Section slides: <http://webdev.slides.com/coltsteele/mysql-106>

# Introduction to Instagram Clone Schema

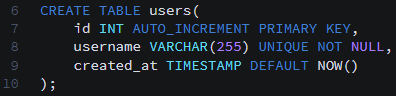
* In this section we will be cloning a portion of the Instagram database, where we’ll have multiple tables to work with and plenty of opportunities to work on our joins
* Let’s start by designing our schema, which will consist of several entities (not necessarily the same as tables)
  + Users
  + Photos/images (represented as an image URL in this case)
  + Number of likes
    - Need a way to prevent people from adding more than 1 like
  + Hashtags
  + Comments
  + Number of followers that user has
  + Number of people the user is following

# Defining the Users Schema

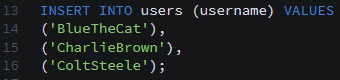
* The *users* schema will look like this



* + The user id is the primary key
  + We want username to be unique as well, but we shouldn’t use it as a primary key because long username strings can take some time to search for. We also want this to be NOT NULL
  + **created\_at** will be a timestamp that defaults the current date and time



* Let’s add a couple of users so that we can play around with them in this section (the next section will add a huge amount of users)



* Code summary

CREATE TABLE users (

id INTEGER AUTO\_INCREMENT PRIMARY KEY,

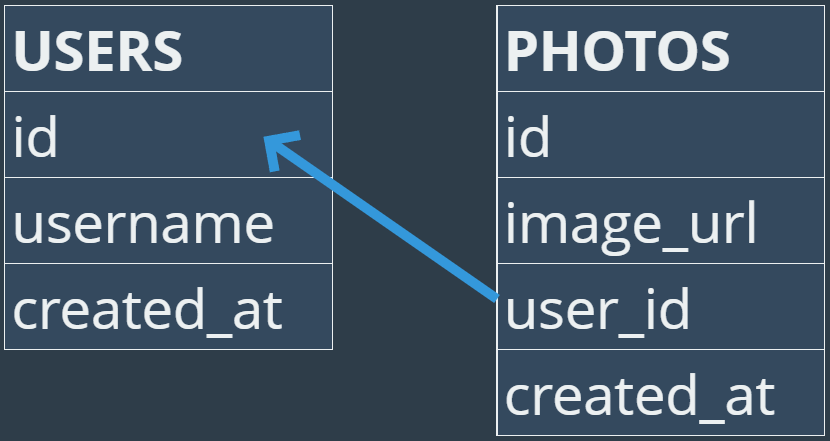
username VARCHAR(255) UNIQUE NOT NULL,

created\_at TIMESTAMP DEFAULT NOW()

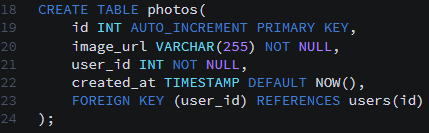
);

# Defining the Photos Schema

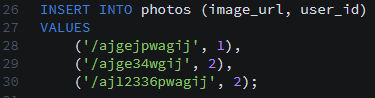
* Here is what our simplified *photos* table will look like, and how it will be connected to *users*



* + We will be using “id” as the primary key, and other tables will reference it
  + The image\_url should be NOT NULL so that we don’t have a situation where we have no photo display
  + “user\_id” must also be NOT NULL – we do not want to have orphan photos with no users



* Let’s insert a couple photos. When adding data to the *photos* table, we will need to supply the **image\_url** and **user\_id**. The others will auto-populate.



* Code summary

CREATE TABLE photos (

id INTEGER AUTO\_INCREMENT PRIMARY KEY,

image\_url VARCHAR(255) NOT NULL,

user\_id INTEGER NOT NULL,

created\_at TIMESTAMP DEFAULT NOW(),

FOREIGN KEY(user\_id) REFERENCES users(id)

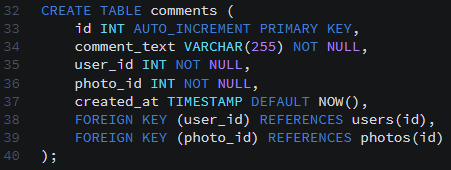
);

# Creating the Comments Schema

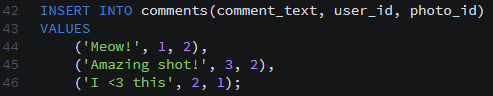
* Comments will rely on users and photos. A comment is authored by someone, and comments are attributed to a specific photo
  + This means we will have two foreign keys
* The *comments* table



* + The **id** will be the primary key once again
  + **comment\_text** cannot be blank. In the real Instagram, a blank comment will not post
  + **user\_id** and **photo\_id** will both be foreign keys and will both be NOT NULL
    - user­\_id will tell us who wrote the comment
    - photo\_id will tell us which photo the comment is on



* Let’s insert some data and practice!



