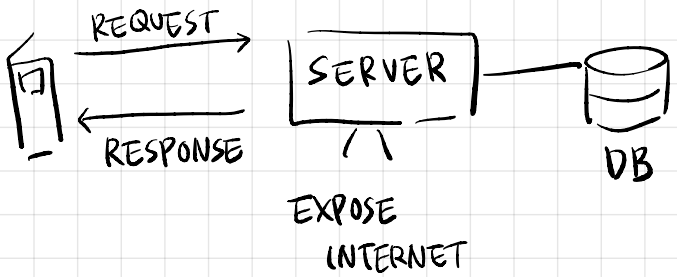
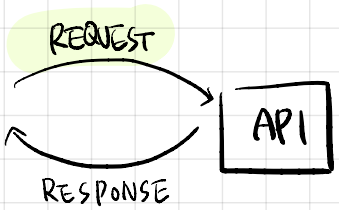
[System Design Basics : Horizontal vs. Vertical Scaling]



what should we do when power loss?



How to handle more requests?

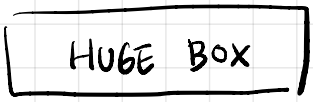
= VERTICAL SCALING

① Buy bigger machine

② Buy more machines

= HORIZONTAL SCALING

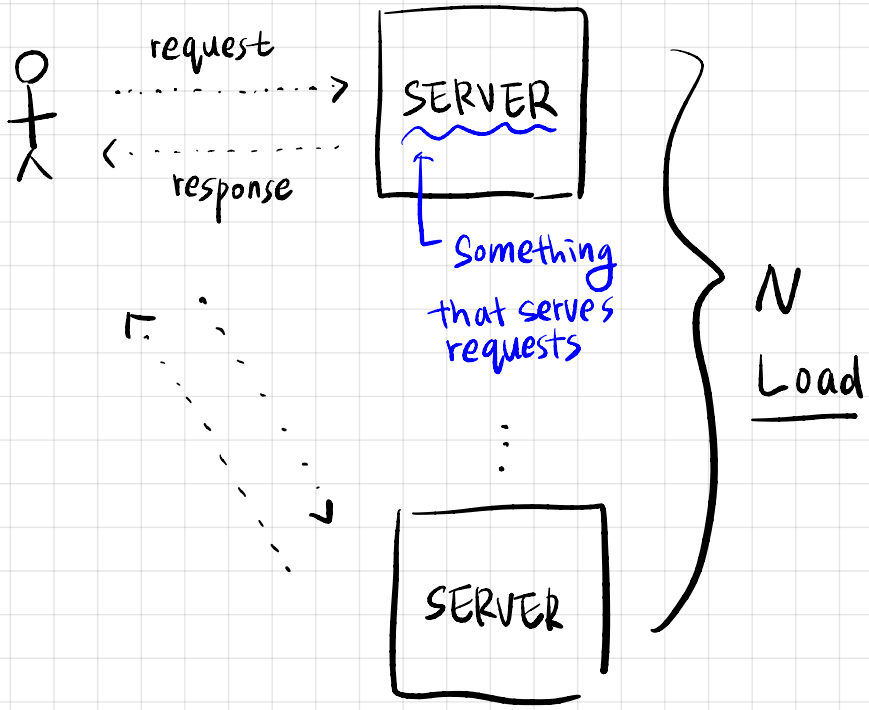
SCALABILITY

HORIZONTAL	VERTICAL
	
① Load balancing required	① Load balancing not required
② Resilient	② Single point of failure
③ Network calls (= Remote Procedure Calls) <slow>	③ Interprocess Communication <fast>
④ Data inconsistency	④ Data consistency
⑤ Scales well as users increase	⑤ Hardware limit

Both used in reality!

[What is LOAD BALANCING?]

LOAD BALANCING is relevant to Consistent Hashing



Load Balancing (try to balance loads evenly for N servers)

How? → Consistent Hashing

[Consistent Hashing] ---> each server allocated uniform loads.

