Recap

Why of HLD

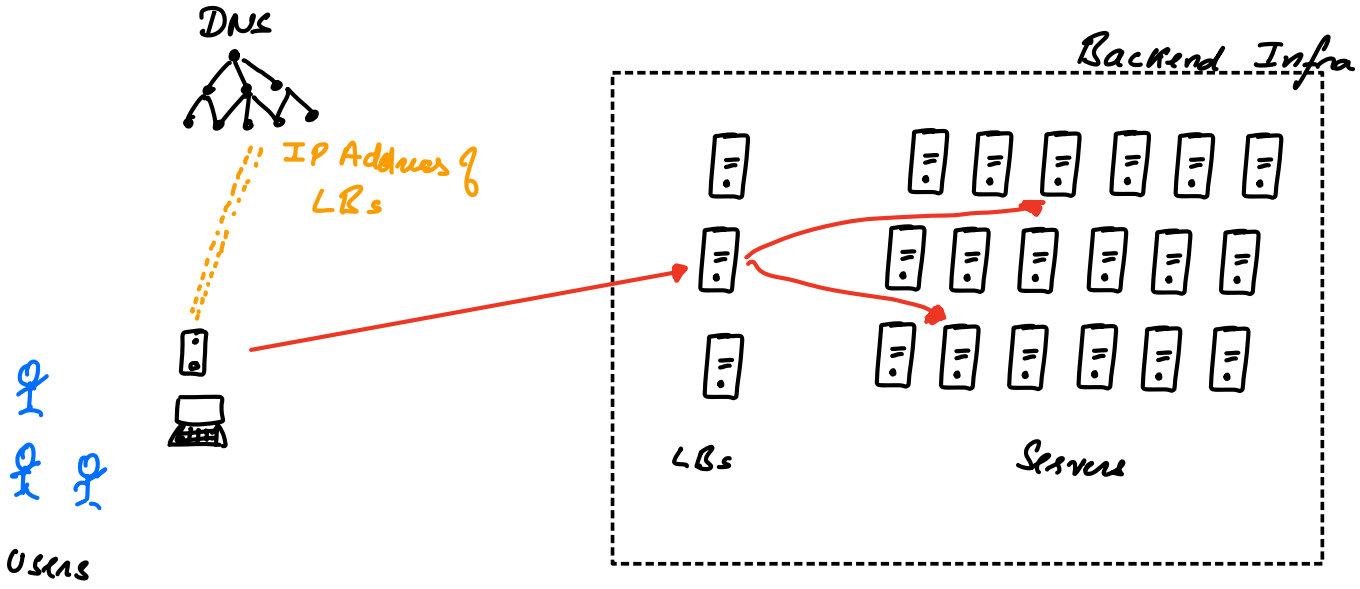
Case Study - Bookmarking website - Delicious

MVP - login / add / view

DNS
- bottleneck
- SPoF

Scaling
- Vertical
- Horizontal

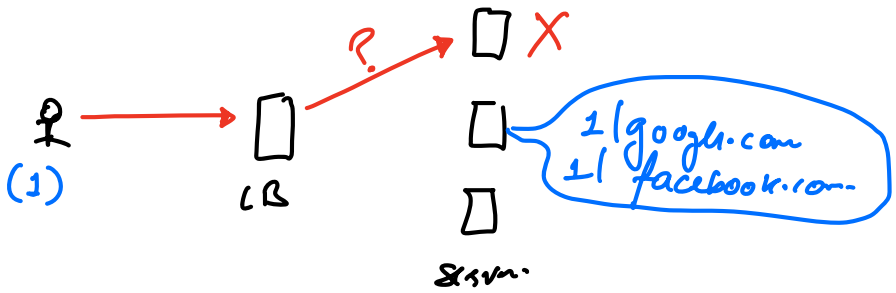
Load Balancer
- Unified view of backend
- Ensures equal load distribution
  - Healthcheck
  - Heartbeat
  - Multiple LBs - DNS assists



Q1 Which request gets forwarded to which server?

Routing Algo

Routing will depend on how the data is stored



Q2 How will you distribute the data? (what data is where?)

