

GITHUB: <https://github.com/mr2447/neighborhoodproject/tree/main>

PART A – TABLE EXPLANATIONS

This project is to design a relational backend for an online service that allows people to communicate with others in their neighborhoods. For this project, the users can register, login, message other users who are their friends, add neighbors, reply to message in threads, apply to be in a block through a member only voting system. These are some of the functionalities. Below, I will explain the reasons behind how I establish a series of tables to manage a community-based platform where users can register, apply to specific blocks within neighborhoods, and interact through messages and relationships. I am using PostgreSQL, Flask in python to construct this application.

1. Users: This table is the center of my platform system. It stores user's unique SERIAL ID, email, password(hashd for security purpose), and their address.

```
2. CREATE TABLE Users (  
3.     user_id SERIAL PRIMARY KEY,  
4.     username VARCHAR(255) UNIQUE NOT NULL,  
5.     email VARCHAR(255) UNIQUE NOT NULL,  
6.     password_hash VARCHAR(255) NOT NULL,  
7.     address VARCHAR(255) NOT NULL,  
8.     profile_text TEXT,  
9.     photo_url VARCHAR(255)  
10.);
```

Figure 1.1

The application will allow users to create user profile, update user information and delete user profiles. In the next stage of the project, I will have address in a separate table along with coordinates for more precise filtering of messages relating to a certain area of the neighborhood. Upon query 'SELECT * FROM Users', you can see all users, address(to the right, not shown) and hashed password. Figure1.2

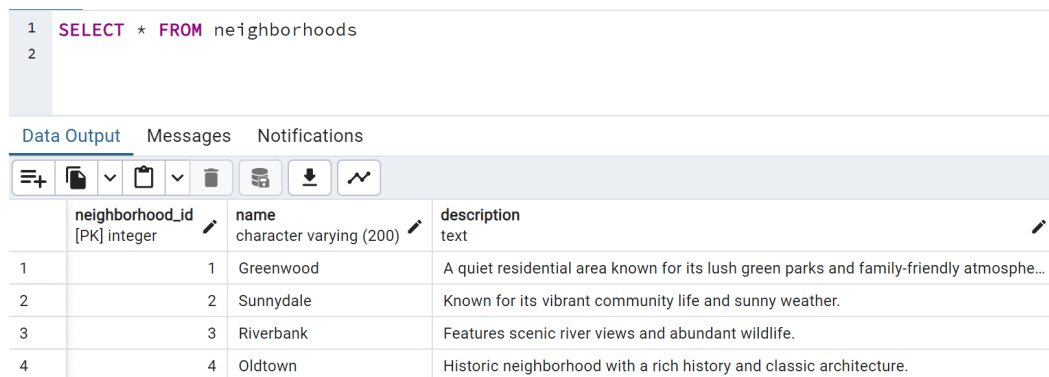
13	13	wearasdzdf	sdfgxs@sdrf.com	scrypt:32768:8:1\$O3Rz220C1437qHW7\$e5d53ca27773fd1d909d86857380099133a2b6419e40816beb917eadfb3c5e13bcb3
14	14	first	first@first.com	scrypt:32768:8:1\$9VFBstn2k4bkST\$03d237be72a40944aba4b8232cb5cf15b6ca5b31ae4c6990653467472a957ad5b6e81e
15	15	test1	test1@test1.com	scrypt:32768:8:1\$FeA095QuVQkgT8FL\$868afc4d36023440f668a6a7ea6b678a99fbc7949cb347568b30b54b6c4664c08d405e
16	16	test2	test2.test2@example.com	scrypt:32768:8:1\$HSitl4Xail9IKXx\$0d04e6bd86d63f480f17cfce770e8052d2864fa5f2eba80921240c9b7a2b5ea6b84a21d63
17	17	test3	test3.test3@example.com	scrypt:32768:8:1\$Hqs904GecM25YQRc\$5a61b6cc5253c343923cd786b951708bb7f76a179514a173f11a5fee08426b41b575b
18	18	test4	test4.test4@example.com	scrypt:32768:8:1\$aJ5g9pNTMK5Vy7qm\$85b27022962fa4e69e0f7788c364a5ae249d2011ccf2b030fd76e7427769459a0c024
19	19	test5	test5.test5@example.com	scrypt:32768:8:1\$P5dF8bZiPSxzf2NUSad7bfc4859792c80919c907f1d9f8b8e33170cbbd5b41860846f9392efefe5471864096
20	20	test6	test6.test6@example.com	scrypt:32768:8:1\$w1oh94n6tWPPQViz\$28a4e5ef70bfcea59a9eba9fb27ac559f4df66ead4c3720ac8e83288d171030d6594a47

Figure 1.2

2. Neighborhoods: Neighborhoods are preexisting divisions with predetermined unique neighborhood name, description, and unique id. No users can change or add neighborhoods. In my project. Figure 2.2 shows a sample of my predefined neighborhoods.

```
CREATE TABLE Neighborhoods (  
  neighborhood_id SERIAL PRIMARY KEY,  
  name VARCHAR(200) UNIQUE NOT NULL,  
  description TEXT  
);
```

