Agenda.

→ Consistent Hashing.

→ Caching Introduction

Consistent Hashing.

→ Stateful Load balancing algorithm.

→ Looking to reduce the chaos in case a m/c goes down or a new m/c comes up.

→ Easy to maintain

→ Fast.

Hash functions.

Servers. : S_1 S_2 S_3 S_4

H_s

↳ $[0-10^{18}]$

$H_s(S_1) = S_1'$

$H_s(S_2) = S_2'$

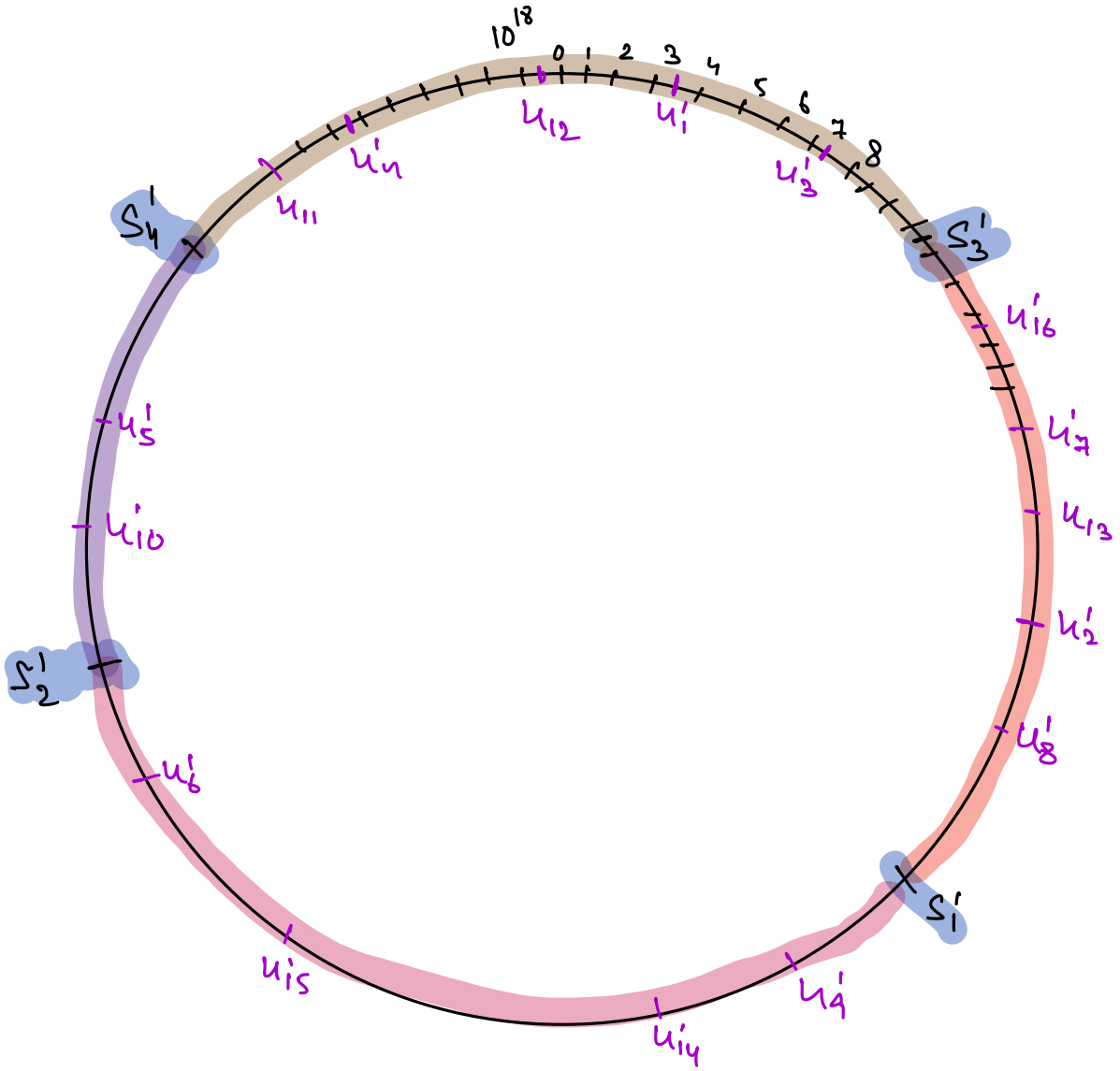
$H_s(S_3) = S_3'$

$H_s(S_4) = S_4'$

H_u

$H_u(u_1) = u_1'$ $H_u(u_3) = u_3'$

$H_u(u_2) = u_2'$ — — — —



⇒ For every user, Allocate the server which is the first server in the clockwise direction.

