Vin Liu

Qi Zhang

Professor Torsten Suel

CS-GY 6083

3 December 2019

Final Project Part 1

In this document, we derive a relational schema for a web application that supports location-based user communities. The web application allows a user to register an account to access the services and build up social networks within two levels of locality, hoods, and blocks. A user can add friends and neighbors in his/her social networks and send direct messages with them. A user can also post and reply messages within a community of a block or a hood to make his or her messages visible to all the community members. A user can filter and display incoming messages with various scopes. We will explain how we achieve those core functional requirements as well as many user scenarios in this document.

The first step is the logical design. We started from ER model. we modelled seven entities as well as the relationships associate among them. Then used ER diagram to express the overall logical structure of our database. The ER diagram is attached below:

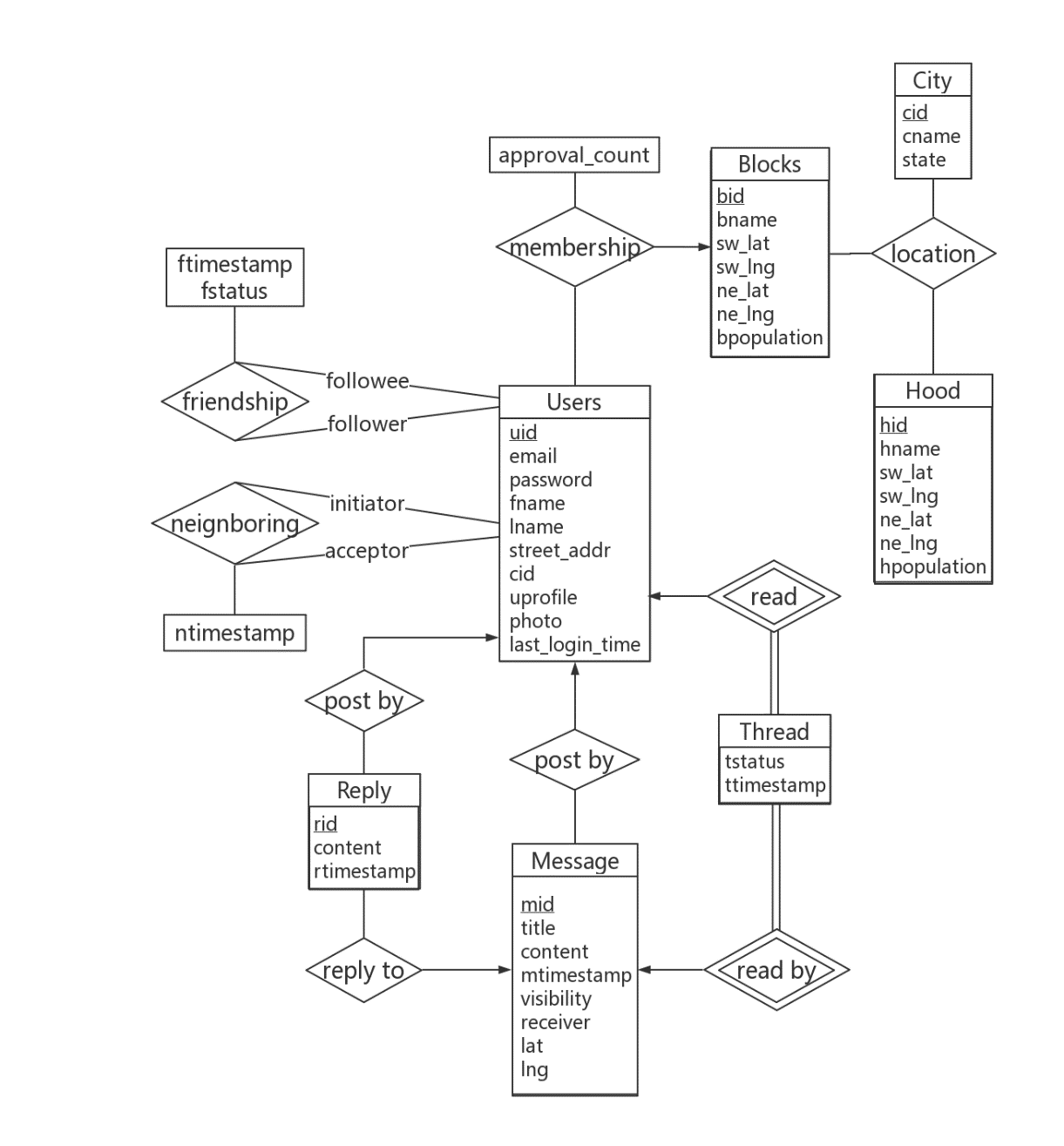


Figure 1 ER diagram

Then we translated the above ER schemas into relational schemas:

