CS-310 Scalable Software Architectures

Lecture 17:
Twitter Design Exercise

Steve Tarzia

Summarizing the quarter so far!

Finally, we have an *end-to-end* view of a basic scalable architecture!

(for *services*, at least)

- *Frontend:* Client connects to "the service" via a **load balancer**.
  - Really, the client is being directed to one of many copies of the service.
  - Global LBs (DNS and IP anycast) have no central bottlenecks.
  - Local LBs (Reverse Proxy or NAT) provide mid-level scaling and continuous operation *(health checks & rolling updates)*.
- *Services:* Implemented by thousands of clones.
  - If the code is **stateless** then any worker can equally handle any request.
  - OS/VM can be abstracted away: develop serverless functions or containers.
- *Storage:* Distributed data stores can handle many requests in parallel.
  - NoSQL DBs are implemented as **distributed hash tables** (shared nothing).
  - SQL databases can scale (but not infinitely) with read-replicas or sharding.

Twitter design example

- Imagine it's represented by a SQL database with three tables.

currently logged-in
user: 17055506

tweets table

id	sender_id	text	timestamp
20	12	just setting up my twttr	1142974214

follower_id	followee_id
17055506	12

follows table

users table

id	screen_name	profile_image
12	jack	1234567.jpg

Original/simplest design

- When a tweet arrives, just copy it to the tweets table. Writes are fast.
- What about reading a user's feed?
- JOIN tweets and follows table.
  - Reads are slower than writes :(

```
SELECT tweets.*, users.* FROM tweets JOIN
users ON tweets.sender_id = users.id JOIN
follows ON follows.followee_id = users.id
WHERE follows.follower_id = current_user
```

currently logged-in
user: 17055506

tweets table

id	sender_id	text	timestamp
20	12	just setting up my twttr	1142974214

follower_id	followee_id
17055506	12

follows table

users table

id	screen_name	profile_image
12	jack	1234567.jpg

Why is building my twitter feed slow?

STOP and THINK

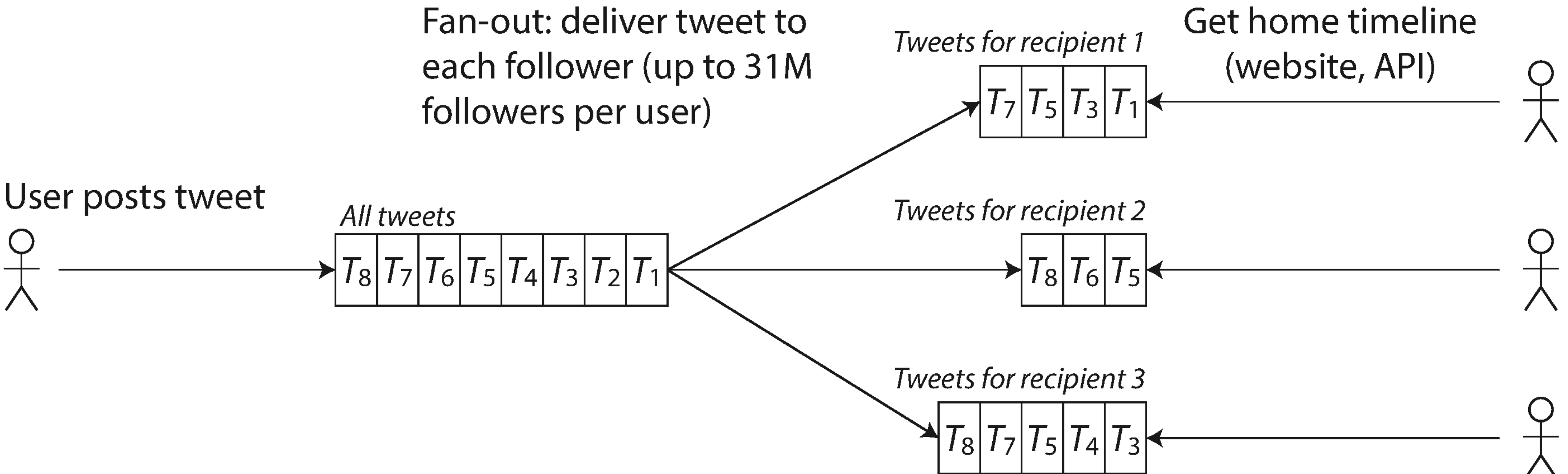
Tweets

id	sender	text	time
1	12	just setting …	2020…
2	4	Hey y'all, …	2021…
3	843	eating this …	2021…
4	12	What are …	2021…
5	234	Found a …	2021…
6	523	Picard manag…	2021…
7	4	in my car…	2021…

