

# Instagram + Twitter + Facebook + Reddit

## Objectives

- 1) Users can quickly see who they are following and who follows them
- 2) We can quickly load all posts for a given user
- 3) Low latency news feed from posts that a user follows
- 4) Posts can have configurable privacy types, as do followers
- 5) Users can comment on posts, and comments can be infinitely nested

Reads >> Writes!!

## Capacity Estimates

- 1) 100 characters a post, ~100 bytes for text, maybe ~100 for other metadata
- 2) 1 billion posts per day  $\rightarrow$  1 billion (200 bytes)  $\cdot$  365 days/year = 73 TB/year
- 3) On average users have 100 followers, some "verified" users have millions
- 4) Comments also 100 characters, so around 200 bytes with metadata
- 5) Most comments on one post = 1 million,  $1 \text{ million} \cdot 200 \text{ bytes} = 200 \text{ MB}$

## Fetching Follower/Following



get Followers(userId) }  
get Following(userId) }

We want these queries to return quickly!

User Id	followerId ...
4	22
4	5
6	23

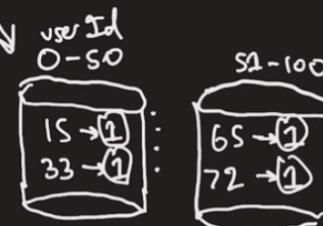
faster for getting  
all of the people  
a user follows!

followerId	userId ...
5	4
22	4
23	6

OR

faster for getting all  
of the people  
a given user follows!

| If we choose one  
| or the other, the othe  
| type of query will be  
| super slow especially  
| in the distributed  
| setting!

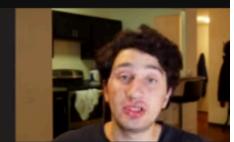


If we index on userId getting all followers will be slow and if we index on followerId getting all the people the user follows would be slow

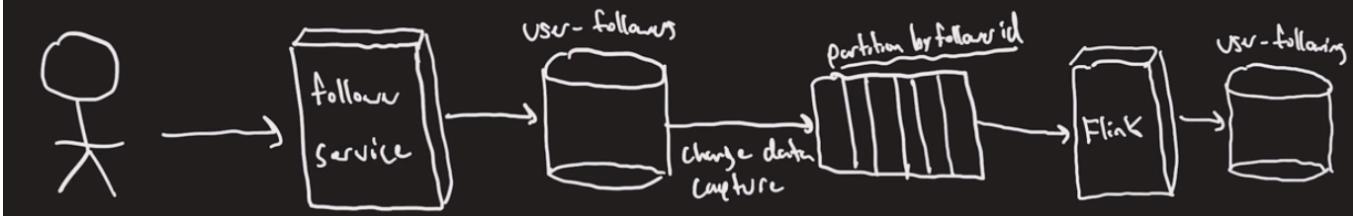
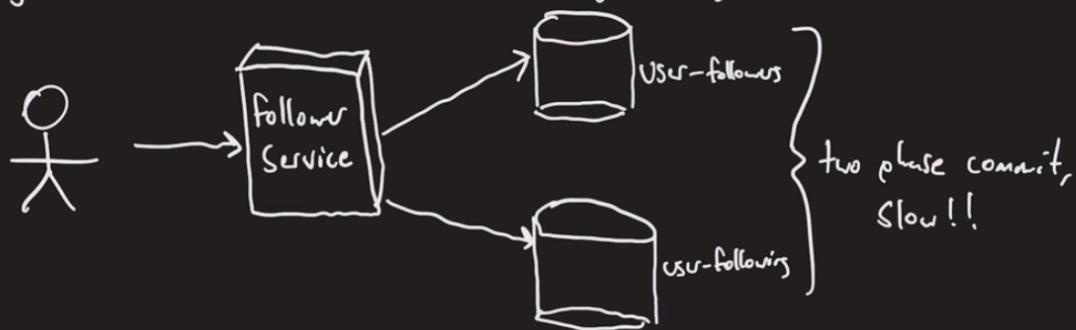
this is becuze in the distributed setting we would have to do a linear scan of all the partitions and then aggregate it which would be quite slow.