* Users (uid, email, pword, fname, lname, street\_addr, *cid*, uprofile, photo, last\_login\_timestamp)
* Hood (hid, hname, sw\_lat, sw\_lng, ne\_lat, ne\_lng, hpopulation)
* Blocks (bid, bname, sw\_lat, sw\_lng, ne\_lat, ne\_lng, bpopulation)
* Location (bid, hid, cid)
* Membership (uid, bid, approval\_count)
* Friendship (follower, followee, ftimestamp, fstatus)
* Neighboring (initiator, acceptor, ntimestamp)
* Message (mid, *author*, title, content, mtimestamp, visibility, *receiver*, lat, lng)
* Reply (rid, *mid*, *author*, content, rtimestamp)
* Thread (*uid*, *mid*, tstatus, ttimestamp)
* City (cid, cname, state)

In the schemas above, primary key is labelled with underline, and foreign key is in *italic*.

Next, we are going to claim the constraints, assumptions and justifications we made for the schemas.

|  |  |
| --- | --- |
| **Users** | |
| uid | INT(10), UNSINGED, PRIMARY KEY |
| email | VARCAHR(45), NOT NULL |
| pword | VARCAHR(45), NOT NULL |
| fname | VARCAHR(45), NOT NULL |
| lname | VARCAHR(45), NOT NULL |
| street\_addr | VARCAHR(45), NOT NULL |
| cid | INT(10), FOREIGN KEY referencing cid in City |
| uprofile | VARCAHR(140) |
| photo | VARCAHR(45) |
| last\_login\_timestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP, ON UPDATE CURRENT\_TIMESTAMP |

* uid is auto-generated and auto-incremented number, serves as a unique identifier of a user. It is generated upon user registration. The value is not recycled when a user is deleted.
* A user does not know his or her uid. An email directly serves as a username. A user registers and logins via his or her email and password.
* fname and lname store the first name and last name of a user, respectively.
* street\_addr stores street address of a user, which be manually entered by the user or by selected on a map upon user registration.
* uprofile stores a profile, like a bio, for a user. It is nullable.
* photo stores an avatar for a user. It is nullable.
* last\_login\_timestamp is a timestamp auto-generated and updated every time a user logins in.

|  |  |
| --- | --- |
| **Hood** | |
| hid | INT(10), UNSINGED, PRIMARY KEY |
| hname | VARCAHR(45), NOT NULL |
| sw\_lat | DOUBLE, NOT NULL |
| sw\_lng | DOUBLE, NOT NULL |
| ne\_lat | DOUBLE, NOT NULL |
| ne\_lng | DOUBLE, NOT NULL |
| hpopulation | INT(11) , NOT NULL |

* hid is auto-generated and auto-incremented number, serves as a unique identifier of a hood. The value is not recycled when a user is deleted. According to the project description, we can predefine hoods. Users register into a block and a hood predefined in the system. So, we assume that addition or deletion of a hood is not common.
* hname stores the name of a hood.
* sw\_lat, sw\_lng, ne\_lat, ne\_lng store the coordinates of a rectangle defining the region of a hood on a map.
* hpopulation stores the population of a hood. It should be a sum of the population in all blocks within the hood. This can be ensured either by defining a trigger in MySQL, or by calculation done in a server application. This will be implemented in project part 2.

|  |  |
| --- | --- |
| **Blocks** | |
| bid | INT(10), UNSINGED, PRIMARY KEY |
| bname | VARCAHR(45), NOT NULL |
| sw\_lat | DOUBLE, NOT NULL |
| sw\_lng | DOUBLE, NOT NULL |
| ne\_lat | DOUBLE, NOT NULL |
| ne\_lng | DOUBLE, NOT NULL |
| bpopulation | INT(11) , NOT NULL |

* The setting of this schema is similar to Hood.
* bpopulation tracks the population of a block. According to the project description, a user’s request to join a block needs to be approved by three members in that particular block or by all members if this block has less than three members. We will need to compare approval\_count in Membership with bpopulation to achieve that.

