Recap

- Case Study - delicious - bookmarking website
- Minimal Viable Product (MVP) features
- Internet Connection - IP Address
- How to find IP Address given Domain Name - DNS
- How to prevent ICANN from becoming SPof, bottleneck } tree like Hierarchy of DNS servers

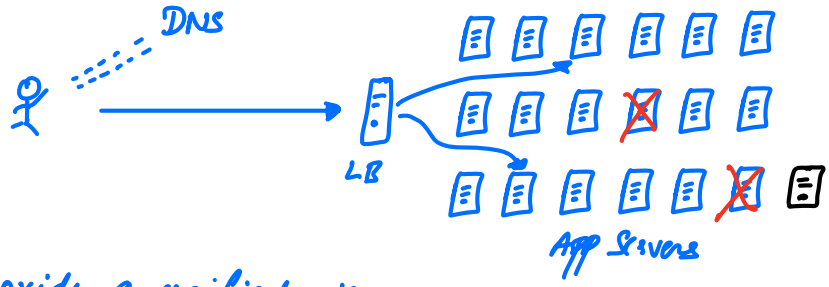
- Scaling challenges - data is too large

Vertical / Horizontal Scaling

Scale UP — Scale Out

- How to present a unified view to user?

Load Balancer



- Load Balancer - Provide a unified view
  - distribute the load evenly across all servers

- How does LB know about what servers are available?

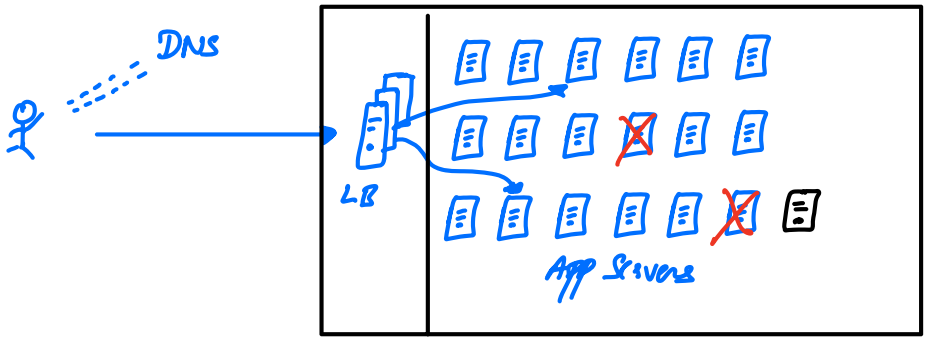
Heartbeat / Healthcheck

push from Server to LB — pull from LB to Servers

- How to prevent LB from becoming a SPof & bottleneck?

bottleneck - LB's are able to handle higher load (∵ simpler)

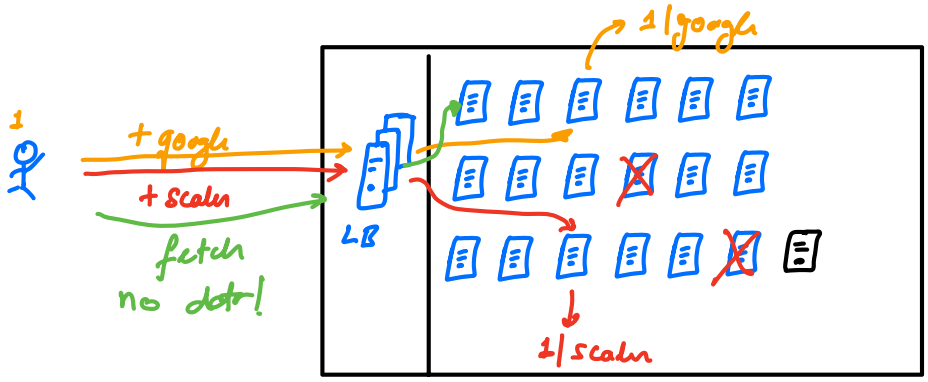
SPof → (v. large scale) multiple LBs → DNS can store multiple IPs for a domain



API gateway / Rate Limiter / Load Balancer / Reverse Proxy – towards class end

---

Q1 How to route the requests? which request to which server?



We cannot answer the routing Question until we know

↓

Q2 How is the data distributed?

① Can all the bookmark (data) be stored in 1 server?

No!

② even if we could store all data on 1 server, should we?

