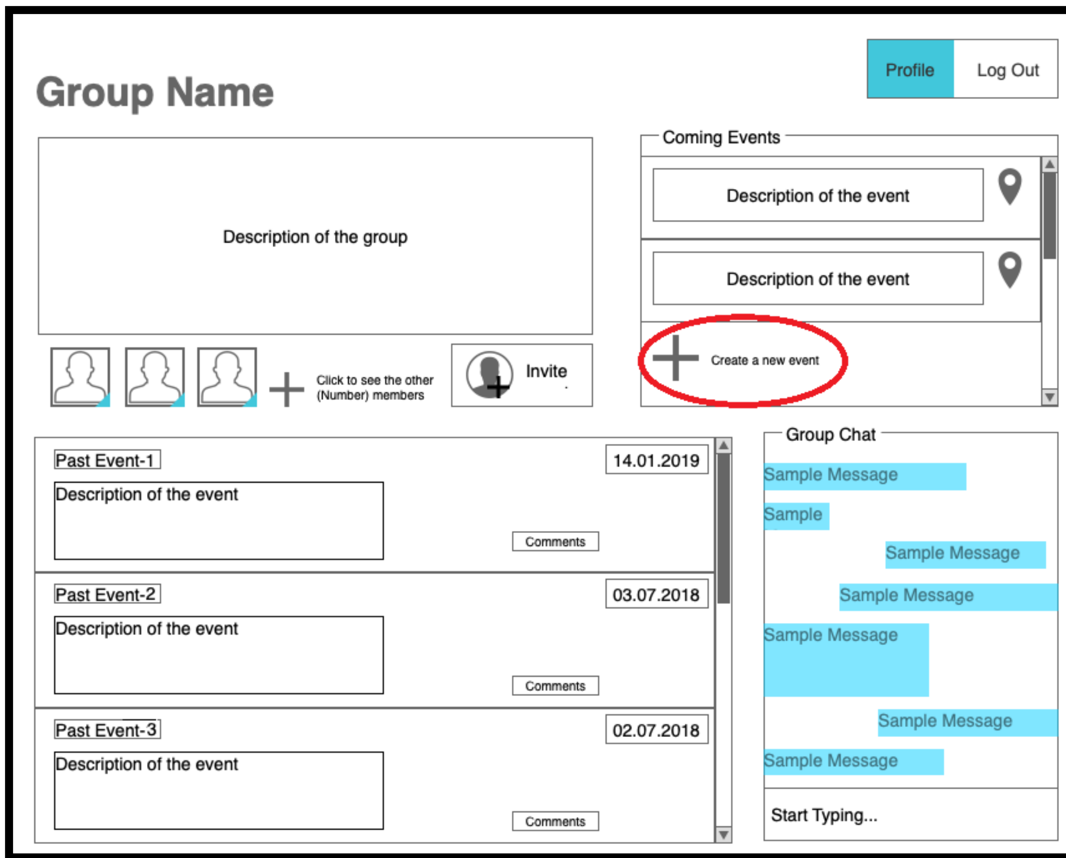

Figure 5. Group page from group admin's account

Name of the event

Starting time: 08:00 End time: 08:00

Private Event

Max capacity

Address...

Description...

Create

Figure 6. Creating an event screen for group admin.

Group admin can create a new event for a group by going to that group's page and pressing the Create a New Event from the upper right part of the page (Figure 5). He is directed into a new page (Figure 6). The information about the event can be accessed through the group's page (Figure 5) by every group member (as displayed in Figure 8) and admin (Figure 9).

Inputs: @name, @date, @startTime, @endTime, @privacy, @capacity, @group_id, @description, @address, @city_name, @country, @categories (array input)

```
WITH new_event( event_id, group_id) AS (
 INSERT INTO event (name, date, start_time, end_time, privacy,
      capacity, group_id, description, address, city_name, country)
 VALUES ( @name, @date, @startTime, @end_time, @privacy,
     @capacity, @group_id, @description, @address, @city_name, @country)
 RETURNING id, group_id
)

INSERT INTO participates (event_id, username)
   SELECT event_id, username
   FROM new_event NATURAL JOIN is_part_of
        WHERE status IN ('admin', 'member');

-- Updating event_category table to store the categories of the new event
CREATE TRIGGER add_event_categories
AFTER INSERT ON event
REFERENCING NEW ROW AS nrow
FOR EACH ROW
BEGIN
     FOREACH elem IN ARRAY @categories
     LOOP
          INSERT INTO event_category
          VALUES (nrow.id, elem)
     END LOOP;
END;
```

Member giving a decision about joining an event.