so we will use cdc/ change data capture

## Follower/Following Tables

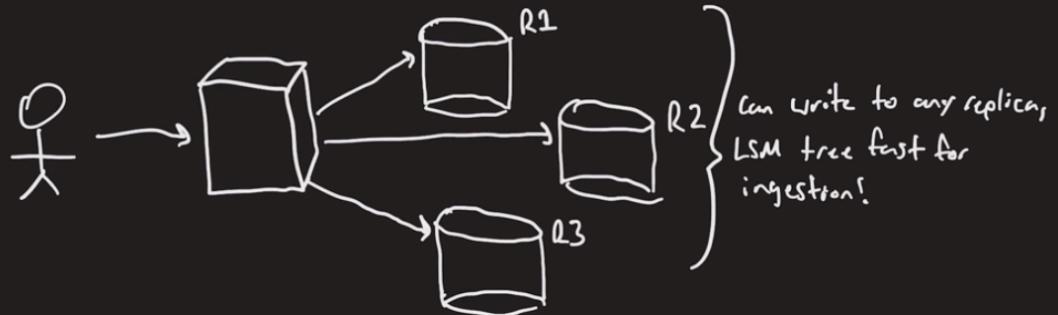


If having just one of these tables isn't good enough, what if we have both?



## Follower / Following Database

There are a lot of writes to these tables, so at least the user-following table needs to be able to ingest them quickly! Let's use Cassandra!



→ Do not need to worry about write conflicts, every write is valid!

→ What should our partition and sort key be?

## Follower / Following Partitioning / Datamodel

User Id → partition key, + sort  
allows us to ensure no cross partition query to get following

(follower) following Id → sort key, quick scanning if we ever need to delete a row

User followers partition 1		User followers partition 2	
1	3	6	3
1	4	6	19
1	19	20	4
18	12	20	12
18	14		

A hand-drawn diagram on a chalkboard showing two partitions of a follower/following table. Partition 1 has rows for User IDs 1, 1, 1, 18, and 18. Partition 2 has rows for User IDs 6, 6, 20, 20, and an empty row. A bracket groups the two partitions together with the text 'Partition by hash range for'.

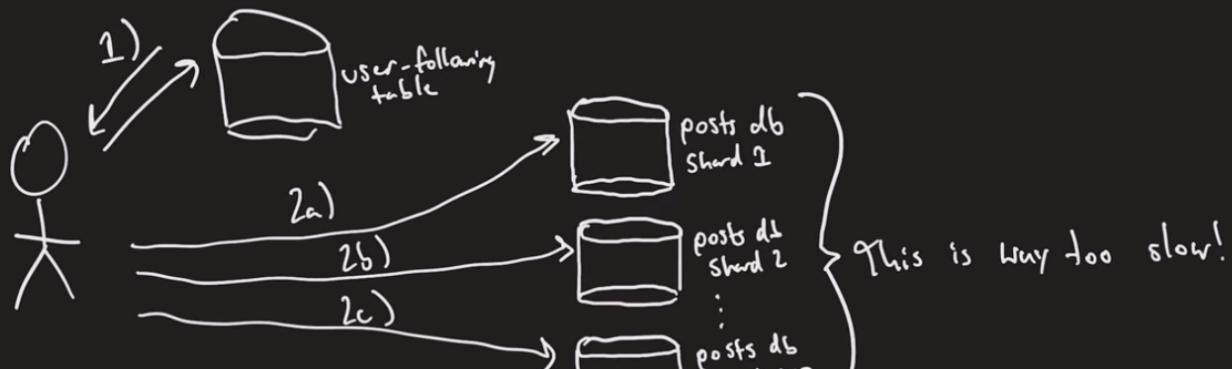
we will use hash based partitioning not range based partitioning

## News Feed (Native)

To build a news feed for user x

→ We need  $\text{following}(x)$

→ We need  $\text{posts}(\text{following}(x))$



## News Feed (Optimal)

How can we avoid reading from multiple different places??

