Basics

30 March 2024 02:07 PM

SHARDING → dividing data into smaller fragments + storing them on different distributed machines

TYPES

- Range Based / Dynamic
  - → Each shard holds a range of values
  - → Ex:- date ranges, time ranges
  - → when ordering + ranges are there in data
- Hash-Based / Algorithmic / key based
  - → use a hash fun^n on data to get which shard
  - → Used when there is no natural order (ids)
- Directory Based
  - → you manually maintain a data → shard mapping
- Geo-Based
  - → shard based on geo locations or names to locations

Partitioning → same logic but logically within same system

PROS

- → Faster and efficient queries (given you pick good sharding strategy)
- → eliminate single pt of failure

CONS

- → outage possible as it is distributed
  - soln: master-slave design
    - handles writes
    - handles reads
- → Fixed number of shards · Need rebalancing / resharding
  - soln: Consistent hashing
- → Skewed shards — Better sharding fn approach
- → joins across shards are costly

IMPROVEMENTS

→ SHARD + INDEXING

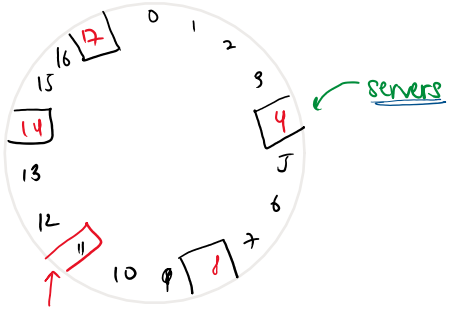
Ex:- shard on CITY + index on AGE

→ here first shard on city + Internally each shard is indexed based on age

Query:- give me all persons in DELHI with age between 20-40

CONSISTENT HASHING

* Normal hashing :- lets say you have (N) servers
  - (h(data) % N)
    - h(data) → hash fn
    - % → you can't change this
    - N → no of servers
  - if done needs rehashing everything
* Consistent Hashing
  - → Consider your slots are in circular form
  - → x → hash_fn(x) → (0 - 360°) example
  - → Each result of hash fn will lie on this circle
  - → some of these slots will be servers also



HOW IT WORKS

- → data → hf^n ( ) → slot_no = (3)
- → move clockwise and see next server to (3)
- → (3) gets stored in (4) ie (4)

New server gets added

- → (11) is now made a server
- → New indexing doesn't change (9,10 will go to (11) instead of (14))
  - Rebalancing is minimal
    - → only (9) (10) which was prev sharded to (14) need to be rendered to (11) Now

* LOAD Factor = 1/N (ideal case) equal load to each server
  - → But this isn't possible when you have sparse

soln: Virtual servers

- → Add more server slots which brings load factor to 1/N
- → Internally a single sever handles (2+ slots)
  - Ex:- (11) might be handling

→ Add more server slots which brings load factor to 1/N
→ Internally a single server handles (2+ slots)
Ex:- (8, 11) might be handled by a single physical server

ACID

→ Maintained in Relational DBs → Good at vert scale (Bad at vertical)
→ Fixed schema
→ Generally not distributed

Atomicity

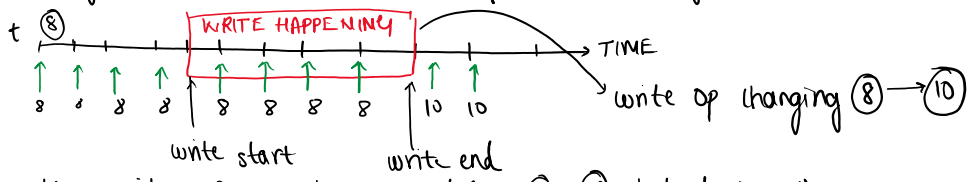
→ A txn can either happen completely or not happen at all
Ex:- Bank amt transfer
① Deduct frm ur acc → ② Add to the desn acnt
Now this shud happen completely. Ex lets say it got deducted from accnt but then failed
→ In such case revert back everything
everything or nothing approach

consistency

→ multiple reads at same time should give same results
Why easy? since its not distributed no issue of out-of-sync replicas

Isolation

→ Every transactn happens independently without caring abt other parallel transactions



→ Here write opn was changing val frm 8 → 10 but during the process any read doesn't know or care abt this and still reads as 8
→ Only after write operation is completed and state is updated, Read returns 10

Durability

Every txn is logged and data is being persisted

BASE

→ shown in NOSQL Databases → No fixed schema
→ vertically scalable
→ Distributed (data stored across machines)

Basically Available

→ Since data is distributed across servers it can handle outages
→ If one server crashes another replica/machine can give u the data

Soft States

→ States/values can auto change without txn or user queries
How? SYNC across replicas
→ since data is stored across machines/replicas it may happen that data copies keep getting updated
dirty data NOT IN SYNC → Latest in-sync data

Eventually Consistent

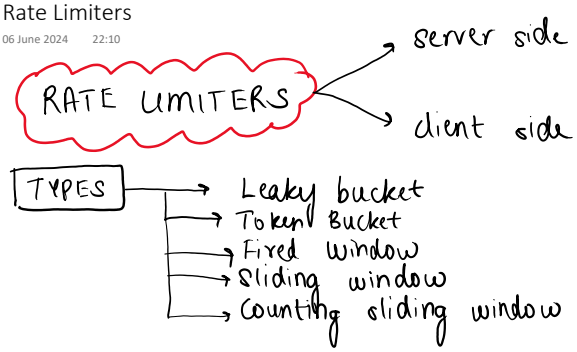
→ Not immediately consistent → ACID ones are immediate consistent

→ 2 simultaneous reads may give diffent Results sometimes
why? sync might have been happening b/w 2 reads

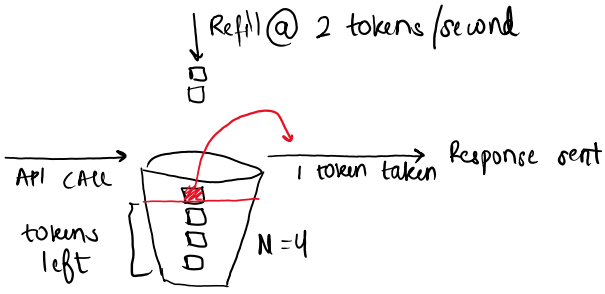
∴ TXNs are not always consistent due to distributed nature
BUT will become consistent aftr some time (once sync/updates are completed)

Rate Limiters

06 June 2024 22:10

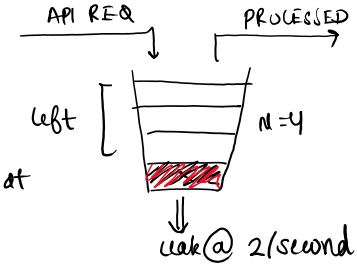


TOKEN BUCKET



- You have (N) tokens in a Bucket (consider this as a currency)
- Whenever a request comes
  1. Check if bucket has token
     - (No) → send (429) response code
     - (Yes) → Take one token from the bucket
- Refiller :→ Refill the tokens in the bucket at certain rate
  - refill_rate ≡ TPS/speed of your api
- * Configurations → (1) N = Bucket size, (2) Refill rate

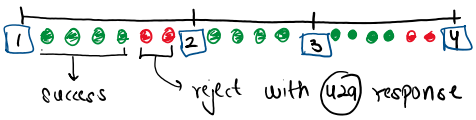
LEAKY BUCKET



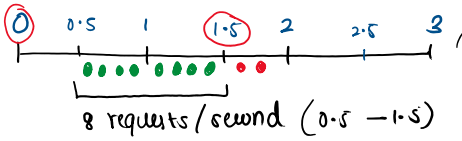
- You have a bucket of size (N)
- A request comes in, if bucket
  - (Full) → 429 code
  - (else) → Add 1 token to the bucket
- Leak rate → You keep on removing tokens at this given rate
  - Similar to refill rate in token bucket
- * IDEAL CASE → Rate of API Hits ≤ LEAK RATE

FIXED WINDOW

- You maintain certain windows (INTERVALS) and each window can only contain (N) items
- Ex:- Window of 1 min each → N = 4



- CON → Doesn't always guarantee rate limit
  - Ex:- N = 4/sec



→ valid since our windows are 0 – 1 – 2 – 3
But truly not limiting at Boundaries

SLIDING LOG WINDOW

→ Instead of fixed window you use timestamps to dynamically calculate it
→ Rate = 4 req/sec

Ex
→ Request comes at 1.5, you count no of req blw 0.5 ⟷ 1.5
if num < 4
→ Insert this log (timestampt reqid)

* Ensures proper dynamic rate limiting

CONS
→ you store each request as one entry
∴ Space needed = N
→ lets say your Rate is 1000 rps ⟹ you will store 1000 log entries

SOLUTION

SLIDING COUNTER WINDOW

→ Instead of pushing each request as log item maintain a counter
→ Rate = 1000 rps

ts	counter
0.1	1
0.2	100
0.3	200
⋮	⋮
0.9	400

→ Here instead of 1000 logs you will only have 10 logs
→ space = 1000 → 10
→ No of ts steps

Microservice Architecture

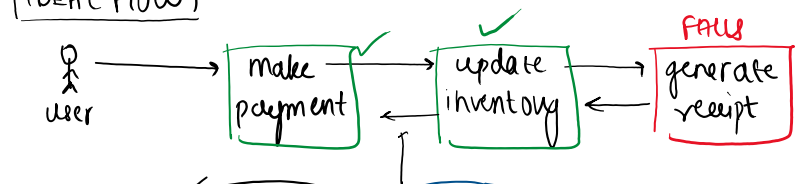
06 June 2024 22:54

DISTRIBUTED TRANSACTIONS

PREMISE → You have 3 services

- PAYMENT → make payment
- INVENTORY → decrease purchased items
- RECEIPTS → generate receipt

IDEAL FLOW



How will you handle this in microservice

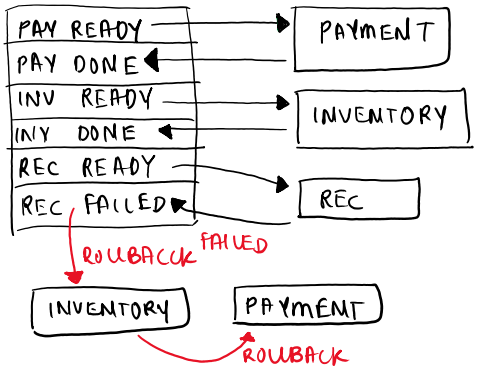
→ ① 2PC [2/3 phase commits] BLOCKING
② 3PC
③ SAGA (ASYNC)