Request ID ··· 0 ~ M-1

$h(r_i) \rightarrow m_i \,\%\, n$

h: hash

s: server

r: request

uniformly random

s_0

s_1

⋮

s_{n-1}

ex) r_1 : 10 (request)

$h(r_1) : 3$

n: 4 (# servers)

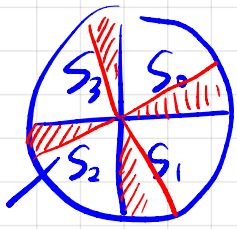
$h(r_1) \,\%\, n = 3$

$r_1 \rightarrow s_3$

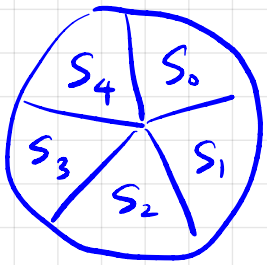
What if a new server is added?

ex)

20 Requests, 4 servers ⟶ 20 requests, 5 servers?



5 requests



Make sure $4 \times$  sums up to 4

and minimize changes to hashing!

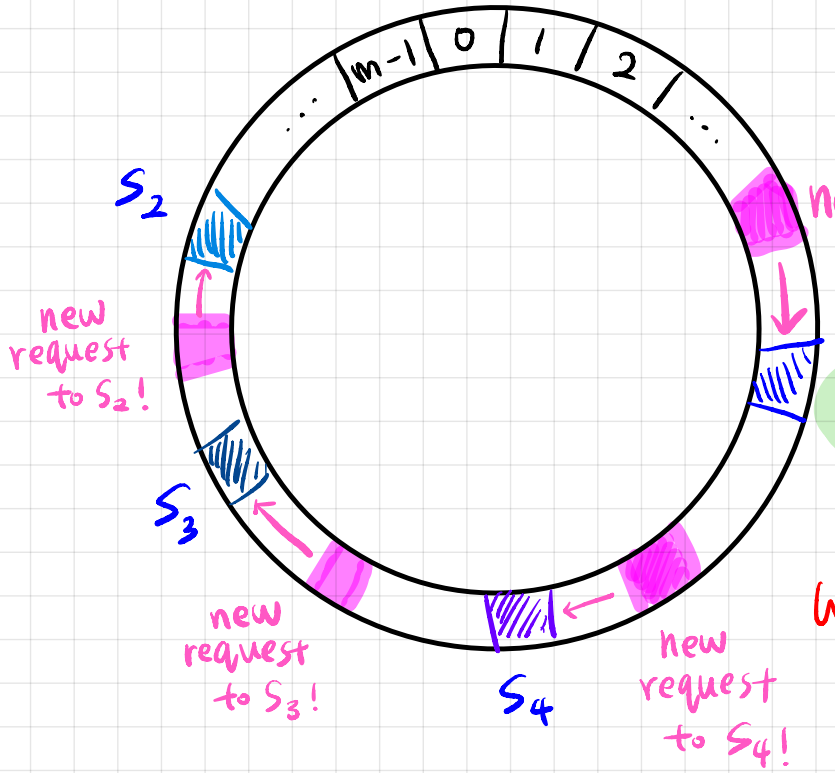
⟿ This is where consistent hashing comes in!

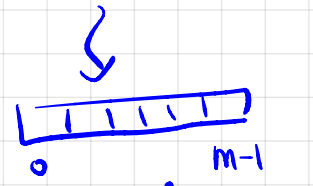
[What is Consistent Hashing and Where is it used?]

Still hash requests according to their IDs. Request ID → $h(r_{ID})$

0 ... m-1

Ring of hash





Instead of array...

use ring!

When request comes in, we go clock wise!

<Previously... 4 servers>

S	SID → hash → h	→ %M
S_1	0 → h(0)	h(0)%M
S_2	1 → h(1)	h(1)%M
S_3	2 → h(2)	h(2)%M
S_4	3 → h(3)	h(3)%M

distances between hashed numbers are uniform!

⇓

uniform load!

