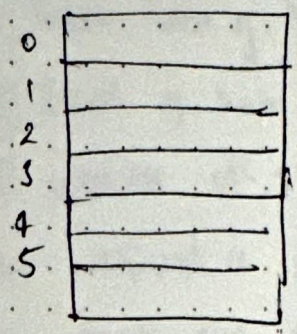
DESIGN CONSISTENT HASHING

Key = "Prodyumna" ← arbitrary length

hash function take this key as the input.

f("Prodyumna") → 12345

A technique called mod hashing



we will have hash table of fixed size.

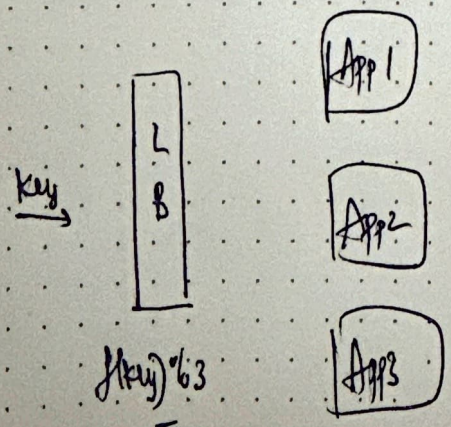
so, we can say 12345 % 6 =

6 → size of hash table

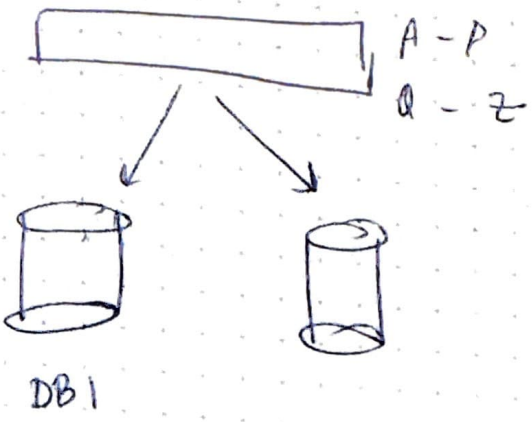
so that prodyumna will go and get stored there.

Problem with fixed hashing?

for ex:- our use case are load balancing for application servers horizontal scaling



If mod hashing or some hashing technique is used then, we dont know how to distribute the request equally

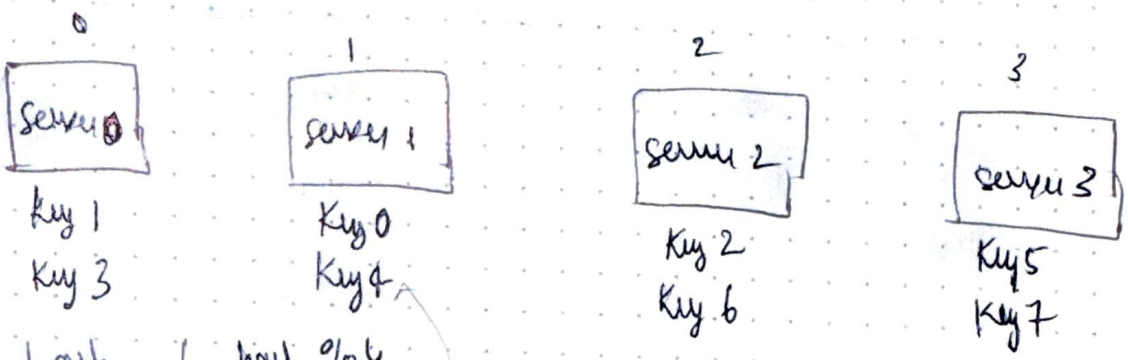


Say y all names are A then everything gets filled in one DB. But we need to divide equally.