SAGA → Sequence of Local transactions

* Asynchronous and non-blocking but complex to implement

→ Queues are used for Inter Process Communication

STEPS



Implementation

→ 3 microservices

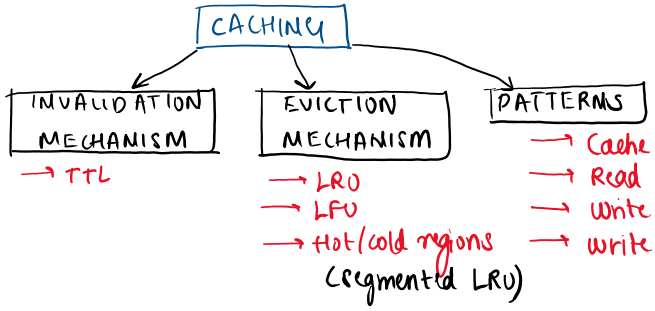
→ (3*2) Queues { 1 success, 1 failure } one b/w each 2 service

→ 2 Queues

① Handles all success (all 3 service listen to this
② Handles all failures OR listen to a specified partition)

Caching

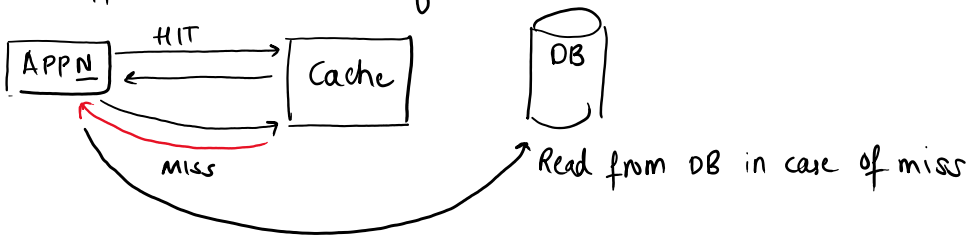
07 June 2024 23:04



PATTERNS

CACHE ASIDE

→ Cache cant communicate directly with DB
Application does that for it



→ Initially data is loaded into the cache by app?
On every cache miss, the app directly fetches data from DB
→ On new data, it will always be MISS
↳ until you update cache

PROS

* Good for HEAVY-READ apps like dashboards
* If cache fails → you can directly hit the DB each time
down
* Since data stored in cache and DB is different, you can have diferent schemas

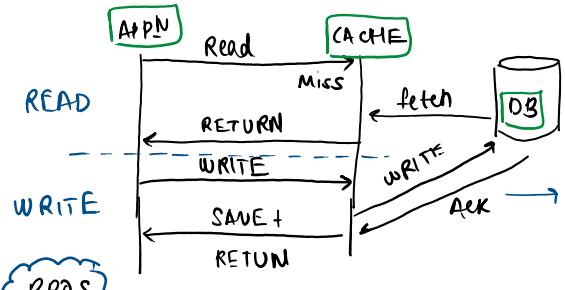
CONS

* New data will be Miss
* Inconsistency in data if it gets updated in DB

READ THROUGH

→ Cache can communicate directly with DB App only accesses Cache
→ READ MISS : Cache itself updates value from DB and then returns
↳ New val, update

→ Writing new data :- you will write to cache + cache will write to DB
→ if this was update invalidate cache which will be updated on next miss



* DB schema ≡ Cache schema

WRITE are SYNCHRONOUS

PROS

→ Great for heavy reads why? (On write cache writes to DB, waits for Ack saves it in cache and then sends it back)
→ You dont [illegible]

PROS
→ Great for heavy reads · why? (On write cache writes to DB, waits for Ack saves it in cache and then sends it back)
→ You dont worry abt Miss and fetch from DB

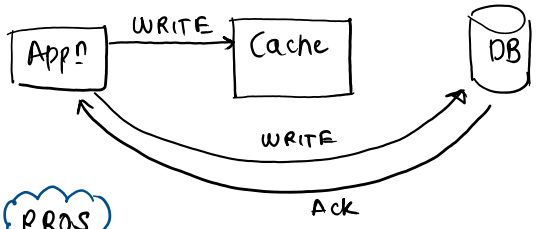
CONS
→ Cache cant fail
→ Cache miss for new data (PRE-HEATING can solve)
→ Cache schema ≠ DB schema
→ Data inconsistency

WRITE THROUGH → used with READ THROUGH (100% CONSISTENT)
→ Here for writes you directly write to DB and invalidate cache (if updated)
→ Next hit will be cache miss and cache will fetch from DB
(saves latency of app ⟷ cache ⟷ DB for write operations)

WRITE AROUND → You ONLY WRITE TO DB and forget
* when you do a get next time: CACHE miss and cache will fetch from DB

WRITE BACK → SYNCHRONOUS — same as write through (BUT cache updates DB → Not app itself)
→ ASYNCHRONOUS — NOT 100% consistent

SYNCHRONOUS
→ App itself writes to both DB and cache whenever a write happens

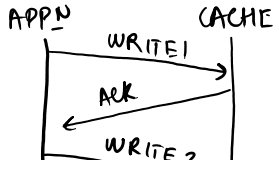


① write to DB
② wait for Ack → use 2PC
③ write to DB

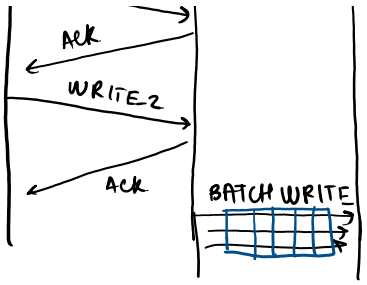
PROS
→ Very high hits and low misses
→ Very rare chance of inconsistency
} you always write to (DB + Cache) each time No lag

CONS
→ High latency due to 2 writes and Ack wait
→ Need 2PC to make sure DB update is ACID

ASYNCHRONOUS
→ App only writes to cache and then forgets
→ Cache then asynchronously writes data into the Database independently
→ done via QUEUES
→ you can publish 1 event for each write OR Batch (n) writes and at once write to DB



PROS
→ WRITE HEAVY applications
→ Lesser write latency and lower inconsistency
→ Better performance



→ lesser write latency and lower inconsistency
→ better performance

CONS
→ sync issues if you immediately read again before DB write is done

Best with READ THROUGH CACHE
why? → when write happens u just write to cache and forget
→ Now you can directly read that data from cache (even though its still not in DB)

GLOBALLY DISTRIBUTED CACHE

① There can be (N) different cache servers distributed across clusters
↳ A cache gateway redirects requests to appropriate servers
ⓐ Default uses consistent hashing for allocation
ⓑ Geographically based - route/store in nearest store
ⓒ Identifier based – Ex: Based on country_id group and store in same server
or multiple countries INTO 1 continent server

REDUNDANCY

* use QUORUM u need consensus for both READ + WRITES by setting values of (R), (W)
→ you need values read from atleast (R) copies
→ you need to write data to atleast (W) copies/machines

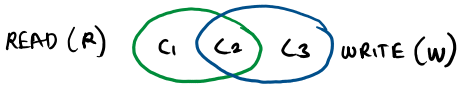
(R) → Read consensus frm (R) nodes
(W) → write consensus frm (W) nodes
(N) → Replication factor

R + W ≤ N
EVENTUALY CONSISTENT

R + W > N
STRONGLY CONSISTENT

WHY?
→ R + W ≤ N : assume N = 3
* R = 1, W = 2
WRITE : u write to 2 copies and leave third
READ : needs only 1 copy
↳ if it reads from 3rd then inconsistent
* R = 2 : W = 1
→ WRITE : to any 1 assume (C1)
→ READ : (C1)(C2) valid
(C2)(C3) INCONSISTENT
(C3)(C1) valid

R + W > N
Ensures that there is atleast 1 overlap blw write and read consensus



Isolation Levels

08 June 2024 14:54

LINEARIZABLE

→ single thread sequential execution of Queries
→ No chance of any mismatches
→ PERFECT consistency

EVENTUAL

→ DB might not be always consistent BUT eventually consistent
→ multiple Queries/read/writes can happen of multiple threads/servers
  ↳ parallel process so is faster
* But sync takes some time after transactions

CASUAL

→ Allows parallel transactions BUT operations are linear/serialize in single transaction
→ Consistent within transaction
→ Better than casual but not as good as linear

CONS: Aggregates/joins are inconsistent

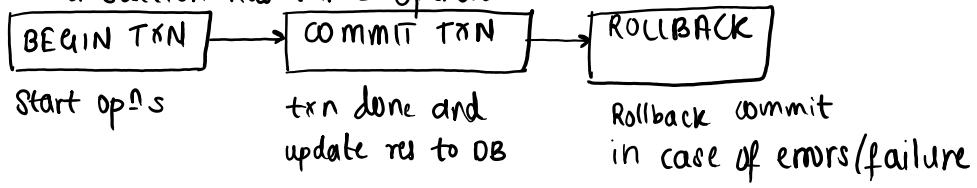
QUORUM

→ Configurable consistency vs performance

R + W > N	Highly consistent
R + W ≤ N	Eventually consistent

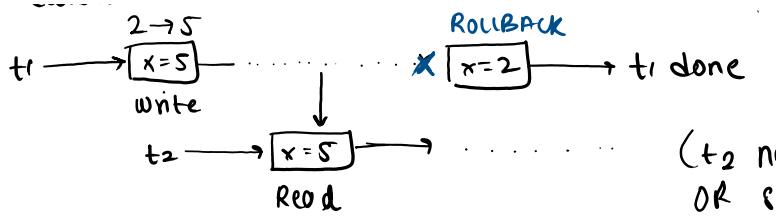
TRANSACTION ISOLATION LEVELS

A transaction has three operations



READ UNCOMMITED

→ Each txn directly writes to DB ≡ No concept of COMMITS (No data replication)
→ Causes DIRTY READS



(t_2 never gets rolled back value of x)
OR some other op? might also have updated it

READ COMMITTED

→ Txn can only read values which are committed by other txns
→ slower since you have commit step ↳ it wont roll back later
and dont directly update DB

* in both read commited/uncomited you cant ensure REPEATABLE READS
→ if you have 2 reads of same var in the txn
you can get 2 different values (someone else must have updated it)

REPEATABLE READS = SNAPSHOT ISOLATION

→ when any row is read by a transaction
its LOCKED until that transaction ends
→ other txns which update this locked value can locally commit and keep it as new version and update in DB once lock is released

SERIALIZABLE

→ All operations in txn are executed sequentially in order
→ if 2 txns dont have any common ops then they can run in parallel
↓
Both writes and reads
→ use serialized Locks

least isolation
most efficient

Least efficiency
most isolation

READ UNCOMMITTED	READ COMMITTED	REPEATABLE READS	SERIALIZABLE

← EFFICIENCY →

→ ISOLATION →

CRDT

19 December 2024 19:34

CONFLICT-FREE REPLICATED DATA TYPE

WHY CRDTS

→ You are in a multi-writer DB duplication setup
→ Now at same time the same key may get updated → WHICH VALUE WILL YOU PICK?