1/N

ex) SID = 0

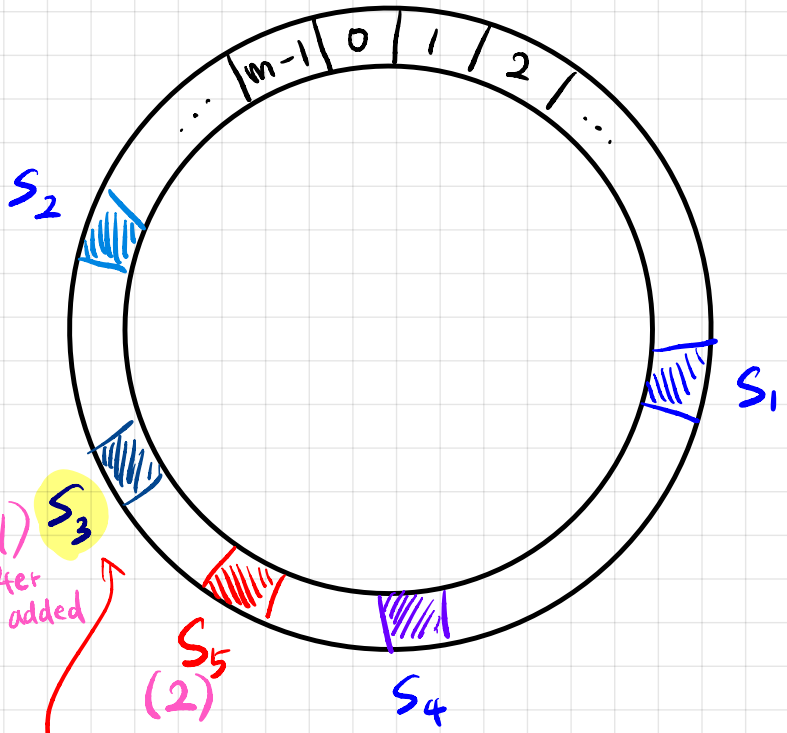
h(0) = 49

M = 30

h(0)%M = 19

What if a new server is added? Add S_5!

Ring of hash



(load)

$(3 \rightarrow 1)$

before

after S_5 added

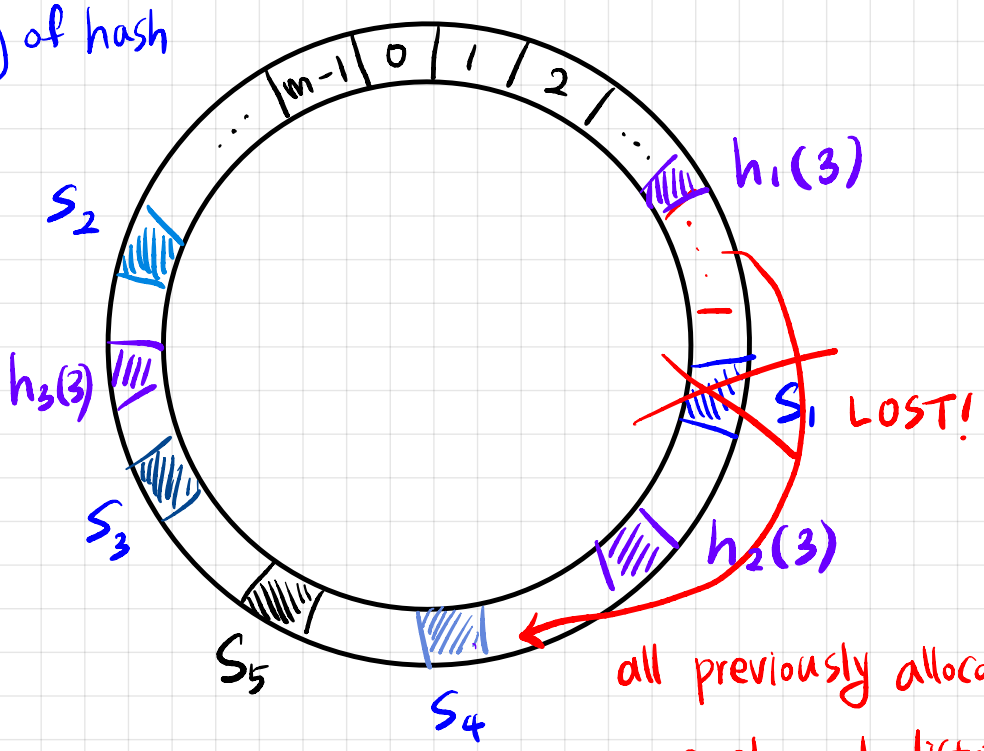
only S_3 is affected from adding S_5!

