System Design

↳ Gaurav Sen Playlist (Quick Intro, overview imp topics)

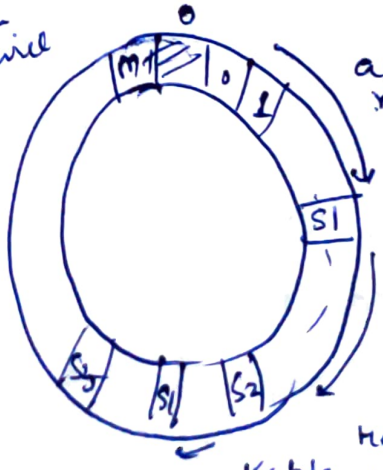
* Horizontal Scaling (Pros): { Scales well, Resilient }

Vertical Scaling (Pros): { Fast inter process communication, Data consistent. }

We make hybrid systems taking big boxes (vertically scaled machines) which scale well horizontally.

* Load Balancing
  - used in distributed cache.
  - Consistent Hashing
    - while adding a new server, we need to make balancing remains proper, so that old request ids, which were mapped to previous servers, keep getting served from old ones so that cache remains in a good state for the existing servers. Hence consistent hashing.

assuming load balancer, uses a hash function value to map to a server.



* each server will have K hash functions. Hence they occur on ring K times. (each)

all req. here mapped to S1

0 → m-1 request Ids hash value

$Hash(r_1) \% m$

* S1 + S2 ... all servers mapped by K hash functions.

S1 → Server value hash mapped

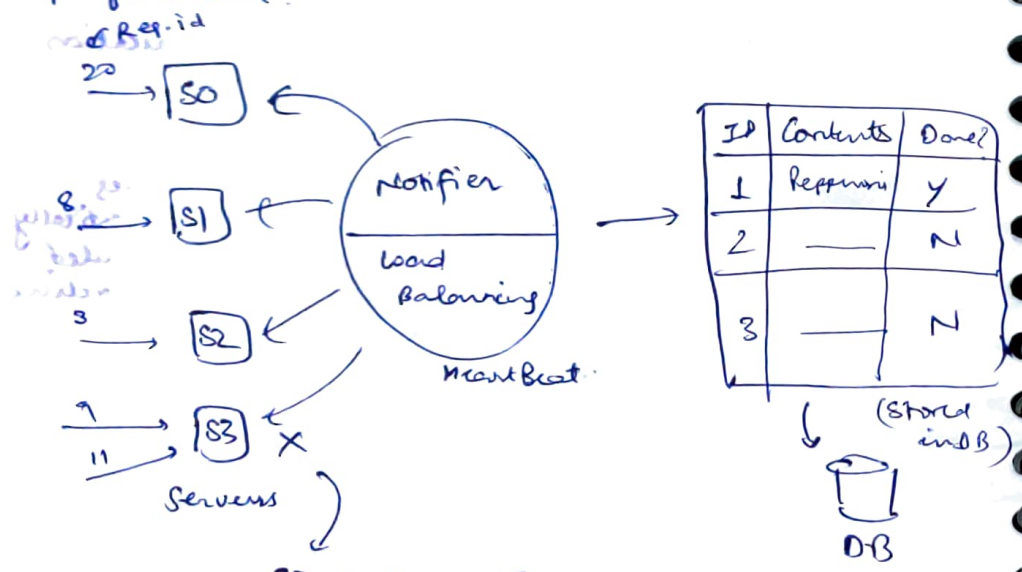
S2 $Hash(\%m)$

Hence they occur on ring K times.

∴ We now have more consistent setup

Message Queue

* Example dominos pizza → for all requests you get, add them in a list. Thus client & pizza shop can work asynchronously and have faster performance.



S3 crashes. ✱ Heart beat check will confirm it's dead. Query DB to check incomplete orders. And route them again (Notifier). Here 3 id is also incomplete, but load balancing will ensure 3 does not go again as it is being done by S2.

This combination of Notifier, Load Balancing + Heart Beat = Message Queue

eg - Rabbit MQ, Zero MQ, JMS

⇒ Monolith vs Microservices

Separation of responsibilities → Microservices.

Microservices
- → Good for smaller teams
- → ~~Complicated~~
- → one service goes down, other still up → hence high availability.
- → Parallel development can be done.

Monolith
- → Max context needed
- → Complicated deployment
- → All calls are Remote Procedure hence faster.

If S1 only calls S2 (S1 → S2) its better to include S2 inside S1 (make monolith)
↳ eg of monolith → stack overflow.

⇒ Database Sharding

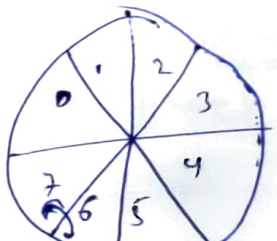
↳ Partioning data

✱ You Shard (Partion) data on a attribute say id / location etc.