1. Last write wins (very bad)
2. Store all writes and decide which to pick later
3. Let DB take that call = CRDT

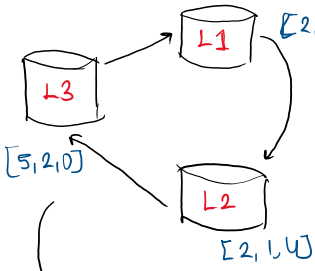
INFO ABOUT CRDTS

→ The conflict merges will happen eventually (NOT immediately)
→ which DB uses CRDTS? → Riak
→ Redis (sets in Redis enterprise)

OPERATIONAL CRDTS

→ You design a distributed counter on a multi-leader/writer setup → INCR(idx) = increments value of `idx` by 1
→ but this can be simultaneously done on separate leaders at a time

VERSION VECTORS



[2,1,0] → version vector → we store one such vector for each index

[2,1,0] → received 2 increments from leader 1, 1 incr from leader 2

value(idx) = sum(version vector of idx)

→ Here instead of passing list of size(N_leaders) we just call incr(si) on the other leaders
→ Ex: – Instead of [5,0,3] being saved you would have just incr(5i)

L3 [5,2,0] → incr(L1,+1) × 5, incr(L2,+1) × 2 → This take O(1) instead of O(n-leaders).

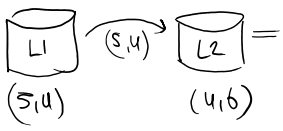
OPERATIONAL CRDT

→ SOME CONS
1. You need to ensure no drops and no duplicates (when incr for example)
2. Need to be casually consistent (same order followed)

(ex: if u make a set using op^n CRDT → remove(idx) can be done only a add(idx) has happened)

STATE BASED CRDTS

→ Instead of sending just the increments, SEND ENTIRE CRDT



= L2 → merge([5,4],[4,6]) = [5,6] → take max of each server

The merge needs to be → 1. COMMUTATIVE → M(a,b) = M(b,a)

The merge needs to be → (1) COMMUTATIVE → M(a,b) = M(b,a)
(2) ASSOCIATIVE → M(a, M(b,c)) = M(M(a,b), c)
(3) IDEMPOTENT → M(a,b) = M(M(a,b), b)
↳ this handles in case of duplicate msgs received

HOW DO U COMMUNICATE AMONG WRITE-LEADERS
↳ using state CRDT
(1) Dont need order (ASOC+COMM) ⟹ GOSSIP PROTOCOL
(2) duplicates r fine (IDEMP)
→ No extra/special infra needed

TYPES OF CRDTS

NAME	operational CRDT	State-based CRDT	final counts
Counter	incr(leader)	incrs: [0,2,1]	↓
Decreasable Counter	incr(leader), decr(L)	incs: [0,2,1] decs: [0,1,0] =	[0,1,1]
Sets	add(x), remove(x)		

↳ hard to implement

Which Databases?

22 December 2024 00:11

DB INDEXES

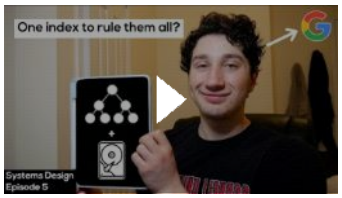
B+ Trees

→ Stored on disk
- → slower writes
- → faster reads
- → NO size issues

LSM TREE + SS Tables

→ LSM : memory
SS-Tables : disk (only when needed)
- → slower reads (read across SS tables)
- → faster writes (write to memory)
- → delete = writing a tombstone

B+ Trees: How do B-Tree Indexes work? | Systems Design Interview: 0 to 1 with Google Software Engineer
LSM Trees + SS Tables: LSM Tree + SSTable Database Indexes | Systems Design Interview: 0 to 1 with Google Software Engineer



REPLICATION

① Single leader → 1 Node handles writes / can read from many replicas

② Multi leader → You can write to multiple nodes + Read multiple
- → chances of write-merge conflicts

③ Leaderless → you write/read from any node/replica
- → uses multicast / gossip protocol among nodes

MY-SQL

→ Relational / Normalized data (uses 2Phase-commits for concurrency)
- → very slow but surety

CACID + transactional guarantee)

→ Uses B+ Trees + Single leader replication

Mongo DB

→ Document based NOSQL DB (data written as large documents)
→ Btrees + transaction support

NOTE Nothing very special but acts like SQLish NO SQL DB
- → transactional support

APACHE CASSANDRA

→ Wide column DB : has shard key + sort key
- ○ Flexible schemas
- ○ Easy to partition

→ Multi leader / Leaderless replication (configurable using Q worms)
- ○ Super fast writes but uses Last-write-wins for write conflicts

→ Index : LSM Tree + SS Tables
- ○ Fast writes

USE case Apps with high write volume/speed. Consistency is not so important

RIAK

→ key-value based NOSQL DB
→ Multi-leader / leaderless BOT supports CRDTs (Manual code for merge conflict)
→ SStables + LSM trees

USES : High write throughput + controllable write conflicts

HBASE

→ Wide column DB — shard key + sort key
→ Single leader replication
- → Built on top of HDFS : durable and reliable
- → slower for frequent ops

→ LSM Trees + SS tables
→ Colum oriented storage
- → column compression + data locality in column

USES : Large data + fast column reads

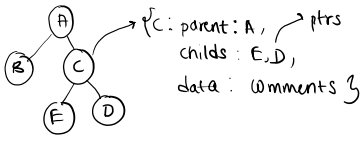
→ Colum oriented storage
→ column compression + data locality in column

Uses :- Large data + fast column reads

GRAPH DATABASES

NATIVE GRAPH DB (NOSQL)

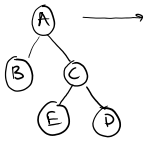
→ You store each node and memory ptrs to its adjacent nodes within the representations



+ Faster lookups since addresses are stored
− Inserts/updates are slower
  ① You need to traverse to insert location
  ② Create node and find nbrs/par/childs
  ③ Create links and store

NON-NATIVE GRAPH DB

→ You store data in some tabular format and later perform joins/ops to get relations



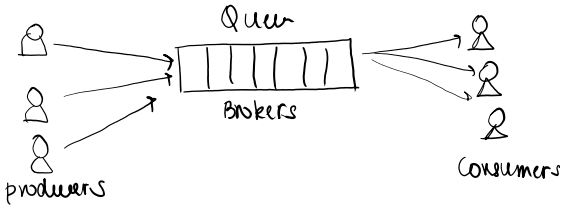
id	par-id	comment
A	–	good post
B	A	agree, +1
C	A	ur joking
E	C	+1
D	C	ur right

+ Insertions are easy → just adding rows
− Queries/getting relations are tough/slower
  → since you dont have addresses you need self joins (in this case)

Streaming

22 December 2024 00:40

- Generally works on concepts of PRODUCERS + CONSUMER



MESSAGE BROKERS

IN-MEMORY (RabbitMQ)

- → stateless + in memory
- → PUSH model
  - → sends msgs to consumers + waits for ACK
- → At-least once semantics
- → Very low latency

LOG-BASED (kafka)

- → stateful + events written to a log
- → POLL model
  - → Consumers read at their own pace
    - → uses offset-based model
- → at least once, at most once + exactly once semantics
- → High throughput and durable

STREAM PROCESSING

- joining streams and processing
- fault tolerance of streams

→ Spark streaming
- → micro batch processing

Flink
- → realtime on event at a time

CHAINED AFTER MSG BROKERS
acts like consumers

APACHE FLINK

- → Gaurantees that each msg affects state only once
- → Flink checkpoints your state and dumps it to storage (S3)
  - you can replay the events from your log-based queue
  - Done by a job manager / zookeeper

Architecture

04 April 2024 18:27

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, $36^8 \approx$ 2Trillion (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
- → 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
  2Trillion 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 Tables 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

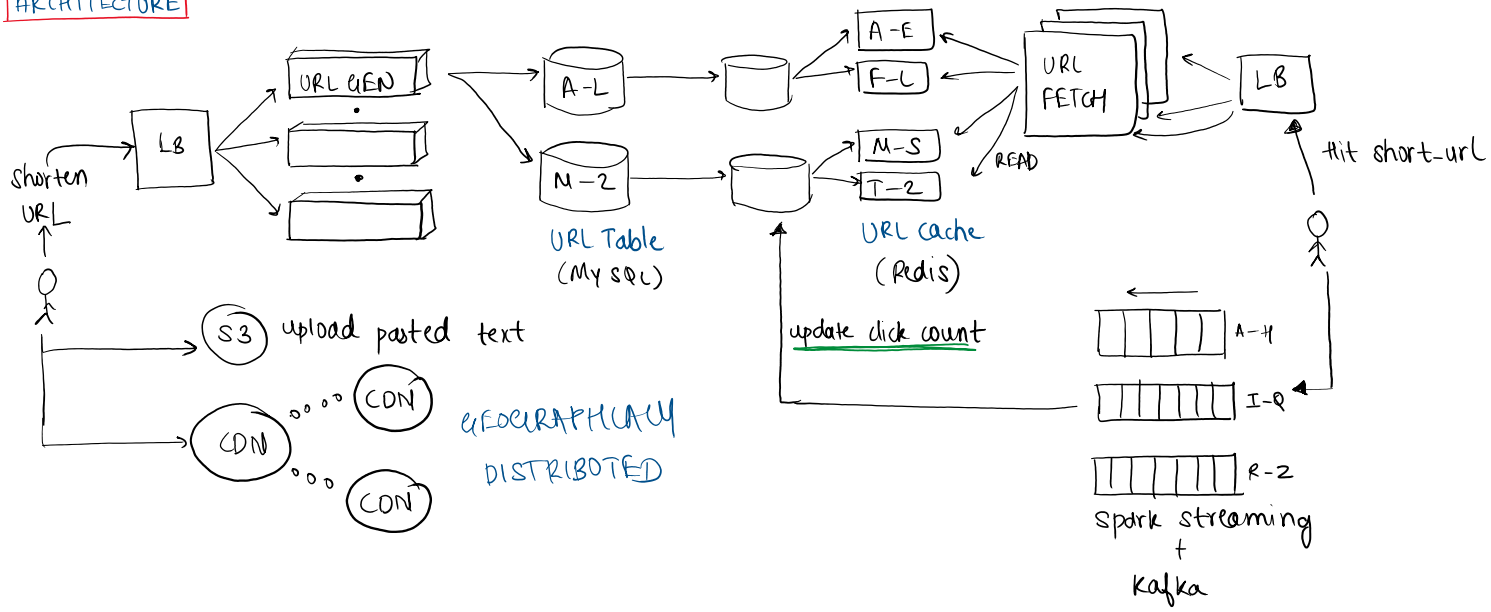
PASTE BIN - HUGE PASTES

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 acounts 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)

