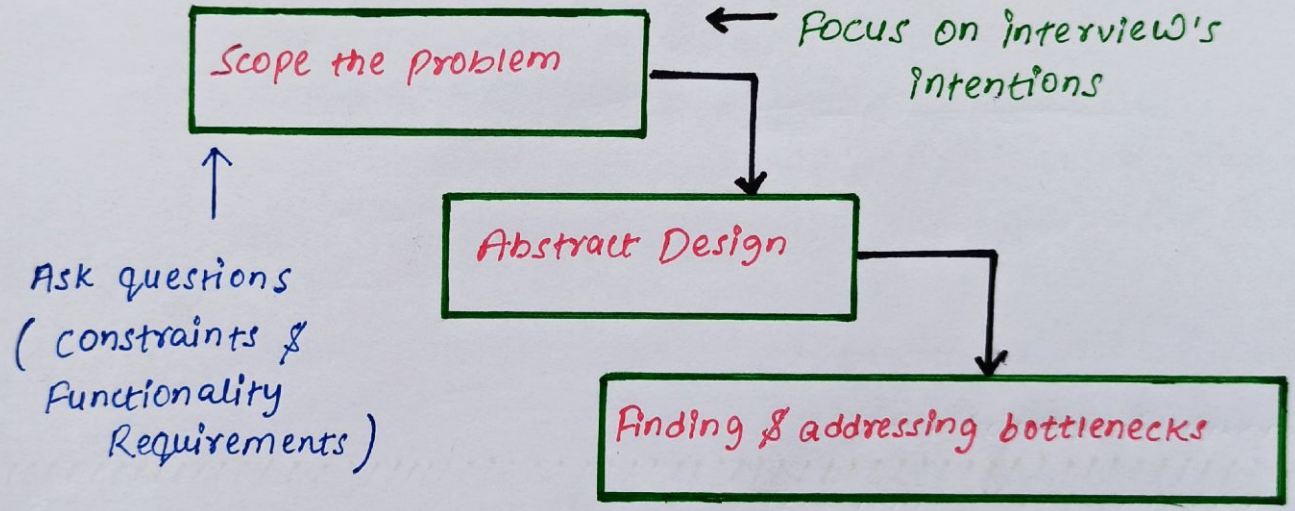
SYSTEM DESIGN NOTES

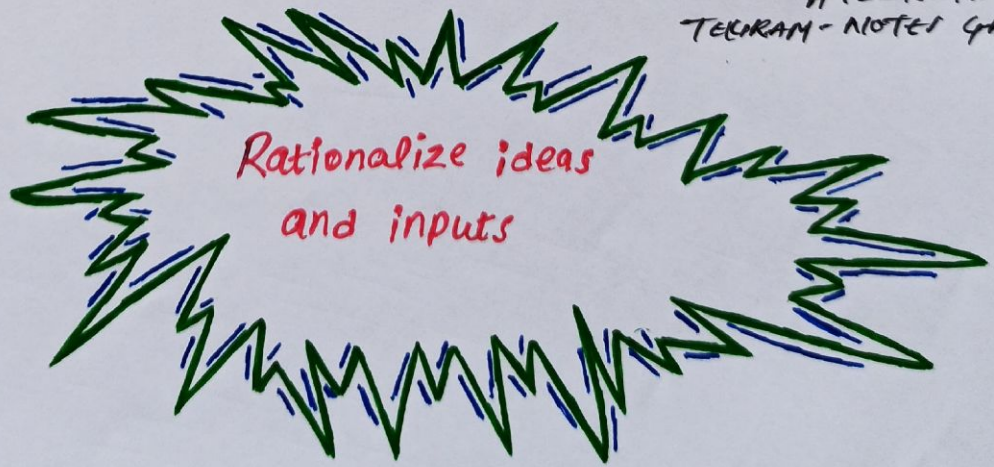
System Design Basics

1). Try to break the problem into simpler modules
(Top down approach).

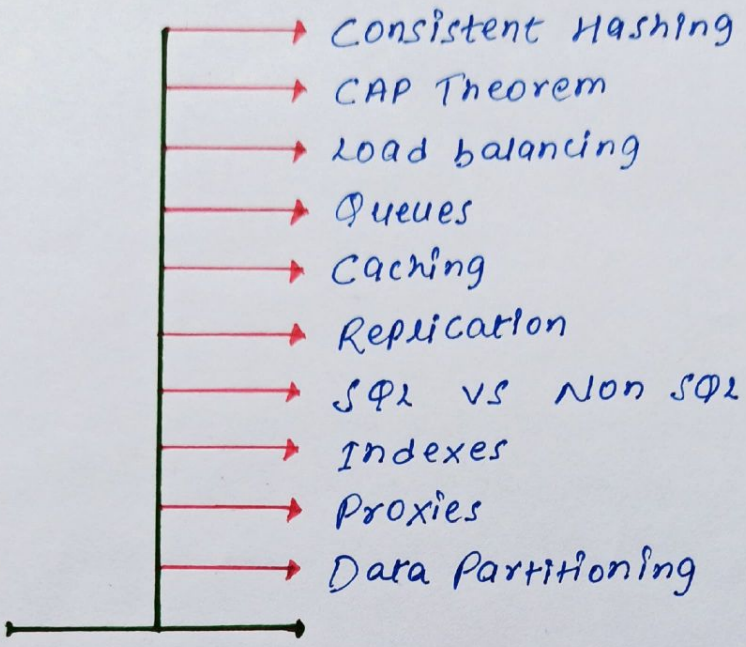
2). Talk about the trade-offs
(No solution is perfect)
Calculate the impact on system based on all the constraints and the end test cases.



ATUL KUMAR (LINKEDIN)
TELEGRAM - NOTES GALLERY.



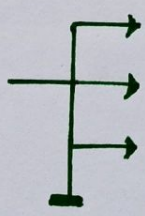
1). Architectural pieces / resources available.
2). How these resources work together.
3). Utilization & TradeOffs



Load Balancing

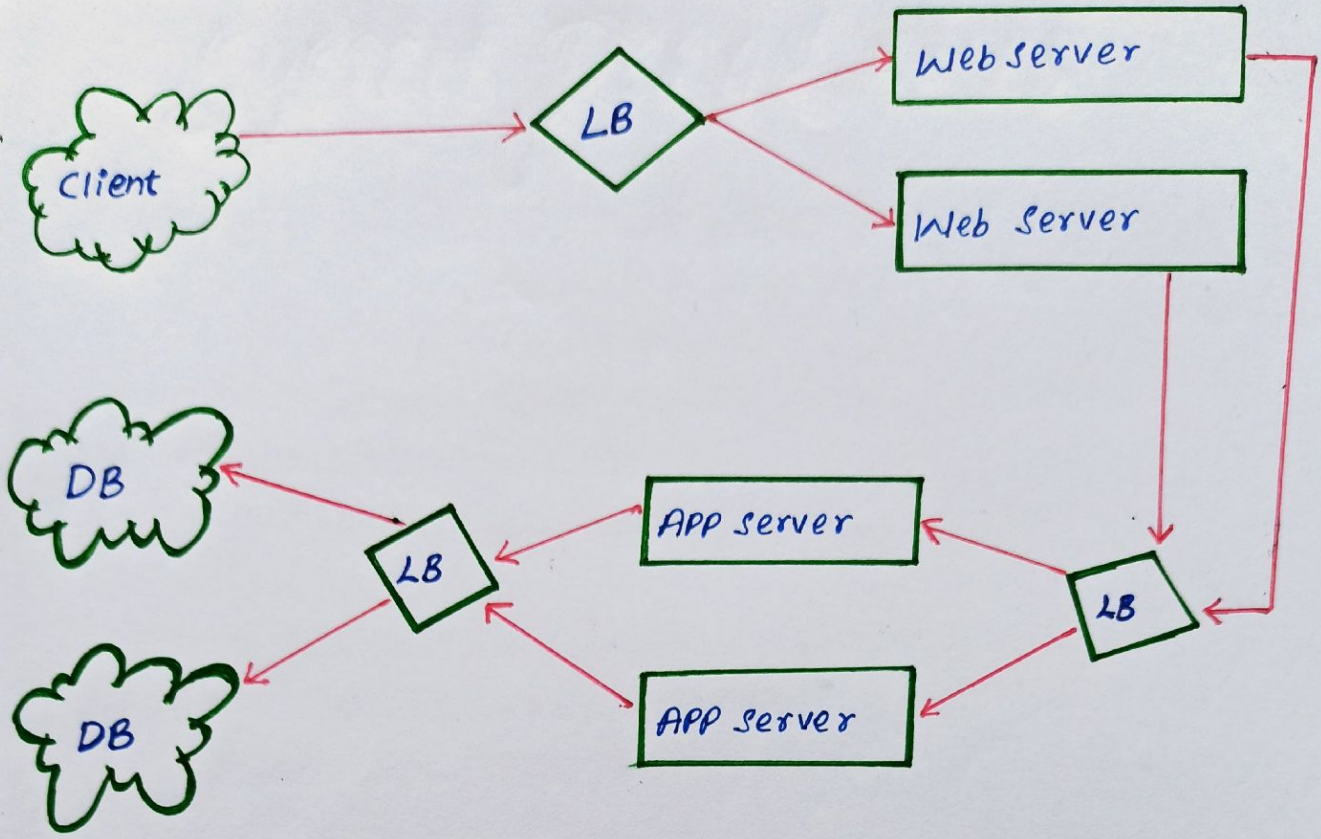
(Distributed System)

