NETS212: Scalable & Cloud Computing

Fall 2020

Final Project

Teams must form by November 2, 2020
Check-ins with shepherd on November 10, 17, 24 + December 1
Code and report due on December 10, 2020
Demos during finals week

1 Overview

The goal of the project is to build PennBook, a "mini-Facebook" application, in teams of four. You will put together many of the techniques you have learned, work as a team, and build a realistic web project with a data analysis backend. This application will have several of the core features of the "real" Facebook. By the end of this project, you will have the experience and necessary skill set to build cloud-based web applications and analyze system design decisions.

During interviews, firms like Facebook and Google often ask you to tell them about a technical project you worked on. If you invest enough time, this project can play that role; you can add it to your "portfolio" of system that you can show a potential employer.

Building a large system with many interacting components is quite different from writing a small program, like we have been doing for the homeworks. Before you begin, please watch the lecture videos about the project and, ideally, the relevant lab videos. These contain many important recommendations, e.g., about starting early, doing a detailed design, defining milestones, making full use of Git, assigning a "lead" for each component, doing regular integration tests, and doing a "code freeze" a few days before the demo, to reserve some time for proper debugging and testing.

1.1 Required stack

We are going to allow for a fair amount of flexibility in your feature set, but we do require that you use certain tools:

- The application itself should be built using Node.js.
- Back-end storage should be hosted on Amazon DynamoDB.
- Back-end data analysis should be via Spark (on Amazon's EMR).
- The front-end should be designed using JavaScript and CSS; you may want to use Bootstrap.
- Your web portal should be hosted on Amazon EC2.
- You should keep your code in a shared Git repository we will provide (*not* on Github etc.!)

If you want to use additional components or third-party libraries, e.g., for extra-credit features, you can ask for permission using a special Piazza post we will create. We will usually say yes, except when the proposed component would provide functionality we want you to build, or when it would replace a technology we require (say, MongoDB instead of DynamoDB).

2 Required features

Your PennBook implementation should contain at least the following key components.

2.1 Accounts, walls, and home pages

User registration: New users should be able to sign up for an account. They should enter, at the very least, a login name, a password, a first and last name, an email address, an affiliation (such as Penn), and a birthday. They also should declare an interest in at least two news categories (see Section 2.3). Be sure to store only hashed passwords in your database (e.g., using SHA-256); do not store them in the clear!

Account changes: Users should be able to change their affiliation after the account has been created, and they should be able to change the news categories they are interested in. Changes to these fields should result in an automatic status update ("Alice is now interested in Quantum Physics"). They should also be able to change their email and their password, without a status update.

Walls: Each user should have a "wall" that contains posts and status updates in reverse chronological order. Each user should be able to post status updates ("Bob is going fishing") on their own wall, and they should be able to post on their friends' walls as well.

Home page: When the user logs in, they should see a Facebook-style home page, with status updates, new friendships, and profile updates (posts) made by friends. When Alice posts on Bob's wall, her post should appear on both Alice's and Bob's walls and home pages.

Commenting: Users should be able to add a comment under any post they can see (that is, both their own posts and their friends' posts). These comments should appear under the post they belong to, wherever that post appears (walls and home pages). Posts should be able to have any number of comments.

Search: There should be a search field that allows the user to search for other users by name. This field should show search suggestions while the user is typing.

Friends: Users should be able to add and remove friends, and they should see a list of their current friends. The list of friends should indicate which friends, if any, are currently logged in.

Visualizer: Users should be able to visualize their network of friends, friends' friends, etc. (We will provide some example code for the visualizer.) Initially, only the user's direct friends should be visible, but it should be possible to "expand" vertexes (that is, add the friends of the corresponding user to the graph) by clicking on them. Each user should only be able to see links to (1) existing friends, and (2) other users with the same affiliation.

Dynamic content: The user home page should be dynamic: the posts and comments, as well as the list of online friends, should be refreshed periodically. Try to do this without making the page 'flicker', e.g., by caching the information received from the server and updating the page only when the information changes, and/or by making changes incrementally rather than regenerating the whole page.

2.2 Chat

There should be a way for users to chat with each other. You can implement this functionality with basic AJAX requests, but you should also consider a library such as socket.io; this should be much simpler.