✱ To have dynamic shards (flexible) ↳ Memcached.

Problem
↳ JOINS

When you need to run a query that uses data from other partition



:(- causes problem (expensive)

Advantages that a DB gives you over file system — (but why they are useless for images)

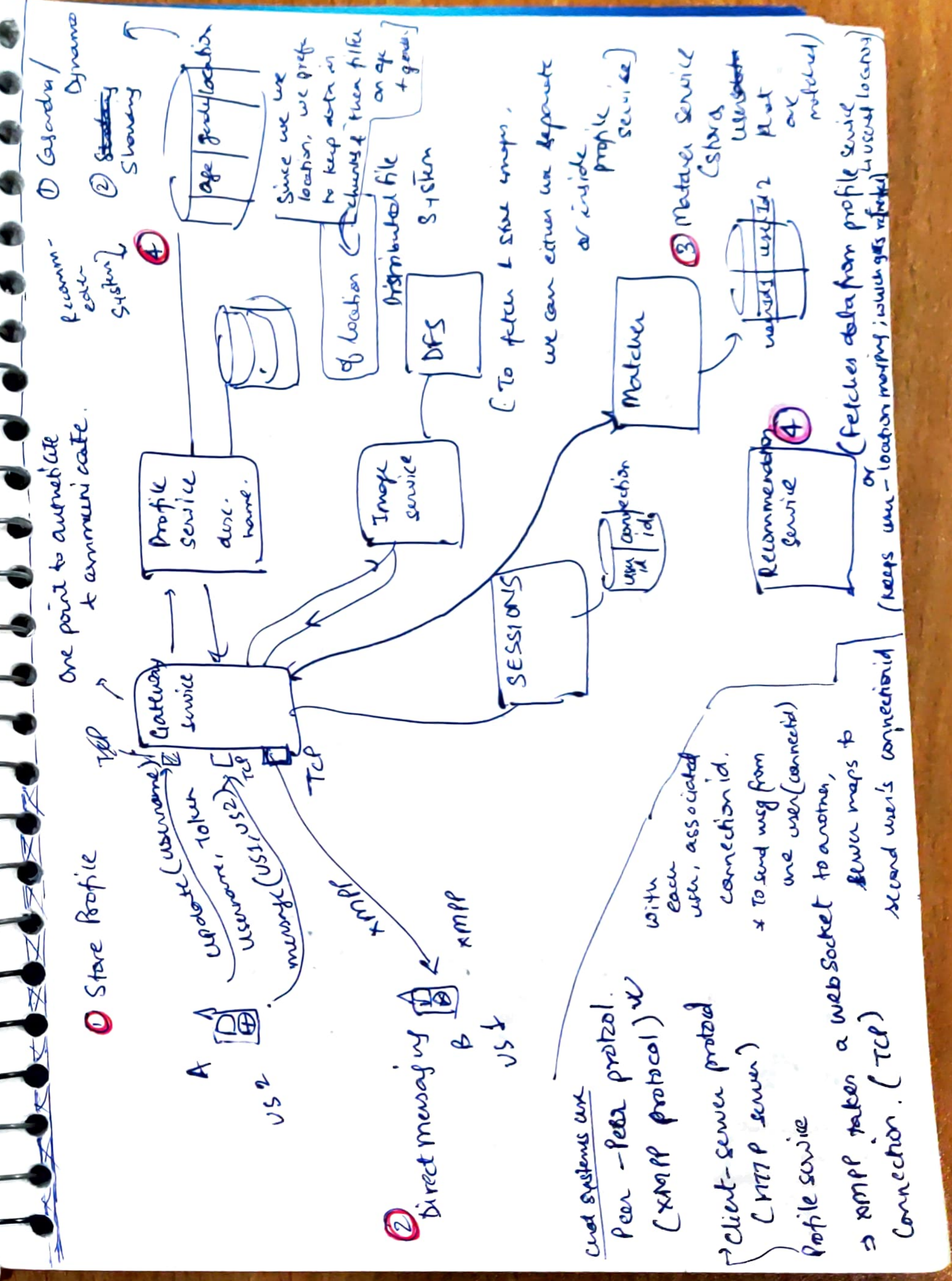
- Mutability [Ability to change] → Not needed by images. They won't be updated with DOCs
- Transaction guarantee → will be risky since if select * performed huge chunk will be moved.
- Indexes (search)
- Access control
  - ↳ one get same using file system.

You will not be searching on the content of the file.

- Advantages of file system ↓
  - cheaper
  - faster (since separate place of large files) Can be done with vertical partitioning using DB but again select * nightmare !!
  - Content delivery network (since these are static files, they can be hosted on a CDN)

Profile Id	Image Id	File url

→ file url of file in distributed file system.