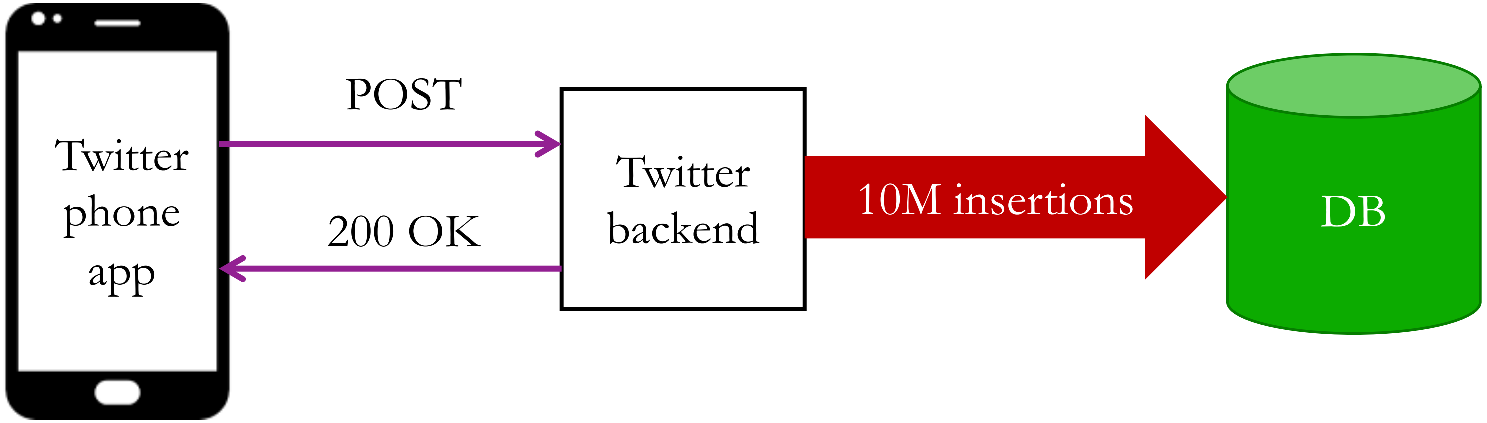
- It reads from three different tables:
  - Users, Follows, Tweets.
- More importantly, the tweets in my feed are **scattered** throughout the Tweets table.
  - Disks and RAM are both much better at reading large blocks of contiguous data.
- If the Tweets table is sharded, it reads from multiple shards.

Second Twitter design

- Pre-build feeds. Schema is *denormalized* – each tweet is duplicated and stored on all follower's feeds. Store feed data in a NoSQL database.
- Each tweet (write) now requires writing to *many* user feeds (maybe millions!)
- Do we want to make tweeting slow for people with 10 million followers?



