# Scalability & System design

HELLO!

## I am Ibrahim Gamil

Software engineer at Amazon

Software engineer at Funnelll

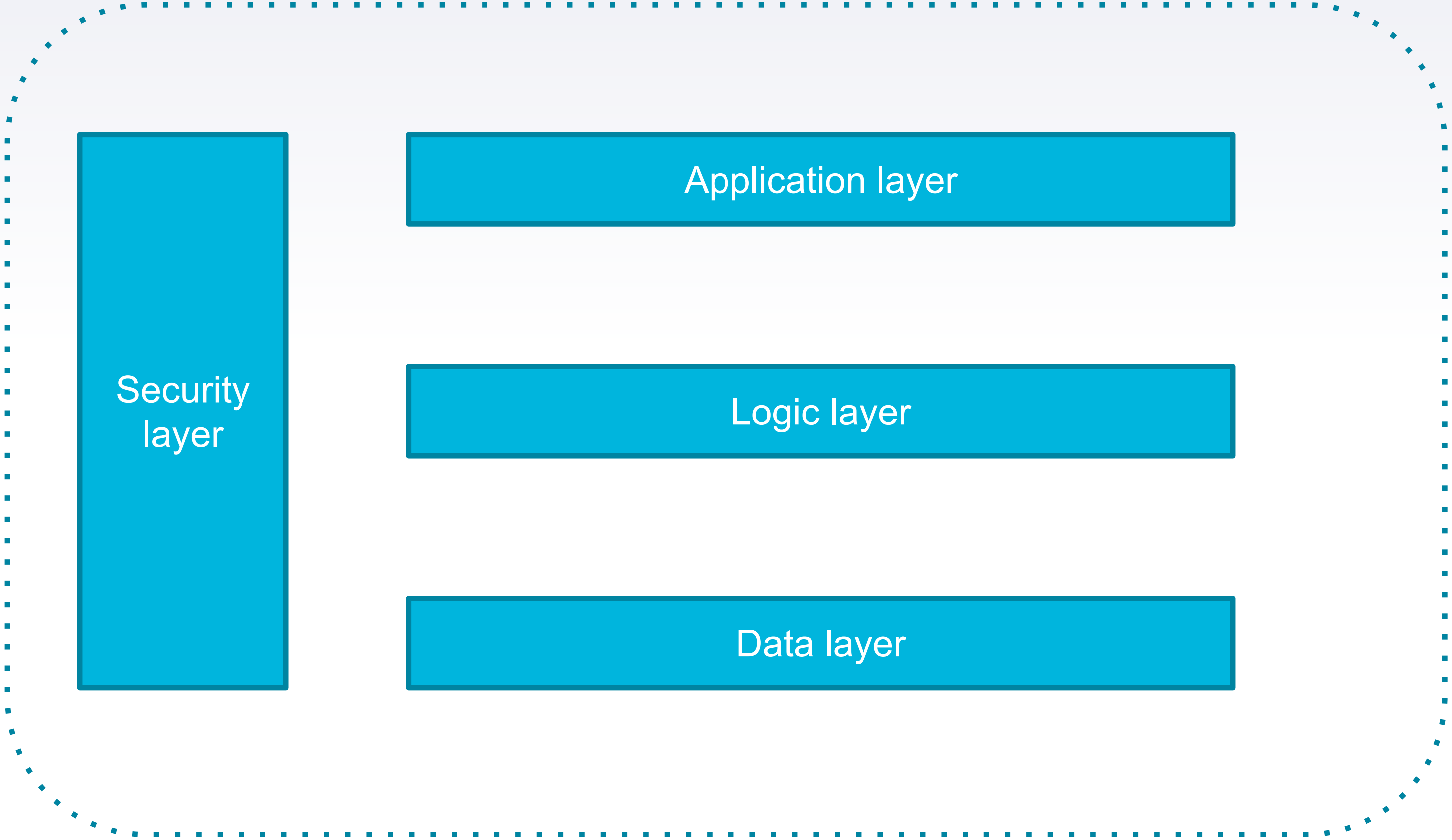
Software engineer at valeo

ITIan intake 39

Past ACMer

## Why this course!?

## Design a system:

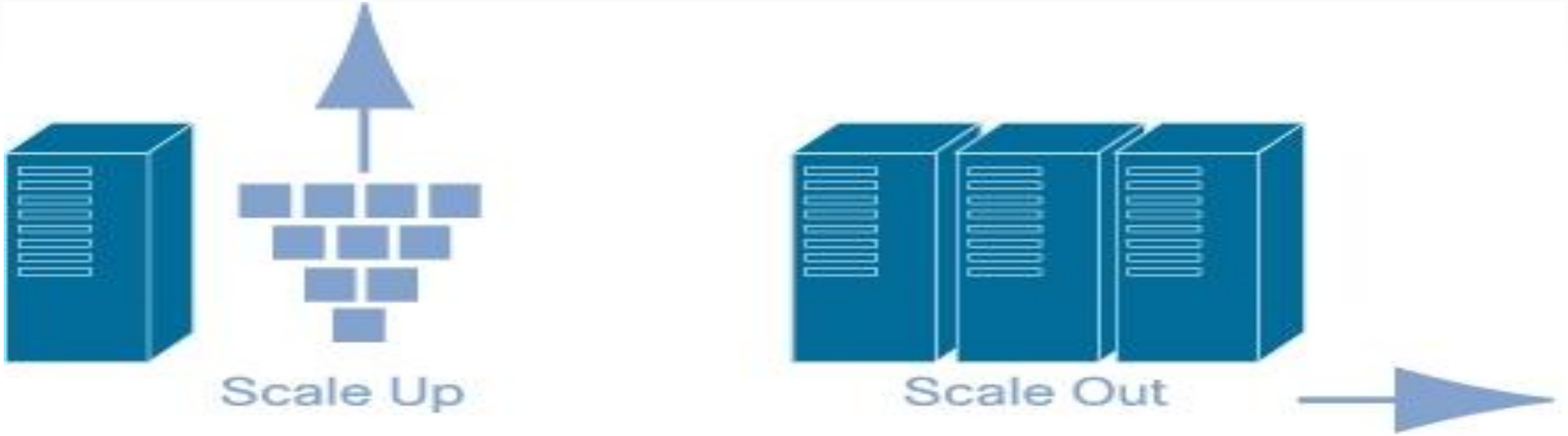


مين Now software architect؟

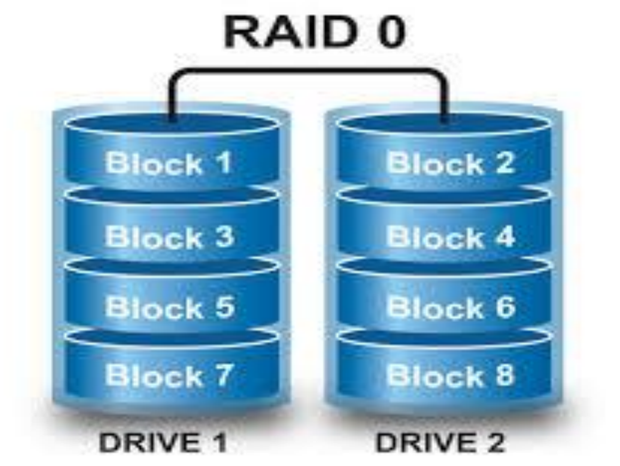
## scalability

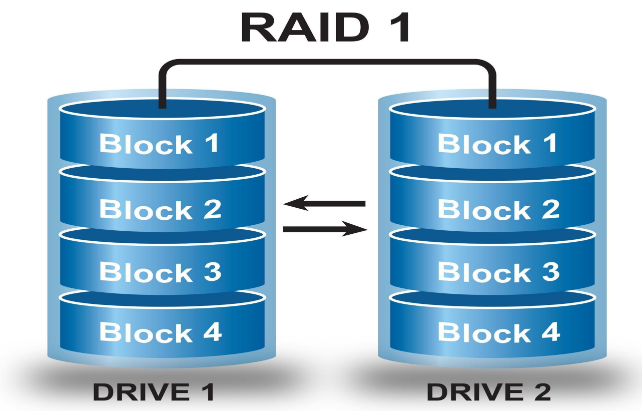
- **What features should we look at in any web host?**
  - On premises or remote or cloud service.
  - Supports secured protocols or not. most of providers support now.
  - How many request you can handle at a single shot (cpu cores).
  - Shared hosting can be contaminated for you and other customers.
  - storage latency (to the level of cables and the hard itself).
  - programming environments allowed.
  - cost
- **What about using Virtual private servers?**
  - VPS shared but with your view of the operating system.(amazon EC2 pay per minute)

- Scaling if you run out of resources (ram / hard disk / cpu ) ?(horizontal / vertical)

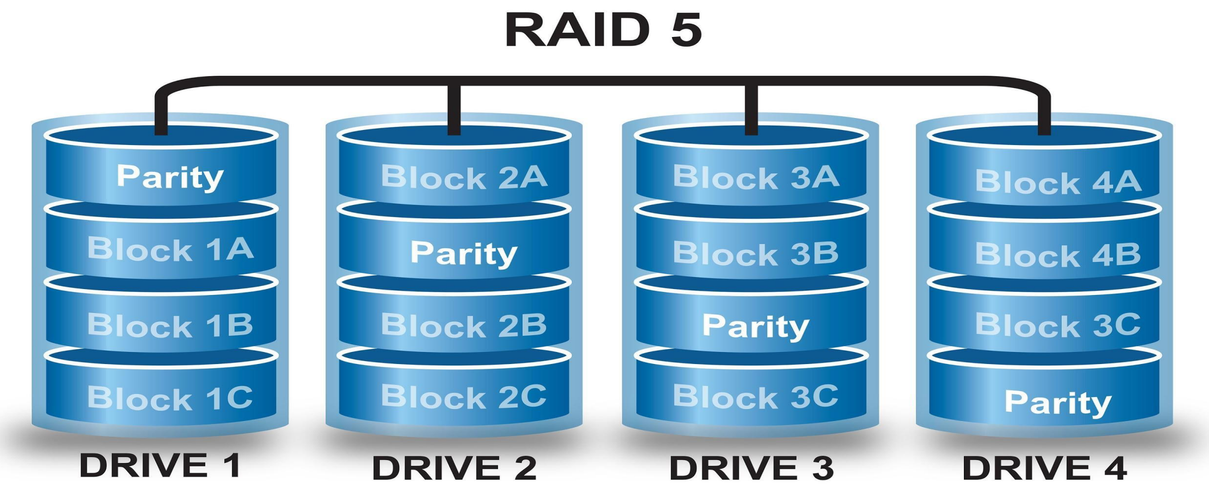


- **RAID technology “Redundant Array of Inexpensive Disks”**



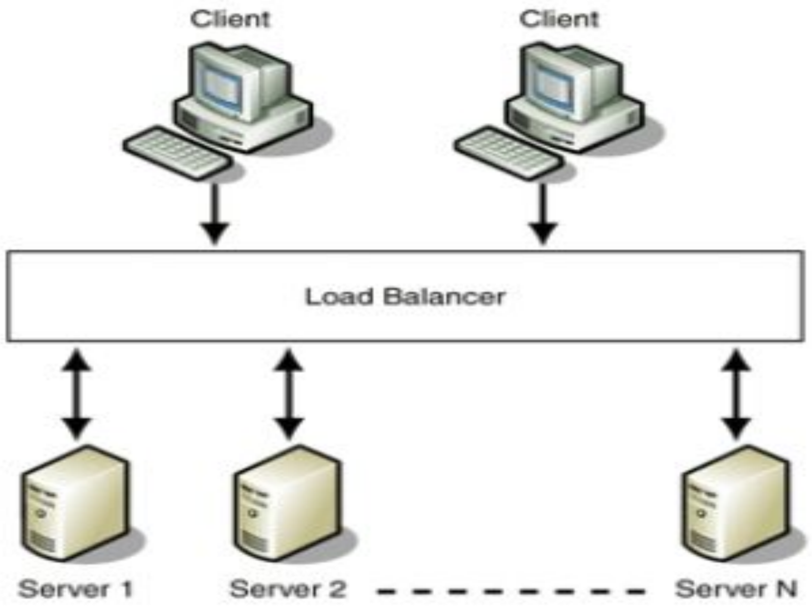


Mirrored Data to both Drives



Parity Across All Drives

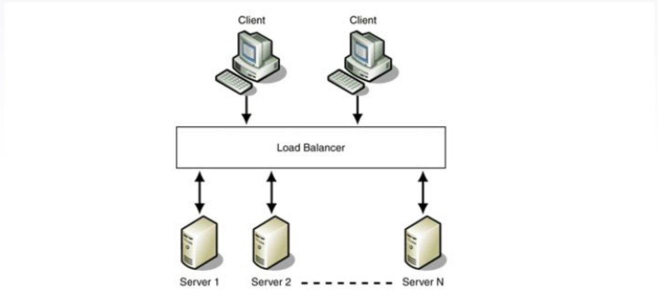
- **Distribute the http requests over vertical servers ?** load balancer



- The dns should return the ip address of which machine?

