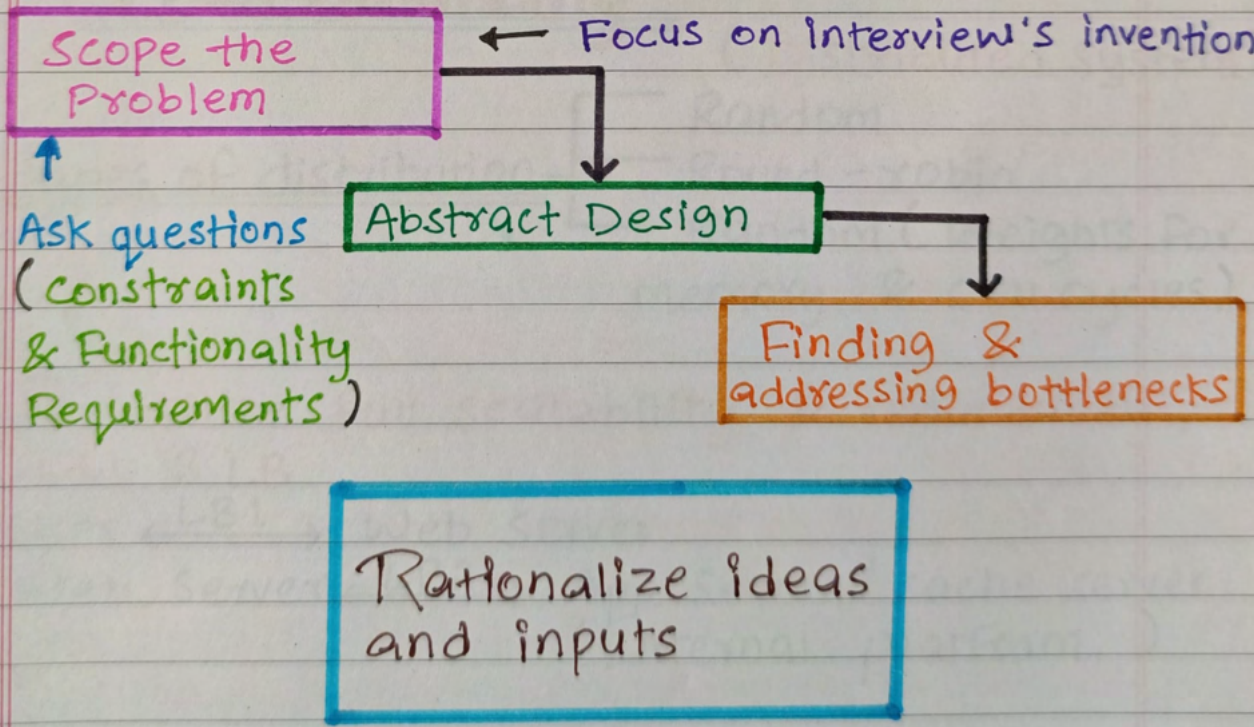
SYSTEM DESIGN

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)
Calculated the impact on system based on all the constraints and the end test cases.



- Architectural Pieces / resources available
- How these resources work together.
- Utilization & Trade offs.

- Consistent Hashing
- CAP THeorem
- Load balancing
- Queues
- Caching
- Replication
- SQL vs NO-SQL
- Indexes
- Proxies
- Data Partitioning.

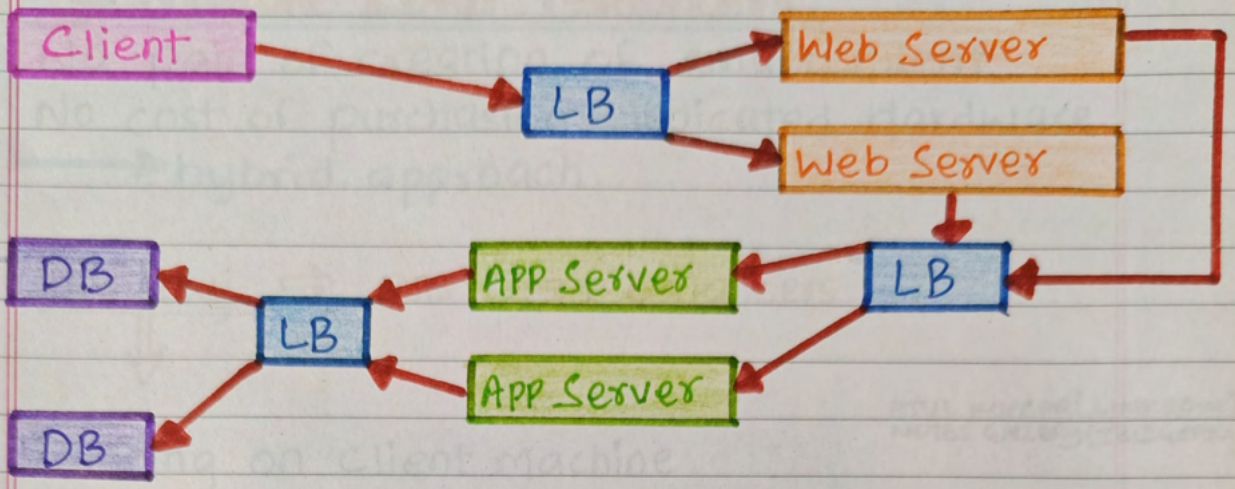
Load Balancing

(Distributed System)

Types of distribution
- Random
- Round-robin
- Random (Weights for memory & CPU cycles)

To utilize full scalability & redundancy, add 3 LB.

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



⇨ Smart Clients

Take a pool of services hosts & balance load.

→ detect hosts that are not responsible.

→ recovered hosts.

→ addition of new hosts.

Load balancing functionality to DB (Cache Services)

☆ Attractive solution for developers. (Small scale system)

As System grows ⟶ LBs (Standalone Servers).

⇨ Hardware load Balancers :

Expensive but high performance.

e.g. Citrix Netscaler

Not trivial to configure

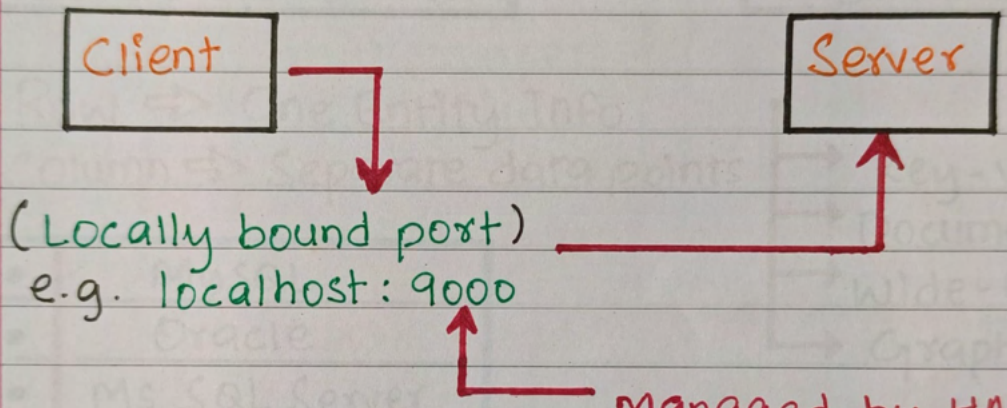
Large companies tend to avoid this Config. or use it as 1st point of contact to their System to serve user request & Intra network uses smart clients / hybrid solution → (Next page) for load balancing traffic.