No! → it will not be able to handle the load (bottleneck (# requests/sec))

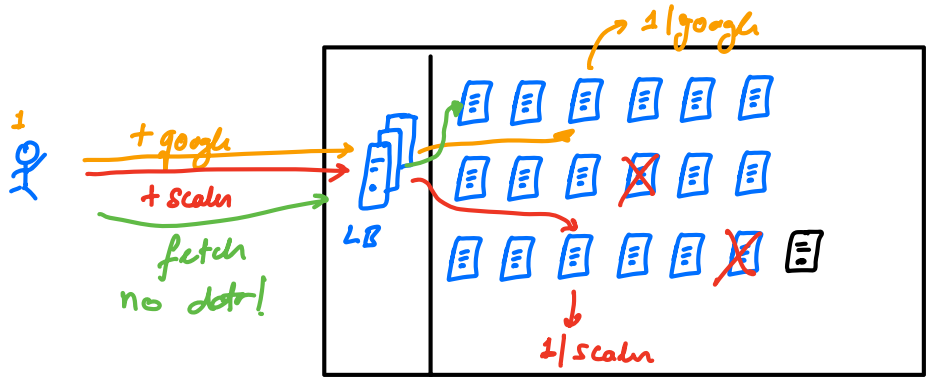
→ It might fail SPoF

∴ we need to distribute the data.

③ Should I distribute data randomly?

can any (user id / URL) go to any server? No!

No ∵ how will we retrieve it then.



∴ we should distribute data using user-id

All book marks of u1 go to the same server S7

" u2 " " " " " S3

" u3 " " " " " S7

⋮

Sharding I never done randomly
always done via some sharding key

user-id (in this case study)

Users

id	name	phone	address	username	pass-hash-salt
1	Umes	..	..	..	..
2	Nidhi				
3	Jai				
4	Rasif				

Vertical Partitioning

Normalization

(inside 1 server)

Users	username	pass-hash-salt
id	..	..
1		
2		
3		
4		

profiles

user-id (FKey)	name	phone	address
	Umes	..	..
	Nidhi		
	Jai		
	Rasif		

Horizontal Partitioning

Users-1

id	name	phone	address	username	pass-hash-salt
1	Umes	..	..	..	..
3	Jai	..	..	..	..

Users-2

id	name	phone	address	username	pass-hash-salt
2	Nidhi	..	..	..	..
4	Rasif	..	..	..	..

- might b within same server
  - ↳ faster indexes
- might do this across multiple servers — Sharding

Sharding — horizontally partitioning data across multiple servers

always done using a sharding-key

for this study our sharding key = user-id

Task: find a way of mapping user id → server id

which user's data should be stored in which server?

Attempt 1

Round Robin ✗

1. ✓ is load even?
2. ✓ fast to compute server id from user id
3. ✗ what happens if # of servers change?

N servers in total
= 5

U1	U2	U3	U4	U5
U6	U7	U8	U9	U10
U11	U12	U13	U14	U15

```
int N = 5; // No of servers
int getServerID(int userId) {
    return (userID % N) + 1;
}
```

N=5

U1	U2	U3	U4	U5
U6	U7	U8	U9	U10
U11	U12	U13	U14	U15

N=4

U1	U2		U3	U4
U5	U6		U7	U8
U9	U10		U11	U12

ideally only the users for whom the server has crashed should be remapped.

However here, there is a lot of unnecessary remapping.
almost all users are remapped

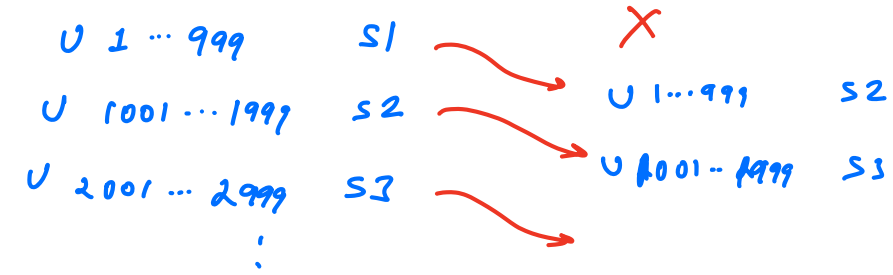
How to prevent data loss? Magic Fairy (Son Pari)
back ups for data using magic

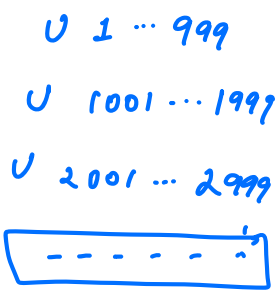
similarly when we add a new server
we have lots of remapping ∴ N changes

Attempt -2 — Range Based ✗

$$\text{Server\_id} = \left\lfloor \frac{\text{user\_id}}{1000} \right\rfloor$$

Same problem!





