Agenda

1. Sharding
2. Load Balancer Routing
3. Consistent Hashing
4. Stateful vs Stateless servers

support@scaler.com

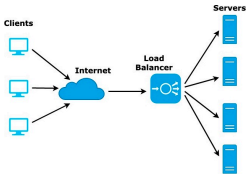
+91 7351 769 231

Delicious → bookmarking service

v. simple → 1 computer.

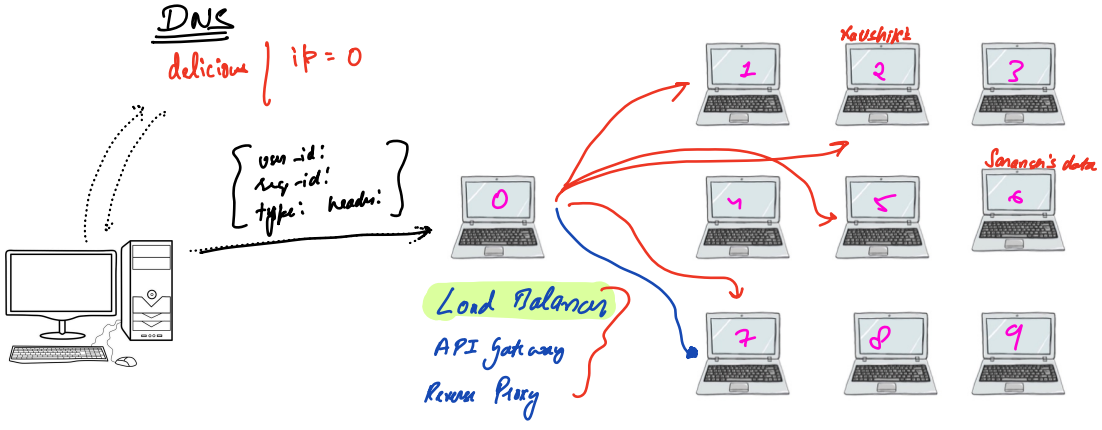
users ↑ ⇒ buy more expensive computers → could not scale infinitely.
vertical scaling

⇒ buy more of cheap servers
horizontal scaling.



Load Balancing

https://docs.google.com/document/d/1DxQzLpu1XPe_mRWsewNWtKL6E4uwKHQhBp7GX6Sg7qI/edit



? How does the Load Balancer track which servers are up & running?

Heart beat

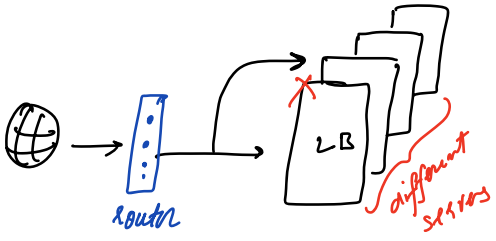
Servers will push update to Zookeeper

Zookeeper

Health check

Zookeeper will pull from servers

? What if the Load Balancer goes down?



DNS can also act as a LB on top of multiple LBs.

? Which machine should we send the request to?

Routing Algorithm

https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/

Goals

1. Even Load Distribution
2. Requests should not get dropped
3. Easily add & remove servers (without large overhead)

How to store the data?

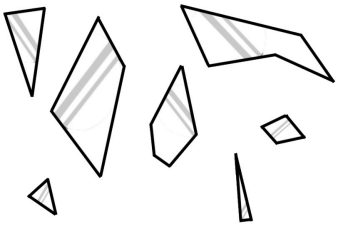
? Can we store all data on 1 machine? No!

data is too large
doesn't fit on single machine

? Split randomly?

challenge → how do we retrieve it??

horizontally
Split data across multiple servers according to some logic



Data Sharding

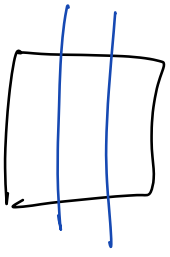
Partitioning

split vertically
↳ normalization

user-details
user-id | name | city | House no | Pincode
extract into separate table

split horizontally
↳ improve index performance
↳ reduce size of table

user-details
user id | name
10M entries



Sharding
→ data is too large
→ improve both read & write performance at the same time

user-details_1
user id | name
server 1

user-details_2
user id | name
server 2

simple mod
(Round - Robin)

Shard by `user_id`

↳ how to choose → later

servers = 5 $S_0\ S_1\ S_2\ S_3\ S_4$

users = 10

$U0 \to S_0$
$U1 \to S_1$
$U2 \to S_2$
$U3 \to S_3$
$U4 \to S_4$

$U5 \to S_0$
$U6 \to S_1$
$U7 \to S_2$
$U8 \to S_3$
$U9 \to S_4$

logic ⇒ [server_id = user_id % 5]

- server_id: calculating
- user_id: req
- 5: known

Pros

1. v. v. simple
2. No memory req.
3. even distribution

LB

```
server_count = 5

fn route_request(r) {
    user_id = r.user_id
    server_id = user_id % server_count
    r.redirect(server_id)
}
```

