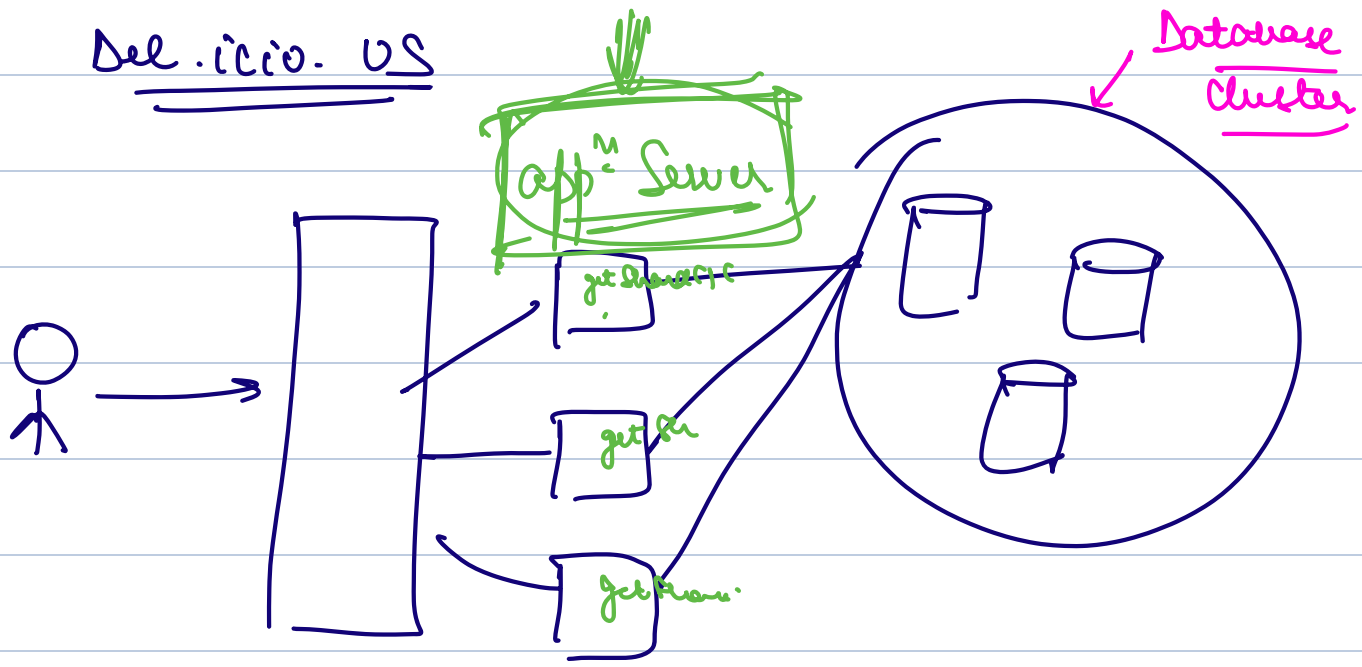
Hashing (How to Shard)