<Previously... 4 servers>

S	S ID	$\xrightarrow{\text{hash}}$ h	$\rightarrow$ $\%M$
S_1	0	$h(0)$	$h(0)\%M$
S_2	1	$h(1)$	$h(1)\%M$
S_3	2	$h(2)$	$h(2)\%M$
S_4	3	$h(3)$	$h(3)\%M$
S_5	4	$h(4)$	$h(4)\%M$

What if an existing server is removed? S_1 Lost!

Ring of hash



all previously allocated to S_1 going to S_4

→ skewed distribution :(

↓ WHAT TO DO?

Use multiple hash functions!

$h_1, h_2, \cdots$

<Previously... 4 servers>

S	S ID	—hash→	h_1	→ %M
S_1	0		$h_1(0)$	$h(0)\%M$
S_2	1		$h_1(1)$	$h(1)\%M$
S_3	2		$h_1(2)$	$h(2)\%M$
S_4	3		$h_1(3)$	$h(3)\%M$
S_5	4		$h_1(4)$	$h(4)\%M$

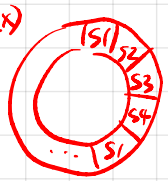
<Multiple hash functions>

ex) K hash functions

→ each server has k points!

ex) $h_1(3)$, $h_2(3)$, $h_3(3)$

ex)