→ Would somehow need to index tweets by what news feed they belong to

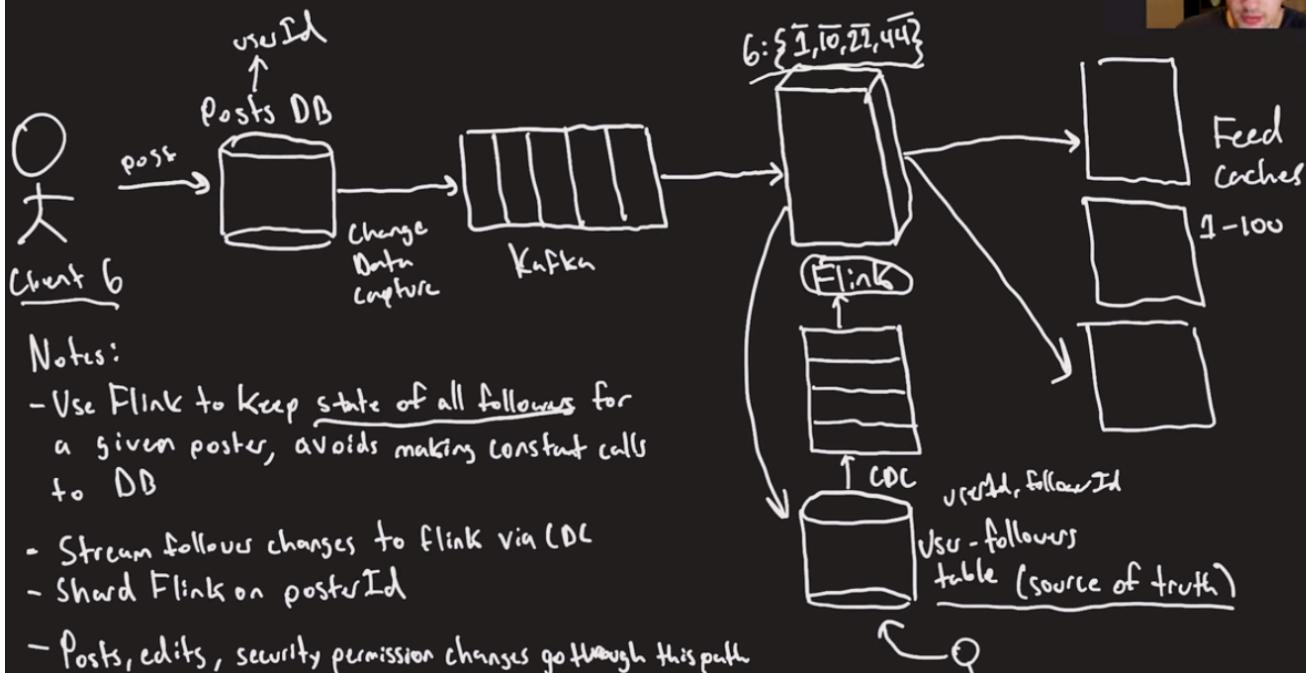
→ But a tweet can be in ~100 different news feeds!

→ 1 billion tweets/day • 200 bytes/tweet • 100 copies = 20 TB of tweets per day

→ 20 TB / 256 gb beefy hosts as caches = 80 in memory caches

→ This is doable, let's sketch it out!

## News Feed Diagram



flink will be ingesting via cdc from the user followers table all the users to send the news feed to, and then flink will send the news to all the feed caches.