Attempt 3 — Store the mapping explicitly ✗

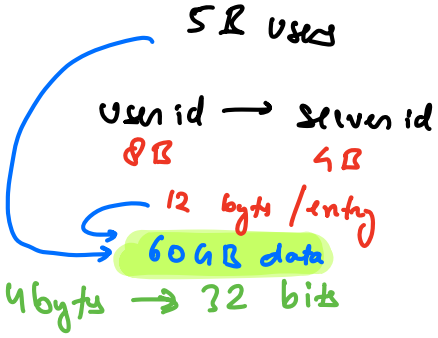
Map < user_id, server_id > in RAM (initialized randomly)

if server X crashes → find all user that were going to this server and assign new servers.
No unnecessary mapping.

if a server is added → select a % of users and move them to

this new server

issue: how to store this mapping?



4 bytes → 32 bits

2^{32} possible values
≈ 4 billion

# of users > 4 billion.

Youtube 4 byte signed int to store view count

$-2^{31} \dots 2^{31}-1$

≈ −2 billion ... 2 billion

① Mapping is large
60 GB

② store in RAM?
we would like to, but
① power fails?
∴ we should store on disk
② is disk too slow

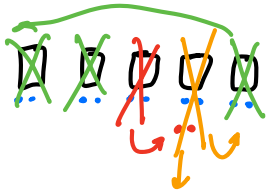
③ what if we have multiple LBs?
How to ensure consistency (same mapping) across all LBs!

V·V·V·V·V· Hard problem
impossible!!

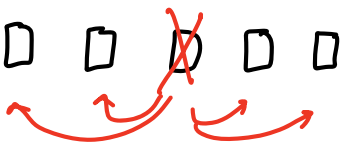
Solution — Consistent Hashing

Goals

1. data should be evenly distributed across servers
2. efficiently compute user-id → server id
3. should not have to store any mapping data
4. if a server crashes each of the rest of the servers should handle part of the load

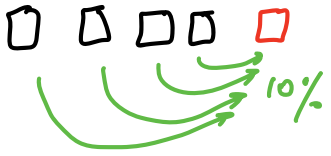


Cascading failure.



all remaining servers should share the burden of fallen server equally.

5. if a new server is added then it should relieve the load of all other servers equally



8:18 → 8:30

Hash Functions

function : mapping b/w two sets

$A \rightarrow B$

domain range (codomain)

$f(x) = y$

given x I can calculate y.

doesn't matter how many times I input x } deterministic fun

I will always get y

Hash function — seem random but deterministic

input : anything

output : no b/w 0 ... K

$$H_1(x)_{100} = e^{\sin\left(\left\lfloor (x^2+3)/5 \right\rfloor\right)_{2713}} \% (10^9+7)$$

(10^9+7) ⟹ (0 ... to .. 10^9+6) range

$H_2(x) = (\quad) \% (10^9+7)$

$H_3(x) =$;

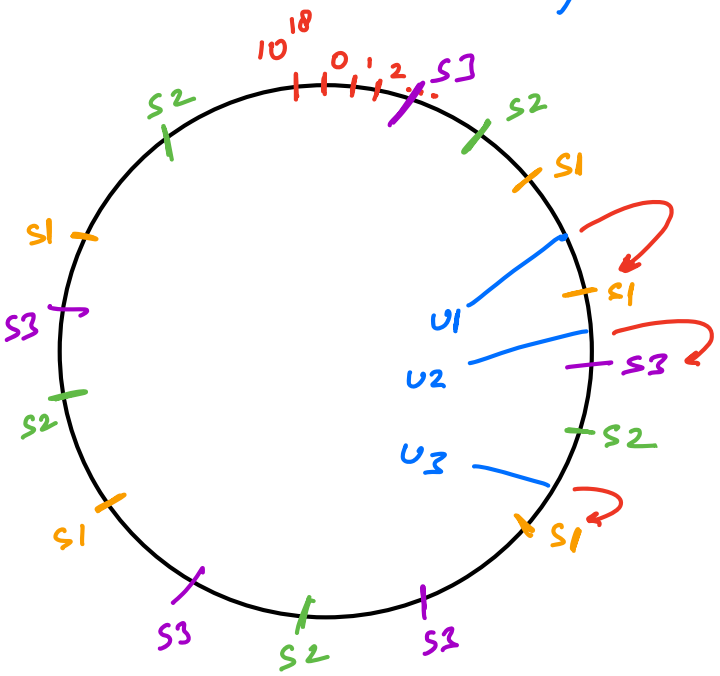
Consistent Hashing

① K hashes for each server → virtual spot

② 1 hash for the user

Hash(~) → ~ 64 bit int

③ All these hash functions have the same range



Consider N = 3 servers

S1 S2 S3

K = 5 hashes for servers

$H_1(S1) = 1000$
$H_2(S1) = 2726$
$H_3(S1) = 32$
$H_4(S1) = 47$
$H_5(S1) = 110736$

randomly distributed but deterministic

each user also has a spot on the ring

$H_u(U1) = 1002$

① Can the hash values collide?