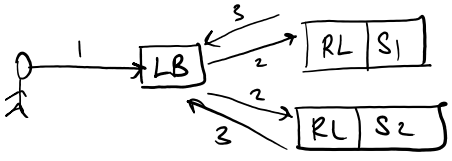
WHERE TO PLACE RL

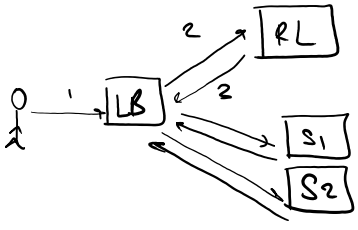
Local to service

+ 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

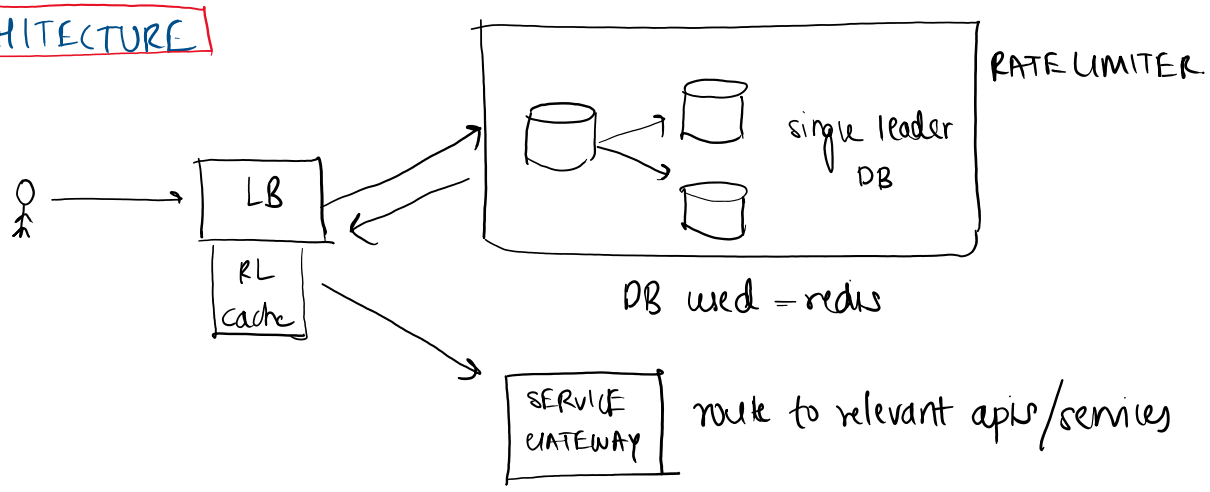
LEADERLESS / multi-leader

+ 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



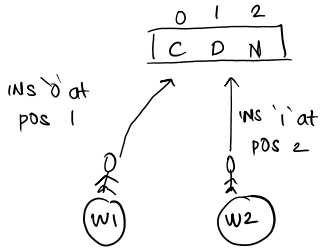
REQUIREMENTS
→ Users must be able to edit documents simultaneously
→ The updates need to be propagated to all users eventually

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

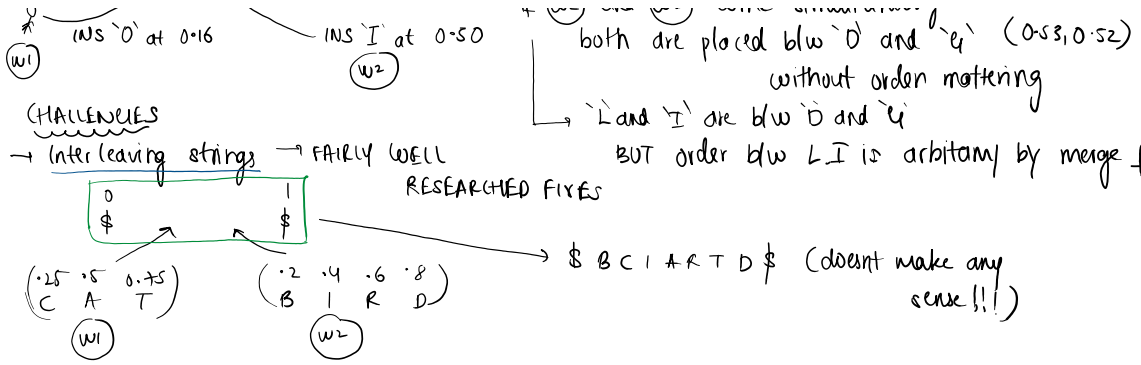
CRDT MERGE FN → Make it order independent
① First char = 0
Last char = 1
② if you want to insert a char then pos = 0+1/2 = 0.5 + some random jitter

Ex:

C	D	N	E
0	.33	.66	1

W1: INS 'O' at 0.16
W2: INS 'I' at 0.50

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 'E' (0.53, 0.52) without order mattering



ACHIEVING IDEMPOTENCE

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

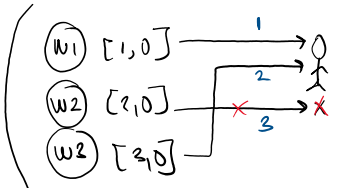
VERSION VECTORS

→ 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

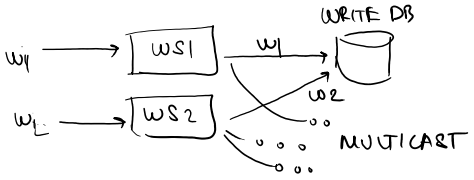
VERSION SKIP ISSUE



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



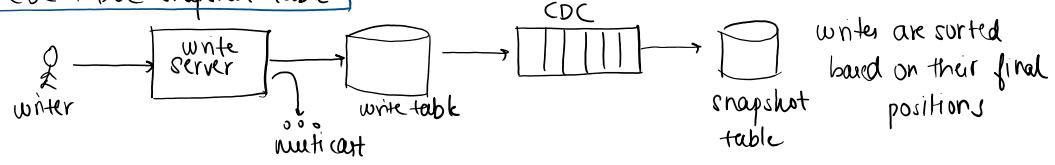
→ In case you get [3,0] first, you can query DB and check if (V-1) write exists in the DB?
→ if yes process (V-1) write and the current one next

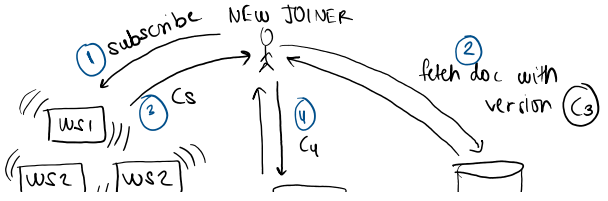
NEW JOINERS → also applies to stale users becoming active

↳ Basic: fetch the base document & apply writes

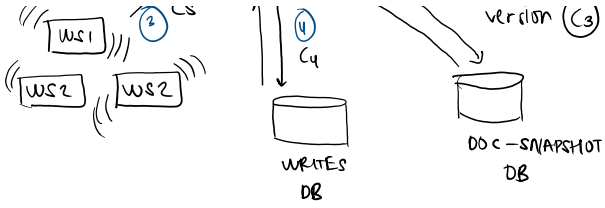
✓ BETTER

CDC + Doc snapshot table



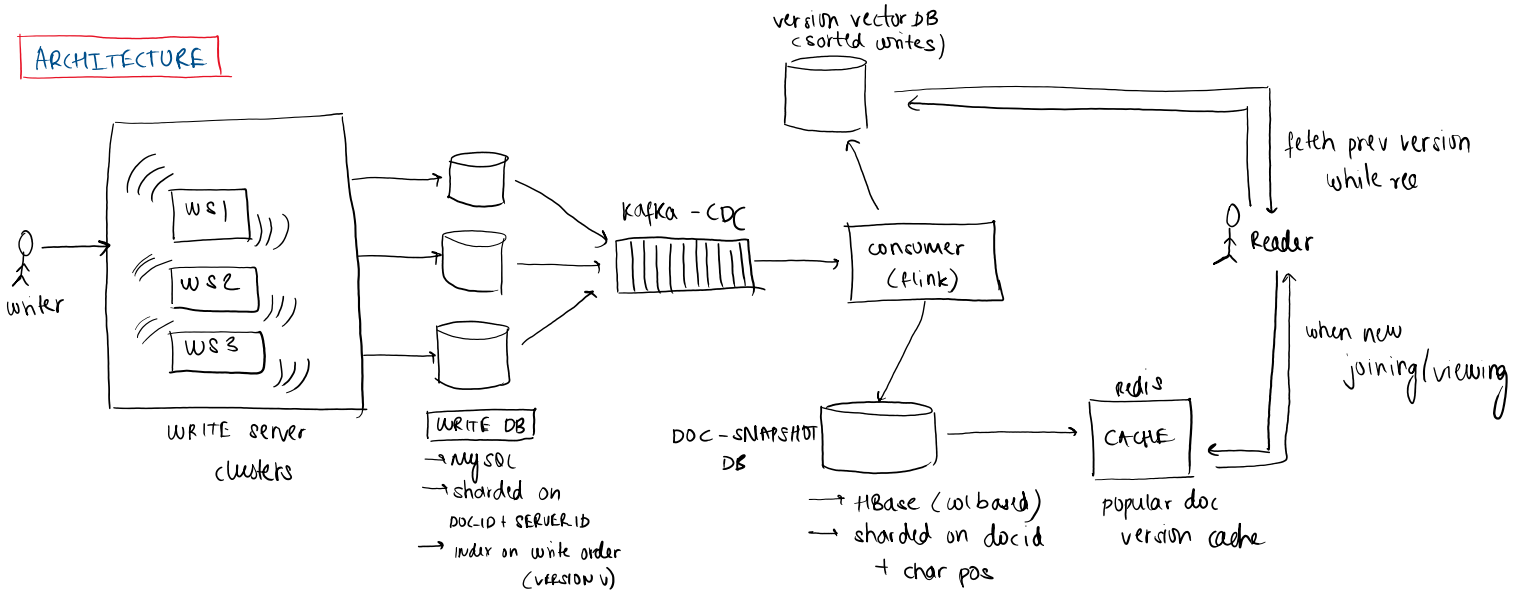


→ why (4)?
→ you get document till (C3) from snapshot DB
→ then you get write with VV → (C5)
but you check if (VV → C4) is there in write DB



→ 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
$(100 \times 10^6) \times (10^6) \times 400 = 4 \times 10^{12} \times 10^4 = 4 \cdot 10^{16}$
$= 40 \times 10^5 =$ 40PB

VIDEO STREAMING

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

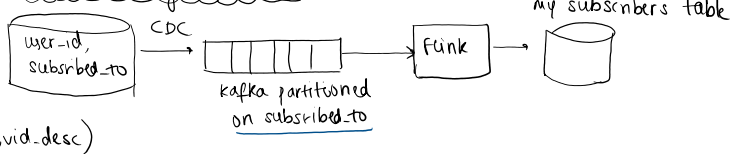
→ CHUNK + MULTIPLE RESOLUTIONS FOR EACH CHUNK

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 curr 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 curr 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 scarce
4) NO state being stored here

which object store?