A group member may change his status of participation when he receives the notification. The default status on creation is “not going”. On updating the decision, a trigger is invoked to check whether the capacity for the event is reached.

Inputs: @username, @decision, @event_id

```
UPDATE participates
SET   status = @decision
WHERE username = @username AND event_id = @event_id
       AND status <> “banned”
```

Listing past events for a particular group.

We extract the name, date and description for the event so that it can be displayed in the group page as shown in Figure 5 in the left bottom section and Figure 7.

Inputs: @group_id

```
SELECT E.name, E.date, E.description
FROM event E
WHERE E.group_id = @group_id AND E.start_time < NOW();
```

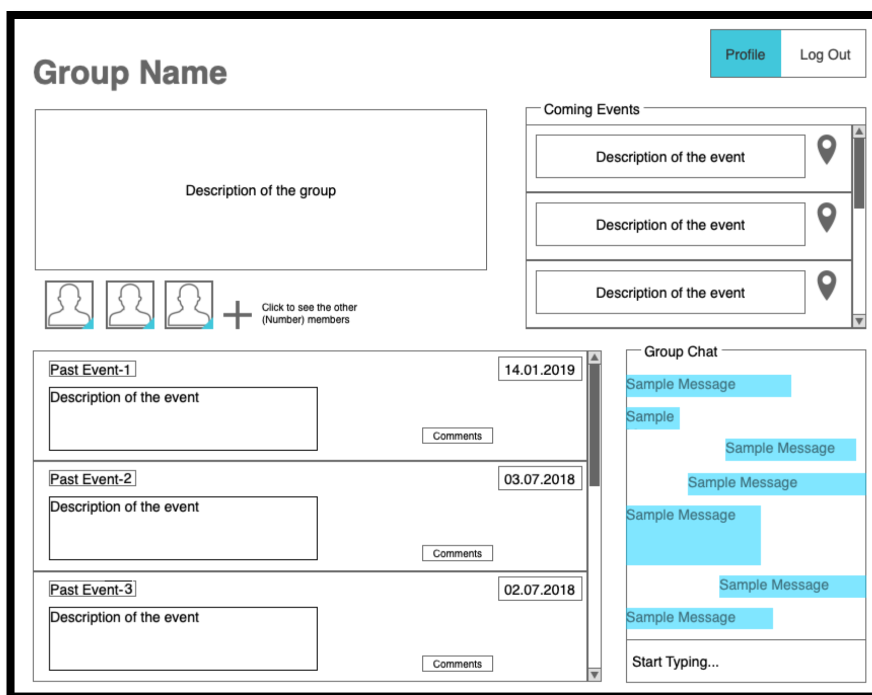

Figure 7. Group page as displayed to a group member

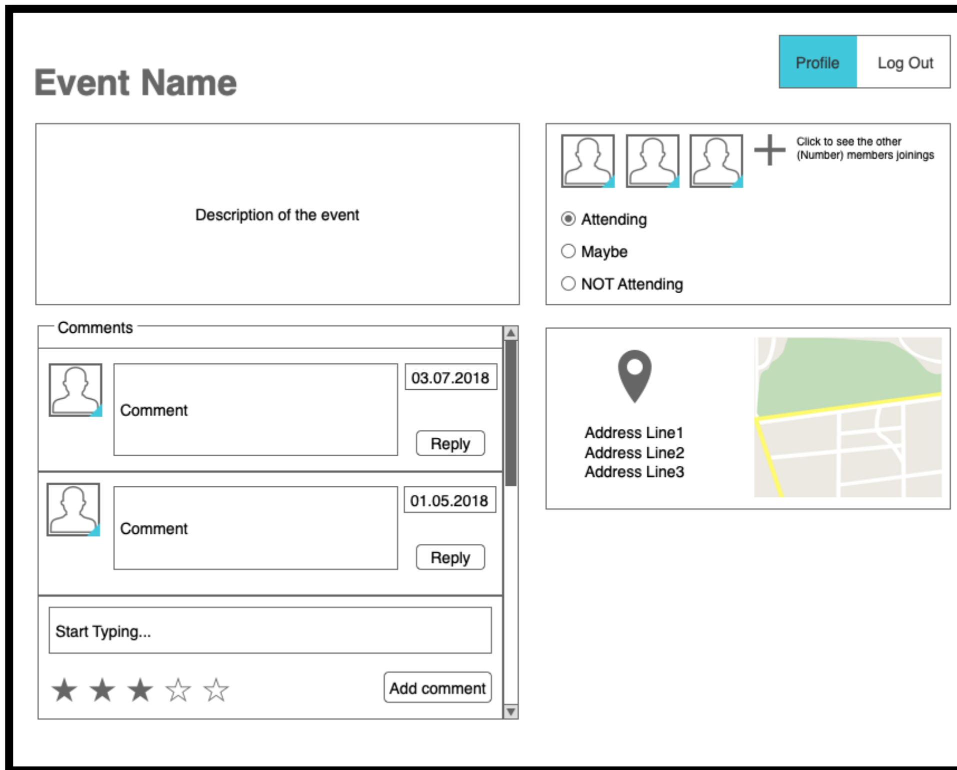

Figure 8. Event information from a group member's account

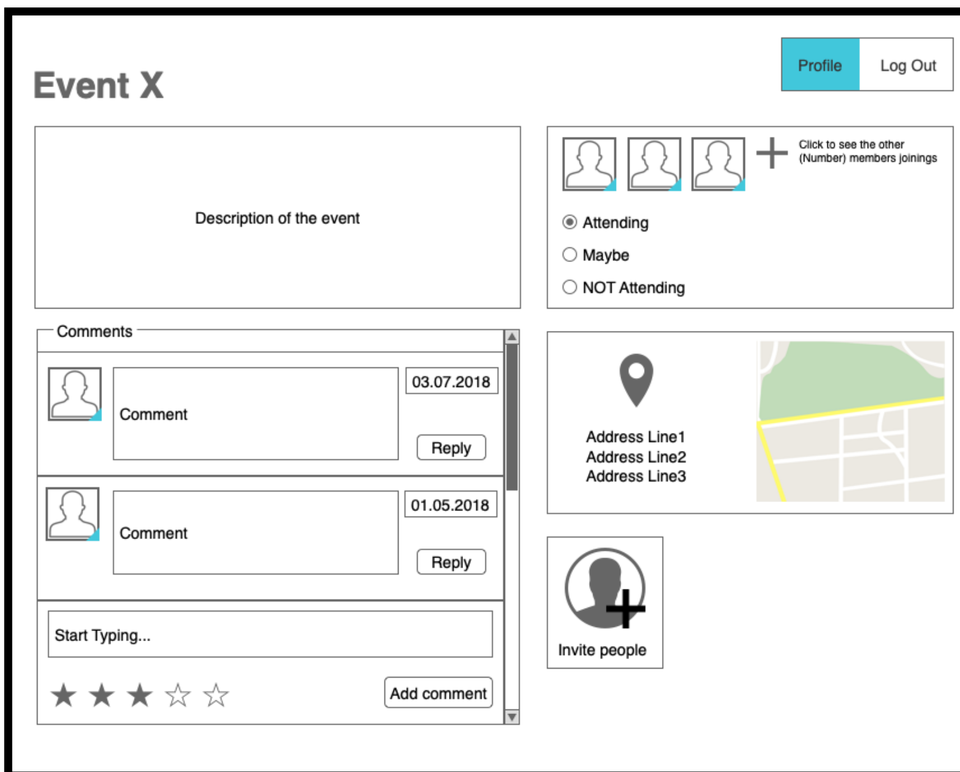

Figure 9. Event Information from group admin's account. Ability to invite people is added.

4.3 Make a comment for a particular event

As seen in Figure 8 and Figure 9 members of a group and admins of the group can make comments about an event of the group by accessing that event through the group's page. It is also possible to reply to comments. Comments are ordered and displayed according to the date and time.

Make a comment to an event:
Inputs: @rate, @text, @event_id, @author

```
INSERT INTO comment (rate, text_content, event_id, username)
VALUES (@rate, @text, @event_id, @author);
```

Make a comment to a comment:
Inputs: @oldCommentId, @rate, @text, @event_id, @author

```
WITH upd AS (
      INSERT INTO comment (rate, text_content, event_id, username)
      VALUES (@rate, @text, @event_id, @author)
      RETURNING id
)
INSERT INTO reply(child, parent) SELECT id, @oldCommentId FROM upd;
```

4.4 Create a new group

Every user can create a new group by specifying its attributes, such as name, description and categories. The user that creates a group automatically becomes the admin of that group.