Yes! However the chances are so low that we don't have to worry!

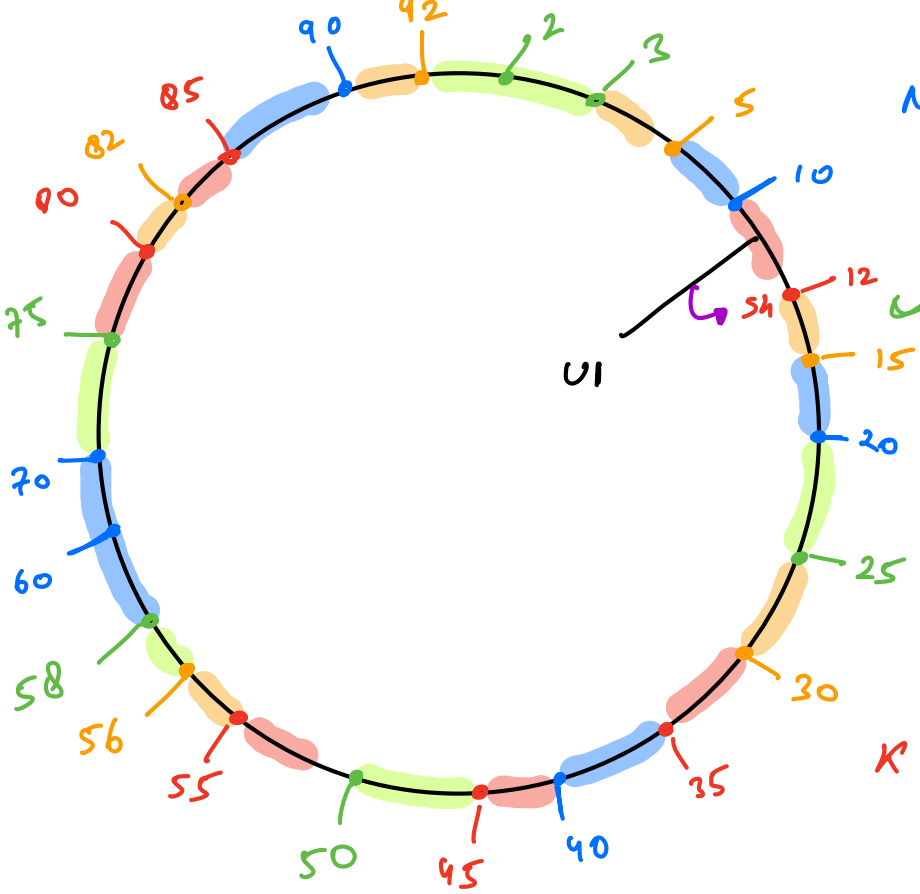
→ Ignore ∵ v. v. improbable!

& if it happens → no issue

② How to route requests?

given user-id compute the user's spot on the ring

find the first server on ring moving clockwise.



$N = 4$ servers $K = 6$ hashes

① is the load even?

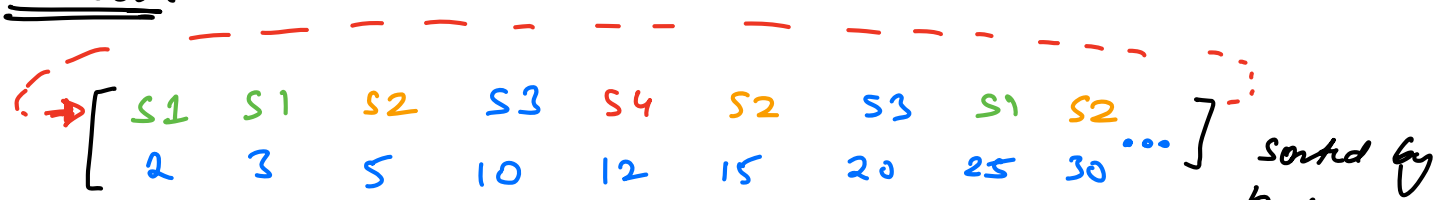
Yes!

∵ of randomness!

More hashes = more even distribution

$K = 16$ or 32 is used

in code



$H_U(\text{user-id}) \rightarrow 14$ (binary search upper bound with wrap around)

② load even? Yes ∵ chance

③ efficient to compute?

$\lg_2(N \cdot K)$

↳ # of servers = 10,000

↳ # of hashes = 32

$\lg_2(32000) \approx 15$

(15 μsec / query)

④ store anything?

