Tiny Url + Pastebin

16 December 2024 19:57

REQUIREMENTS

TINY URL

PASTE BIN

1. Generate short url for each long url
2. user must be able to see number of clicks for each url

PERFORMANCE REQS

1. edian url has 10K clicks. Most popular may have MILLIONS clicks
2. At time, you may have 1 Trillion urls
3. Certain posts can allow sizes upto 1GB → avg - 1 kb → 1 Trillion * 1kb = $10^{12} * 10^3 = 10^{15}$ (PB)
5. Readspeed > write speed (in terms of count also)

LINK GENERATION

→ Hash (long-url, user ID, createTimestamp)

↳ combination minimizes collisions

DELETING / INVALIDATING

↳ simple CRON job to delete

* How many chars will your tiny url contain? 8 chars → HASH COLLISIONS can still happen

→ (0-9, a-z) = 36 chars ⇒ 36^n possibilities for n=8, $36^8 \approx$ 2 Trillion (which is more than enough)

DB REPLICATION (FASTER WRITES)

* Leaderless or multi-leader replication? NO
  ↓ USE
  SINGLE LEADER REPLICATION
  +
  Partioning/sharding on HASH (speeds up write + reads)

Lets say you get hash of url, but that already exists in DB
↳ you need to immediately find this (so that you can create another hash)
→ In multi-leader the node you write to might not have hash but another node might have it (got parallely written)

USES → Can support multiple writes since sharded across clusters
→ In case of conflict you write nearest next hash value ↳ most probably will lie in same node
→ You can additionaly use consistent hashing

LOCKING

→ Lets say 2 users try to write/update same row. You would need some locking mechanism

① Predicate Query → ② Materialize Locks

① Predicate Query
→ Obtain locks on rows which dont even exist yet
→ This needs you to scan entire table for `short-url`
↳ can be sped-up by indexing on `short-url`

② Materialize Locks
→ Make sure all possible short id rows are present in DB
2 Trillion x 1 byte x 8 chars = 16×10^{12} = 16 TB
→ You can feasibly have a DB with size 16TB with all rows pre-computed/filled

DB Engine Impl

(cant store TB data on memory)

B-TREE → BEST SUITED FOR US
→ Fast reads, slower writes
→ stored on disk

LSM + SS Tables
→ Faster write, slower reads
→ In memory + SS Table stored on disks
↳ This makes retrieval/reads slower

CACHING

HOT/POPULAR LINKS

* Can you use PUSH cache → where DB loads certain keys to pre-populate cache? NO you cant predict which url might become popular

MECHANISMS

EVICTION METHOD? → LRU

PUSH cache — keys to pre-populate cache? — url might become popular

EVICTION METHOD? → LRU

MECHANISMS

1. write back cache (NO) → you cant wait and later inform abt conflicts
2. write through (NO) → overhead to write both cache + DB (you dont need it in cache immediately)
3. write around (YES) → directly write/update in DB
   - first read will be cache miss but we r fine with it

CLICK COUNTS

1. Traditional DB (NO) → too many concurent writes
2. Atomic counters (NO) → too complex implementations
3. MESSAGING QUEUES → IN-MEMORY msg brokers (NO) → Fast but not durable
   - LOG-BASED msg brokers (YES) (use kafka) → Partition brokers based on short_urls to make it faster

MESSAGE CONSUMPTION

1. HDFS + Spark batch job (NO) → too infrequent updates
2. Flink (NO) → it processes each event/msg at a time (might get overwhelmed)
3. Spark streaming (YES) → supports 1 msg at a time processing + mini batch processing in case of bottlenecks

use EXACTLY ONCE semantics

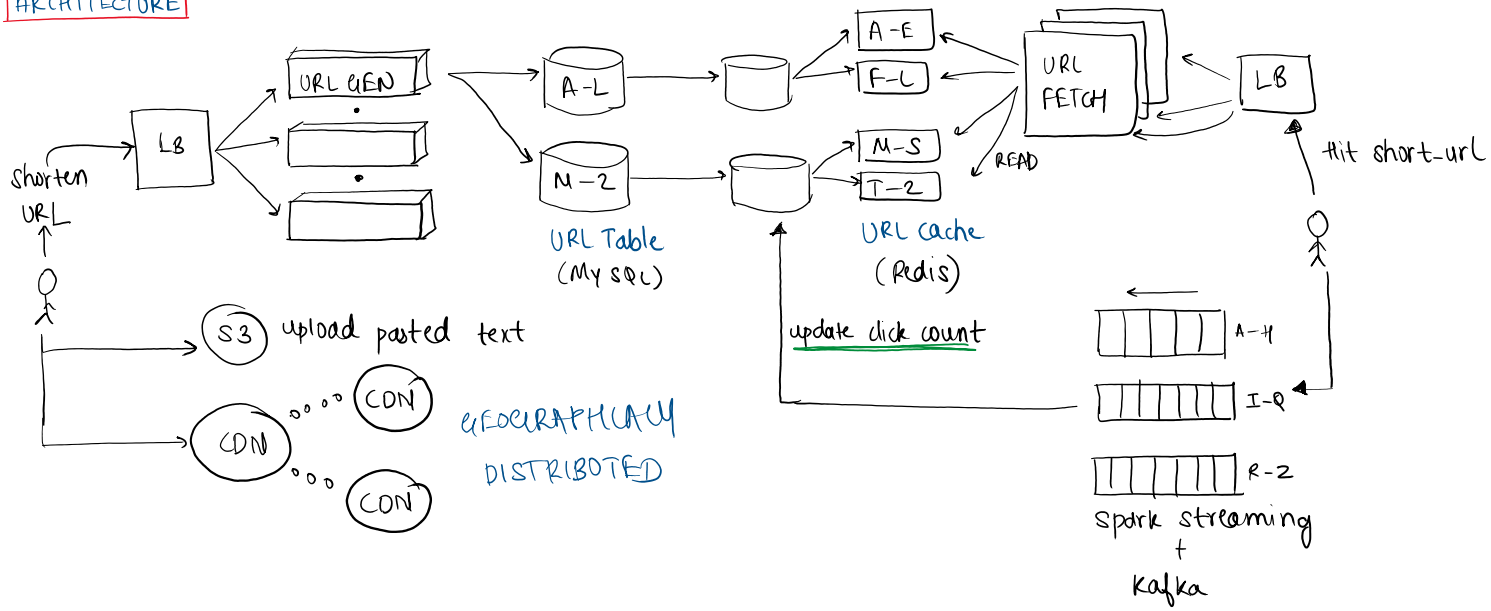
PASTE BIN - HUGE PASTES

DATA STORAGE :- Object based blob stores (S3)
- only put and fetch
- No updats or queries r done
- cheaper than HDFS / DBs