① Can we store all data on a single server? No!
(bookmarks)

∴ we need to split the data
Partitioning

Data Partitioning → splitting

① Vertical Partitioning – Database Normalization

② Horizontal Partitioning
- ↳ within the same server
- ↳ partition across multiple servers Sharding

DSA / DBMS / LLD

Users

id	name	phone	avatar-url	email	passhash+salt
1	A	—	—	—	—
2	B	—	—	—	—
3	C	—	—	—	—
4	D	—	—	—	—

Vertical (normalization)

accounts

id	email	passhash+salt
1	—	—
2	—	—
3	—	—
4	—	—

Profile

acc-id	name	phone	avatar-url
1	A	—	—
2	B	—	—
3	C	—	—
4	D	—	—

Horizontal (Distribution)

Users 1

id	name	phone	avatar-url	email	passhash+salt
1	A	—	—	—	—
4	D	—	—	—	—

Users 2

id	name	phone	avatar-url	email	passhash+salt
2	B	—	—	—	—
3	C	—	—	—	—

can be done on a single server too (to improve write throughput)

Server 1

Server 2

when done across servers = Sharding.

How to Distribute the data?

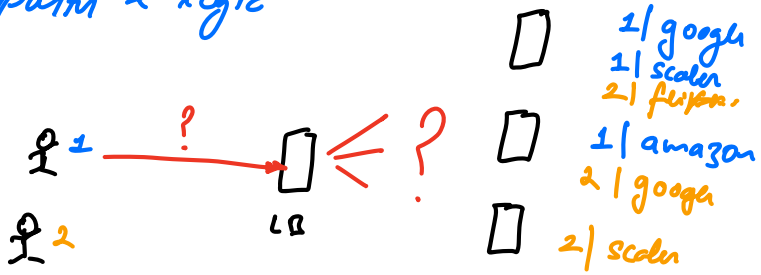
question – write it down aha moment – take note of that

① Can we distribute data randomly? No!

∴ there must be some path & logic

↓

1 user's data should be on only 1 server



fan-out node

Japanese hand held fan

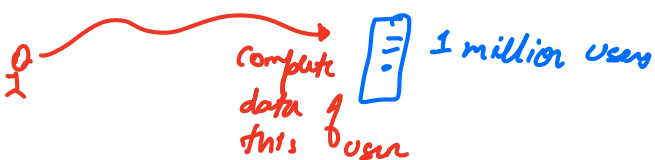
Sharding Key – user_id

Sharding always happens according to a key

Each user's data will be on only 1 server.

Note: 1 server can still store multiple users.

1 billion users



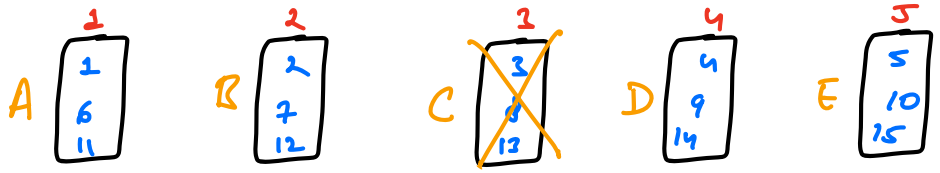
10000 servers

Task: come up with a good mapping from user_id → to server_id

this determines how data is distributed & this determines our Routing algo.

1. Is the data & load distributed evenly (skewness)
2. Is it easy to add & remove servers (minimize data movement)
3. How fast is it to calculate the mapping
4. Do we need to store anything to calculate this mapping.

~~Attempt~~ 1 Round Robin



even? ✓

fast? ✓

store anything? No ✓

add/remove servers ✗ bad

N = ~~5~~ 4 → via healthcheck

```
void handleRequest(req) {
    user_id = req.user_id
    server_id = (user_id % N) + 1
    forward to server_id
}
```

pretty much all users have been remapped to some other servers.

a lot of data needs to be moved.