→ Almost equal distribution.

$H_s(s_1) = 10^7$

$H_s(s_2) = 10^9$

$H_s(s_3) = 100$

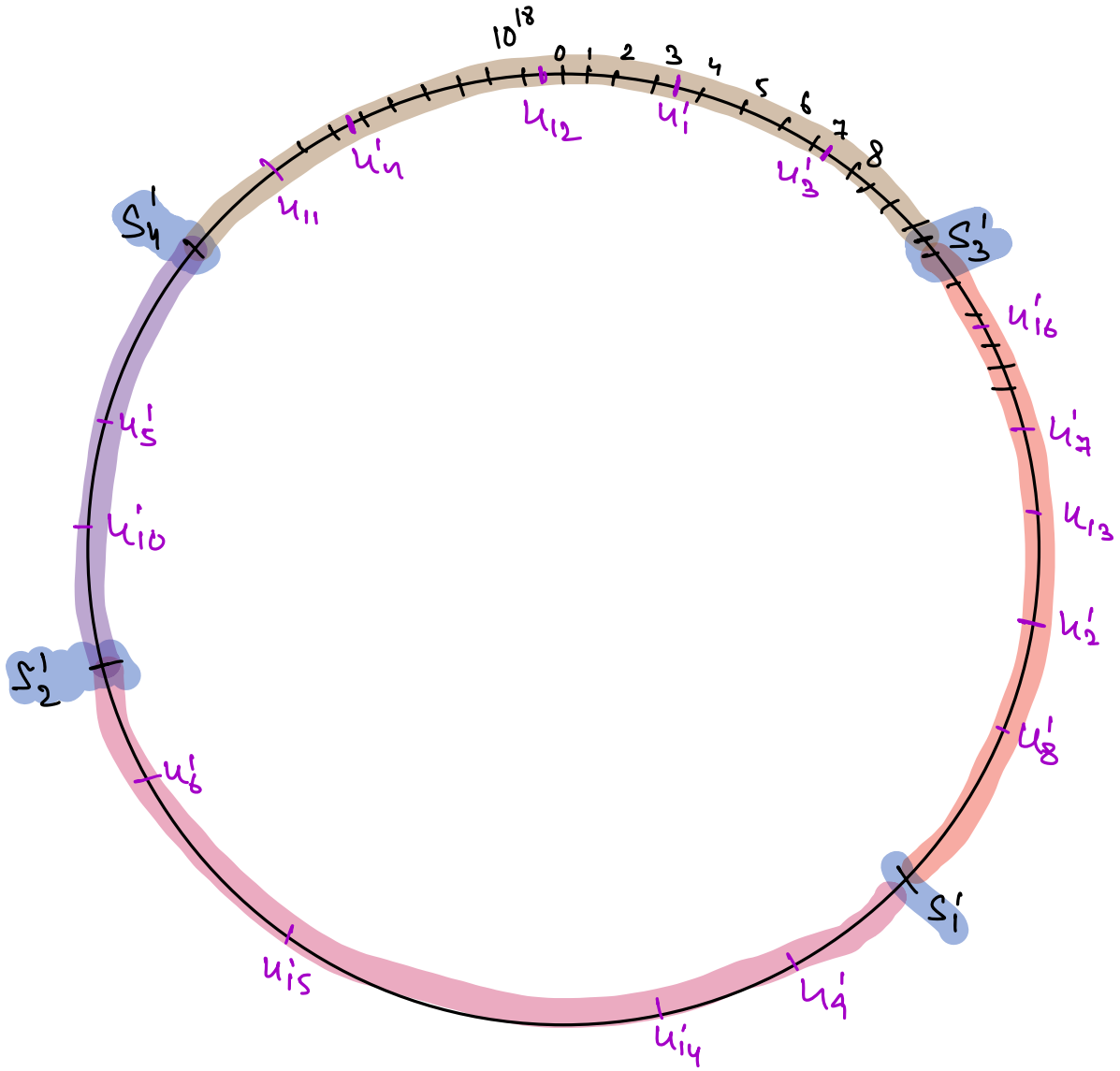
$H_s(s_4) = 10^{12}$

100	10^4	10^6	10^7	10^{12}

$H_u(u_1) \Rightarrow u_1' \rightarrow 10^5$

To allocate, find out the next greater element in the sorted array.