Inputs: @name, @description, @adminId, @categories (array input)

```
WITH temp AS (
      INSERT INTO group(name, description)
      VALUES (@name, @description)
      RETURNING id
)
INSERT INTO is_part_of(username, group_id, status)
SELECT @adminId, id, ‘admin’ FROM temp ;

-- Updating group_category table to store the categories of the new group
CREATE TRIGGER add_group_categories
AFTER INSERT ON group
REFERENCING NEW ROW AS nrow
```

```
FOR EACH ROW
BEGIN
	FOREACH elem IN ARRAY @categories
	LOOP
		INSERT INTO group_category
		VALUES (nrow.id, elem)
	END LOOP;
END;
```

Admin giving decision about accepting/ rejecting a user request to join the group

Inputs: @username, @decision, @group_id

```
UPDATE is_part_of
SET    status = @decision
WHERE  username = @username AND group_id = @group_id
       AND status = “requested”
```

Name of the group

Description...

Interests

Outdoor

Cinema

Chess

Outdoor

Add a new one

Create

Figure 10. Mockup: creation of a new group

Inviting people/ friends to the group.

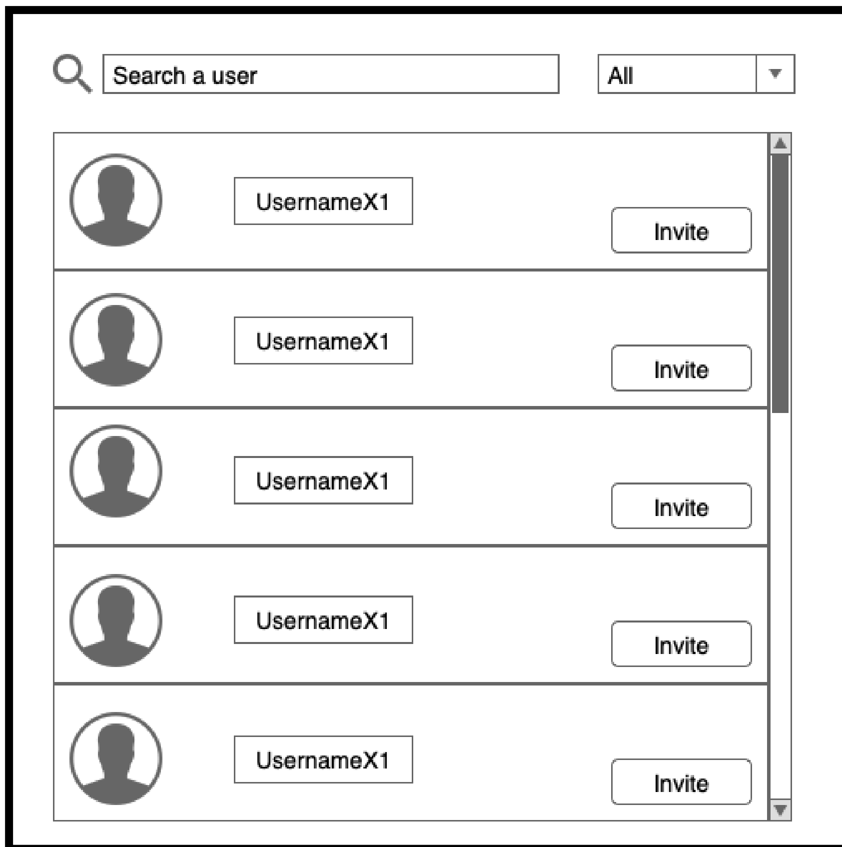

Figure 11. Inviting people to join a group

Inputs: @username, @group_id

```
INSERT INTO is_part_of(username, group_id, status)
VALUES (@username, @group_id, 'invited');
```

4.5 Additional Functionality

4.5.1 Group messages (chat)

A group member can open the page of the group in which he wants to send the message. He types the message in the chat box (Figure 7).

Inputs: @sender, @group_id, @text

```
WITH temp AS (
   INSERT INTO message (text)
   VALUES (@text)
   RETURNING id
)
INSERT INTO sends
SELECT id, @sender, @group_id FROM temp;
```

4.5.2 Messages between users

A user can send a message to any other user by opening the profile page of the recipient. He can press the message icon in the other user's profile page and a pop-up is displayed (Figure 12). User writes the message and chooses the Send option (Figure 13). System notifies the recipient about the new message.