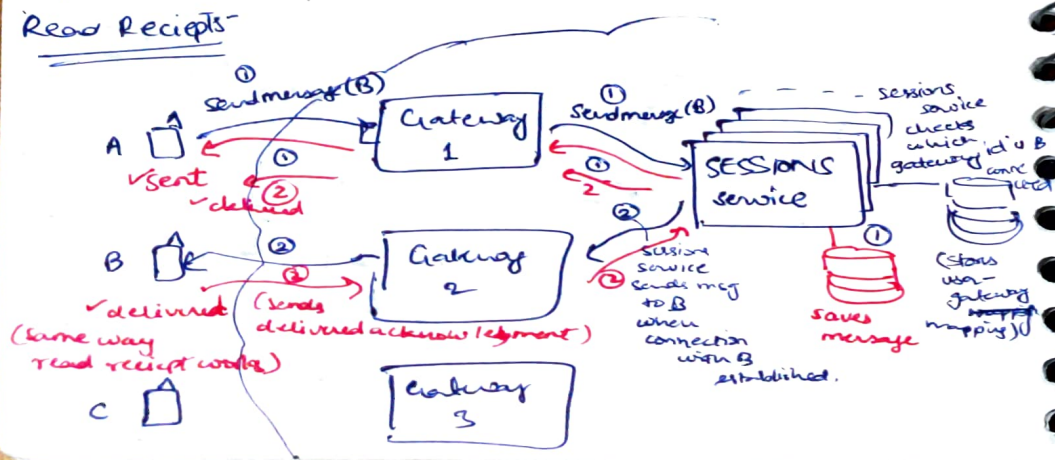
WhatsApp System Design

Will be design these features -

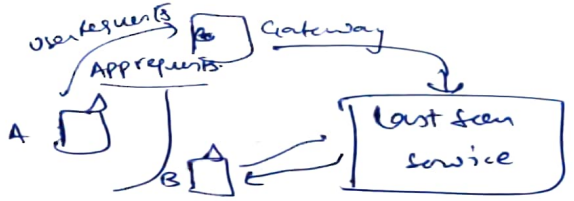
1. Group messaging
2. Sent + delivered + read receipts
3. Online / last seen
4. Image sharing → similar to Tinder design
5. Chats are temporary / permanent
6. One to one chat

* TCP connection (~~Rabbit~~ over ~~or~~ web Sockets) used as server needs to communicate with client.

One to one chat + Read Reciepts -



3. Online / last seen.



like long polling etc. make sure you don't update last seen table for app requests

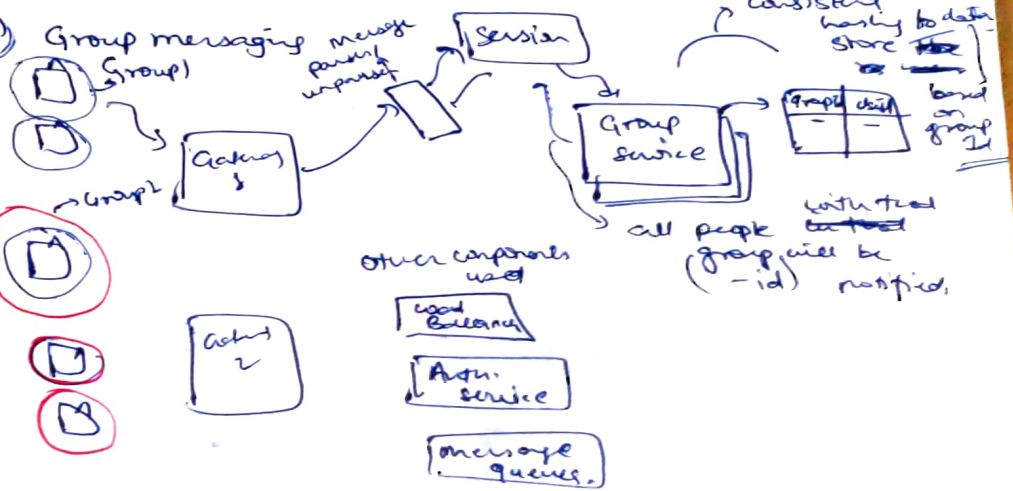
user	lastActive

* when user exists main app - a request is sent to update last seen table.

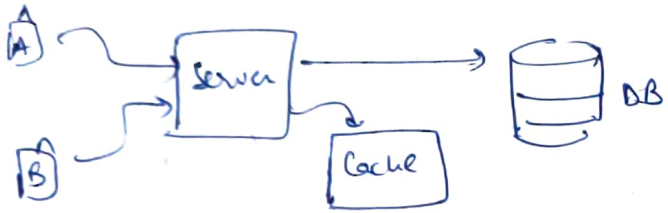
* B is checking last seen of A ie

* Till user is online, the user-requests are periodically sent which keep the last seen table entries updated (to current timestamp).