Figure 2.1



The screenshot shows a database query interface. At the top, a SQL query is entered: `1 SELECT * FROM neighborhoods`. Below the query, there are tabs for "Data Output", "Messages", and "Notifications". The "Data Output" tab is active, displaying a table with the results of the query. The table has three columns: `neighborhood_id` (integer, primary key), `name` (character varying (200)), and `description` (text). The table contains four rows of data:

	neighborhood_id [PK] integer	name character varying (200)	description text
1	1	Greenwood	A quiet residential area known for its lush green parks and family-friendly atmosphe...
2	2	Sunnydale	Known for its vibrant community life and sunny weather.
3	3	Riverbank	Features scenic river views and abundant wildlife.
4	4	Oldtown	Historic neighborhood with a rich history and classic architecture.

Figure2.2

3. Blocks: Related to neighborhoods, this table stores details about specific blocks within neighborhoods, including a unique ID, the associated neighborhoodID, geographical center point, radius (for postGIS mapping purposes), and a textual description. For simplicity sake, I divided the community into four quadrants with two blocks in each quadrants. Figure3.2 shows all predetermined blocks.

```
CREATE TABLE Blocks (  
  block_id SERIAL PRIMARY KEY,  
  neighborhood_id INT NOT NULL,  
  name VARCHAR (200) NOT NULL,  
  description VARCHAR (200),  
  latitude FLOAT,  
  longitude FLOAT,  
  radius FLOAT,  
  column location geography(Point, 4326),  
  -- use postGIS extension for radius  
  FOREIGN KEY (neighborhood_id) REFERENCES Neighborhoods(neighborhood_id)
```

Figure 3.1

```

1 SELECT * FROM Blocks
2

```

	block_id [PK] integer	neighborhood_id integer	name character varying (200)	description character varying (200)	location geography	latitude real	longitude real	radius real
1	1	1	Block A1	Near the central park.	[null]	40.759	-73.922	1
2	2	1	Block A2	Close to the library.	[null]	40.76	-73.914	1
3	3	2	Block B1	Main shopping district.	[null]	40.763	-73.931	1
4	4	2	Block B2	Residential area with schools.	[null]	40.766	-73.938	1
5	5	3	Block C1	Riverside walking paths.	[null]	40.748	-73.917	1
6	6	3	Block C2	Old Mill area.	[null]	40.746	-73.911	1
7	7	4	Block D1	Historical center.	[null]	40.743	-73.929	1
8	8	4	Block D2	Ancient market area.	[null]	40.74	-73.934	1

Figure 3.2

Home
Profile
Logout

Meet New People
My Threads
Create New Threads
My Friends
Friend Requests
My Neighbors
My Application
Vote

neighborhoods

Neighborhoods:

Neighborhood Name:	Description:	Select:
Greenwood	A quiet residential area known for its lush green parks and family-friendly atmosphere.	See Blocks
Sunnydale	Known for its vibrant community life and sunny weather.	See Blocks
Riverbank	Features scenic river views and abundant wildlife.	See Blocks
Oldtown	Historic neighborhood with a rich history and classic architecture.	See Blocks

See Blocks:

Block Name:	Description:	Select:
Block A1	Near the central park.	apply
Block A2	Close to the library.	apply

My Application:

Application ID:	Username:	Block Name:	Status:	Created Date:	Updated Date:	Select:
17	test1	Block B1	approved	2024-05-16 00:00:00	None	

Figure 3.3

4. Block Memberships: Users should apply for block memberships. This table contains all the users_id and their associated blocks. Furthermore, a column called status to indicate whether the user is an active member of the block or an in-active member of the block, in which case it would mean that the user has moved away from the block and no longer a member. When first applied, the status is default to pending.

```

CREATE TABLE Block_Memberships (
  membership_id SERIAL PRIMARY KEY,
  block_id INT NOT NULL,

```

```

user_id INT NOT NULL,
status VARCHAR(20) DEFAULT 'pending' NOT NULL CHECK (status IN ('active',
'in-active')),
join_date DATE NOT NULL,
-- if membership canceled.
update_date Date NOT NULL,
FOREIGN KEY (block_id) REFERENCES Blocks(block_id),
FOREIGN KEY (user_id) REFERENCES Users(user_id)
);

```

Figure 4.1

The block membership keeps a record of all user's status. The status is updated based on the approval of the application table via membership_id which we will discuss next. There will be a delete membership option in which case, the record will update to inactive but the record is still kept for record keeping sake. Figure 4.2 shows a sample of all users with their membership status associated with a block. Once a user's membership status gets approved, the block name will appear on users' profile page along side their username, address, and email address etc.

```

1 SELECT * FROM Block_memberships
2