Software Load Balancers.

- No pain of creation of smart client
- No cost of purchasing dedicated Hardware.
  - hybrid approach.

HAProxy ⇒ OSS Load balancers

1]. Running on client machine



2]. Running on intermediate server:
Proxies running between different server side components

HAProxy
- manages health checks
- removal & addition of machines
- balances requests a/c pools.

World of Database

ATUL KUMAR (LINKEDIN).
NOTES GALLERY (TELEGRAM).

SQL VS NOSQL

Relational Database

- Structured
- Predefined schema
- Data in rows & column

Non-relational Database

- Unstructured
- distributed
- dynamic schema

Row ⇒ One Entity Info
Column ⇒ Separate data points

- MySQL
- Oracle
- Ms SQL Server
- SQLite
- Postgres
- MariaDB

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

NOSQL

Key-Value Store

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

- Redis
- Voldemart
- Dynamo

Document DB

Data ⇒ documents
⇓ group into
Collections
Each doc can be different

- Couch DB
- Mongo DB

Wide-Column DB

Instead of tables, Columns, families.
↓
Container for rows.
No need of knowing all columns upfront.
Each row ⇒ diff. no. of columns
Analysis of large datasets.

- Cassandra
- HBase

Graph DB.

Data whose relati-ons are best repre-sented in Graphs.
⇒ Nodes (Entities)
⇒ Properties (Informa-tion of entities)
⇒ Lines (Connection between entities)

- Neo 4J
- Infinite Graph.

⇒ High Level differences between SQL & NOSQL

PROPERTY	SQL	NOSQL
Storage	Tables (Row → Entity, Column → Data Point) e.g. Student (Branch, Id, Name)	Diff. data storage models. (Key, Value, document, graph, columns)
Schema	Fixed schema (Columns must be decided & chosen before data entry) Can be altered ⇒ Modify whole database (need to go offline)	Dynamic schemas. Columns 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. (+ horsepower of h/w) Expensive. Possible to scale accross 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 scalabi-lity & performance. (Acid - Atomacity, Consistency, Isolation, Durability.

Reason to use SQL DB

1). You need to ensure ACID compliance :
ACID Compliance
⇨ Reduces anomalies
⇨ Protects Intigrity of the database.
For many E-commerce & financial application
⟶ ACID Complaint 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.

Reason to use NOSQL DB

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

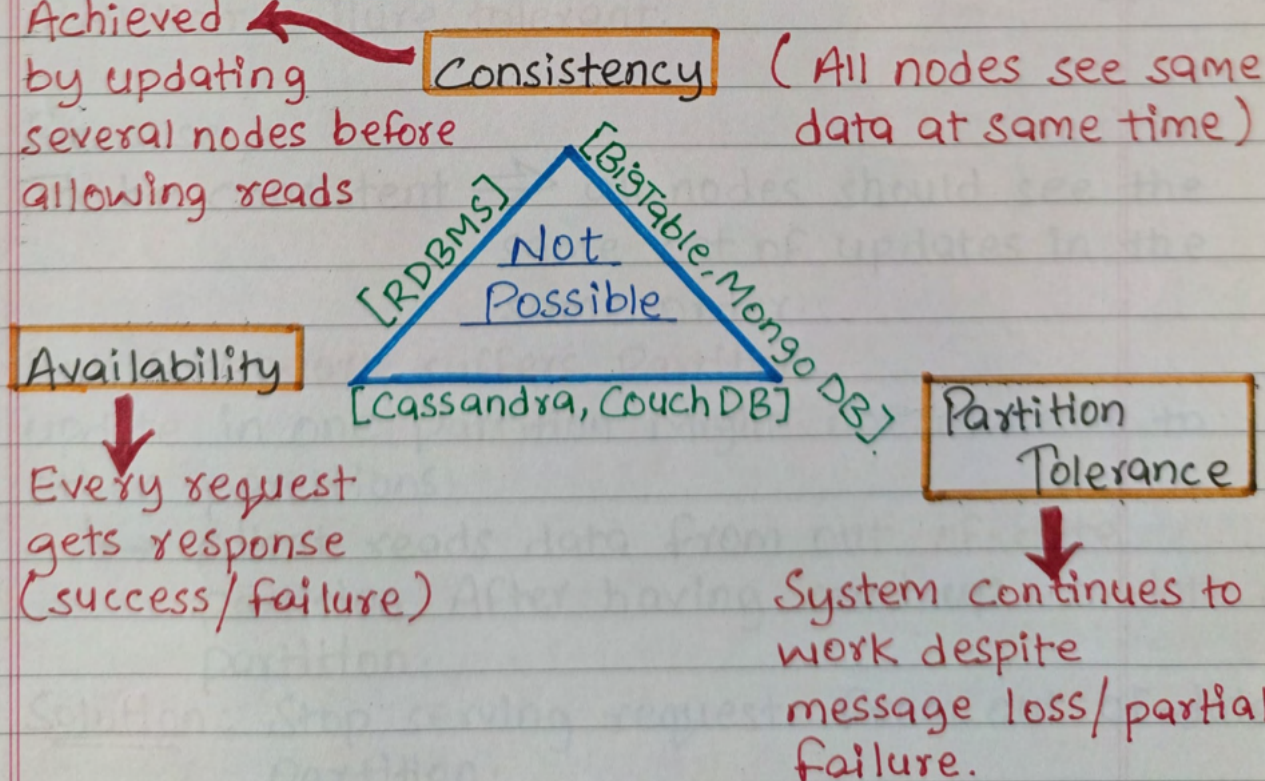
1). To store large volumes of data (little / no structure)

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 DBs like Cassandra ⇨ designed to scale across multiple data centers out of the box.

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

CAP Theorem.



Achieved by replicating data across different servers.

(can sustain any amount of network failure without resulting in failure of entire network.)

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

It is impossible for a distributed system to simuntaneously provide more than two of three of the above gurantees.

We cannot build a datastore which is :

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

Because,
To be consistent ⇒ all nodes should see the same set of updates in the 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 **up**-to-date partition.

Solution: Stop serving requests from out-of-date partition.