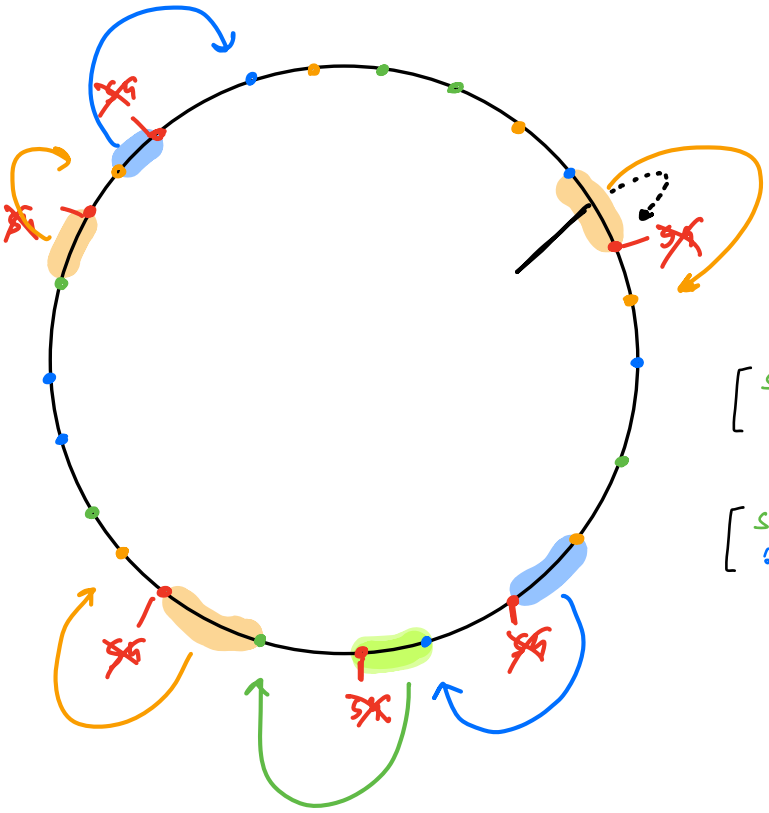
Yes, this sorted array

RAM? Yes ∵ < 1 Mb in size

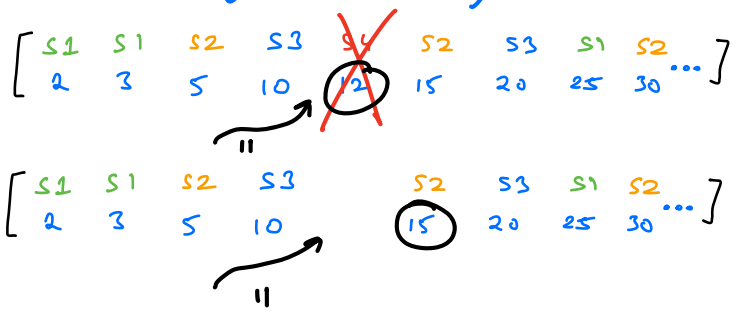
✓ Power failure? you can always recompute this array
∴ hashes & servers remain same!

✓ Ensure consistency across multiple LBs?
∴ all LBs are config with the same algo
& the servers are all shared for all LBs

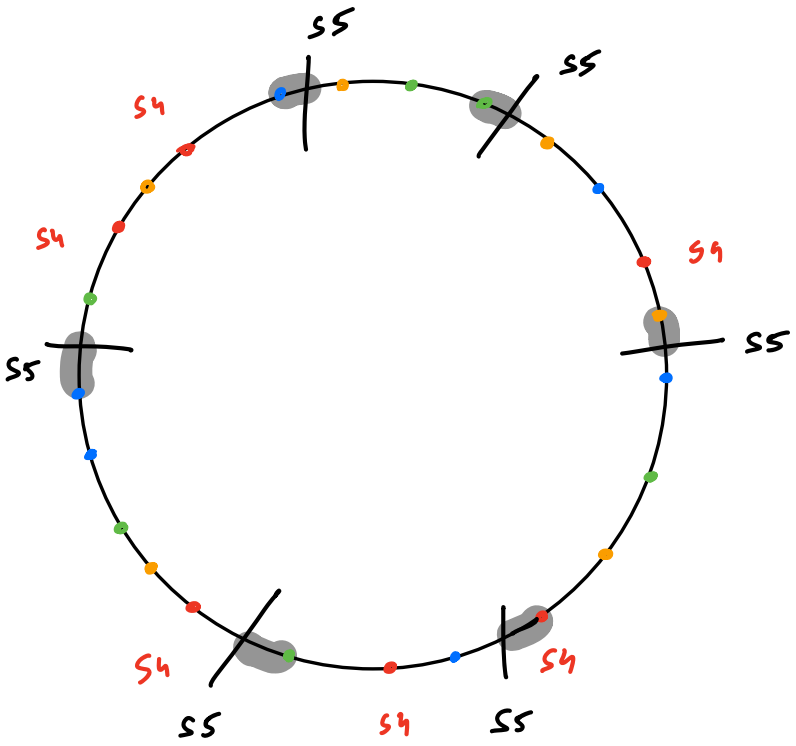
Server Crashes



S4 crashes
① Son Pari (Magic fairy) will somehow move data to next server
② Load of crashed server is equally handled by remaining servers



New Server is added



S5 is added

↳ passed through K hashes

S5 has taken some load from • • • • each server!