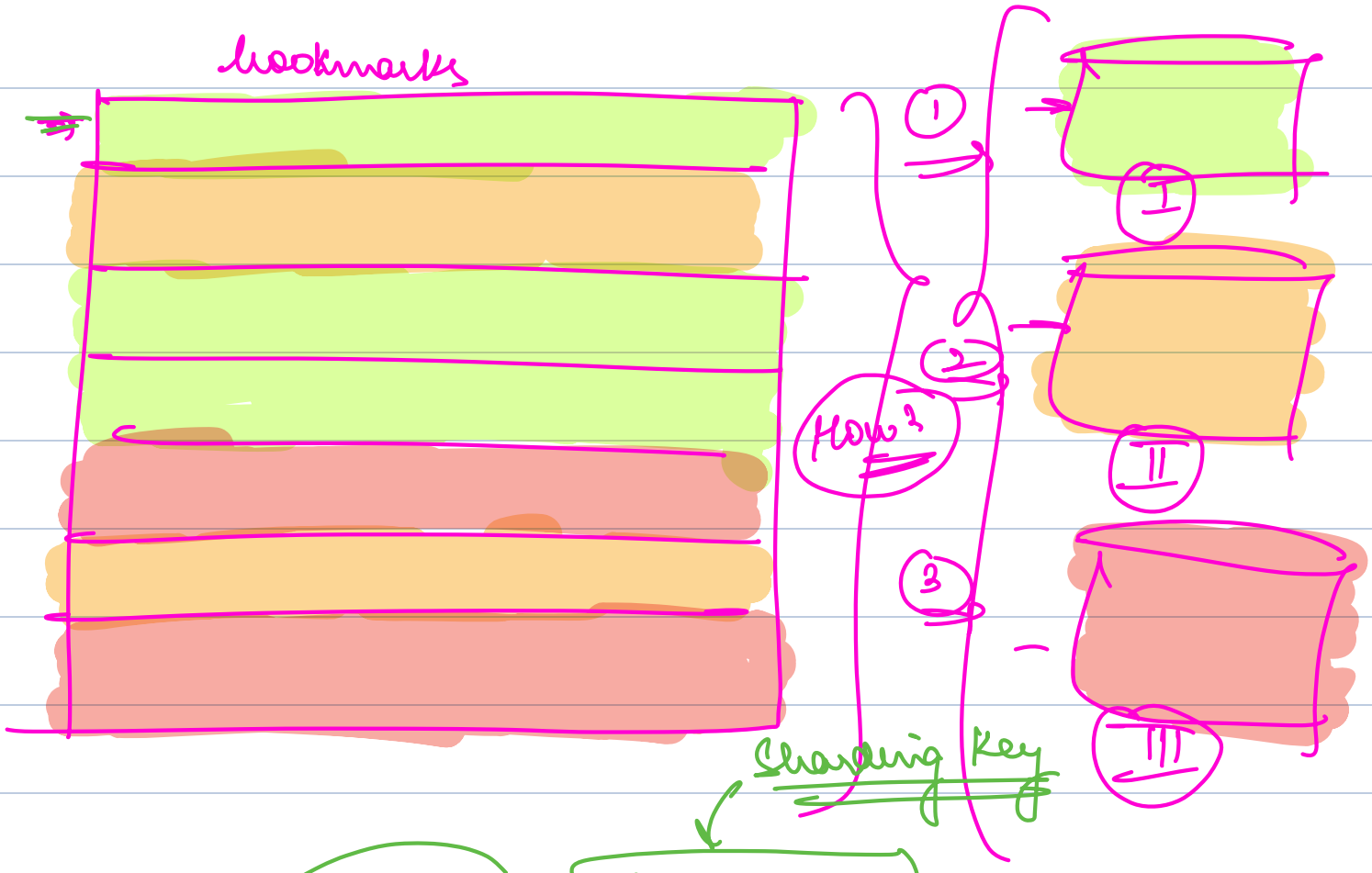
Consistent Hashing

Agenda

1. Modulo Hashing
2. Range Based Hashing
3. Consistent Hashing
4. Optimized Consistent Hashing

Del.icio.us





Sharding Key

int ← getShard (column of table) {

}

}

mongo.save (row)

App^n Server

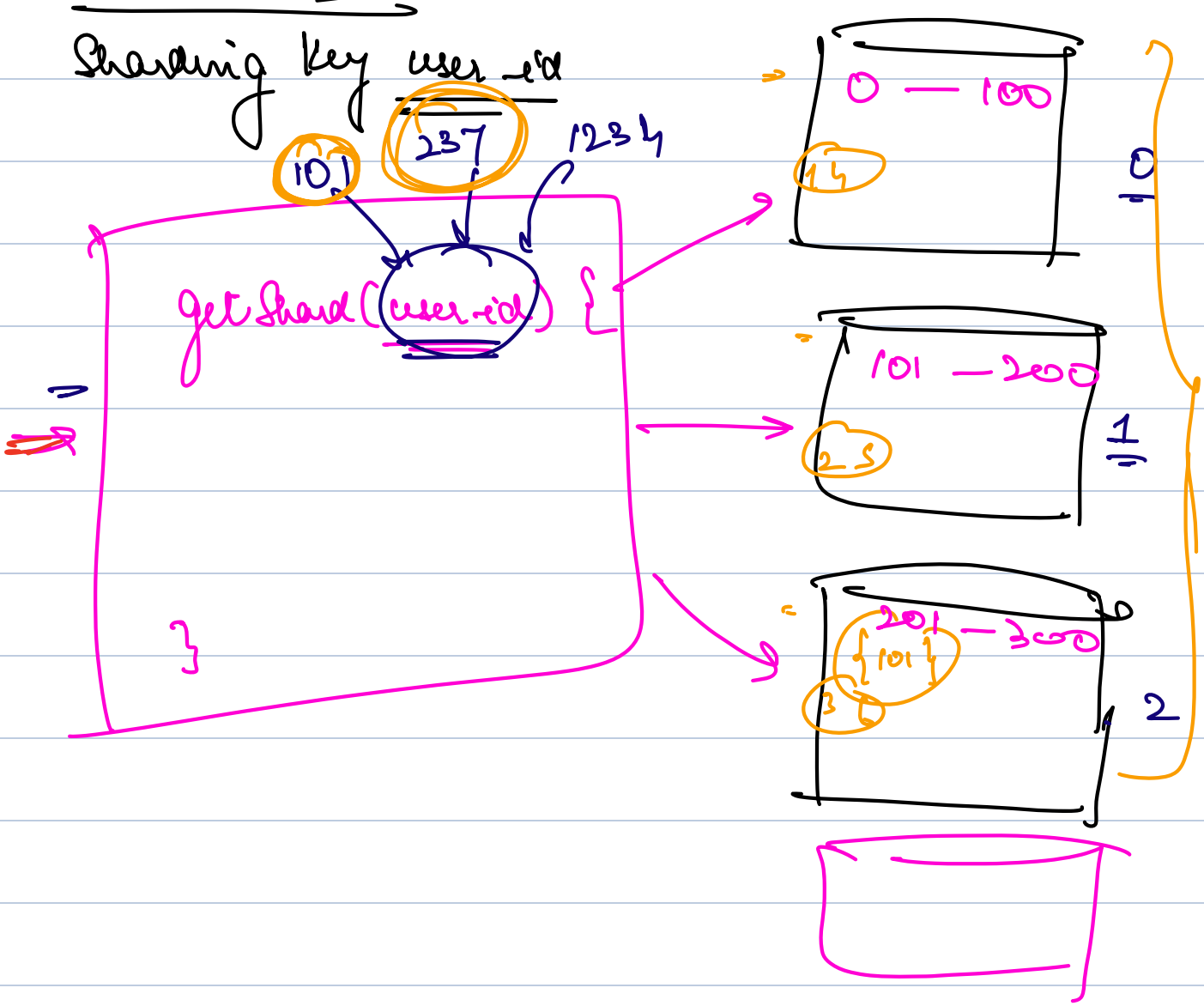
DB1	DB2	DB3
i/p	i/p	i/p
conn1	conn2	conn2
0	1	2

get Shard (row) → 1

DB2 save (———)

Del.icio.us

Sharding key user_id



① MODULO HASHING

Shard = (Sharding key value) % (# Shards)