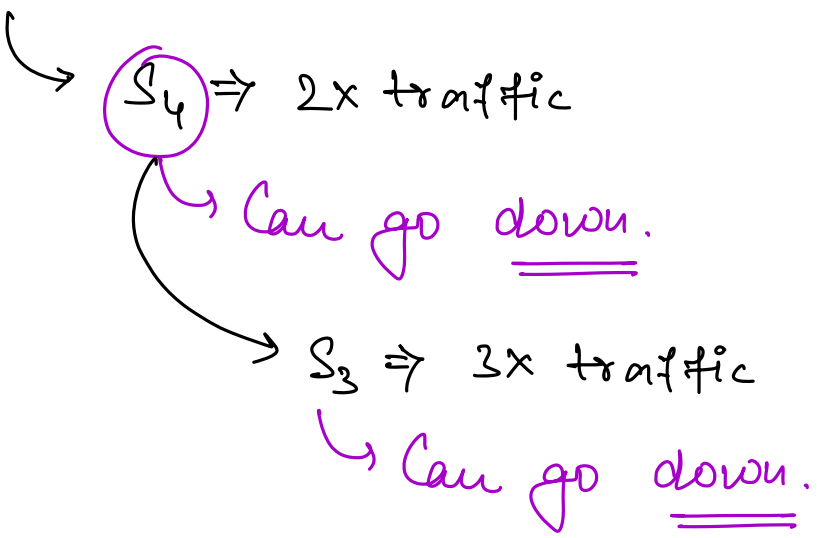
- Binary Search.
- upper bound.
- $\log N$. Time Complexity.



If a m/c goes down then all the traffic of that m/c will go to the next m/c in the clockwise.

→ Only the users which were tagged to the m/c which is going down will be impacted.

S_2 goes down.