HUGE FILES → use a CDN → will distribute geographically

CACHING → write around? (NO) writes is fast but first cache miss is expensive (huge files)
write through (YES) → slower writes is fine but no initial cache miss

ARCHITECTURE



Rate Limiter

17 December 2024 08:19 PM

CAPACITY EST

1. 1B users
2. 20 services to rate limit
3. fields → user_id, count
   (8 bytes) (4 bytes)
3. Total storage needed = $10^{12} * 20 * 12 =$ 240 GB → pretty small considerably

RATE LIMITING KEY

USER_ID

+ Easy to track and block users
− Dummy/fake accounts
− Not all services need user auth

IP_ADRESS

+ No need for auth + user data
+ Multiple accounts handle
− Multiple users using same n/w (wifi)

Choice? Authenticated service? → user_id based
Unauthenticated service? → IP-address based

API format → bool rateLimit (user_id, ip-address, service_name, request TS)

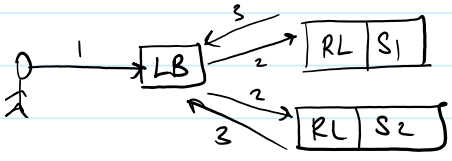
WHERE TO PLACE RL

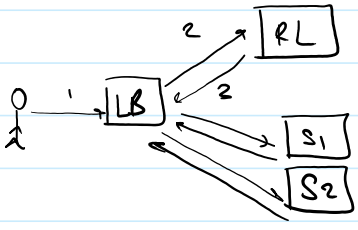
Local to service

+ less components to manage
+ lesser n/w calls
− spammed requests may use up n/w bandwidth (payload size)
− tightly coupled with service

Distributed as service → THIS IS BETTER FOR US

+ Easily scale up/down
+ Handle spammed requests easily
− Added api calls (call RL → call service)





CACHING for LB

→ Best is write back → whenever you hit LB it will cache count of hits and return the redirected url
→ (+1) increment will eventually get updated in (RL)

and return the redirected url
→ (+1) increment will eventually get updated in (RL)

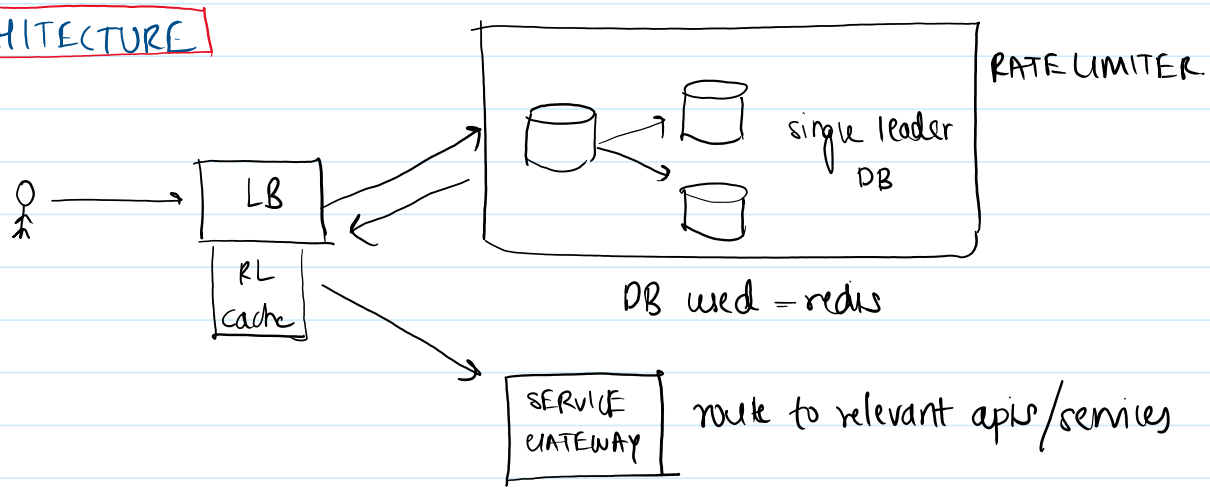
which cache to use?
(1) Disk based (NO) → we want it to be quick + data here is smaller
(2) In-memory based → (YES)
(REDIS)

DB REPLICATION

LEADERLESS / multi-leader
+ Faster write throughput due to multi-parallel writes
− Counts may not be accurate since it needs to get propagated among replicas

single-leader write → WE USE THIS AS ACCURACY IS MORE IMP
+ Accurate counts since you write to a single leader
− slower writes since single write leader

ARCHITECTURE



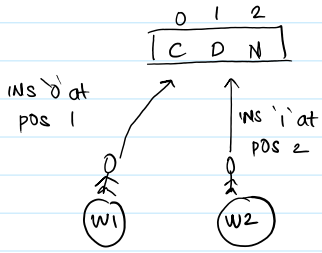
REQUIREMENTS
→ Users must be able to edit documents simultaneously
→ The updates need to be propagated to all users eventually

CAPACITY
→ Billions of users and documents
→ Each doc can have 1000s of concurrent editors/viewers
→ Doc size is in kbs

SOME BAD IDEAS
① using websockets → propagate updates / propagate entire document } These r sent by each user to a centralized server

CONS
→ File size and num_users may be huge
→ Propagate updates - you need handle conflicts

OPERATIONAL TRANSFORMS
→ Clients send writes to server as they please. Server handles them gracefully



→ if (W1) was processed then, CDN → 0 1 2 3 CODN
(W2) which was INS 'I' at (2) should become (3)
→ This is handled by operational transfers

WORKING
→ Centralized server takes in all the writes and transforms them if needed
→ Here order in which writes were processed matters
→ Can be bottlenecked due to so many users + concurrent writes

CRDT → BEST DESIGN SOLUTION HERE

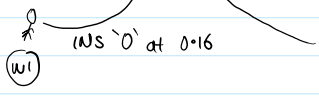
STATE CRDT
→ Send entire document
→ Merge fn merges docs
ISSUE: Doc size and writers are huge in number here

OPERATIONAL CRDT
→ Just send op: (INS (X) at pos (i))
→ Merge fn is commutative and associative (ORDER wont matter)
→ Idempotent? NO (you will need state info for this impl → using sort of a write id)
↳ if you get same op twice it should be applied once

