ECE 356 Project

Matt Belisle

April 5, 2020

1 Part 1 – Entity Relationship Diagram

I apologize for the picture in Figure 1 not being the best quality, I will describe each relationship here for clarity. There are three entities in my social media design, the first being User, it is simply the data I decided that I needed to store on any of the social media user's. Note, the ID field is simply a numeric key used to identify a user, and the UserID in the user table is the string that the person would sign up with (also a unique identifier). The ID field is simply quicker to match on than a string for many reasons, the simplest being it is of definite length and comparisons are simple with numbers. Also note, in other entities the UserID field corresponds to the ID of the user not their userID. The second entity is Tweet, it represents a singular tweet by a user, similarly to User it contains only the most important information for a tweet, and doesn't hold any unneeded data for this project (Twitter stores a field for where in the world the tweet was tweeted from, I'm sure they have their reasons but I don't see any benefit of that). Finally the HashTag entity which is what I am using for topics in my social media database, it is simply a HashTag string.

The majority of the relationships are between tweet and user so those will be explored first, then the remaining three.

1. Favorites

   The favorites relationship is the "thumbs-up/like" of my social media, for any tweet zero or more users may have favorited it, and similarly for any user, they may have favorited zero or more tweets.

2. Tweets

   The integral relationship between Users and Tweet is the actual tweets relationship, this is how tweets will be mapped to the user who tweeted it, in this design every tweet has been tweeted by exactly one user, but not every user must have made a tweet.

3. Mentions

   This relationship describes tweets that mention some other user, via typing "@userID" into their tweet somewhere, similarly to Favorites, a user may be mentioned in zero or more tweets, and a tweet may mention zero or more users.

4. Retweet

   This relationship describes the retweet functionality in Twitter, where a user re-posts a tweet from another person, similarly to favorites, a user may retweet zero or more tweets, and a tweet may be retweeted by zero or more users. This relationship has the extra timestamp value, which will correspond

with the exact time a user retweeted it, allowing old tweets to be displayed at the top of a timeline if somebody retweets it.

5. Follows

   This relationship is exactly what you would expect, any user can follow any other user, (note there is no concept of blocking a user, or a private profile) and this is unidirectional so you may be followed without following the other user. As should be expected, any user can follow and be followed by 0 or more users.

6. TweetHashTags

   This relationship is similar to mentions in that it relates the content of a tweet to something else, in this case the content of the tweet to the hash tags contained within it. As such any tweet can contain zero or more hash tags, and every hash tag in the relationship should be contained within at least one tweet as a hash tag is used to make a collection of tweets and there shouldn't be any empty collections.

7. SubHashTag

   The final relationship is a subhash tag, this is essentially a subtopic of the parent hashtag. I have chosen to parse these as #parent##subHashTag, where the consecutive '##' within a hashtag correspond to the beginning of the subHashTag. A parent may be the parent of any number of subhash tags, but a subhash tag can belong to exactly one parent.

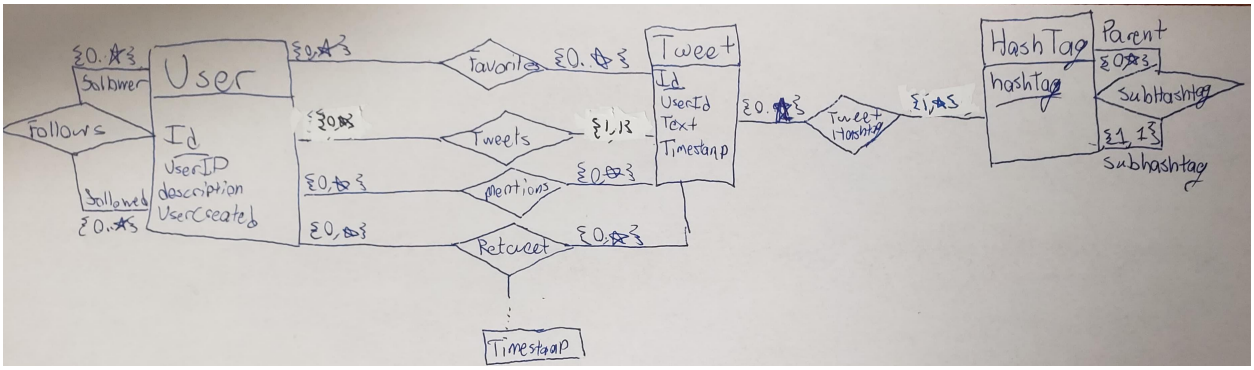

Figure 1: Proposed Entity Relationship Diagram for a simple social media application