```

Data Output Messages Notifications							
	membership_id [PK] integer	block_id integer	user_id integer	status character varying (20)	join_date date	update_date date	
1		1	1	active	2023-01-15	2023-01-15	
2		2	1	active	2023-02-20	2023-02-20	
3		3	2	active	2023-03-05	2023-04-01	
4		4	2	in-active	2023-03-05	2023-06-01	
5		5	3	active	2023-04-10	2023-04-10	
6		6	4	active	2023-05-15	2023-05-15	
7		7	4	active	2023-07-20	2023-07-20	
8		8	5	active	2023-08-25	2023-08-25	
9		9	6	in-active	2023-08-25	2023-09-30	
10		10	7	active	2023-08-25	2023-10-15	

Figure 4.2

5. Applications: Each applications record is uniquely identified by applicationID and it includes references to UserID for the applicant and blockID for the block they wish to join. The 'status' field indicates the current state of the application. It is default to 'pending' upon the creation of an instance. The status of the application can be changed based on the aggregated query of the Votes table. Figure 5.2 shows a sample of a newly created application that is pending.

```

CREATE TABLE Application (

```

```
application_id SERIAL PRIMARY KEY,
block_id INT NOT NULL,
applicant_id INT NOT NULL,
status VARCHAR(20) DEFAULT 'pending' CHECK (status IN ('pending', 'approved',
'rejected')),
created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_date date,
FOREIGN KEY (block_id) REFERENCES Blocks(block_id),
FOREIGN KEY (applicant_id) REFERENCES Users(user_id)
);
```

Figure 5.1

1 SELECT * FROM application

2

Data Output Messages Notifications

	application_id [PK] integer	block_id integer	applicant_id integer	status character varying (20)	created_date timestamp without time zone	updated_date timestamp without time zone
1	4	8	11	pending	2024-04-24 00:00:00	[null]

Figure 5.2

My Application:

Application ID:	Username:	Block Name:	Status:	Created Date:	Updated Date:	Select:
17	test1	Block A1	approved	2024-05-16 00:00:00	None	

Your application has been updated to the new block.

Figure 5.3 User may apply for a block, choose a different block while keeping the older application until the new one gets approved and the older application is dropped.

6. Votes: This table ties votes to a specific application a user makes to join a block. It will store the applicationID, voter_id, status, created_at, and updated_at timestamps. Since there must be three votes who are already members of a block, the first three members are preexisting admin who are already in the block eligible to admit the first new member. From then on, all future voters would have to have the same block_id as the associated application block_id to vote. The voting system is simple. If a qualified user approves an application the vote_count will increase by 1. There is a query that checks all votes to a particular application_id and sums up the vote_count. As soon as the vote count becomes greater or equal to 3, it triggers the approval of the application, changes the status of the application to approved, and the block_memberships status becomes active.

```

CREATE TABLE Votes (
    application_id INT NOT NULL,
    voter_id INT NOT NULL,
    -- plus 1
    vote_count INT
    FOREIGN KEY (voter_id) REFERENCES Users(user_id),
    FOREIGN KEY (application_id) REFERENCES Application(application_id),
    -- each voter can vote once for each application
    PRIMARY KEY (application_id, voter_id),
    created_at TIMESTAMP,
    updated_at TIMESTAMP
);

```

Figure 6.1

```

        cursor.execute("""SELECT SUM(CAST(vote_count AS INTEGER))
AS total_votes
                        FROM Votes
                        WHERE application_id = %s;""",
(application_id,))
        current_vote_result = cursor.fetchone()
        current_vote = current_vote_result['total_votes'] if
current_vote_result['total_votes'] else 0

        # Insert new vote
        cursor.execute("""INSERT INTO Votes (application_id,
voter_id, vote_count, created_at, updated_at)
                        VALUES (%s, %s, %s, CURRENT_TIMESTAMP,
CURRENT_TIMESTAMP);""",
                        (application_id, session['user_id'],
current_vote + 1))

        # CHECK VOTE AGAIN
        cursor.execute("""SELECT SUM(CAST(vote_count AS INTEGER))
AS total_votes
                        FROM Votes
                        WHERE application_id = %s;""",
(application_id,))
        current_vote_result = cursor.fetchone()
        new_vote = current_vote_result['total_votes'] if
current_vote_result['total_votes'] else 0

        if new_vote >= 3:
            cursor.execute("""UPDATE Application
                            SET status = 'approved'

```

```

WHERE application_id = %s;""",
(application_id,))

# Become a member of that block
cursor.execute("""INSERT INTO Block_Memberships
(block_id, user_id, status, join_date, update_date)
VALUES (%s, %s, 'active',
CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);""",
(application_block_id,
session['user_id']))

```

Figure 6.2 current_vote_result checks the current tallie and stores it in a variable. Upon member approval, it increments the tallie by 1 and makes a comparison. If ≥ 3 , it triggers the update of application status and insertion of applicant into block_membership.