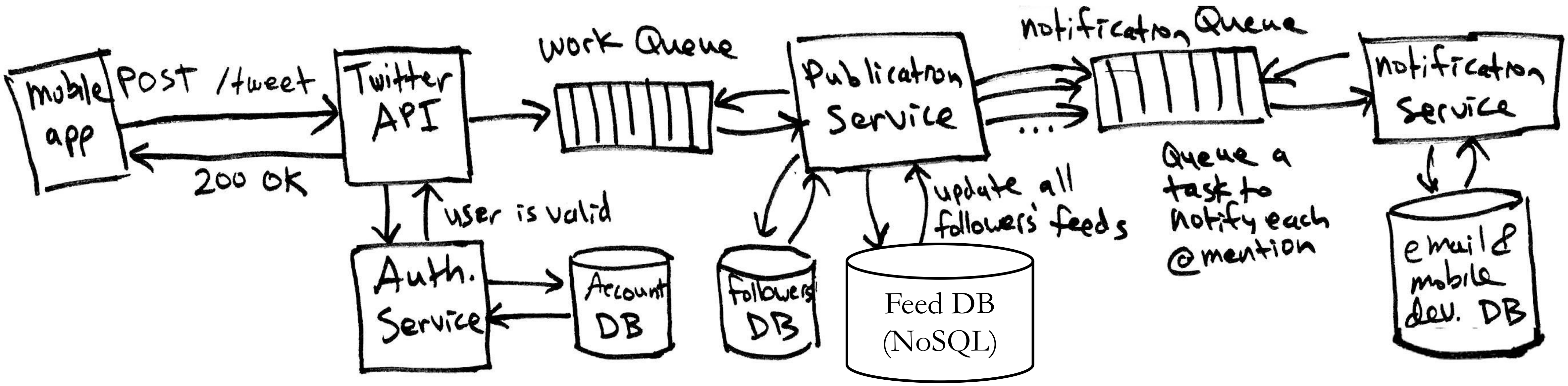
The slow celebrity tweet



- A celebrity's tweet triggers 10M database writes, so the request could take up to a full minute to complete!
- Solution? 

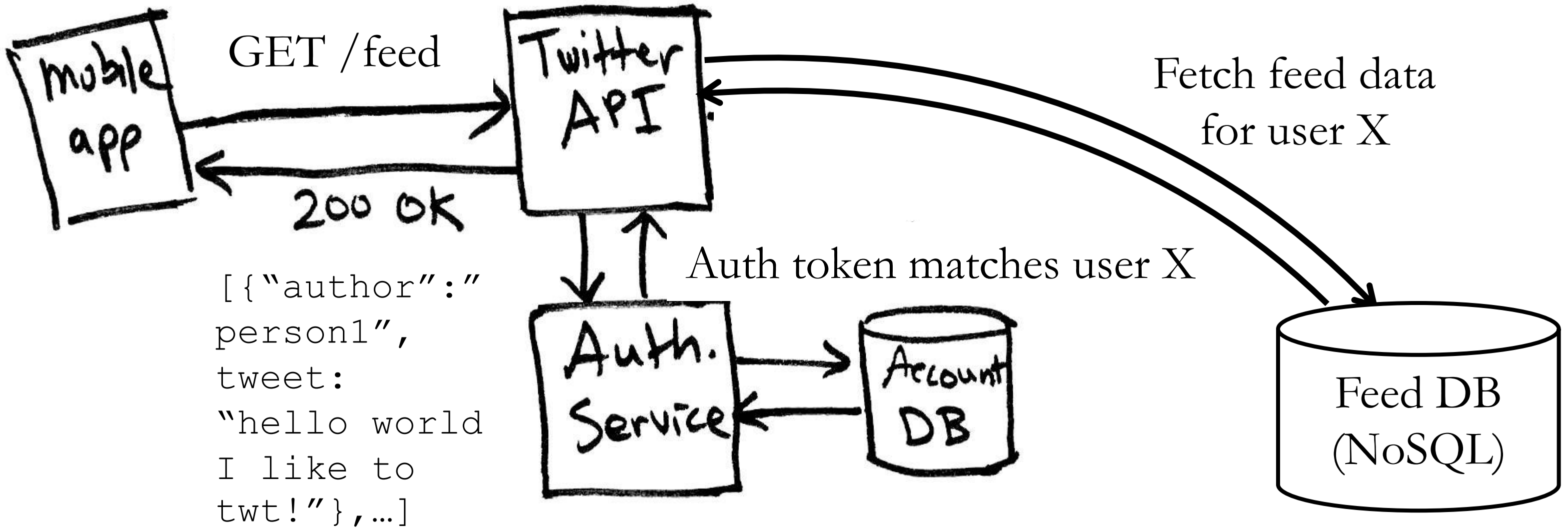
    1. Do the writes asynchronously.
    2. Store celebrity tweets differently.

Our theoretical Twitter architecture 2.0



- First design used a relational database and did a JOIN to build feeds.
- But pre-building each user's feed will make reads much faster.
    - Also, allows us to use a NoSQL database, putting all of a given user's data in one place that's easy to find (using a distributed hash table).
    - Each tweet must be duplicated to all followers. Do this **asynchronously**.

Getting your feed



The common case (reading a feed) is synchronous and efficient.

1. Validate the authentication token and get the userId.
2. Query a NoSQL database for the feed, with the userId as the key.
   - All of the users' data is on one set of replicas (maybe 3 nodes) so it's scalable.
3. Build and return a JSON object to the client.