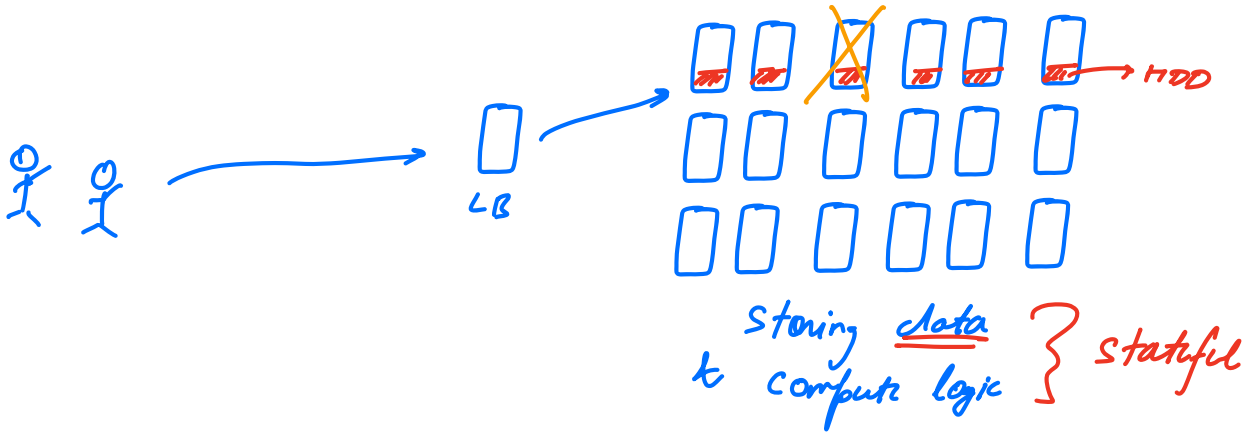
how does S5 get the data?

Son Pari *

Extremely rare that you will use consistent hashing!

explicitly - rarely used

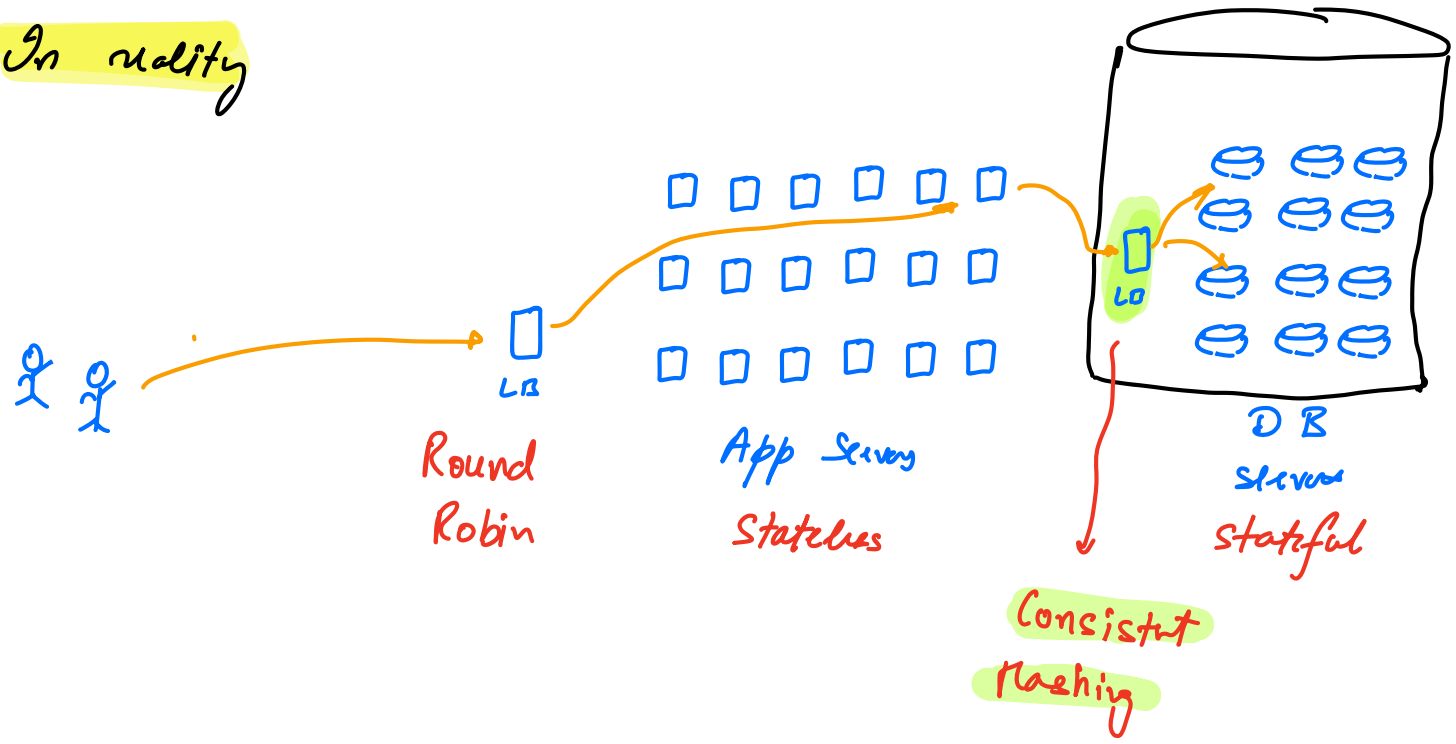
implicitly - used everywhere!!