* Code summary

CREATE TABLE comments (

id INTEGER AUTO\_INCREMENT PRIMARY KEY,

comment\_text VARCHAR(255) NOT NULL,

photo\_id INTEGER NOT NULL,

user\_id INTEGER NOT NULL,

created\_at TIMESTAMP DEFAULT NOW(),

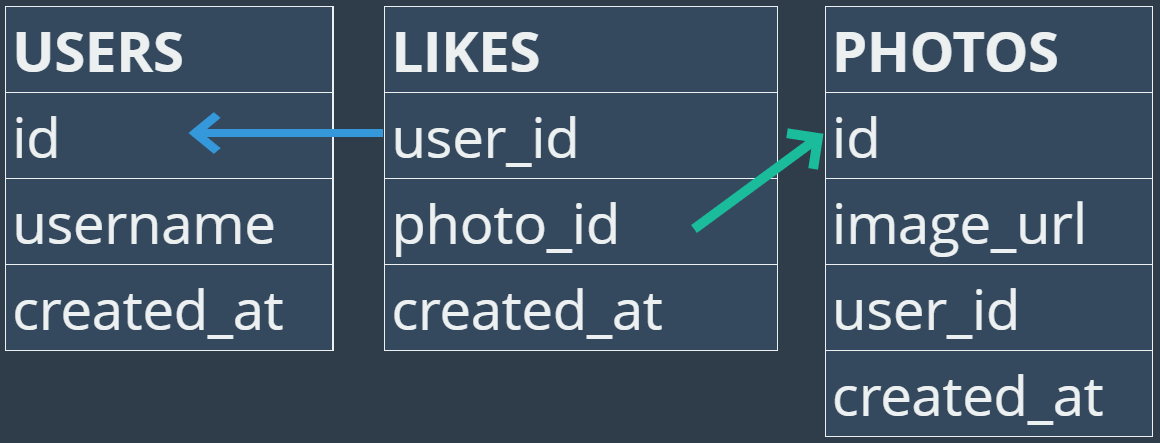
FOREIGN KEY(photo\_id) REFERENCES photos(id),

FOREIGN KEY(user\_id) REFERENCES users(id)

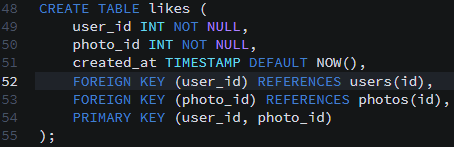
);

# The Likes Schema

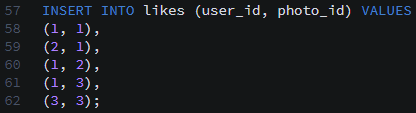
* “Likes” will be somewhat similar to comments. A like must be attributed to the user who is giving the “like”, and must be attributed to the photo being “liked”
* Here’s what the *likes* schema will look like, with the foreign key references



* + In more complicated versions, you could store things like whether the “like” is coming from the web version versus the mobile version
  + Why don’t we have an ID for the likes? It’s because we won’t be referring to unique “likes” in any important manner. Likes refer to other objects, but other tables will not be referring to likes
  + How do we make sure we only have one “like” per user/photo unique combo? This is important because we don’t want to same person to post likes over and over again on the same photo. We can do that by **creating a PRIMARY KEY for the user\_id, photo\_id combination**
* Putting all that together, here is how we make our table



* Optional practice content: having our example people like photos



* Now, if you try to give another like with the same user and photo combo, we get an error