Cons

old = red → 5 servers
new = blue → 6 servers (+1)

$U0 \to S_0$ S_0
$U1 \to S_1$ S_1
$U2 \to S_2$ S_2
$U3 \to S_3$ S_3
$U4 \to S_4$ S_4

$U5 \to S_0$ S_5
$U6 \to S_1$ S_0
$U7 \to S_2$ S_1
$U8 \to S_3$ S_2
$U9 \to S_4$ S_3

Server goes down
↳ data??
prevent data loss → replication
↳ diff class

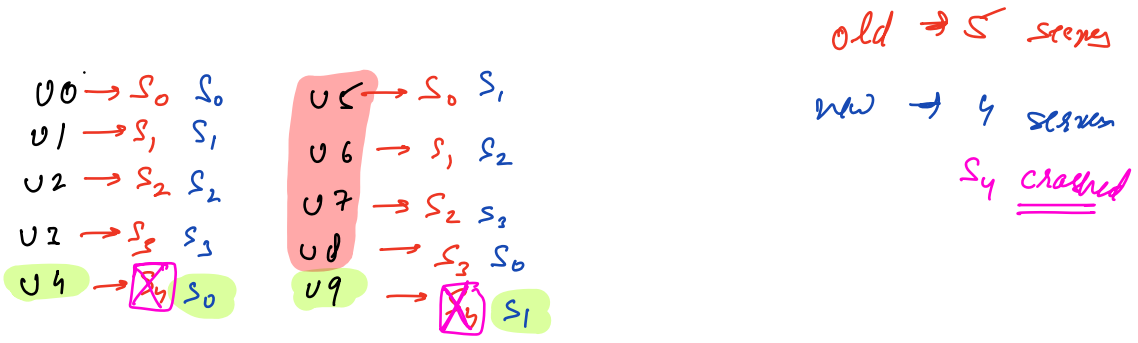
assume: central backup of all data at all times

with → some load of old servers gets moved to new server ✓

what all users got moved?

	new server	showing load
U5	0 → 5	
U6	1 → 0	
U7	2 → 1	
U8	3 → 2	
U9	4 → 3	

need less shuffling.

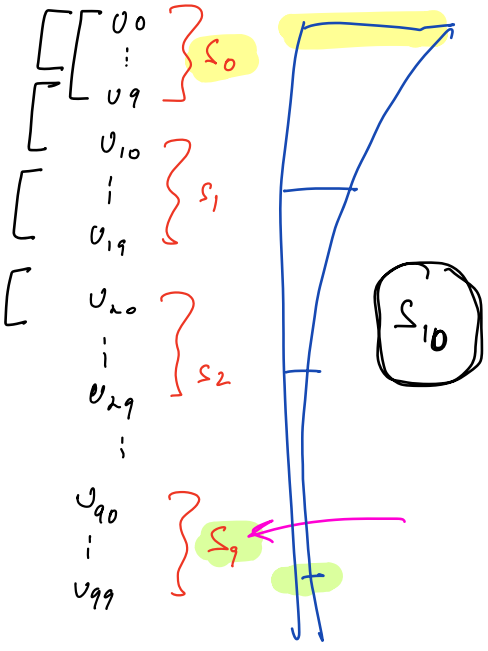


aim: data earlier stored at crashed server (S_4) gets remapped to other servers.

Some users get remapped for no reason

Needless data shuffling.

Range



Pros

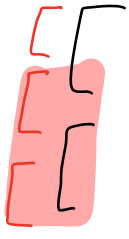
① Simple

Cons

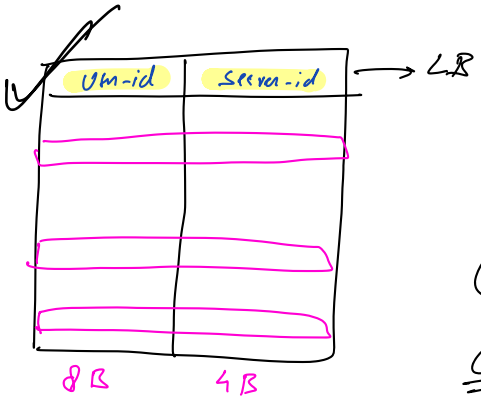
① Equal distribution?

~~to add new users, you need to add new servers~~

② Add / remove ✗



Mapping Table



12B for entry.

2B users

⇒ needed memory = 12 bytes * 2B ⇒ 24GB

RAM → fast

HDD → slow

Pros

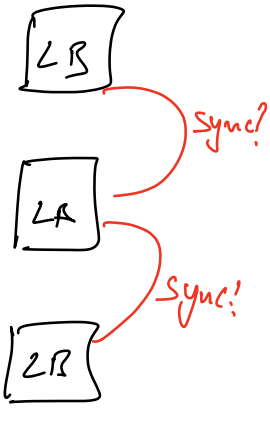
① equal distribution

② reasons 3

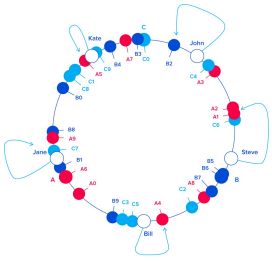
③ added.