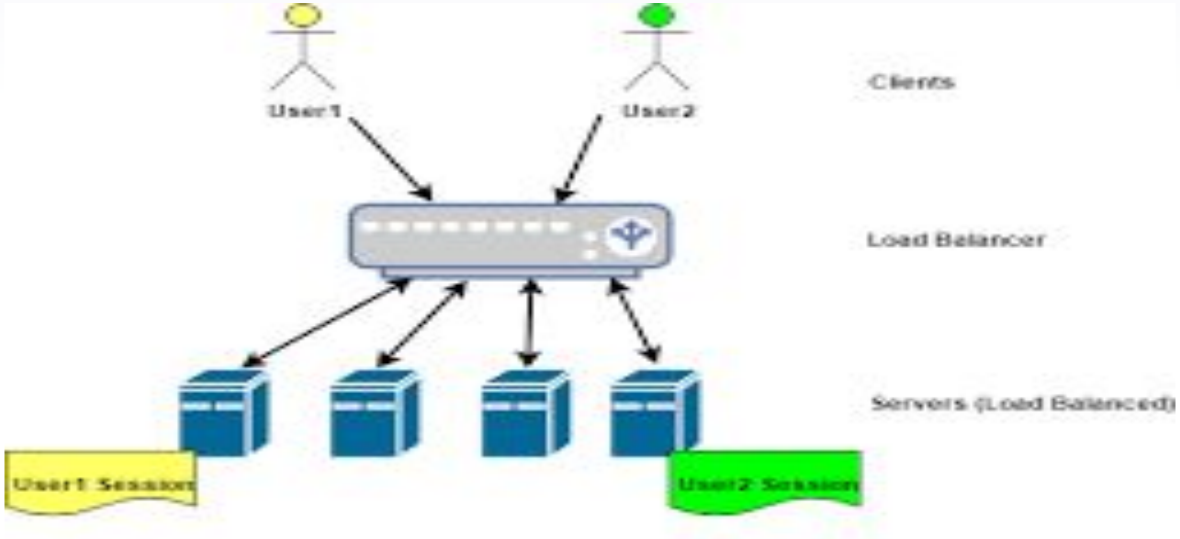
- Load balancing with bind (based on dns + round robin mechanism)
  - simple
  - high cpu usage can go to one server while others take low weight users

- other load balancing techniques:
  - report availability to load balancer.
  - load balance based on heuristics.



- what about sessions with load balancers in the system?!!!
  - Different server can be assigned for the same user -> session details will be lost.

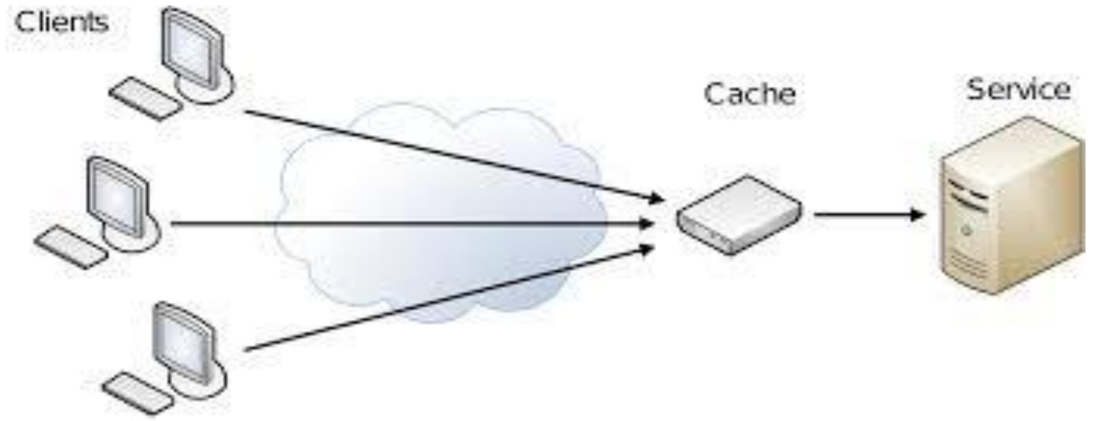
- how to maintain the session details for the user?
  - shared state server. -> high load -> RAID
  - caching in the load balancer.
  - cockies.



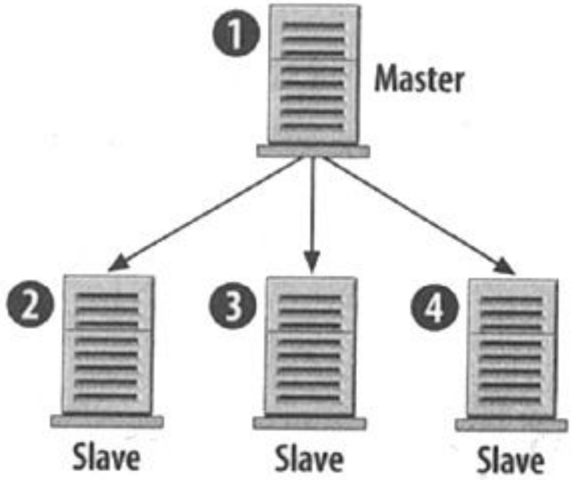
Sticky Session

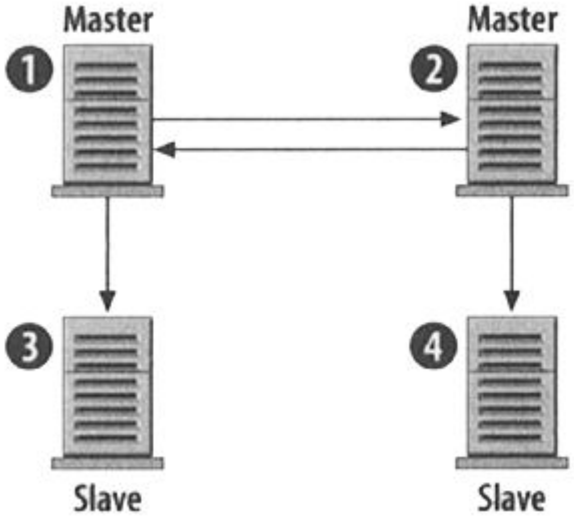
- **Cashing in load balancer or as a stand alone layer can  improve the performance:**
  - cash generated html files.
  - cash frequent queries to the database.

- **Cashing can be done in the memory:**
  - Redis.
  - Memcash.

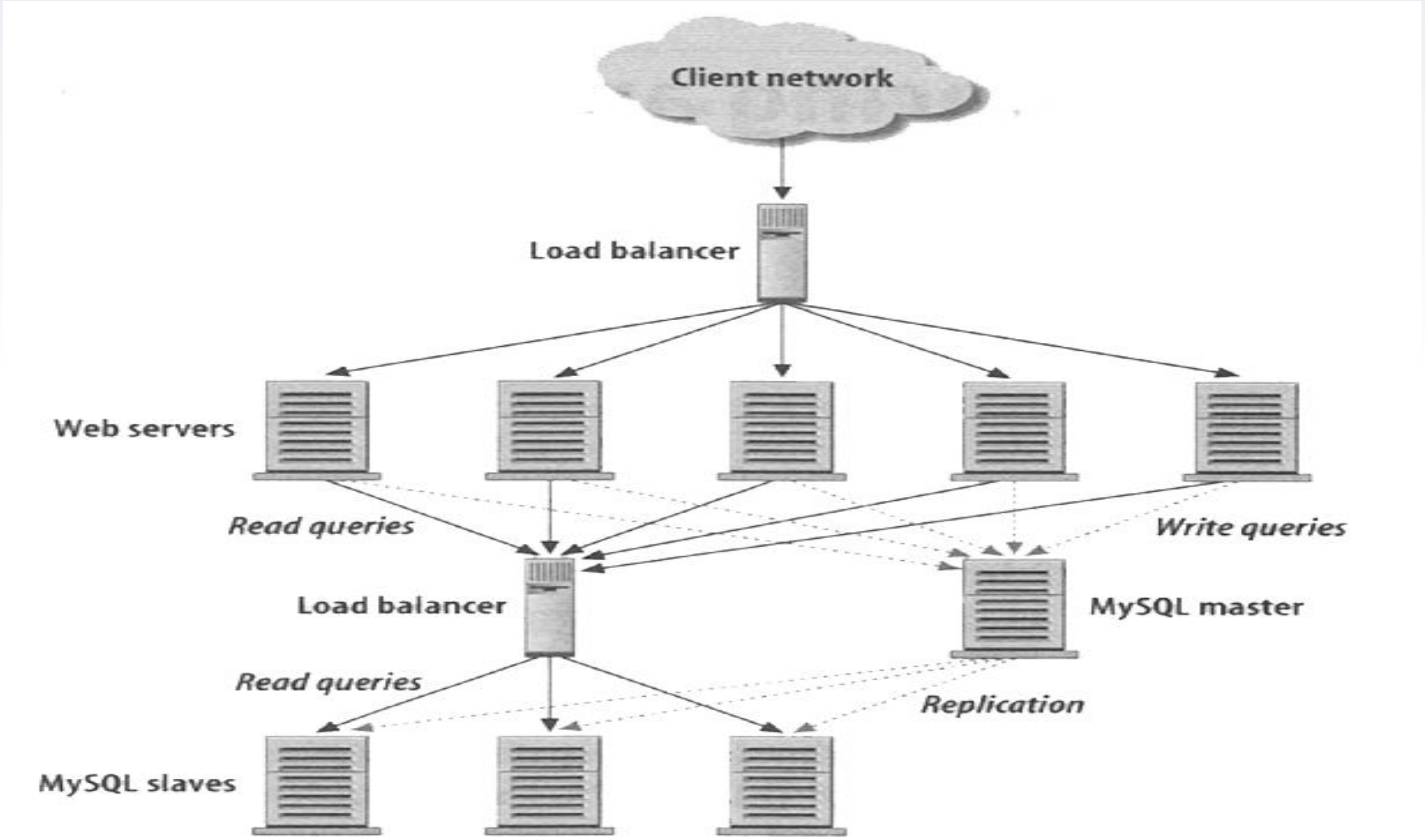


- **Replication is used to save data:**
  - **primary/secondary**
  - **primary-primary**