ServerIndex = hash % no: of servers

This approach works well when the serverPool size is fixed, and data distribution is even. Problems arise when new servers are added, or existing servers are removed

Eg :-



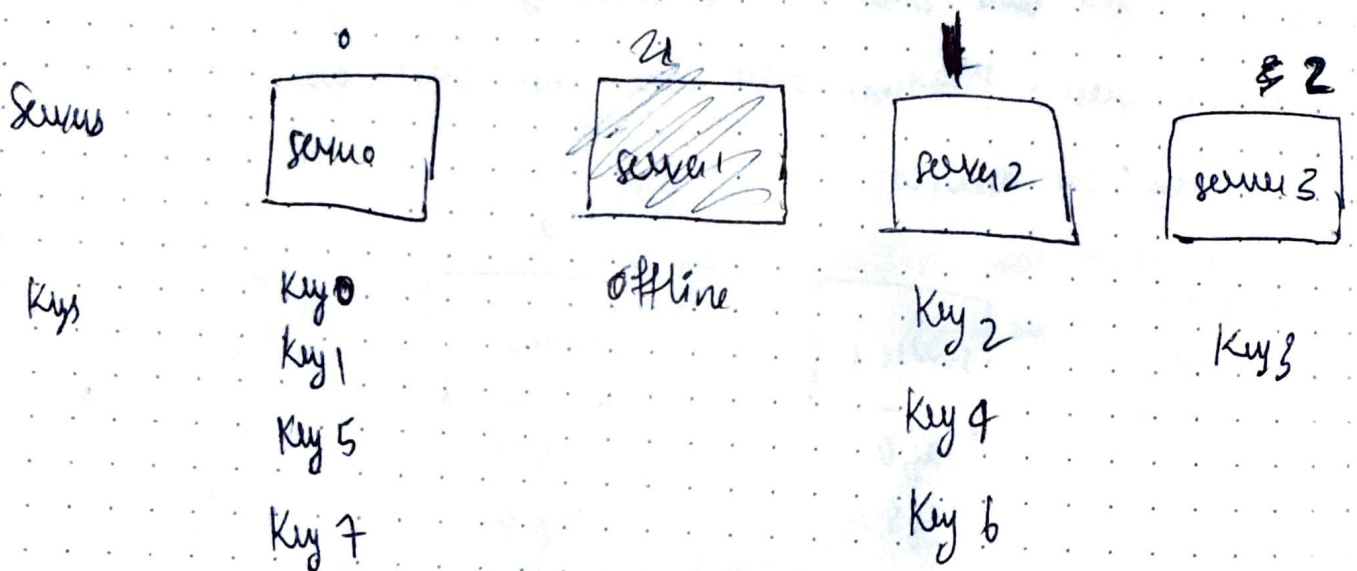
Keys

Key	hash	hash %4
Key 0	85	1
Key 1	84	0
Key 2	86	2
Key 3	44	0
Key 4	45	1
Key 5	47	3
Key 6	46	2
Key 7	87	3

Say one server goes down then, we have to do % 3

	hash	hash % 3
Key 0	85	1
Key 1	84	0
Key 2	86	2
Key 3	44	2
Key 4	45	1
Key 5	47	0
Key 6	46	1
Key 7	87	0

→ just for example



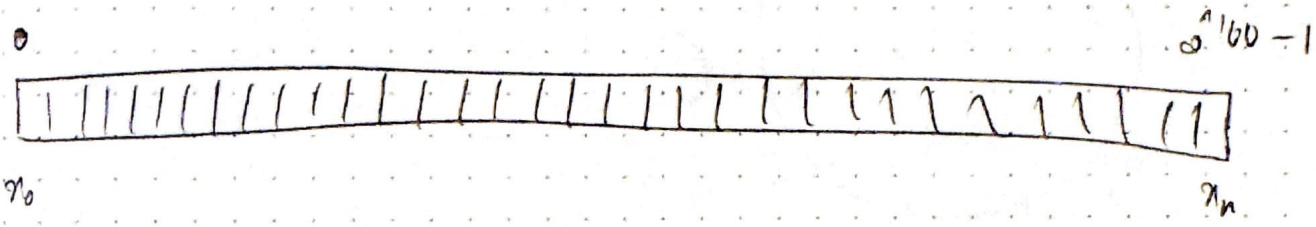
So, when server 1 goes offline, most cache clients will connect to the wrong servers to fetch data. This causes a storm of cache misses. Or say more bad just on one server.

Consistent hashing mitigates this problem.