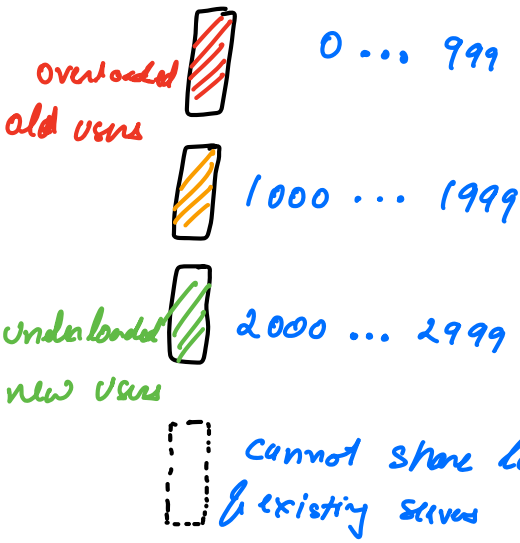
How is the data moved? Who does that? How do we take backups?

later class — Magic Faing (Son Pari)

↳ Database Orchestration

later class

~~Attempt~~ 2 — Range Based



~~1~~ even — each server has 1000 users

2 fast — [user id/1000]

3 store anything no

~~4~~ add/remove servers

- adding new user mandates adding more servers bad!
- old users generate more load than new users

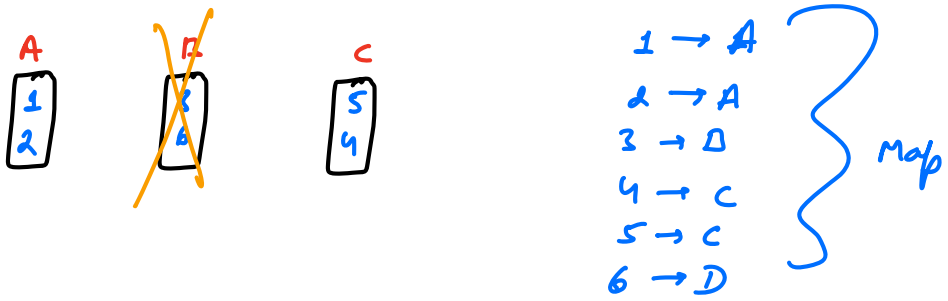
~~Attempt~~ 3 — Hash Map Based

> the thing about chance is that it is fair — Jovien

Randomly assign user to servers but

Keep track of the assignment via a Hash Table

HashMap < user-id, server-id > map



① fast ② even ③ add/remove

loop through map

find all users that are part of

crashed server & remap them.

~~④~~ store? Yes!

Rad

① Can we store in RAM? No — restarts?

Volatile

∴ we must store on disk — v.v. slow

Random read in RAM ≒ 10ns (CL latency) (DDR5 5600 MHz CL 28)

1 million x

Random read in HDD ≒ 10ms (7200 RPM Magdisk)

≒ 2ms (Samsung 980 Pro gen 3)

RAM is 1 million x faster than disk

(for random read of 1 Byte)

② How long is this mapping?

Google has 5B users

≈ 100,000 servers

8b 4b

user id → server id

12 bytes / user entry

$12 \frac{\text{bytes}}{\text{user}} * 5 \times 10^9 \text{ users} = 60 \text{ Billion bytes}$

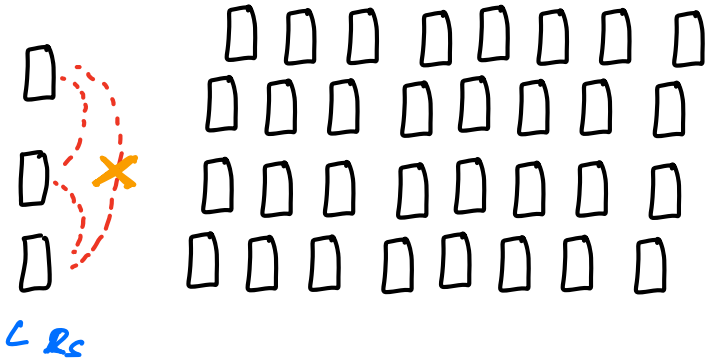
$= 60 \text{ GB}$

③ what if there are multiple LBs?

is it okay to have diff mapping in each LB? ✗

mappings should be in sync

n/w overhead



Keeping lots of data in sync — fast, consistently, distributed — IMPOSSIBLE! (CAP)