→ 101 % 3 → 2

→ 237 % 3 → 0

79 ← 237/3 → 79

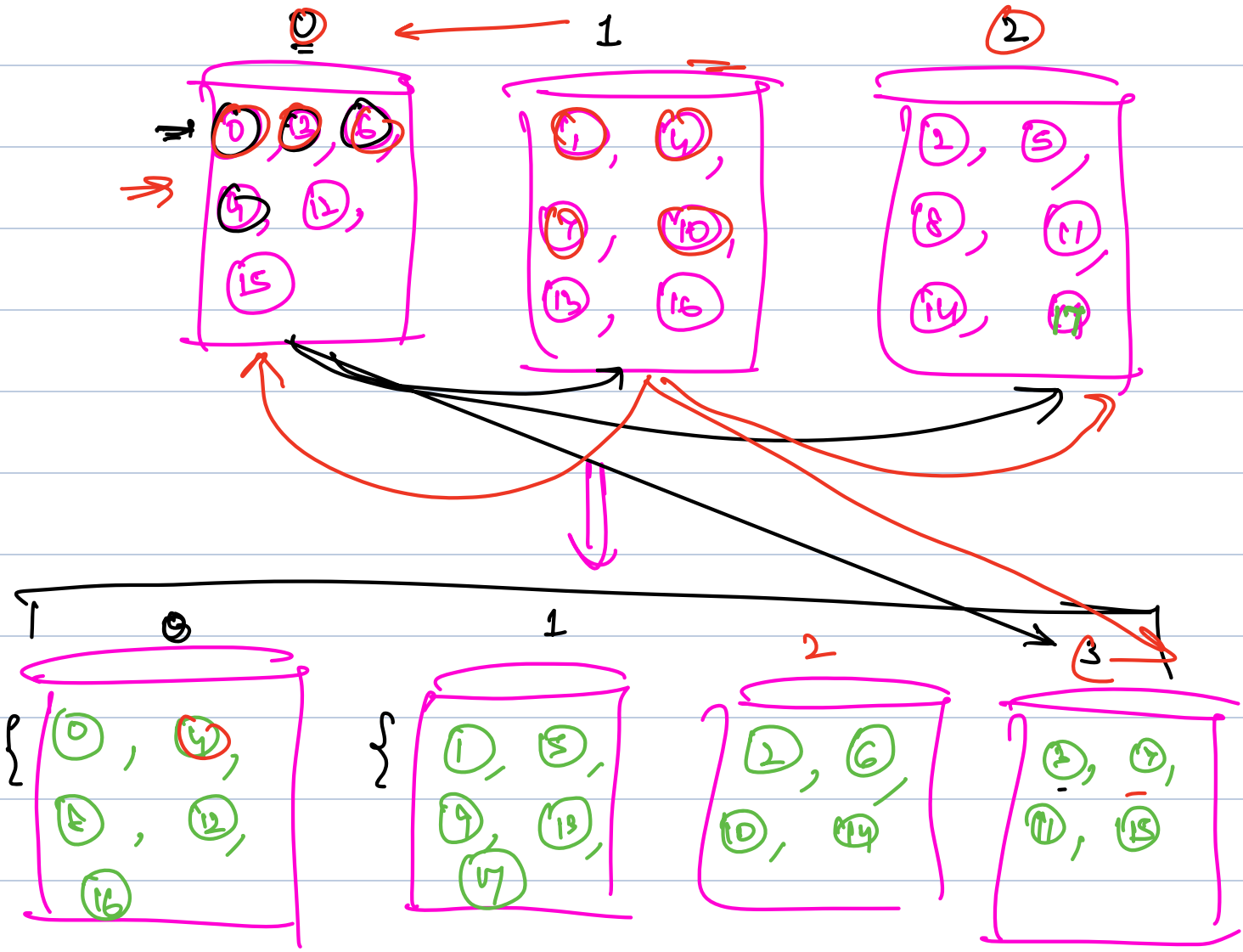
```
get Shard( Sharding_key) {
        return Sharding_key % Shard_count
}
```

Pro:

1.) Simple

2.) Equal Dist Data (v.v.v. High Prob)