Types of distribution



To utilize full scalability & redundancy, add 3 LB

1). User <-- LB1 --> Web Server
2). Web Server <-- LB2 --> App server / Cache server (Internal platform).
3). Interal Platform <-- LB3 --> DB.



Smart Clients

Takes a pool of services hosts & balances load.

→ detects hosts that ave not responsive

→ recovered hosts

→ addition of new hosts

Load balancing functionality to DB (cache, services)

☆ Attractive solution for developers.
(small scale systems)

As system grows → LBs (standalone servers)

Hardware Load Balancers :

Expensive but High performance

eg. Citrix Netscaler

Not trival to configure.

Large companies tend to avoid this config. or use it as 1st point of contact to their system to serve user request $.

Intra network uses smart clints / hybrid solution → (Next Page) for load balancing traffic.

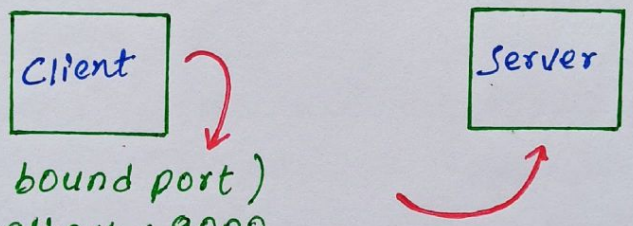
Software Load Balancers

No pain of creation of smart client.
No cost of purchasing dedicated hardware.
→ hybrid approach

HA Proxy ⇒ OSS Load Balancers

⇓

1). Running on client machine.



(locally bound port)
eg. localhost : 9000

↑ Managed by HA Proxy
(with efficient management of requests on the port)

2). Running on Intermediate Server:
Proxies running between different server side components

HA Proxy
- → manage health checks
- → removal & addition of machines
- → balances request a/c pools.

WORLD OF DATABASE

SQL VS. NOSQL

Relational Database

1). Structured
2). Predefined schema
3). Data in rows & columns

Row ⇒ One Entity Info
Column ⇒ separate data Points

- MySQL
- Oracle
- MS SQL server
- SQLite
- Postgres
- MariaDB

Non - relational Database

1). Unstructured.
2). distributed.
3). dynamic schema.

- Key-value stores
- Document DB
- Wide-Column DB
- Graph DB

ATUL KUMAR (LINKEDIN),
TELEGRAM - NOTES GALLERY.

NOSQL

Key-value stores

Data = array of key-value pair
Key = attribute
Value ← linked to

Redis
Voldemart
Dynamo

Document DB

Data = documents ⇓ grouped into collections

Each doc can be different

CouchDB
MongoDB

Wide Column DB

Instead of tables, columns families → container for rows

No need of knowing all column upfront.
Each row ⇒ diff. no. of columns.

Analysis of large datasets.

Cassandra
HBase

Graph DB

Data whose relations are best represented in graphs.

⇒ Nodes (Entities)
⇒ Properties (information of entities)
⇒ Lines (connection betn entities)

Neo 4J
Infinite graph

HIGH LEVEL DIFF. B/W SQL & NOSQL

Property	SQL	NO SQL
Storage	Tables (ROW → Entity, column → Data point) eg. student (Branch, Id, Name)	Diff. data storage models. (key value, document, graph, column)
Schema	Fixed Schema (column must be decided & chosen before data entry) Can be altered ⇒ modify whole database (needs to go offline)	Dynamic Schemas. Column addition on the fly. Not mandatory for each rows to contain data.
Querying	SQL	UnQL (Unstructured query language) Queries focused on collection of documents. Diff. DB ⇒ diff. UnQL.
Scalability	Vertically scalable (+ horse-power of h/w) Expensive Possible to scale across multiple servers, ⇒ challenging & time-consuming.	Horizontally scalable. Easy addition of servers. Hosted on cloud or cheap commodity h/w → cost effective.
Reliability or Acid Compliancy	ACID compliant ⇒ Data reliability ⇒ Guarantee of transactions ⇒ Still a better bet.	Sacrifice ACID compliance for scalability & performance. (Acid - Atomicity, consistency. Isolation, Durability.

Reason to use SQL DB

1). You need to ensure ACID Compliance:

ACID compliance
⇒ Reduces anomalies
⇒ Protects integrity of the database.

For many E-commerce & financial appn
→ ACID compliant DB
is the First Choice.

2). Your data is structured & Unchanging.
If your business is not experiencing rapid growth or sudden changes
→ No requirements of more servers.
→ data is consistent.
then there's no reason to use system design to support variety of data & high traffic.

Reasons to use NoSQL DB.

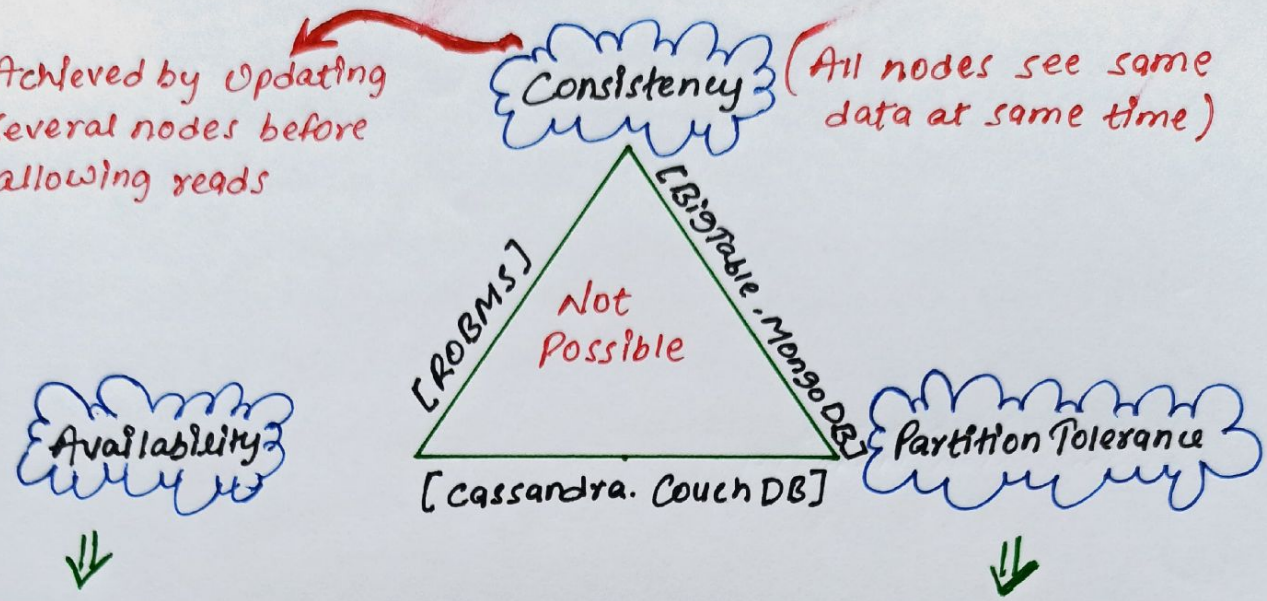
When all other components of system are fast
→ querying & searching for data ⇒ bottleneck.
NOSQL prevent data from being bottleneck.
Big data ⇒ large success for NOSQL.

