NETS 2120 Final Project Report

Group 31

Awad I., Daniel S. P., Aditi C., M. Abdullah K.

Introduction

In this project, we implement a Social Media Service called ‘PennBook’. To use PennBook, we have implemented a login and signup process for authenticating users. Once a user is authenticated, their session remains active until they sign out or close the browser. While logged in, a user can update their password, affiliation, interests, email, profile picture, and privacy settings. Users can also send and receive friend requests, post on their own and their friends' walls, and see status updates and posts on the homepage. PennBook includes search functionality for both users and news articles, as well as personalized news recommendations based on an adsorption algorithm. Finally, users can communicate with their friends through a chat feature, including the ability to create group chats, invite friends to groups they created, and leave existing chats.

Our tech stack is as follows:

- Front-end: EJS, CSS, Javascript
- Back-end: Javascript (Node.js) with Express
- Database: AWS DynamoDB (Key-Value NoSQL Database)
- Recommendation Algorithm: Java with Spark, run on AWS Elastic MapReduce
- Chat Feature: Socket.io in Javascript

Technical Description of Key Components

User Profiles: A core part of our application was the implementation of user profiles, which enabled each user to have a personalized experience on Pennbook. This was implemented by maintaining a list of records storing information about the user's details and interests within a single DynamoDB table, with the username as the partition key. To elicit more complex queries, usernames were also used as the partition key in the friends, chat groups, and chat invites tables allowing information across tables to be joined. Additionally, a username prefix table was

maintained, which stored entries for all substrings of a given username, so searches could be performed more efficiently and scale as the number of users grows.

Friends: To maintain a list of relationships between users a friends table was constructed that stored a bidirectional entry for each friendship. Though this was less compact than just storing a single entry, it made querying the table easier as well as more extensible as additional information such as the ‘friend request status’ could be stored if necessary.

Posts and Comments: Posts are stored in a table with a partition key indicating the wall of the user where they appear, which allows the posts for a given user to be retrieved with a single call to the database. Similarly, comments utilize a complex partition key that references both the user whose wall the post is on and the timestamp of the post. This allows the comments for each individual post to be retrieved. This is useful in our implementation, as we can retrieve comments for an individual post when a new post is detected in our AJAX call, without needing to fetch unnecessary information from the database.

Chat: The chat feature is implemented through socket.io, which allows for bidirectional communication between the client and the server. The socket handlers on the server side retrieve the data from tables in DynamoDB (which store group chats members' invite information, and the content of the messages) and send them to the client side. The client-side then renders this information and informs the server of any button clicks that occur (e.g creating a group, sending messages, etc). Since the implementation is based on creating groups and sending invites to other users, the implementation was naturally scalable to a group chat with more than two users.

Newsfeed Recommendation: The newsfeed article recommendations are based on an adsorption algorithm. This algorithm involves creating a graph between users, their news likes, interests, and the news articles themselves. The adsorption weights (for each user and article) are calculated based on this graph and written to a DynamoDB table. This algorithm runs every hour and updates these weights in the table. When a user navigates to their newsfeed, the data is retrieved from this table and displayed based on descending order of weights. The user also has the option to like any news article, which affects future calculations of weights and hence news recommendations.

News Search: A user can search for an article, which queries a search index table in DynamoDB. This table maps each keyword to a list of articles. The keywords were derived after processing the text in the headline. The order of the search results is based on the number of search keywords matched, and then in descending order of the weights (which are retrieved from the table written by the adsorption algorithm)

Nontrivial Design Decisions

Our database schema uses the user wall and the time of creation as the key for posts, in contrast to a generated ID. A generated ID would be the stable approach, as the backend could assign an ID to incoming posts and atomize put operations to the database, ensuring all posts get added and there is no ID overlap. If we use time as an index, we can increase the scalability of our implementation, as there is no need for atomic operations. However, this means that if two users post on the same wall at the same instance of time to the millisecond, one of them will be rejected. In our design we accept this compromise, as we do not expect several posts on a wall to get added per second, the user would simply get an ‘error adding post’ message and would be able to resend successfully.

Another design decision was choosing between socket.io and AJAX jQuery requests for implementing chats. While we were more familiar with the syntax of AJAX jQuery, socket.io provided a much better framework for a task like chat, since it allowed real-time, scalable, bidirectional, and event-based communication between the web client and server. It also has a straightforward API and ensures automatic reconnection when compared with AJAX jQuery, which would have involved multiple HTTP requests and responses.

Changes Made and Lessons Learned

One challenge in developing a full-stack application was ensuring that there was an intuitive user interface, with consistency across pages. As we decided to split our tasks by features, we ended up getting divergent elements and stylings across the application making it confusing to use. In retrospect, drafting some wireframes for all the pages would have been useful to gain a big-picture view of the application before coding. However, during the process, we also ended up componentizing elements such as the navbar to reduce repeated code and enforce consistency.

Initially, comments used a similar schema as posts, where they pointed to a comment and were sorted by time. However, in order to introduce nested comments this schema presented problems, as a nested comment would have to point to the timestamp of its parent, which was not intuitive to implement. Because of this, the schema was modified to an ID-based approach, where each comment had a unique ID and a custom sort function was implemented in the backend to send the nested structure to the frontend.

One aspect of the project that was not originally considered when designing our schema was the requirement to search through all users in the database. Initially, this was implemented by performing a scan on the user's table upon each keypress which proved to be an expensive computation, especially as the number of users scaled. Ultimately, this was adjusted to create a separate user prefix table, which mapped a given prefix to the usernames that it corresponded to. This was a much more efficient search, however, there are still some limitations to the current implementation as it does not allow for search by name.

Getting socket.io working took a considerable amount of time. There were a lot of moving parts, libraries to be installed and some issues with version differences. Learning and getting around the API was also challenging at first but once the API was understood, implementing the chat feature became much easier.

Extra-credit Features

As an extra-credit feature, PennBook allows users to reply to comments on posts, reply to replies, and so on in a nested fashion. Since we retrieve all comments under a post by having the Primary key for comments pointing to a post, we can retrieve nested comments without additional database calls. A new column is added to the schema pointing to the comment id a comment is replying to, or -1 if it is in the first nested layer. In our backend, we sort comments with a custom sort, which is sent to the frontend in nested order.

Another extra-credit feature we have implemented is chats content persistence for group chats, in addition to only direct messages as required for the project. This means that if some group of users has chatted before and a new chat session between the same group of users is established later, the new session will already contain the messages from the previous chat.

Screenshots of the System