Chat invites: When a user's friend is currently online, the user should be able to invite the friend to a chat session, and the friend should receive a notification of some kind that enables him or her to join the chat. Any member of a chat session should be able to leave the chat at any time.

Persistence: The contents of a chat session should be persistent - that is, if two users have chatted before and a new chat session between the same pair of users is established later, the new session should already contain the messages from the previous chat.

Group chat: There should also be a way to establish a group chat by inviting additional members to an ongoing chat. When a third member joins a chat, this should open a new (group) chat session. The group chat should continue to work as long as it contains at least one member - even if all of the three original members have left.

Ordering: The messages in a chat (or group chat) session should be ordered consistently - that is, all members of the chat should see the messages in the same order. One way to ensure this is to associate each chat message with a timestamp, and to sort the messages in the chat by their timestamps.

2.3 News feeds

Your solution should have a news feed with relevant news (headlines + URL to an article). You'll be using real news data from this file, which comes from https://www.kaggle.com/rmisra/news-category-dataset. Note that this data file is not itself in JSON format, but rather consists of a set of lines of JSON data. To parse, you need to first read one line at a time into a `String`, then parse that `String` using a JSON parser. Suggestion: you will probably want to update your `pom.xml` to add a JSON parser such as Jackson or GSON. The data contains articles from 2012–2018; you should take the dates in the file and add four years.

Feed updates: New articles should appear in each user's feeds 1) at least once every hour, and 2) whenever the user changes the interests in their profile. To add an article, you should randomly fetch a new news post from the list of articles (as described below) that were published on the current date (see below for details), and add it to the user's social feed. No article should be added to the feed more than once. Users should be able to "like" articles that are shown to them.

News recommendation: Each user should have some set of interests that correspond to news categories (see the data file's "category" column). The articles in their feed will be selected in a weighted random fashion, using weights computed with an implementation of the *adsorption algorithm* in Spark. This should run periodically, at least once per hour, and on demand whenever a user changes their interests.

The Spark job should start by building a graph from the data underlying your social platform. The graph should have a node for each user, each news category, and each article that was published on the current day or earlier. It should also have undirected edges 1) (u, c) if user u has selected category c as an interest; 2) (c, a) if article a belongs to category c; 2) (u, a) if user u has "liked" article a; and 3) (u_1, u_2) if users u_1 and u_2 are friends. The spark jobs should assign weights to the edges. For each c, the weights on the (c, a) edges adjacent should be equal and add up to 1. For each user u, 1) the weights on the (u, c) edges should be equal and sum up to 0.3, 2) the weights on the (u, a) edges should be equal and sum up to 0.4, and 3) the weights on the (u, u') edges should be equal and sum up to 0.3.

Now run adsorption from the users to assign a user label + weight to each node (including the article nodes). Given the ranked graph as above, the social network recommender should take the set of potential articles (those from the same day, minus ones that have already been recommended), and normalize the adsorption-derived weights on these articles. Then it should randomly choose an article based on this weighted random distribution.

News search: In addition to seeing articles that are being recommended to them, users should also be able to search for news articles explicitly. This generalizes what was done with the TED Talk search from your homeworks. Index all titles by keyword, remove stop words from the file given in the homeworks; and regularize case and stemming. Likewise for queries. Articles should be ranked based first on the number of search terms they match, and then by their weights in the graph.

Interfacing with the Web Application: We recommend your group thinks carefully about how the different components (data storage, Spark job to recompute weights, search / recommendation) interface. You likely will want to invoke the Spark task via Livy or the equivalent, with an easily configurable address

for the Spark Master Node. Most likely you'll want to use some form of persistent storage (DynamoDB?) to share the graph and other state.

2.4 Security and scalability

You should ensure that your design is *secure*: for instance, users should not be able to impersonate other users (without knowing their password), and they should not be able to modify or delete content that was created by other users. Your design should also be *scalable*: if a million users signed up for your system tomorrow, your system should still work and provide good performance (although of course you will have to add some extra resources, e.g., additional EC2 instances). Keep in mind that many of the cloud services we have discussed, such as DynamoDB and Spark/MapReduce, are naturally scalable.