1). To store large volumes of data (little/no structure)
No limit on type of data.
Document DB ⇒ stores all data in one place
(No need of type of data)

2). Using cloud & storage to the fullest.
Excellent cost saving solution. (Easy spread of data across multiple servers to scale up)
OR commodity h/w on site (affordable, smaller)
⇒ No headache of additional s/w.
& NOSQL DB like Cassandra ⇒ designed to scale across multiple data centers out of box.

3). Useful for rapid / agile development.
If you're making quick iterations on schema
⇒ SQL will slow you down.

CAP Theorem



Every request gets response (success/failure)
Achieved by replicating data accross different servers.

System continues to work despite message loss/partial failure.
(can sustain any amount of network failure without resulting in failure of entire network).

Data is sufficiently replicated across combination of nodes / network to keep the system up.

> It is impossible for a distributed system to simultaneously provide more than two of three of the above guarantees.

We cannot built a datastore which is:

1). Continually available
2). Sequentially Consistent
3). partition failure tolerant

Because,

To be consistent ⇒ all nodes see the same set of updates in same order

But if network suffers partition,
update in one partition might not make it to other partitions.

↳ Client reads data from out-of-date partition
After having read from up-to-date partition.

Solution: stop serving request from out-of-date partition.

↓

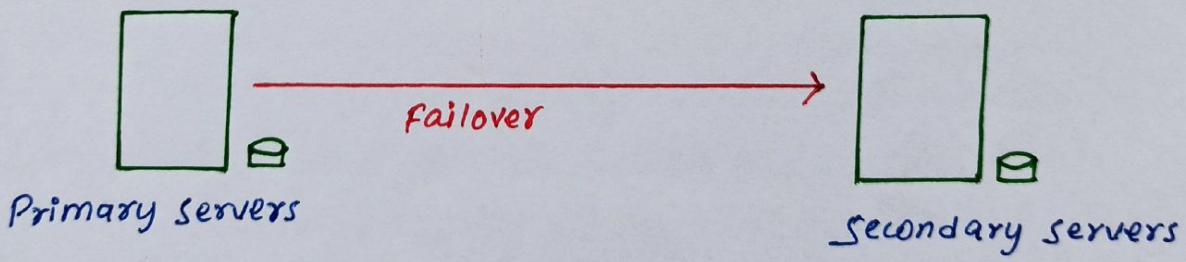
Service is no longer 100% available.

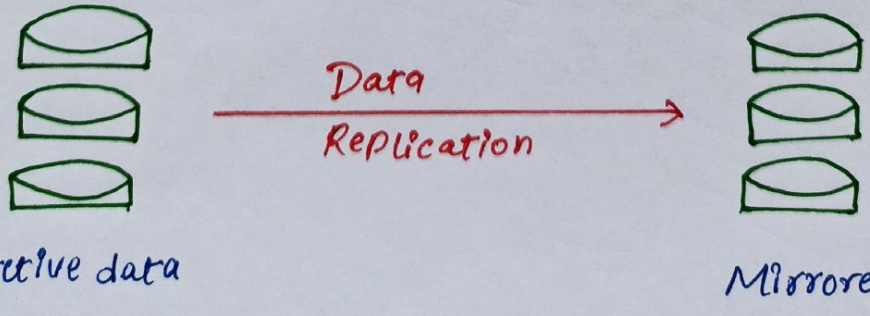
Redundancy & Replication

⇒ Duplication of critical data & services
↳ increasing reliability of system.

for critical services & data ⇒ ensure that multiple copies/versions are running simultaneously on different servers / databases.

⇒ Secure against single node failures.
⇒ Provides backups if needed in crisis





Service Redundancy: Shared-nothing architecture.

Every node ⇒ independent. ∴ No central service managing state.

Helps in scalability ← New servers addition without special conditions. ← more resilient to failures

No single point of failure

Caching

Load balancing ⇒ Scales horizontally

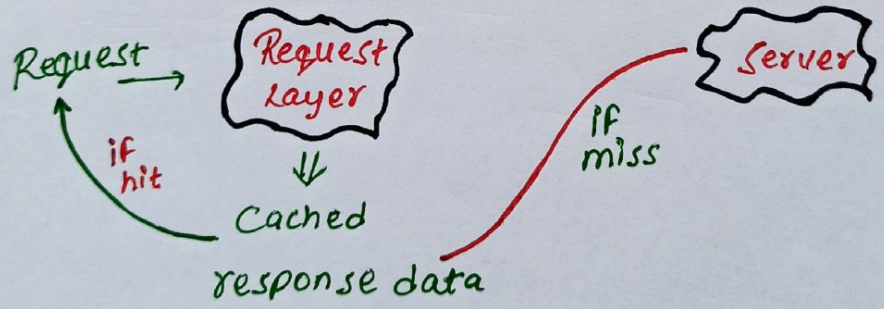
Caching = Locality of reference principle

↳ used in almost every layer of computing.

1). Application server cache:

Placing a cache directly on a request layer node.

↳ Local storage of response.



Cache on one Request layer node can be located

- Memory (very fast)
- Node's local disk (faster than going to network storage)

Bottleneck: If LB distributes request randomly

↳ Same request ⇒ different nodes

↓

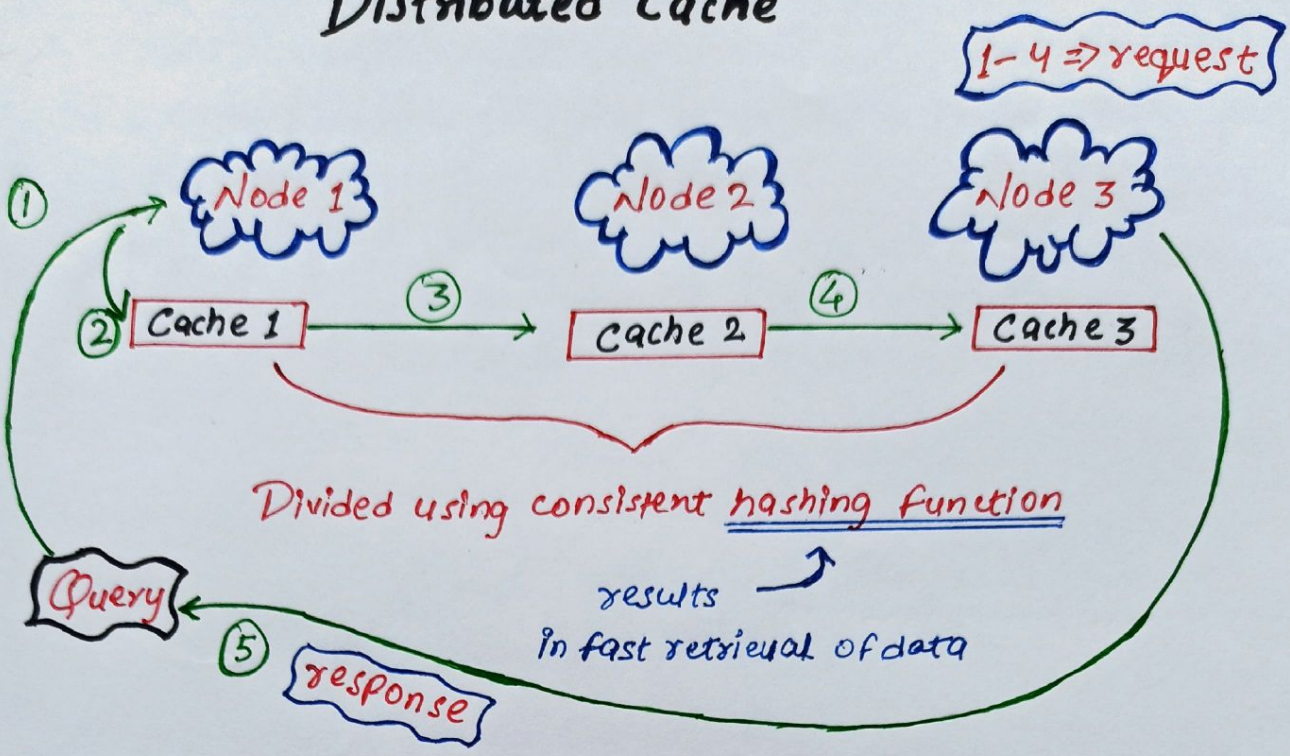
more cache miss

Can be overcomed by

1). Global caches

2). Distributed caches

Distributed Cache



Easy to increase cache space by adding more nodes

Disadvantage : Resolving a missing node.

storing multiple copies of ⟵ can be handled by data on different nodes

⟹ Were making it more complicated.