Cons

long memory req.



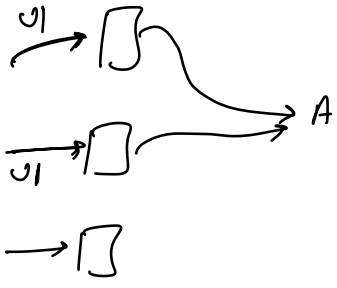
Sync → v.v.v.v v.difficult

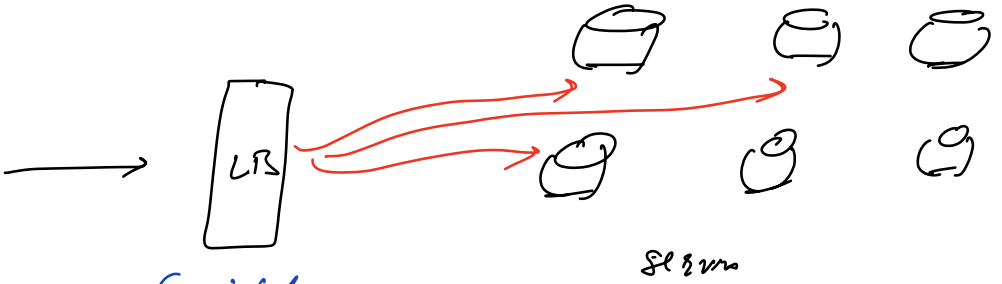


Consistent Hashing

Pros

1. Simple & fast ✓ ✓
2. Does needs v. less memory (RAM) ✓
3. Can be distributed (no sync needed) ✓
4. Easily add & remove servers ✓
5. Equal load distribution





Consistent Hashing.

Hash → function that takes in 1 value (input → anything) & returns another value (output → well defined range)

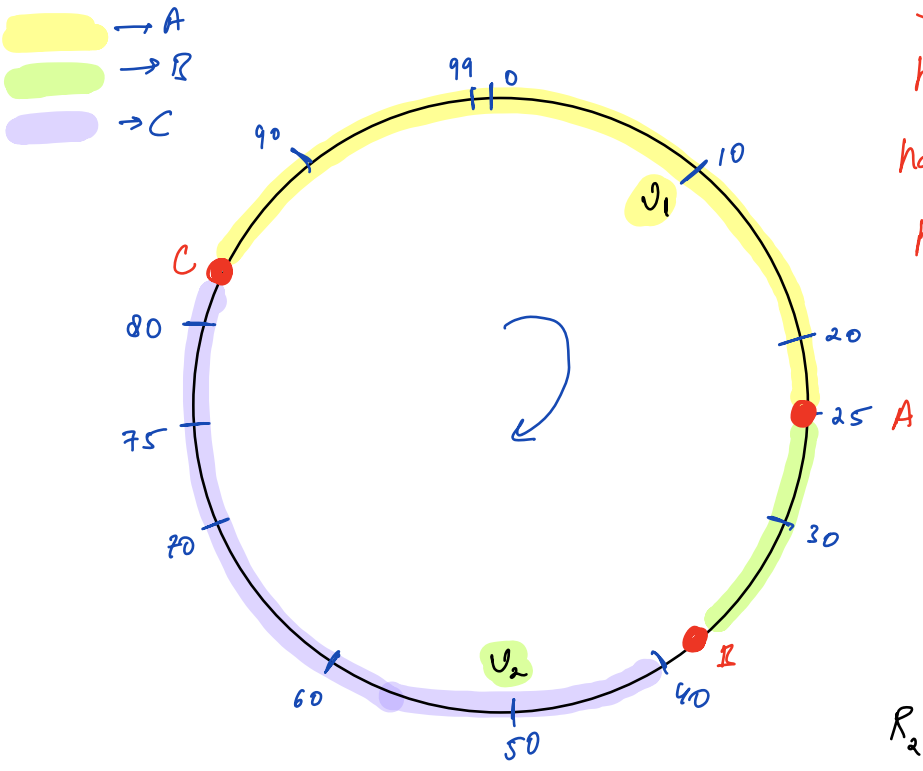
$h_1(\text{string}) \rightarrow \left(\sum_c \text{ascii}(c)\right) \% 12$

$h_1('abc') = (97+98+99)\%13 = 8$

Possible output ⇒ 0 – 12

$h_1(\text{—})$ ⋮ $h_{10}(\text{—})$ } 64 bit integer ($0 \rightarrow 10^{18}$ range)

$P_{collision} = \frac{1}{10^{18}}$



Servers ⇒ A B C

hash (IP_A) = 25

hash (IP_B) = 37

hash (IP_C) = 84

Requests

$R_1 \rightarrow v_1 \rightarrow A$

hash(user-id)