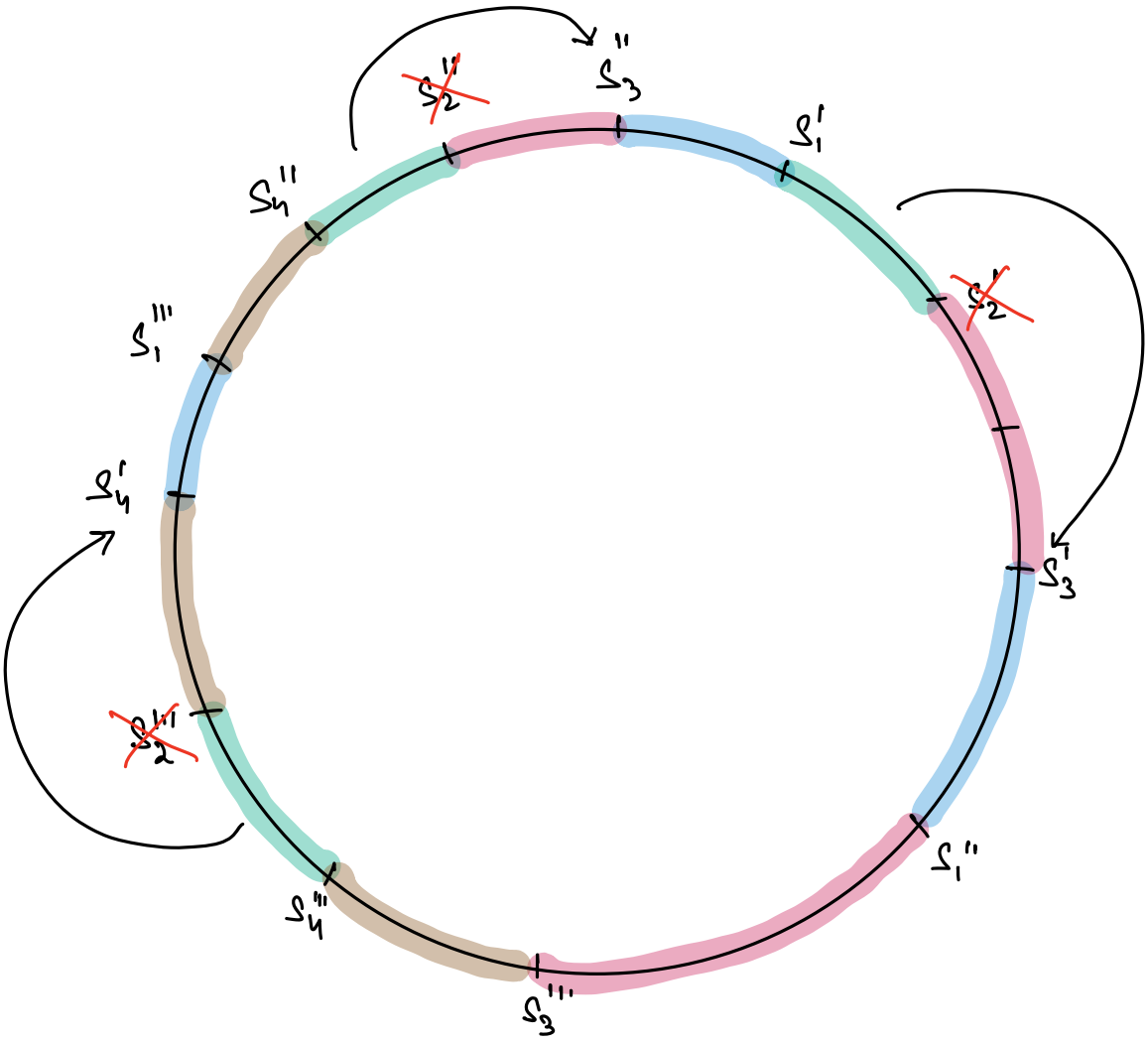


Cascading failure.

⇒

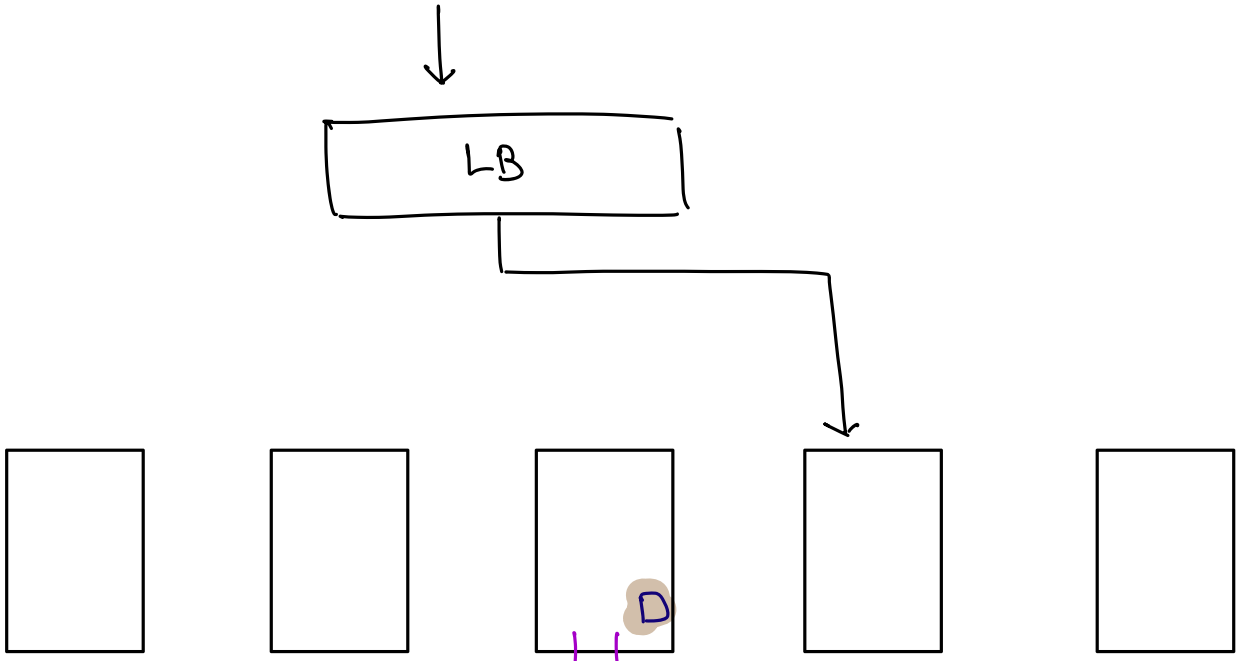
$H_{S_1}(S_1) = S_1'$	$H_{S_1}(S_2) = S_2'$
$H_{S_2}(S_1) = S_1''$	$H_{S_2}(S_2) = S_2''$
$H_{S_3}(S_1) = S_1'''$	$H_{S_3}(S_2) = S_2'''$

- - - -



S_2 goes down

Caching.