4. Group messaging



Distributed Caching



Benefits -

- Save Recomputation
- Reduce load on network (less calls)
- Reduces load on DB

- You can't store everything on cache as cache hardware is expensive → SSDs compared to commodity hardware of DB.
- Large data on cache will ~~reduce~~ increase search times at cache.
- Hence only relevant data (that seems re-usable) should be kept on cache.

Hence to clean cache; [Cache eviction] we have Cache policy

LRU
Least Recently Used.

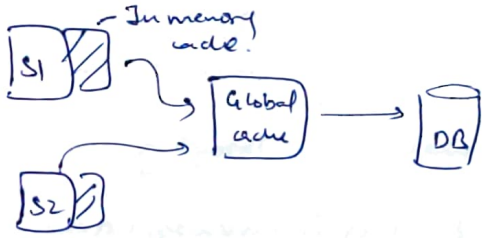
↳ oldest entry keeps getting pushed below, until it is evicted from the cache.

↳ eg- Celeb posts. Kept at top of cache. Everyone views it. Slowly keeps getting pushed below. Finally removed.

* If cache not used correctly it is harmful

everytime before going to DB, check in cache → * extra call
* thrashing

data keeps getting added/removed from cache without ever getting used



Cache can be placed in-memory at server level or can be a global cache.

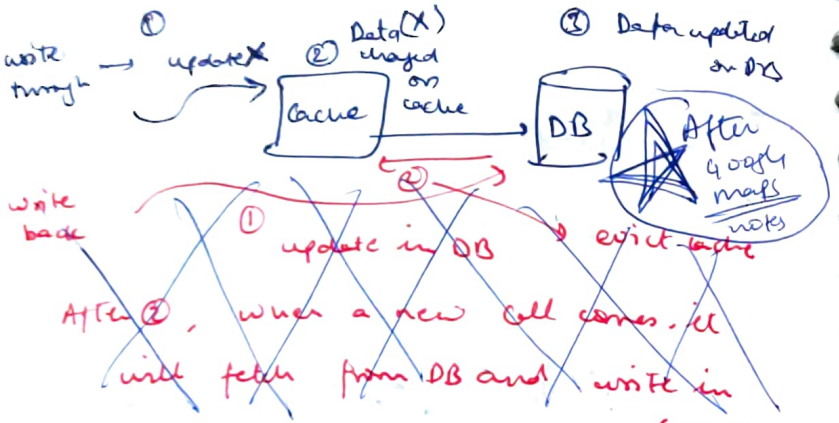
In-memory → Faster.
→ Not resilient (server crashes, cache gone)
→ Not accurate/
* Cache consistency an issue.
(Cache at all servers need to be at sync)

Global cache → Slightly less fast
→ Accurate
→ Can be scaled independently of servers.

Types of Characteristic of cache – determines accuracy of cache data

① write through

② write back



Problem with write Through]

* if it is in-memory cache, write through limit wale

Write Back]

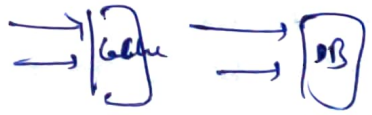
* with write back, ~~out~~ for each update we invalidate cache which is expensive.

What can be done]

Hybrid!!

if it is some non critical data, we can have data updated on cache (write through), then after some defined time, bulk change can be done at DB.

For financial data, we only have to use write back.



⇒ API design

⇒ If you have large response, good way is to break it by either →

- Pagination
  - client says return first 10, then next 10, so on
  - client asks for first 10, then next 10 ... so on
- Fragmentation
  - Server gives ~~not~~ broken responses one-by-one with an annotated piece say TCP no. 10, 11 ... last one -1.

Imp things →

① API name

② Parameters defining
- Only what is required should be a parameter

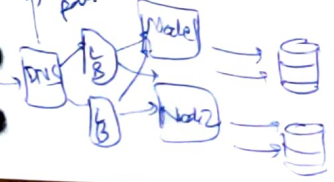
③ Response object defining
- Send only what is required

④ Errors
- Throw relevant errors

⇒ Single Point of Failure

* If your system has a single point of failure it means system is not resilient.

DNS points to single point of failure → We can have load balancer



We can have more nodes (servers)

master-slave (DB)

system in multi-region

=> Youtube Capacity Estimation ↗

=> NOSQL Database

SQL -> Structured Query Lang.

No - SQL

Data stored as →

ID	Name	Address	Age	Role
123	John Doe	1	23	

Address Table

ID	County	City
1	India	Lucknow

Data stored as

ID { name: {ID}, adress: { }, Age: 1, Role: 2 }

ID2 { name, gender: 1 }

JSON

Not necessarily same

Adv. of NoSQL over SQL

① Insertions & retrievals easy (requires JOINS in SQL)

② Schema is easily changeable. (In SQL, would need to add data for all. Hence easily queueing as DB locked for that interval)

③ Built for scale

By default supports horizontal partitioning (easy sharding)

④ Built for metrics/analytics/aggregates

Disadvantages →

① Not good for updates (delete+ insert) (Consistency across nodes is an issue)

② Read / Fetch of a value is slower. (Goes to a block, finds key (say age) then returns)

③ Data doesn't have relational property.