Even if node disappears ⇒ request can pull data from Origin.

Global Cache

Single cache space for all the nodes.

↳ Adding a cache server/ file store (faster than Original store)

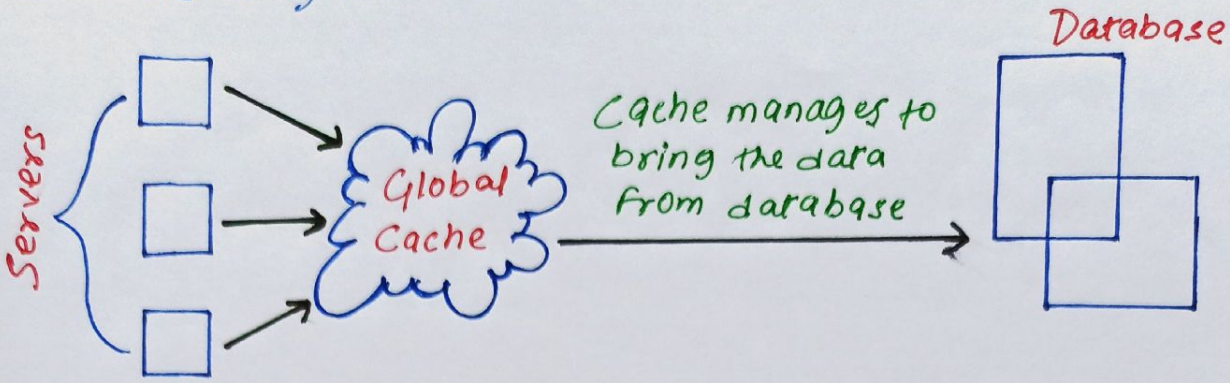
Difficult to manage if no. of clients/request increases.

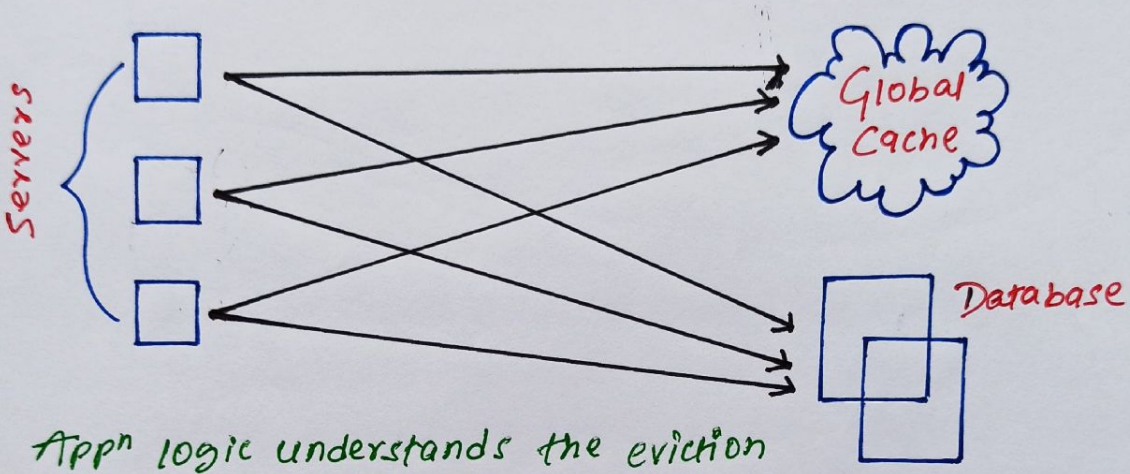
Effective if

1). Fixed dataset that needs to be cached.

2). Special H/W ⇒ fast I/O

forms of global cache:

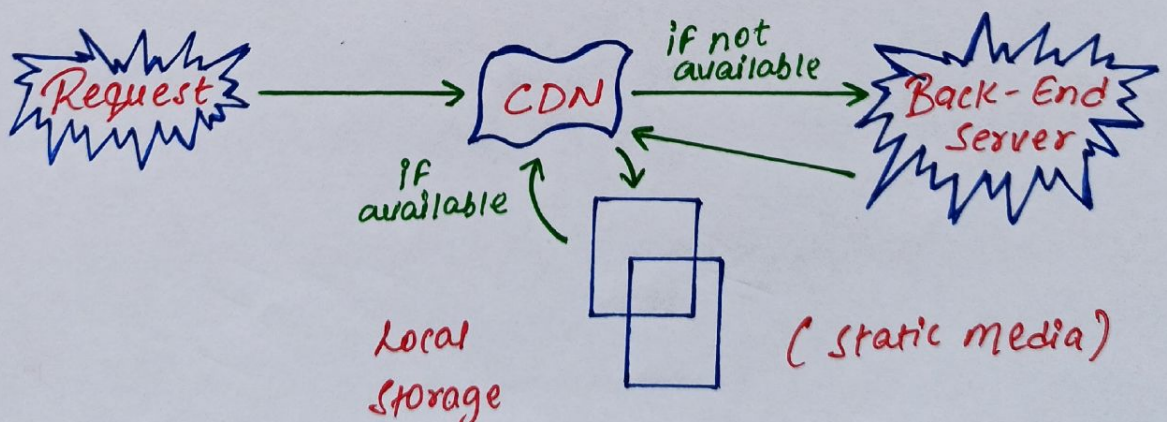




Appⁿ logic understands the eviction strategy / hotspots better then cache.

CDN: Content Distribution network

↳ Cache store for sites that serves large amount of static media.



If the site isn't large enough to have its own CDN
↳ for letter & easy future transition.

Serve static media using separate subdomain.
(static.yourservice.com)

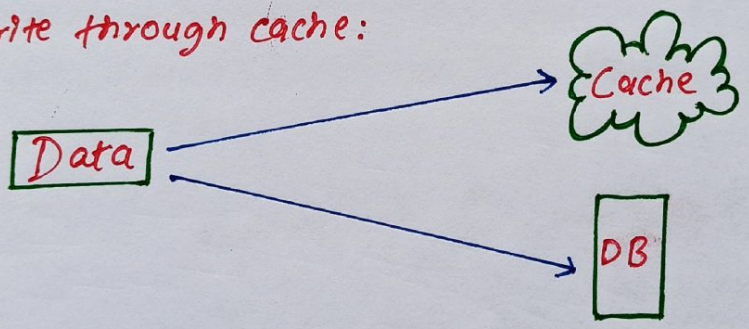
Using lightweight nginx server
↳ cutover DNS from your server to a CDN later.

Cache Invalidation

Cached data ⇒ need to be coherent with the database
If data in DB modified ⇒ invalidate the cached data.

3 schemes:

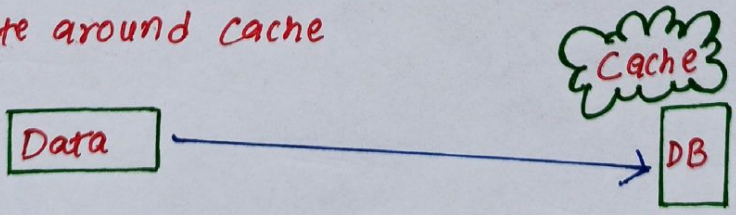
1). Write through cache:



Data is written same time in both cache & DB.