|  |  |
| --- | --- |
| **Location** | |
| bid | INT(10), PRIMARY KEY, FOREIGN KEY referencing bid in Blocks |
| hid | INT(10), PRIMARY KEY, FOREIGN KEY referencing hid in Hood |
| cid | INT(10), PRIMARY KEY, FOREIGN KEY referencing cid in City |

* We use the composite primary key here to handle the problem of repetition of hood names or block names in different cities. For example, names like “High Street”, “Main Street”, and “Midtown” often appear in many cities.

|  |  |
| --- | --- |
| **Membership** | |
| uid | INT(10), PRIMARY KEY, FOREIGN KEY referencing uid in Users |
| bid | INT(10), PRIMARY KEY, FOREIGN KEY referencing bid in Blocks |
| approval\_count | INT(11), NOT NULL, DEFAULT 0 |

* The uid and the bid here represent a potential membership, which mean a user is a member of a block. This membership is uniquely identified by the composite primary key.
* As we mentioned in Blocks, we will compare approval\_count with bpopulation to determine if this potential membership is pending or passed. Arguably, we could have a status attribute to represent it. We would figure out if it is necessary in project part 2.

|  |  |
| --- | --- |
| **Friendship** | |
| follower | INT(10), PRIMARY KEY, FOREIGN KEY referencing uid in Users |
| followee | INT(10), PRIMARY KEY, FOREIGN KEY referencing uid in Users |
| ftimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP, PRIMARY KEY |
| fstatus | VALUE IN (‘pending’, ‘rejected’, ‘accepted’), DEFAULT ‘pending’ |

* follower and followee here are foreign keys referencing to uid in Users. A follower adds a followee to create a potential friendship at the ftimestamp, and a followee can accept the request to confirm the friendship or not.
* Because a friendship request can be made multiple times, we add ftimestamp to identify them. For example, when A adds B, B rejects it, then A adds B again.
* fstatus is set to ‘pending’ when a friendship request is created. A followee will receive a notification and be able to accept or reject the request. Upon acceptance, the follower will receive a notification. Also, when a fstatus is ‘accepted’, the follower and the followee will have access messages with visibility of ‘friend’ from each other.

|  |  |
| --- | --- |
| **Neighboring** | |
| initiator | INT(10), PRIMARY KEY, FOREIGN KEY referencing uid in Users |
| acceptor | INT(10), PRIMARY KEY, FOREIGN KEY referencing uid in Users |
| ntimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP, PRIMARY KEY |

* The setting of this schema is similar to Friendship.
* Adding of a neighbor is uniliteral according to the project description. An initiator can add an acceptor to his/her neighbor directly without permission. So, we do not need to mark its status like ‘pending’ or ‘accepted’.

|  |  |
| --- | --- |
| **Message** | |
| mid | INT(10), UNSINGED, PRIMARY KEY |
| author | INT(10), FOREIGN KEY referencing uid in Users |
| title | VARCAHR(45), NOT NULL |
| content | VARCAHR(280), NOT NULL |
| mtimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP |
| visibility | VARCAHR(45), NOT NULL, VALUE IN (‘direct’, ‘friend’, ‘neighbor’, ‘block’, ‘hood’) |
| receiver | INT(10), FOREIGN KEY referencing uid in Users, NULLABLE, DEFAULT NULL |
| lat | DOUBLE, NULLABLE |
| lng | DOUBLE, NULLABLE |

* mid is an auto-generated and auto-incremented number, serves as a unique identifier of a message. It is generated upon posting a message. The value is not recycled when a reply is deleted.
* mtimestamp here is for displaying when the message is posted.
* A message has visibility of ‘direct’, ‘friend’, ‘neighbor’, ‘block’, and ‘hood’. When a user chooses to post a direct message, a text field for entering a message receiver is shown. This is done by the front end. Then the receiver is recorded in receiver attribute. However, when visibility is set to be ‘friend’, ‘neighbor’, ‘block’, or ‘hood’, the receiver text field is hidden, and receiver attribute is set to be NULL.
* lat and lng are coordinates where the message is pinned on a map. Users can choose to not pin the message.

|  |  |
| --- | --- |
| **Reply** | |
| rid | INT(10), UNSINGED, PRIMARY KEY |
| mid | INT(10), FOREIGN KEY referencing mid in Message |
| author | INT(10), FOREIGN KEY referencing uid in Users |
| content | VARCAHR(280), NOT NULL |
| rtimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP |

* rid is an auto-generated and auto-incremented number, serves as a unique identifier of a reply. It is generated upon posting a reply. The value is not recycled when a reply is deleted.
* mid represents that this reply is replying to the message with the particular message.
* rtimestamp here is for displaying when the reply is posted.
* According to the project description, a replay has the same visibility as the original message, so we do not need visibility attribute here.
* According to the project description, we can decide the reply depth on our own judgment. So, we decided the reply depth to be 1, which means all replies are replying to the original message. They are organized by chronological order with the aid of rtimestamp.

|  |  |
| --- | --- |
| **Thread** | |
| uid | INT(10), FOREIGN KEY referencing uid in Users, PRIMARY KEY |
| mid | INT(10), FOREIGN KEY referencing mid in Message, PRIMARY KEY |
| tstatus | VARCAHR(45), NOT NULL, VALUE IN (‘read’, ‘unread’), DEFAULT ‘unread’ |
| ttimestamp | TIMESTAMP, NOT NULL, DEFAULT CURRENT\_TIMESTAMP |

* A thread here is defined as a piece of information consisting of a message and corresponding replies. In essence, we decide to pre-compute what information a user should be able to see when he or she accesses the timeline news feed. Such pieces of information are identified by uid and mid. Twitter uses pre-computed information for its timeline too. But the information is stored in memory by the aid of Redis. Definitely, Twitter’s algorithm will be much more sophisticated. We would like to try a simplified approach here by storing pre-computed timeline in Thread schema.
* By pre-computing the timeline, we mean that when a message is posted, we immediately determine who can read this message. For example, when A post a direct message to B, with A’s uid=1, B’s uid=2, and mid=3, we add (2, 3, ‘unread’, CURRENT\_TIMESTAMP) to Thread schema. So, when B loads his or her timeline news feed, message with mid=3 will be displayed. Similarly, when A post a message to his or her friends, entries with A’s friends’ uid will be added. Anyone who is not his or her friend will not even know the message is posted by querying Thread table, and hence the thread is not going to be in his or her timeline. In this way, we pre-compute the timeline for each user. Automatically computing uids and inserting entries into Thread table are done by a server application. We will write it in Python during project part 2.
* tstatus determines whether a user has read the message. It helps when we need to display the unread message only.
* Each time a reply is posted for a message, we automatically set tstatus back to ‘unread’. This is done by the same server application. So, the audience will be about to notice the reply.
* ttimestamp can help when we need to display messages or replies after the last login.

|  |  |
| --- | --- |
| **City** | |
| cid | INT(10), UNSINGED, PRIMARY KEY |
| cname | VARCAHR(45), NOT NULL |
| state | VARCAHR(45), NOT NULL |

* cid is an auto-generated and auto-incremented number, serves as a unique identifier of a city. The value is not recycled when a reply is deleted. According to the project description, blocks and hoods are predefined for simplicity. So, we assume cities and states are also predefined. We will retrieve cities and states information from open-source data on the internet. Users select them upon user registration.
* By using cid as a primary key, we can distinguish the cities with the same name but in different states. For example, the name ‘Portland’ appears in both Oregon and Maine.

We create our schema by using MySQL. The tasks in (c) are done using the following SQL queries:

1. User Y lives in city X and becomes a member of block Z

**Sign up:**

INSERT INTO Users(email, pword, fname, lname, street\_addr, cid) VALUE (‘zl1477@nyu.edu’, ‘pword’, ‘Vin’, ‘Liu’, ‘110 1st St.’, X);

**Become a member of a block:**

INSERT INTO Membership(uid, bid) VALUE (X,Y);

**Create or edit profile:**

UPDATE Users SET uprofile = 'I love SQL!'