↳ Service is no longer 100% available.

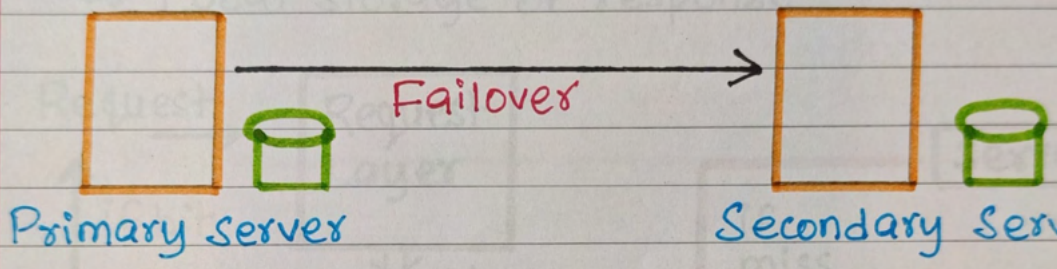
Redundancy & Replication.

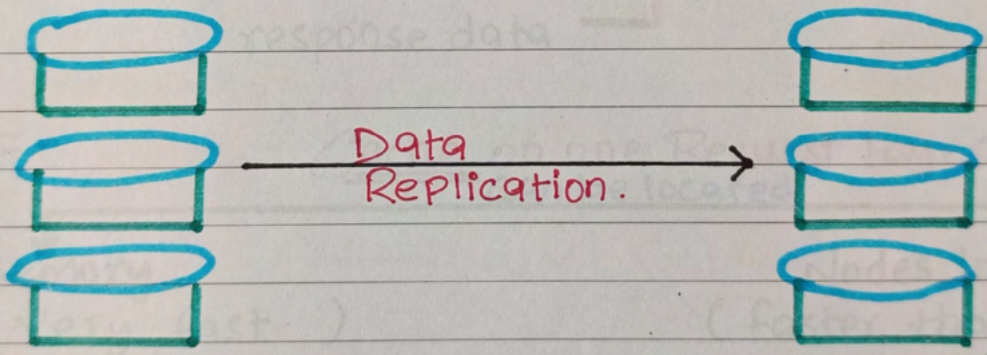
⇒ Duplication of critical data & Services.
↳ increasing reliability of system.

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

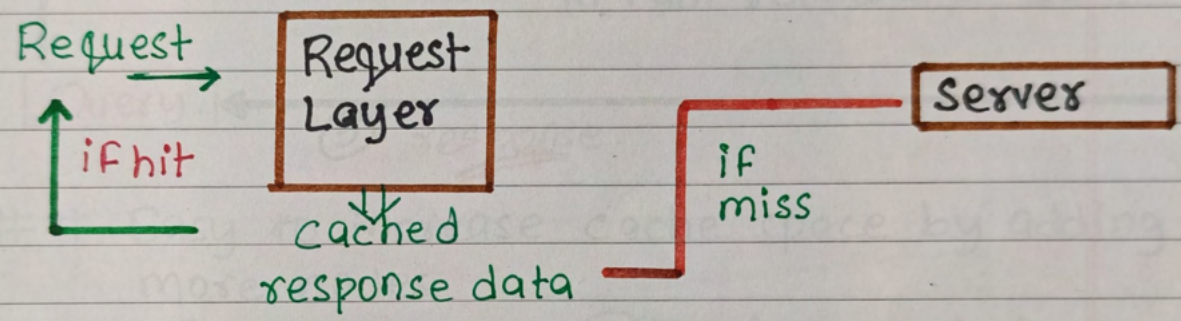
No single points of failure

Caching

ATUL KUMAR (LINKEDIN).
NOTES GALLERY (TELEGRAM).

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

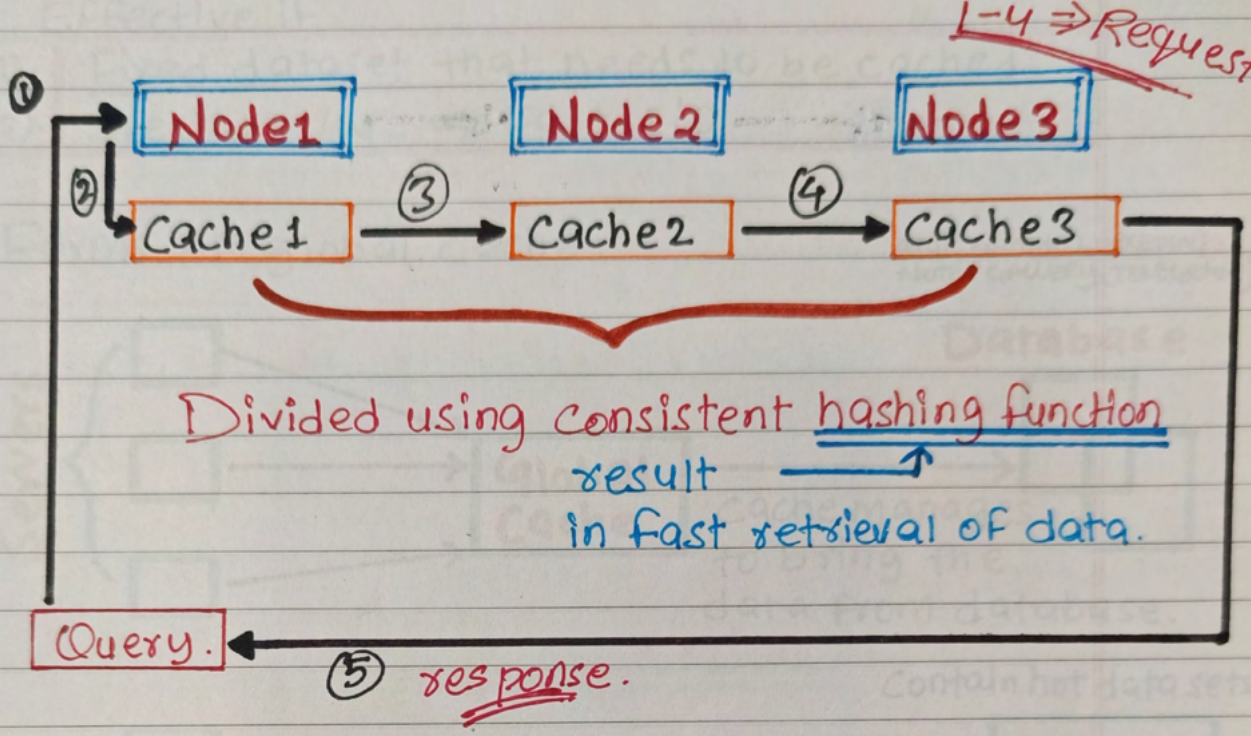
- Memory (very fast)
- Node's local disk. (faster than going to network storage)

: If LB distributes request randomly
Bottleneck ↳ 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.