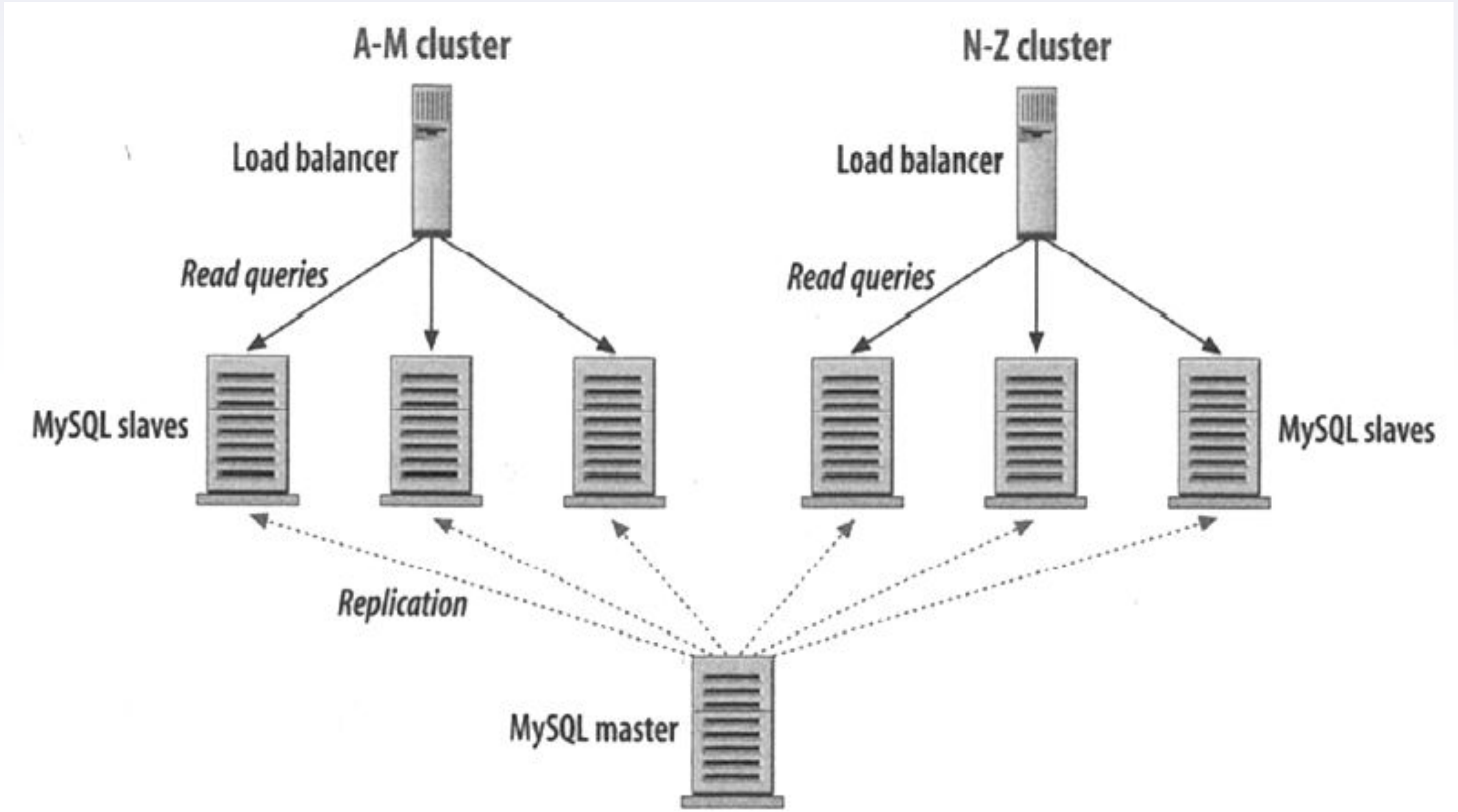




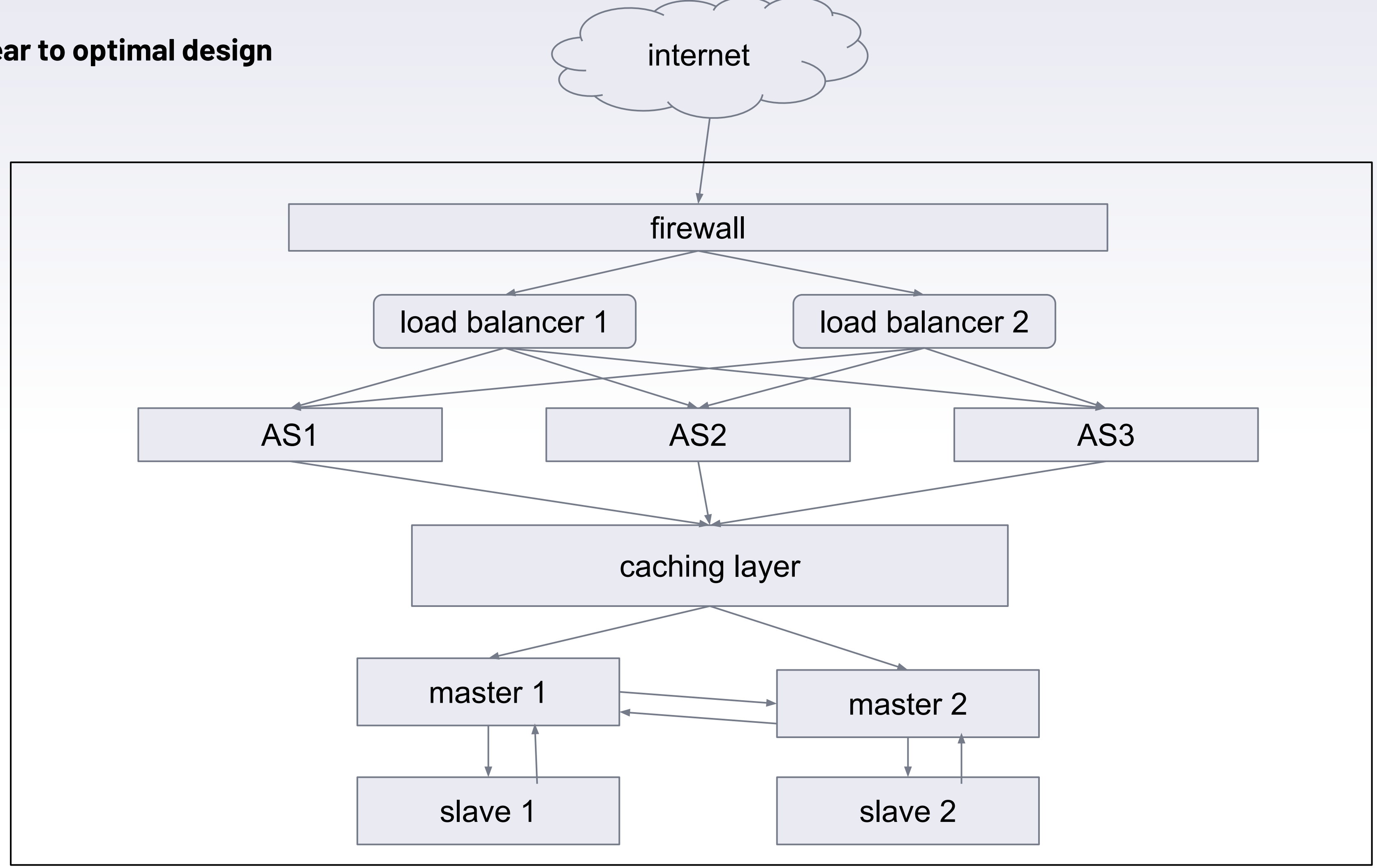
## load balancing with replication



## load balancing with replication + partitioning



## near to optimal design

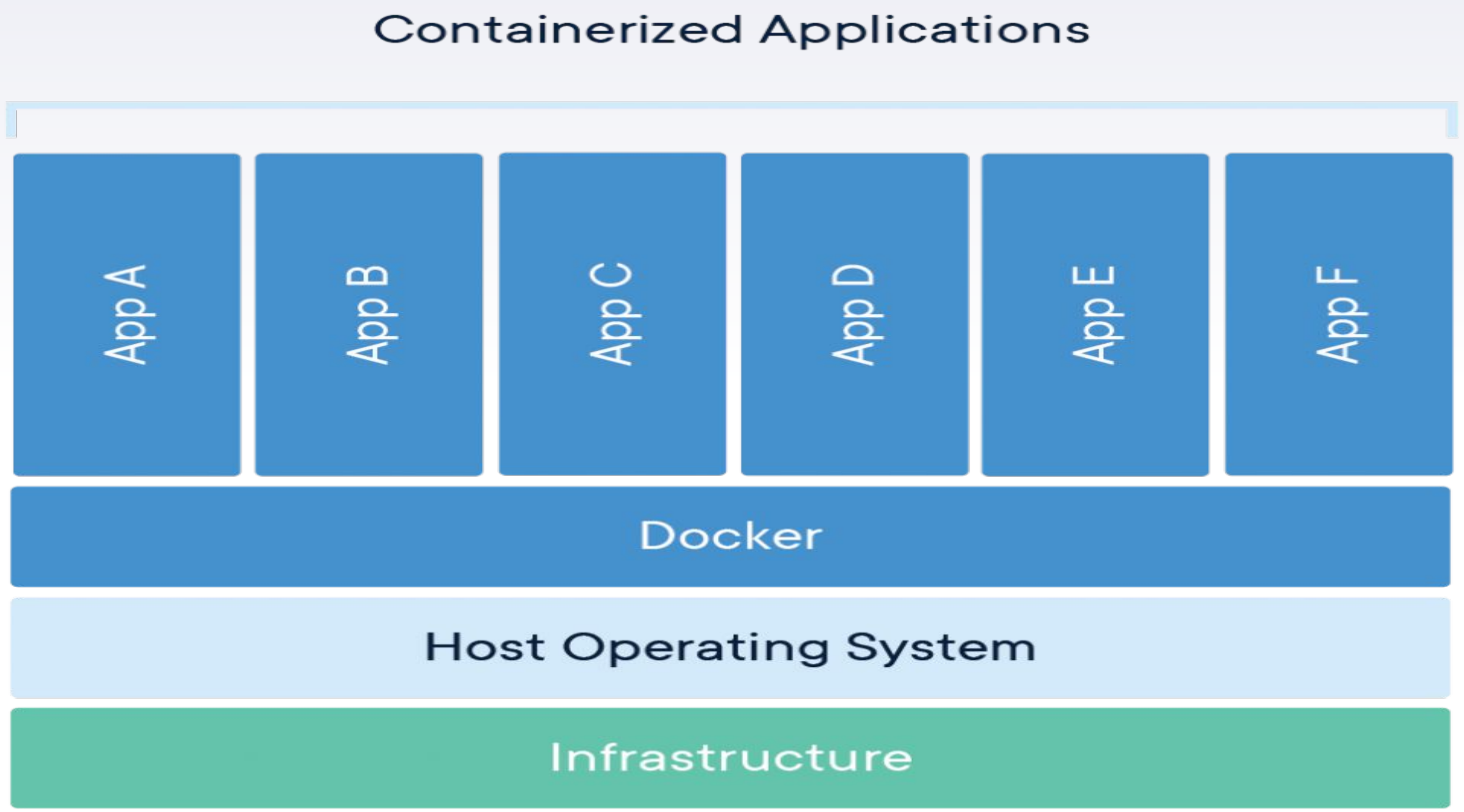


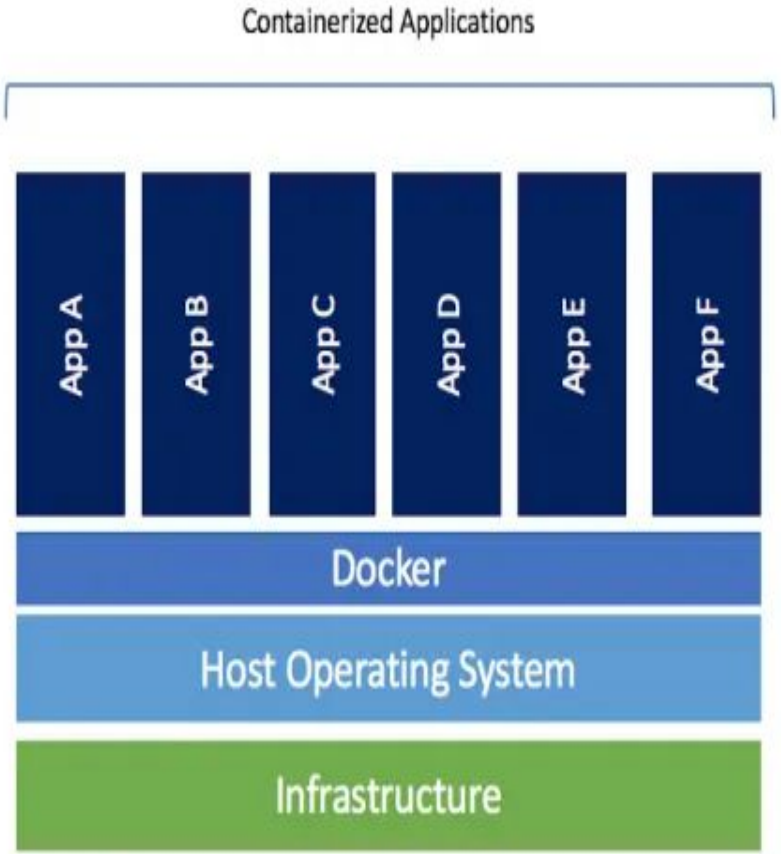
## CAP theorem

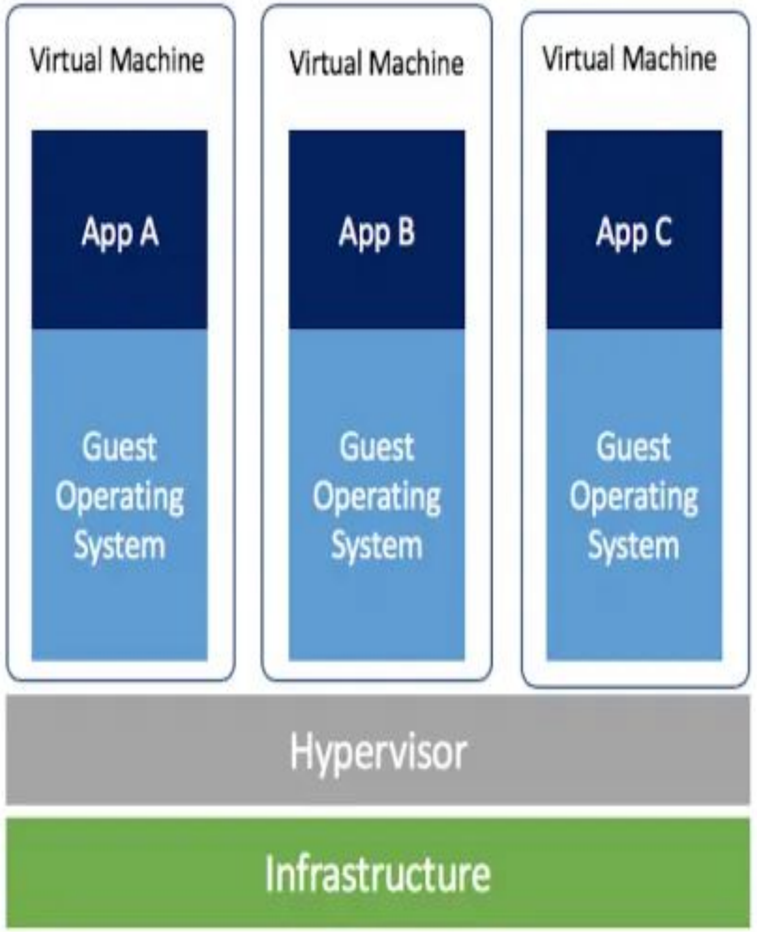
- Consistancy.
- availability.
- partition tolerance.