Solution — Consistent Hashing

Relies on Randomness

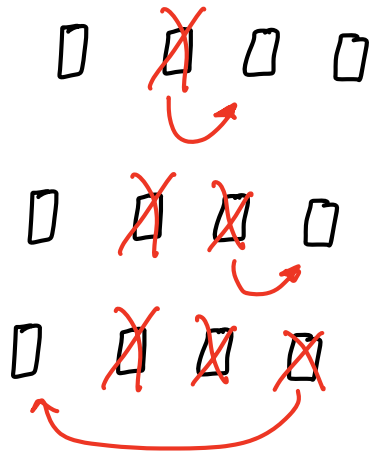
① even load distribution

② fast to compute

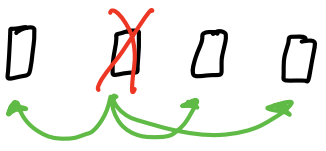
③ won't need to store & sync any mapping

④ adding / removing servers will be easy

⑤ if a server crashes its load will be handled by all remaining servers equally!

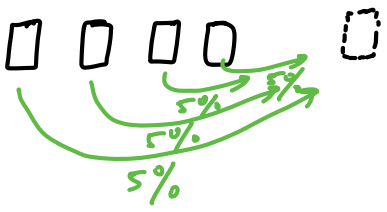


Cascading failure



No cascading failure

⑥ if you add a new server it will automatically relieve the load of all other servers equally.



Hash Functions

↳ input → output
domain range

$f(x) = x^2 + 3$

— deterministic
no matter how many times you call it it always produces same result.

a function which has a finite range & behaves pseudo-randomly is a hash function.
difficult to predict what the output will be

Mapping (inp → out) is random but deterministic.

hash(x)
↳ represent x in digits
↳ split it in middle
↳ reverse each part
↳ cube it
↳ join
↳ mod M
% (10^9+7)

1 2 2 4 5 6 7
↓
1 2 3 4 | 5 6 7
4 3 2 1 | 7 6 5

0 0 6 7 7 5 6 8 1 6 1 | 4 4 7 6 9 7 1 2 5

0 0 6 7 7 5 6 8 1 6 1 4 4 7 6 9 7 1 2 5

b/w 0 ... (M-1)

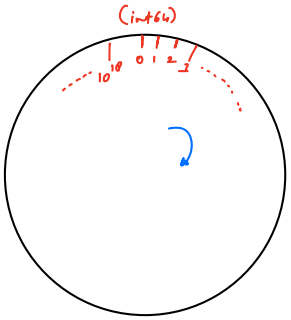
family of hash functions

Consistent Hashing

k hash functions for servers
1 hash function for request (user_id)

} the range for all these is the same 0 ... 10^{18}

int64

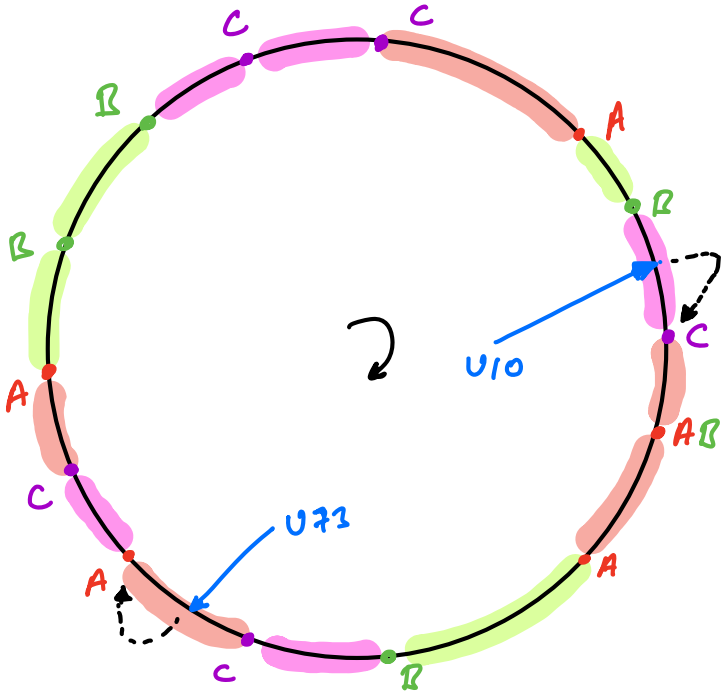


Algo

① for each server
pass it through all K hashes
each hash value will be a spot for this server on the ring
virtual spots