Cons



⇒ Data transfer from everywhere to everywhere else

N $\quad$ $O(N^2)$

⇒ decent downtime needs to be taken during trak

(2) RANGE BASED SHARDING

2008 - 2010 2011 - 2013

Simplicity ✓

→ Uniform Dist of Data ✗ → Shards for older users may be overloaded

User_id 3021

Overloaded User id

100 — 1100 30K

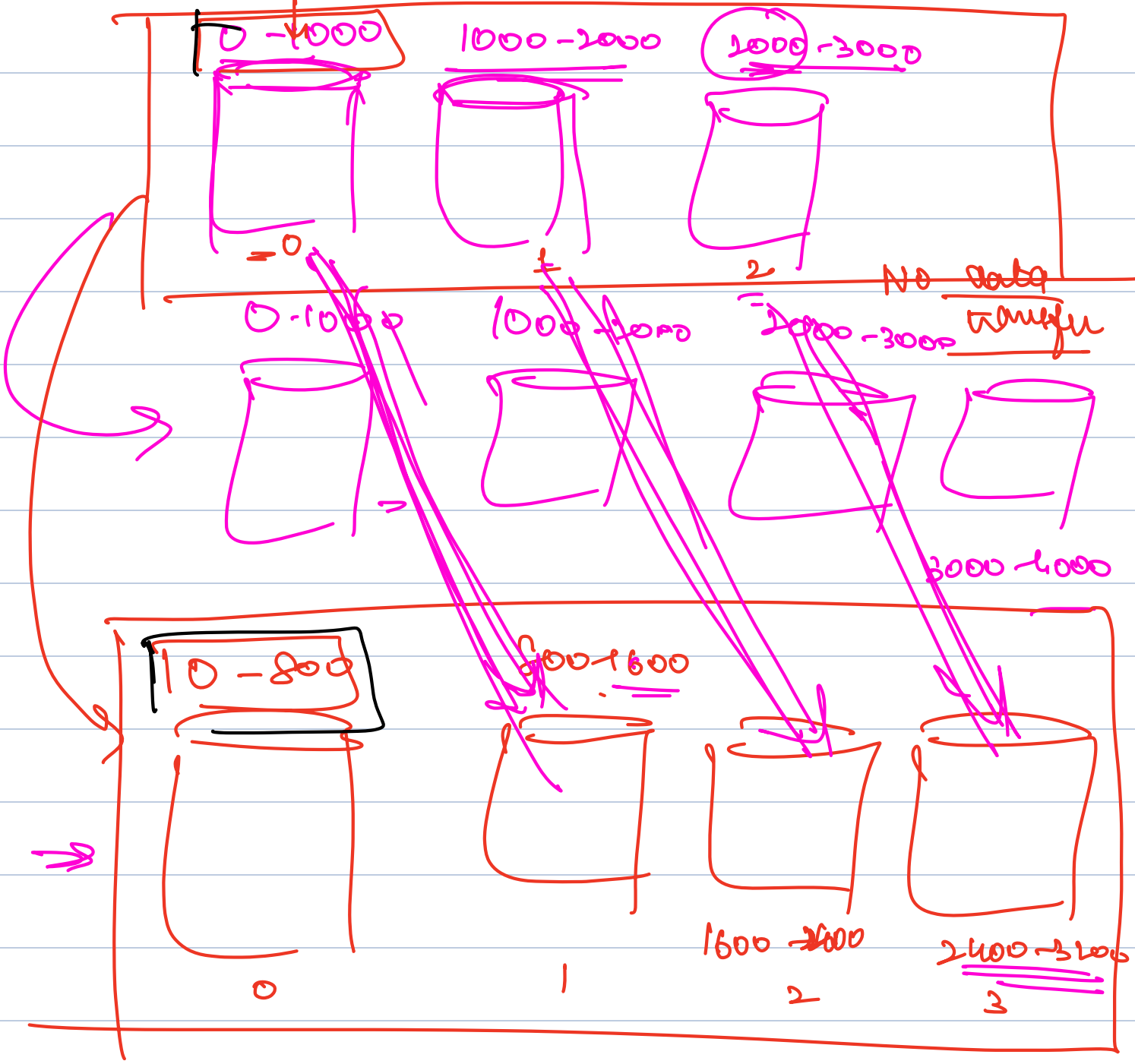
1100 — 2100 11K

2100 — 3100 3K

2002 - 2008 2008 - 2014 2014 - 2020

Data Transfers at the time of Rebalancing

Sol-



O(N) ⇒ lesser data Transfer