remembrance inc

## containers

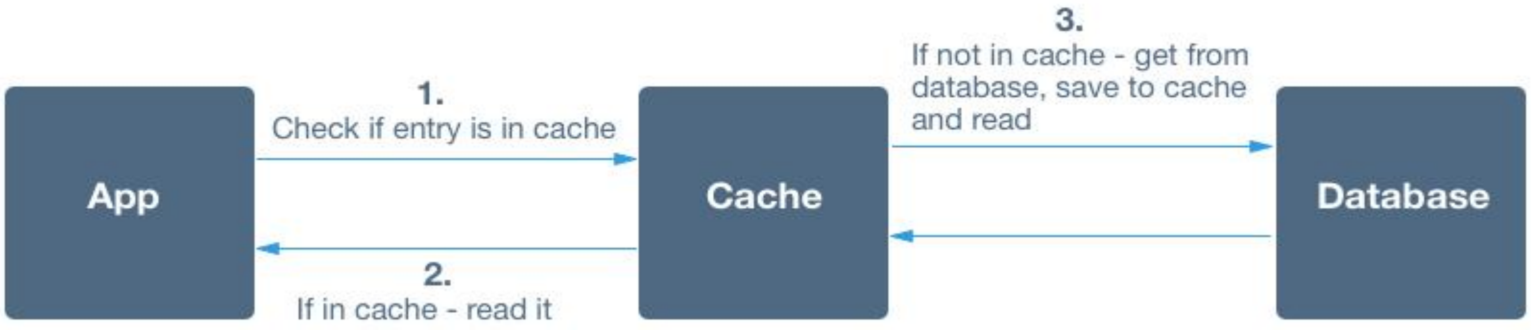

Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure

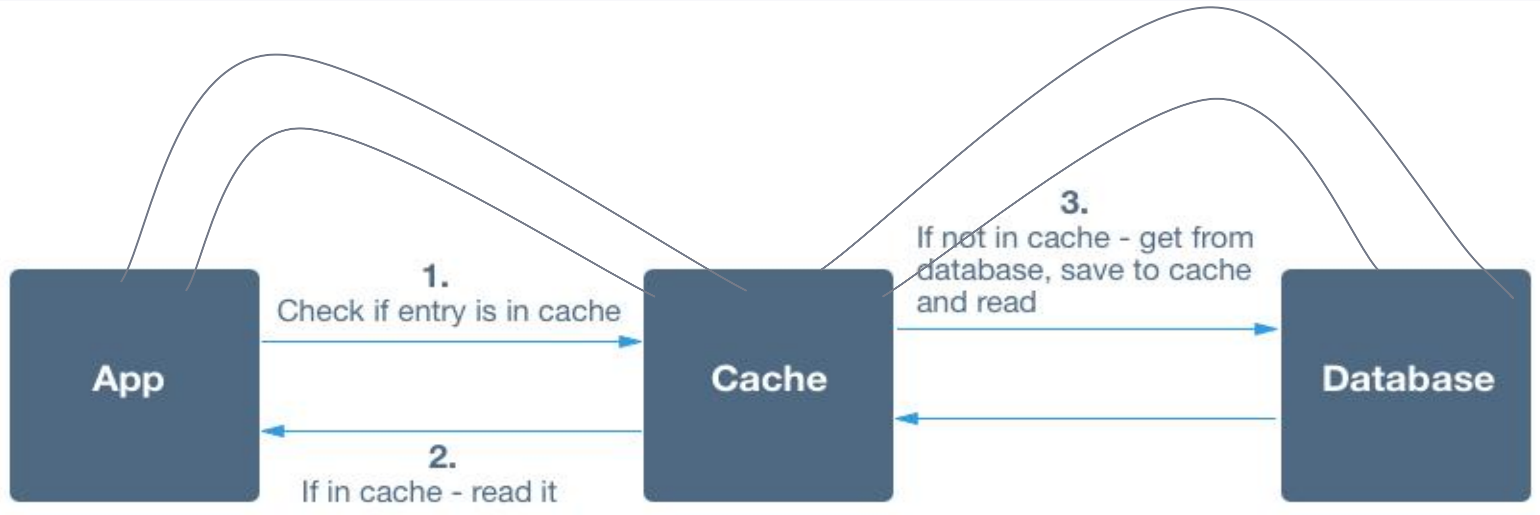

Virtual Machine
Virtual Machine
Virtual Machine
App A
App B
App C
Guest Operating System
Guest Operating System
Guest Operating System
Hypervisor
Infrastructure

## Design caching system

## Problem :


App
1.
Check if entry is in cache
Cache
2.
If in cache - read it
3.
If not in cache - get from database, save to cache and read
Database

## Problem :


1.
Check if entry is in cache
App
Cache
Database
2.
If in cache - read it
3.
If not in cache - get from database, save to cache and read

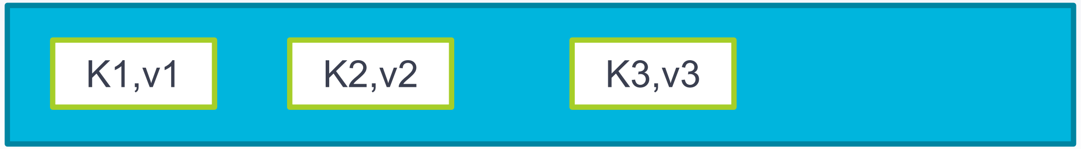
## Features:

- **What is the amount of data that we need to cache?**
    - cache on the scale of **Google** or Twitter. The total size of the cache would be a few **TBs.**

- **What should be the removal strategy? when should you move a cached item from memory?**
    - **LRU least recently used**

- **What should be the access pattern for the given cache?**
    - Write through cache.
    - Write around cache - cache aside write.
    - Write back cache.

## Features:



1.Add k1,v1

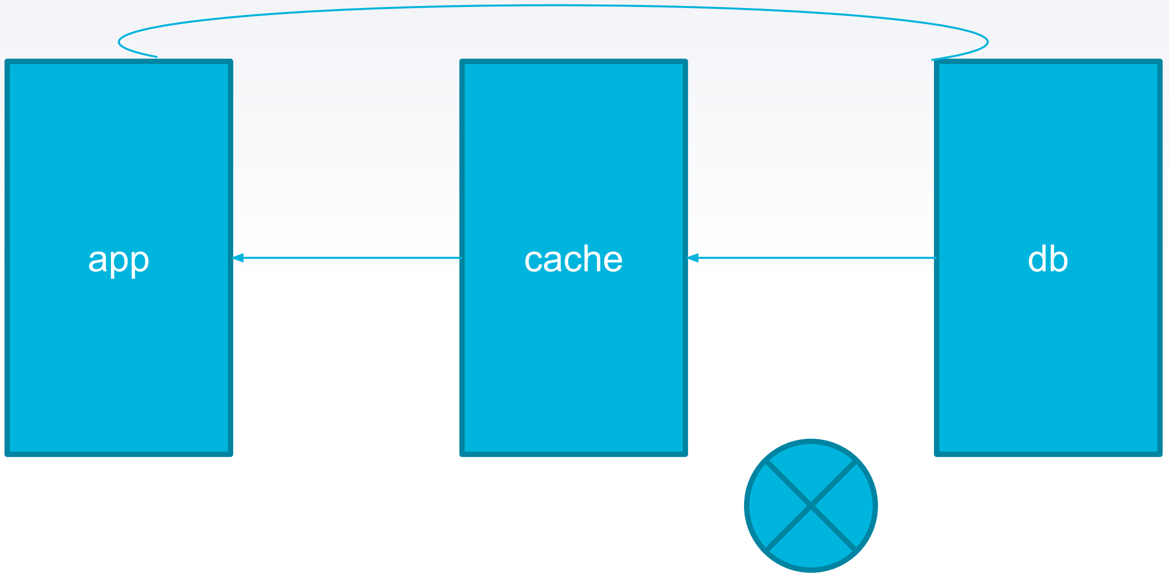
2.Add k2,v2

3.Add k3,v3

4.retreive k1,v1

5. Add k4,v4

## Features:



Eventual consistency

At 12 am data will be consistent

## Estimation :

- **What is the kind of QPS we expect for the system?**
  - 1M QPS
  - 10 M QPS **this is a better assumption in case of google or twitter**.

- **How many machines needed to meet the estimated requests per second?**
  - If you have 30 TB of data that would be needed to be cached at the same time and a machine can have 72 GB of RAM so you need **420 machine minimum**.

## Design goals :

- Latency : Are requests with high latency and a failing request, equally bad?
- Consistency : is eventual consistency allowed or it’s sensitive to high consistency.
- Availability : should the system be 100% available?

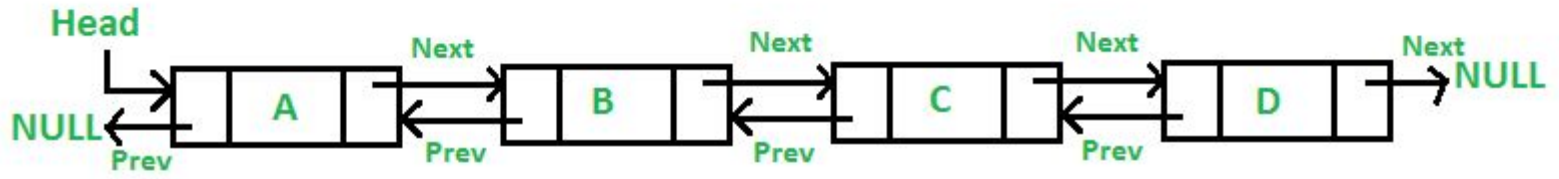
- Is Latency a very important metric for us?
  - sure

- Consistency vs Availability?
  - choose availability.

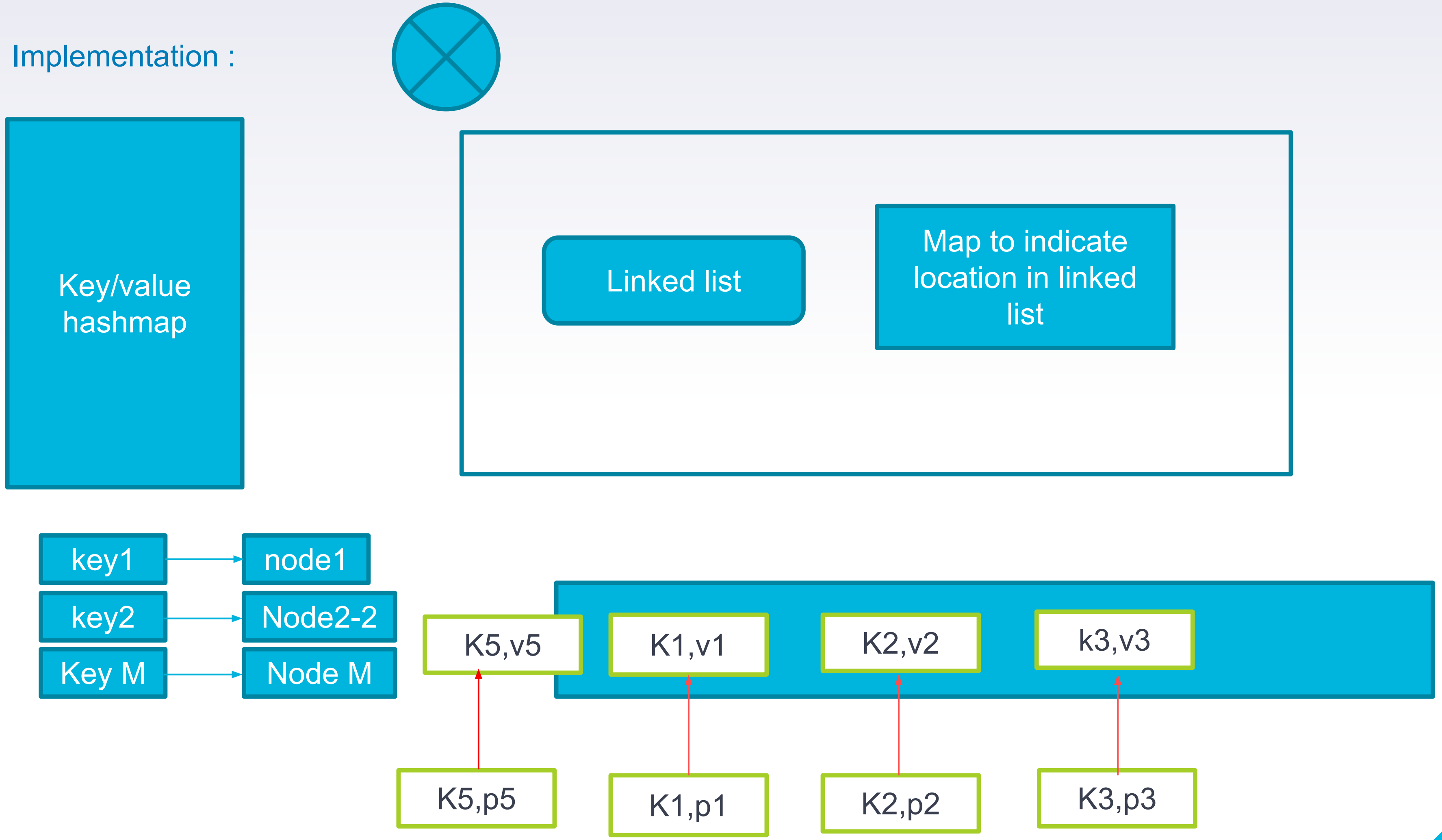
## Implementation :

- **If you will not run out of memory and single thread machine what data structure should we use?**
    - map/hashmap

- **If we will run out of memory so removals need to be done so what data structure should we use?**
    - we need ordering based on usage time.
    - use doubly linked list and map to addresses in the linked list.(**how insert and delete operations will look like?**)



## Implementation :



## Implementation :

- What about the implementation when dealing with multiple threads?
- **How would you break down cache write and read into multiple instructions?**

**Read path : Read a value corresponding to a key. This requires :**

- Operation 1 : A read from the HashMap and then,
- Operation 2 : An update in the doubly LinkedList

**Write path : Insert a new key-value entry to the LRU cache. This requires :**

- If the cache is full, then
  - Operation 3: Figure out the least recently used item from the linkedList
  - Operation 4: Remove it from the hashMap
  - Operation 5: Remove the entry from the linkedList.
  - Operation 6: Insert the new item in the hashMap
  - Operation 7: Insert the new item in the linkedList.