[Home](#)
[Profile](#)
[Logout](#)

Meet New People
My Threads
Create New Threads
My Friends
Friend Requests
My Neighbors
My Application
Vote

All Applications in your block

All Applications:

Application ID:	Username:	Block Name:	Status:	Created Date:	Updated Date:	Vote:	Vote:
18	test2	Block B1	pending	2024-05-16 00:00:00	None	<input type="button" value="Yes"/>	<input type="button" value="No"/>
17	test1	Block B1	approved	2024-05-16 00:00:00	None	<input type="button" value="Yes"/>	<input type="button" value="No"/>

Figure 6.3 All votable applications is accessible to all qualified users.

7. Friendships: This table captures friendship relationships, which are mutual. If User A wants to be friends with User B, then the status is pending until User B accepts User A as friends. For efficiency, although the columns store user_id1 and user_id2 and it is always the case that user_id1 is the user who makes the request. When storing, we store the smaller value as user_id1 via a CHECK to ensure that user_id is always less than user_id2. As soon as user1 makes a friend request, it pops off the list of names that are either 'rejected', 'nill', or 'pending'. If user_2 rejects friend request, user_2 will appear back on user_1's 'Meet new people' list.

```

CREATE TABLE Friendships (
    user_id1 INT NOT NULL,
    user_id2 INT NOT NULL,
    status VARCHAR(20) DEFAULT 'pending' CHECK (status IN ('pending', 'accepted',
'rejected')),
    created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    update_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

```

```
-- when storng, store as smaller value as user_id1
CHECK (user_id1 < user_id2), -- Ensure that user_id1 is always less than
user_id2
FOREIGN KEY (user_id1) REFERENCES Users(user_id),
FOREIGN KEY (user_id2) REFERENCES Users(user_id)
);
```

Figure 7.1

1 SELECT * FROM friendships

Data OutputMessagesNotifications

	user_id1 integer	user_id2 integer	status character varying (20)	created_date timestamp without time zone	update_date timestamp without time zone
1	1	11	accepted	2024-04-24 03:34:32.660079	2024-04-24 03:36:35.775566
2	1	2	pending	2024-04-24 17:46:19.497431	2024-04-24 17:46:19.497431
3	1	2	accepted	2024-04-24 17:46:19.497431	2024-04-24 17:46:19.497431

Figure 7.2

HomeProfileLogout

Meet New People
My Threads
Create New Threads
My Friends
Friend Requests
My Neighbors
My Application
Vote

Meet Users Page

Not Your Friends!:

(Includes: N/A & rejected)(Not Include: Friends & Pending)

john_doe	1	Request friend
wearaszdf	13	Request friend
test5	19	Request friend
carol_white	5	Request friend
alice_jones	3	Request friend
Logan_grey	11	Request friend
test6	20	Request friend
Peter_Ren	12	Request friend
jane_smith	2	Request friend

Figure 7.3

8. Neighbors: Similar to friends, this table captures neighbor relationships between two users, neighbor_id1 and neighbor_id2. However, the key difference is that neighbor is one directional. Neighbor 1 can assign neighbor 2 only. If the user of neighbor_id2 wants to have a neighbor relationship with the user of neighbor_id1, then user 2 would have to set

up a new record stating this intent. Here, it is ok to have neighbor 1 and neighbor 2 to have a duplicated relationship since it is recommended or allowed that both can request to be neighbor of each other. Mutual consent is not required.

```
CREATE TABLE Neighbors (  
  neighbor_id1 INT NOT NULL,  
  neighbor_id2 INT NOT NULL,  
  created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  update_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  FOREIGN KEY (neighbor_id1) REFERENCES Users(user_id),  
  FOREIGN KEY (neighbor_id2) REFERENCES Users(user_id));
```

1 SELECT * FROM neighbors

Data OutputMessagesNotifications

	neighbor_id1 integer	neighbor_id2 integer	created_date timestamp without time zone	update_date timestamp without time zone
1	1	11	2024-04-24 15:26:36.956034	2024-04-24 15:26:36.956034

Figures 8.1 and 8.2

HomeProfileLogout

Meet New People

My Threads

Create New Threads

My Friends

Friend Requests

My Neighbors

My Application

Vote

Neighbors Page

Your Neighbors details are below:

john_doe-----	remove neighbor
alice_jones-----	remove neighbor
grace_blue-----	remove neighbor

Figure 8.3, Neighbor list is instantly populated upon adding neighbor to your list. Once, user1 added user2, user2 can begin sending messages to user1.

9. Thread: Thread works a lot like emails. This table contains all the threads identified by unique threadID. There is no need for the id of the original message since all message is treated as equal and differentiated by each unique timestamp and messageIDs. There is a button for “New Thread” to create a new thread. In addition, sender needs to specify the recipient type of the message. Is it personal friends, neighbor, or block. This will determine the recipient_id so the system know it is dealing with a user_id, block_id or a neighbor_id; each will be a different function. Based on the reference ID, the system will query all the users who are in that block_id or friend_ID and display the message to them.