## Posts Database/Schem

- In the last slide database writes need to be fast since all posts go there first!
- Utilizing some database with lots of partitioning + leaderless replication + LSM tree could help us here! (Cassandra)

User ID (partition key)	timestamp (sort key)	tweet body
69	01/06/2021	~~~~~

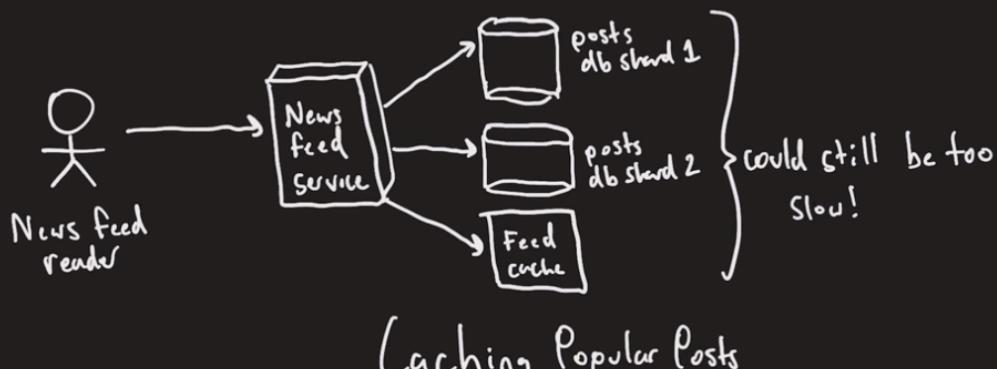
- Partitioning / sorting this way also ensures that loading all tweets for a given user is fast!

## Popular Users



Some users will have millions of followers, our previous setup won't work!

- Too many changes to followings to stream to flink
- Too many caches to update per post
- Can we use a hybrid approach?



for our verified users we will read from our posts db and for our non verified users from our feed cache

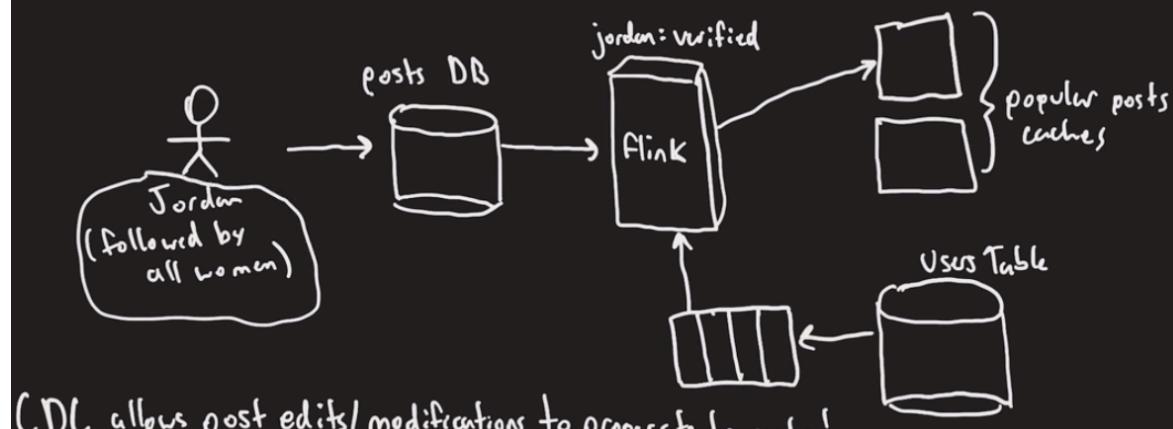
but this may be too slow so we can introduce a cache for the popular users

## Caching Popular Posts



We can introduce a caching layer for posts of popular users to speed up fetching them!

- Since we know in advance that these posts will be popular, we can use a "push" based approach to preload the data!



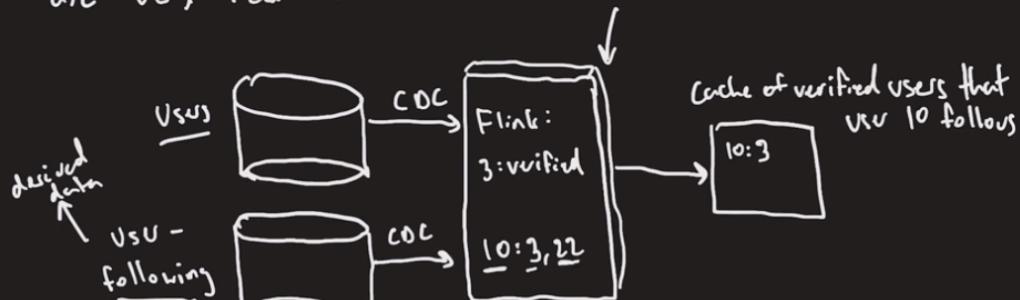
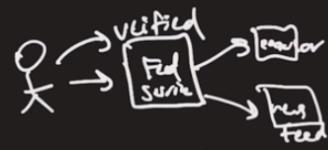
from the users table we get through cdc whether the user is verified or not and if it is verified we will populate the popular posts cache.

