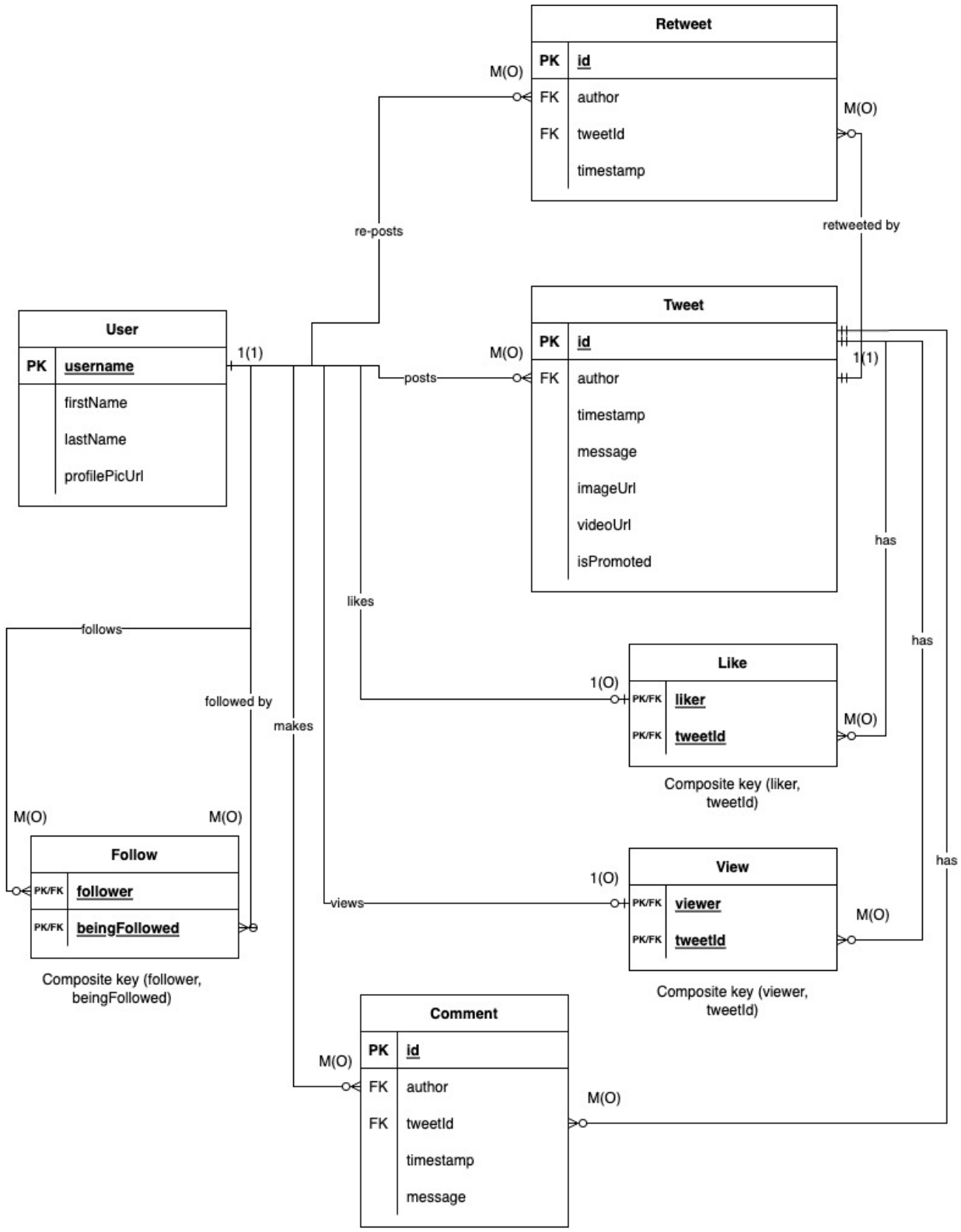
452 Twitter Midterm Write-up

Author: **Brigham Andersen**

Section: 1


Retweet
PK id
FK author
FK tweetId
timestamp
M(O)
M(O)
re-posts
retweeted by
User
PK username
firstName
lastName
profilePicUrl
1(1)
Tweet
PK id
FK author
timestamp
message
imageUrl
videoUrl
isPromoted
M(O)
posts
1(1)
has
has
has
likes
follows
followed by
makes
views
Like
PK/FK liker
PK/FK tweetId
1(O)
M(O)
Composite key (liker, tweetId)
M(O)
M(O)
Follow
PK/FK follower
PK/FK beingFollowed
Composite key (follower, beingFollowed)
View
PK/FK viewer
PK/FK tweetId
1(O)
M(O)
Composite key (viewer, tweetId)
Comment
PK id
FK author
FK tweetId
timestamp
message
M(O)
M(O)

Entities, Attributes, and Normalization

User

I made a user entity so there can be a table to store the users who register for and use Twitter. I knew that every user has a username/handle, so I added that as an attribute. Since usernames have to be unique, I knew that would make it a good primary key. I made this decision assuming that you can't change your username once it has been created. If I received an updated spec that said username could be changed, then I would have just made an auto-incrementing integer the column to make it more resistant against changes. I also made other basic attributes that would be essential to a user, like their first name, last name, and their profile picture. I'm assuming that all pictures and videos being used on the platform are being stored in AWS S3, so for the `profilePictureUrl` attribute, I just store a URL to where that picture is stored. As far as normalization, this is in Boyce Codd Normal Form (BCNF). No columns depend on non-unique columns, all are unique.