$h(v_1) \Rightarrow 10$

$R_2 \rightarrow v_1 \rightarrow A$

$h(v_1) = 10$

$R_3 \rightarrow v_2 \rightarrow C$

$h(v_2) = 50$

(1) visualize hash output as a ring

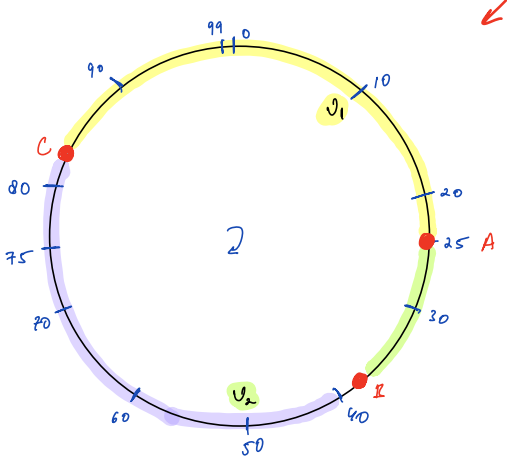
(2) hash each server & place it on ring

(3) given a req, find user-id from req. hash it.
find nearest server going clockwise.

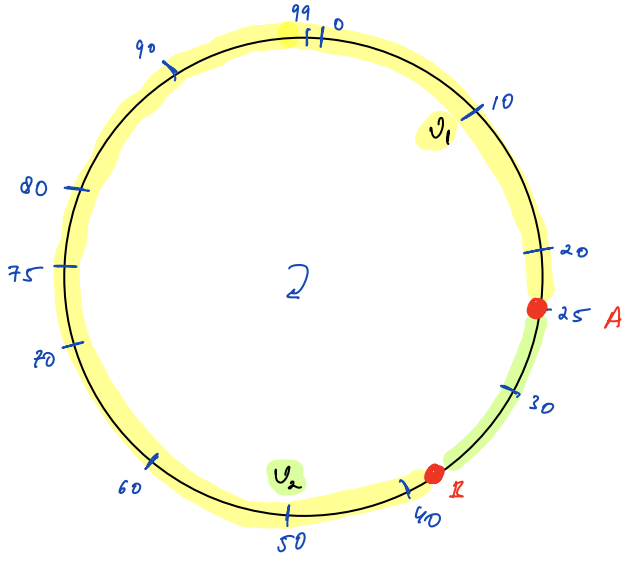
(1) User i always goes to same server (if server is up)

Server Crash

old



Servers → A
B
↳ Crashes



result →

All users earlier mapped to Server C now get re-mapped to server A.