* Code summary

CREATE TABLE likes (

user\_id INTEGER NOT NULL,

photo\_id INTEGER NOT NULL,

created\_at TIMESTAMP DEFAULT NOW(),

FOREIGN KEY(user\_id) REFERENCES users(id),

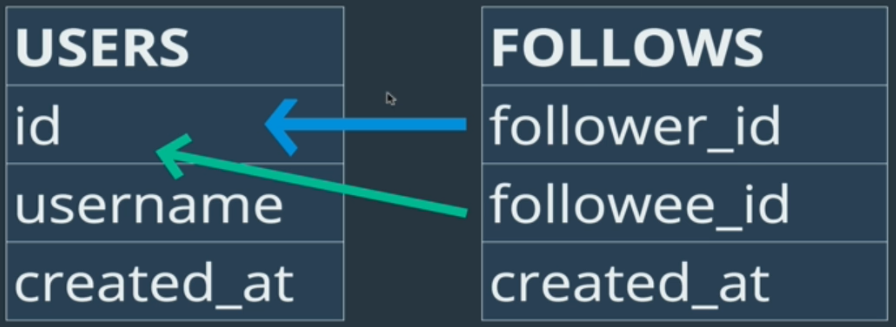
FOREIGN KEY(photo\_id) REFERENCES photos(id),

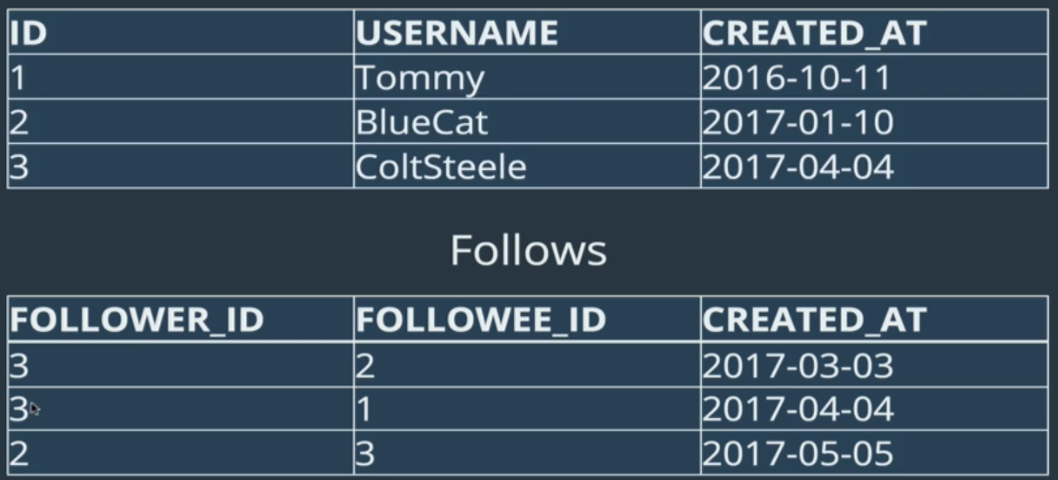
PRIMARY KEY(user\_id, photo\_id)

);

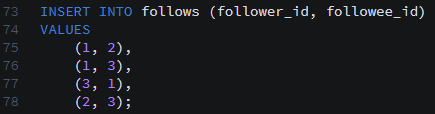
# Relationships

* This refers to relationships between people on Instagram, in terms of users following other users
* We’ll do this in a single table called *follows*, which contains the **follower\_id** (the person doing the following) and the **followee\_id** (the person being followed). Both will be foreign keys referencing the *users* table.
  + We’ll also use **created\_at** to track the date and time
  + We don’t want to have duplicate follows, so the ordered combination of follower\_id, followee\_id must be unique and occur only once
    - The order is important, because we DO want to allow two people to follow one another in unique follow relationships. We just don’t want the same person to follow the same other person twice
  + We do not need a unique ID in this table, as we won’t be referencing follow relationships from any other tables