Follow

I made a follow entity to describe relationships between users as far as who follows who. `follower` represents which User performed a 'Follow' on another user. `followingBy` represents which user is now being followed by the user who performed the follow. I initially was going to have a separate auto-incrementing integer for the primary key, but then I realized that it would leave redundancy if multiple follows were made by a user. To alleviate that and still achieve BCNF, I just kept the `follower` and `followingBy` attributes. I combined them into a composite key. Each of them alone can have repeats, but the combination of the two of them guarantees uniqueness, so it makes for a great primary key when they are composite (A specific user can only follow another specific user one time max, so no duplicates). Therefore it achieved BCNF.

Tweet

I made a tweet entity to represent each tweet that a user makes. I knew this one would be hard to ensure complete uniqueness since authors can have many posts and the message in the posts can be the same, so I opted to have an auto-incrementing integer `id` for my primary key. Its attributes included the `author` (so you can know who tweeted, a foreign key to the User table), `timestamp` (this was particularly useful for the query of what day people tweet most), the `message` itself (for the hashtag #GoCougars query, I just parse the message to look for that hashtag), the image URL and video URL (both again links to media in an S3 bucket), and a boolean (internally an integer) attribute `isPromoted` (especially useful in queries that only dealt with tweets with promotions). This entity achieved BCNF once again because there's no duplicated data across columns. Its attributes that make up the key, the whole key, and nothing but the key.

Retweet

This was created to represent when a user retweets another user's tweet. For the same reasons above of difficulties with uniques, I made an auto-incrementing integer `id` my primary key. This entity required 2 foreign keys, the first foreign key `author` linking back to the User table to identify which user was performing a retweet, then the second foreign key `tweetId` connecting to the Tweet table to grab the exact id of the tweet retweeted. Lastly I threw on a timestamp attribute. This wasn't necessary for any queries but retweets on Twitter show the date/time of when they were posted, so I included it. This entity achieved BCNF because I avoided duplication across columns by just having linking foreign keys to other tables.

Comment

Comment was created because Twitter allows users to comment on tweets. To achieve this, my primary key was again an auto-incrementing integer `id` to ensure uniqueness and robustness. I had a foreign key `author` pointing to the User table so you know who made the comment, a foreign key `tweetId` for which tweet it referenced, a `timestamp` referring to when the comment was made, and a `message` for what was commented. Once again BCNF was the result because anywhere where data would have been duplicated (regarding users and tweets), I only had a foreign key pointing to the primary keys of those tables.

Like

The Like entity is to track which users like which post. In a simplistic form of Twitter I could have just had an attribute on the Tweet entity that was `numLikes`, but only storing an integer for the number of likes doesn't allow me to track who liked each tweet. By breaking this out to its own entity, I can now store a primary key for the `liker` of the post (user who liked it) and another primary key id of the tweet they liked. For this entity I opted to make `liker` and `tweetId` combine into the composite key. Each of them alone can have repeats, but the combination of the two of them guarantees uniqueness, so it makes for a great primary key when they are composite (A specific user liking a post can only like the same post up to one time, so no duplicates). Once again, by using this composite key, it allowed the table to remain in BCNF. Its attributes that make up the key, the whole key, and nothing but the key.

View

Similarly to the Like table, you simply could track the views of a tweet on the entity itself, but we don't just want to know the number of views — we want to know which users specifically viewed each post. In its own entity I made another composite key combining data across tables, `viewer` and `tweetId`. This allows there to be a connection between who viewed a post and the post they viewed. Having connections between the user and the tweet via this table allowed for no duplication, only to have the attributes that make up the key, the whole key, and nothing but the key.

Relationships and Cardinalities

> 💡 Before discussing relationships and cardinalities, as a note, I often worked under the assumption that entities were serialized, so that's why some of the relationships may seem a little less intuitive. If the specs said to make everything non-serialized, the ERD would like quite different.

User → Other

The User table/entity has the most relationships across tables. In all relationships, from the other entities' perspective, they have a cardinality of 1 and only 1 with the user table, because the entities wouldn't exist without a user to make them, but for each serialized entity, only 1 user can own them (see note above for serialization note). A single user can have zero to many tweets, and they can have zero to many retweets. For a given tweet, a single user can like or view that tweet 0 to 1 times. A single user can comment on a single tweet zero to many times. Lastly, a single user can follow zero to many users and a single user can be followed by zero to many users.

Tweet → Other

The second most complicated table/entity as far as relationships is Tweet. A tweet can be authored by 1 and only 1 user (it wouldn't exist without an author). It can be retweeted by zero to many users. It can have zero to many likes, views, and comments. All those other entities (Retweet, Like, View, and Comment) can only have 1 and only 1 serialized tweet associated with them.

Conclusion of Learnings

This midterm has honestly been one of my favorite midterms/projects of my time as a Computer Science major. I've had a lot of experience in the industry as a Software Engineer, mostly with startup companies, so I've often been tasked with designing databases. I'll admit that I went in blind creating those initially. Now that I've taken this class and have had time to master my craft, I feel way more confident. As far as concrete things I've learned, first is to diagram your solution out in an ERD. When you're making the different entities, attributes, and relationships visually, it's easier to connect the dots on how the entities will connect with each other and where you need to fill things in. It also lets you see where your data may be de-normalized and where you might need to instead have a foreign key to join across tables.

Aside from the designing with the ERD, perhaps one of the biggest learnings I had is how helpful it is to know in advance the different queries you will make. As I was making the tables it was great to see what kind of queries I would have to be making, and how I could best store my data to simplify and speed up those queries. I've really enjoyed this first half of the semester on SQL and have seen its power. Unless the second half of the semester changes my mind, I'll default to using SQL databases for my back end architecture until speed or scaling requirements force me to use NoSQL.

SQL File (also found in .zip)

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/09520b99-0efe-4d85-ab5e-8fd4d752514d/main.sql