The Thread creation also returns the threadID, used for all message associated with this thread. I will demonstrate this later.

```
CREATE TABLE Threads (  
  thread_id SERIAL PRIMARY KEY,  
  -- don't need the original message id since there is none at creation  
  original_message_id INT,  
  -- specify the relationship who you'd like this to go to  
  recipient_type VARCHAR(20) NOT NULL CHECK (recipient_type IN ('friend',  
'neighbor', 'block')),  
  -- if References block_id, then we query all the users who are in that  
  block_id and display the message to them  
  -- if References user_id, then we query that specific friend  
  -- if References neighbor, then we query all users who are in a neighborhood.  
  recipient_id INT NOT NULL,  
  -- user_id where block, freind, or neighbor is a condition. ie if  
  user_id.freind = true  
  FOREIGN KEY (recipient_id) REFERENCES Users(user_id),  
  FOREIGN KEY (original_message_id) REFERENCES Messages(message_id)  
);
```

1 **SELECT** * **FROM** Threads

Data Output

Messages

Notifications

	thread_id [PK] integer	original_message_id integer	recipient_type character varying (20)	recipient_id integer
1	11	[null]	friend	1
2	12	[null]	friend	1
3	28	[null]	friend	2

Figure 9.1 and 9.2

The screenshot shows a web application with a blue header containing 'Home', 'Profile', and 'Logout'. A dark sidebar on the left lists navigation options: 'Meet New People' (highlighted), 'My Threads', 'Create New Threads', 'My Friends', 'Friend Requests', 'My Neighbors', 'My Application', and 'Vote'. The main content area is titled 'Compose Page' and contains a form for creating a new thread. The form has a title 'Compose a new thread' and a section 'Select Recipient Type' with a dropdown menu labeled 'Recipient Types...'. Below this is a 'Recipient Email' field with the placeholder 'Enter Recipient Email' and a blue 'Create' button. To the right of the main form, there is a smaller, semi-transparent version of the same form, which also includes a 'Write a message' section with a 'Message Title' field (placeholder 'Enter Message Title') and a text area (placeholder 'Write your message here...') with a green 'Send' button. A green notification box with the text 'New thread created' and a close icon is visible between the two forms.

Figure 9.3, 9.4. Only friends can start threads. If check friend, a pop up textbox appears for user1 to write to user2 via the /newThread route.

The screenshot shows the 'Threads Page' of the application. It has the same blue header and dark sidebar as the previous figure. The main content area is titled 'Threads Page' and contains the text 'Your Thread details are below:'. Below this text is a table with three columns: 'Author:', 'Thread Title:', and 'Date:'. The table lists seven threads. The first six threads are from 'test1' and 'test2', and the last one is from 'john_doe'. The thread titles are hyperlinks. The dates are in ISO 8601 format.

Author:	Thread Title:	Date:
test1	from test1 to john	2024-05-15 23:05:37.160679
test2	hello test1	2024-05-15 23:17:57.066084
test1	is the recorder woring?	2024-05-16 00:47:57.600533
test1	recorder	2024-05-16 00:49:27.759682
john_doe	welcome test!	2024-04-24 17:43:20.275612
test1	from test2 to test1 real	2024-05-16 02:32:59.962006

Figure 9.5 This page contains the title of each thread. User can click on the title of the thread to view all messages.

10.Messages. Manages messages posted by users. Each message is linked to a specific threadID and includes detailes like the author’s ID, timestamp, title, body of the message, and coordinates related to the message content. The send_to and reply_to are IDs that tracks who are the target audience, ensuring only a certain group of people can view and reply to a particular message.

```
11.CREATE TABLE Messages (  
12.     message_id SERIAL PRIMARY KEY,  
13.     thread_id INT NOT NULL,  
14.     reply_to INT,  
15.     send_to INT,  
16.     title VARCHAR(200),  
17.     body TEXT NOT NULL,  
18.     author_id INT NOT NULL,  
19.     location TEXT,  
20.     timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
21.     FOREIGN KEY (send_to) REFERENCES Users(user_id),  
22.     FOREIGN KEY (reply_to) REFERENCES Users(user_id),  
23.     FOREIGN KEY (author_id) REFERENCES Users(user_id)  
24.);
```

1 SELECT * FROM Messages

	message_id [PK] integer	thread_id integer	reply_to integer	send_to integer	title character varying (200)	body text	author_id integer	location text	timestamp timestamp without
1	13	28	[null]	2	Rumor	Rumor has it someone got hit by a car	1	New City	2024-04-24 17:43:
2	14	28	[null]	2	yes	Yes I heard, but he is fine	1	New City	2024-04-24 17:44:
3	1	11	[null]	1	Hello	Hello, I am the first new user yo	11	New City	2024-04-24 02:41:
4	2	11	[null]	11	Hello back	Hello new user, I am the first user	1	New City	2024-04-24 02:48:

Figure 10.1 and 10.2

Home

Profile

Logout

Meet New People

My Threads

Create New Threads

My Friends

Friend Requests

My Neighbors

My Application

Vote

Message Thread Page

Your message details are below:

From:	Title:	Date:	Status:	Reply
test1	Hello test2	2024-05-16 04:47:35.894442	Read	<div>Reply</div> <div>Read text</div>
test1	check	2024-05-16 04:48:04.736823	Read	<div>Reply</div> <div>Read text</div>
test1	are we friends?	2024-05-16 05:08:07.282378	Read	<div>Reply</div> <div>Read text</div>

Write a message

Message Title

Enter Message Title

Write your message here...