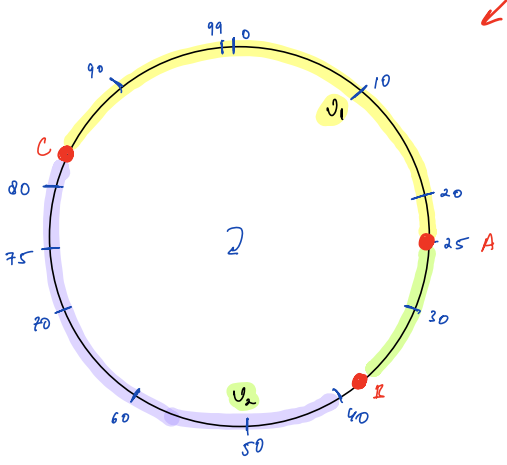
data move
backup C → A

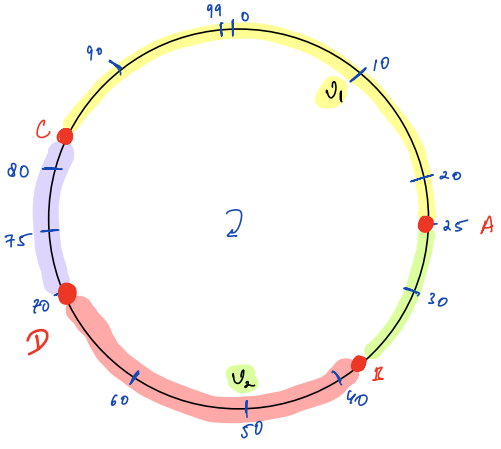
Un neccessary data movement? ✗

New Server Purchased

old

Servers → A
B
C





add Server D

hash(ip_D) ⇒ 70

Some users earlier mapped to C are now mapped to D

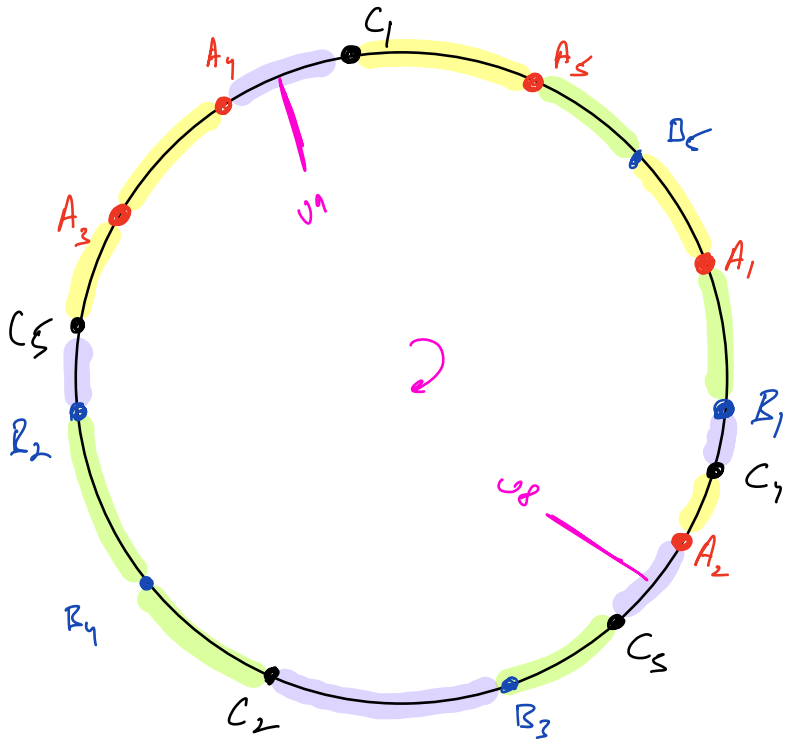
no unnecessary data shuffling.

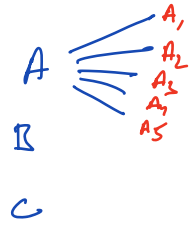
hash(1) → 

hash(u) → 

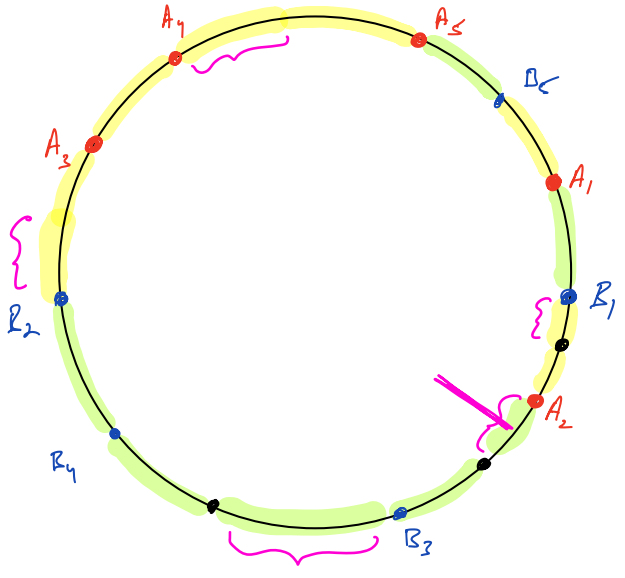
Magic ① Use K- hash functions to assign multiple locations to each server.

② User still gets hashed only 1 time.