④ Joins are hard (to join on a property, run through each block then merge ----) in SQL, easy as it is built for joins.

Cassandra (Eg of NoSQL DB) →

(0 → 200)

(0-100)

Hash (reqId)

requestId

→ Data is replicated with RF of 2. (Replication factor)

So when data added

• Hence load balancing good.

• RF provides redundancy/replication. eg: data at 2 replicated 3 nodes cluster

★ Cassandra has concept of concensus. → achieved by Quorum.

(100 - 200 on node 2)

How internally used another cluster

Cassandra cluster

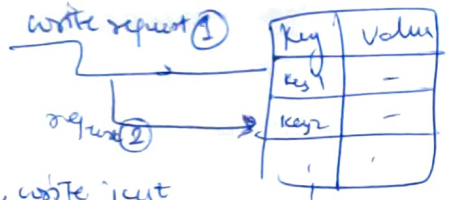
If quorum = 2, with RF = 3,

⇒ Two nodes must agree for a value out of 3 before sending it to user.

else → DB error.

⇒ Another feature of Cassandra which stands out is way of writing to DB.

So, it maintains a log file

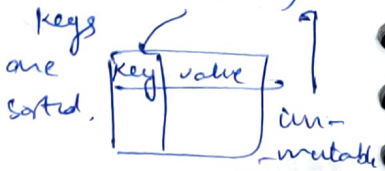


So, write just happens where the pointer is currently at.

⇒ NO Searching of last index to insertion

Secondarily, after a period of active this log file is moved to a SST (Sorted String Table) table.

contents of



Say initially a write for key 123 → name = John Doe

again later, key 123 → name = John Do-Ne

So, multiple keys in DB.

No problem for consistency, we can use latest Time stamp.

But storage ↑.

So, Cassandra does compaction. (+ Elastic Search)

↓

Two SSTs are merged.

↗

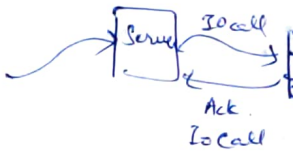
Merge sort → 2 sorted arrays merged.

Space complexity O(min(m, n))

Time O(n).

⇒ For delete, Cassandra has concept of Tombstone so basically when key 123 deleted, tombstone (kind of flag) is set. Before read, you verify tombstone to be unset.

⇒ Scaling write operations.



- Lesser IO calls ↑ Condensed write queries.
- Write using Linked-List. O(1) time.
  ↳ Log files use linked list DS to store things.

Hence Adv → ① lesser IO
② fast writes

DisAdv → ① Slow reads
② Additional memory consumed at server (for condensing queries)

kind of ok (Not big problem)

Problem!!

↑

To improve reads, we can use Sorted Arrays.

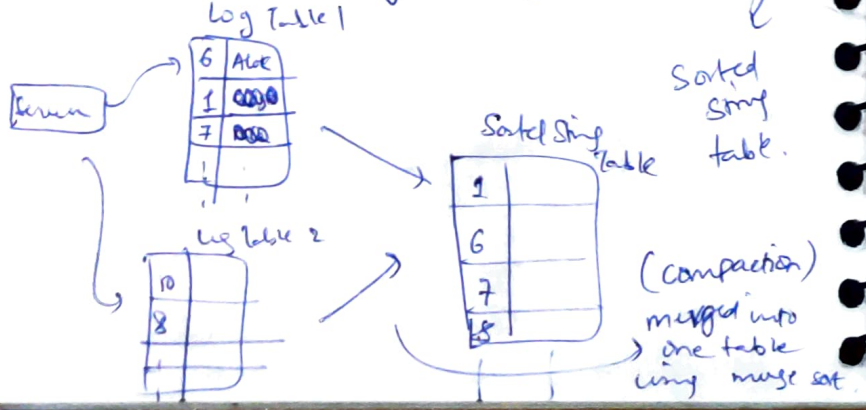
Hence for write → O(1) → Linked list

read → O(log n) → Sorted array table.

★ So at DB, we sort the writes by client and persist them inside sorted array table.

or

Sorted string table.



★ Deciding when to merge,

↓

merging every small bit is a heavy operation. To insert 6 size table, in 10^6 records, required n log n time!

⇒ Hence we merge when compaction time is reasonable.

Also merge intelligently,

another 6
another 6
→ merge. then merge to get 24.

6 ⟷ 6

12 ⟷ 12

⋮

To speed up reads, another thing can be done is to keep BLOOM FILTER.

Eg → when a word like CAT, CORONA present, flag for C is set.

Do q search → Helps reduce search.
comes. D not set → Not present

So for each sorted string table, we have a bloom filter.