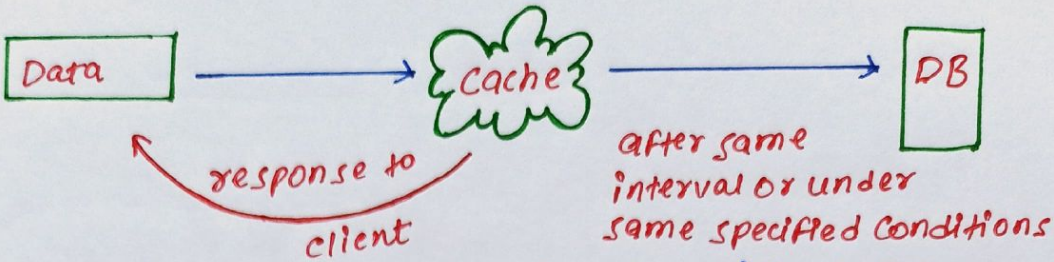
+ Complete data consistency (cache ≡ DB)
+ fault tolerance in case of failure (↓↓ data loss)
− High latency in writes ⇒ 2 write operations

2). Write around cache



+ no cache flooding for writes
− read request for newly written data ⇒ Miss
Higher latency

3). Write back cache:



data is written to DB from cache.

+ low latency & high throughput for write-intensive appn
- Data loss ↑↑ (only one copy in cache)

Cache Eviction Policies

1). FIFO
2). LIFO or FILO
3). LRU
4). MRU
5). LFU
6). Random Replacement

Sharding || Data Partitioning

Data Partitioning : Splitting up DB/table across multiple machines ⇒ Manageability, Performance, availability & LB

** After a certain scale point, it is cheaper and more feasible to scale horizontally by adding more machines instead vertical scaling by adding.

Methods of Partitioning:

1). **Horizontal partitioning**: Different rows into diff. tables.

Range based Sharding

e.g. Storing location by zip
Table 1: Zips with < 100000
Table 2: Zips with > 100000
and so on. } ⇒ different ranges in different tables.

★★ cons: if the value of the range not chosen carefully
⇒ leads to unbalanced servers
e.g. Table 1 can have more data than table 2.

Vertical Partitioning:

feature wise distribution of data
↳ in different servers
eg. Instagram → DB server 1: user info
DB server 2: followers
DB server 3: Photos

★★ Straightforward to implement
★★ low impact on app.

⊖⊖ if app → additional growth
need to partition feature specific DB across various servers
(e.g. it would not be possible for a single server to handle all metadata queries for 10 million photos by 140 million users.

Directory based Partitioning

⇒ A loosely coupled approach to work around issues mentioned in above two partitioning.
★★ Create lookup service ⇒ current partitioning scheme & abstracts it away from the DB access code.

Mapping (tuple key ⟶ DB servers)

Easy to add DB servers or change partitioning scheme.

Partitioning Criteria

1). Key or Hash based partitioning :

Key attr. of the data → Hash function → Partition number

Effectively fixes the total number of servers/Partitions so if we add new server/partition → Change in hash function → downtime because of redistribution → Solution: Consistent Hashing

2). List Partitioning : Each partition is assigned a list of values.

new record → Lookup for keys → store the record (Partition based on the key)

3). Round Robin Partitioning :

Uniform data distribution

With 'n' Partitions

⇒ the 'i' tuple is assigned to partition (i mod n)

4). Composite Partitioning :

Combination of above partitioning schemes

Hashing + List ⇒ Consistent Hashing

⇓

Hash reduce the key space to a size that can be listed.

Common Problems of sharding:

Sharded DB : Extra constraints on the diff. Operations.

⇓

Operations across multiple tables or multiple rows in the same table ↘

no longer running in single server.

1). Joins & Denormalization:

Joins on tables on single server ⇒ Straightforward

★ not feasible to perform joins on sharded tables.

↳ Less efficient (data need to be compiled from multiple servers)

Workaround ⇒ Denormalize the DB.

so that the queries that previously reqd. joins can be performed from a single table.

Cons: Perils of denormalization

↳ data inconsistency

2). Refrential integrity : foreign keys on sharded DB

↳ difficult

★ Most of the RDBMS does not support foreign keys on sharded DB.

If appn demands referential integrity on sharded DB

↳ enforce it in appn code (SQL joins to clean up dangling references)

3). Rebalancing:

Reasons to change sharding scheme :

a). non-uniform distribution (data wise)

b). non-uniform load balancing (request wise)

Workaround : 1). add new DB

2). rebalance

↳ Change in partitioning scheme

↳ data movement

↳ down time.

We can use directory-based partitioning

↳ highly complex

↳ Single point of failure.

(lookup service / table)

ATUL KUMAR (LINKEDIN).
TELEGRAM - NOTES GALLERY.

INDEXES

⇒ Well known because of databases.

⇒ Improves speed of retrieval.

– Increased storage overhead.

– Slower writes

↳ Write the data

↳ Update the index

⇒ can be created using one or more columns.

★ Rapid random lookups

& efficient access of ordered records.

Data Structure

Column → Pointer to whole row.

→ Create different views of the same data.

↳ very good for filtering / sorting of large data sets.

↳ no need to create additional copies.

Used for datasets (TB in size) & small payload (KB)

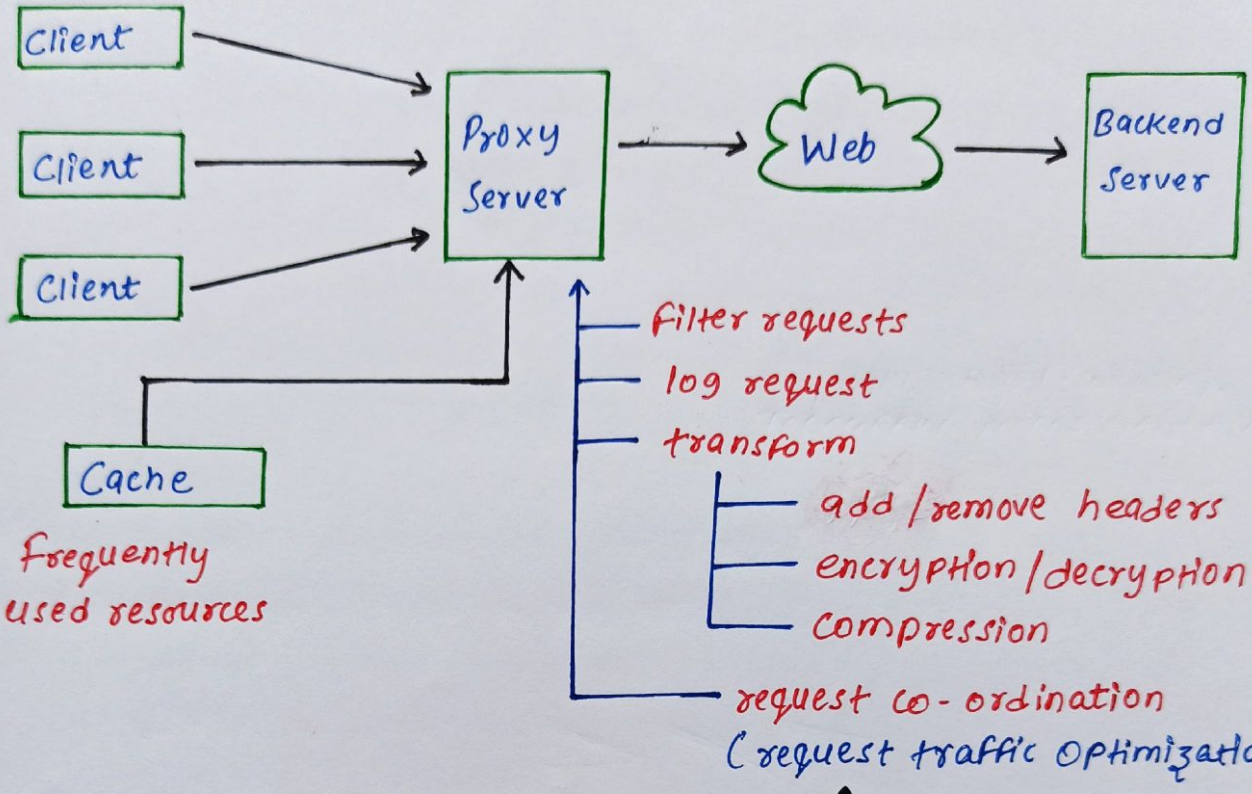
Spred over several → Physical devices → We need some way to find the correct physical location

i.e. **Indexes**

Proxies

Useful under high load situations if we have limited caching.