Send

Figure 10.3 user can view all messages in a thread. By clicking ‘Read text’ the body of the message will appear and the record_read_status will mark the status of the message read by session[user_id]. User can also reply to a message, grabbing the message id from the click action and the thread_id from the url.

11. Read_status_recorder: a table that stores information that tracks whether a message has been read or not. It tracks the message_id and reader_id so that upon login_in, it remembers what th user has seen and only display the ‘unread’ message via its status.

```
CREATE TABLE Read_Status_Recorder (
  read_status_id SERIAL PRIMARY KEY,
  reader_id INT,
  message_id INT,
  status VARCHAR(20) DEFAULT 'Unread' CHECK (status IN ( 'Read', 'Unread')),
  updated_at timestamp DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (message_id) REFERENCES Messages(message_id),
  FOREIGN KEY (reader_id) REFERENCES Users(user_id)
);
```

```
1 SELECT * FROM read_status_recorder
```

Data Output
Messages
Notifications

	read_status_id [PK] integer	reader_id integer	message_id integer	status character varying (20)	updated_at timestamp without time zone
1	2	11	2	Unread	2024-04-24 10:31:55.620845
2	1	1	1	Read	2024-04-24 10:34:01.058701

Figure11.1 and 11.2

```
# FORM 3 READ TEXT
elif form_id == "form3":
    text_display = True
    message_id = request.form['message_id']
    cursor.execute("""SELECT * FROM Messages WHERE message_id
= %s""", (message_id,))
    to_read = cursor.fetchone()
    cursor.execute(""" UPDATE Read_Status_Recorder
SET status = 'Read', updated_at =
CURRENT_TIMESTAMP
WHERE message_id = %s; """, (message_id,))
    conn.commit()
    flash('Reading text')
```

Figure 11.3 Upon clicking 'read text', an update query change the status to 'Read'.

PART B – SAMPLE QUERIES

1) Create new user profile:

```
2) INSERT INTO Users (username, email, password_hash, address, profile_text,
    photo_url) VALUES
3) ('Logan_grey', 'logan.grey@example.com', 'hashpassword11', '104 Royal St,
    Beachside', 'Politician', 'http://example.com/photos/logangrey.jpg');
```

2) Update user information

```
UPDATE Users
SET email = 'newLogan.grey@example.com', address = '456 New Address St, New City'
```

3) Apply to block_membership

```
4) INSERT INTO Application (block_id, applicant_id) VALUES
5) (8, 11);
6) WHERE user_id = 11; -- Assuming you know the user's ID is 11
```

System will fetch block_id from blocks where block name is "block D2". This will return block_ID = 8. System will also fetch application_id from Users where username is 'Logan Grey'. This will return 11.

4) Create new Thread

```
INSERT INTO Threads (recipient_type, recipient_id) Values ('friend', '1')
RETURNING thread_id;
```

Click 'create new thread', fetch 'type' by selecting from a drop down menu of ('friend', 'neighbor', or block). Fetch recipient_id (user_id) via email if type = friend or neighborhood_id if type = neighborhood, or block_id if type = block. The query will only query the user with the same recipient-type to show them the message. Finally, fetch RETURNING thread_id from previously created thread, store in a variable for all future message related to this thread.

5) Create new message

```
INSERT INTO Messages (thread_id, send_to, title, body, author_id, location)
VALUES
(11, 1, 'Hello', 'Hello, I am the first new user', 11, 'New City');
```

Fetch recipient_id from thread as send_to. Fetch thread_id previously stored as a variable, in this case, thread_id = 11. And Author is the current user.

6)reply to message

```
INSERT INTO Messages (thread_id, send_to, title, body, author_id, location)
VALUES
(11, 11, 'Hello back', 'Hello new user, I am the first user', 1, 'New City');
```

Userid = 1 is going to reply to userID 11 under the same thread. This page will only show messages with the same thread_id.

7) Make friend request

```
8) INSERT INTO Friendships (user_id1, user_id2) VALUES
9) (1,11);
10)INSERT INTO Friendships (user_id1, user_id2, status)
11)VALUES (LEAST(user_id_A, user_id_B), GREATEST(user_id_A, user_id_B),
'pending');
```

User_id1 makes friend request to User_id2. If user_id2 accepts, it updates this instance's status to 'accepted'. Again, we can take any user id but we store it as least to greatest for easier filter later.