* Verification/practice code: inserting into our tables



* + Test of database integrity



* Code summary

CREATE TABLE follows (

follower\_id INTEGER NOT NULL,

followee\_id INTEGER NOT NULL,

created\_at TIMESTAMP DEFAULT NOW(),

FOREIGN KEY(follower\_id) REFERENCES users(id),

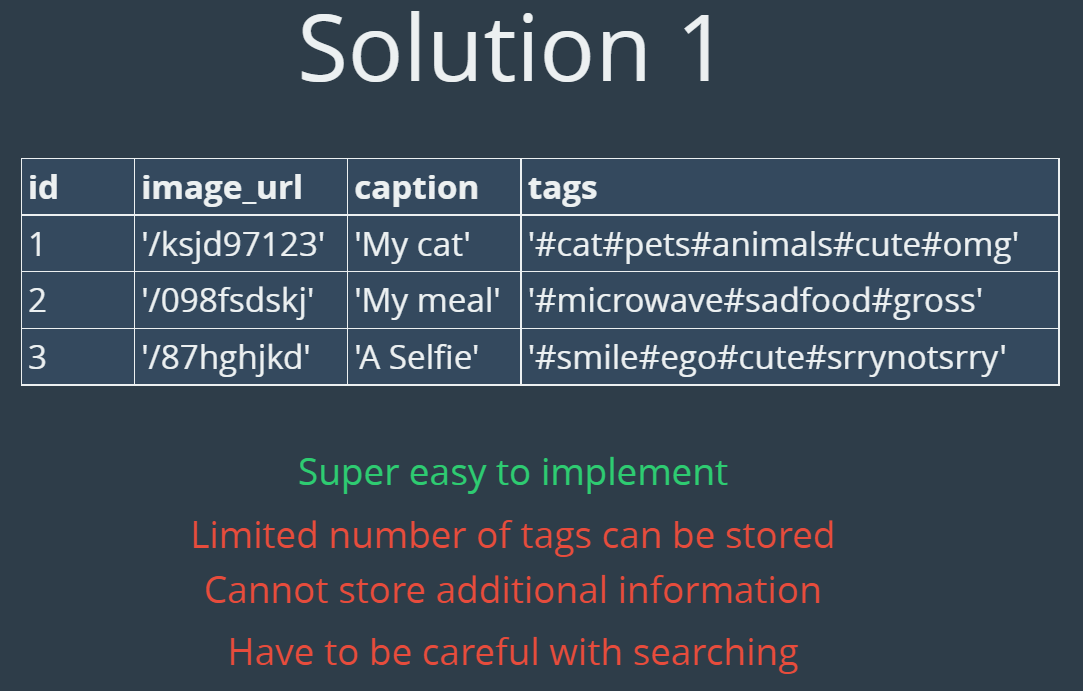
FOREIGN KEY(followee\_id) REFERENCES users(id),

PRIMARY KEY(follower\_id, followee\_id)

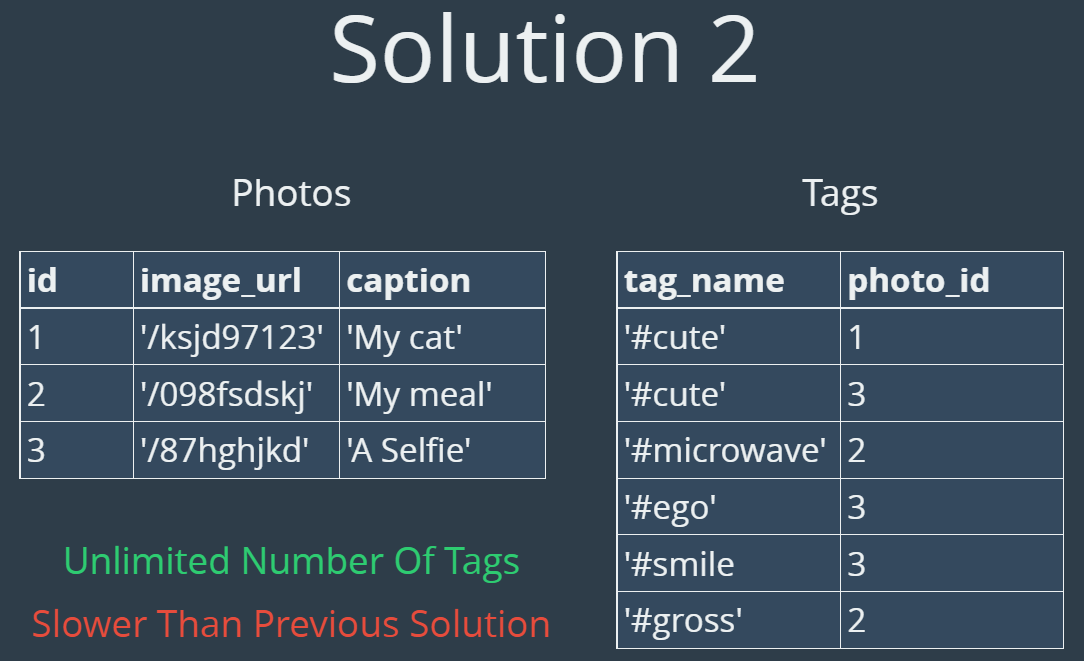
);