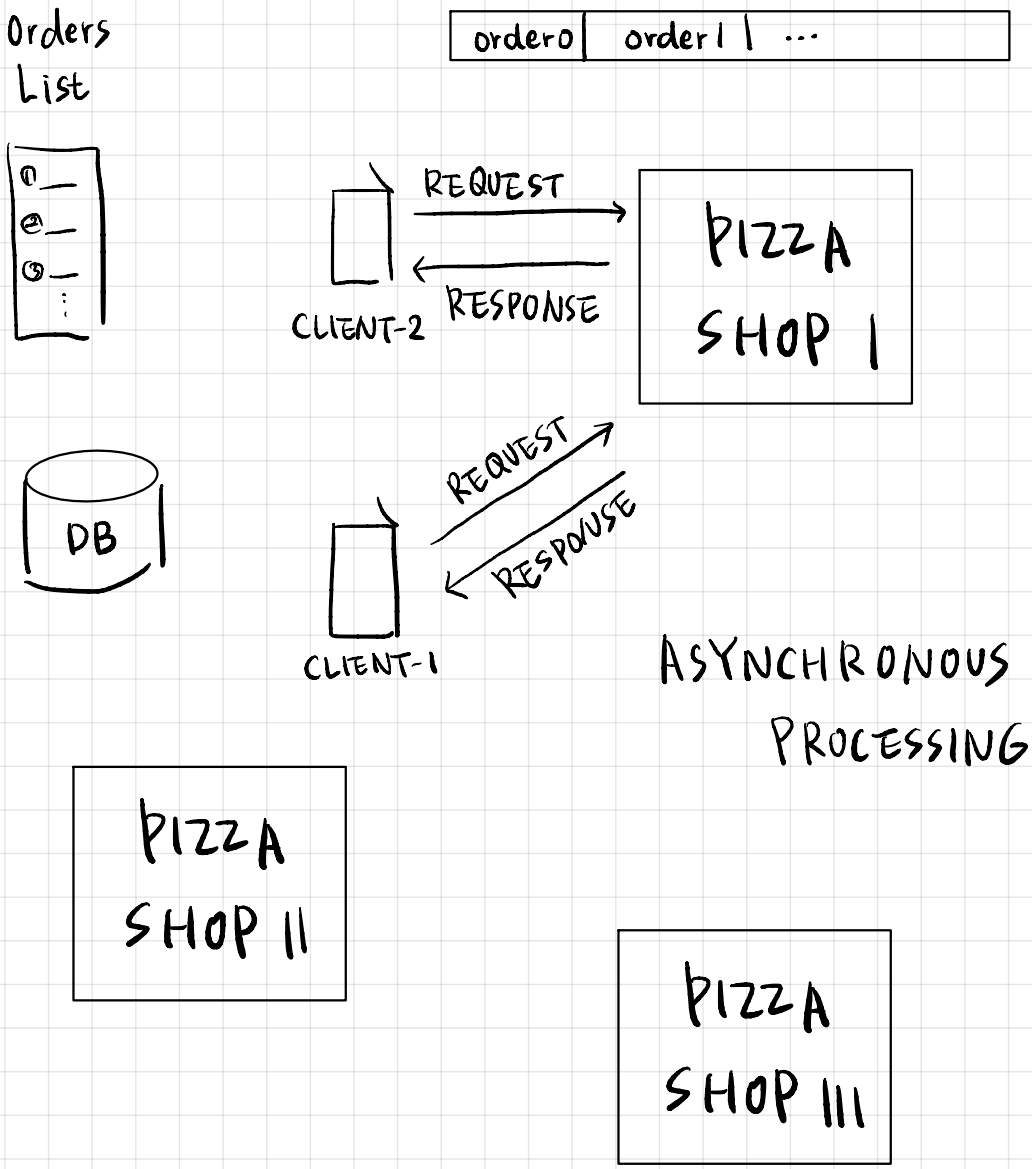


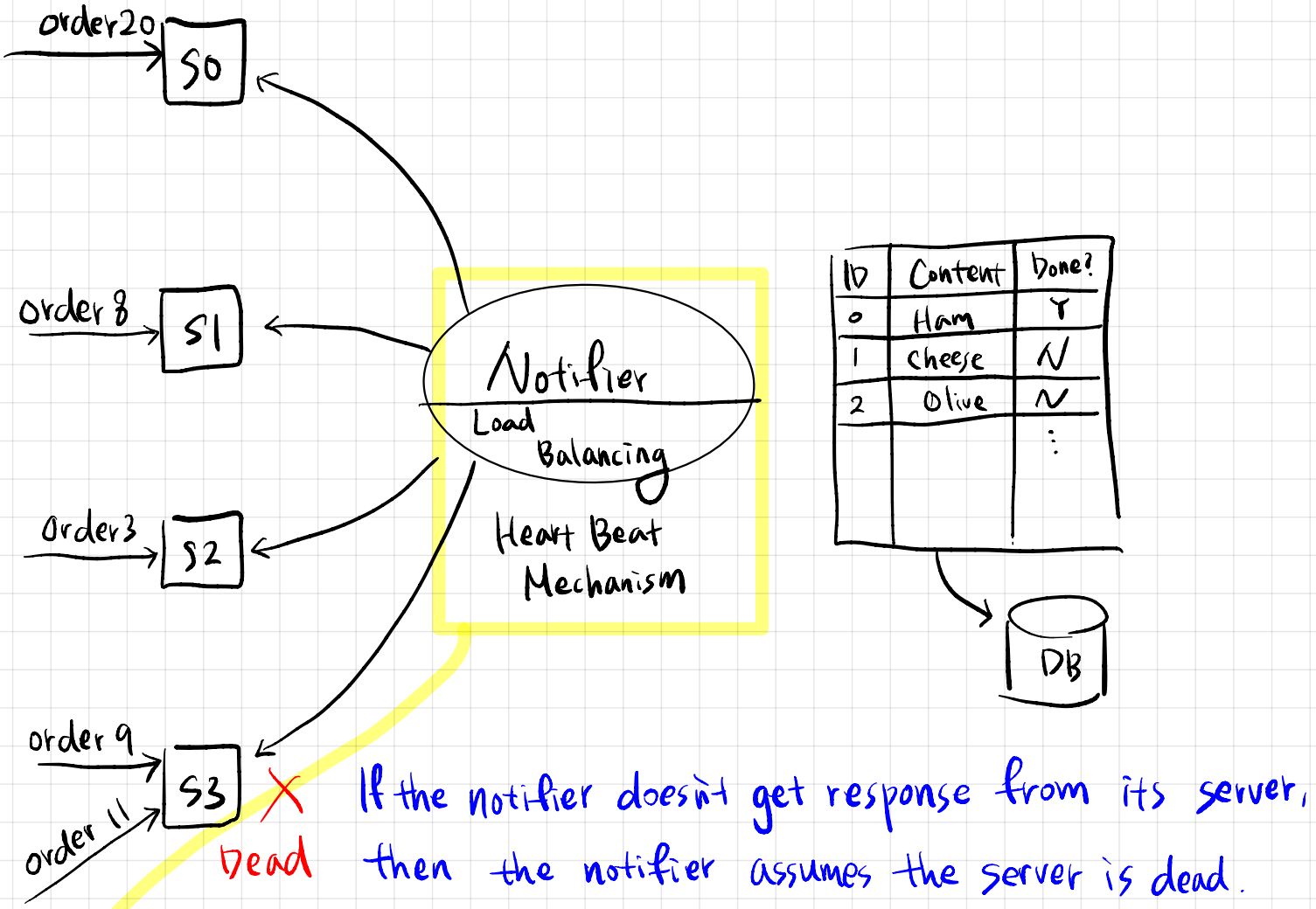
↳ if $k = \log(m)$,

almost remove all possibility of skewed distribution.

(= one server getting extreme load)

[What is a MESSAGE QUEUE and where is it used?]





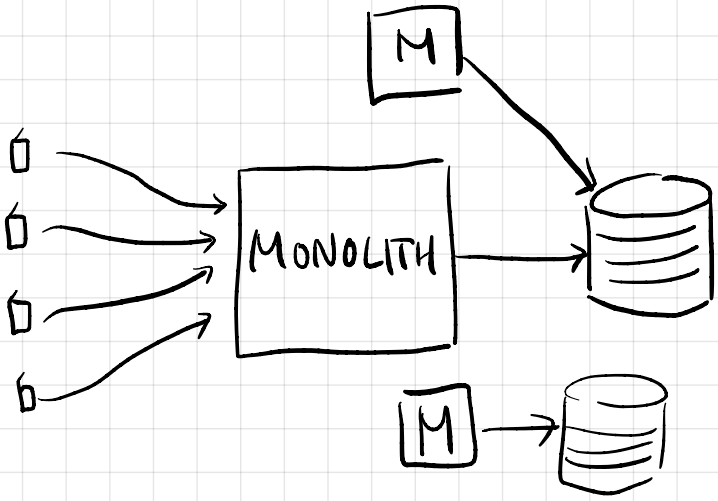
If the notifier doesn't get response from its server, then the notifier assumes the server is dead.

MESSAGE/TASK QUEUE) ex) RABBIT MQ, ZERO MQ, JMS, ...

: Takes tasks, persists them, assigns them to the correct server, waits for them to complete, if it's taking too long for the server to give acknowledgement, it considers the server to be dead, and assigns it to the next server.