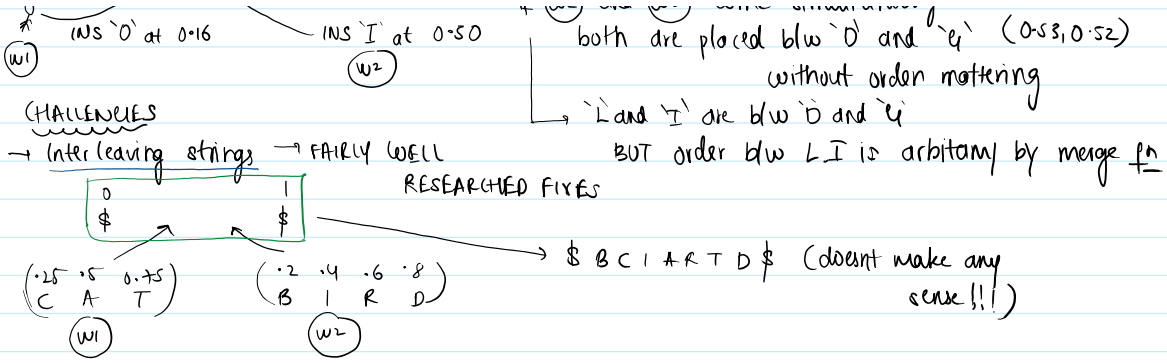
CRDT MERGE FN → Make it order independent
① First char = 0
Last char = 1
② if you want to insert a char then pos = 0+1/2 = 0.5 + some random jitter

Ex:

C	O	N	G
0	.33	.66	1



IMP
→ lets say (W3) insert 'L' at 0.52 } you add small jitter value
(W2) insert 'I' at 0.53
→ if (W2) and (W3) come simultaneously both are placed b/w 'O' and 'G' (0.53, 0.52) without order mattering



ACHIEVING IDEMPOTENCE

ways → UUID (extra space to be stored)
→ version vectors (we go with this)

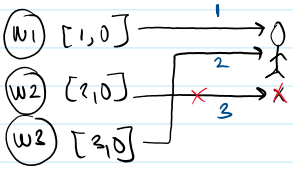
VERSION VECTORS

→ VV: which server made the write and how many writes that server did
↳ VERSION

Ex – There are 2 write nodes/servers

W1 : [2,0] → if you have processed [2,0] now and later u get [1,0] you ignore (u have already processed later version)
↳ No writes by S2
2 writes by S1

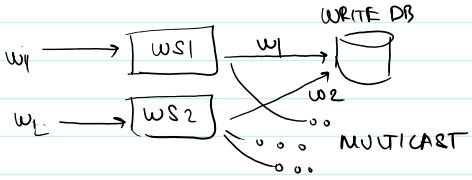
VERSION SKIP ISSUE



W2 gets rejected since W3 came first with VV [3,0]
→ so now when [2,0] came it said NO! I have seen later version

SOLUTION

write all writes to a DB before multicasting



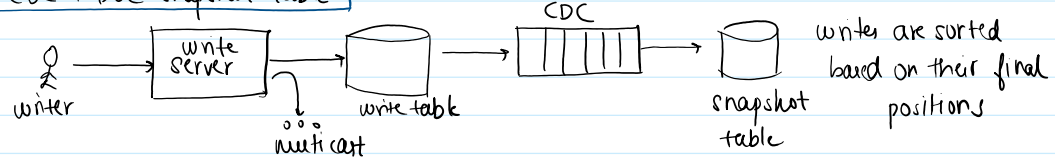
→ In case you get [3,0] first, you can query DB and check if (V-1) write exists in the DB?
→ if yes process (V-1) write and the current one next

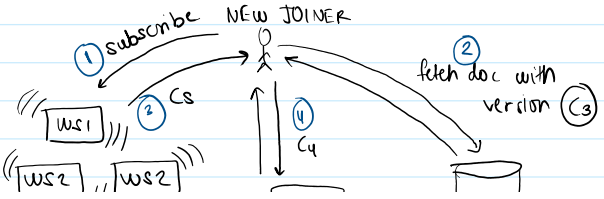
NEW JOINERS → also applies to stale users becoming active

↳ Basic: fetch the base document & apply writes

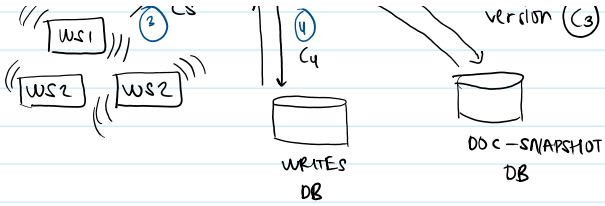
✓ BETTER

CDC + Doc snapshot table



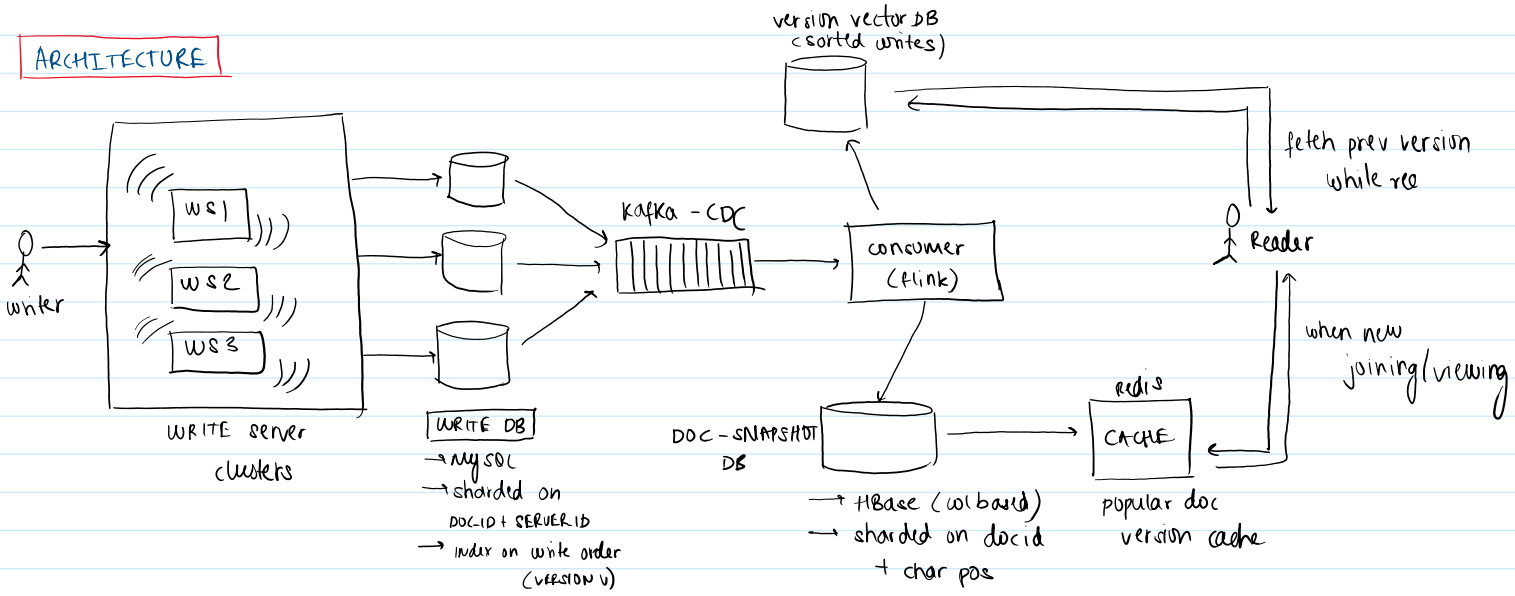


→ why 4?
→ you get document till C3 from snapshot DB
→ then you get write with VV → C5
but you check if (VV → C4) is there in write DB