Review

- NoSQL databases can be designed as shared-nothing distributed systems.
- Clients can find servers without consulting a centralized resource.
- Servers need not coordinate with each other.
- Each request involves a constant number of servers

To handle larger crowds, just keep adding more ticket booths.
Sales are independent, so this is a shared-nothing distributed system.

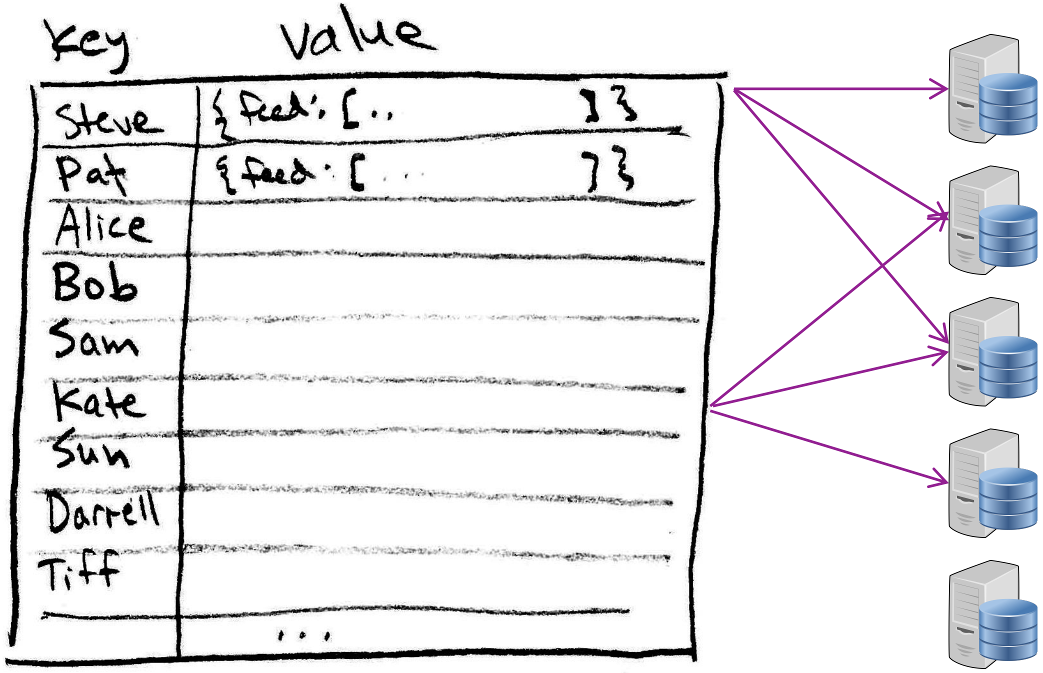
Thus, regardless of the number of clients or servers:

- The number of clients that can be handled scales directly with the number of servers. This is **perfect scalability**.
  There is no overhead for growing the system.

Twitter feeds in a NoSQL database

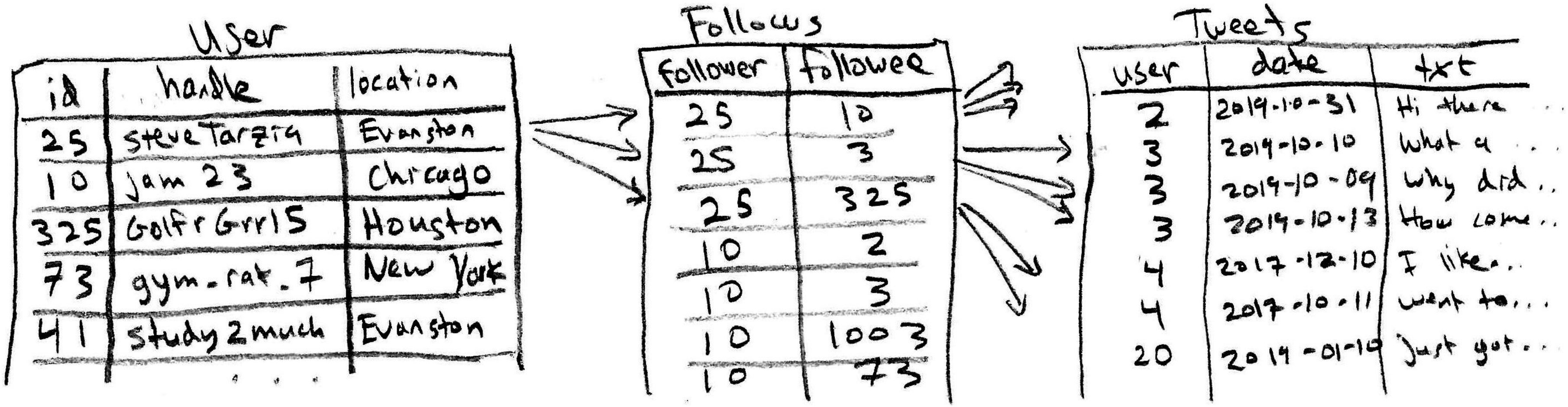
- We must somehow store everything using the key-value abstraction.
- Keys are users, value includes the latest feed data and other items that are commonly needed.
- Hash the key (user) to assign each user's data to set of replicated storage nodes:

Notice that a tweet now requires writing to all of the followers' feeds! Data is duplicated! This is called denormalization.



Twitter in a Relational Database

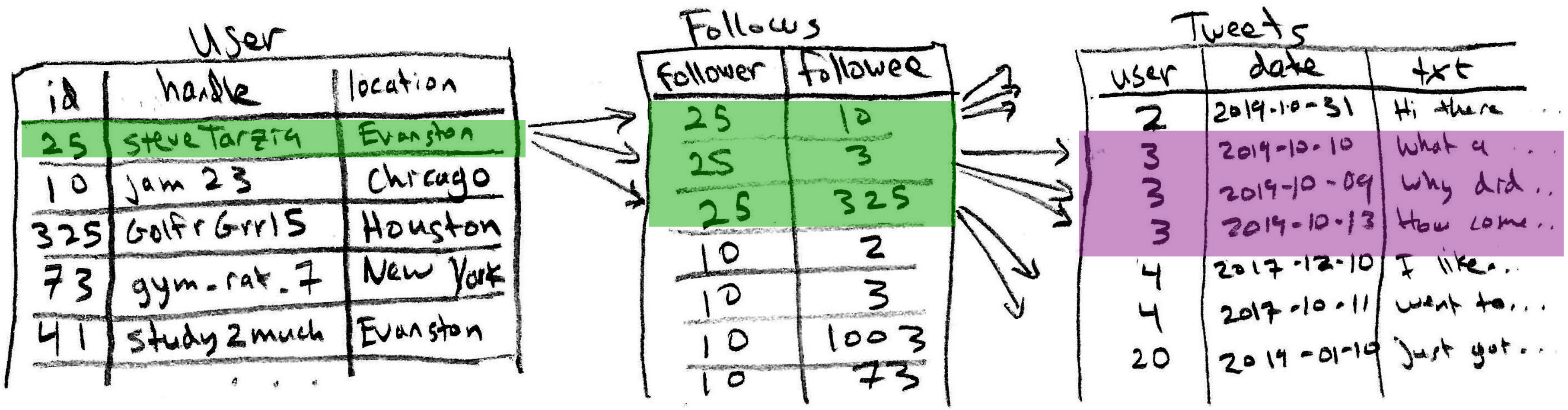
- A relational database would give a more logical design.
- Data is normalized, without any duplication.
- A JOIN is done to build a user's feed:



- If the system gets large, we must **partition** the data into multiple storage nodes, and this presents a problem in the Tweets table.