bad architecture

∴ Coupled your data with code

In reality



State → data

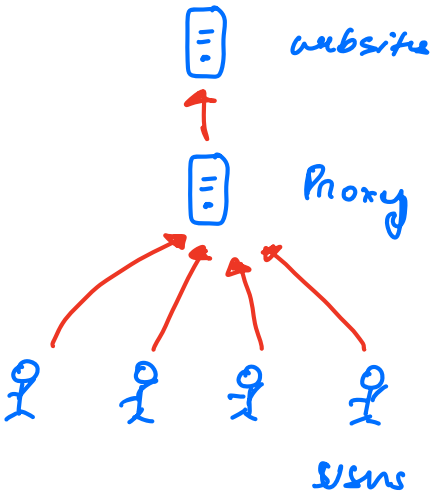
Stateful → storing data → DB is Always stateful

Stateless → no data stored

Reverse Proxy

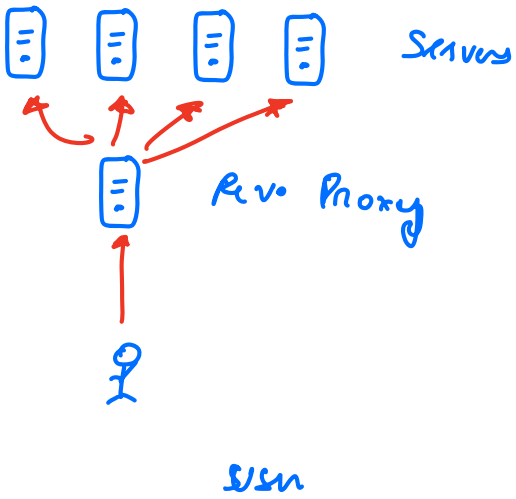
Proxy Server

hide the users from the server



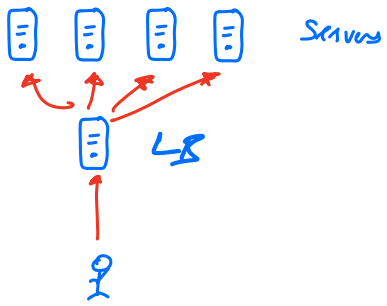
Reverse Proxy

hide the servers from the user



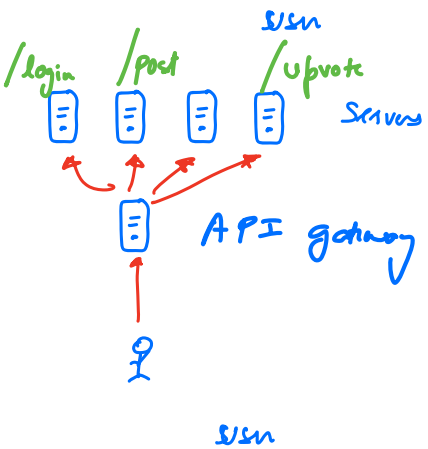
Load Balancer → unified view
└ distribute load evenly

is a reverse proxy

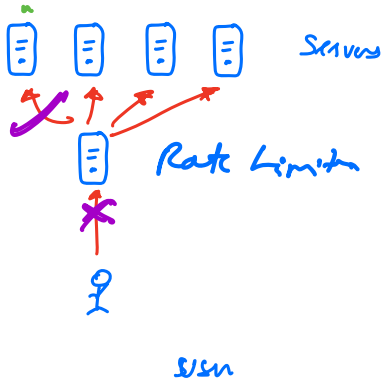


API gateway → microservices
└ API endpoints
└ route the req to the correct server.

is also a reverse proxy



Rate Limiter → prevents abuse
↳ throttles req
is also like a rev proxy



Nginx → Rate Limiter
→ API gateway
→ LB

Kong/...