## < Fetching popular users that a given user follows

To perform our hybrid news feed solution we also need to be able to quickly figure out who a user follows that is verified so that we can fetch their posts from the popular cache.

→ We can again perform this using derived data!

→ Have to stream changes in verified users to each flink consumer, which is ok because there are very few verified users



## < Security levels on Posts

Let's say a user can specify whether a post is for "all" followers or "close friend" followers. Put this information in the followers table.

User-follower table:		
User	follower	Security level
1	2	all
1	3	close friend

Flink  
1:((1, all)  
((3, close friend))  
post(close friend)

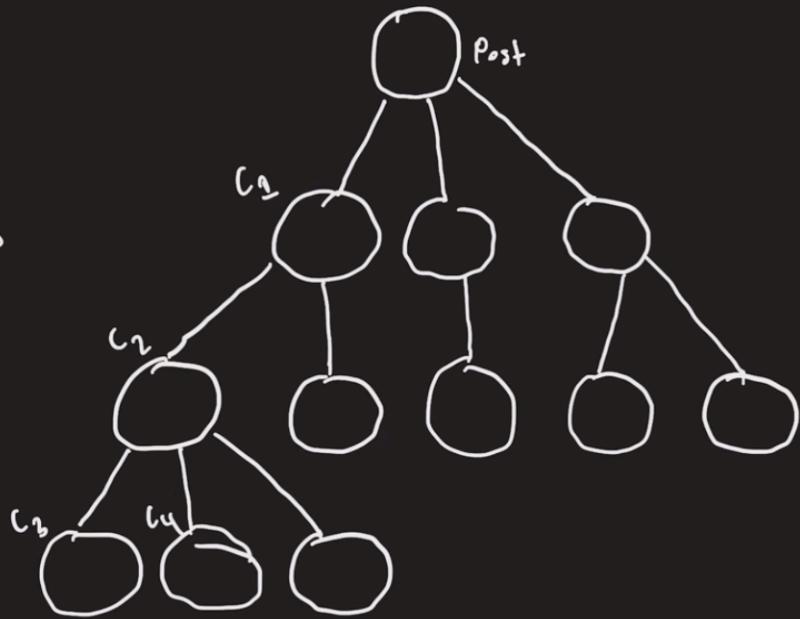
Changes to post security/follower security level will flow through our posting pipeline and eventually flow to the news feed caches

→ Unfortunately: expensive and asynchronous

# Nested Comments Access Patterns

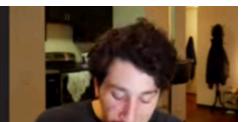


Sometimes  
we do previous  
of top comments  
too!