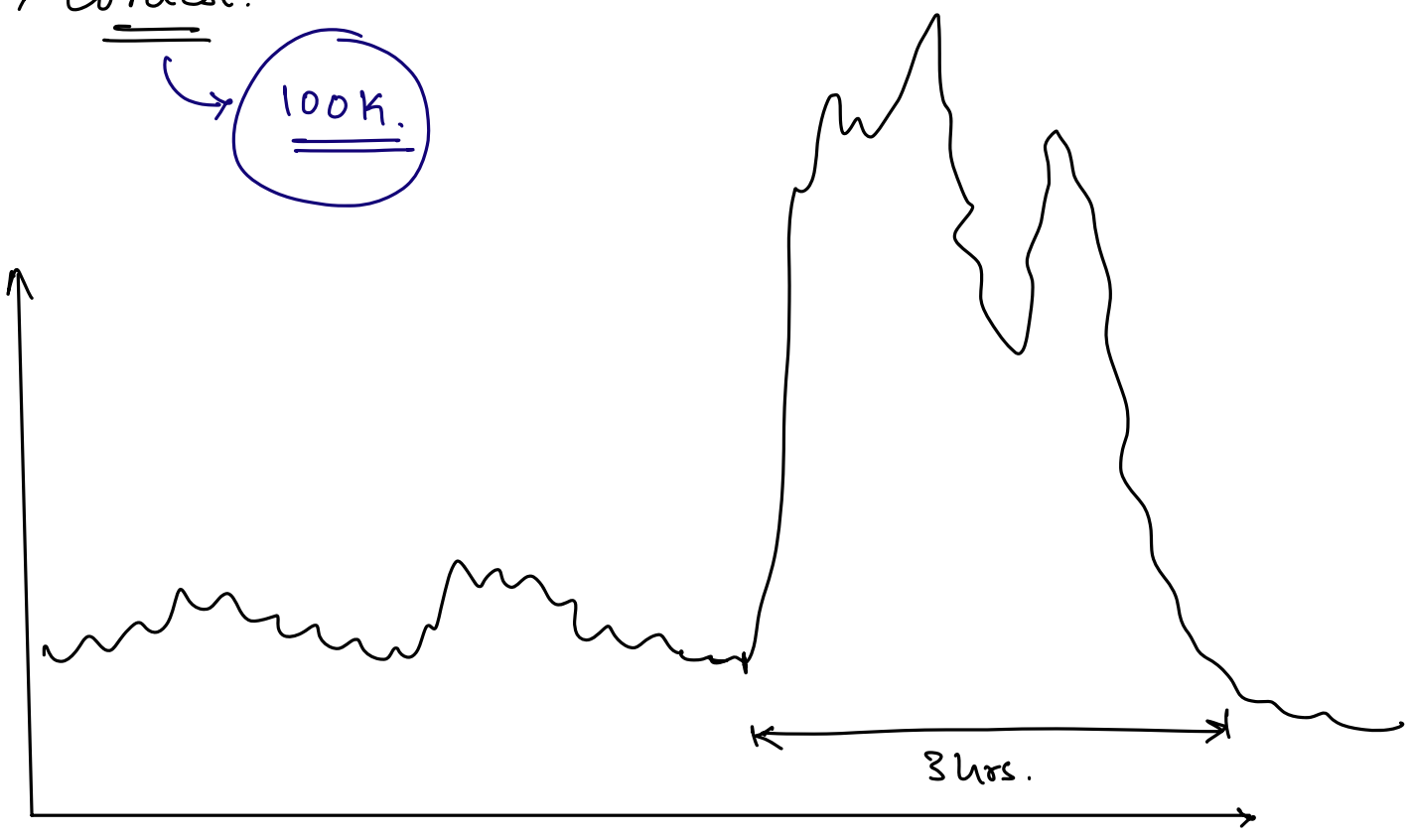
Code

Storage.

Scaler.

↳ Contest.