↳ batches several requests into one.



We can also use spatial locality

↳ collapsing request for data that is spatially close

collapse same data access request into one.

⇒ Collapsed forwarding

↳ minimize reads from Origin.

Queues

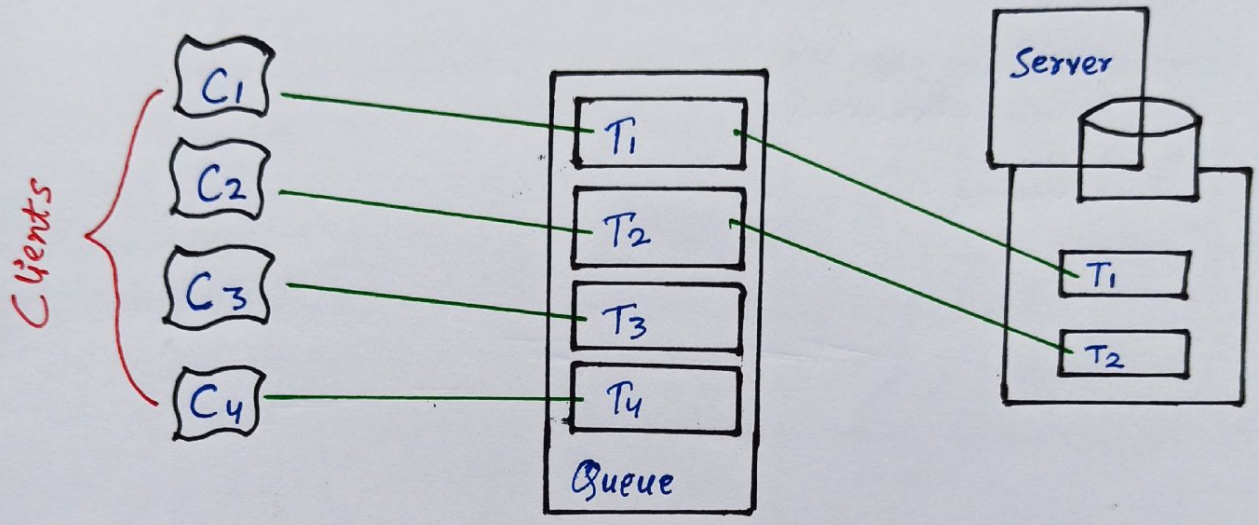
⇒ Effectively manages requests in large-scale distributed system.

→ In small system → Writes are fast.

→ In complex system → High incoming load & individual writes take more time.

* To achieve high performance & availability
  ↳ System needs to be asynchronous
    ↳ Queues

Synchronous behaviours → degrades performance
↓
Can use load balancing
↑
difficult for fair & balanced distribution



Queues : asynchronous Communication protocol
  ↳ Client sends task
    ↳ gets ACK from queue (receipt)
      ↑ Serves as reference for the results in future
    ↳ Client continues its work.

Limit on the size of request & number of requests in queue.

Queue : Provides Fault - tolerance
  ↑ Highly Robust
  ↳ Protection from service outage/failure
    ↳ retry failed service request
    ↑ Enforce Quality of Service gurantee
    (Does NOT expose clients to outages)

Queues: distributed communication
↳ Open source implementations
↳ RabbitMQ, ZeroMQ, ActiveMQ, BeanstalkD.

Consistent Hashing

ATUL KUMAR (LINKEDIN).
TELEGRAM - NOTES GALLERY.

Distributed Hash Table

index = hash_function (key)

Suppose were designing distributed caching system with n cache servers
↳ hash_function ⇒ (key % n)

Drawbacks:

1). NOT horizontally scalable
↳ addition of new servers results in
↳ need to change all existing mapping.
(downtime of system)

2). NOT Load balanced
(because of non-uniform distribution of data)
↓
Some caches: hot & saturated
Other caches: idle & empty

How to tackle above problems?
Consistent Hashing

What is Consistent Hashing?

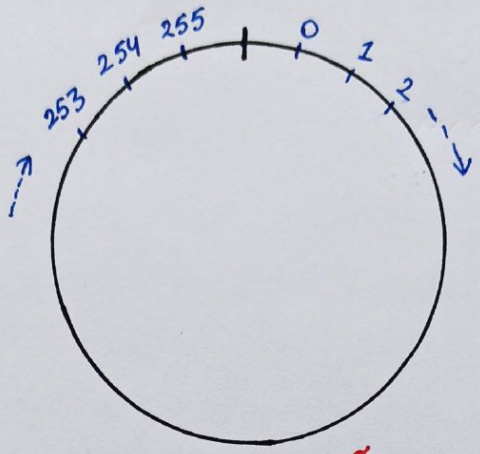
→ Very useful strategy for distributed caching & DHTs.
→ Minimize reorganization in scaling up/down.
→ Only $\boxed{K/n}$ keys needs to be remapped.
K ⇒ total number of keys
n ⇒ numbers of servers

How it works?

Typical hash function suppose outputs in [0.256]

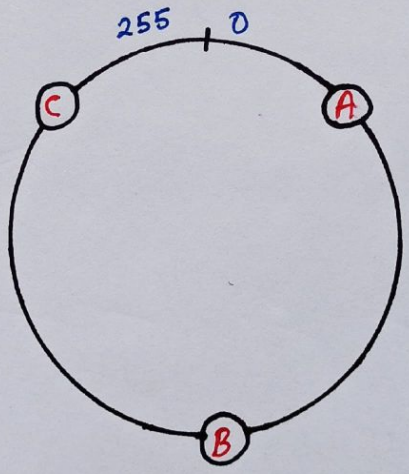
In consistent hashing.

imagine all of these Integers are placed on a ring.



∴ We have 3 servers: A, B & C.

1). Given a list of servers, hash them to integers in the range.

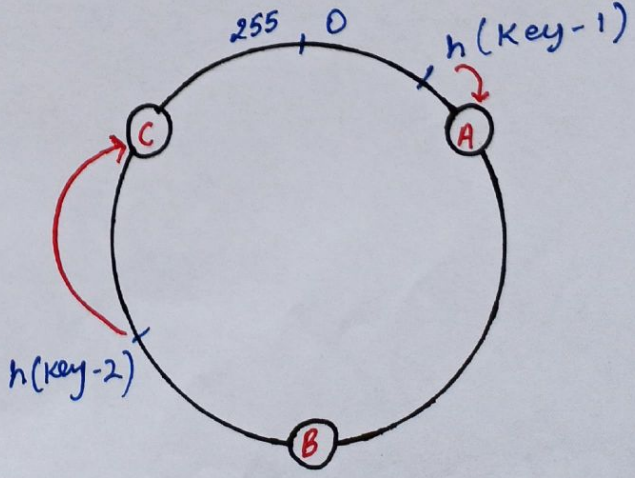


2). Map key to a server:

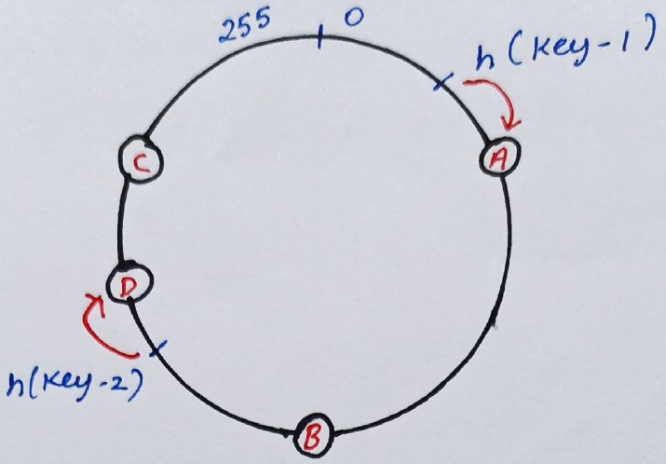
a). Hash it to single Integer

b). Move CLK wise until you find server.