Disadvantages: Resolving a missing node. starting multiple copies of data on different nodes. ← can be handled by.

⟹ Were making it more complicated.

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

ATUL KUMAR (LINKEDIN).
NOTES GALLERY (TELEGRAM).

Global Cache.

Single cache space for all the nodes.
↳ Adding a cache servers / File store (faster than Original store)

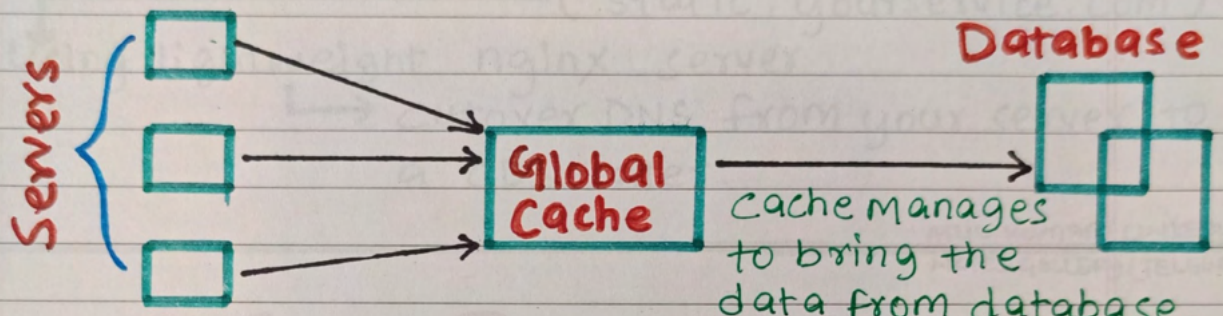
Difficult to manage if no clients / request increases.

Effective if

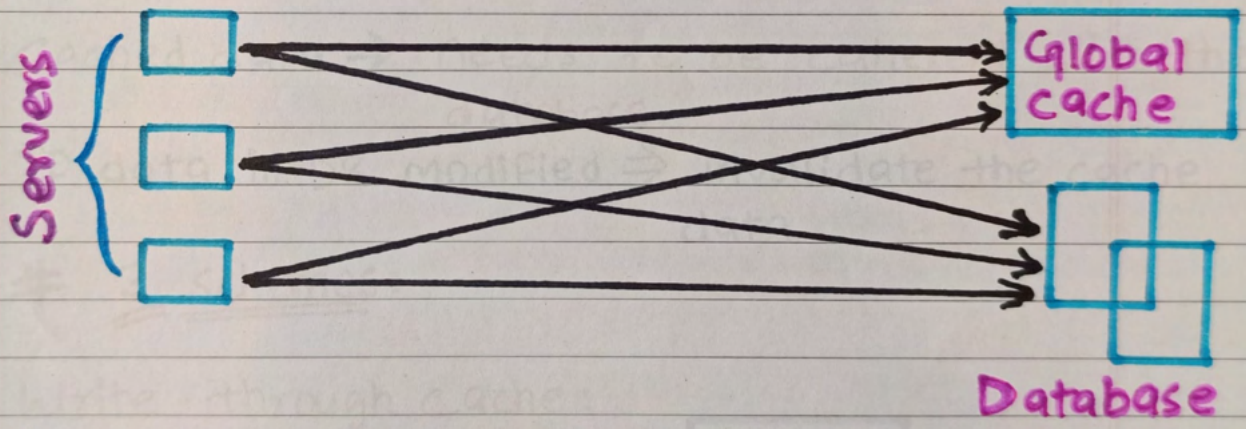
1). Fixed dataset that needs to be cached.
2). Special H/W ⇒ Fast I/O.

Forms of global cache:

ATUL KUMAR (LINKEDIN).
NOTES GALLERY (TELEGRAM).



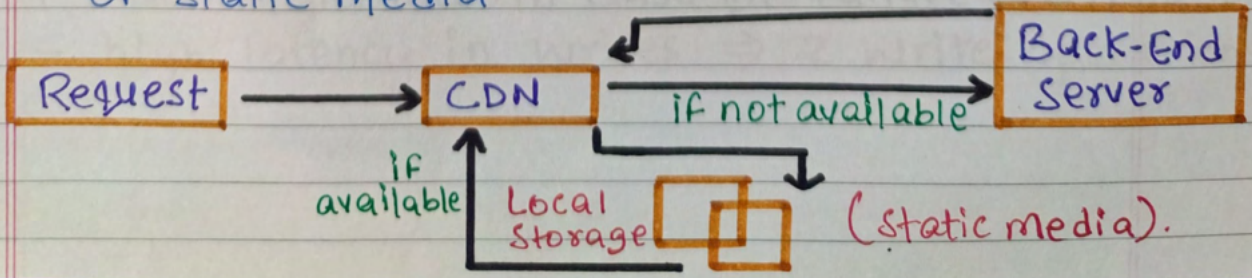
Contain hot data sets.



Application I logic understands the strategy / hot spots better than 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 better & easy future transition.

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

Using lightweight nginx server

→ cutover DNS from your server to a CDN later.

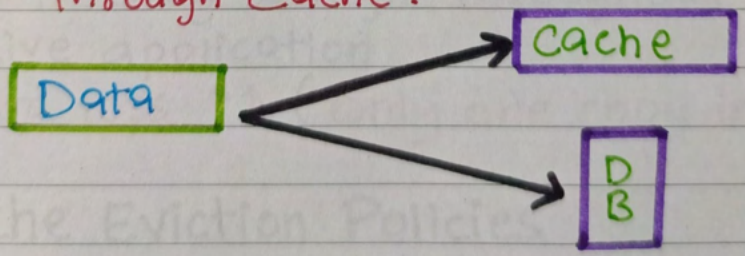
Cache Invalidation

Cached data ⇒ needs to be caherent with the database.

If data in DB modified ⇒ invalidate the cache data.

3 schemes:

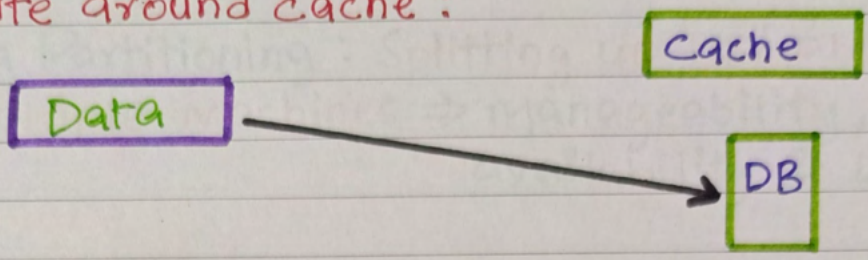
1). Write-through cache:



Data is written same time in both Cache & DB.