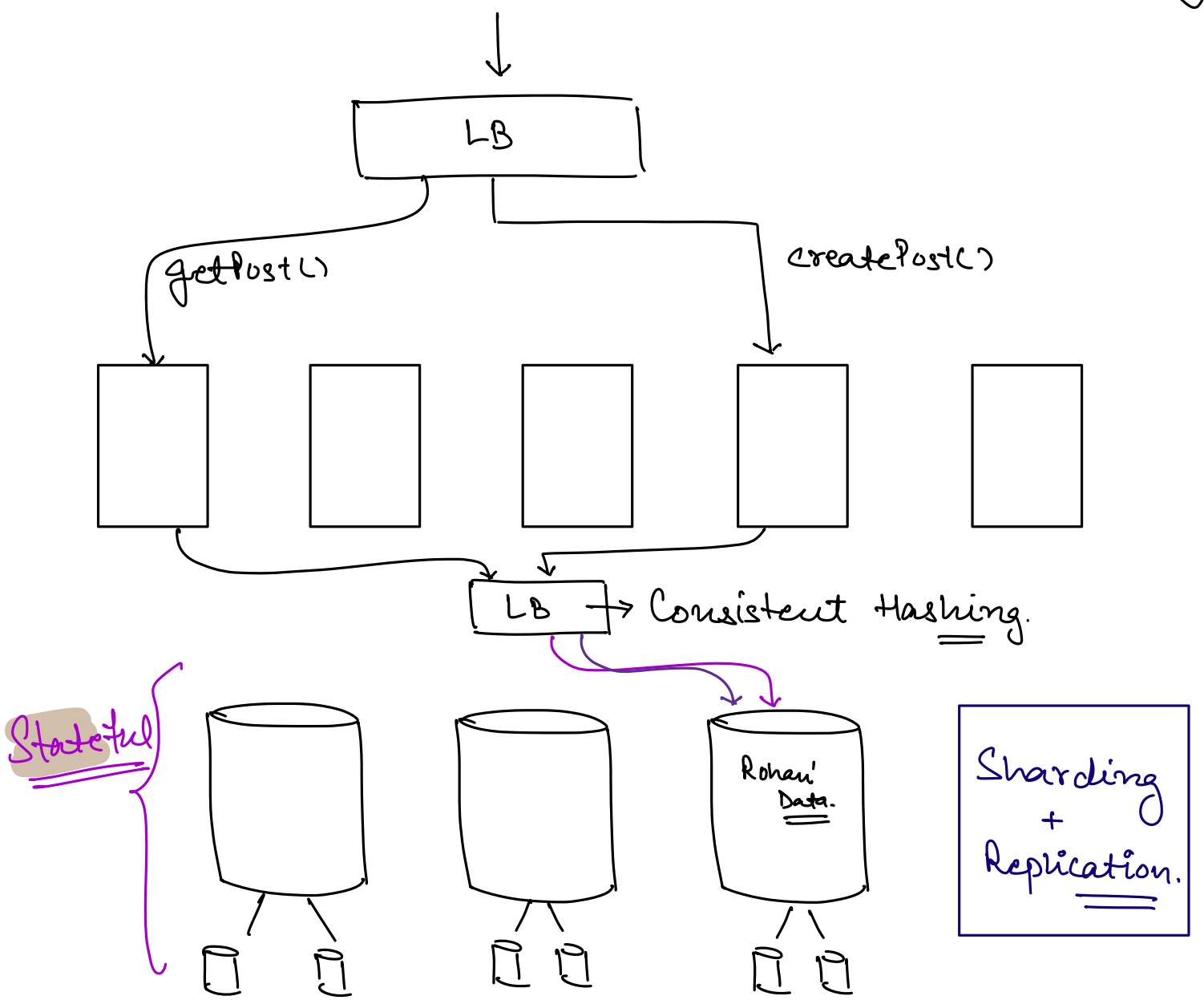
100K.



Compute. → CPU.
↳ RAM.

⇒ Scale app servers & DB independently.

⇒ Constant deployments to the app servers will make the machines unavailable for some time, and during this time our DB will also become unavailable, that is unnecessary.



→ Additional n/w hop

↳ Latency ↑ } Trade Off

Caching.

↳ Process of storing copies of data near to the user so that user can access it quickly and the latency can be reduced.

Client Side Caching

CDN.