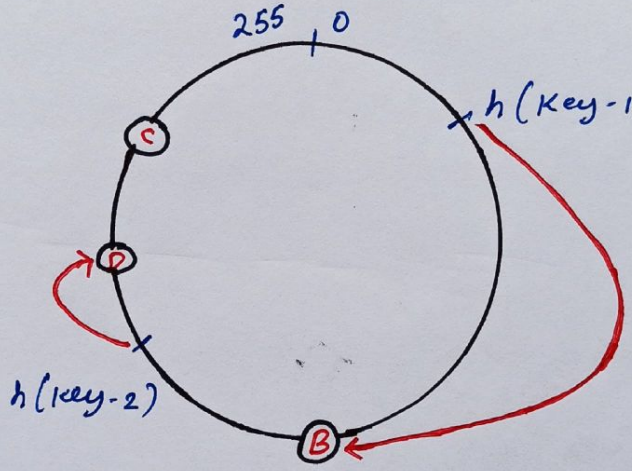
c). Map key to that server.



Adding a new server 'D', will result in moving the 'key-2' to 'D'.



Removing server 'A', will result in moving the 'key-1' to 'B'



Consider real world scenario

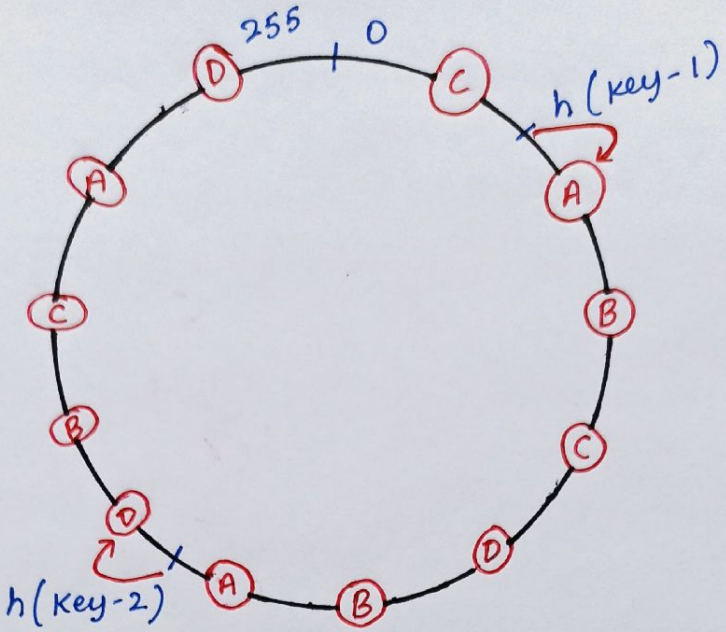
ATUL KUMAR (LINKEDIN).
TELEGRAM - NOTES GALLERY.

data ⟶ randomly distributed
↳ Unbalanced caches.

How to handle this issue?

Virtual Replicas

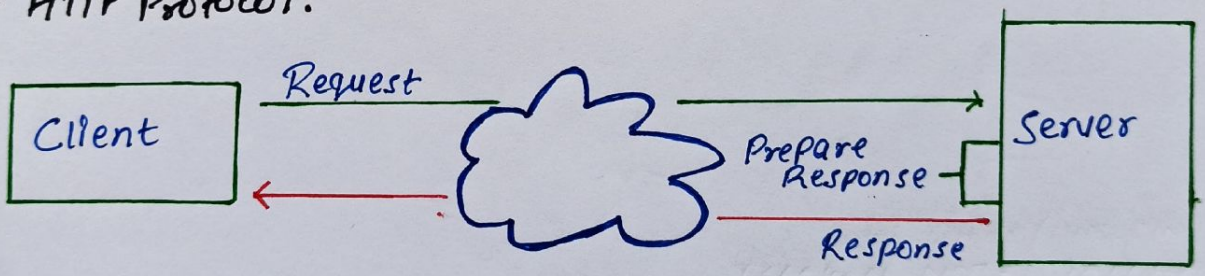
⇒ Instead of mapping each node to a single point we map it to multiple points
↳ (more number of rep
↳ more equal distribution
↳ good load balancing)



Long-Polling vs WebSockets vs Server-Sent Events

↳ Client-server communication protocols

HTTP Protocol:



AJAX Polling :

Clients repeatedly polls servers for data

Similar to HTTP Protocol

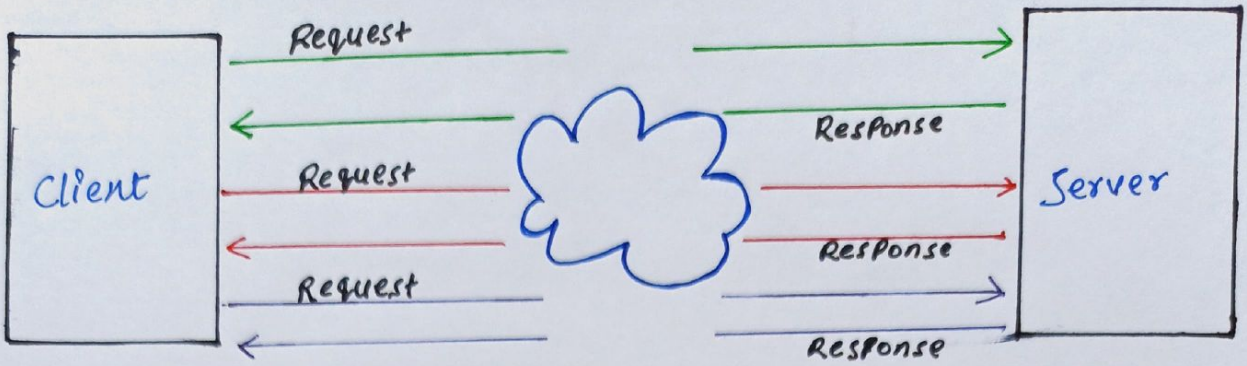
↳ request sent to server at regular intervals (0.5 sec).

Drawbacks:

Client keeps asking the servers new data

↳ Lot of responses are 'empty'

↳ HTTP Overhead.



HTTP Long Polling: 'Hanging GET'

Server does NOT send empty response.

Pushes response to clients only when new data is available.

1). Client makes HTTP request & wait for the response.

2). Server delays response until update is available or until timeout occurs.

3). When update → Server send full response.