② when a req comes
extract user_id (sharding key) & pass it through the hash & note the spot on ring

③ forward req to the nearest server moving clockwise



N = 3 servers A B C
K = 5 hash functions

$h_1(A) = 10$ $h_2(A) = 60$
$h_3(A) = 30$ $h_4(A) = 70$
$h_5(A) = 35$

$h_1(B) = 15$ $h_2(B) = 80$
$h_3(B) = 20$ $h_4(B) = 85$
$h_5(B) = 50$

$h_1(C) = 25$ $h_2(C) = 90$
$h_3(C) = 55$ $h_4(C) = 5$
$h_5(C) = 65$

✓ even ∵ of randomness

as K increases it becomes more even

K = 16 or K = 32

✓ fast

✓ Store anything? NO

$h_1(A) = 10$ $h_2(A) = 60$
$h_3(A) = 30$ $h_4(A) = 70$
$h_5(A) = 35$

$h_1(B) = 15$ $h_2(B) = 80$
$h_3(B) = 20$ $h_4(B) = 85$
$h_5(B) = 50$

$h_1(C) = 25$ $h_2(C) = 90$
$h_3(C) = 55$ $h_4(C) = 5$
$h_5(C) = 65$

array = [ ]

Servers = [...] // healthcheck →

for each server

serverid → each hash → consistent hash function
→ value
→ store value in array

Sort array by hash

upperbound