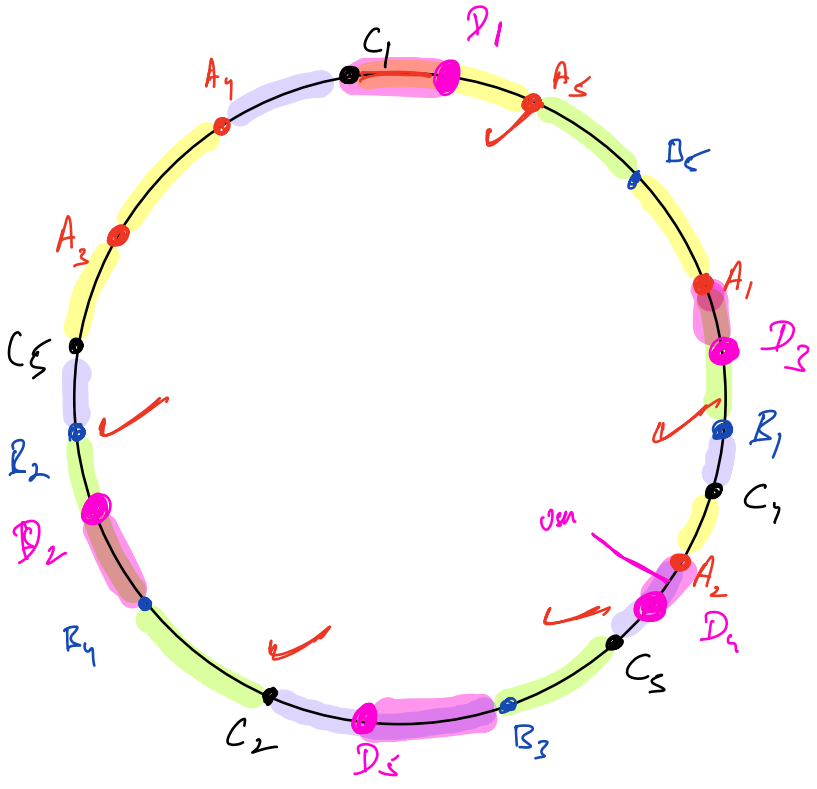


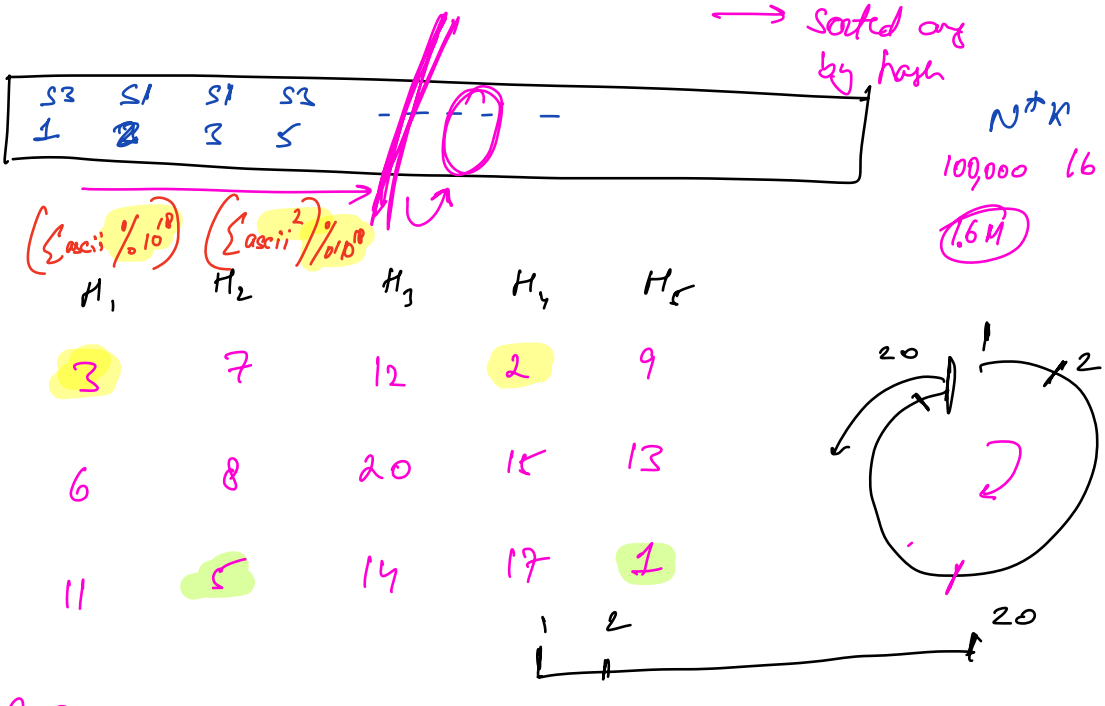
Server C Crashes



Server C's load was equally distributed across remaining servers.

Purchase Server D





$H_0(v_1) = 15$

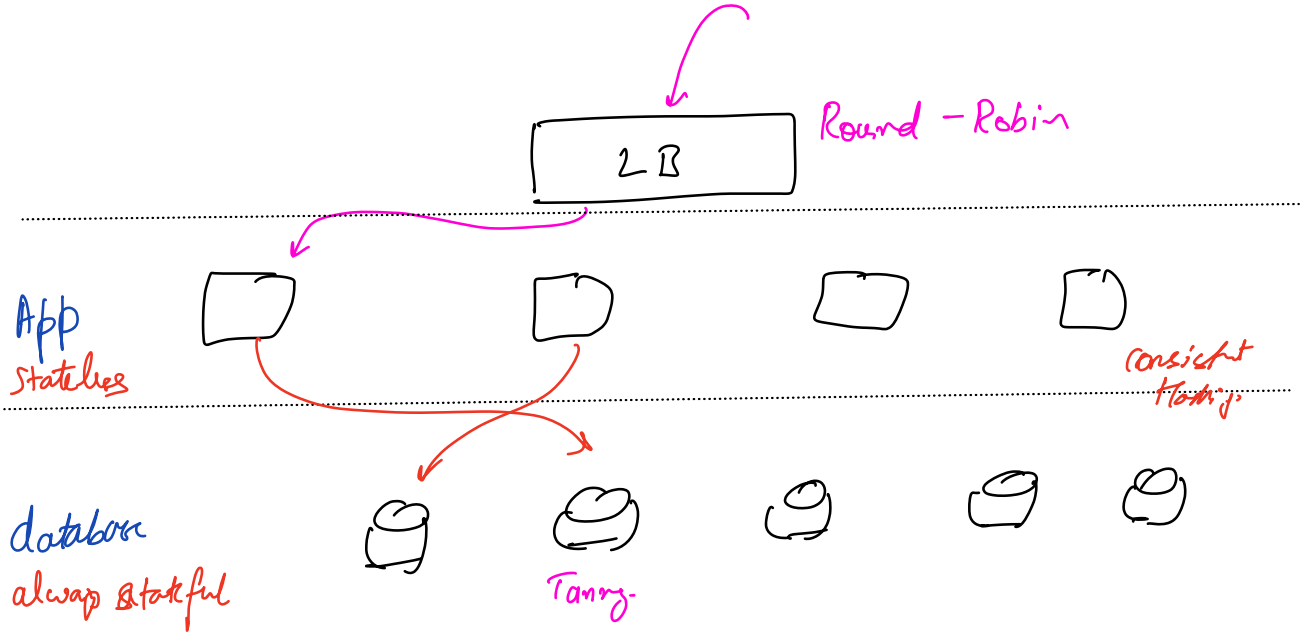
Stateless Servers

↳ any data that needs to be stored.