---

store passwords in db

- Never store this in plain text

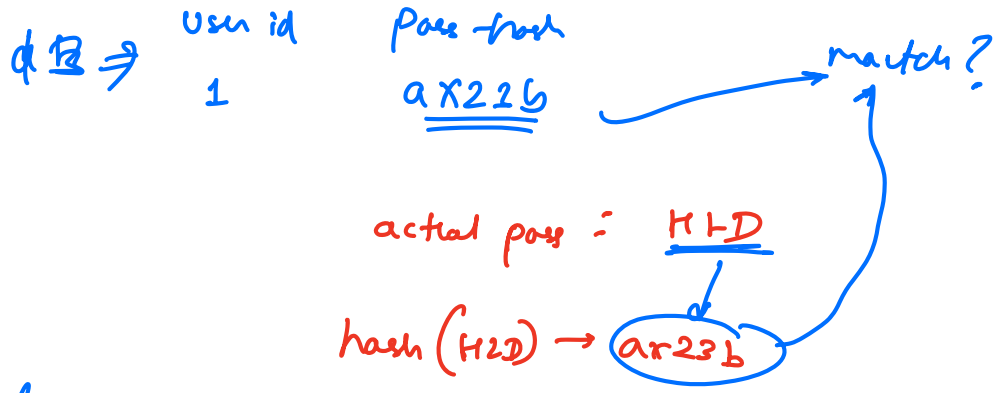
upper limit on pass length
(8-20 letters)
↓ security
↳ only when your store pass as plain text

users
id | pass
varchar(20)

- hash the password

destroys the value — anyone looking at DB cannot see the actual pass

how to validate password login?



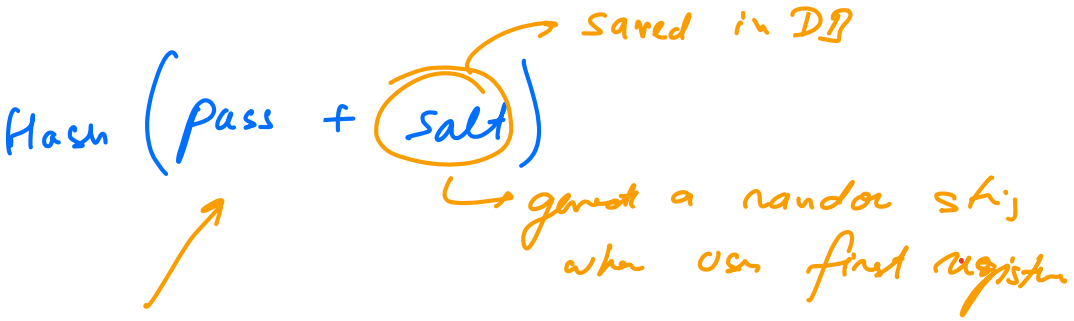
- Anubhava & Jai both have same pass

Naruto

User id	Pass_hash
1	ab22c
2	ab32c

= Same :: pass is same.

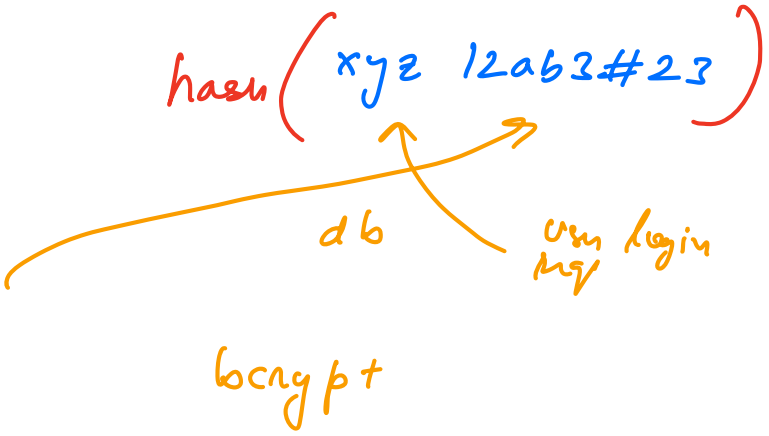
dictionary attack



dilip
user id = 10

pass = xyz

salt = 12ab3#23
(random)



Encryption ≠ Hash

reversible

lossless

irreversible

lossy

never find original data