# Hashtags Part 1

* Hashtags are the most challenging entity to implement
* One photo can have up to around 30 hashtags, which allow you to “tag” a photo and have it be “related” to all other photos that have that tag
* There are multiple ways we could go about handling hashtags.
  + Adding a column called “tags” to the *photos* table
    - Advantages
      * Would allow you to use the LIKE function to find the tags that have that, well, tag!
      * Very easy to implement
      * But you cannot
    - Disadvantages
      * Limited number of tags can be stored based on the datatype of the *tags* column (e.g. VARCHAR)
      * Cannot store additional information with your tag, such as when it was created, the first person to use it, whether it was tagged before or after another tag, etc.
      * Searching can be problematic if you’re not specific enough with how you look for tags using LIKE



* + Use two tables – one for *photos* and one for *tags*. In the *tags* column we’ll have a tag\_name and a photo\_id for which photo that tag is associated with
    - Advantages
      * You can have an unlimited number of tags associated with a given photo
      * A given tag can be associated with more than one photo
    - Disadvantages
      * You’re storing lots of duplicated strings over and over again, which is never ideal
      * This is computationally slower than the previous solution when performing operations like inserting or deleing

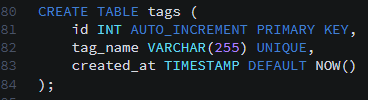


* + Best solution: use THREE tables!
    - The *photos* table is used in conjunction with a *tags* table and a *photo\_tags* table.
    - The *tags* table includes a tag “id” as well as a “tag\_name”. Each unique id is a different tag
    - The *photo­\_tags­* table associates each photo with a tag\_id. Each line is an instance of a hashtag being applied to a photo. Photos can appear multiple times, as many times as it has tags
    - Advantages
      * You can associate an unlimited number of tags to any given photo
      * Allows you to add additional information, such as when a tag is first created (within the *tags* table)
    - Disadvantages
      * Takes more work when inserting or updated, such as when the hashtag is new to the database and has not been used
      * You have to worry about orphans. If a tag gets deleted (as Instagram sometimes does), you have to remove it from the *tags* table and the *photo\_tags* table

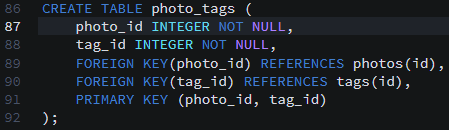


# Hashtags Part 2

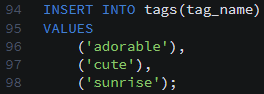
* Let’s go ahead and create our tables
* *Tags* table
  + Remember we’ll have a tag **id**, **tag\_name**, and a timestamp for **created\_at**

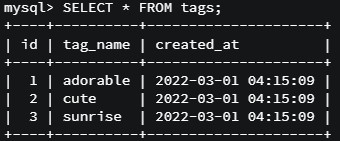


* *Photo\_tags* table
  + We just need a photo\_id and a tag\_id. Both are FOREIGN KEYS and are NOT NULL
  + We also want sure that a photo does not receive the same exact hashtag more than once. For that, we create a PRIMARY KEY as a combo (photo\_id, tag\_id)

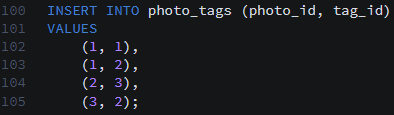


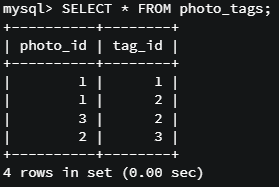
* Now let’s insert some tags for practice
  + The *tags* table





* + The *photo\_tags* table





* + Checking database integrity



* Code summary

CREATE TABLE tags (

id INTEGER AUTO\_INCREMENT PRIMARY KEY,

tag\_name VARCHAR(255) UNIQUE,

created\_at TIMESTAMP DEFAULT NOW()

);

CREATE TABLE photo\_tags (

photo\_id INTEGER NOT NULL,

tag\_id INTEGER NOT NULL,

FOREIGN KEY(photo\_id) REFERENCES photos(id),

FOREIGN KEY(tag\_id) REFERENCES tags(id),

PRIMARY KEY(photo\_id, tag\_id)

);