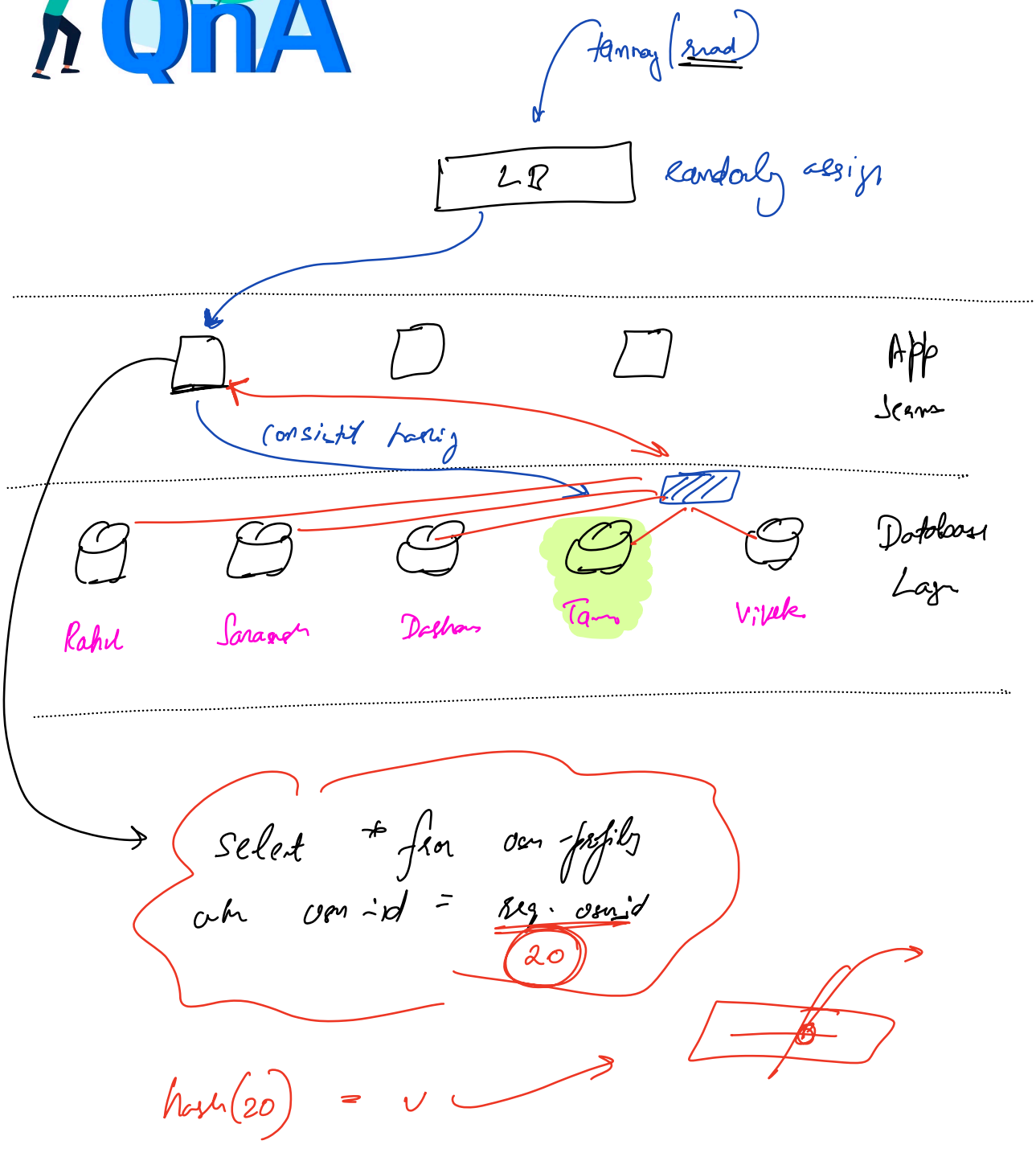
Both app & data live on each server.