WHERE uid = Y;

1. User X posts a message U at coordinate (Y, Z). User V replies it.

**Post an initial message:**

INSERT INTO Message(author, title, content, mtimestamp, visibility, receiver, lat, lng) VALUE (X, 'Posting test case', 'Hi there! I am testing the c2 by posting a new message', CURRENT\_TIMESTAMP, 'hood', NULL, Y, Z);

**Reply a message:**

INSERT INTO Reply(mid, author, content, rtimestamp) VALUE

(U, V, 'Hi there! I my testing the c2 by replying a new message', CURRENT\_TIMESTAMP);

1. User X adds Y as a friend. User X adds Y as a neighbor. User Y accepts the friendship.

**Add friend:**

INSERT INTO Friendship VALUE

(X, Y, CURRENT\_TIMESTAMP, 'pending');

**Add neighbor:**

INSERT INTO Neighboring VALUE

(X, Y, CURRENT\_TIMESTAMP);

**Accept friendship:**

UPDATE Friendship SET fstatus = 'accepted'

WHERE followee = Y AND follower = X;

**List all friends:**

(SELECT follower FROM Friendship

WHERE followee = X)

UNION

(SELECT followee FROM Friendship

WHERE follower = X);

**List all neighbors:**

SELECT accepter FROM Neighboring

WHERE initiator = X;

1. User X

**List all threads that have new message since last access:**

SET @llt = (SELECT last\_login\_timestamp FROM Users AS u

WHERE u.uid = X);

SELECT mid FROM Thread

WHERE ttimestamp > @llt AND uid = X;

SELECT \* FROM Message

WHERE mid IN (SELECT mid FROM Thread

WHERE ttimestamp > @llt AND uid = X);

SELECT \* FROM Reply

WHERE mid IN (SELECT mid FROM Message

WHERE mid IN (SELECT mid FROM Thread

WHERE ttimestamp > @llt AND uid = X));

**List all unread thread in friend feed:**

SELECT \* FROM Message AS w,

(SELECT mid FROM Message

WHERE author IN

(SELECT follower FROM Friendship

WHERE followee = X AND fstatus = 'accepted'

UNION

SELECT followee FROM Friendship

WHERE follower = X AND fstatus = 'accepted')

AND visibility = 'friend') AS m

WHERE w.mid IN

(SELECT mid FROM Thread

WHERE uid = X AND tstatus = 'unread') AND m.mid = w.mid;

**List all “bicycle accident” threads:**

SELECT \* FROM Thread

WHERE uid = X AND mid IN

(SELECT mid FROM Message

WHERE title LIKE '%bicycle accident%' OR content LIKE '%bicycle accident%');

We list some sample data below:

Users:

A screen shot of a computer

Description automatically generated

City:

A picture containing scoreboard, text

Description automatically generated

Hood:

A close up of a black background

Description automatically generated

Blocks:

A close up of a white wall

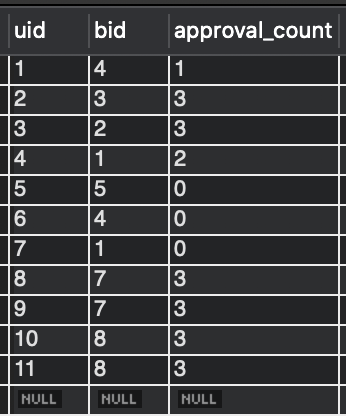
Description automatically generated

Location:

A screenshot of a cell phone

Description automatically generated

Membership:



Friendship:

A close up of a piece of paper

Description automatically generated

Neighboring:

A close up of a logo

Description automatically generated

Message:

A screenshot of a computer screen

Description automatically generated

Reply:

A screenshot of a video game

Description automatically generated

Thread:

A close up of a logo

Description automatically generated

Then we test the queries in part (c):

1. Sign up:

A screenshot of a cell phone screen with text

Description automatically generated

Become a member of a block:

A picture containing screenshot, indoor

Description automatically generated

Create or edit profile:

A picture containing indoor

Description automatically generated

1. Post an initial message:

A screenshot of a flat screen television

Description automatically generated

Reply a message:

A screenshot of a cell phone screen with text

Description automatically generated

1. Add friend:

A screenshot of a computer screen

Description automatically generated

Add neighbor:

A picture containing screenshot

Description automatically generated

Accept friendship:

A screenshot of a computer screen

Description automatically generated

List all friends:

A screenshot of a cell phone

Description automatically generated

List all neighbor:

A screenshot of a video game

Description automatically generated

1. List all threads that have new message since last access:

A screenshot of a computer

Description automatically generated

and replies:

A screenshot of a cell phone

Description automatically generated

List all unread thread in friend feed:

A screenshot of a video game

Description automatically generated

List all “bicycle accident” threads: (we omit the actual message content, just showing thread)

A screenshot of a video game

Description automatically generated