→ then you get write with VU → C5
but you check if VU → C4 is there in write DB
→ YES pull C4 process it before C5

ARCHITECTURE



Netflix / YouTube

26 December 2024 15:27

REQUIREMENTS

→ Users can post, watch, search (by name) videos
→ You can comment on videos (single level of comments)

CAPACITY ESTIMATES → 1 Billion users
→ Avg views on video in 1000's (some may have millions views)
→ Avg video size = 100 Mb
→ 1 Million new videos each day

100 MB × 1 M users × 400 days = 40 PB/year storage
$(100 \times 10^6) \times (10^6) \times 400 = 4 \times 10^2 \times 10^4 = 4 \cdot 10^{16}$
$= 40 \times 10^5 =$ 40 PB

VIDEO STREAMING

① Support multiple types of devices } Basically Chunking + diff video resolutions
② Support multiple different n/w speeds

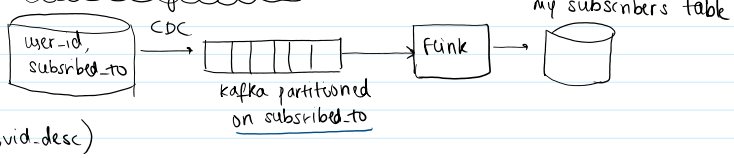
→ CHUNK + MULTIPLE RESOLUTIONS FOR EACH CHUNK

PROS → Parallel uploads (each chunk uploaded parallely)
→ Lower barrier to start video (fetch first few chunks instead of whole video)
→ Grab next chunk based on n/w speeds

You somehow maintain cur chunk + fetch lower (higher res of next chunk based on n/w speed

DATABASE TABLES

SUBSCRIBERS → Get count/all my subscribers
(user_id, subscribed_to_id)
→ Index + partitioned



USER_VIDEOS
(user_id, video_id, ts, vid_name, vid_desc)
partition | sort key

VIDEO COMMENTS
(video_id, ts, user_id, comment)
you can also use a channel_id + video_id

VIDEO CHUNKS
(video_id, encoding, resolution, chunk order, hash, url) (s3 etc.)
→ partition on video_id ; all chunks must be on same node
→ sort by video_id, encoding, resolution, chunk order
→ if your cur is (1104 + 360p) → next chunk will likely be the same

WHICH DB TO USE?

→ We have 1000x Reads than writes ; Btrees based (indexes) ones are fine
→ lots of video comments
* use CASSANDRA → supports high write throughput (multi leader repl?)
→ write conflicts are rare (we dont have nested comments)

VIDEO UPLOAD

→ This is not very frequent and can be async
→ We want this to be cheap + correct
→ for each video → split into chunks → multiple resolution
(we dont care about order of processing here)

STEPS FOR UPLOAD

① Chunk video
② upload chunks to object storage
③ Let a Queue process chunks
④ monitor till all chunks are processed

which msg Broker to use?

- In-memory (RabbitMQ)
- log based (Kafka)

→ why? 1) Order doesn't matter
2) we dont want control over delivery semantics
3) Replicating missed logs is scarse
4) NO state being stored here

which object store?

HDFS

+ Data locality (huge cluster)
+ high processing power

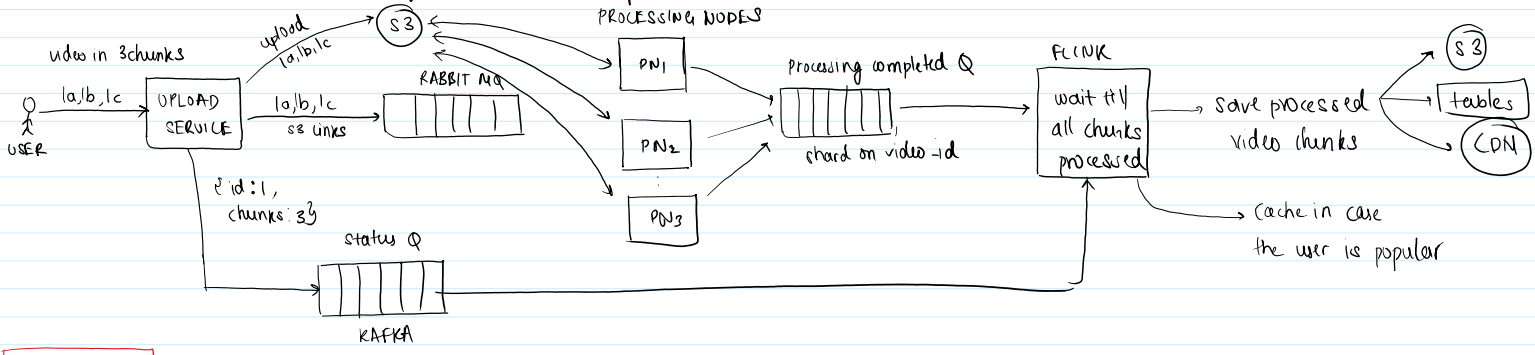
- Costly due to CPU needed

OUR CHOICE → S3

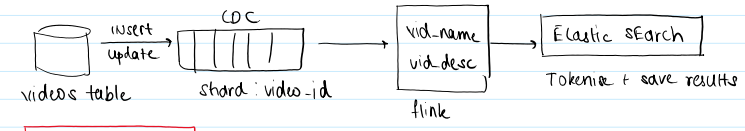
+ Cheaper storage costs
+ No cpu costs → just upload n forget

- n/w calls vary (each chunk needs a call and geographically locat? might change)