	Simplicity	Uniform Load	Less Data Transfer
Modulo	✓	✓	✗
Range	✓	✗	✓

0 — 1000

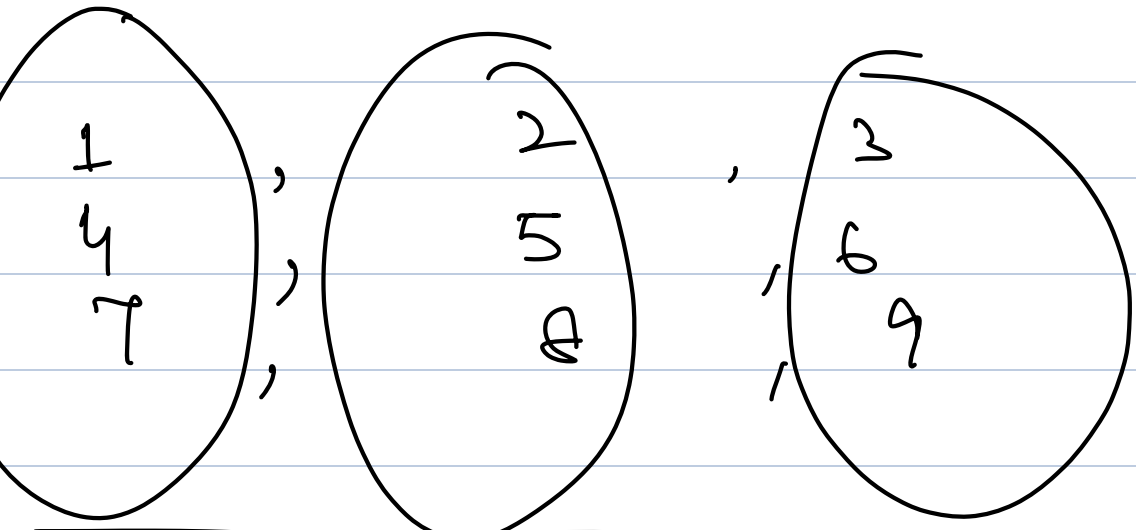
{ ⇒ Storage
⇒ % of CPU used }

III Consistent Hashing

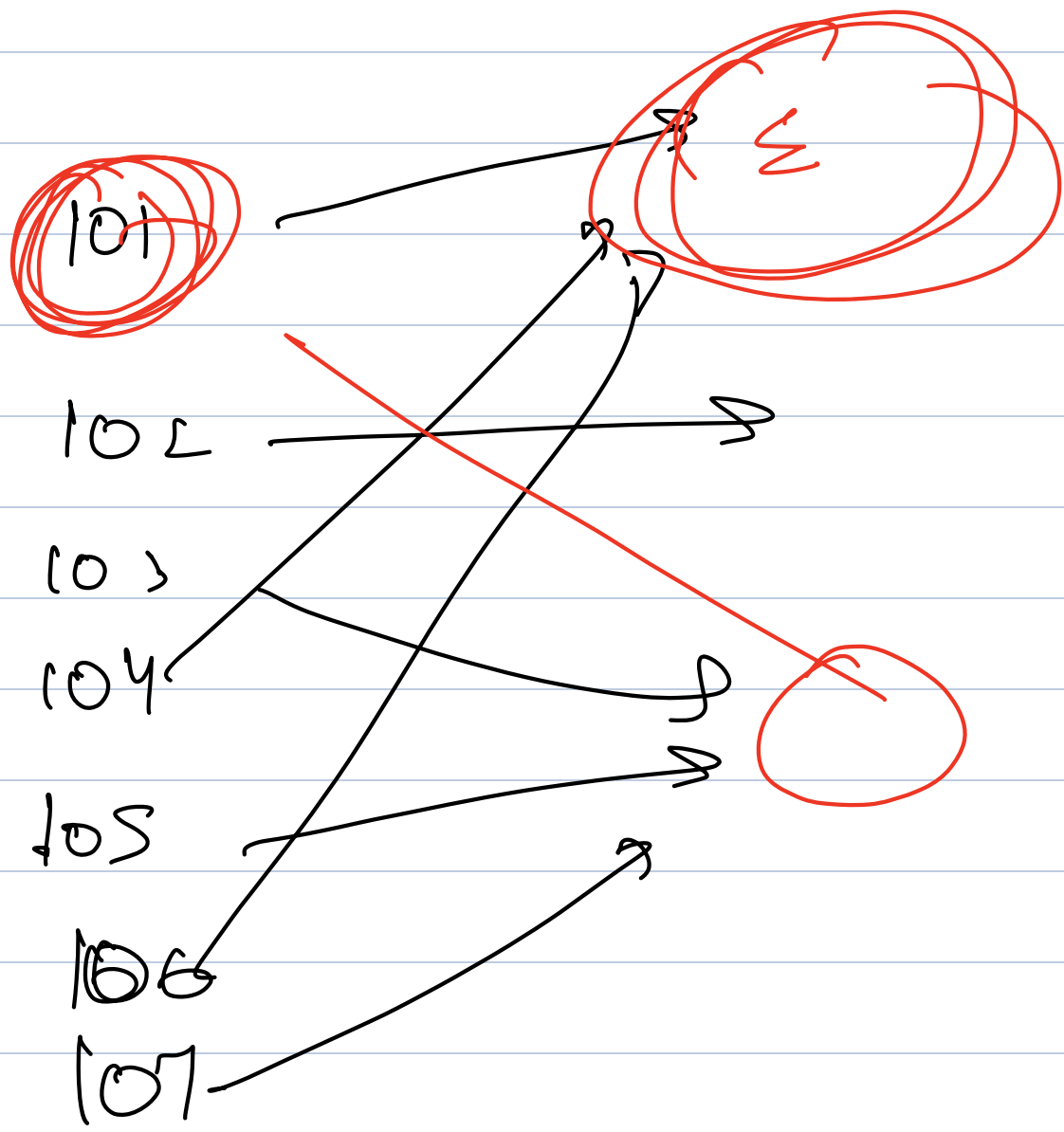
→ Modulo Based → Range Based.

⇓ Uniform Dist ⇑ Less Data Transfer

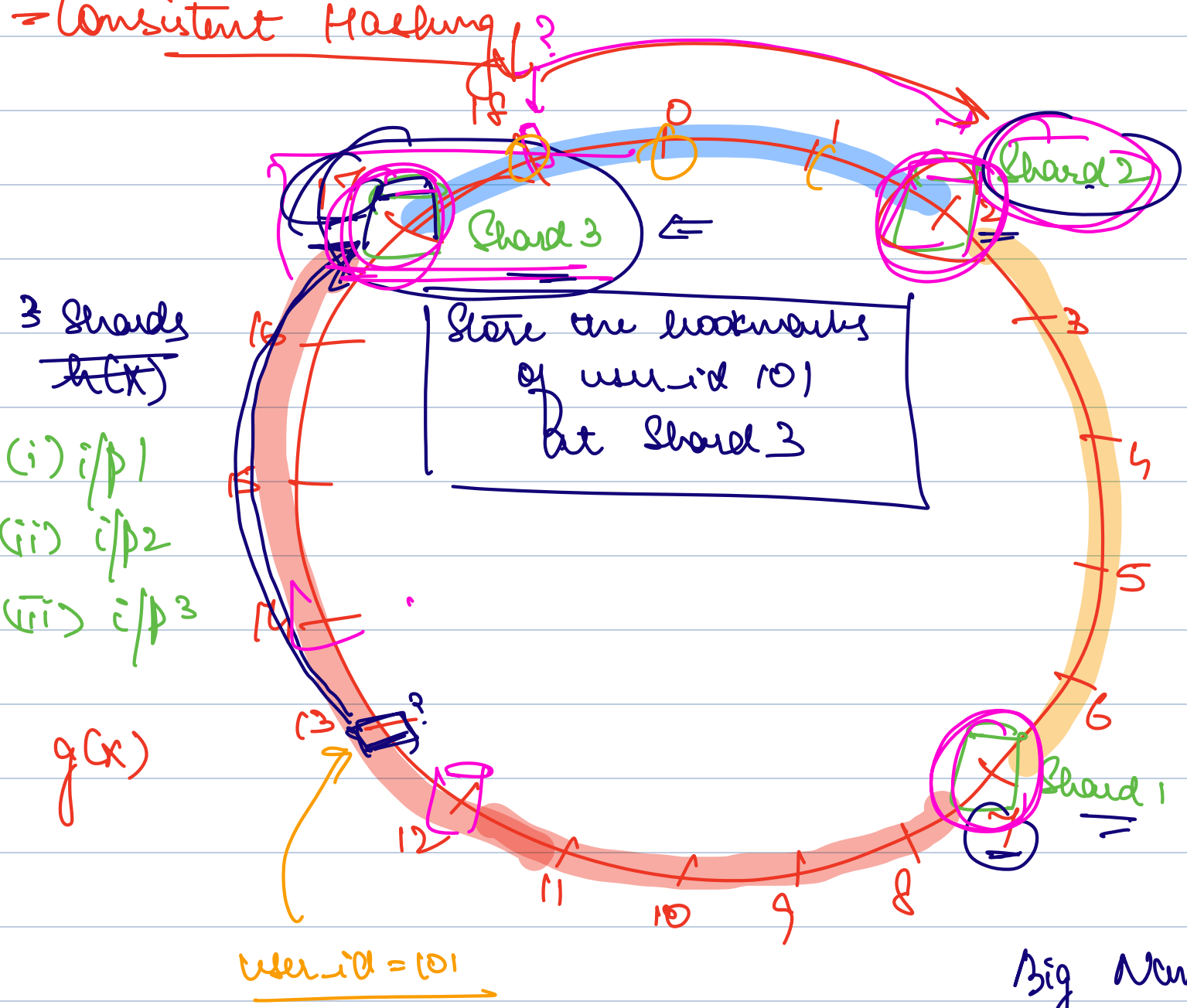
↳ probability



$$\boxed{\Sigma x == \Sigma y == \Sigma z}$$



Consistent Hashing



Big Number

① Create a circular ring have numbers on it (large Range) → (0 – 10 Million)