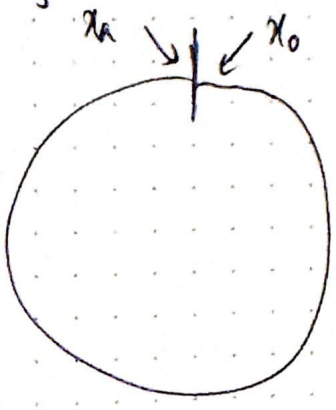
Consistent Hashing

Quoted from wikipedia "Special kind of hashing such that when a hash table is re-sized and consistent hashing is used, only K/n keys need to be re-mapped, where K is no: of keys and n is no: of slots. In most traditional hash tables, a change in number of array slots causes nearly all keys to be remapped".

Assume SHA-1 is used as Hash function f and o/p range is $x_0, x_1, \ldots x_n$. In cryptography SHA-1 ~~goes~~ hash space goes from $2^{160}-1$. Meaning $x_0 \rightarrow$ corresponds to 0 and $x_n \rightarrow 2^{160}-1$.



By collecting both ends we get a hash ring (virtual / circular queue)



Hash servers

using the same hash function f, we map servers based on server IP or name onto the ring.

server → mod hashing

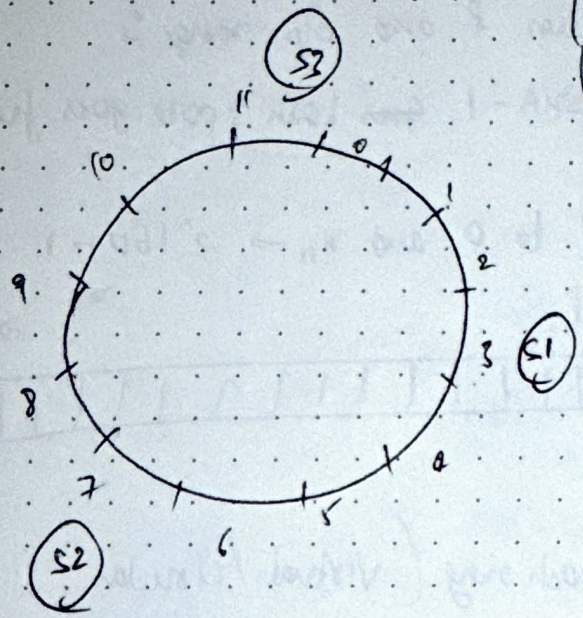
say we have 3 servers say,

hash(server1) % 12 = 3

hash(server2) % 12 = 7

hash(server3) % 12 = 11

→ equally distanced hashing.



Hash Keys

cache Keys

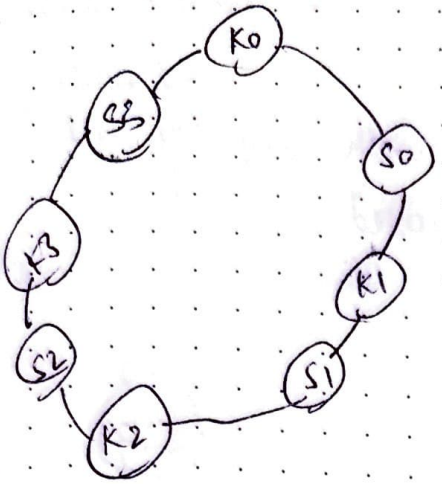
Key1 = hash(Key1) % 12

Key2 = hash(Key2) % 12

⋮

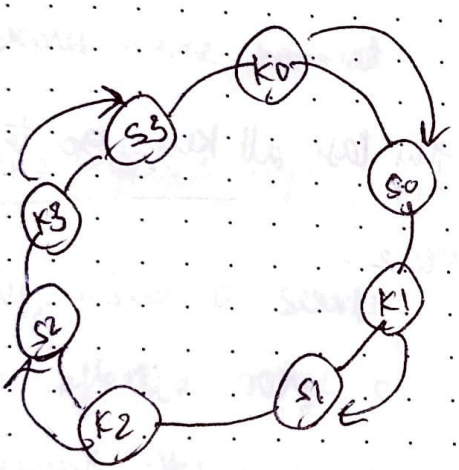
Key6 = ←

etc



Server lookup

To determine which server key is stored on, we go clockwise from the key until server is found.