2 Part 2 – Database Design

From the entity relationship diagram the various relationships and entities were turned into their respective tables and foreign keys were created to represent the links in the relationships, these can be found in Database/tables.sql. Two procedures were created as well, one for handling the insertion of a tweet, and the other for getting the immediate replies to a tweet, as well as the chain of tweets above it to produce a coherent conversation for a tweet. The reason the reply chain procedure only returns one level of replies below the tweet is because of the nature of replies, many tweets can reply to one and those can have many replies themselves etc, this recursive nature is very difficult to represent in MySQL as far as I am aware. These two stored procedures are located inside of Database/StoredProcedures.sql

3 Part 3 – Data Insertion

My data was acquired from Kaggle and is the Australian Election 2019 Tweets dataset located at https://www.kaggle.com/tar
election-2019-tweets.

4 Part 4 – Application

My application has 3 layers, and consists of the database itself, the back-end http server that exposes various endpoints and is the single source of communication with the database, and finally a simple React web-app that utilizes the endpoints exposed by the API.

Specifically the back end database queries are made through a Kotlin library called Kotlin Exposed, it provides a simple DSL to make queries that mirrors MySQL, I asked before I implemented with this and it was okay at the time.

All API endpoints are located in src/main/kotlin/api/SocialMediaApplication.kt and appropriate comments have been made for each endpoint.

5 Difficult Parts

In terms of difficulties I had while making the project the biggest was the web app, when I read the requirements I thought what it meant I had to create a GUI application that runs on top of an API, so that is what I did. I don't have the most experience with React (or any front-end framework/tools) and it was a bit of a steep learning curve but I'm reasonably happy with the results.

In terms of difficulty in the back-end the most difficult part was a single query, the FullTweet query (located inside of FullTweet.kt) is what transforms a Tweet from the Database into a single object that contains all of the information a tweet actually needs. This information is things like how many favorites/retweets a tweet has, is this tweet a reply if so who and what tweet was it replying to, is this a retweet and if so who was it retweeting? I originally had this as about six simple queries that each gathered a small part of the total data, but that seemed inefficient to call the database six times for things I should be able to get with one, so I created a fairly complex query to get all of the information at once and I'm very happy with the results.

That new query is here so you do not have to find it in the code base:

```
Select T.userID as userID, T.id as tweetID, T.timestamp as timestamp, T.text as 'text',
       U.userID as 'userName',
       RetweetedUser.userID as 'originalPoster', RetweetedUser.id as 'originalPosterID',
       RepliedUser.userID as 'repliedTo', RepliedUser.id as 'repliedToID',
       IF(R.originalID is null, false, true) as 'isReply',
       IF(R3.userID is null, false, true) as 'isRetweet',
       IF(F.userID is null, false, true) as 'isFavorited',
       IF(R2.userID = $userID, true, false) as 'isRetweeted'
from Tweet T
         INNER JOIN User U on ${tweet.userID} = U.id
         LEFT JOIN Reply R on T.id = R.replyID
         LEFT JOIN Tweet T2 on T2.id = R.originalID
```

```
        LEFT JOIN User RepliedUser on T2.userID = RepliedUser.id
    -- basically we want to get the original tweeter as this may just be a
    -- retweet if this is null then we should assume it is not a retweet
        LEFT JOIN Retweet R3 on R3.userID = ${tweet.userID} and R3.tweetID = T.id
        LEFT JOIN User RetweetedUser on T.userID = RetweetedUser.id
        LEFT JOIN Retweet R2 on R2.userID = $userID and R2.tweetID = T.id
        LEFT JOIN Favorite F on F.userID = $userID and F.tweetID = T.id
where T.id = ${tweet.id}
```

Where anything wrapped in ${...} is string templating in Kotlin and specific to the query.