Stateful servers

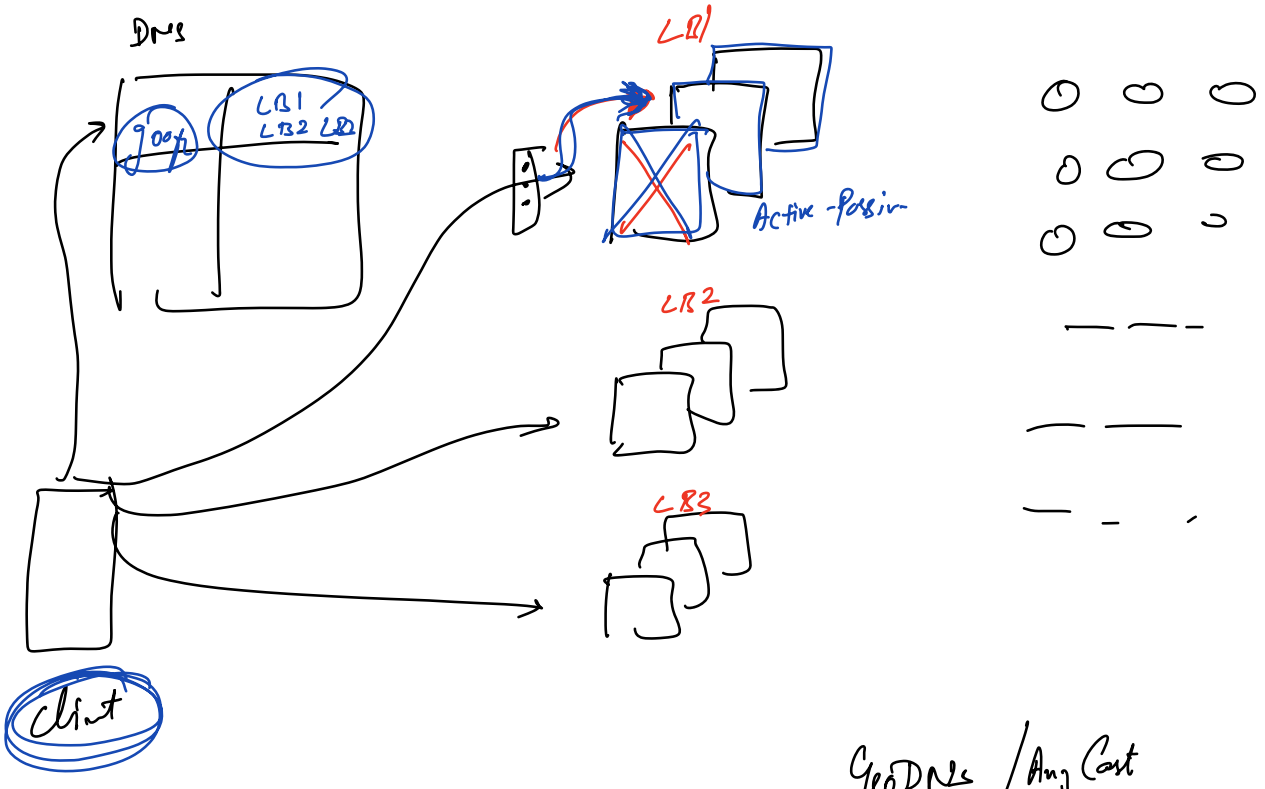
State → Consistent Hashing.




QnA


LB
App Servers
Database Layer
Rahul
Tarun
Vivek
select * from user_profile
where user_id = req.user_id
20
hash(20) = 0

DNS as a LB ⟶ Balancing Load
not for fault tolerance

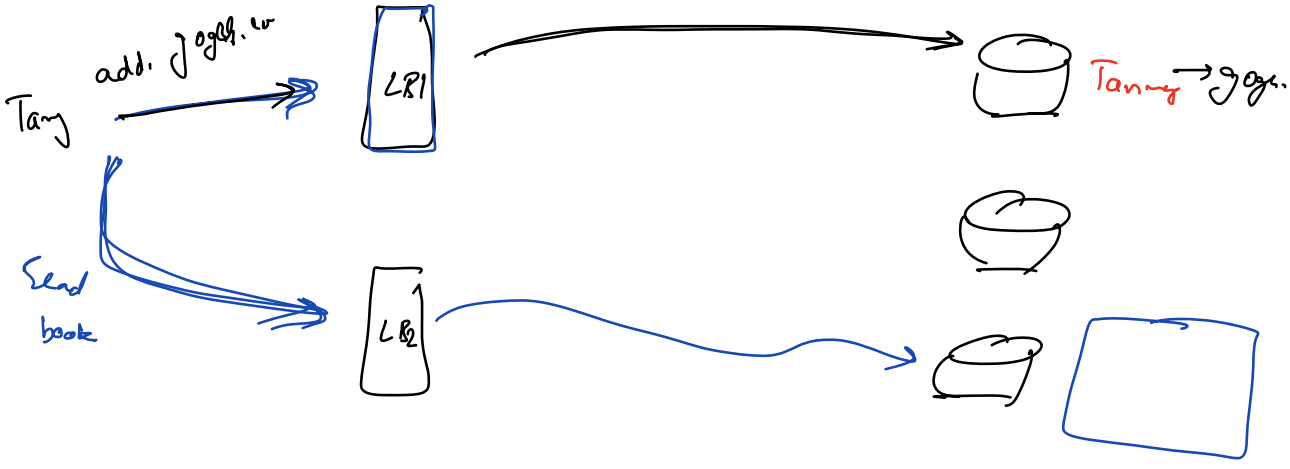


GeoDNS / Anycast

Prod - market fit

CTO

bias for action



$U_1 \longrightarrow LB_1 \longrightarrow S_1$

$U_1 \longrightarrow LB_2 \longrightarrow S_2$