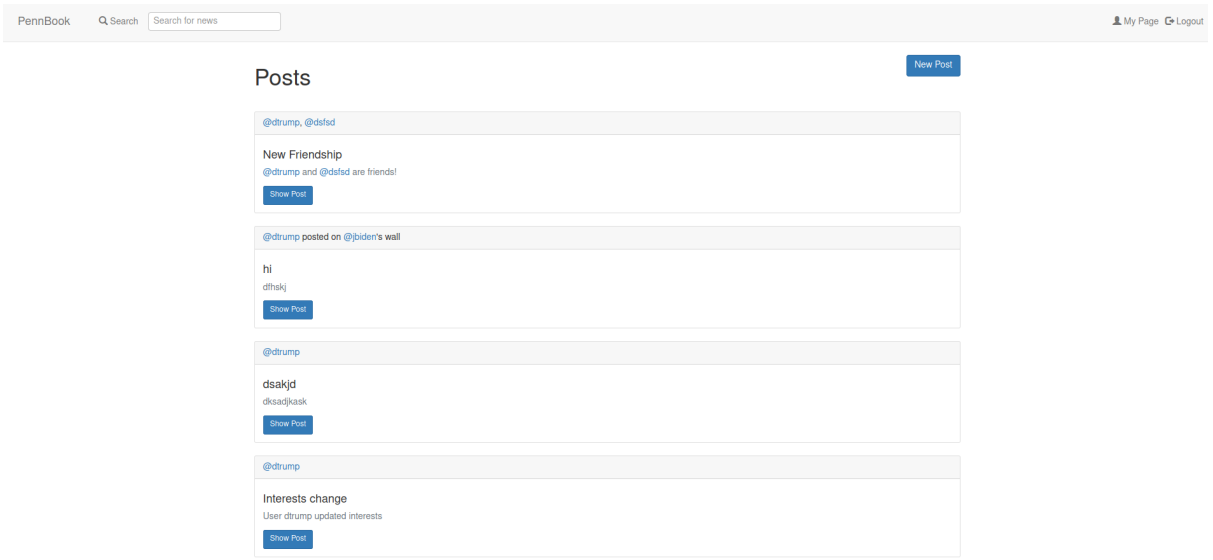

Features overview

Chat

From your list of friends, you can invite your friends that are currently available to chat. After you send an invitation, the invitee and you get a notification (sent with socket.io) with a link to the chat session. You open the chat session on a separate page. On that page, you can add available friends to a new group chat session in a similar way. The whole chat works without the need to refresh a page, all the information, including who leaves the chat, are implemented with socket.io.

News

The data of 200k news from the JSON file are parsed in a Node.js script and added to a `news` table in DynamoDB with batch write. The publish date is being saved as a timestamp which is sort key allowing the adsorption algorithm to work efficiently. All news have IDs assigned. The data about news, users, friendships, news likes, and users' interests are being loaded from DynamoDB by the Spark job, which then converts it into a graph in the form of RDD edges and nodes. The adsorption algorithm is performed on the graph. The final labels on news nodes are getting filtered in order to remove the labels for news that were not published today and to remove labels that were already recommended to users. Then, based on those labels, for each user, there is one article added to a `news_recommendations` table, together with the current timestamp, which allows the news to be properly ordered from the newest to the oldest in the news feed.

Website

SignUp/Login/Logout

On the signup page the user inputs their information like username, first name, last name, etc. The usernames must be unique. Additionally, when the user initially signs up they will have to select some interests, which corresponds to the news algorithm. On the login page, the user can login using their username. We also added security questions that are to verify before the user wants to change the password and a profile photo using Multer, uploading the files on S3 bucket.

Userpage

The user will have a profile page where they can see their information like their name, email, affiliation, and interests. We implemented account changes for each of the fields. They also will see a list of their friends and will see their wall where they can post updates and their friends can also post to. You must be friends with a user to be able to post on their wall.

Search

When we search by first or last name, we get the output of the first and last name of the user. It is done by the prefix. For example, “Aida Akuyeva”, “Aida”, “Akuyeva”, “AidaAku” will all output the user’s first and last name & username : Aida Akuyeva (aakuyeva)

Friends

Friendships are mutual. When users become friends, a post will be created saying that user1 and user2 became friends. Additionally, only friends of a certain user will be able to post on their wall. When your friends are logged in, you will see a green dot next to their name.

Posts

Users can post status updates on their homepage or their wall. Whenever users become friends or a user changes their affiliation or interests a post will be made, which all the users’ friends will be able to see on their homepage. We can also delete the post, like it or comment it.

Comments

Users can comment on any of their friends' posts. Users can also delete their comments if they want.

Visualizer

The logged in user is able to see their direct friends and the friends of their friends by clicking on them. Unfortunately, users are only able to see links to their existing friends but not their affiliation.

Likes

When creating a new post, the user can click the “like” button, which operates on the unique post_id that we assign for each user. So if the user likes the post, this information gets stored in a DynamoDB table for likes so that we can add likes.