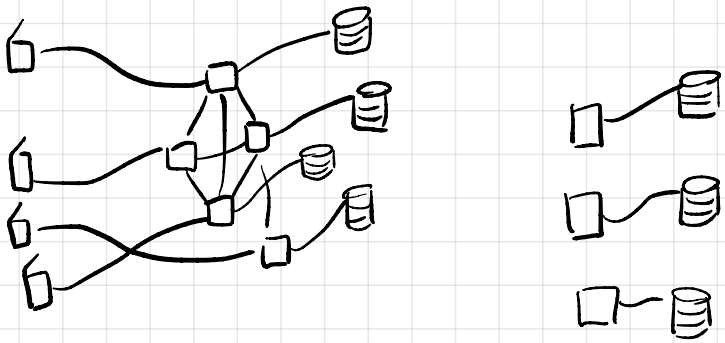
[What is a MICROSERVICE ARCHITECTURE and what are its advantages?]

MONOLITH vs. MICROSERVICE

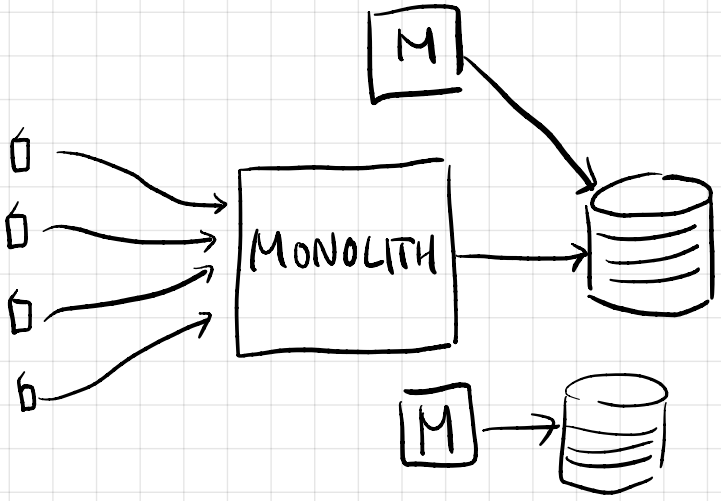


← doesn't mean "one huge"

[MONOLITH]



[MICROSERVICE]



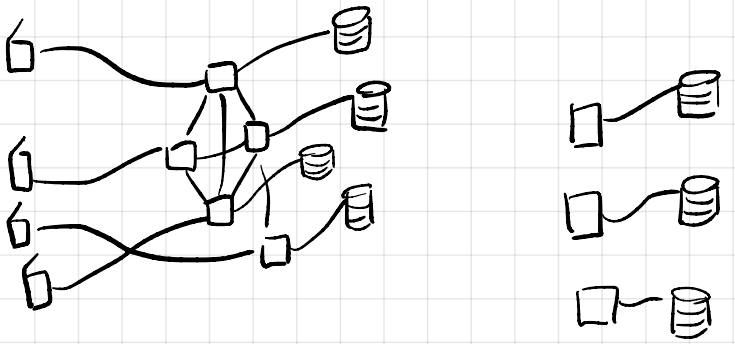
[MONOLITH]

Pros

- Good for small teams (Cohesive teams)
- Less complex (no need to worry about how to break into pieces)
- Less Duplicates (ex. code for setting systems up)
- Faster (Remote Procedure call X, Same box O, local call O)

Cons

- More context required (If a new member is added to your team, the new member has to go through the entire system and understand it.)
- Complicated Deployments (everything is touching everything, less decoupling)
- Single Point Failure (Too much responsibility in each server.)



[MICROSERVICE]

- Scalability
- Easier for new team members
- Working in Parallel
- Easier to reason out (ex. easier to tell what's used frequently)
- Not easy to design

ex) [S1] → [S2]