HDFS

+ Data locality (huge cluster)
+ high processing power

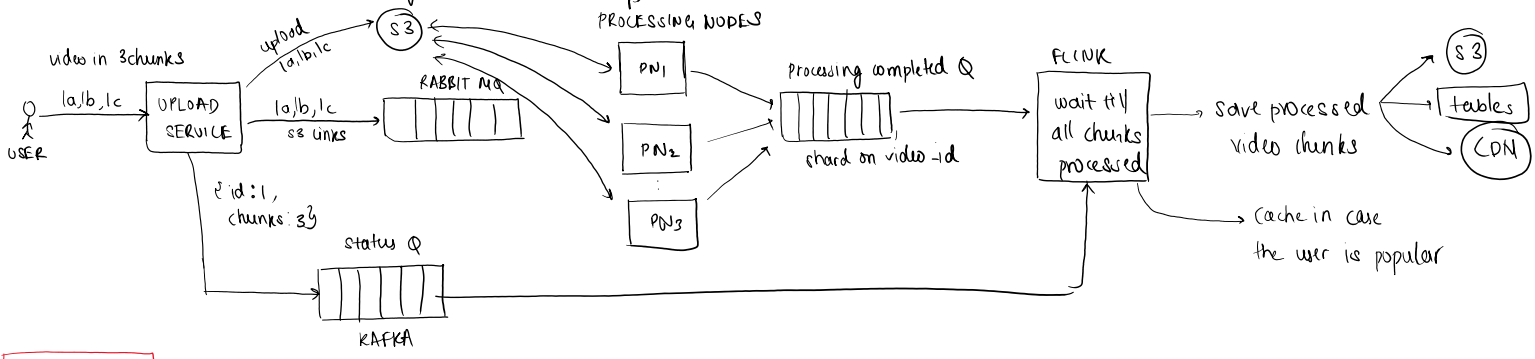
− Costly due to CPU needed

OUR CHOICE → **S3**

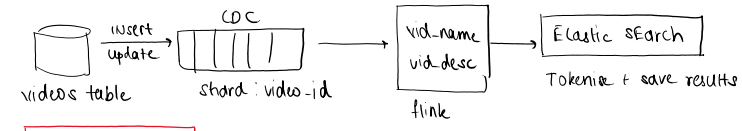
+ Cheaper storage costs
+ No cpu costs → just upload n forget

− n/w calls vary (each chunk needs a call and geographically locate might change)

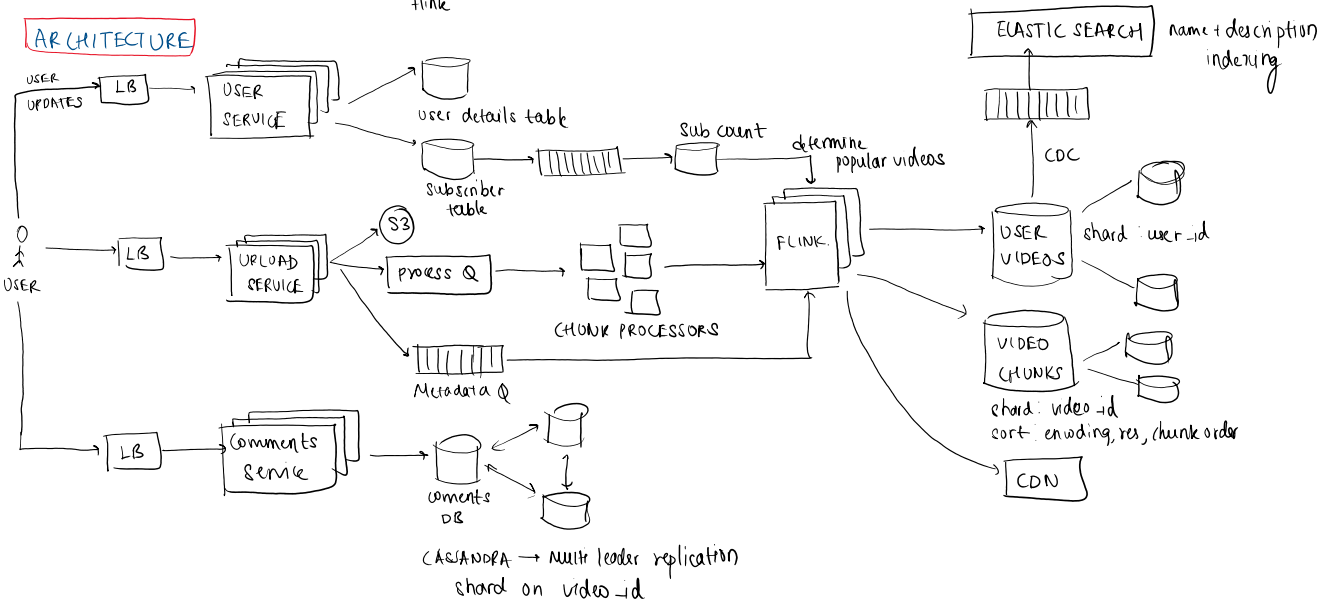
✱ we pick (S3)
→ cheaper in storage costs
→ we are fine with n/w call delays (CPU is more costly)



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

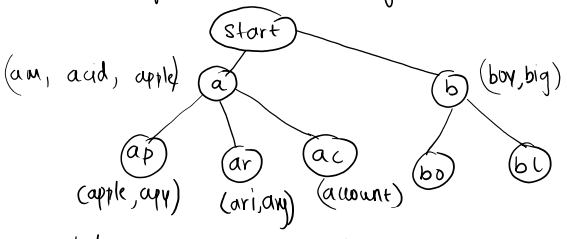
→ can be stored locally on client device

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)
P = len of prefix
k = len of max word with prefix P

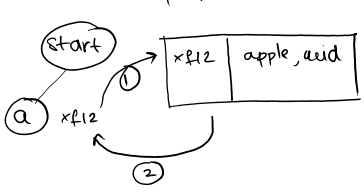
→ while typing TC for getting next char suggestions is O(1)
"a" – am, acid, apple
"ap" = apple, apply
→ just one node traversal below (a) → (ap) so O(1)

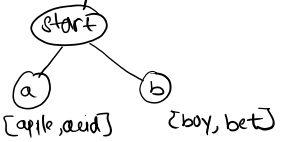
MEMORY EFFICIENT

→ 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



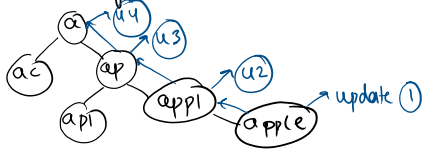


UPDATING KEYS

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

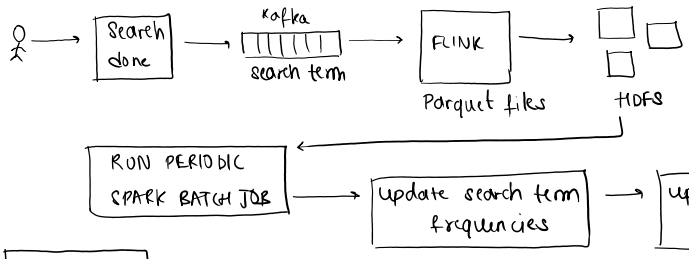
apple: "apple buy, apple big → apple pie, apple phone"

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

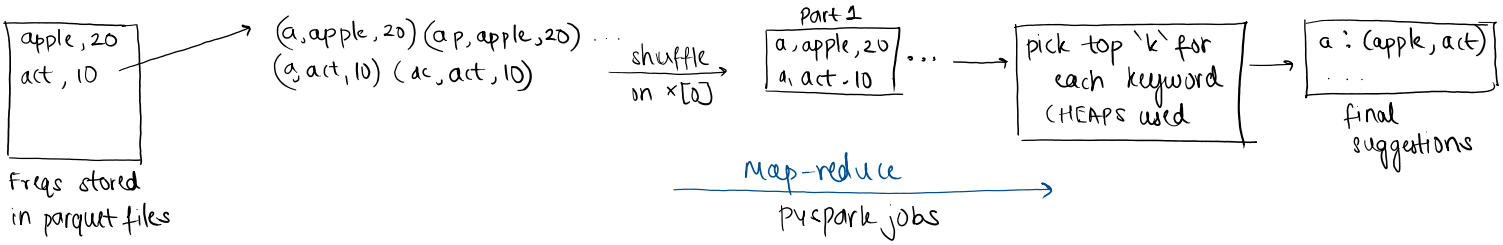


→ You might need to acquire locks for this. (In case u update 2 words with common prefixes)

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)

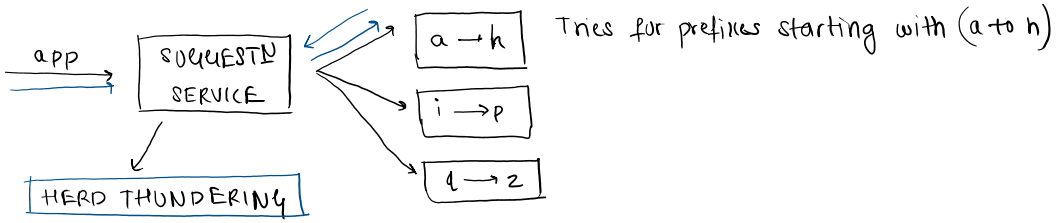
(3) SSE (server sent events) → NO (you need bidirectional flow)

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

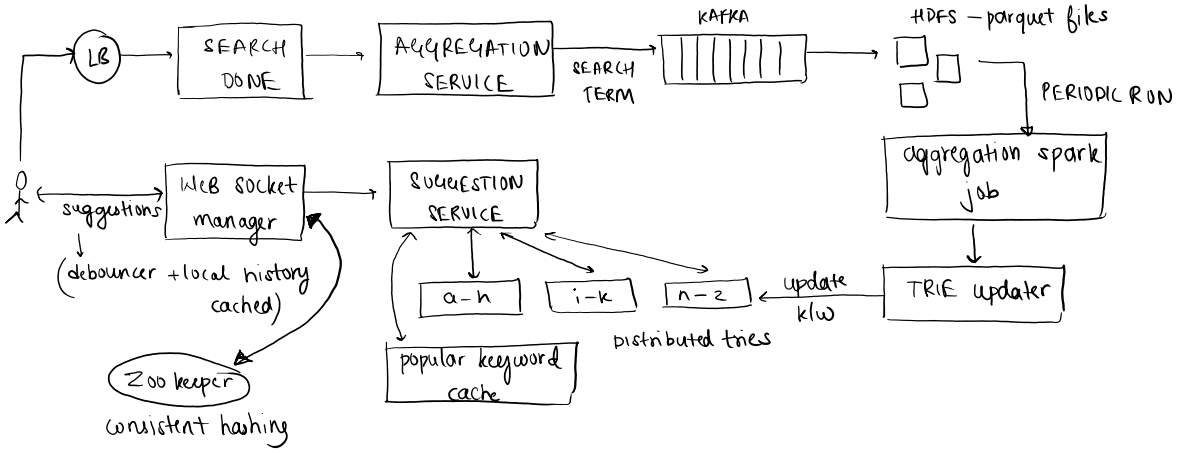
what if you get lot of requests at a time for certain keywords