## Implementation :

- **How would you prioritize above operations to keep latency to a minimum for our system?**
  - operation one must be with most priority as it supports low latency (our design goal).
  - operation one competes with operation 4,operation 6

- **How would you implement HashMap?**
  - hashing with linked list
    - hashmap size is N and we wish to add (k,v) to it
    - Let H = size N array of pointers with every element initialized to NULL
    - For a given key k, generate g = hash(k) % N
    - newEntry = LinkedList Node with value = v
    - newEntry.next = H[g]
    - H[g] = newEntry

**instead of having a lock on a hashmap level, we can have it for every single row. This way, a read for row i and a write for row j would not affect each other if i != j**

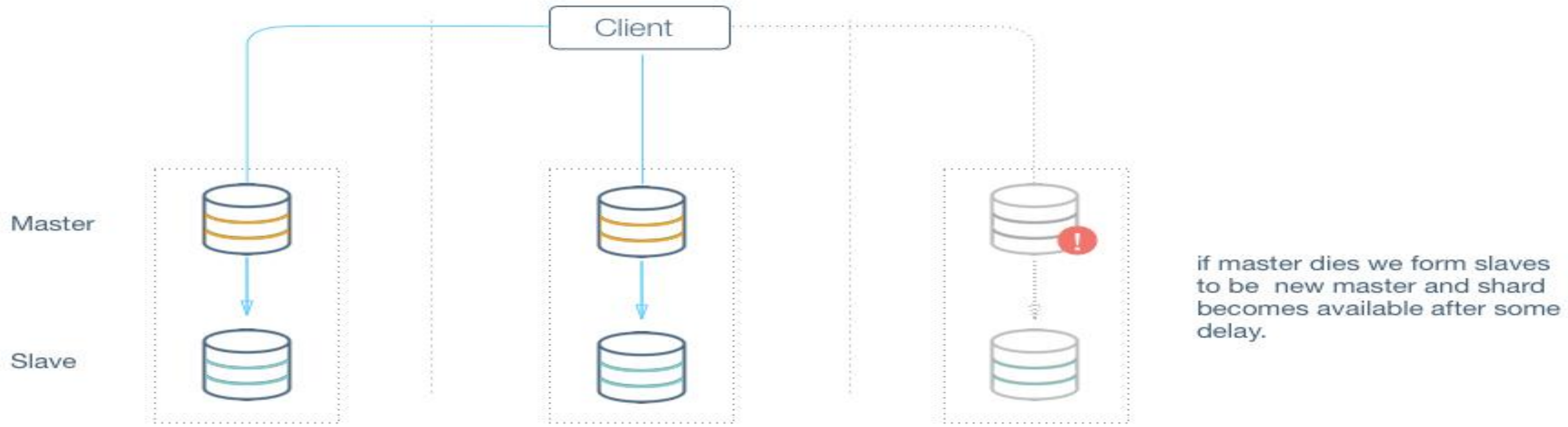
## Implementation :

- **What QPS would a machine have to handle if we shard in blocks of 72GB?**
  - around 23000 QPS
  - CPU time available for 23k queries : 1 second * 4 = 4 seconds
  - CPU time available per query = 4 * 1000 * 1000 / 23000 microseconds = 174us
  - with a perfectly written asynchronous server, we would have much less than 174us on our hand

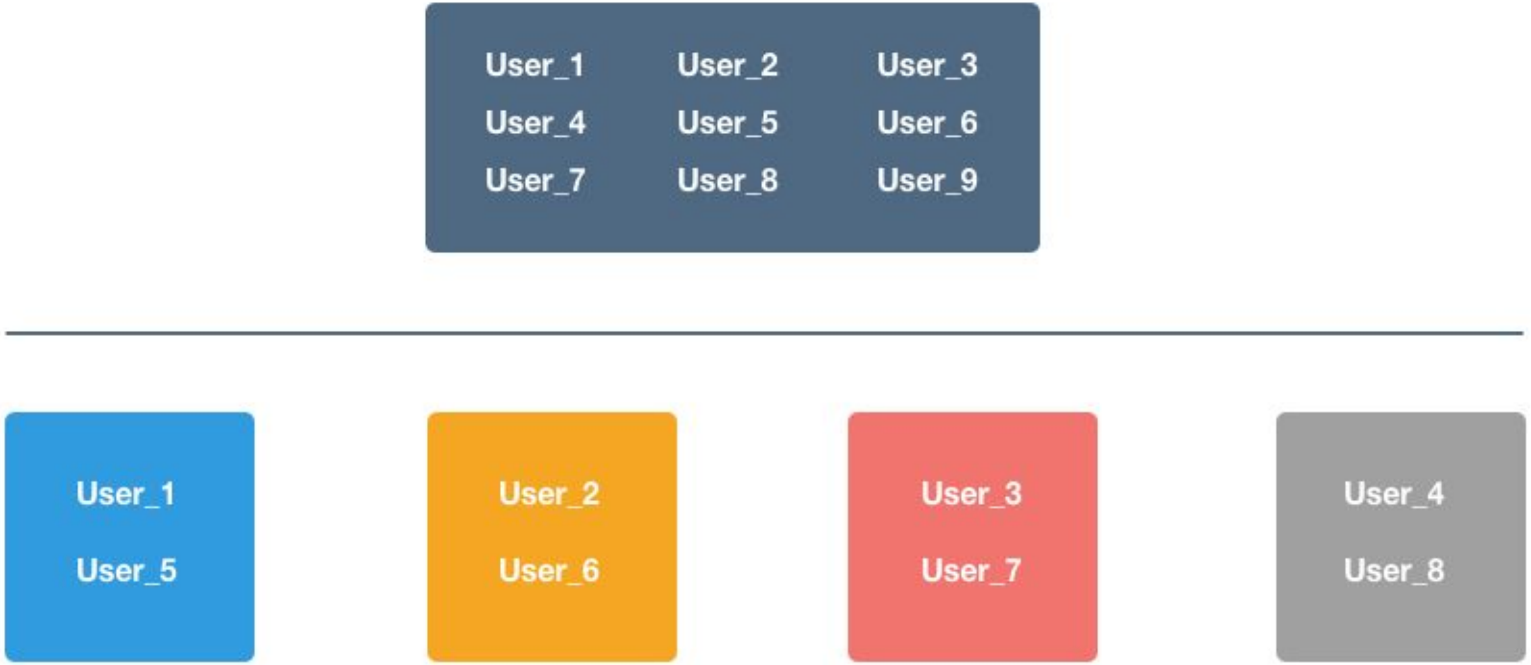
- **What if we shard among machines with 16GB of RAM?**
  - hashing function will need to be changed as the number of n elements will change(this is a problem in the hashing mechanism)

## Implementation :

- **What happens when a machine handling a shard goes down?**
  - **r**equests will fail in cach and go to database which will increase the latency
  - replication is required.


Client
Master
Slave
if master dies we form slaves to be new master and shard becomes available after some delay.

## Design a sharding mechanism for database


User_1
User_2
User_3
User_4
User_5
User_6
User_7
User_8
User_9
User_1
User_5
User_2
User_6
User_3
User_7
User_4
User_8

## Features:

- **What is the amount of data that we need to store?**
  - 100 TB.
- **Will the data keep growing over time? If yes, then at what rate?**
  - Yes. At the rate of 1TB per day.
- **What are the available machines and what are their specs?**
  - 20 machines each have a RAM of 72G and a hard disk capacity of 10TB.
- **Are all key value entries independent? No relations?**
  - Yes. A typical query would ask for value corresponding to a key.

## Estimation :

- **What is the minimum number of machines required to store the data?**
    - minimum of 100TB / 10 TB = 10 machines to store the said
- **How frequently would we need to add machines to our pool ?**
    - we generate data that would fill the storage of 1 machine ( 10TB ) in 10 days. Assuming, we want to keep a storage utilization of less than 80%, we would need to add a new machine every 8 days.

## Implementation :

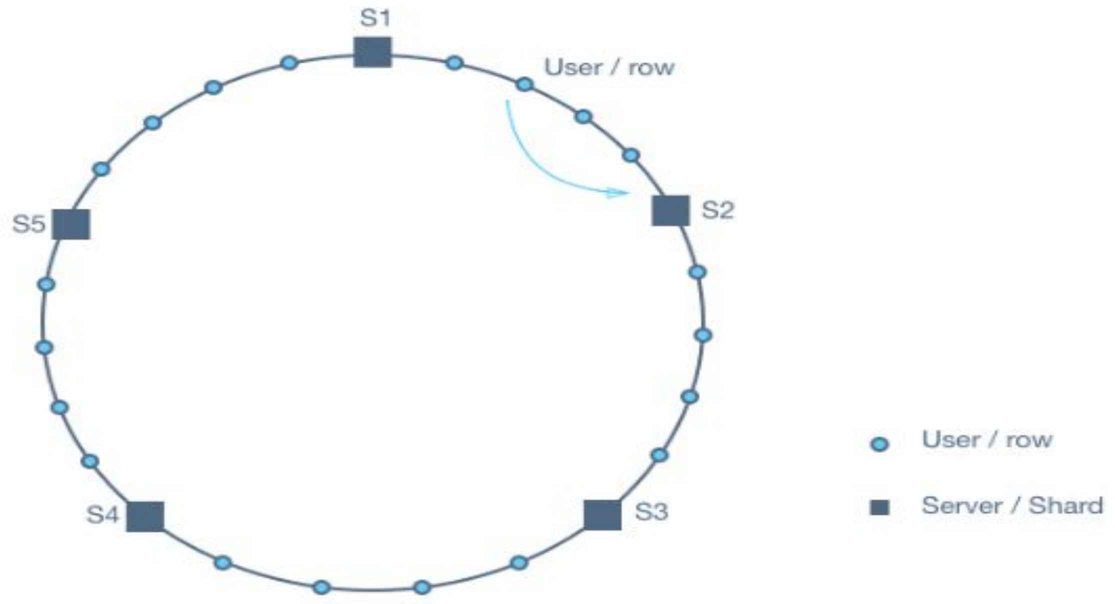
- **Can we have a fixed number of shards?**
  - shard = one machine.
  - data within a shard will keep growing and exceed the 10TB mark we have set per machine
    **So no we shouldn’t have fixed number of shards.**

- How many shards do we have and how do we distribute the data within the shard?
  - Lets say our number of shards is S = 5.
  - we calculate a numeric hash H = hash(key=‘ay haga’) = 14, and assign the key to the shard corresponding to H % S = 14 % 5 = 4. 0-4
  - the number of shards will have to increase. And when it does, our new number of shard becomes S+1.
  - As, such H%(S+1) changes for every single key causing us to relocate each and every key in our data store.