Partitioning the Tweets table

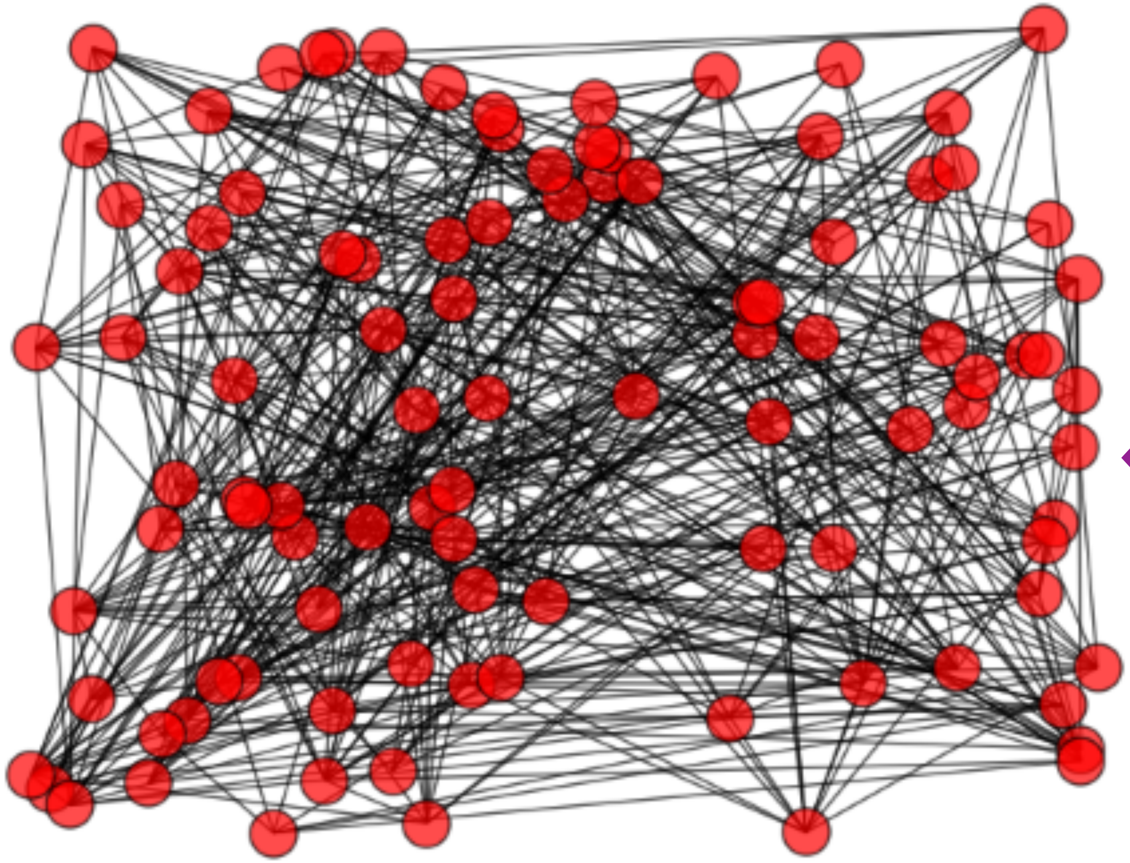
Green: partition 1



We cannot avoid conflicts when assigning tweets to partitions! 😞

- For scalability, we want to JOIN to only involved a few partitions.
- Follows table can be reasonably partitioned:
  - Place follow rows in the same partition as the follower's user row.
- However, Tweets must be quickly accessible to all followers.
- Followers can be many and diverse, & distributed on many partitions.
- Assigning a tweet from user 3 to partition 1 is great for SteveTarzia, but it's probably not the ideal placement for most of the other followers.

Data partitioning problem



A random graph, representing a set of people (red dots) with random twitter follow relationships (black edges)

- We want to split the data into partitions (storage nodes) such that:
  - Related data is on the same node.
  - Thus, queries can be served by one (or a few) nodes.

However, human social networks are not orderly, there are lots of random connections.

  - Thus, the table of Twitter "follower" data cannot be cleanly partitioned.
  - Any balanced partitioning of the graph will lots of edge crossings.

Twitter storage tradeoff

How can Twitter get the best features of both designs?

Relational Design:

- Writes are fast/simple.
- Cannot handle lots of data/users.
- Reads are slower.

Pre-built Feeds:

- Can use NoSQL,
  so much more scalable.
- Duplicates tweets.
  - Very wasteful for **celebrities** with millions of followers.
- Writes are slow.
  - Celebrities' tweets may not reach all user feeds within 5 seconds.
  - Lots of publication work is done.

Hybrid Design – Twitter 3.0

- Pre-build feeds for most users.
- But celebrity tweets are stored in a small relational database.
- Fetch a user feed in two steps:
  - Get normal-user tweets from pre-built NoSQL feed.
  - Query relational database read-replica to get recent tweets from any celebrities that the user is following.
- Celebrity tweets are relatively rare, so a single primary SQL database can handle these writes.
  - Many read-replicas handle the reads.

Twitter Architecture Recap

- Twitter's storage design choices offer a tradeoff between:
  - Relational DB: space-efficient, fast writes, but slow reads.
  - NoSQL DB: duplicative, slow writes, but fast reads.
- A hybrid design is ideal:
  - Most users are **consumers** (reads > writes): put their tweets in NoSQL.
  - **Celebrities** are different (writes > reads): put their tweets in SQL.