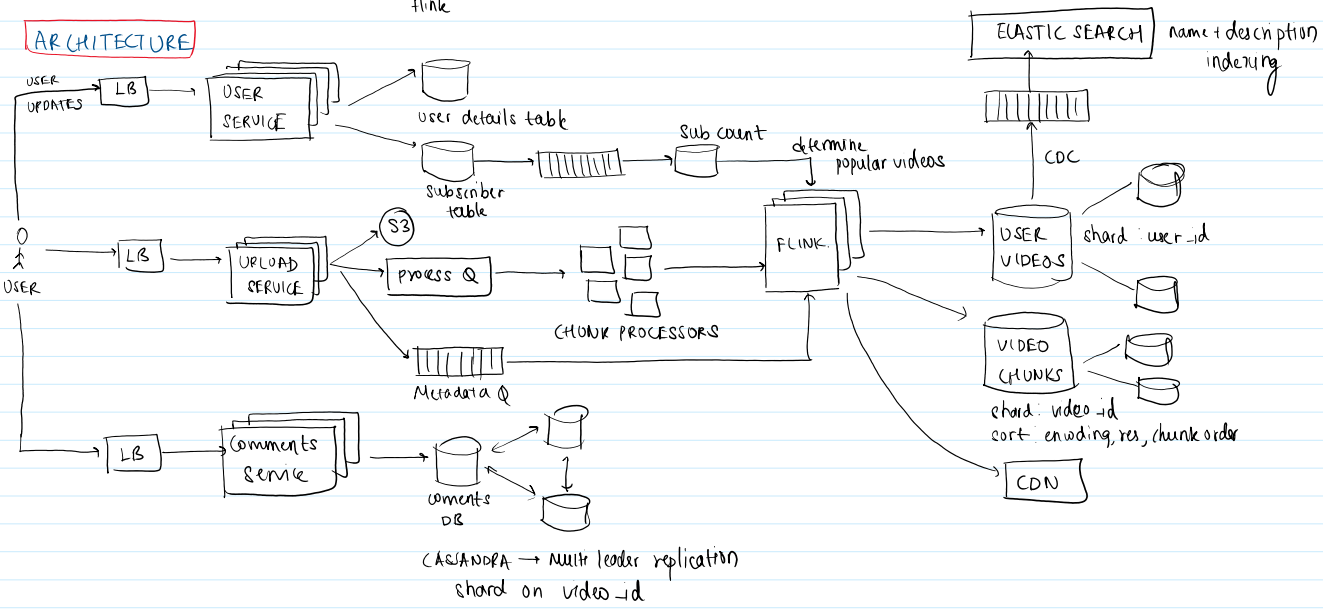
✱ we pick (S3)
→ cheaper in storage costs
→ we are fine with n/w call delays (CPU is more costly)



VIDEO SEARCH → Elastic Search



ARCHITECTURE



Type-Ahead System

27 December 2024 01:58 PM

Resources:
- System design : Design Autocomplete or Typeahead Suggestions for Google search

REQUIREMENTS

→ When typing in textbox, quickly load the suggestions
→ Top keywords are updated regularly (every 30-60 mins)
→ Include words + entire sentences in suggestions
→ Reads need to be almost instantaneous

CAPACITY ESTIMATES → assume word suggestions (NOT SENTENCE)

→ ∀ word predict 3 more words
→ 200K english words, avg len = 5 chars
→ ∀ word you need to store result of all its prefixes
apple = "a", ap, app, appl, apple

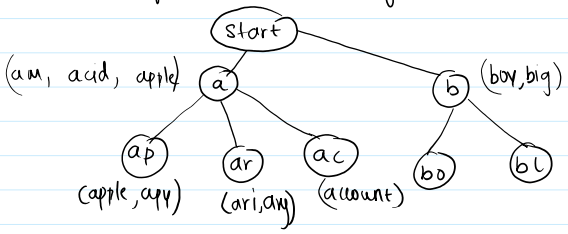
200K * 5 = 1×10^6 = 1M entries
1M x (5b word + 15b for suggestns)
= 1M * 20b = 200 MB
→ can be stored locally on client device

Query Based

→ There are 1B unique search queries each day ⇒ storing counts of each = 4B bytes = 4 GB → cant be stored locally
→ Assume avg search term is 20 chars
① 1B entries * 20 prefixes each = 20B entries
② 10 suggestions fr each word = 10 * 20 = 200b/entry
20B * 200b = 4×10^{12} = 4 TB storage each day

TRIE

→ Obviously we will be using TRIE for storing our recommendations online

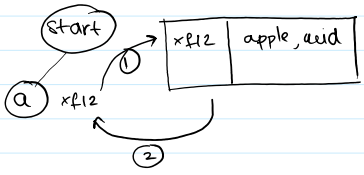


TIME COMPLEXITY
① insert word = O(len(word))
② delete word
③ search word
④ Search prefix = O(P+k)
len of prefix
len of max word with prefix (P)

→ while typing TC for getting next char suggestions is O(1)
"a" – am, acid, apple
"ap" = apple, apply
→ just one node traversal below (a) → (ap) so O(1)

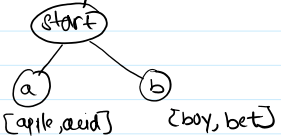
MEMORY EFFICIENT

→ store the references of suggestions at each node
→ You have to fetch it from another place – DB, disks etc
+ LESS MEMORY → only store addresses
– MORE TIME → fetch takes time



TIME EFFICIENT

→ store entire suggestions directly as list of words ∀ node
→ No need to fetch other places to get data
+ LESS TIME
– MORE MEMORY

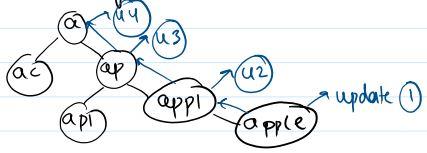


UPDATING KEYS

→ lets say for "apple" the recommendations have changed
"apple": apple buy, apple big → apple pie, apple phone
you may need O(len(word)) operations · You update each prefix of word

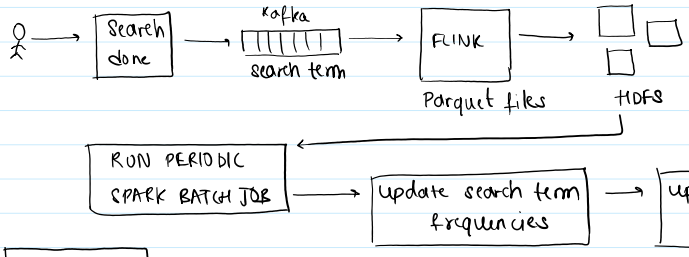
apple: "apple buy, apple big → apple pie, apple phone"