**bad approaches**

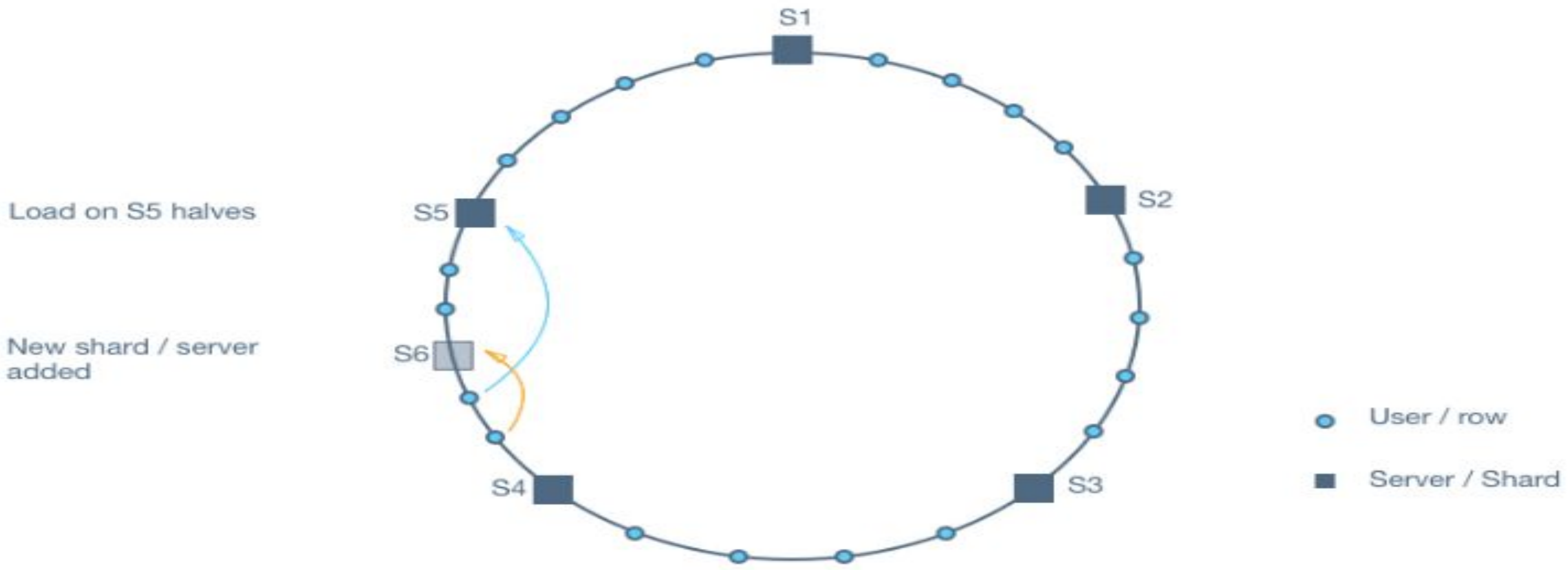
## Implementation :

**Consistent Hashing.**
we calculate a 64 bit integer hash for every key and map it to a ring. Lets say we start with X shards. Each shard is assigned a position on the ring as well. Each key maps to the first shard on the ring in clockwise direction.



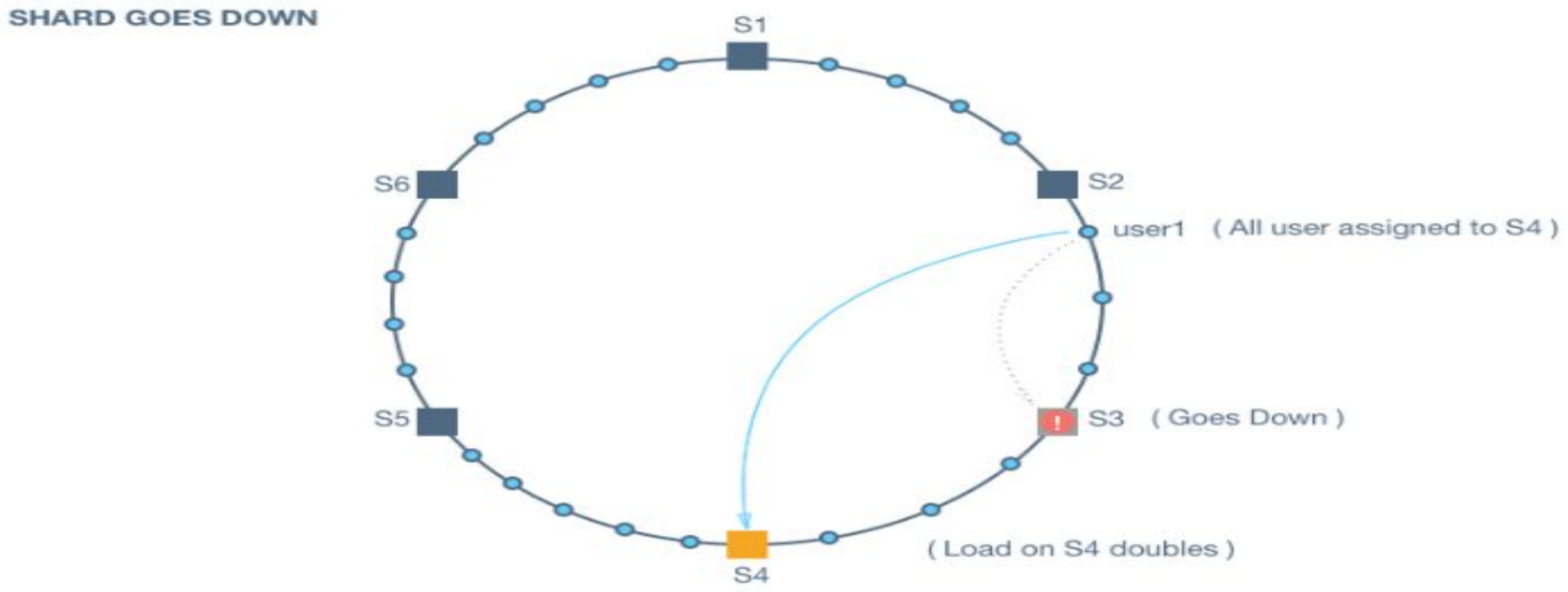
## Implementation :

What happens if we need to add another shard ? Or what if one of the shard goes down and we need to re-distribute the data among remaining shards?



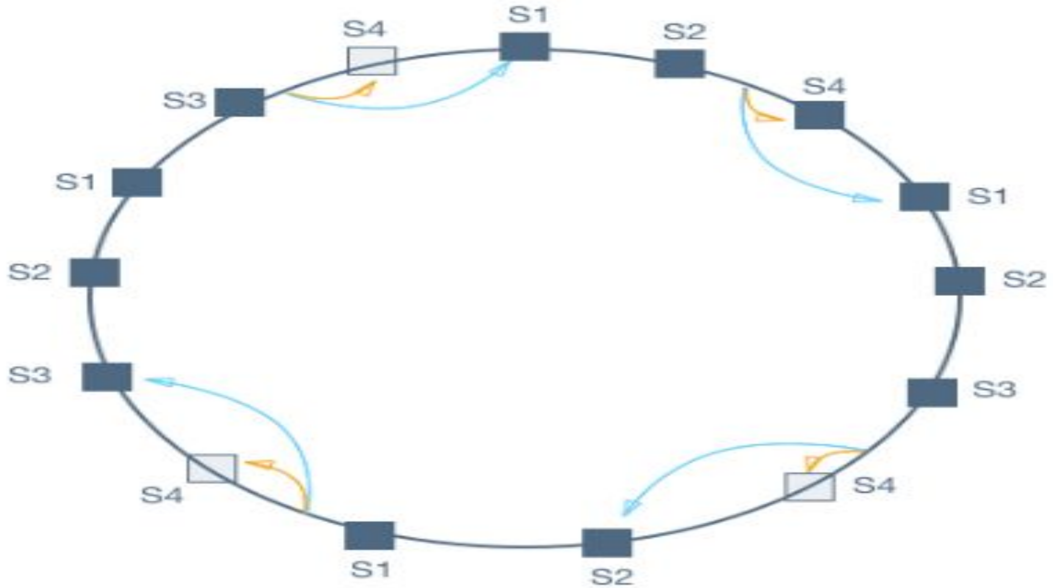
## Implementation :

Similarily, there is a problem of cascading failure when a shard goes down



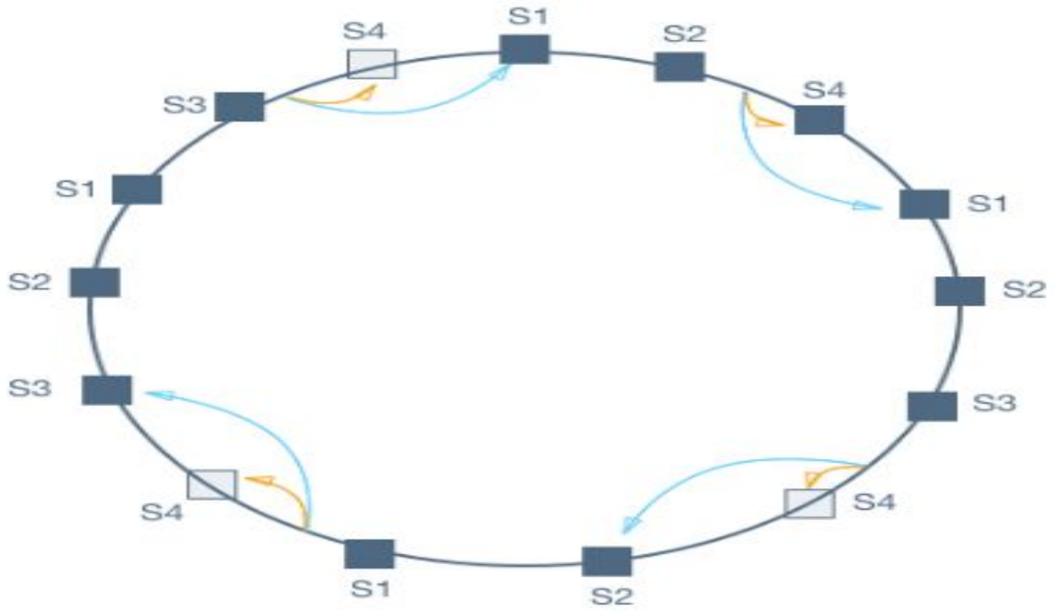
## Implementation :

**Modified consistent hashing:**
What if we slightly changed the ring so that instead of one copy per shard, now we have multiple copies of the same shard spread over the ring.



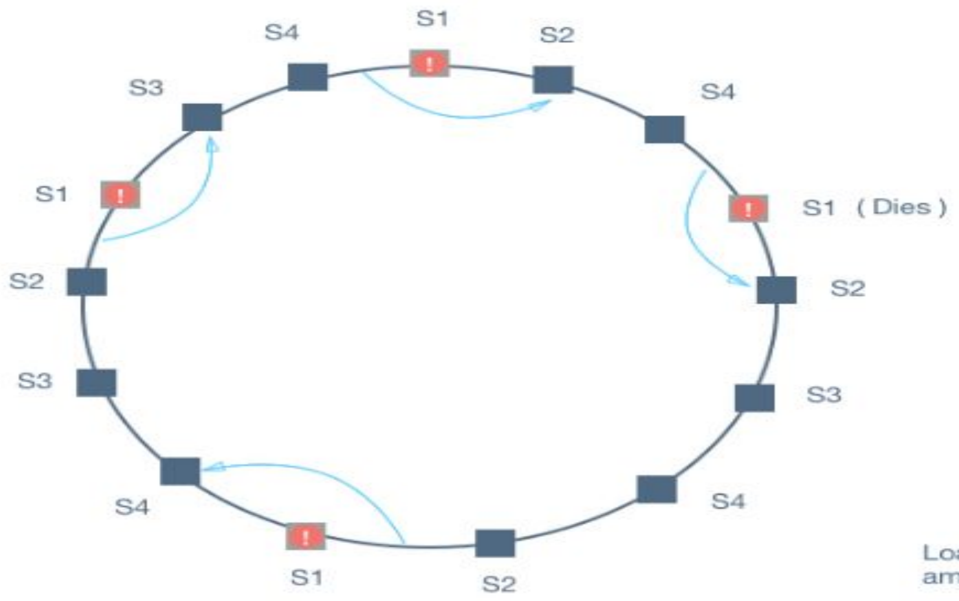
## Implementation :

Case when new shard is added



## Implementation :

Case when a shard goes down : No cascading failure. Yay!



Load is distributed almost equal amongst remaining server / Shard

## day 1 assignment

design a load balancer

## Day 2

## Design a highly available database

## Features:

- **What is the amount of data that we need to store?**
    - Let's assume a few 100 TB.

- **Do we need to support updates?**
    - Yes.

- **Can the size of the value for a key increase with updates?**
    - Yes with updates.

## Estimation :

- **Can a value be so big that it does not fit on a single machine?**
  - No. Assume upper cap of 1GB to the size of the value.

- **What would the estimated QPS be for this DB?**
  - Let's assume around 100K.

- What is the minimum number of machines required to store the data?
  - a machine has 10TB of hard disk, we would need minimum of 100TB / 10 TB = 10 machines (**minimum**)

## Design goals :

- Latency : Are requests with high latency and a failing request, equally bad?
- Consistency : is eventual consistency allowed or it’s sensitive to high consistency.
- Availability : should the system be 100% available?

- **Is Latency a very important metric for us?**
    - Since we want to be available all the time, we should try to have lower latency.

- Consistency vs Availability?
    - availability.
    - we should aim to have eventual consistency.