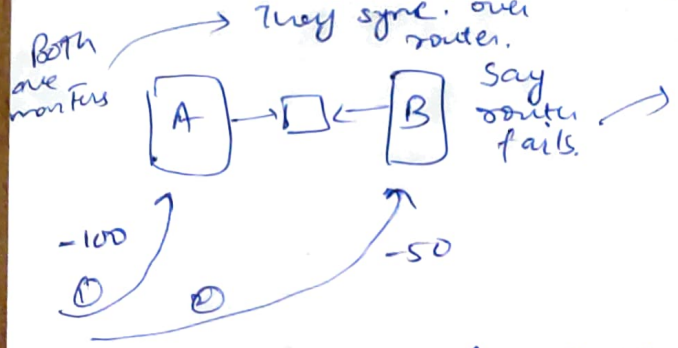
when merged → a new bloom filter created.

* before searching in actual SST, if not present in bloom filter, it will not be present in SST.

Distributed Consensus & Data Replication strategies

* for Databases, we use Master–Slave arch.

why can't we use Master–Master Arch?

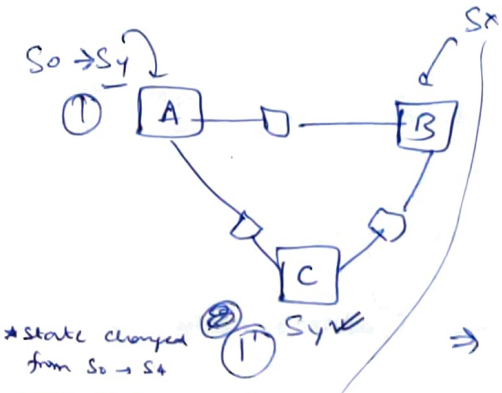


Call ① to -100 comes, A assumes himself to be master (as connection with B) lost.

Call ② to -50 goes to B, (B has entry of 120 still) assumes B master, deducts 50. So in balance 120, 150 was deducted.

* This problem is called split-brain problem.

To solve this we can have another node (C),



Assumption, router b/w A–B & node C can't fail at the same time.

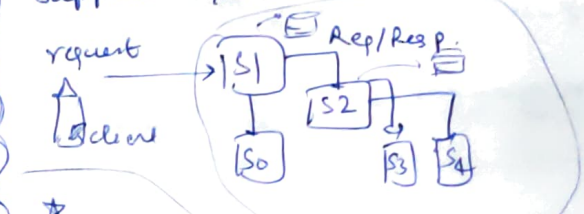
⇒ Requests can ad A & B.

when call at B for Sx, B calls C to check state. Here state no longer S0, it's S4. So transaction fails for user. B then pulls state of C. (i.e. S4)

27/06 Publisher Subscriber Model

Need →

Suppose system like



S1, S2 → servers

S1 → S2 } Async calls
S1 → S0 }

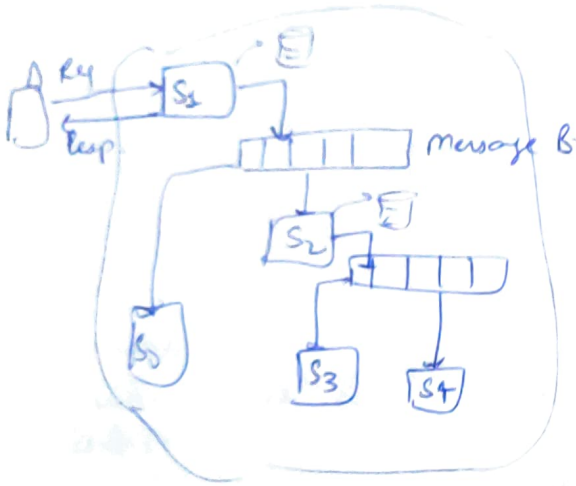
S0, S2 waits for resp. from S3 & S4 then returns to S1

* Hence problem → long wait time at S1. ⇒ If S1 fails, data inconsistent at S1 & S2.

So, better way to use message brokers

↕

Kafka, RabbitMQ



S1 publishes, S2 & S0 are subscribers to message broker.

S1 writes to message broker/message queue, (publishes)
and returns resp to client without any wait.

Resp of message broker that request will be send to S2 & S0.

Advantages :

- Generic message/event can be sent by services to message broker.
- Transaction guarantee : even if S2 down, broker ensures that at same point of time (when it is live) these messages (in queue) will be replayed on S2.
- More scalable.

Disadvantages :

① Poor consistency → Say financial system

S0 → invoice service (deducts bank commission per transaction)

S2 → fund transfer service

→ S2 down.
→ for each req from client S1 will be processed.
S2 req in message queue. Hence inconsistency in bank amount.

② Not idempotent ✗

↓

Say when S2 writes on its message broker, it fails. In S2 DB, change done.

Since it fails, upper message broker replays message. Hence again in S2 DB, change done.