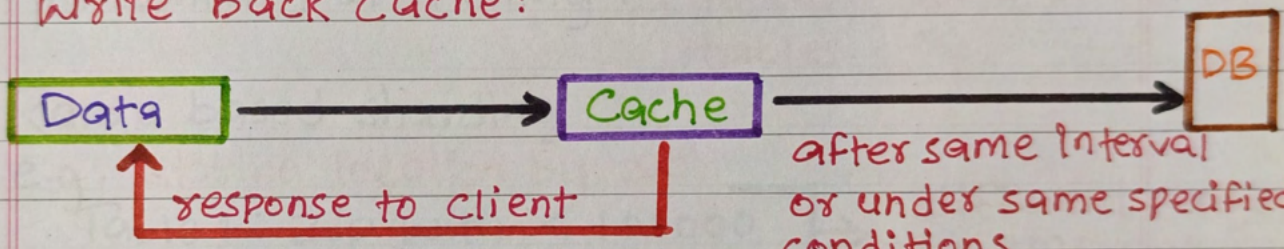
+ Complete data consistency (cache = DB)
+ Fault tolerance in case of failure (↓↓ data loss)
- high latency in writes ⇒ 2 write operations.

2). Write around cache :



+ No cache I for writes.
- read request for newly written data ⇒ Miss
higher latency.

3). Write back cache :



after same interval or under same specified conditions data is written to DB from cache.

+ low latency & high throughout for write-intensive application.
- 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 of vertical scaling by adding buffer services.

ATUL KOMAR (LINKEDIN).
NOTES GALLERY (TELEGRAM).

Methods of Partitioning :

1). Horizontal Partitioning : Different rows into diff. tables.

Range based sharding

e.g. storing location by zip

Table 1 : zip with < 100000
Table 2 : zip 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.

e.g. 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 millions users.

⇨ Directory based Partitioning

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

Mapping (tuple Key → DB Server)

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 / partition.

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 key.] → 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 reduces the key space to a size that can be listed.

Common Problems of sharding:

Sharded DB: Extra constraints on the different 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 ⇒ Straight forward.

★ not possible to perform joins on standard tables.

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

Workaround ⇒ Denormalize the DB.

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

→ Cons: Perils of denormalization.

↳ data inconsistency.

---

2). Referential 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 jobs to clean up dangling reference).

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.
↳ downtime.

We can use directly-based partitioning.
↳ highly complex
↳ Single point of failure.
(lookup service/table)

Indexes

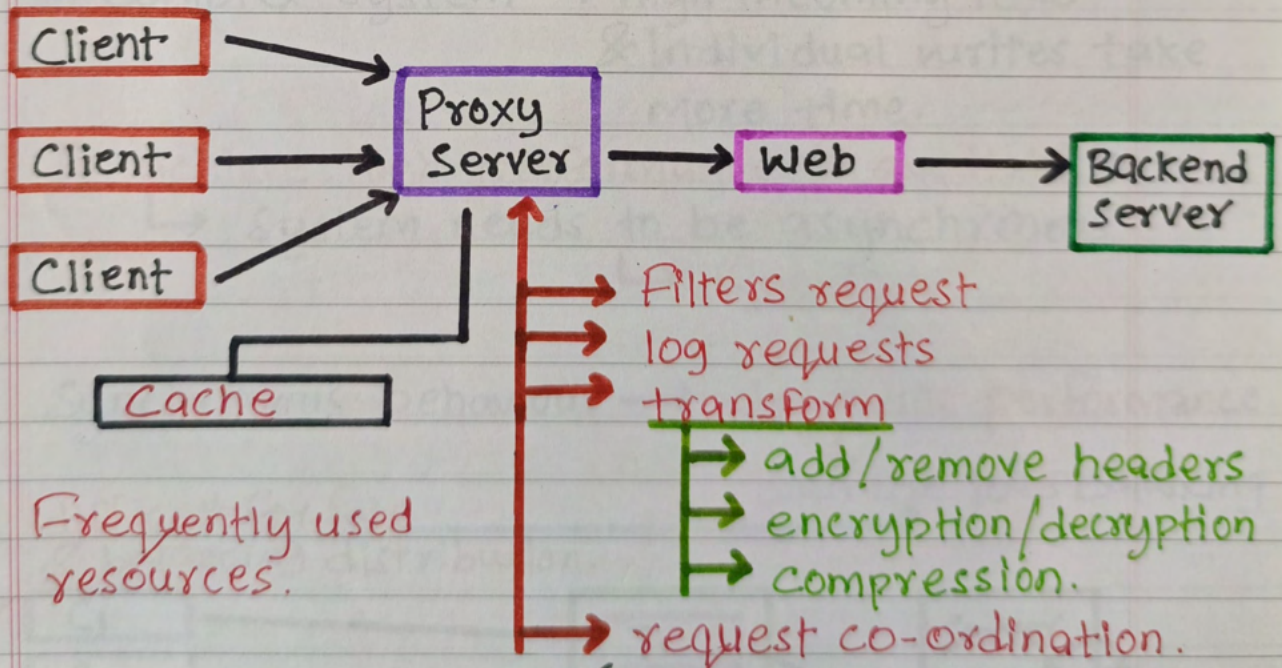
⇒ Well known because of database.
⇒ Improves speed of retrieval.
— Increased storage overhead.
— Slower writes.
↳ Write the data.
↳ Update the index.
⇒ Can be created using one or more columns.
★ Rapid random lookups.
& efficient access of ordered records.

Data Structure.
Column ⟶ Pointer to whole raw.

→ Create different views of the same data.
↳ Very good for filtering / sorting of large data sets.
↳ no need to create additional copies.

Using for datasets (TB in size) & small payload (KB).
Spread 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

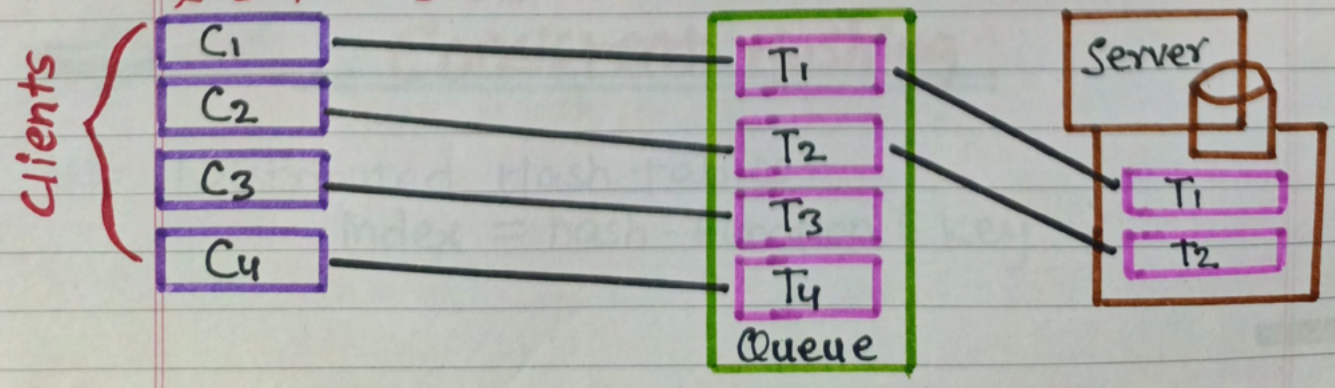
⇒ Effectively manages requests in large-scale distributed system.