8) update friendship

```
UPDATE Friendships
SET status = 'accepted',
    update_date = CURRENT_TIMESTAMP
WHERE user_id1 = 1
    AND user_id2 = 11
```

9) become neighbors

```
INSERT INTO neighbors (neighbor_id1, neighbor_id2) VALUES
(1,11);
```

Neighbor relationships are not mutual, so there is no need for status update.

10) List all friends of user_id11

```
SELECT
CASE
    WHEN f.user_id1 = 11 THEN f.user_id2
    ELSE f.user_id1
END AS friend_id,
u.username,
f.status,
f.created_date,
f.update_date
```

```

FROM Friendships f
JOIN Users u ON u.user_id = CASE
                        WHEN f.user_id1 = 11 THEN f.user_id2
                        ELSE f.user_id1
                        END
WHERE (11 = f.user_id1 OR 11 = f.user_id2)
    AND f.status = 'accepted';
-- return just the user_ids of friends
SELECT
    CASE
        WHEN f.user_id1 = 11 THEN f.user_id2
        ELSE f.user_id1
    END AS friend_id
FROM Friendships f
WHERE (11 = f.user_id1 OR 11 = f.user_id2)
    AND f.status = 'accepted';

```

Figure 10.1.1

1	SELECT
2	CASE
3	WHEN f.user_id1 = 11 THEN f.user_id2
4	ELSE f.user_id1
5	END AS friend_id,
6	u.username,
7	f.status,
8	f.created_date,
9	f.update_date
10	FROM Friendships f
11	JOIN Users u ON u.user_id = CASE
12	WHEN f.user_id1 = 11 THEN f.user_id2
13	ELSE f.user_id1
14	END

Data Output	Messages	Notifications
-------------	----------	---------------

friend_id	integer	🔒
1	1	

Figure 10.1.2

Because I have user_id1 and user_id2, I need to check all permutations of user 11 being either user_id1 or user_id2. If their status is accepted then return the user ids of just the friends. The query is seen in figure 10.1.1 and the resulting 1 friend can be seen in figure 10.1.2.

11) insert and update new record of read_status_recorder


```

INSERT INTO Read_status_recorder (reader_id, message_id) Values
(1, 1),
(11, 2);
-- Update
UPDATE Read_status_recorder
SET status = 'Read',
    updated_at = CURRENT_TIMESTAMP
WHERE reader_id = 1
    AND message_id = 1;

```

Result can be seen in figure 11.2

12) Show only unread messages to user_id=11

```

13) SELECT * FROM Messages
14) JOIN read_status_recorder ON Messages.send_to = 11
15) WHERE read_status_recorder.status = 'Unread';

```

1	SELECT * FROM Messages
2	JOIN read_status_recorder ON Messages.send_to = 11
3	WHERE read_status_recorder.status = 'Unread';

Data Output									
	message_id integer	thread_id integer	reply_to integer	send_to integer	title character varying (200)	body text	author_id integer	location text	timestamp timestamp without time
1	2	11	[null]	11	Hello back	Hello new user, I am the first user	1	New City	2024-04-24 02:48:37.39

Figure 12.1.1 Result of unread messages to user_id=11

13) Show only unread messages to and from userID = 11 and FROM and TO userID=11's friends

```

SELECT m.message_id
FROM Messages m
JOIN read_status_recorder r ON m.message_id = r.message_id
JOIN (
    SELECT
        CASE
            WHEN f.user_id1 = 11 THEN f.user_id2
            ELSE f.user_id1
        END AS friend_id
    FROM Friendships f
    WHERE 11 IN (f.user_id1, f.user_id2)
        AND f.status = 'accepted'
) friends ON (m.send_to = friends.friend_id AND m.author_id = 11)
OR (m.author_id = friends.friend_id AND m.send_to = 11)

Where r.status = 'Unread';

```

```

SELECT m.message_id
FROM Messages m
JOIN read_status_recorder r ON m.message_id = r.message_id
JOIN (
    SELECT
        CASE
            WHEN f.user_id1 = 11 THEN f.user_id2
            ELSE f.user_id1
        END AS friend_id
    FROM Friendships f
    WHERE 11 IN (f.user_id1, f.user_id2)
    AND f.status = 'accepted'
) friends ON (m.send_to = friends.friend_id AND m.author_id = 11)
OR (m.author_id = friends.friend_id AND m.send_to = 11)

Where r.status = 'Unread';

```

Output Messages Notifications

message_id	[PK] integer
2	

First select only user11's friends. Then join on the condition of the appropriate sender and receiver IDs where the status is 'Unread'.