→ (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 bkg)

(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

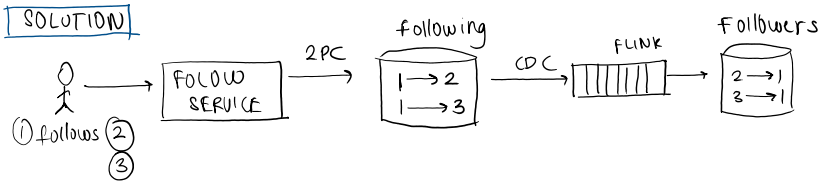
FOLLOWER TABLE

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

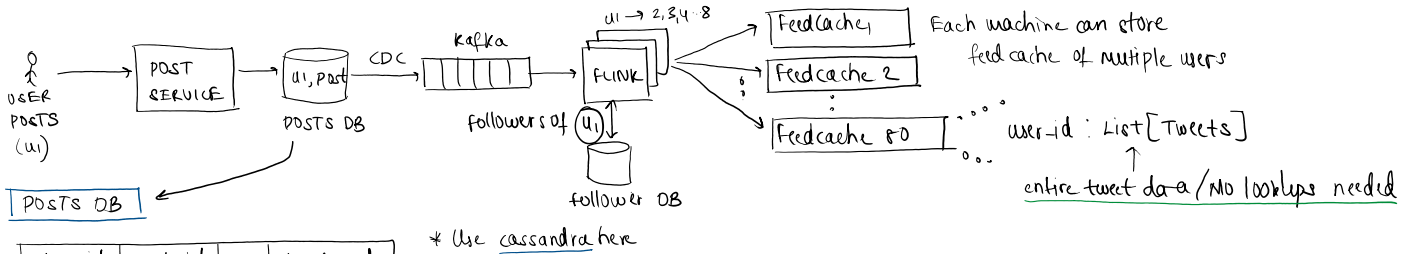
→ Partition key : user_id , sort key : follower/following_id

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)



* Use cassandra here

follower DB

entire tweet data / NO lookups needed

POSTS DB

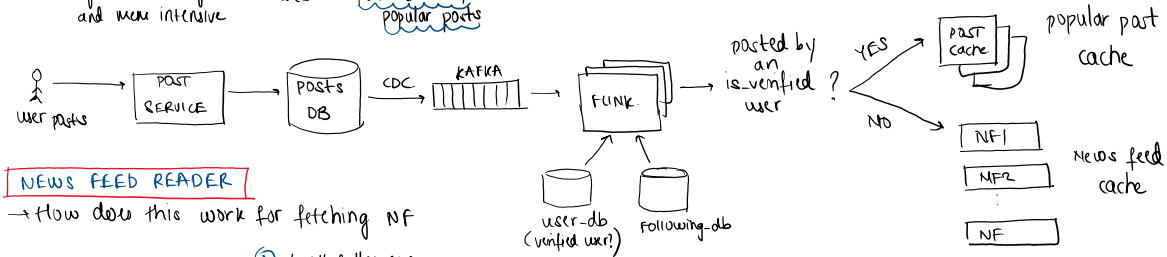
user_id	post_id	ts	tweet_body

shard — sort

* Use cassandra here
  - → High write throughput
  - → No write conflicts (Multiple users cant make same post, same users create post only once, update posts also happens at once)

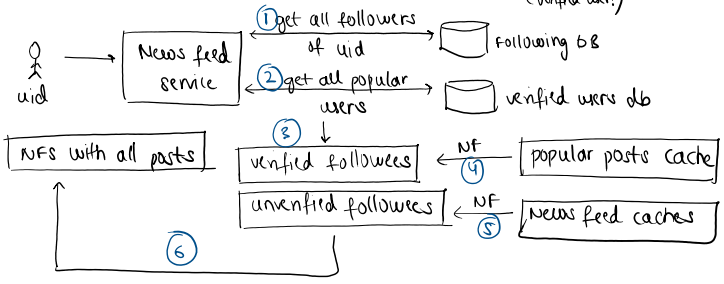
POPULAR PEOPLE POSTS

→ 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 → SOLN → post cache for popular posts



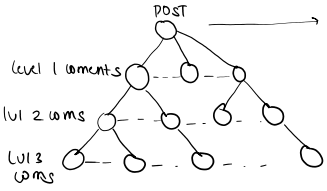
NEWS FEED READER

→ How does this work for fetching NF



COMMENTS

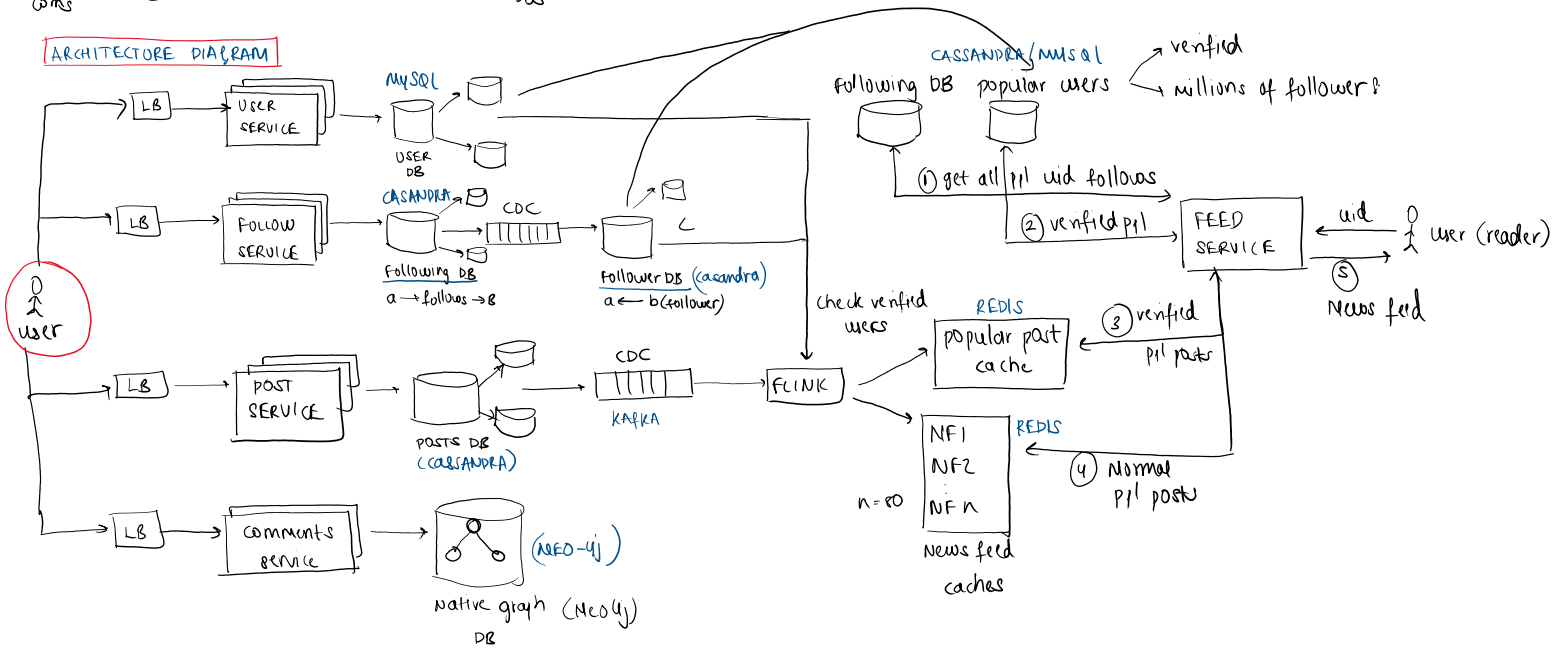
① Only one level → CASSANDRA (high write throughput / NO conflicts)
② Nested comments → GRAPH DB needed



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

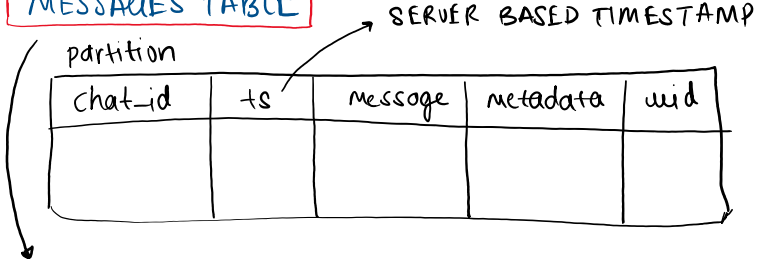
→ multiple writes may happen : single leader replication (MySQL)
→ can think of SET CRDT (too complex for this)

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



which DB to choose?

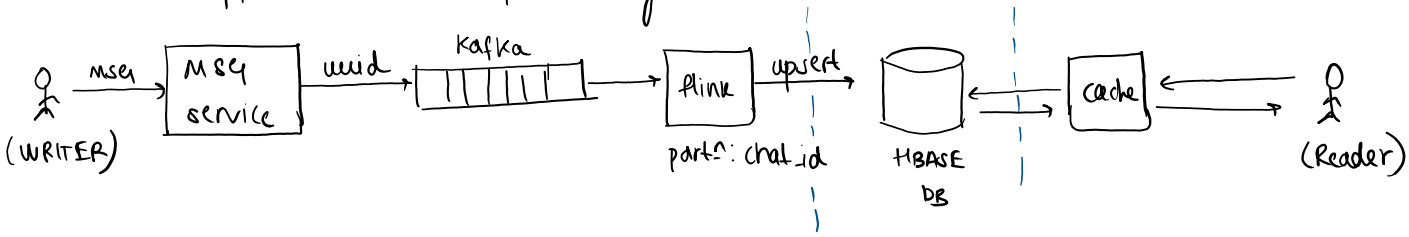
→ Reads and writes are both imp + frequent
→ Column oriented storage? → usually you read only part of colums
(metadata is usually not read)