② Have a hash function $h(x)$ which given a value x returns a number in range defined above

$h(101) \rightarrow$ Same value

$h(x) \in [0, 10 \text{ Million}]$

$\in [0, 18]$

(3) Hash the i/p add of each shard and put it at the corresponding position on the ring

$h(i/p\ 1)$	$\rightarrow$	7
$h(i/p\ 2)$	$\rightarrow$	2
$h(i/p\ 3)$	$\rightarrow$	17

(4) Have another hash function

$g(x)$

which also returns values in range above

$g(x) \in [0, 10 \text{ Million}]$

$\in [0, 18]$

(5) For every row that has to be saved, pass its sharding key via $\boxed{g(x)}$

eg

bookmark

user_id	title	content
101		
311		

$\boxed{g(101)} \Rightarrow \boxed{[0, 18)}$

$\rightarrow \boxed{13}$

(6) Store that row at the shard that is at that posn or to immediate right

$g(311) \Rightarrow 18$

Implementation

① $h(x)$ $\qquad\qquad$ $g(x)$

② create hash of each server i/p using $h(x)$

you store hash value in array in sorted way

Sorted array

RING = [(2, i/p 2) (7, i/p 1) (17, i/p 3)]

③ while storing row:

g(Sharding key)

$\Rightarrow$ eg $\Rightarrow$ 12

Something in ring >= 12

Nearest greater

$\Rightarrow$ Binary Search

↳ if null $\Rightarrow$ array[0]
else array[ans]

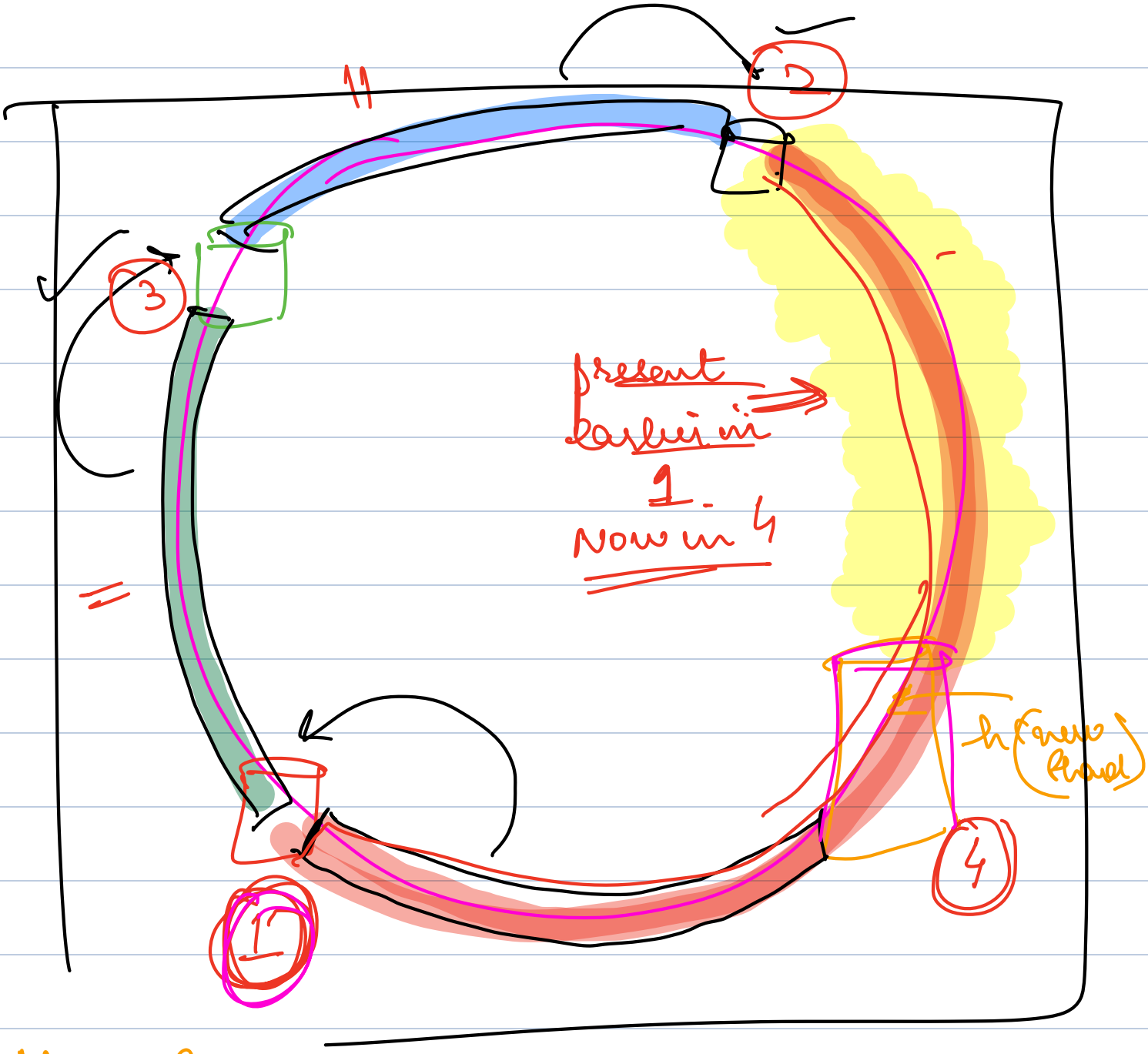
Simplicity → (diff to learn)
→ easy to imp