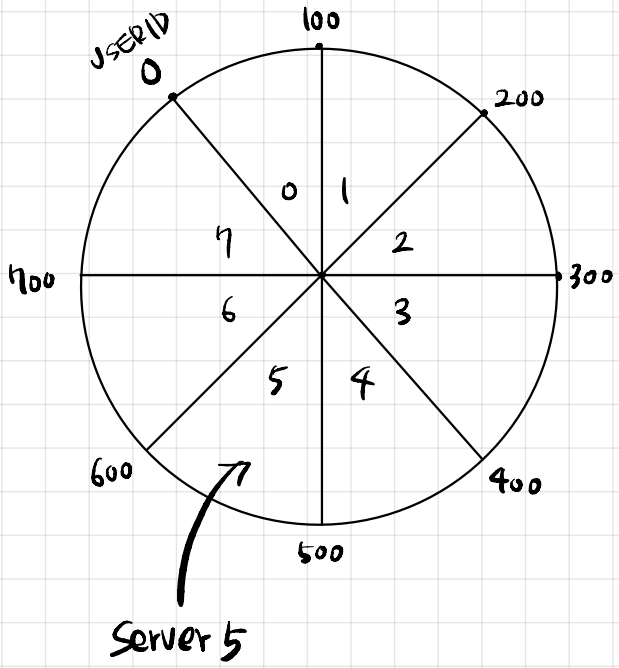
If S1 is only talking to S2, ...> [S]

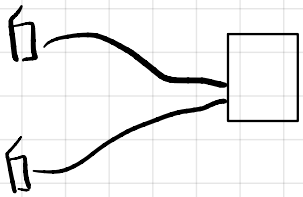
it probably means it should be in one single service

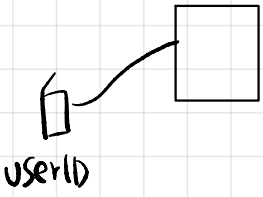
~> In interview, you may need to justify

Why you chose monolith or micro service architecture.

[What is DATABASE SHARDING?]







[Horizontal Partitioning] ⊃ SHARDING

Use key to partition data into pieces

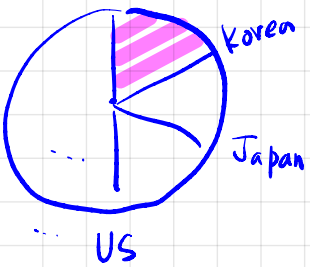
SHARDING

: Taking one attribute (ex. UserID) in the data

and partitioning the data so that each server gets one trunk

- Consistency
- Availability

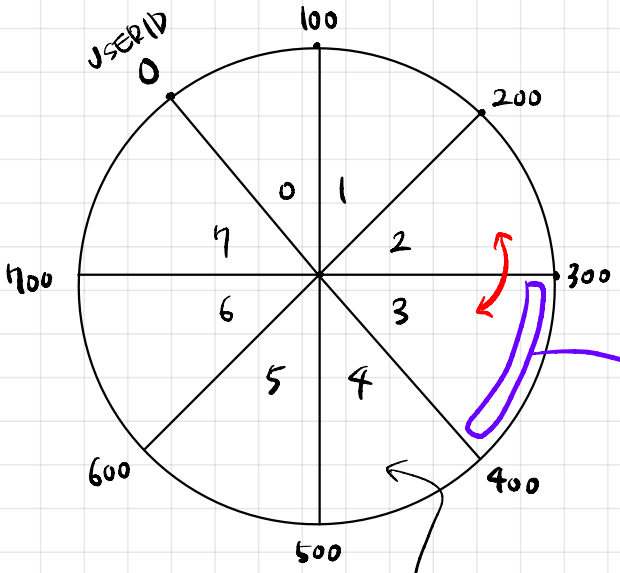
→ ex) TINDER



Give me users who are located in Korea

↓

Read through

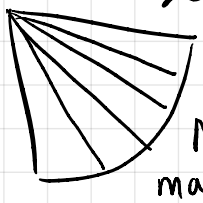


Problems:

① JOINS

② FIXED NUMBER OF SHARDS
(shards are inflexible)

② Solved!
Use hierarchical sharding!



Manager maps to mini slice!

What if electricity loss?

MASTER SLAVE Architecture! ← Good single point failure tolerance

Create index on the SHARDS!

↑ Multiple slaves which are copying the master.

- Whenever write request, it's always on master
  ↑ the most updated copy
  The slaves continuously pull from the master and read from it.
- If read request, it can be distributed across slaves
- In case the master fails, the slaves choose the master amongst themselves.