You may need O(len(word)) operations. You update each prefix of word

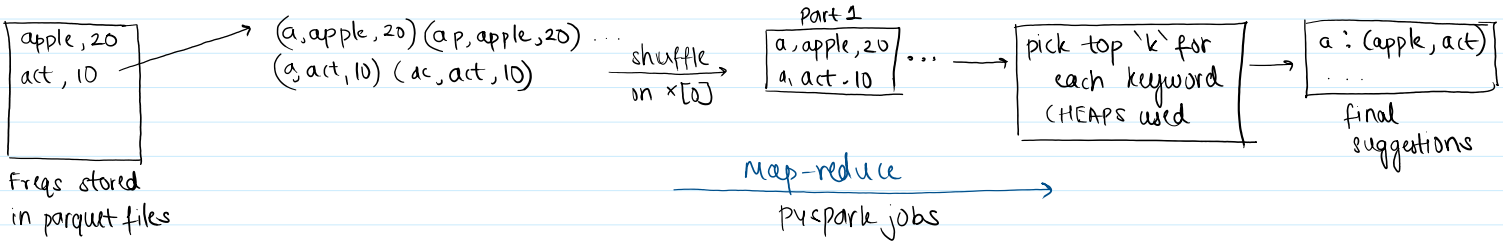


→ You might need to acquire locks for this. (In case u update 2 words with common prefixes)

UPDATING FREQUENTLY HIT TERMS — AGGREGATION SERVICE



BATCH JOB



How users will READ

(1) API calls — BAD CHOICE
→ You will make 1 api call for each character being typed
Too much overhead due to headers + payload to fn

(2) Long polling (same issues as api over HTTP)

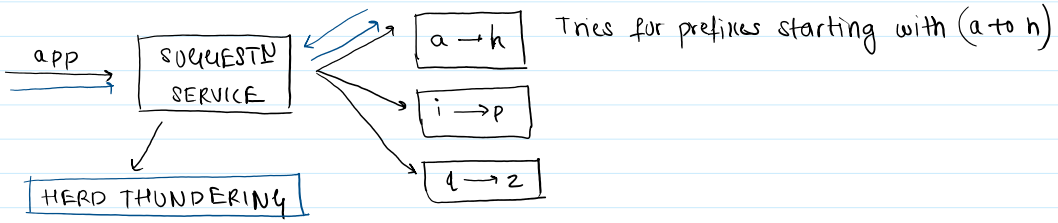
(3) SSE (server sent events) → NO (you need bidirectional flow)

SOLUTION — **WEB SOCKETS**
- Bidirectional
- No record of headers

→ You need a websocket manager
→ Zookeeper + consistent hashing + Load balancer

SCALING — DISTRIBUTED TRIE

→ You can't store all keywords in single Trie (TOO HUGE to fit on disk)
Sol? : distribute tries across machines using range based partition.



HERD THUNDERING

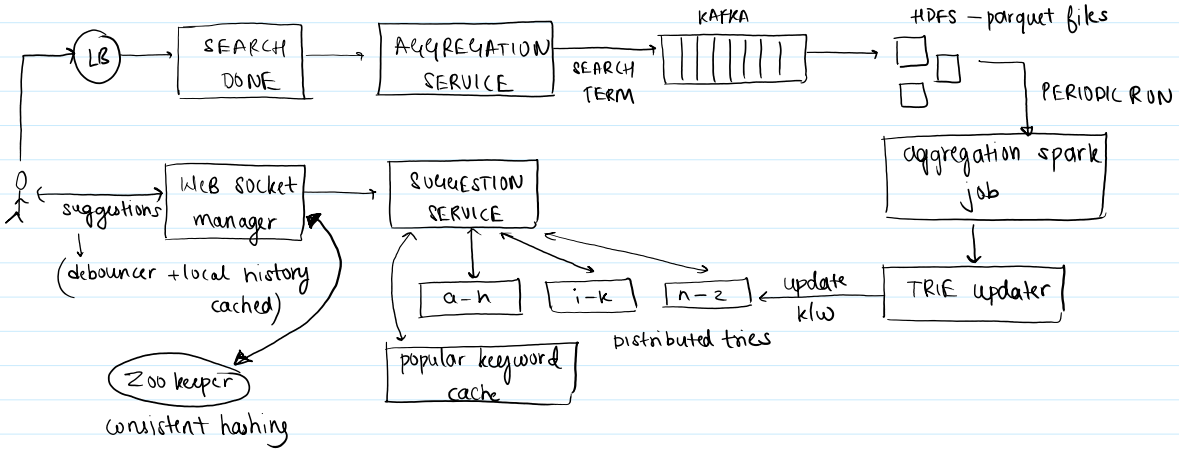
What if you get lot of requests at a time for certain keywords

→ (1) Dynamic Repartition : Reduce or increase range partitions pre-emtively
→ will take some to redistribute and recalculate (done in bg)

→ (1) Dynamic Repartition : Reduce or increase range partitions pre-emptively
→ will take some to redistribute and recalculate
(done in blg)

(2) Cache popular keywords

ARCHITECTURE



Facebook/Twitter

26 December 2024 21:19

REQUIREMENTS

1) Users can see who they follow + their followers instantly
2) Low latency news feed for each user
3) Posts can have configurable privacy types
4) Users can comment on posts (Normal vs nested both)

capacity estimates

READS >>> WRITES

1) 100 b for each tweet text + 100b for metadata
2) 1 Billion daily posts : $1B \times 200b \times 365 = 73TB$/year
3) On avg user has 100 followers : Verified users have millions of followers
4) Comments also are 200 b (100b content + 100b metadata)
5) Max comments in a popular post - 1M = $10^6 \times 200b = 200MB$

FETCHING FOLLOWERS/FOLLOWING

① get_followers (userid) → all ppl who follow user
② get_following (user_id) → all ppl who user follows

FOLLOWING TABLE

user_id	following
1	2
1	3

→ Index on user_id
→ user ① follows ② ③

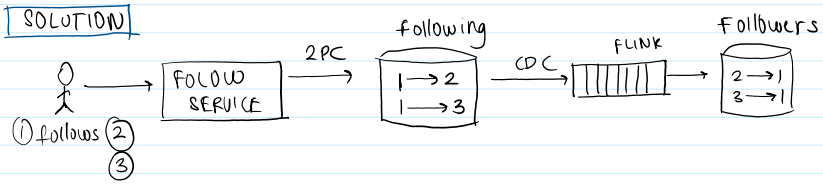
FOLLOWER TABLE

user_id	follower
1	2
1	3

→ users ② ③ follow ①

→ getFollowing (user) is efficient (indexed) → REVERSED
→ getFollowers (users) is inefficient
↳ To make both efficient you need both tables BOT, writing to both tables in extra time

SOLUTION



WHICH DB? → CASSANDRA
→ Multiple concurrent writes happen
→ No need to handle write conflict/duplicates
→ 1 follows 2] just deduplicate, nothing complex
1 follows 2