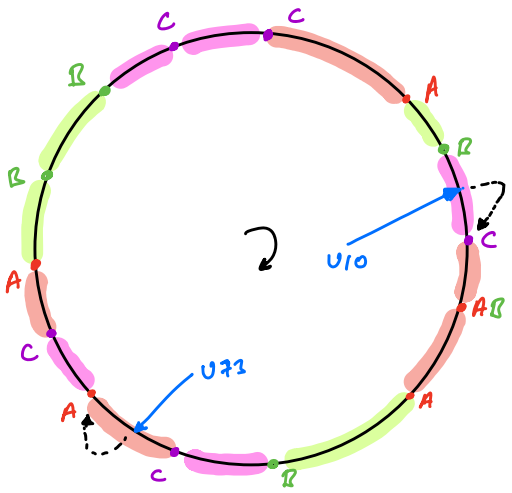
[C A B C A B A ....] → sorted

C	A	B	C	A	B	A
5	10	15	25	30	70	85

N*K

binary search

req → user-id → h_u(user id) = 28



$\lg_2(N*K)$

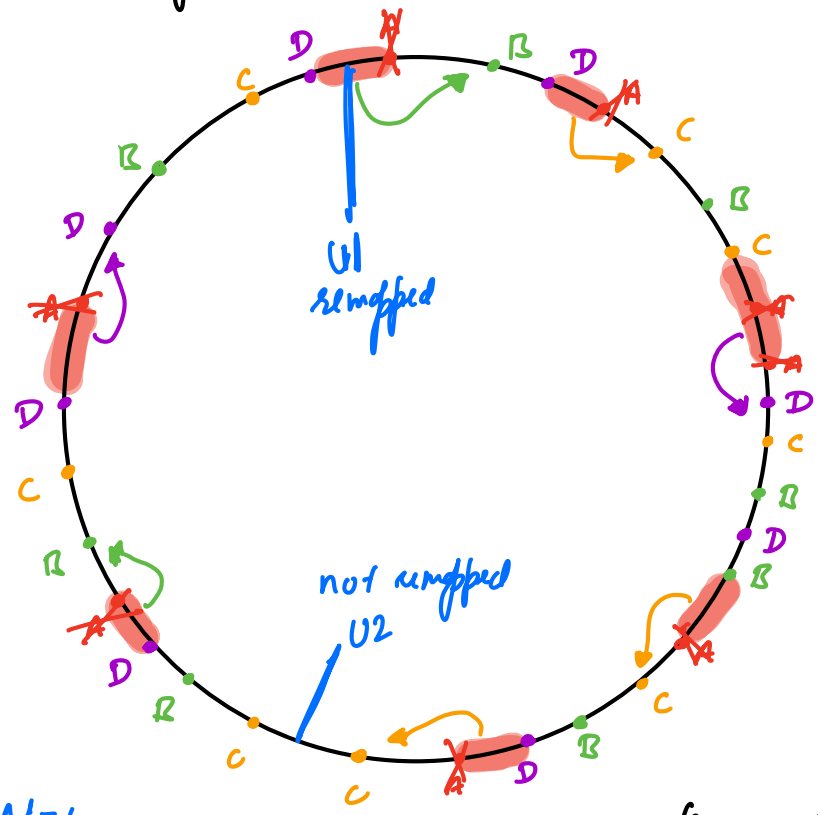
N = 100,000
K = 22
} Google

Time $\lg_2(3,200,000)$ ≈ 21 ops ≈ few μs

$2^{10} \approx 10^3$
$2^{20} \approx 10^6$
$2 \cdot 2 \times 10^6$

Storage 3,200,000 entries if each entry is 12B (server id + hash) = 36MB RAM

What if server crashes?



N = 4

K = 8

Server A dies!

① LB will detect it (all of them)

② Recompute the array

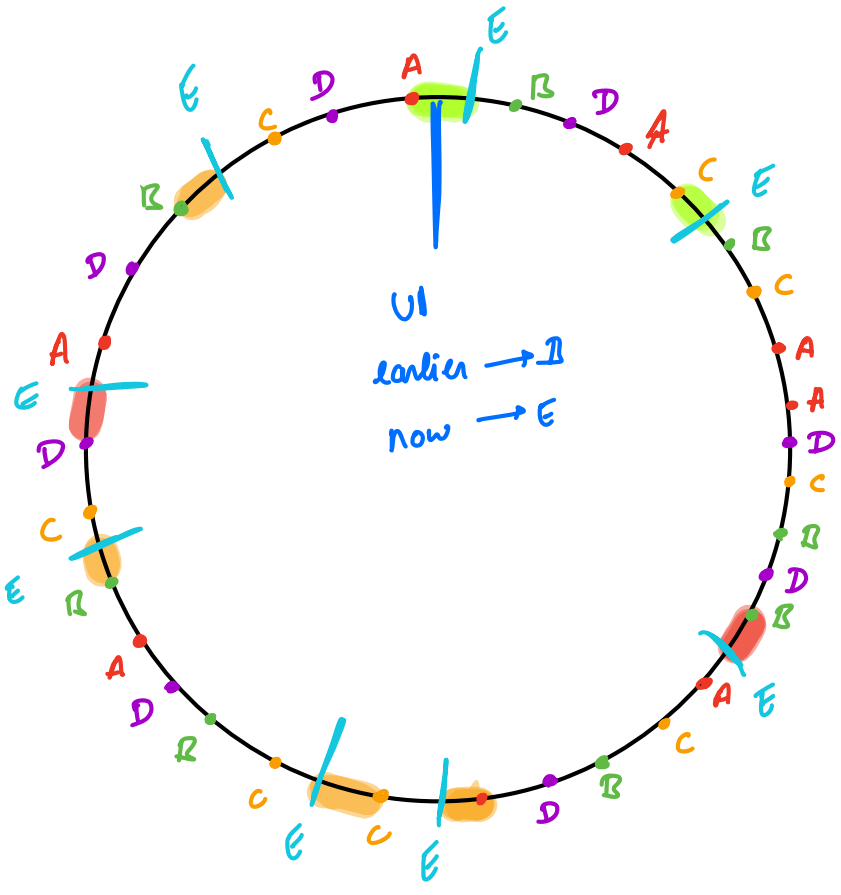
③ Continue working as if nothing happened!!

(④ Note : Data Movement Magic Fairy)

the load of server A was taken over by servers

R C D

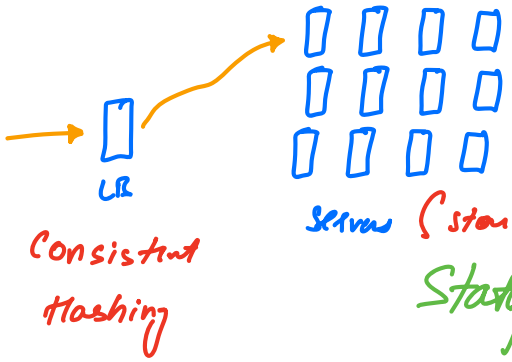
If a new server is added



add server E

It will take over some load from the all remaining servers

Consistent Hashing → used when distributing data.