Example

→ Gaming services (in analytics)

→ TWITTER uses it ✓

Perfect use case.

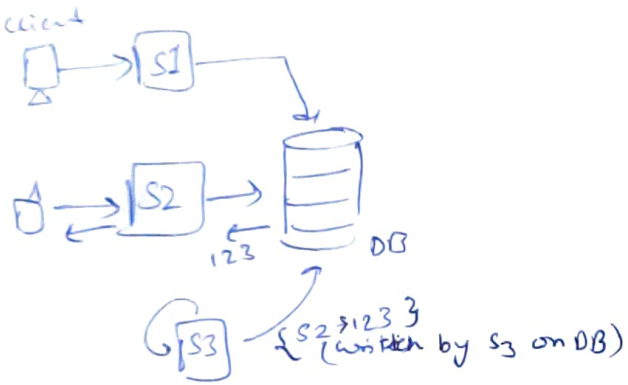
Tweets get published, subscribed accounts get notified

⇒ 4/7/20

ANTI-PATTERNS

↳ Using DB as message queues is considered (should be avoided) an Anti pattern

Disadvantages of using DB as message queues



① Polling intervals measured to be correctly.
Short intervals → high load on DB
Long intervals → inefficient / bad user experience.

② Read / write messages from DB will have problems like locking etc.
Also DB's are optimized for either read or write.

③ Delete (of read messages) will be an overhead.

④ Scaling difficult.

So generally, if system is a small service → DB fine for message queues.
if scalable → message queues.

Adv of message queues →

① No poling. When new message for a server, the message queue pushes it to the server.

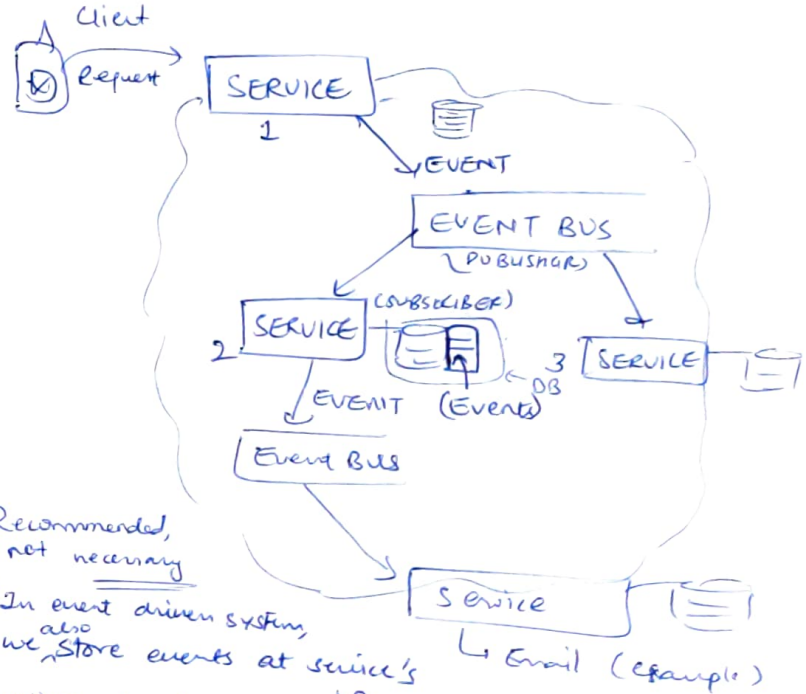
② Read and write / delete of messages all responsibility of message queue. For us, it is a black box
+Scale

Disadv of message queues →

① For not a large server, DB already handles. So unnecessary complexity added.

② Infra setup costly; requires trainys.

⇒ EVENT DRIVEN SYSTEM



Recommended, not necessary

* In event driven system, we also store events at service's DB.
So when eventbus sends to service 2, it adds in its DB as well.

Examples of Event Driven Systems → GIT, React, Game servers (Pubg, CS)

Advantages of ~~adding~~ storing events in DB | Overall advantages of EDS

① Availability: For some reason, message queue / parent service unavailable, if event stored in DB (not get consumed), service can consume/work on it.

Hence high availability, but we have poor consistency.

(~~message queue~~

② Easy Roll back

↳ If some bug happens in service after execution of an event, if event stored in service, we can rollback to previous state.

③ Replacement → (Say service 5 wants to replace service 2) ..

So you can replay all events done by service 2 and then subscribe to corresponding events ✓

④ Transactional guarantee –

If an imp. task like sending invoice mail done by service, if we store event at service, we ensure that it will be executed ✓

⑤ Stores intent

↓

with intent of each event stored, you can select / remove set of events & can create desired data state.

eg - Remove new entries after 6 PM time

Disadvantages →

① Consistency (as in adv first pt.)

② ~~Replace~~ Replacement not possible for Gateways. ✓ (Creating new service from a Gateway → service ✓ difficult)

③ Less control (as request - response, you can control timeout, etc. here message bus as a component reduces control of service)

⇒ Since there can be many events coming, (re-applying events, rolling back can be difficult)

Hence it is good to squash events in DB.

