NYU School of Engineering
Computer Science and Engineering
CS 6083, Fall 2015

November 9, 2015

Project #1 (due November 20)

In this project, you have to design a relational backend for a website that allows people to communicate with others in their neighborhood. That is, people should be able to sign up for the service and specify where they live; they can then send and receive messages to other users living close by, and participate in discussions with those users. There are existing services such as `nextdoor.com` that you could look at to understand the general concept. Note that in this first project, you only have to design the relational schema for the site; the web interface will be designed in the second project. You may work along or in groups of two, but you need to register your group members with the GAs at least 7 days before the deadline if you work as a team (no last-minute switches).

More precisely, at a minimum your system will have to support the following functionality. Users should be able to register for the service and specify where they live; this could be either done by providing an address, or by allowing them to click on their location on a map, but you can decide how to do this in the second project. Users should also be able to post a short profile where they introduce themselves and maybe their family, including optionally a photo. In the website, there are two levels of locality, *hoods* (neighborhoods, such as Bay Ridge or Park Slope) and *blocks* (a part of a neighborhood but not necessarily one block, e.g., "7th Avenue between 3rd and 6th Street" in Park Slope). Users can apply to *join* a block; they are accepted as a new member if at least three existing members (or all members if there are less than three) approve. A user can only be member of one block, and is also automatically a member of the neighborhood in which the block is located. For simplicity, we assume that the names and definitions of blocks and neighborhoods are predefined by the company, so that users cannot create new ones; also, blocks and neighborhoods are modeled as (possibly overlapping) axis-aligned rectangles that can be defined by two corner points (say, their southwest and northeast corner).

Members can specify two types of relationships with other members. They can *friend* other members in the same hood, and they can specify (direct) *neighbors*, i.e., members living next door or in the same building or very close by. Friendship is symmetric and requires both sides to accept, while neighbors can be chosen unilaterally. Also, people should be able to post, read, and reply to messages. To start a new topic, a user chooses a subject, and also chooses who can read the message and reply to it. A user can direct a message to a particular person who is a friend or a neighbor, or all of their friends, or to the entire block or the entire hood they are a member of. When others reply to a message, their reply can be read and replied to by anyone who received the earlier message. Thus, messages are organized into threads, where each thread is started by an initial message and is visible by the group of people specified in the initial message. A message consists of a title and a set of recipients (specified in the inital message), an author, a timestamp, a text body, and optionally the coordinates of a location the message refers to; thus, a message about a stoop sale or a traffic accident can potentially be placed on a map in the second part of the project. For simplicity, you do not have to worry about how to include image files and hyperlinks in a message.

It is important to think a little about how users will interact with threads and messages. Most likely, when they visit the website, they will first be directed to a main page listing all recent threads they can read, maybe separated into neighbor feeds, friend feeds, block feeds, and hood feeds. Your system should also store information about past accesses to the site by each user, so that the system can optionally show only threads with new messages posted since the last time the user visited the site, or profiles of new members, or threads with messages that are still unread. Also, a real system would probably have settings where a user can choose to be notified by email of any, or just certain types of, new messages.

Users who move may leave one block and/or hood and apply for membership in another block. For this case, you need to decide what content the user can access. The simplest approach might be to treat messages like email, where the user would still have access to her old messages from the previous block and hood, and only see future messages from the new block. Or should the user lose access to old threads (except maybe those she posted to), and gain access to old messages in the new block? The same problem arises for new friends and neighbors. Make a reasonable choice and justify it.

In this first part of the project, you are asked to design a database backend for such a system, that is, a suitable relational schema with appropriate keys, constraints (and maybe views, but maybe not), plus a set of useful queries that should be created as stored procedures. You may use either your own database installation on your laptop (based on Windows, Linux, or Mac), or on the internet, but no MS Access. Make sure to use a database system that supports text operators, such as "like" and "contains", since you should allow users to search the threads by keywords in the second project. You should of course not use database permissions to implement individual content access restrictions - there will not be a separate database account for each user, but the web interface and application itself will log into the database. Thus, the system you implement can see all the content, but has to make sure at the application level that each user only sees what she is allowed to see.

Note that in the second part, to be handed out in about two weeks and due at the end of the semester, you will have to extend this part to provide a suitable web-based interface. Thus, you cannot skip this project. Following are more details about the problem domain you are dealing with. Obviously, we will have to make some simplifying assumptions to the scenario to keep the project reasonable.

Project Steps: Note that the following list of suggested steps is intended to help you attack the problem. You do not need to follow them in this order, as long as you come up with a good overall design that achieves the requested functionality. Note again that in this first problem, you will only deal with the database side of this project - a suitable web interface will be designed in the second project. However, you should already envision, plan, and describe the interface that you plan to implement.

(a) Design, justify, and create an appropriate relational schema for the above situation. Make sure your schema is space efficient. Show an ER diagram of your design, and a translation into relational format. Identify keys and foreign key constraints. Note that you may have to revisit your design if it turns out later that the design is not suitable.

(b) Use a database system to create the database schema, together with key, foreign key, and other constraints.

(c) Write SQL queries (or sequences of SQL queries or scripting language statements) for the following tasks.

(1) **Joining:** Write a few (3-4) queries that users need to sign up, to apply to become members of a block, and to create or edit their profiles.

(2) **Content Posting:** Write queries that implement what happens when a user starts a new thread by posting an initial message, and replies to a message.

(3) **Friendship:** Write queries that users can use to add or accept someone as their friend or neighbor, and to list all their current friends and neighbors.

(4) **Browse and Search Messages:** Write a few (say 3-4) different queries that might be useful when a user accesses content. For example, list all threads in a user's block feed that have new messages since the last time the user accessed the system, or all threads in her friend feed that have unread messages, or all messages containing the words "bicycle accident" across all feeds that the user can access.

(d) Populate your database with some sample data, and test the queries you have written in part (c). Make sure to input interesting and meaningful data and to test a number of cases. Limit yourself to a few users and a few messages and threads each, but make sure there is enough data to generate interesting test cases. It is suggested that you design your test data very carefully. Draw and submit a little chart of your tables that fits on one or two pages and that illustrates your test data! Print out and submit your testing.

(e) Consider defining appropriate stored procedures for various common tasks. Each stored procedure should have a few specified input parameters (such as the user name and a set of keywords on which a search is performed, or the title, body, and set of recipients of a new message to be posted), and should return a defined result. Note that in the second project, you can call these procedures from a web-based interface outside your database, so find out how to define stored procedures, how they can be called, what restrictions there are, and what sort of technologies (like Java+JDBC, PHP, CGI, Python, or Ruby on Rails) there are for interfacing from a web server.

(f) Document and log your design and testing appropriately. Submit a properly documented description and justification of your entire design, including ER diagrams, tables, constraints, queries, procedures, and tests on sample data, and a few pages of description.