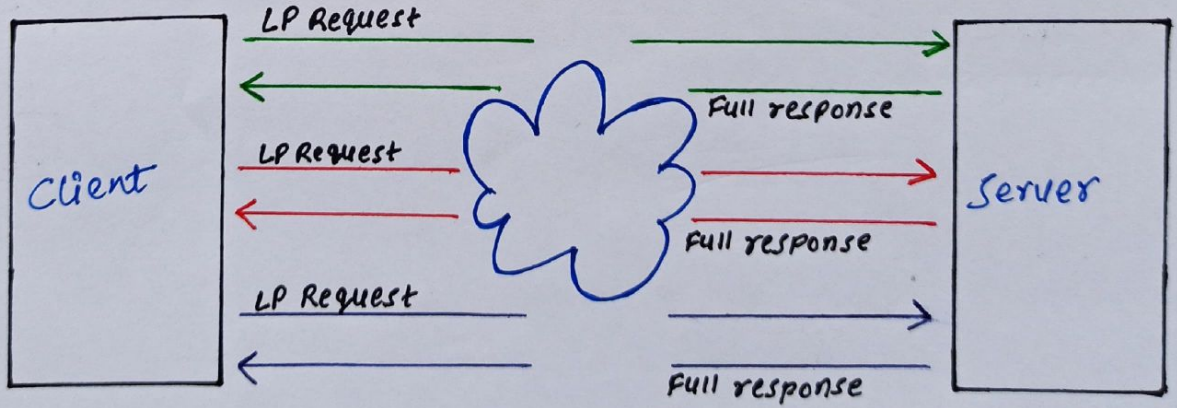
4). Client sends new long-poll request

   a). immediately after receiving response

   b). after a pause to allow acceptable latency period

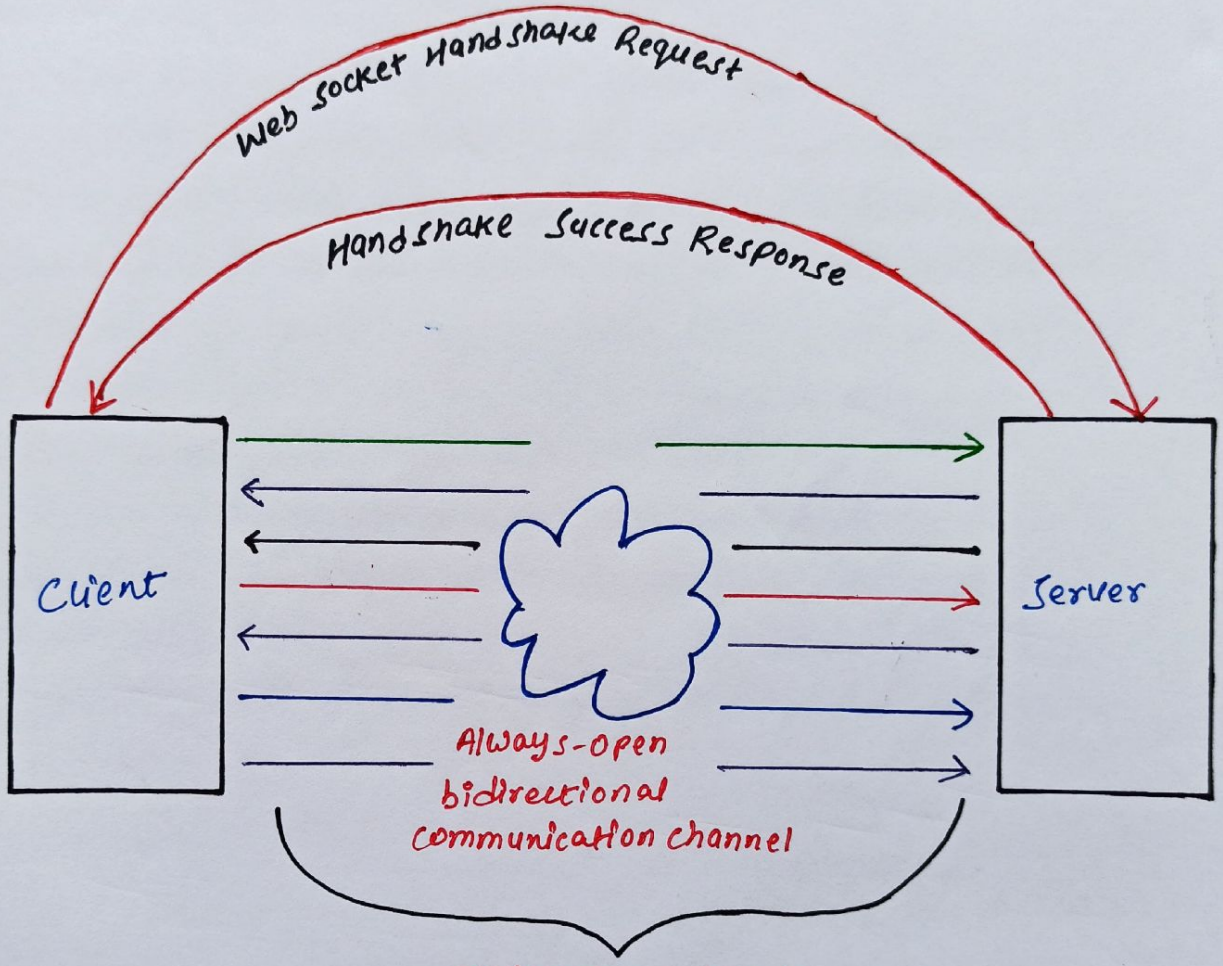
5). Each request has timeout.

   Client needs to reconnect periodically due to timeouts



Web Sockets

→ Full duplex communication channel over single TCP connection.
→ Provides 'Persistent communication'
(client & server can send data at anytime)
→ bidirectional communication in always open channel.



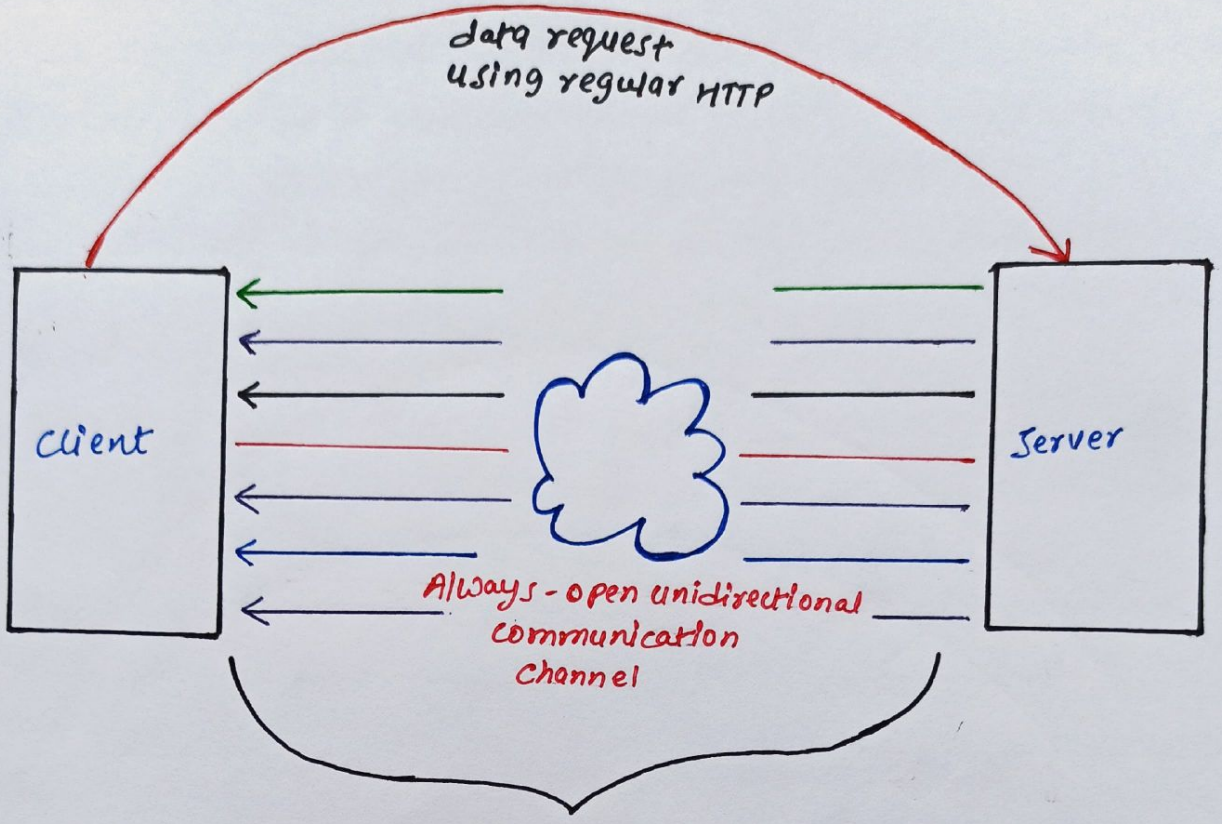
→ Lower overheads
→ Real time data transfer.

ATUL KUMAR (LINKEDIN).

Server-Sent Event (SSE)

Client establishes Persistent & long-term connection with server.
Server uses this connection to send data to client.

** If client wants to send data to serve
↳ Requires another technology / Protocol.



→ best when we need real-time data from server to client OR server is generating data in a loop & will be sending multiple event to the client.