Other Practical disadvantages ↓ (practically faced)

① Since S_1 ~~resp.~~ is just Event Bus, it's difficult to determine complete flow of input → email.

② Migration difficult (to move ~~from~~ a Event listening / sending service, to a req - resp. service is costly migration)

NUTSHELL → In Event based systems, services send events if they think needed by other service.

In req/res, a service gets request, fetches data and completes execution. ✓

~~Thus~~, all adv & ~~disadv~~ disadv. around this point.

INSTAGRAM SYSTEM DESIGN

1. Store / Get images → Same as Tinder (Filesystem, behind CDN, etc.)
2. Like / comment } on images
3. Follow someone.
4. Publish a news feed.

2. (Like / Comment)

Entity Relation (ER Diagram) -

Like

Like WID	ACTIVITY ID	USERID	TYPE	timestamp	State
	ID / POSTID / comment ID		comment / POST		0/1

COMMENT

ID	ACTIVITY ID	TEXT	USER ID	timestamp

LIKES (AGGREGATED) ← for get all likes (we dont want to do select * on like table)

↑ imagine above on news feed for every post

ACTIVITY ID	LIKECOUNT

POST (Image)

ID	Image URL	UserID	Timestamp	ActivityID

* Different way → + ACTIVITY TABLE

ActivityID	TYPE
	COMMENT
	POST

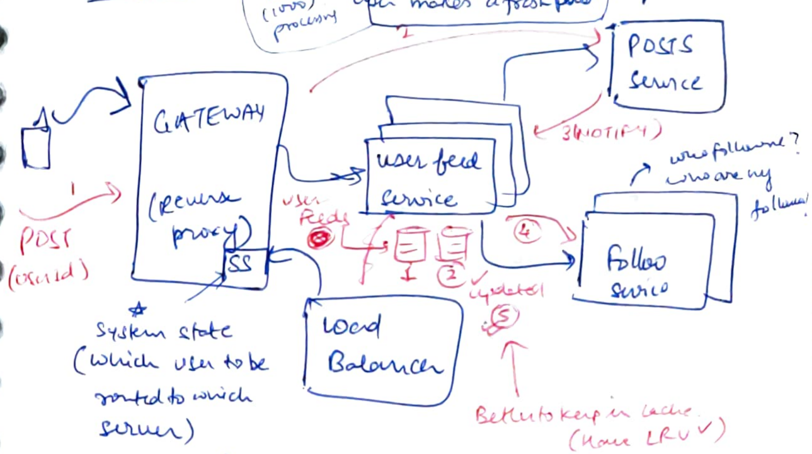
3. Follow → (my followers, who am I following)

User	Followers
Alok	X
Alok	Y
Alok	Z
X	Alok
X	Y
Y	Alok

★ Lets say a celebrity makes a post, now we are updating cache of each user's list of posts. This is put on top of each user's cache.
But if million followers, it means many changes!
→ we can say avoid it for celebrity. When user makes a fresh post, we can update it. / call, returns posts by given userId.
(we have batch (1000) processing)

4. News feed I

Design



To get news feed I

getUsersFollowed by (userId) ← at follow service
← returns Set < UserId >

getPosts by userId (Set<userId>)
← returns Set<Posts>

★ But for every request for news feed, if we query POST service, very heavy.
So better to PRE-COMPUTE.

whenever we get POST (url) request,
1. Post service updates posts
2. Notifies User feed service

To update the news feeds of all followers of userId.
2. (queries follow service internally)

GOOGLE MAPS Algorithm:

~~Design~~ Designing a location based database

We need → (Inp: params) ① measurable distance blw 2 points → uniform assignment of locations → Scalable granularity of locations

② Proximity (people close to you)

⇒ Cache :- [Grokking]

Write through:

- Data is written in cache and thereby in DB. Single operation.
- Adv :- Consistency maintained.
- Disadv :- Since per a write request, 2 writes happen (Cache & DB). High latency.

Write Back:

- Data written in ~~both~~ at cache alone, Then periodically or after some event, all writes moved to database.
- Adv: Very fast operation. Low latency.
- Disadv: If cache fails, data lost.

CDN :- Content Delivery Network

↑ eg of Cache.

→ Helps in serving static data.

⇒ Indexing in NoSQL

⇒ Indexing in NoSQL is based on a partition Key.

↑ Key (or hash result) on basis of which data is partitioned.

→ You can create custom index key to reduce search times. eg → if an data we want to have all rows where country = India & gender = female ; Better to have index (Sort-Key) eg country + gender + city.

Now, Select * from users where sort_id like "INDIA + FEMALE %"

Regex ↙

☆ Thus we can create custom keys if required by duplicating data. Not a concern. NoSQL can't have data normalized at all places.