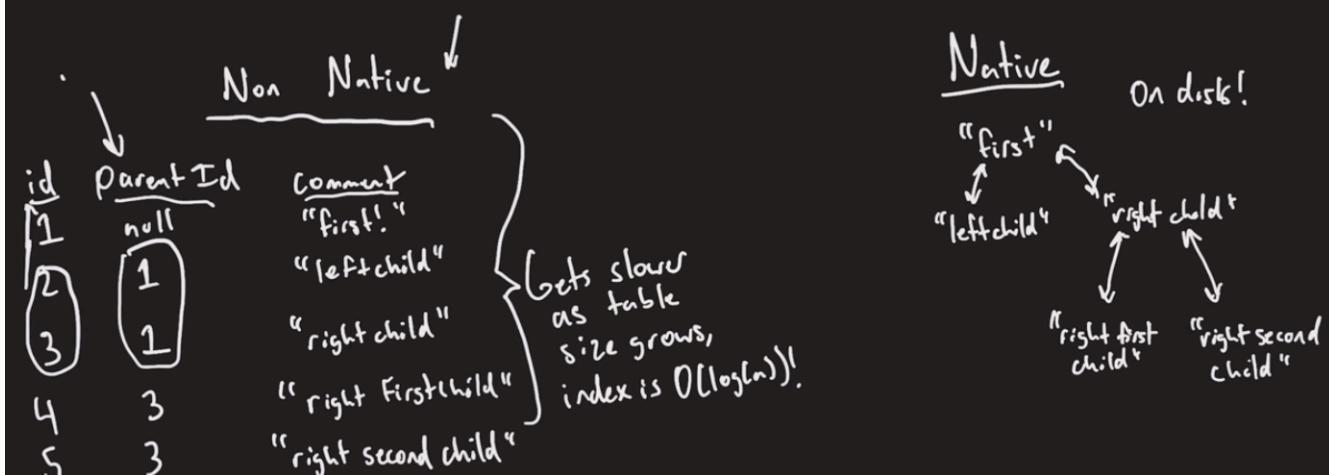
we can use graph databases for nested comments

## Graph Database

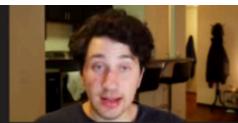


Allows us to have a generalized query approach for traversing comments.

Could use a native graph DB like Neo4J!



## Alternatives to Graph DB



Graph DBs are still going to have somewhat high latency, jumping around on disk is very slow so we should try to avoid it when we can!

Can we do anything smart with traditional DBs to make a better index?

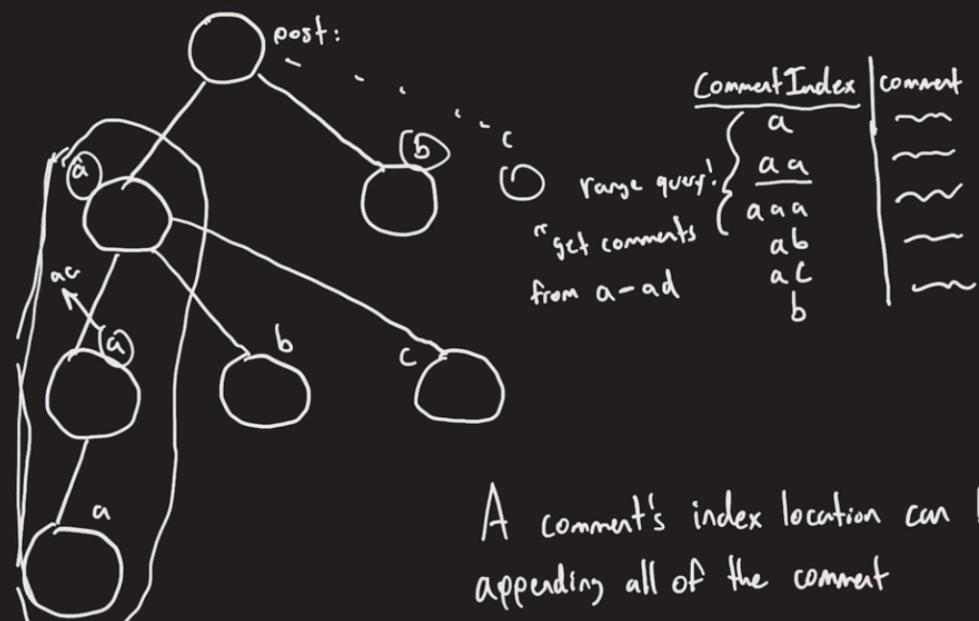
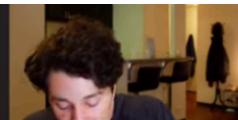
Let's try to build a depth first search index, as

breadth first search comment traversal doesn't make much sense,

We want to be able to see the parent comment when we look at a child comment!



## DFS index (similar to a Bocchari!)



A comment's index location can be generated by appending all of the comment