Technically interesting features

Chat

In order to make the chat feature more scalable, as a hash key (chat ID) for a chat session, we used a string made up of usernames that were invited to a particular chat session. The usernames were sorted in order to ensure the uniqueness of the chat ID. For example, if users u2, u1, and u3 were in a chat session, their chat ID would always look like ‘u1-u2-u3’. This design choice also made it very easy to preserve the chat history for every conversation. The messages just don’t get removed after the users leave the chat session, and whenever they invite the same people to a chat session, their chat ID would always look the same so they would always restore the old chat session with the old messages. At first, with this design choice, we planned not to store members of a chat session in a separate DynamoDB table because all the members could be just retrieved from the chat ID, therefore, we would just need a `chat_messages` table. However, we needed to change this later, because it made it very difficult to allow users to leave the chat. Basically, since everyone invited was in the ID of the chat, there was not really a concept of “leaving the chat session” because the session was defined by its member. Therefore, we’ve created a separate DynamoDB table `chats` with chat ID as the hash key and username as the sort key. This allowed us to separate the concept of users invited to the chat from users present in the chat - now, a user `u2` might have been invited to a chat session `u1-u2-u3` but then could leave the chat session, and the chat ID stayed `u1-u2-u3` but only users u1 and u3 could access the chat. If u1 or u3 invited u2 back, u2 could access the chat again and see all the messages. The lesson learned: too primitive solutions may add significant complexity later on.

Dynamic content

At first, the chat has been implemented such that there was an AJAX request being sent to the database every 2 seconds in order to download new content such as messages. However, it turned out that that solution was not scalable - if there were a billion users using the chat at the same time, they would be sending a billion requests every 2 seconds, more than 99% of which would be unnecessary. Therefore, instead, we implemented the socket.io solution, which allows the updates to be sent to specific users only when there are some updates to be sent. However, with WebSockets, new challenges came. Socket.io allows you to broadcast a message to a specific socket id or a specific room, however, it doesn’t support broadcasting to specific users identified by their username. For example, if we want to invite a user to a chat session, then broadcasting the invite to all the users would be a security issue, but sending the invite to a specific user was only possible if you knew their socket id. Another issue could be that one user could have several windows opened, therefore, one user could have several socket ids. As a solution to these issues, we maintain

a hash map with keys being usernames and values a list of socket ids identifying the WebSocket connections of a corresponding user. This lets us create a method for broadcasting socket.io messages to users identified by usernames. This turned out to be helpful in many places and made developing dynamic content for the whole website easier and more efficient.

News recommendation algorithm

There were 8 types of directed edge types. User and category nodes were identified Strings, and news were identified by Integer IDs. All the edges also had different edge weights. It would be very complex to deal with all the edge types separately during the label propagation, therefore, after processing the edge weights, we generalized all the edges to have a common type (Object, (Object, Double)), and unioned them into a single general PairRDD for all edges in the graph, and execute the label propagations on that. The problem that could have potentially arisen with that edges generalization was that if a user had, for example, username “entertainment,” they would be confused with a category “ENTERTAINMENT” in this algorithm. We solved it by adding “CATEGORY “ prefix to all Strings identifying category nodes in the graph. A username “CATEGORY entertainment” wouldn’t be valid, so it solves this problem. This problem doesn’t exist with news because they are identified by Integers so they would have a different hash code.

In order to prevent the majority of labels in the graph from being insignificant labels with microscopic weights and let the algorithm converge, we tuned the algorithm to discard labels with weights smaller than 0.009 and consider the graph as converged as soon as no label changes by more than 0.05.

The spark job run by Livy on the EMR pulls and saves the data from and into DynamoDB, therefore, it required AWS authorization. For this purpose, we used static credentials configured in the Config.java file that gets serialized and sent by Livy to EMR.

Posts

Creating the database schema for posts was fairly challenging. This is because of the nature of having different types of posts. Users obviously had to be able to create their own posts, however, whenever users became friends or whenever a user changed their affiliation or interests, a post would also need to be created. Additionally, there was also an issue storing the posts for when a user posted on their friends’ wall. Thus, I ran into issues with storing the posts in the database and having the posts correctly displayed to all their friends, and in terms of the post containing friendship information, I had the problem where the post would display twice.

In order to solve this problem, I ended up having to create a separate column in the database called ‘type.’ If it was a regular post, I left the column blank. If the post was created for a friendship, I put ‘friendship’ in the column and for a users’ post on their friends wall, I called it ‘wall’.

Essentially, by categorizing the posts, it made the backend of fetching posts a lot easier to handle, and solved some bugs I was having.

Extra credit

- Ability to have multiple chat sessions open at one time (Damian)
- Notification when a user invites you to chat (Damian)
- Profile picture (Aida + Damian)
- Welcoming e-mail (Aida)
- Security questions (Aida)
- Delete post (Aida)
- Delete comment (Aida)
- Like post (Aida)

Work distribution

2.1 Accounts, walls, and home pages

User registration: Aida (backend), Amanda (front end), account changes: Aida, walls: Amanda, home page: Amanda, commenting: Amanda, search: Aida, friends: Amanda, add friend, unfriend, and list of current friends: Anaick, Visualizer: Anaick, dynamic content: Damian

2.2 Chat (Damian)

2.3 News (Damian news recommendation & connect to web interface & search + Aida news feed display & liking news)