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

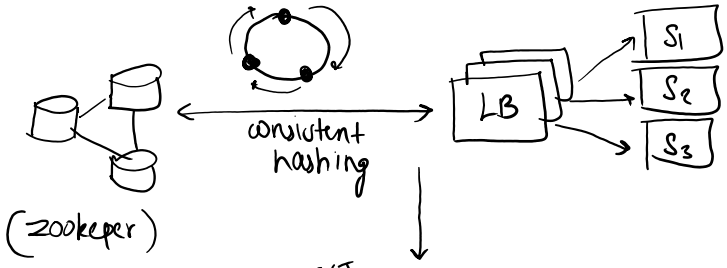
Tuning for faster writes

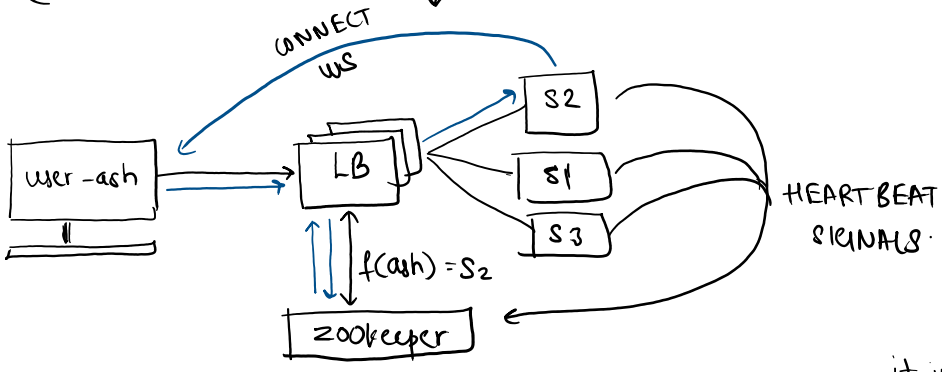
① Read directly from HBase
② Writes happen via stream processing (uuid + msg handles idempotence)



LOAD BALANCING

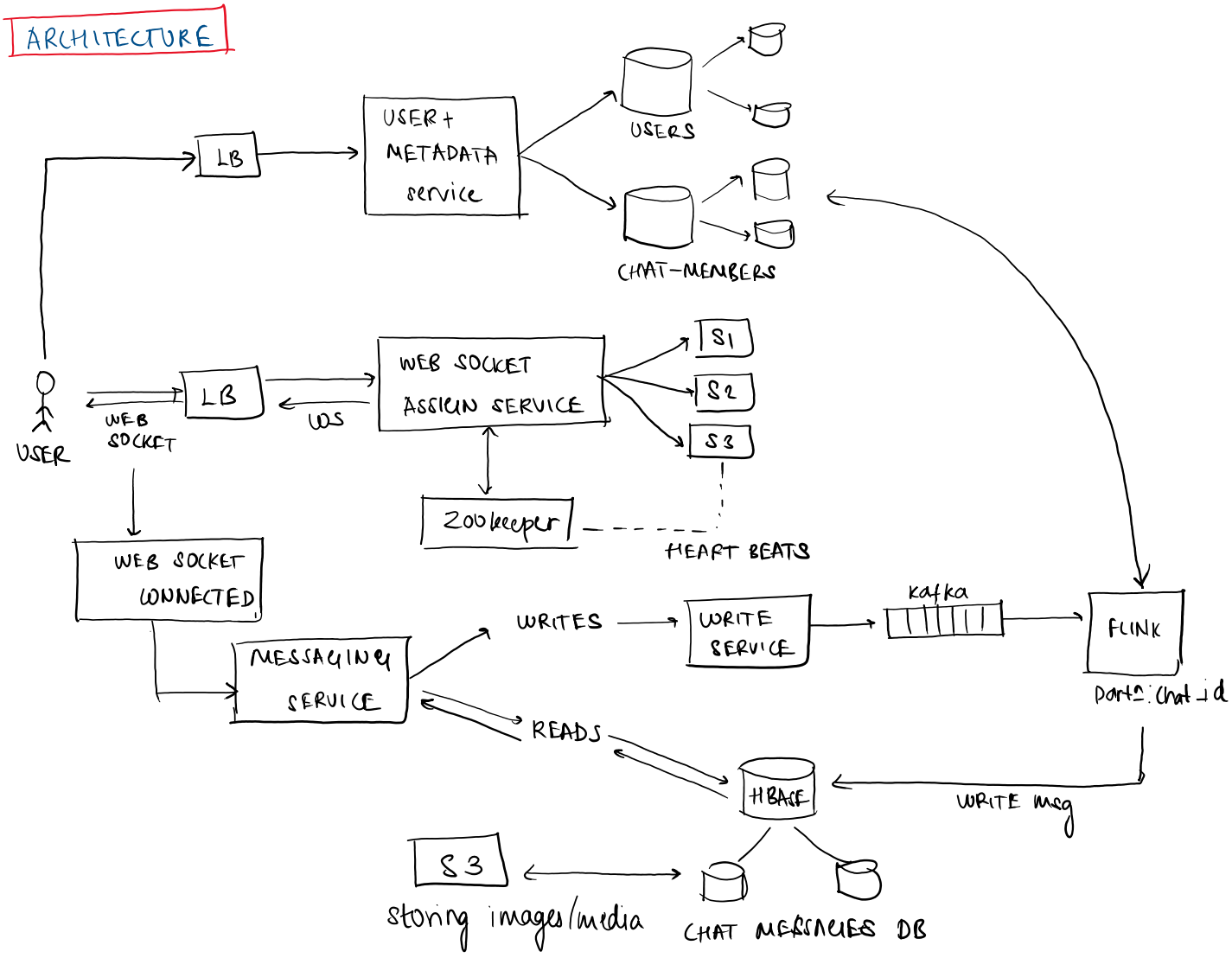
→ WEBSOCKETS (Allow bidirectional data transfer)
→ Load Balance on user_id : requests from user must go to same websocket
  ↳ will use zookeeper to keep track of active servers
  ↳ internally use Consistent hashing for dynamic mgmt
  ↳ LB will listen to zookeper





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)

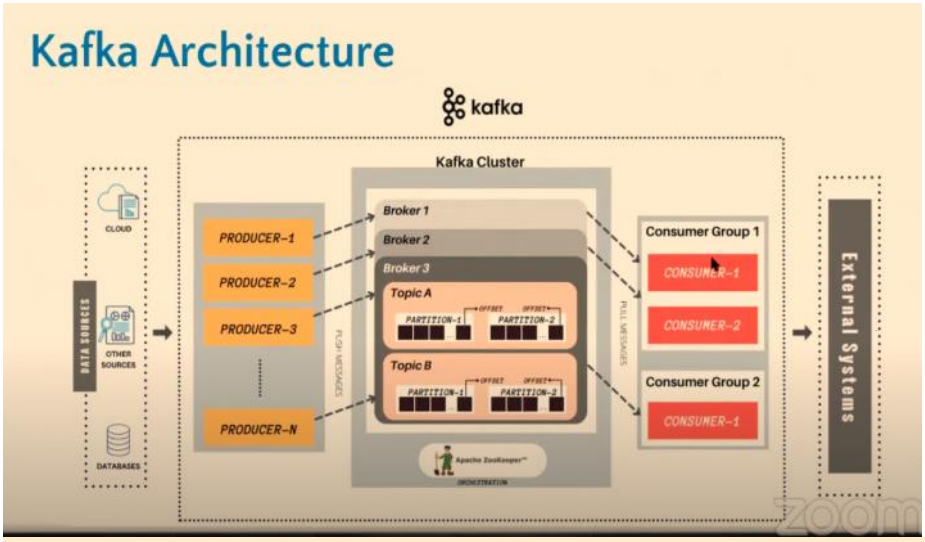
IN CASE (S2) goes down
  → LB asks zookeper again (after some interval)
  → zookeper knows (S2) is dead (didnt get heartbeat)
  → It assigns another server to user (ash) using consistent hashing

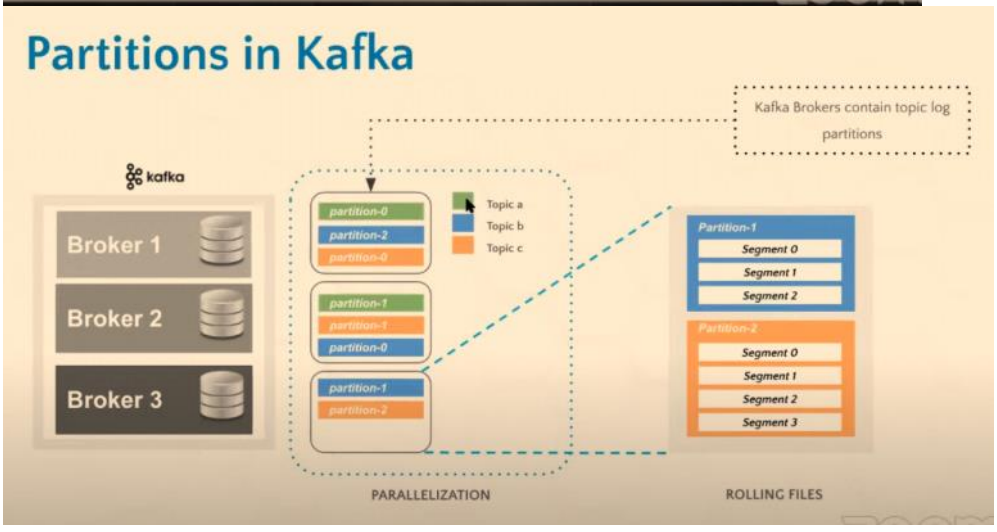

ARCHITECTURE
LB
USER + METADATA service
USERS
CHAT-MEMBERS
USER
WEB SOCKET
LB
WS
WEB SOCKET ASSIGN SERVICE
S1
S2
S3
Zookeeper
HEART BEATS
WEB SOCKET CONNECTED
MESSAGING SERVICE
WRITES
WRITE SERVICE
kafka
FLINK
Partn: chat_id
READS
HBASE
WRITE msg
S3
storing images/media
CHAT MESSAGES DB

Kafka Architecture

08 April 2021 14:03

KAFKA ARCHITECTURE BASED





PRODUCER DETAILS

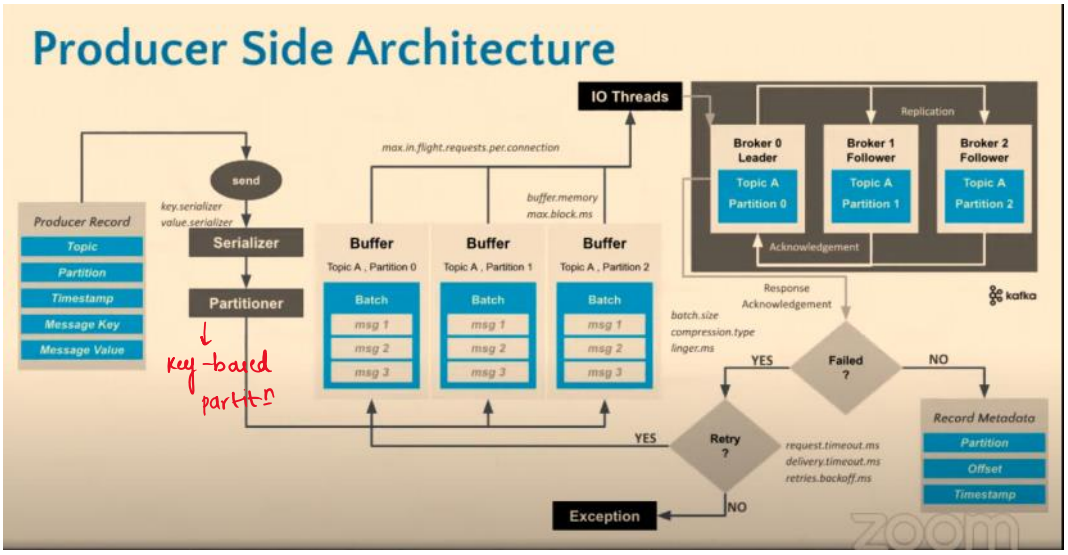
Every msg you push you need the following detai