→ 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
↳ Queue

Synchronous behaviour → degrades performance.
↓
Can use load balancing.
difficult for fair & balancing distribution.



Queues: asynchronous communication protocol.
- Client sends task
  - get ACK from queue (receipt)
    - Server as refrences for the results in future.
  - Client continues its work.

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

Queue: Provides fault tolerance.
- Protection from service outage / failure.
- highly request
  - retry failure service request
- Enforce Quality of services gurantee (Does NOT expose clients to outage)

Queues: distributed communication.
- Open Source Implementation.
  - RabbitMQ, ZeroMQ, activeMQ, BeanstalkD.

Consistent Hashing

Distributed Hash tables

index = hash-function (key)

Suppose were designing distributed caching system with n cache services

↳ hash-function ⇒ (Key % n)

Drawbacks:

1). NOT horizontally scalable

↳ addition of new server results in

↳ needs 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.

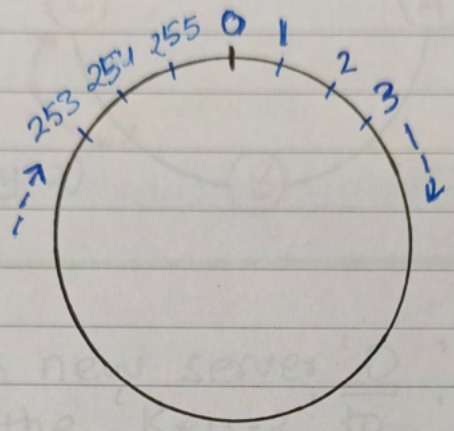
K = total numbers of keys

n = number of servers.

- How it works?

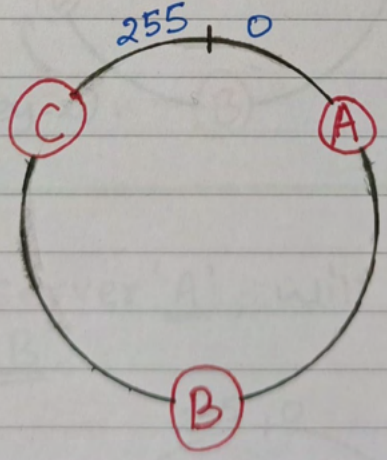
→ Typical hash function suppose outputs in the range [0, 256]

In consistent hashing,
imagine all of these integers are placed on a ring.

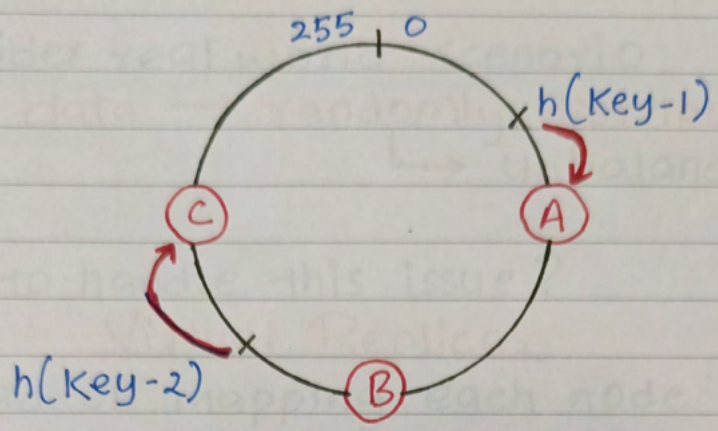


& we have 3 servers : A, B & C.

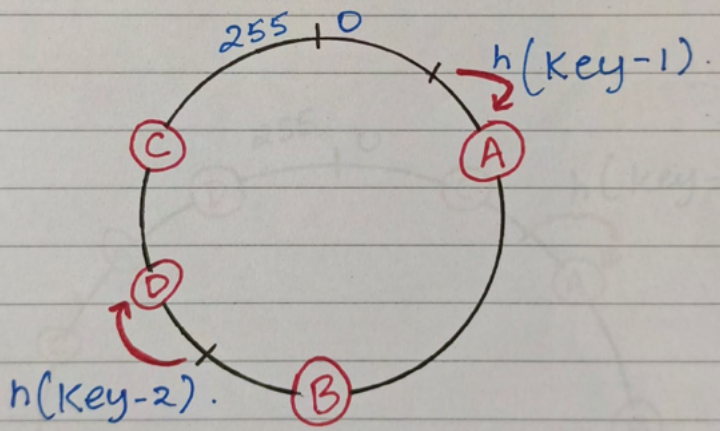
1). Given a list of servers, hash them to integers in the range.



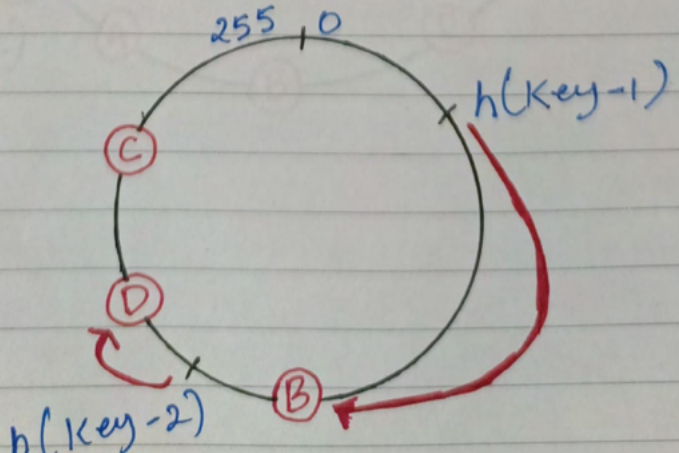
2). Map key to a server:
   a). Hash it to single integer
   b). Move clockwise until you find server
   c). Map key to that server.



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

data → randomly distributed

↳ unbalanced caches.

How to handle this issue?