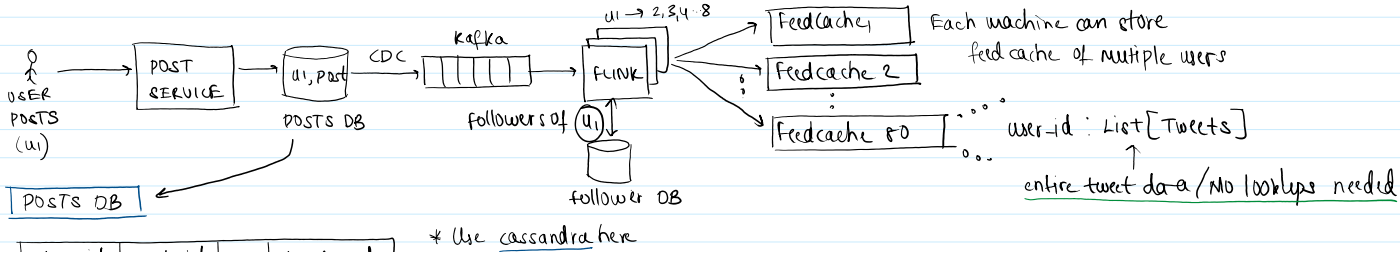
→ Partition key: user_id, sort key: follower/following_id

NEWS FEED

Naive + Bad:
↳ getFeed (user) → Fetch followers of user → fetch all posts made by ppl followed → FEED

PRE COMPUTE FEEDS FOR EACH USER? → is this possible?

→ Somehow distribute tweets to feeds they belong to
→ Assume a tweet belongs to 100 feeds
(1B tweets/day) · (200b each tweet) · (100 copies) = $2 \cdot 10^{13}$ = 20TB tweets/day (Reasonable)
20TB distributed into 256GB caches = 80 such machine caches (DOABLE)



follower DB

entire tweet data / NO lookups needed

POSTS DB

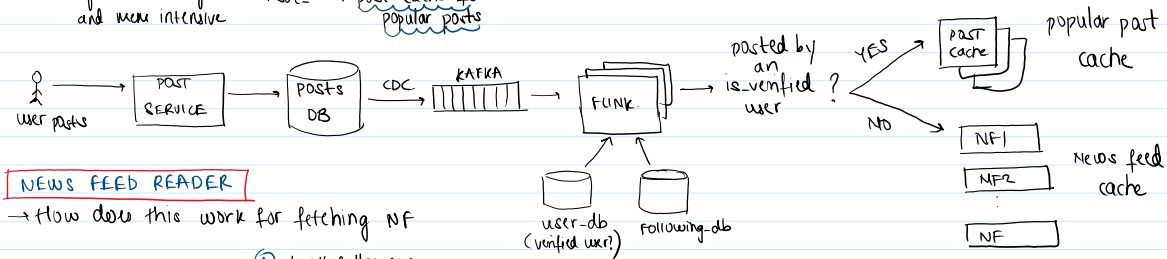
user_id	post_id	ts	tweet_body

shard — sort

* Use cassandra here
  → High write throughput
  → No write conflicts (Multiple users can't make same post, same users create post only once, update posts also happens at once)

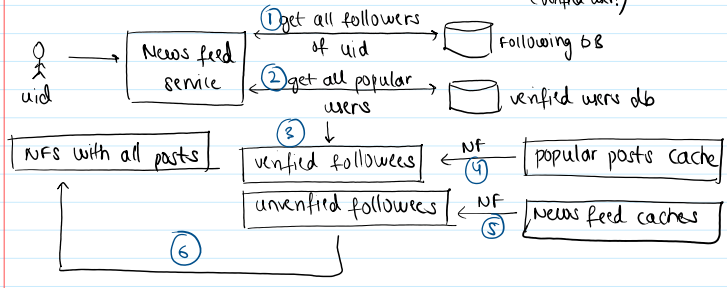
POPULAR PEOPLE POSTS

→ Lets say a popular person posts (has millions of followers OR verified)
In this case you will copy that post to IM News feed caches
↳ very time taking and mem intensive → sol'n → post cache for popular posts



NEWS FEED READER

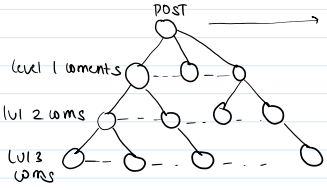
→ How does this work for fetching NF



COMMENTS

① Only one level → CASSANDRA
(High write throughput / NO conflicts)