Uniform Data Dist → ✓

less Data Transfer → increase shard count
✓

ConsHashing() {
binary Search

}



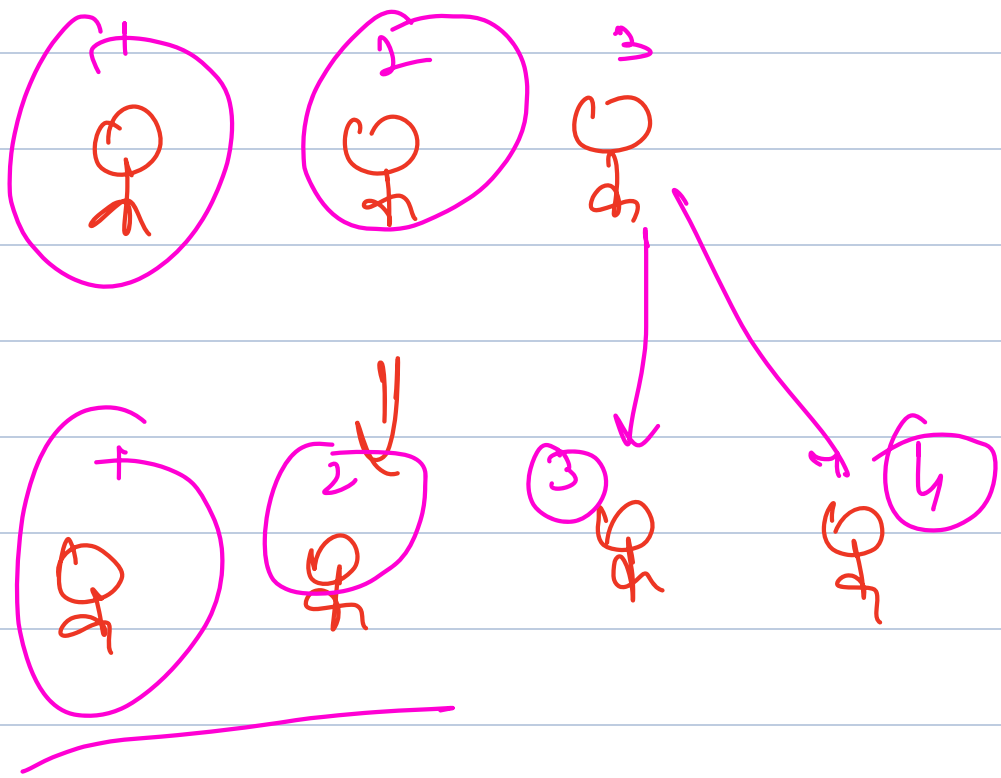
New Shard

1. do h(x) for the new shard's i/p address
2. put that in Hash Ring (array) at correct position

⇒ ONLY 1 DATA TRANSFER

2 + 3

2 + 3 + 4



In our case, load of every shard | was reduced

OPTIMIZED CONSISTENT HASHING

① Have Multiple hash function (eg: 3)

$h(x)$ $i(x)$ $j(x)$

that will be used to hash i/p add of shards.