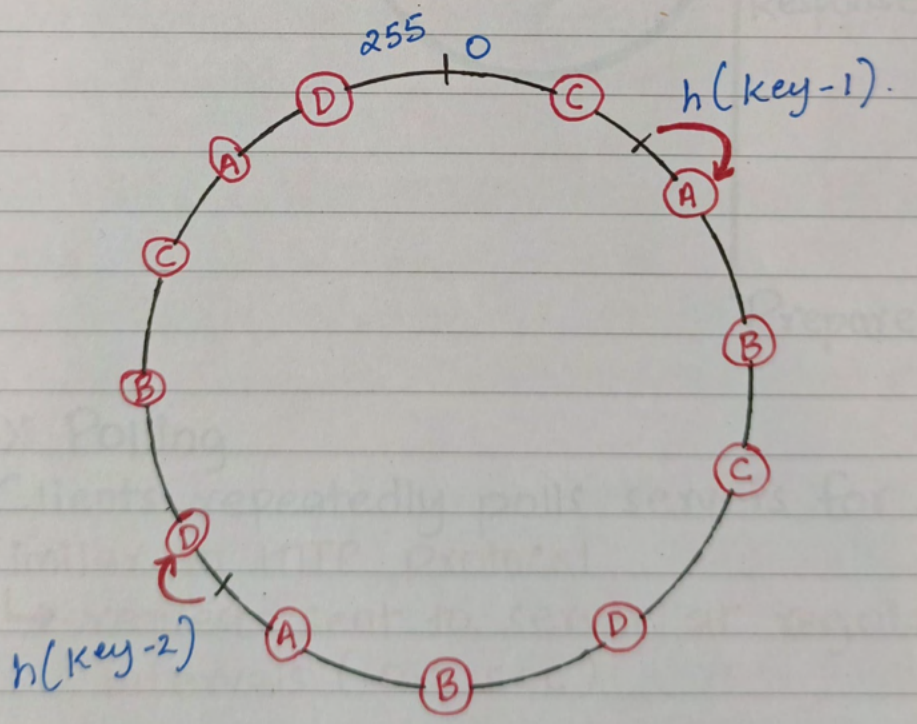
Virtual Replicas

⇒ Instead of mapping each node to a single point we map it to Multiple points.

↳ (More number of replicas

↳ More equal distribution

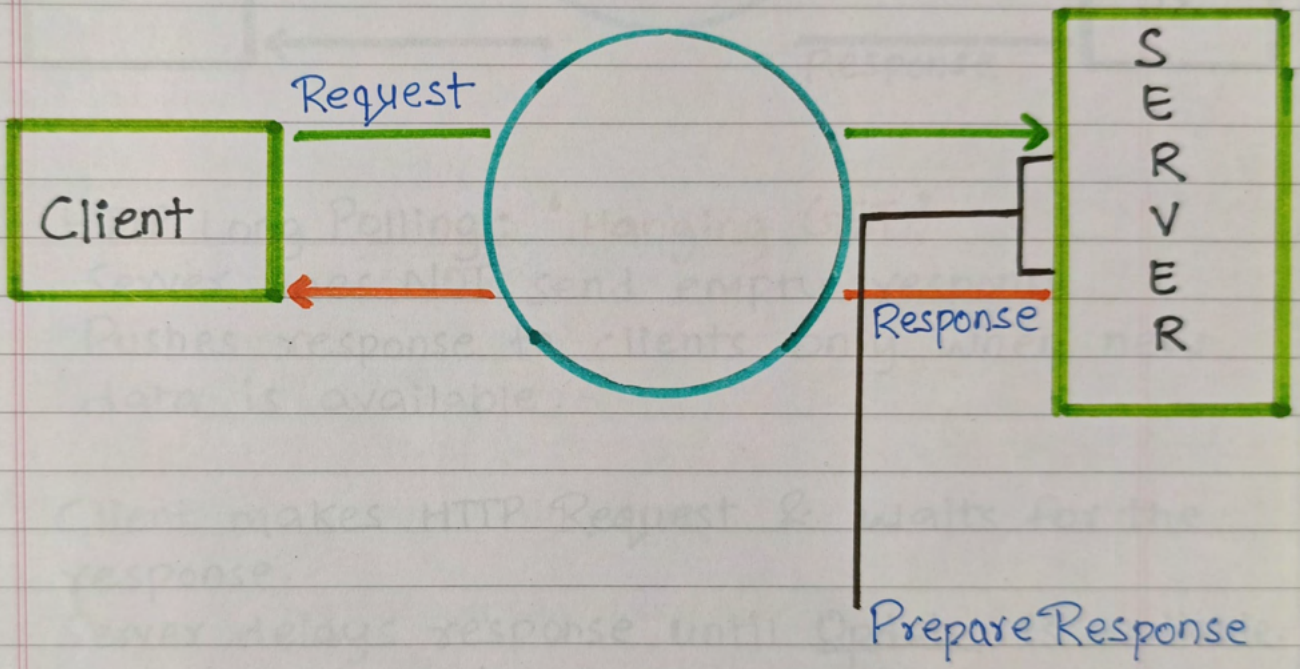
↳ good load balancing)



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

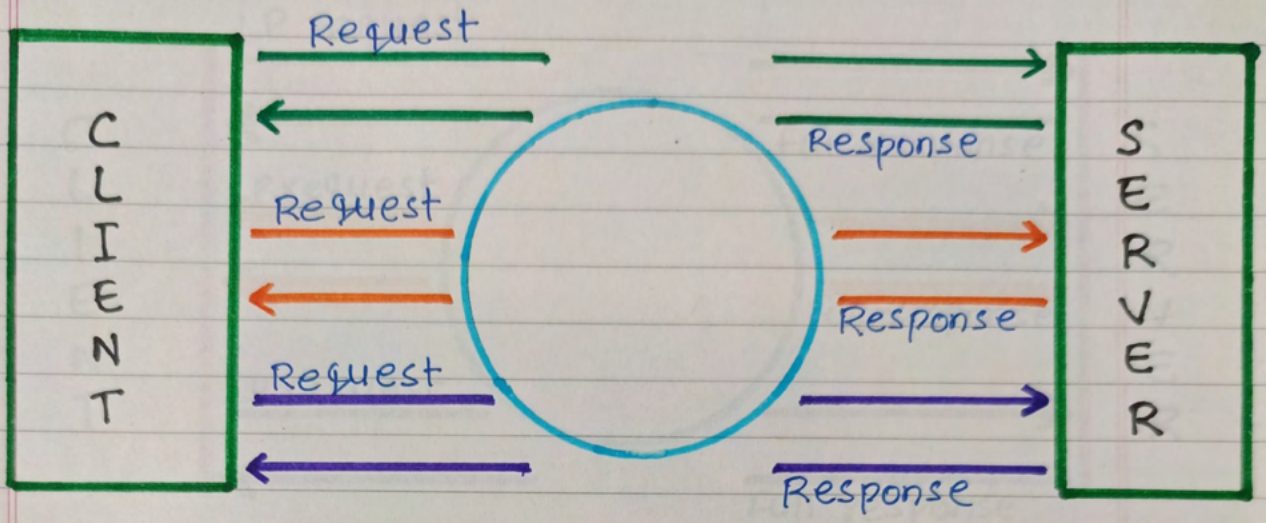
↳ request sent to server at regular intervals (0.5 sec)

Drawbacks:

Client keeps asking the server new data

↳ Lot of response are 'empty'

↳ HTTP Overhead.