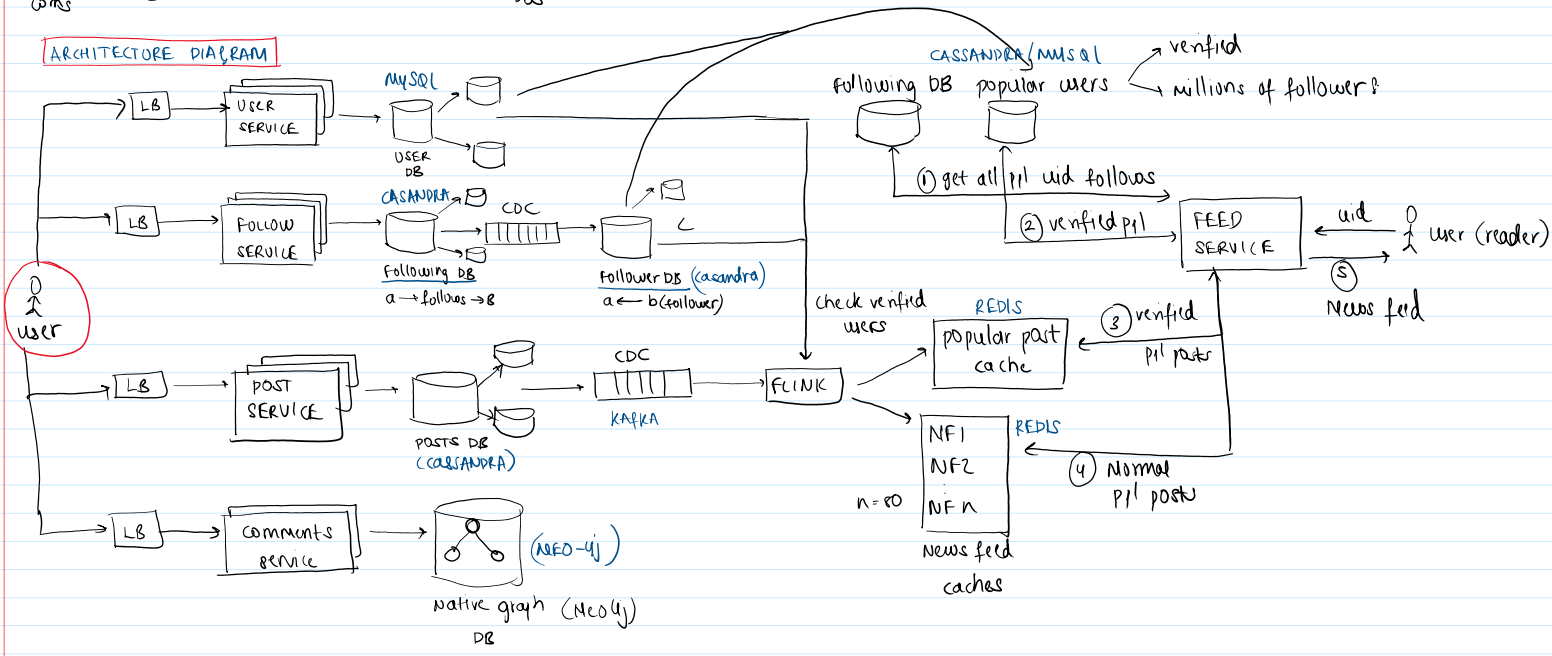
② Nested comments → GRAPH DB needed



How is it shown?
① FB, Twitter : BFS (You are shown top levels, you click to go deeper)
② Reddit : DFS (when you click a comment, entire depth is shown)

SOLN NATIVE GRAPH DBS → Neo4j

ARCHITECTURE DIAGRAM



Whatsapp

28 December 2024 12:57

REQUIREMENTS

→ Group chat with 10 members
→ Sending + receiving messages in realtime
→ Persist messages to read later (store conversations)

CAPACITY ESTIMATES

→ 1 Billion users, each sends 100 messages/day
→ Each message is 100b = $(1 \times 10^9) \times (100) \times (100) = 10^{13}$ = 10 TB/day × 400
= 4 PB/year

CHAT MEMBERS DB

→ Find all chats user is part of (Left pane in whatsapp web)

user_ID	chat_id
1	12
1	10
2	3
3	10

user_id → Index + shard

→ multiple writes may happen : single leader replication (MySQL)
→ can think of SET CRDT (too complex for this)

why MySQL? users dont frequent start chat/leave or join groups
→ performance impact significantly less

USERS DB

<userid, email, pwd_hash, usr_metadata> ⇒ MySQL Infrequent changes to this table
index + partition

MESSAGES TABLE

SERVER BASED TIMESTAMP

partition

chat_id	ts	message	metadata	uid

which DB to choose?

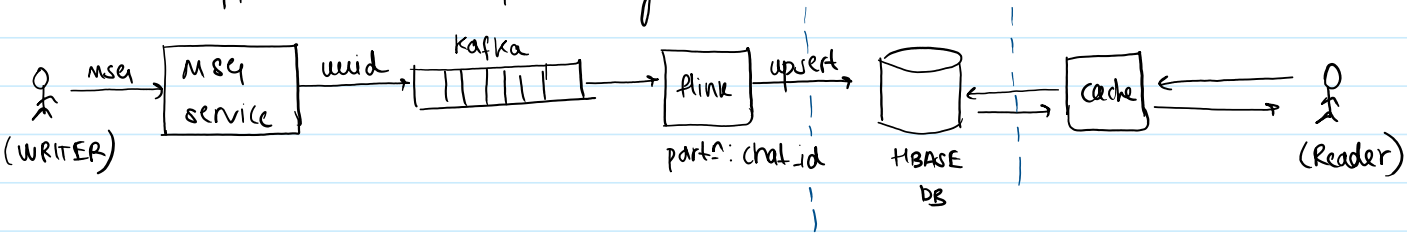
→ Reads and writes are both imp + frequent
→ Column oriented storage? → usually you read only part of columns
(metadata is usually not read)

① High writes → cassandra? dealing with conflicts
② HBase → single leader (NO need of conflicts)
column based DB
LSM Tree Indexes → faster writes

column based DB
LSM Tree Indexes → faster writes

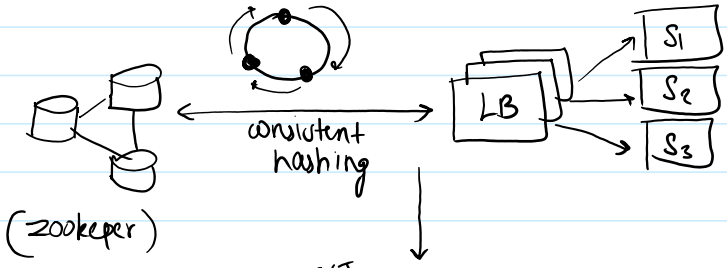
Tuning for faster writes

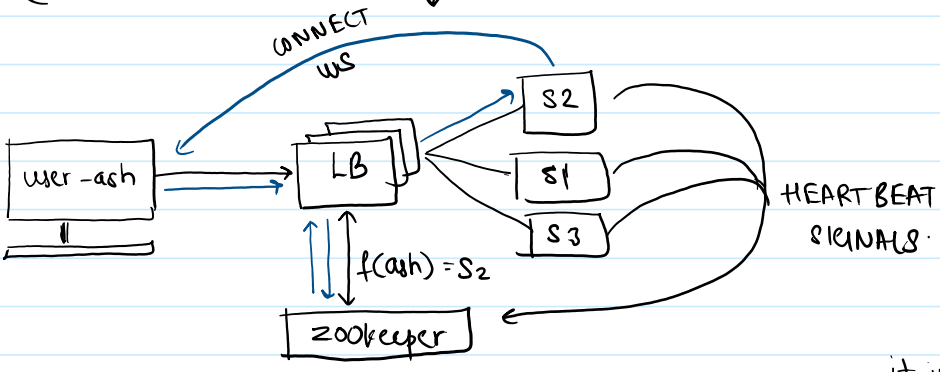
① Read directly from HBase
② Writes happen via stream processing (uuid + msg handles idempotence)



LOAD BALANCING

→ WEBSOCKETS (Allow bidirectional data transfer)
→ Load Balance on user_id : requests from user must go to same websocket
  ↳ will use zookeeper to keep track of active servers
  ↳ internally use Consistent hashing for dynamic mgmt
  ↳ LB will listen to zookeper





① user hits LB ② LB hits zookeper with user_id → it just has a map user → server
③ Zookeper updates (LB) with which server to assign (S2)
  → All servers send heartbeat to Zookeper at freq times
④ LB now connects client ⟷ (S3)

IN CASE (S2) goes down
  → LB asks zookeper again (after some interval)
  → zookeper knows (S2) is dead (didnt get heartbeat)
  → It assigns another server to user (ash) using consistent hashing

ARCHITECTURE