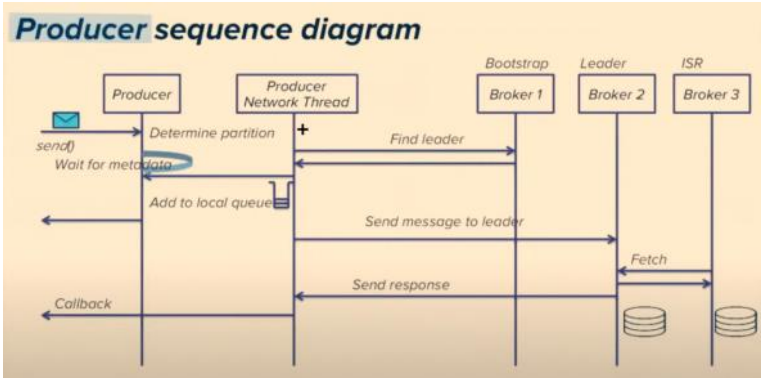
ls:

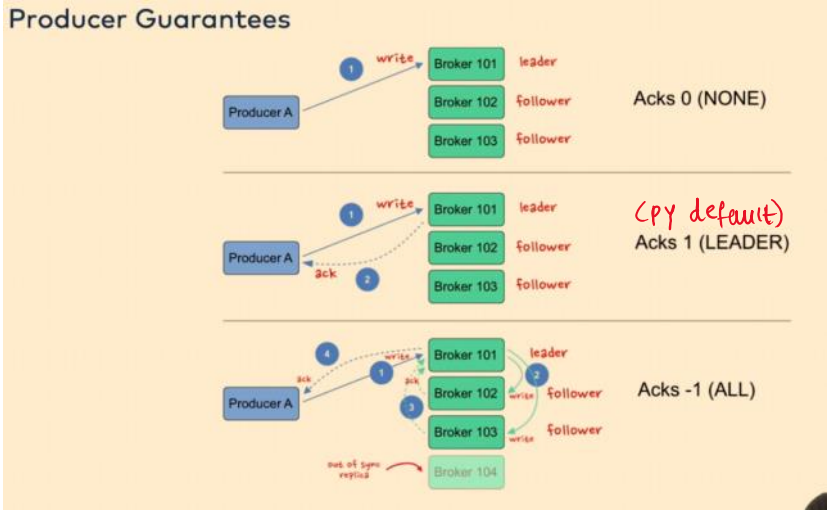
a) key (optional)
b) value
c) topic
d) group_id
e) parition_num (optional)

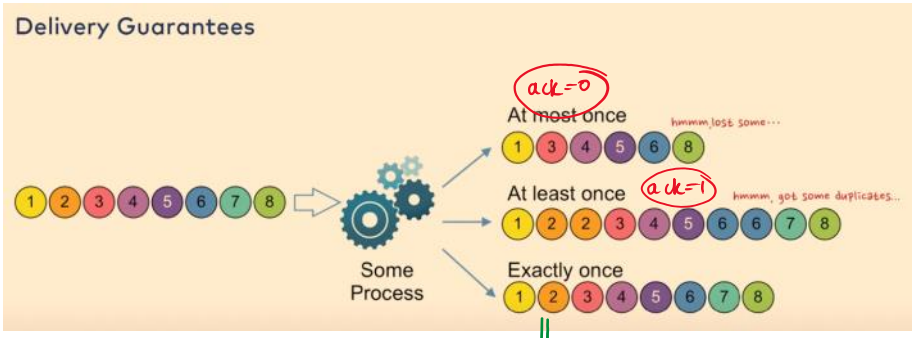
How to decide which partition to write to?

1. key is passed: Use hashCode(key) % n_partitions
2. key is not passed: simple round robbin
3. You explicitly mention which partition to write to
4. You define custom Hashing function


Producer Side Architecture
IO Threads
Replication
Broker 0 Leader
Broker 1 Follower
Broker 2 Follower
Topic A Partition 0
Topic A Partition 1
Topic A Partition 2
Acknowledgement
max.in.flight.requests.per.connection
buffer.memory
max.block.ms
send
key.serializer
value.serializer
Producer Record
Topic
Partition
Timestamp
Message Key
Message Value
Serializer
Partitioner
key-based partition
Buffer
Topic A , Partition 0
Topic A , Partition 1
Topic A , Partition 2
Batch
msg 1
msg 2
msg 3
Response Acknowledgement
batch.size
compression.type
linger.ms
Failed ?
YES
NO
Retry ?
request.timeout.ms
delivery.timeout.ms
retries.backoff.ms
Record Metadata
Partition
Offset
Timestamp
Exception
kafka
zoom

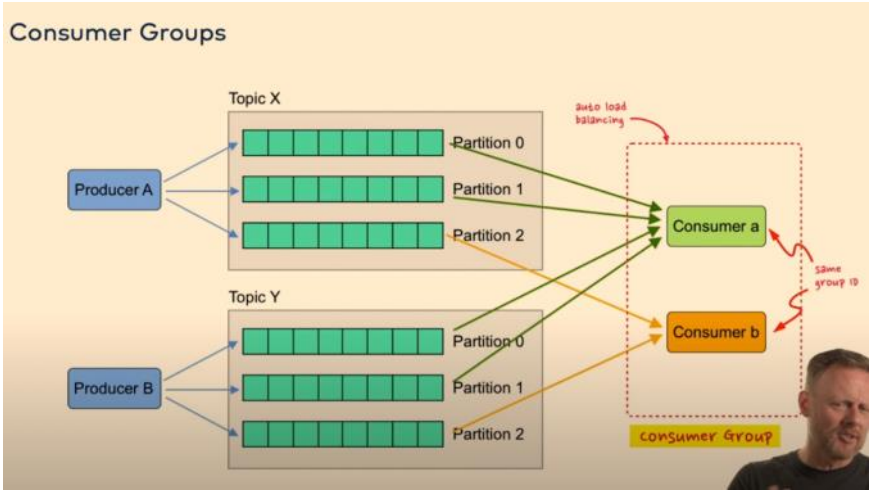

Producer sequence diagram
Producer
Producer Network Thread
Bootstrap
Broker 1
Leader
Broker 2
ISR
Broker 3
send()
Determine partition
Wait for metadata
Find leader
Add to local queue
Send message to leader
Fetch
Send response
Callback

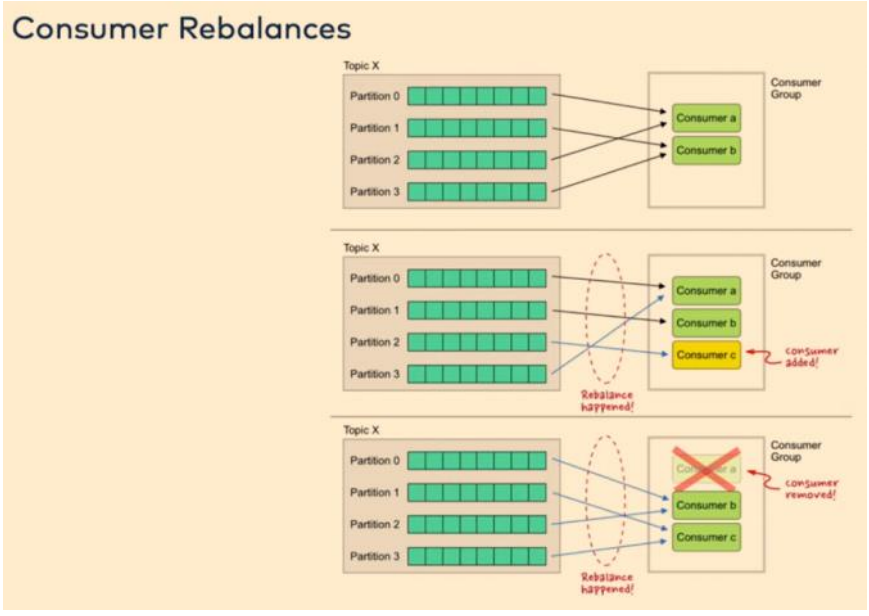

Producer Guarantees
Producer A
write
ack
Broker 101
Broker 102
Broker 103
Broker 104
leader
follower
Acks 0 (NONE)
(by default)
Acks 1 (LEADER)
Acks -1 (ALL)
out of sync replica



* Hard to implement
* You need to sync both (prod, cons) → you need something as a transactional api server

CONSUMER DETAILS





OFFSETS

- What if your Consumer loses connection? or new consumer group gets added
- Kafka Offsets Committed vs Current vs Earliest vs Latest Differences | Kafka Interview Questions

Types of Offsets

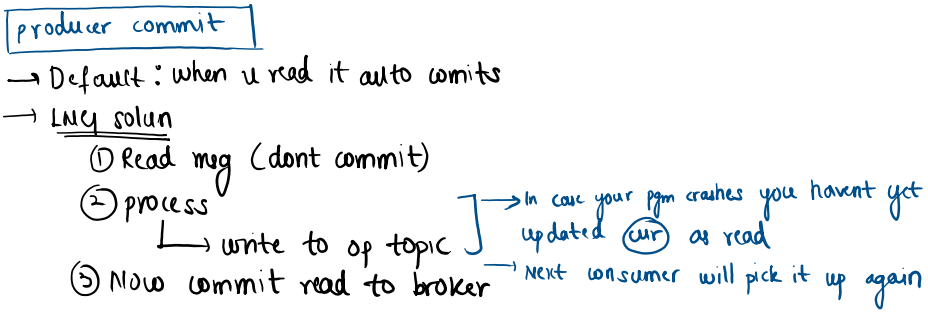
- Current offset: Kafka maintains this offset of current message being read by the Consumer

Types of Offsets

- Current offset: Kafka maintains this offset of current message being read by the Consumer
- Committed offset: Consumer after successfully reading message will commit this (Auto in python) indicating that till "i" all messages have been read

Auto.offset.rest = "earliest | latest"

- earliest: start reading again from start of the partition
- latest: start reading from "latest_commited_offset" of last read



7 − 2 = 1

DOCS

- Best Intro: 4. How Kafka Works | Apache Kafka Fundamentals
- Confluent Playlist: Getting Started with Apache Kafka®
- Delivery Semantics: Message Delivery Semantics in Distributed Systems: Kafka included