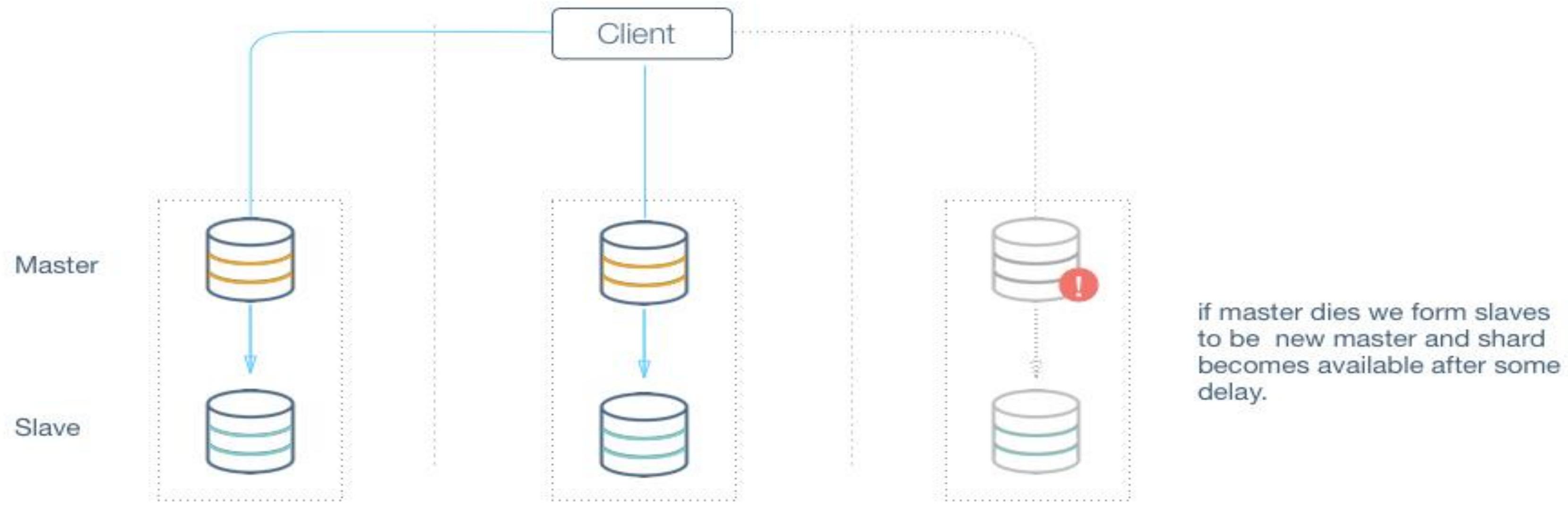
## Implementation :

- **Is sharding required?**
  - we have 100Tb -> can’t be stored on a single machine.even if stored on a single machine it would be required to handle a lot of requests which will lead to unavailability at some time.
  - sharding is a must.

- **Should the data stored be normalized?**
  - no redundancy.
  - joins will be required.
  - joins can require communication between two different shards -> high latency.
  - normalization was made to save memory.
  - replicating data fields is bad for memory but good for latency.

## Implementation :

- **Should we shard the data? How the architecture of each shard should look like?**
  - **Master/slave**


Client
Master
Slave
if master dies we form slaves to be new master and shard becomes available after some delay.

  - **Multi Master system can lead to inconsistency ((x*2)+1,(x+1)*2)**

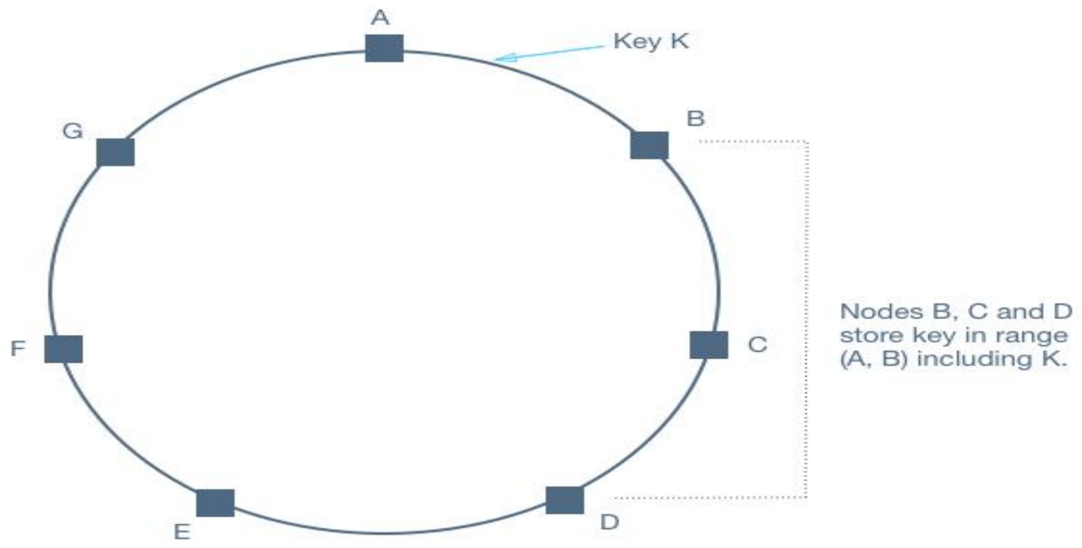
## Implementation :

- What is peer to peer system?
  - all machines can accept reads/writes.
  - all machines can communicate with each other.
  - cassandra and dynamo db are examples of this technology.

- How will we exactly shard data for a peer to peer system?
  - the same as the sharding system.

## Implementation :

- **How do we store redundant data in a sharded system?**
  - duplicate data among p nodes p is a replication factor chosen (ex p=3)
  - client can read/write any key in range of a..b from any of the next p nodes clockwise.



## Implementation :

- **How does a write/read happen in our system now?**
  - **write**:
    - user call any node on the ring to write the data.
    - the node will work as a coordinator for the data.
    - the node will find the nodes the data should be written into and write data into them.
    - the coordinator will return success write only when W acknowledgements returned.
  - **Read**:
    - user call any node on the ring to read the data.
    - the node will work as a coordinator for the data.
    - the node will find the nodes the data should be read from and read data from them.
    - the coordinator will return success read only when R repliesreturned.

- W and R are called write and read consistency number
- For a read to be consistent(return the latest write), we need to keep $W + R > P$.
- Depending on the feature requirement W and R can be adjusted
- To have very fast writes we can keep $W = 1$ and $R = P$
- If our system is read heavy we can keep $R = 1$ and $W = P$
- If read and write are equally distributed, we can keep both R and W as $(P+1)/2$

## Implementation :

- What if a machine goes down?
  - no node is responsible for data so no high effect as long as W,R are met correctly.
  - at in case of less than W nodes available to write for some data, we can relax our write consistency(sacrificing data consistency for availability).

- What kind of consistency can we provide?
  - If we keep W = P, we can provide strong consistency but we won't be available for some writes
  - in case of master/master nodes tell each other the data until they converge.
  - here all nodes are masters.
  - we can use eventual consistency concept here but some conflicts can happen
  - if conflict data is one value use last written
  - if augmented data and one is contained in the other use the parent.
  - else application logic should resolve the conflicts

## Design a URL shortener

## Features:

- **Shortening:** Take a url and return a much shorter url
- Input: https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.nawaret.com%2F%25D9%2585%25D9%2586%25D9%2588%25D8%25B9%25D8%25A7%25D8%25AA%2F%25D8%25AC%25D8%25B1%25D9%258A%25D8%25AF%25D8%25A9-%25D8%25A7%25D9%2584%25D9%2585%25D8%25B3%25D8%25AA%25D9%2582%25D8%25A8%25D9%2584-%25D8%25A7%25D9%2584%25D9%2584%25D8%25A8%25D9%2586%25D8%25A7%25D9%2586%25D9%258A%25D8%25A9-%25D8%25AA%25D9%2588%25D8%25AF%25D9%2591%25D8%25B9-%25D9%2582%25D8%25B1%25D9%2591%25D8%25A7%25D8%25A1&psig=AOvVaw2NIukHnPOxrtnTLELkSAvF&ust=1619342546237000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCMCKu5zHlvACFQAAAAAdAAAAABAJ
- output.: https://cutt.ly/1vMEej1
- output.: https://bit.ly/1vMEej
    - 2 states: 0,1
    - 4 states: 00,01,10,11
    - a1,daZ5
    - 1vMEej
    - https://t.co/key

## Estimation :

Assuming average lifetime of a shortened URL is 2 weeks and 20% of websites creates 80% of the traffic, we see that we'll receive around 1 Billion queries in a month.

- **How many queries per second should the system handle?(Assuming 100 Million new URLs added each month)**
    - 400 queries per second in total. 360 read queries and 40 write queries per second.

- **How many URLs will we need to handle in the next 5 years?**
    - Earlier we saw, we would see 100 Million new URLs each month. Assuming same growth rate for next 5 years, total URLs we will need to shorten will be 100 Million * 12 * 5 = 6 Billion.

## Estimation :

- **What is the minimum length of shortened url to represent 6 Billion URLs?**
  - We will use (a-z, A-Z, 0-9) to encode our URLs. If we call x as minimum number of characters to represent 6 Billion total URLs, then will be the smallest integer such that x^62 > 6*10^9. A: Log (6*109) to the base 62 = 6.

- **Data read/written each second?**
  - Data flow for each request would contain a shortened URL and the original URL. Shortened URL as we saw earlier, will take 6 bytes whereas the original URL can be assumed to take atmost 500 bytes.

- **How much data will we need to store so that we don't have to restructure our architecture for the next 5 years considering constant growth rate?**
  - 3 TeraBytes for URLs and 36 GigaBytes for shortened URLs Note that for the shortened URL, we will only store the slug(6 chars) and compute the actual URL on the fly.

## Design goals :

- Latency : Are requests with high latency and a failing request, equally bad?
- Consistency : is eventual consistency allowed or it's sensitive to high consistency.
- Availability : should the system be 100% available?

- **Is Latency a very important metric for us?**
    - Yes. Our system is similar to DNS resolution, higher latency on URL shortener is as bad as a failure to resolve.

- **Should we choose Consistency or Availability for our service?**
    - should be a tradeoff as both are important.

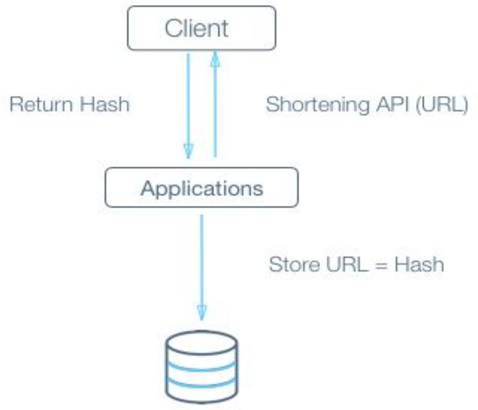
**URL shortener by definition needs to be as short as possible. Shorter the shortened URL, better it compares to competition**

## Implementation :

- **How should we compute the hash of a URL?**
  - ShorteningAPI(url) {store the url_mapping and return hash(url)}
  - RedirectionAPI(hash) {redirect_to url_mapping[hash]}

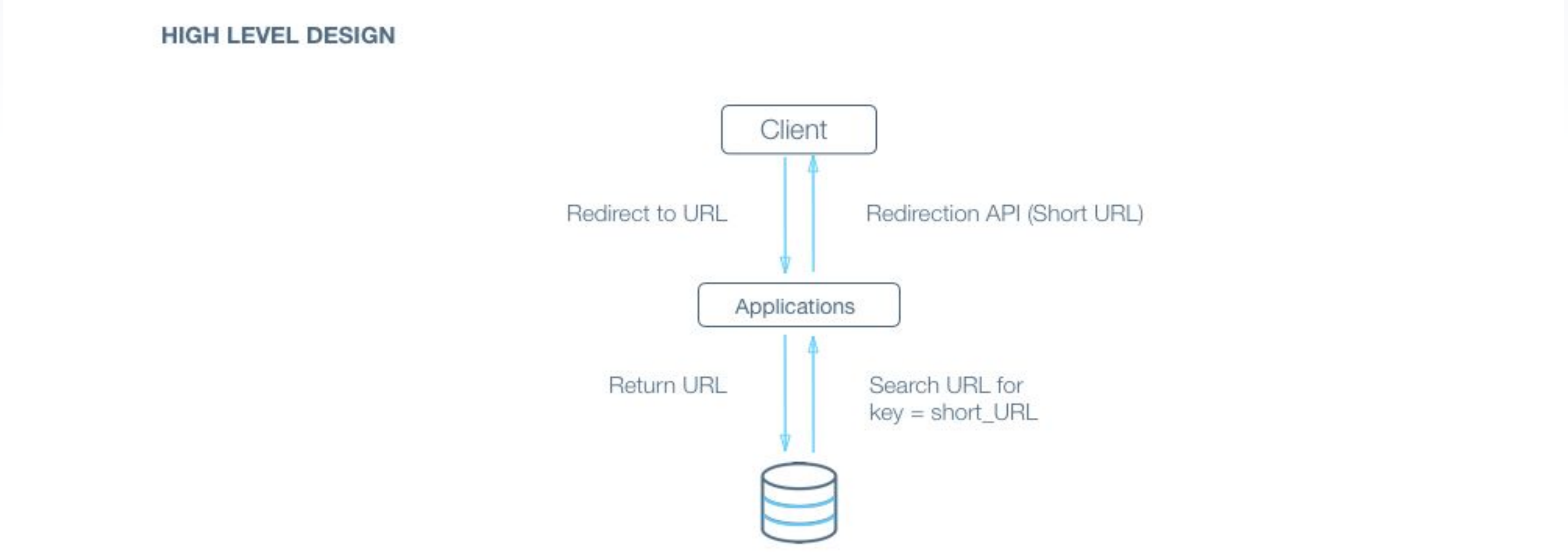
- How would a typical write query look like?
  - Client ( Mobile app / Browser, etc ) which calls ShorteningAPI(url)
  - Application server which interprets the API call and generates the shortened hash for the url
  - Database server which stores the hash => url mapping