14) Insert Neighbors

```

INSERT INTO Neighbors (neighbor_id1, neighbor_id2, created_date, update_date)
VALUES (1, 11, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);

```

To ensure strangers cannot send emails to strangers. Neighbor messages can be viewed and received based on invitation only. More specifically, only user1 can set up and receive messages and only user2 can send messages. If user2 wants to receive messages, user2 needs to setup a relationship with user1 since the relationship is one directional. Users can get each other's information in-person for their first meeting.

15) SELECT all user_id=11's neighbors

```

16) SELECT u.user_id, n.neighbor_id2 FROM Users u
17) JOIN neighbors n ON u.user_id = n.neighbor_id1
18) WHERE u.user_id = 11

```

16) Fetch all unread messages across all available types (ie all friend, all neighbors, all block) containing sentence like "Hello "

```

SELECT m.message_id
FROM Messages m
JOIN read_status_recorder r ON m.message_id = r.message_id
LEFT JOIN (
    SELECT
        CASE

```

```

        WHEN f.user_id1 = 1 THEN f.user_id2
        ELSE f.user_id1
    END AS friend_id
FROM Friendships f
WHERE 1 IN (f.user_id1, f.user_id2)
    AND f.status = 'accepted'
) friends ON (m.send_to = friends.friend_id AND m.author_id = 1)
    OR (m.author_id = friends.friend_id AND m.send_to = 1)
LEFT JOIN (
    SELECT DISTINCT bm.user_id
    FROM Block_Memberships bm
    JOIN Block_Memberships bm2 ON bm.block_id = bm2.block_id
    WHERE bm2.user_id = 1
) blocks ON (m.send_to = blocks.user_id AND m.author_id = 1)
    OR (m.author_id = blocks.user_id AND m.send_to = 1)
LEFT JOIN (
    SELECT u.user_id, n.neighbor_id1, n.neighbor_id2
    FROM Users u
    JOIN neighbors n ON u.user_id = n.neighbor_id1
    WHERE u.user_id = 1
) neighbors ON (m.send_to = neighbors.neighbor_id1 AND m.author_id = 1)
    OR (m.author_id = neighbors.neighbor_id2 AND m.send_to = 1)
WHERE r.status = 'Unread'
    AND LOWER(body) LIKE LOWER('%Hello %')
    AND (friends.friend_id IS NOT NULL OR blocks.user_id IS NOT NULL OR
neighbors.user_id IS NOT NULL);

```

Query Query History

```

1 SELECT m.message_id
2 FROM Messages m
3 JOIN read_status_recorder r ON m.message_id = r.message_id
4 LEFT JOIN (
5     SELECT
6         CASE
7             WHEN f.user_id1 = 1 THEN f.user_id2
8             ELSE f.user_id1
9         END AS friend_id
10    FROM Friendships f
11    WHERE 1 IN (f.user_id1, f.user_id2)
12        AND f.status = 'accepted'
13 ) friends ON (m.send_to = friends.friend_id AND m.author_id = 1)
14        OR (m.author_id = friends.friend_id AND m.send_to = 1)
15 LEFT JOIN (
16     SELECT DISTINCT bm.user_id
17     FROM Block_Memberships bm
18     JOIN Block_Memberships bm2 ON bm.block_id = bm2.block_id
19     WHERE bm2.user_id = 1
20 ) blocks ON (m.send_to = blocks.user_id AND m.author_id = 1)
21        OR (m.author_id = blocks.user_id AND m.send_to = 1)
22 LEFT JOIN (

```

Data Output Messages Notifications

message_id	[PK] Integer
1	2

This query needs to satisfy either one of the three sub queries, being friend = accepted or block matches users.block or user_id matches neighbor_id. Furthermore, the message as to

be unread and the sentence is changed to lower case for all possible entry of the target words. The three tables are join using LEFT JOIN, allowing for a NULL result if the relationship condition is not met.

17) Location filter

```
CREATE EXTENSION postgis;

CREATE TABLE blocks_area (
    blocks_area_id SERIAL PRIMARY KEY,
    geom GEOMETRY(Point, 4326)
);
-- Insert latitude and longitude values into the location column
UPDATE Blocks
SET location = ST_SetSRID(ST_MakePoint(longitude, latitude), 4326);

--FILTER DISTNACES
SELECT *
FROM Messages
WHERE ST_DWithin(
    location::geography, -- Cast the location column to geography type
    ST_SetSRID(ST_MakePoint(-73.9857, 40.7484), 4326)::geography, -- Cast the
point to geography type
    200 -- Distance in meters
);
```

Added columns for latitude and longitude to the blocks table. The data type for these columns are 'geography(Point, 4326)' which specifies the spatial reference system used by PostGIS. For the final project, I would like to use google map to test exact longitude and latitude values using the PostGIS functions. '200' is the distance threshold in meters from the user's home.

The setup handles the users interactions within the platform. Improvements should be made regarding the scalability of certain tables like read_status_recorder as it is not useful to keep a table of all the read and unread status in long-term storage. Coordinates need to be further tested with real world values and make the corresponding adjustment to the block and message tables where more precise location is needed.