State = data
Stateful → containing data

Modern System Design

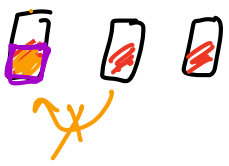
Push data away from app servers into DB layer.

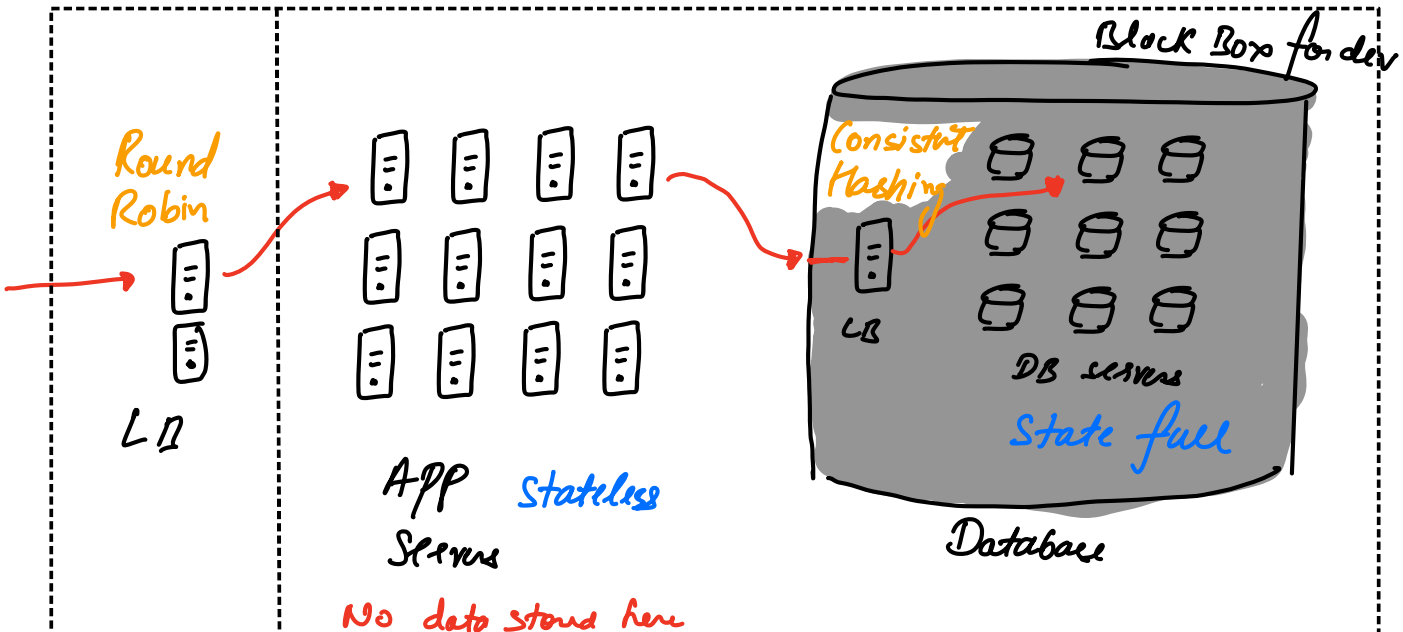
why?

if the same server has both app logic & data

① tightly coupled — cannot scale independently

② code deployment — temporary data outage!

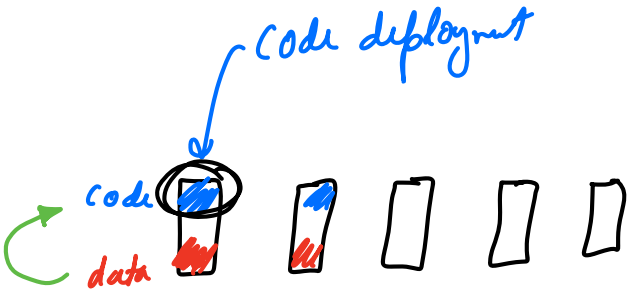




only app logic

Untrusted n/w

Trusted VPC



CI/CD

↳ @ Microsoft

≈ 4000 code deploys/day

Next class — Caching

1. Cache Aside — types
2. Case studies

Please ensure that you've done assignments — 5 MCQs