HIGH LEVEL DESIGN (WRITE)



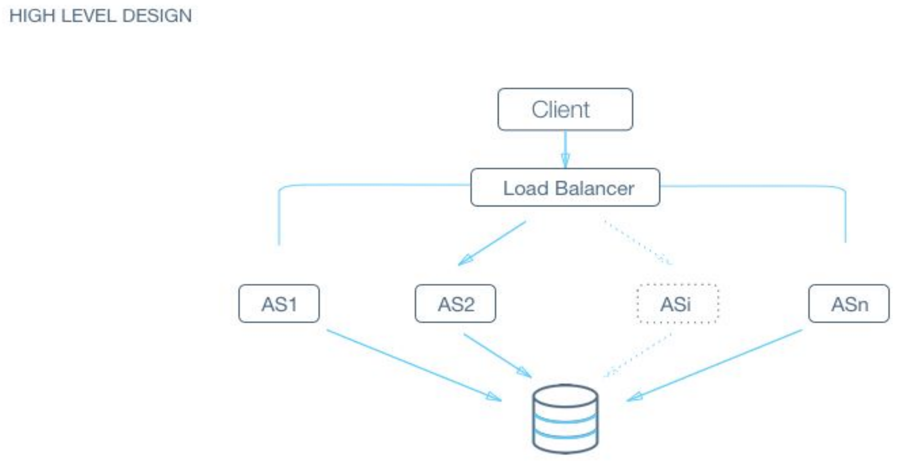
## Implementation :

- **How would a typical read query look like?**
  - Client ( Mobile app / Browser, etc ) which calls RedirectionAPI(short_url)
  - Application server which interprets the API call, extracts the hash from short_url, asks database server for url corresponding to hash and returns the url.
  - Database server which stores the hash => url mapping



## Implementation :

- **How should we compute the hash of a URL?**
  - convert_to_base_62(md5(original_url + salt))[:6](first six characters)
  - Save Result -> original_url
  - Directly encoding URL to base_62 will allow a user to check if a URL has been shortened already or not
  - reverse enginnering can lead the user to the exact hash function used
  - if two users shortens the same URL, it should result in two separate shortened URL
- How do we handle the case where our application server dies?



## Implementation :

- **How does our client know which application servers to talk to. How does it know which application servers have gone down and which ones are still working?**
    - stateless.

- **What all data should we store?**
    - Hash => URL mapping.
    - Billions of small sized(1kb) object.
    - We would also need to store data for analytics, for example, how many times was the url opened in the last hour?

- **Should we choose RDBMS or NOSQL?**
    - no joins required so no sql
    - if data stored will be small so use sql.
    - Assumptions :
    - # of writes per month: 100 Million (Refer to estimations section)
    - Avg size of URL : 500 bytes
    - Provisioning for : 5 years
    - Space required : 500 bytes * 100M * 12 * 5 = 3TB (can fit on a single machine)
    - Given read latency is critical for us, we can’t store the data on a single machine
    - **so use no sql database**.

## Implementation :

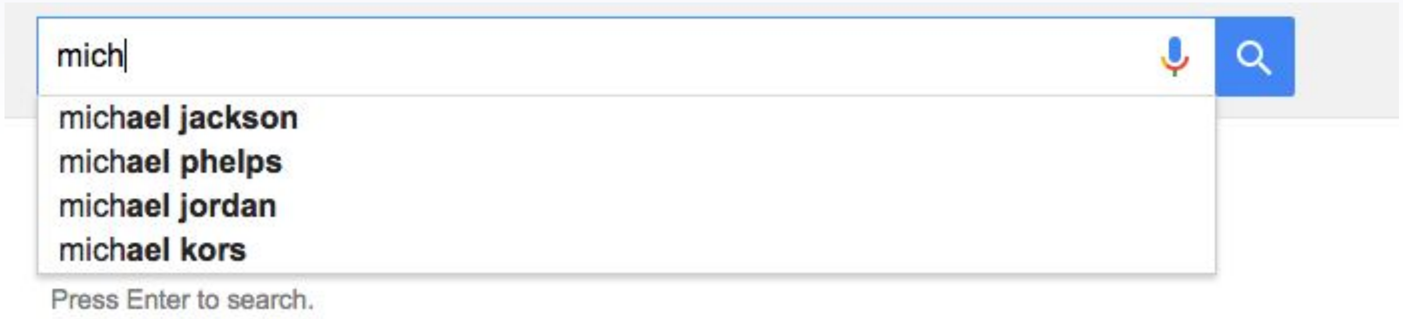
- **What would the database schema look like?**
    - We want to store the mapping from hash -> URL which is ideal for a key value store

- **How would we do sharding?**
    - as mentioned in the sharding system.

- **How would we handle a DB machine going down?**
    - our sharding system survives the down time of a machine.

- **How can we optimize read queries?**
    - using caching.

## Design Search Type ahead



mich

michael jackson
michael phelps
michael jordan
michael kors

Press Enter to search.

## Features:

- **How many typeahead suggestions are to be provided?**
  - 5
- **Do we need to account for spelling mistakes ?**
  - Should typing mik give michael as a suggestion because michael is really popular as a query?
- **What is the criteria for choosing the 5 suggestions ?**
  - most popular 5 queries that have the same prefix.

- **Does the system need to be realtime ( For example, recent popular events like “Germany wins the FIFA worldcup” starts showing up in results within minutes ).**
  - yes sure.
- **Do we need to support personalization with the suggestions? ( My interests / queries affect the search suggestions shown to me).**
  - no for now

## Estimation :

- How many search queries are done per day?
    - around 2-4 Billion queries per day.
- How many queries per day should the system handle?
    - Total Number of queries : 4 Billion
    - Average length of query : 5 words = 25 letters ( Since avg length of english word is 5 letters ).
    - Assuming, every single keystroke results in a typeahead query, we are looking at an upper bound of 4 x 25 = 100 Billion queries per day.

- How much data would we need to store?
    - approximately 50TB.

## Design goals :

- Latency : Are requests with high latency and a failing request, equally bad?
- Consistency : is eventual consistency allowed or it’s sensitive to high consistency.
- Availability : should the system be 100% available?

- Is Latency a very important metric for us?
    - sure
- How important is Consistency for us?
    - not important
- How important is Availability for us?
    - yes sure

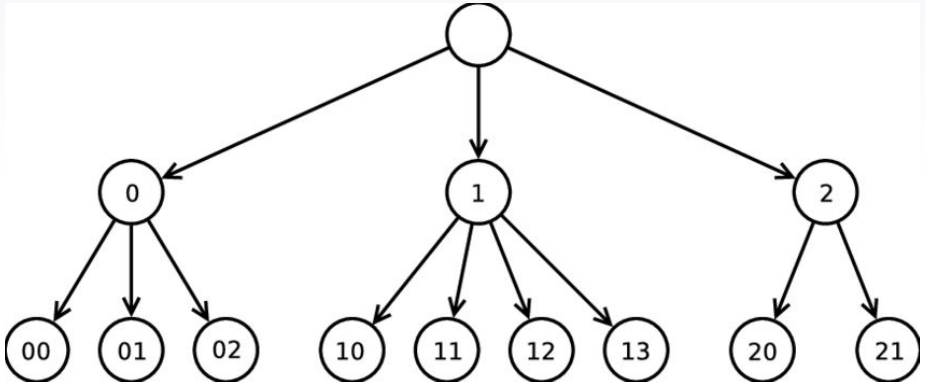
## Implementation :

- **Application layer:**
- How do we handle the case where our application server dies?
    - replication
- How does our client know which application servers to talk to. How does it know which application servers have gone down and which ones are still working?
    - stateless
- **Database layer:**
- How would a read query on the trie work?

## Implementation :

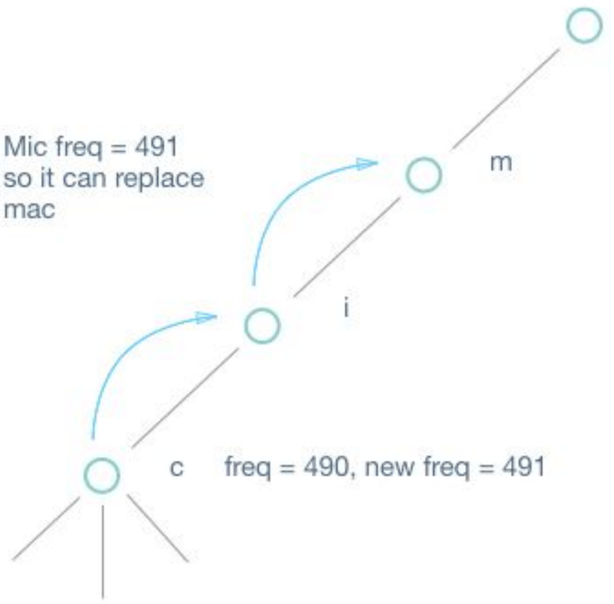
- What would the API look like for the client?
  - Read: List(string) getTopSuggestions(string currentQuery)
  - Write: void updateSuggestions(string searchTerm)

- What is a good data structure to store my search queries so that I can quickly retrieve the top 5 most popular queries?
  - a prefix tree



## Implementation :

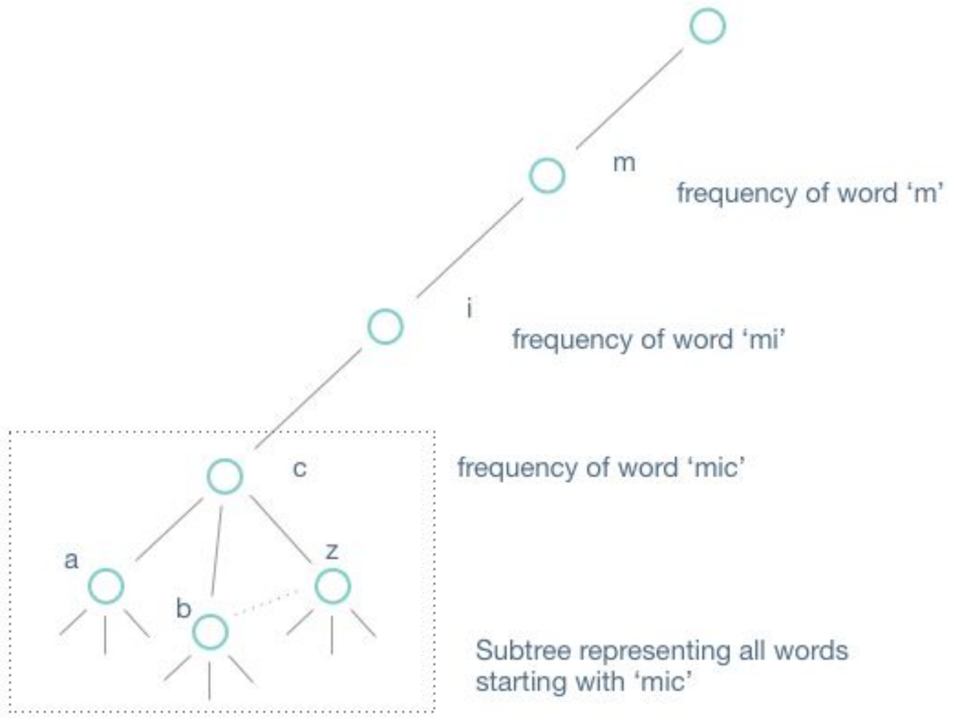
ADD ‘Mic’



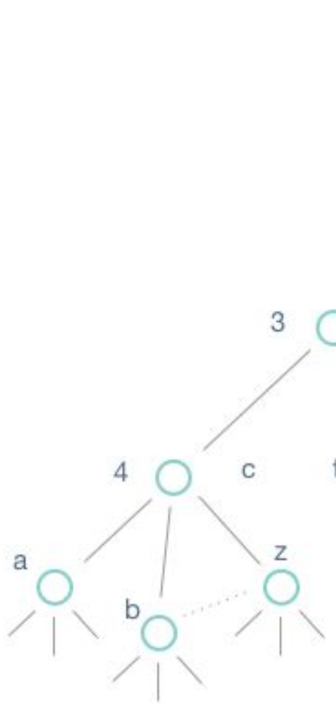
| Michael | Mac |
|---|---|
| 500 | 490 |

( Top 5 phase with their current frequency )

## Implementation :



## Implementation :



top 5 phrase with prefix as 'm'

top 5 phrase with prefix as 'mi'

top 5 phrase with prefix as 'mic'

Query for 'mic' just return data stored on node 4
What about updates though?

## Implementation :

- How would a typical write work in this trie?
- Can frequent writes affect read efficiency?
- What optimizations can we do to improve read efficiency regarding to frequent updates?(using sampling)
- Offline update?
- What if I use a separate trie for updates and copy it over to the active one periodically?
- Would all data fit on a single machine?
- Would we only shard on the first level?
- What is the downside of assigning one branch to a different shard?
- Alright, how do we shard the data then?
- How would we handle a DB machine going down?

references :

- https://www.interviewbit.com/courses/system-design/
- https://www.educative.io/courses/grokking-the-system-design-interview