Group G03: Nicholas Kuo, Madelyn Dempsey, Alan Du, Arnav Ghatiwala

Mini Meta

Introduction

Welcome to Mini Meta, a Facebook-like web application which allows users to post updates, stay connected with friends, chat, and stay up to date with the latest news.

System Architecture

The following technologies were used to build "Mini Meta." In the frontend, we used React.js with JavaScript, HTML/CSS (with Bootstrap) to develop a user-friendly social media interface. Specifically for the frontend friend visualizer, our team utilzied React-graph-vis. For backend development, Node.js and Express was used to create the necessary API routes for the main functionality like posting, adding friends, commenting, and more. We used AWS DynamoDB for our application's database to store user, posts, news, and chat data. For asynchronous HTTP data exchange between frontend and backend, we used Axios. For our news adsorption algorithm, we used Apache Spark on Amazon's EMR for our back-end data analysis. Lastly, for our chat functionalities which required data to be sent from backend to frontend, we utilized socket.io.

Web App Description

Backend

Accounts, Walls, Home pages

1. Created users table, with partition key "username." User data like password, firstname, lastname, email, affiliation, birthday, list of article category interests, list of chat IDs, list of chat invites, and logged in status was stored in users. All these data points were stored in the users table so that the frontend only requires 1 query to obtain most necessary information for user information and chat, which is then cached on the frontend.
2. Created friends table, with partition key "username" and sort key "friend." By adding a "friend" sort key, 1 query of the friends table yields a list of all of a user's friends.
3. Created posts table, with partition key "author" and sort key "timestamp." The table stores the content of a home page or wall post, including author of post, recipient of post (for posting on other's pages), and content. To get all the posts of a user, we iterate through all the user's friends (assuming user is friends with himself), and for each friend, query the posts he/she is an author for, then for each post querying the comments. 3 subsequent promises are used for this task.
4. Created comments table, with partition key "authorAndTime" and sort key "timestamp." The table stores the comment's content and author. The partition key is generated by the concatenation of a post's author, a comma, and the post's timestamp. This allows the frontend to query for posts, concantenate each post's author and timestamp to form a foreign key, and then use this new key to easily query the comments table for the post's comments. Since a post can have any number of comments, comments had to be another table as opposed to being in the posts table for scalability purposes.

Chat

1. Utilized the users table to store a user's chat rooms (as chatIDs) and incoming chat invites.
2. Created chatMessages table, with partition key "chatID", sort key "timeStamp" and attributes "sender" and "message.

News Feed

1. Loaded the articles data from the file, and created an articles table with partition key "articleId" (generated by program). Article data like headline, date, url, etc. is stored here, along with a "proxy" attribute which is "y" by default. This is because the table also has a global secondary index "proxy-date-index" with "proxy" as a partition key and "date" as a sort key to efficiently query all articles ("proxy" = "y") with "date" less than or equal to today for adsorption.
2. Created inverted table with partition key "keyword" and sort key "articleId". It stores all keywords (stemmed, non-stopwords) which are unique for each article from the article's "headline", "short_description", "authors", and "category".
3. Created an articleLikes table and an adsorptionScore table, both with partition key "username" and sort key "articleId". The latter also stores the score and the article's date. The date allowed for efficient, scalable queries using FilterExpressions.
4. Ran adsorption by combining data from articles, users, articleLikes, and friends tables into JavaPairRDDs. This was run on EMR using Livy.

Frontend

User registration

We build 2 pages, login and signup page. For the login page, when a user attempts to login, we query the DynamoDB query for the username to see if the user exists and if his/her hashed password equal. If there is a valid login, the user is redirected to the home page, else there is an alert message. For the signup page, the user must fill in the necessary fields stored in the backend users table such as affiliation and article interests. They are then redirected to home if the username they selected is unique, else they get an alert.

Account Changes

We have an Edit Account page where the user can change their email, affiliation, interests, and password in four different fields. Once a change has been made, it will be reflected on the panel on the left side of the page where their account overview is shown. These status updates also add a new post of the user's new information to the backend posts table.

Posts, Share Post, Search Bar

We build a React component for each post, which accepts a post's author, content, recipient (whose wall post is for), and list of comments. Posts is a child component of the Walls and Home Page components discussed below, which will supply each post's relevant data, queried from the backend, to the post component. The Post component contains a comment submission button which sends new comments (keyed by concatenation of post's author and timestamp) to the backend table "comments." We also built a Share Post component which allows a user to create a new post and saving it in our backend. Lastly, we build a search bar which dynamically queries

the backend for all users filtered by their full names containing characters searched. A search bar suggestion, if clicked, links to a user's wall.