Inputs: @sender, @receiver, @text

```
WITH temp AS (
   INSERT INTO message (text)
   VALUES (@text)
   RETURNING id
)
INSERT INTO chat
SELECT id, @sender, @receiver FROM temp;
```

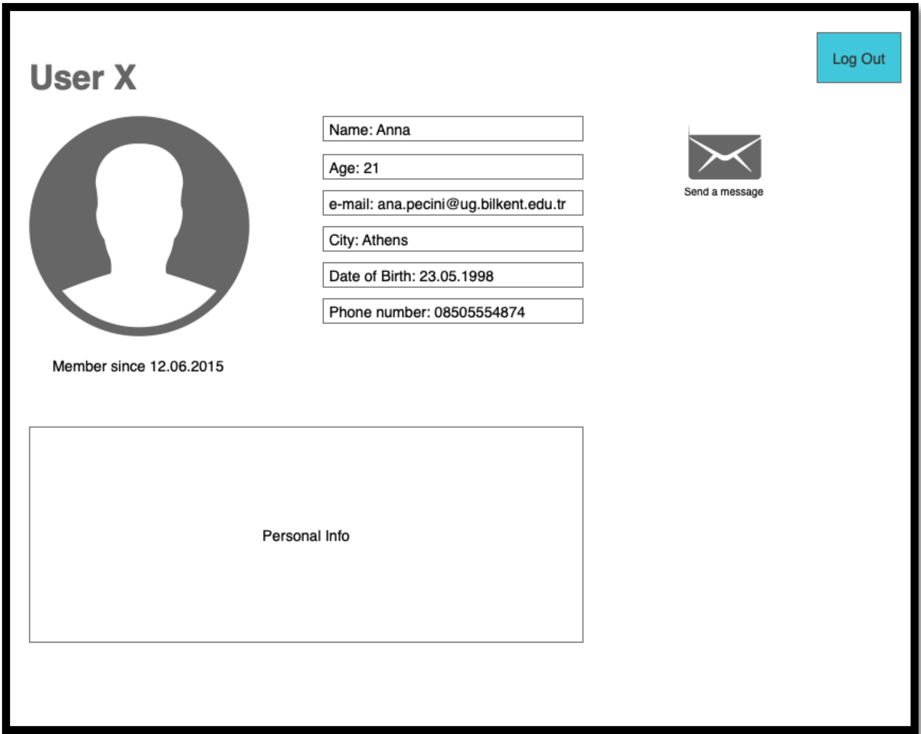

Figure 12. Profile of another user you would like to message

Figure 13. Pop-up after pressing the message icon

5. Advanced database components

In this section we will provide some examples of advanced components that we will be using in the implementation of our system.

5.1 Views

5.1.1 Ten most popular upcoming events of the current month.

The user will be able to see the ten most popular upcoming events of the current month. We consider an event as more popular than another if there are more people “interested” or “going” to that event. The view will have as attributes the ids, names and the number of people interested about the event.

```
CREATE VIEW popularEvents AS

 WITH goingCount( id, count) AS
   (SELECT id, count(*)AS count
    FROM participates p, event e
    WHERE p.event_id = e.id AND EXTRACT( MONTH FROM date)=EXTRACT(MONTH
    FROM CURRENT_DATE) AND EXTRACT(DAY FROM date) > EXTRACT (DAY FROM
    CURRENT_DATE) AND status IN ('interested', 'going')
    GROUP BY id),

    popular( id, name, count) AS
    (SELECT id, name, count
    FROM goingCount NATURAL JOIN event
    ORDER BY count DESC)

 SELECT id AS event_id, name AS event_name, count AS interested
 FROM popular
 LIMIT 10;
```

5.1.1 Most popular groups for each category

The user can request to join groups that are related to his interests. So all of the users will be able to see which are the five most popular groups for each category. The popularity of the groups is the number of members they have. This view will contain the group name and the tag of the category in which it is ranked among the top five. A group will be shown as popular in all of its categories.

```sql
CREATE VIEW popularGroups AS
   WITH temp(group_id, tag, count1) AS
       (SELECT group_id, tag, count(*) AS count1
        FROM is_part_of NATURAL JOIN group_category
        WHERE status = 'member'
        GROUP BY group_id, tag
        ORDER BY tag, count1 DESC)

   SELECT name, tag
   FROM ( SELECT group_id AS id, tag, count1, ROW_NUMBER() OVER
          (PARTITION BY tag) AS row_no
          FROM temp ) t1 NATURAL JOIN group
   WHERE row_no < 6
   ORDER BY tag, count1 DESC;
```

5.2 Triggers

- When a user is invited to join a group, a notification should be generated to show the user his invitation.
- When a group creates an event, all of the users of the group should be notified about the new event.

5.3 Constraints

- The capacity available for an event will be checked before a user is accepted to participate in the event.
- Users must sign up or log in to search for events
- Users should be part of the group in order to participate in an event created by that specific group.

6. Implementation Details

We are considering using PostgreSQL as the DBMS of our project. To implement the business logic we will be working with Java, possibly using JDBC, and PHP will be used for the web service layer.