② Have single hash f^n

$g(x)$

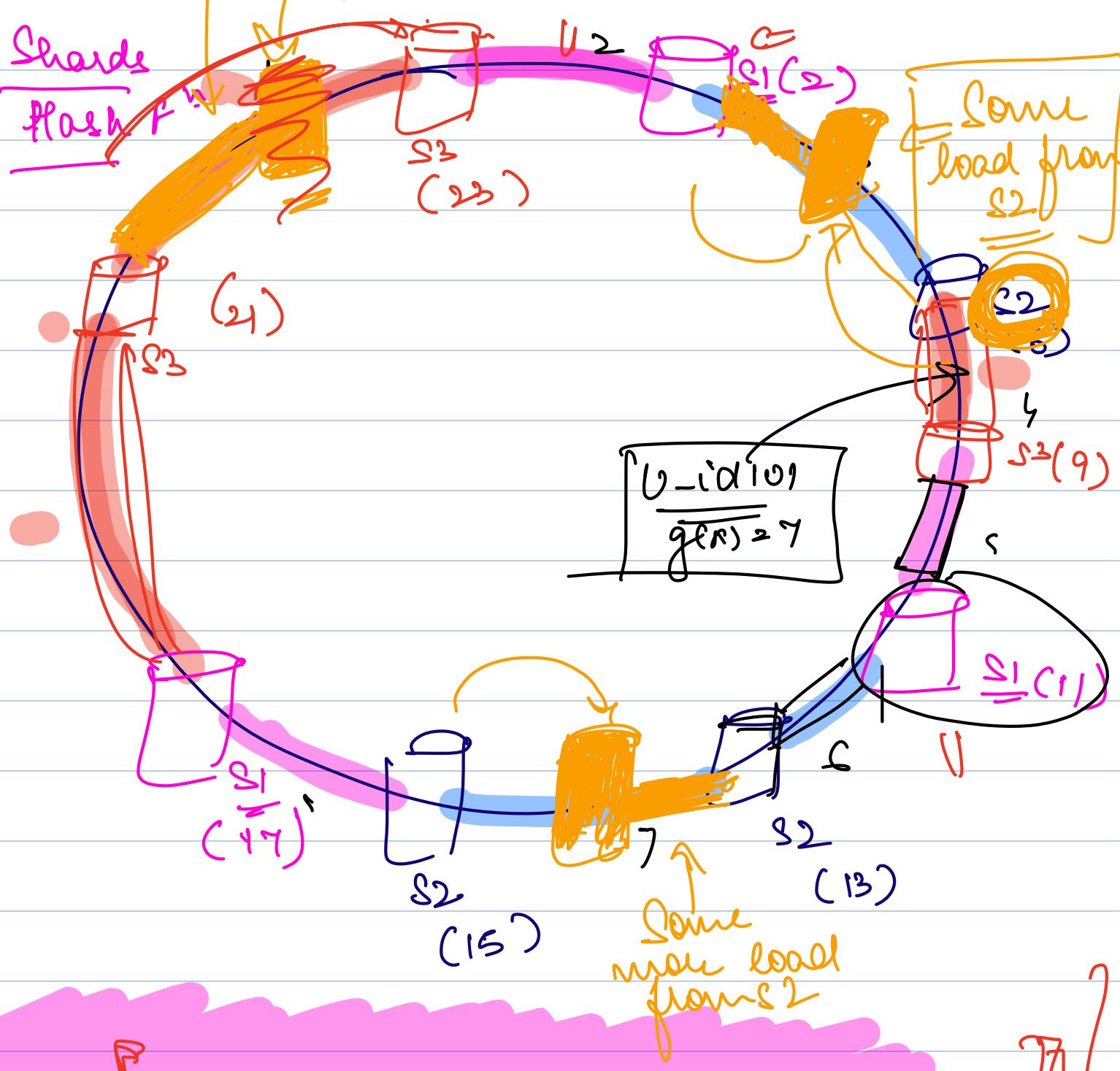
⇒ to hash the Sharding Key

③ For each shard, place it at the position up by each hash f^n of it

Some load from S3

3 Shards

3 Hash F



RING = [(2, i/p1) (6, i/p2) (9, i/p3) (11, i/p1) (13, i/p2) (15, i/p2) (17, i/p1) (21, o/p3) (23, i/p3)]

BOOKMARK ⇒ [u_id → 101] g(101) ⇒ 7

100 — 200 → S1

200 - 300 ⇒ S2

300 - 400 → S3

YES

100 - 125 → S1

210 = 245 ⇒ S1

240 - 330 ⇒ S1

125 - 210 ⇒ S3

330 - 400 ..

99 → 3 equal chunks
→ each chunk = 33
→ each shard = 33

NOW

99 → 9 equal chunks
→ each chunk = 11
→ each shard = 11 * 3 = 33

add^n of new shard

Simple ✓

uniform dis ✓

less transfer ✓

Balances load on add ✓

$O(\text{Max } k^n)$

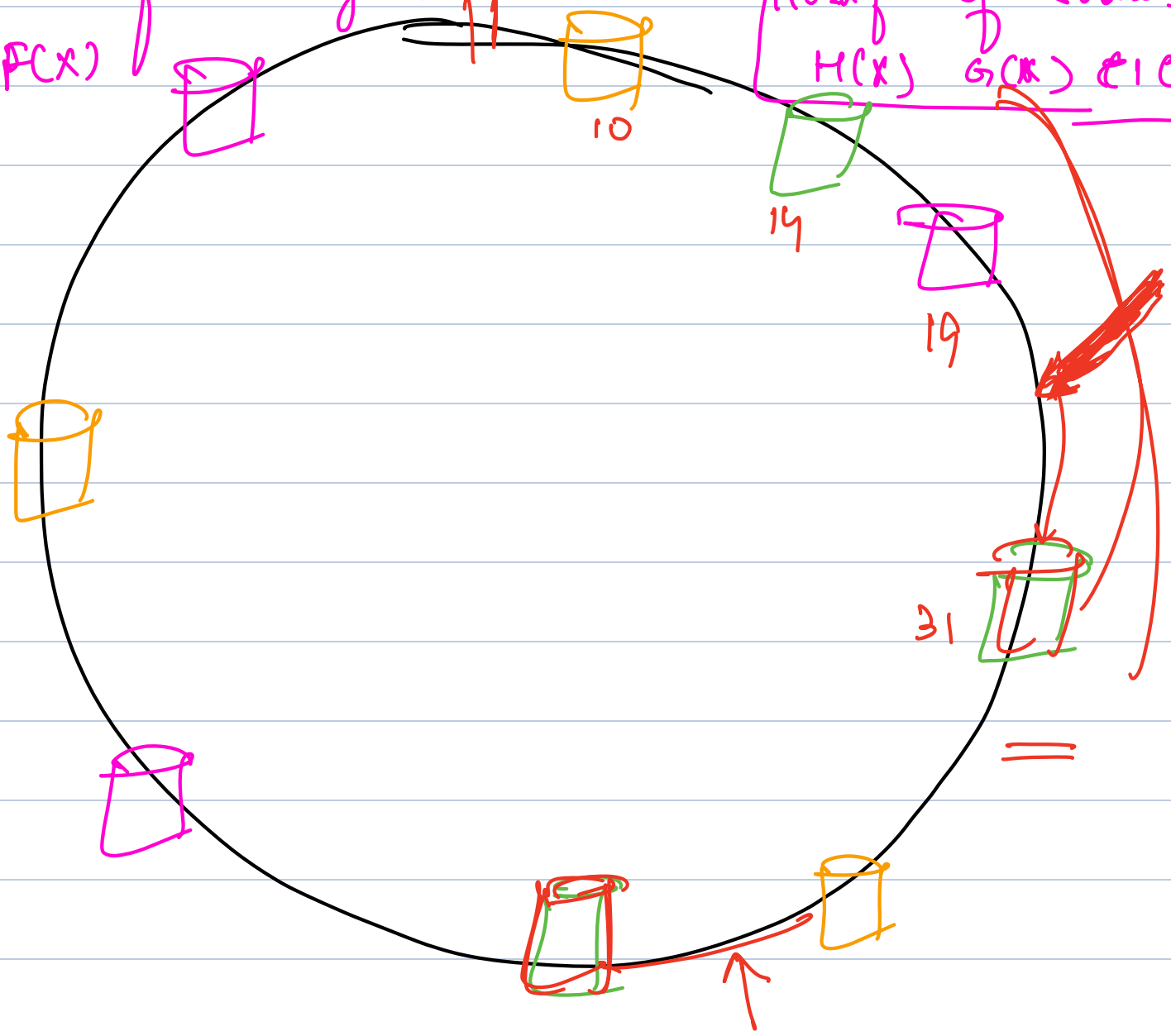
$\boxed{\text{KISS}}$

Hash F^n for Sharding begin from 0

$F(x)$

Hash f^n of Shards

$H(x)$ $G(x)$ $C1(x)$



ring = [(10, ip_y) (14, ip_g) (19, ip_p) (31, ip_g) ,]

10K

1mn