Walls
The walls page uses the url parameter to determine which profile wall the user is on. A profile page has a React card of the profile user's basic information, as well as a list of online or offline friends which dynamically refreshes by querying the backend every 15 seconds. The page queries the backend to see if the logged in user and the user of profile page are friends in order to display an add-friend or remove-friend button. If both are friends, the page queries the backend for the profile user X's wall posts, which iterates through all X's friends' authored posts (and posts' corresponding comments) where recipient equals X. A user posting on his own page with the Share Post component has his post have a recipient equal himself. We implement dynamic refresh every 10 seconds, which fetches all the relevant posts and comments corresponding to the user, passing this data down to the Posts components on this page to update their information.

Home Page
The home page, similar to the Wall page, queries the backend for a logged in user's posts and populates Post components. For logged in user Y, it queries all Y's friends and adds all friends' posts to the Home page without filtering. The Home page dynamically refreshes each 10 seconds to query the most updated posts (sorted in decreasing chronological order) and comments.

Visualizer
To create the friend visualizer using React, we chose to use react-graph-vis. Each node stores a username, so when a node is pressed, the page queries for the username's friends and adds these new nodes to the graph visualizer through a setState() handler.

Chat
The chat page is divided into four main modules: list of friends, list of invites, list of chat rooms, and the chat module.

- The friend list displays the user's friends and their online statuses. A user can invite online friends to chat with them through this module. Once the invitation is sent, the inviting user is automatically added to a chatroom, whereas the invited user will join only after they accept the invitation.
- The list of invites displays the user's incoming invitations. A user can either accept or reject an invitation.
- The list of chat rooms display the user's joined chats. A user can switch between these chat rooms, and the chat module will update accordingly.
- From the chat chat module, the user can send, receive messages and leave the current chat. They can also add more friends to the current chat through userID. When a user does this, they will be automatically added to a new chat room with all the current users and the invited users, and invitations will be sent out to everyone in the chat.

News Feed
The news feed page has two main components: the recommended article and the search results.

- News article recommendation: a user is regularly recommended a unique news article based on his interest.
    - Implemented using a probability distribution based on adsorption scores.
- News search functionality
    - Users can enter queries to search for news articles, and the top 30 results are projected. This is sorted by, a) the number of keywords matched, and b) the adsorption score of the article for that user (if one exists).
- Liking: users can easily like/unlike news articles through a button on each news post (both the recommended posts and search results).

Non-trivial Design Decisions

One nontrivial design decision was the table breakdown and partition key choices for fetching all relevant posts of a given username. Whenever a new username is registered, we store that the user is friends with himself. This ensures that to fetch all home page posts, we just need to iterate through all friends, query each friend's authored posts, and then query each post's comments using the strategically chosen partition key of post author concantenated with timestamp. The breakdown of tables also ensured a scalable backend design; for instance, the use of a comments table ensures that any number of comments can be under a post.

Another important design decision was how we strategically designed the frontend React components' hierarchical architecture to efficiently cache backend queries for a logged in user. For example, a Wall page component had children components like "Share Post," "User Info.," and "Post." Hence, when the Wall renders, it sends 1 query to get the user information and then passes these data points to all children components for a seamless UI experience.

For Adsorption, it was crucial to decide the structure of the JavaPairRDDs to store the label weights, edge weights, etc. such that no Spark operation became a major bottleneck. Moreover, for scalability, the algorithm stopped iterating if no weights changed by more than x, weights less than y were removed in each iteration, and existing scores in DynamoDB were only updated if they changed by more than z. Given the size of the data, it was non-trivial to decide these thresholds to ensure that important weights weren't pruned off but a satisfactory speed was achieved, and they were set experimentally through querying the adsorptionScore table.

Changes and Lessons Learned

Changes
One significant change we made was a shift from using EJS to React for our frontend. React supports dynamic content and refreshing as well as nesting components and fetching requests, and we have a few team members with significant experience using it, so we decided it would be a good challenge that would help create a more optimized and scalable product. However, this also became a challenge when we realized that a group member may not have had as much React experience as we had anticipated. This also created a challenge with the visualizer because we had to research other packages to use since the ejs sample code wouldn't apply.

Lessons

- Careful planning of frontend and backend architecture, as well as data flow, is important for the first brainstorm sessions. Changing the architecture later in development is costly.
- We need to be careful about the css we use individually before merging because sometimes we would merge and then an entire page's css would reference other pages.
- We need to focus on certain implementation tasks that seem trivial, such as running jobs on EMR, running the web app on EC2, setting cron jobs, etc. along with the logical implementation itself. These tasks have many lower-level issues that are hard to debug.

Extra Credit

- **Infinite scrolling**: both the profile wall and homepage feature have infinite scrolling, where users can continuously scroll to see older posts

Selected Screenshots for Web App