2.5 Project report

At the end of the project, each team must submit a short final report, as a PDF file with no more than five pages. This report will be part of your project score; grading will be based both on clarity of writing and on technical content. The report should contain, at least:

- a short overview of the system you built;
- a technical description of the components you built;
- a discussion of nontrivial design decisions;
- a discussion of changes you made along the way, and lessons you learned;
- a brief description of the extra-credit features, if any; and
- a screenshot of your system in action.

Please try to choose the right level of detail (not too nitty-gritty, not too high-level), and please avoid repeating points that are already in this handout. For instance, don't write that your solution has a news feed (this was required!); instead, write how you designed the DynamoDB table that the newsfeed uses, and why you did it that way.

2.6 Opportunities for extra credit

We will give a liberal amount of extra credit for creativity in the project. Below are some suggestions for extra-credit items:

- LikedIn-style Friend requests with confirmation (i.e., users can accept or deny friend requests);
- Hashtag support;
- Privacy controls (e.g., limiting who can see a particular post);
- Profile picture, and/or ways to post pictures and not just text;
- Notifications (e.g., when users are friended or when friends join);
- Infinite scrolling;
- Groups;
- Targeted advertising, based on users' interests or on words in their posts;
- Site-wide "what's trending" feature

However, these are just examples - feel free to consider other features as well, e.g., based on functionality that the current Facebook has, or even novel features that you define yourself. If you are considering a particular feature and are not sure whether it would be counted as extra credit, feel free to post a question

on Piazza. Extra credit will be awarded based on the complexity of a feature, not just based on its presence; for instance, profile pictures are easier than groups and thus would be worth less, and adding a simple input field where users can declare group memberships would yield considerably less extra credit than a full implementation with separate pages for groups where members can post messages, etc.

3 Logistics

3.1 Team registration

Once your have formed a team, one member of your team should send an email to Andreas on or before the "team formation" deadline specified on the first page; the email should include the names and SEAS logins of all four team members, and the other team members should be CC'd on the email. Your team will then be assigned a team ID, which will also be the ID of your team's Git repository. If you do not register a team by the deadline, we will assign you to a team.

3.2 Shepherding process

Once your team has registered, it will be assigned a TA as a "shepherd". You should schedule 15-minute sessions with your shepherd on the dates specified on the first page of this handout. During these sessions, which all team members should attend if possible, you should give a quick overview of your design and describe any features you have already implemented by then. Your shepherd will give you some feedback on your design and your progress, and he or she will answer any questions you may have.

The shepherding sessions will not be graded, but we reserve the right to impose a penalty if a team fails to schedule sessions or shows up completely unprepared.

3.3 Project demos

Your team must do a short demo (about 15-20 minutes) via Zoom during the finals period. A number of time slots on different days will be posted to the discussion group near the end of the semester. We cannot reserve slots or create additional slots, so, if you or some of your team members have constraints (holiday travel, final exams, etc.), please discuss this well in advance and pick a suitable slot. Once your team has reserved a slot, the only way to change the slot is to find another team that agrees to swap slots with you.

All team members must be present for the demo. If a team member is not present for any reason other than documented medical emergencies, he or she will not receive credit for the project.

3.4 Submission checklist

- ☐ Your code contains a reasonable amount of useful documentation.
- ☐ You have checked your final code into the Git repository.
- ☐ You have removed all AWS credentials from the code you are submitting.
- ☐ You are submitting your code as a .zip file, which contains all the files needed to compile and run your solution (including all .js/.ejs files, and all Spark code); as well as README file that describes 1) the full names and SEAS login names of all team members, 2) a description of features implemented, 3) any extra credit claimed, 4) a list of source files included, 5) a declaration that all of the code you are submitting was written by you, and 6) instructions for building an running your project. The instructions must be sufficiently detailed for us to set up and run your application.
- ☐ Your .zip file is smaller than 10MB. Please do not submit large binaries, large data files, or your `node_modules` directory. If you used third-party libraries and are not including them in your

submission, please state in the README file where these libraries can be obtained (URL plus one sentence saying what the library does).

- ☐ You are submitting your final report as a PDF file of no more than five pages (including appendices, screenshots, images, and any references).
- ☐ You submitted your code archive and your final report via the web interface (!) before the deadline on the first page of this handout. Notice that there are two separate uploads: `ProjectCode` and `ProjectReport`. Submissions in any other form (email etc.) will not be accepted. Jokers cannot be used for the project.