Add a Server

Adding a new server will only require redistribution of a fraction of keys.

Similarly with removing a server as well.

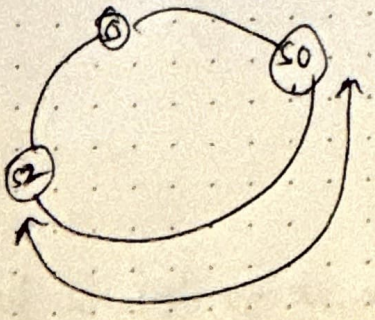
Unlike fixed hashing where all the keys had to be redistributed.

2 issues in this basic approach

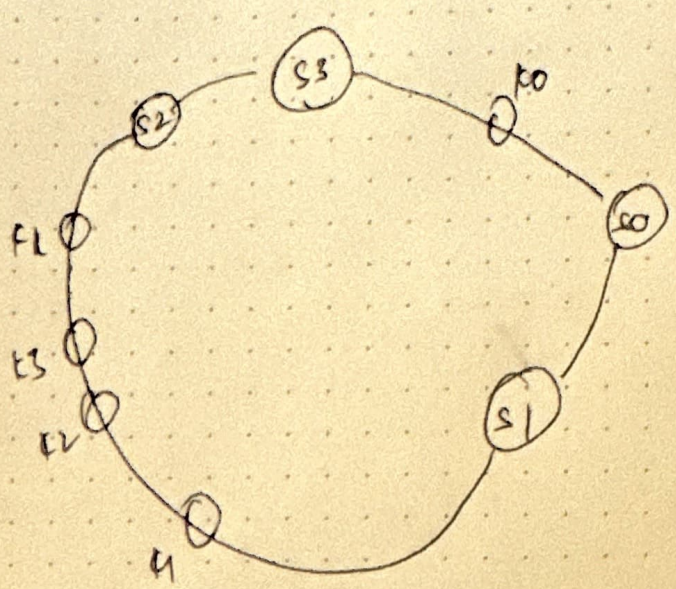
(1) Impossible to keep the same size of partitions on the ring for all servers considering a server can be added or removed

partition → Hash space between the servers

Eg:-



(2) Possible to have a non-uniform key distribution on the ring.

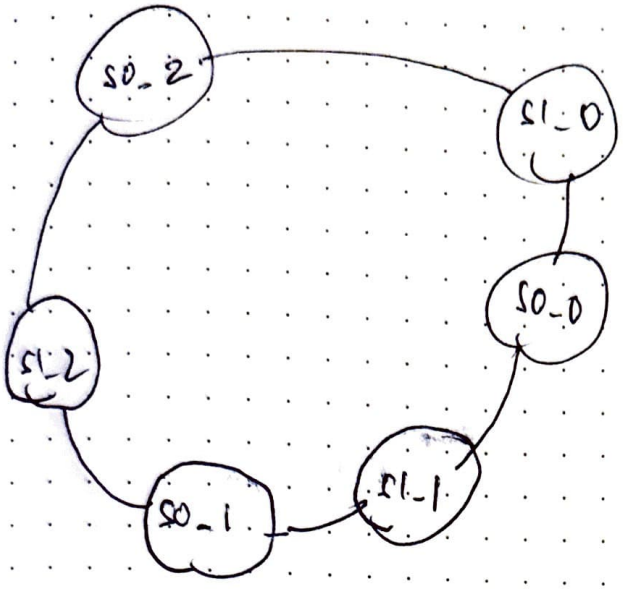


So, this case, all keys go to server 2.

→ A technique called virtual nodes or replicas is used to solve this problem.

Virtual node refers to the real node, and each server is represented by multiple virtual nodes on the ring.

with virtual ~~server~~ nodes, each server is responsible for multiple partitions.



To find the key go clockwise

As the number of virtual nodes increase, the distribution of keys becomes more balanced.

Find Affected Keys

Say, when a server is added Say server 4 is added to the ring. The affected range starts from S4 and moves anti-clockwise around the ring until a server is found (S3). So keys b/w S3 and S4 have to be redistributed to S4, as before they were in S0