HTTP Long Polling: 'Hanging GET'

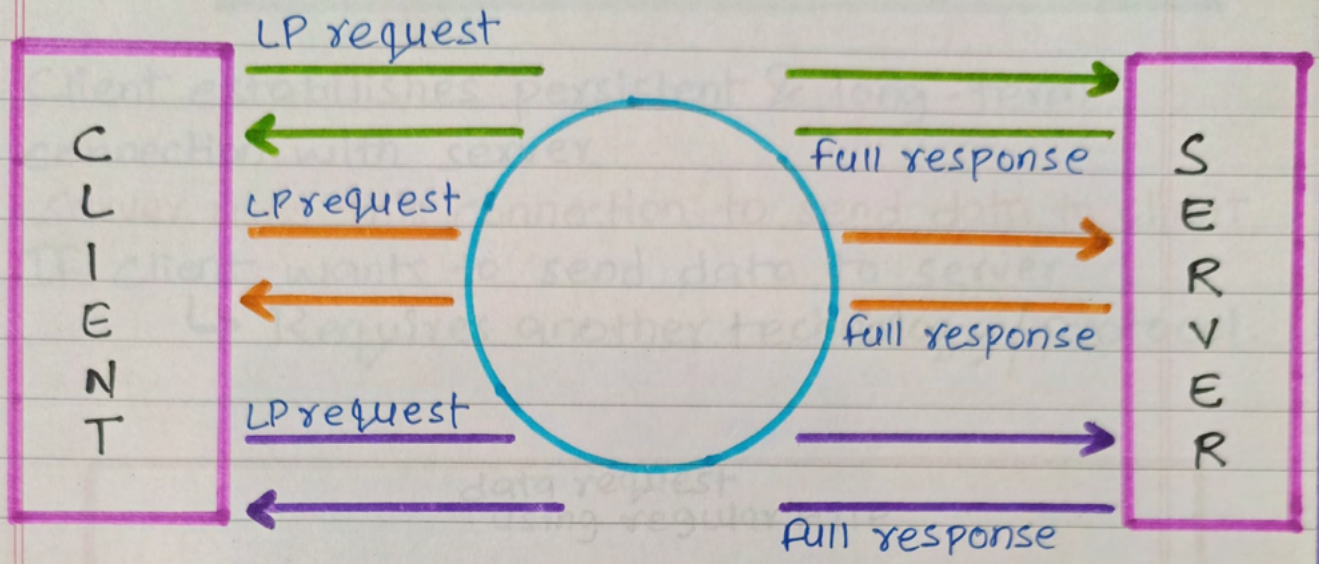
Server does NOT send empty response.
Pushes response to clients only when new data is available.

1). Client makes HTTP Request & waits for the response.
2). Server delays response until Update is available. or until timeout occurs.

3). When Update → Server sends full response.
4). Client sends new long-poll request.

a). immediately after recieving response.
b). after a Pause to allow acceptable latency Period.

5). Each request has timeout.
Client needs to reconnect periodically due to timeouts.

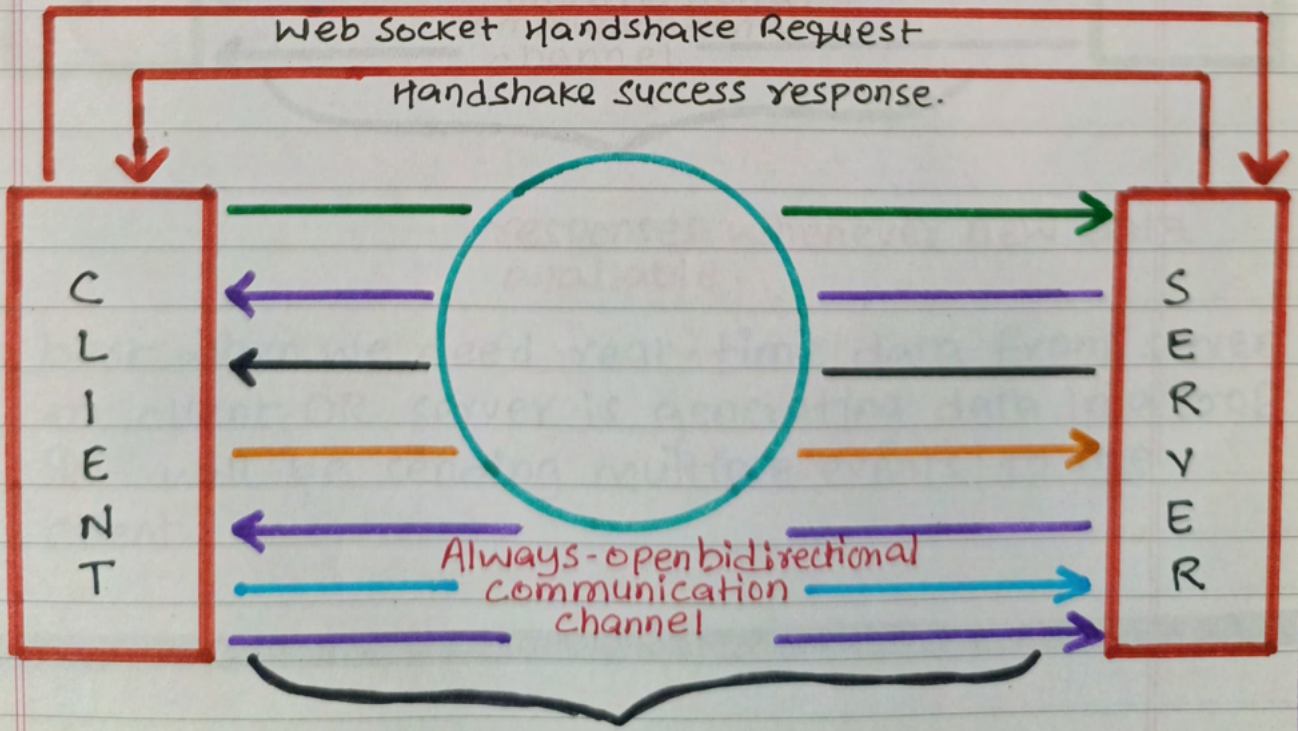


Web Sockets

→ Full duplex communication channel over single TCP connection.

→ Provides 'Persistent communication'
(client & server can send data at anytime)

→ bidirectional communication is always open channel.



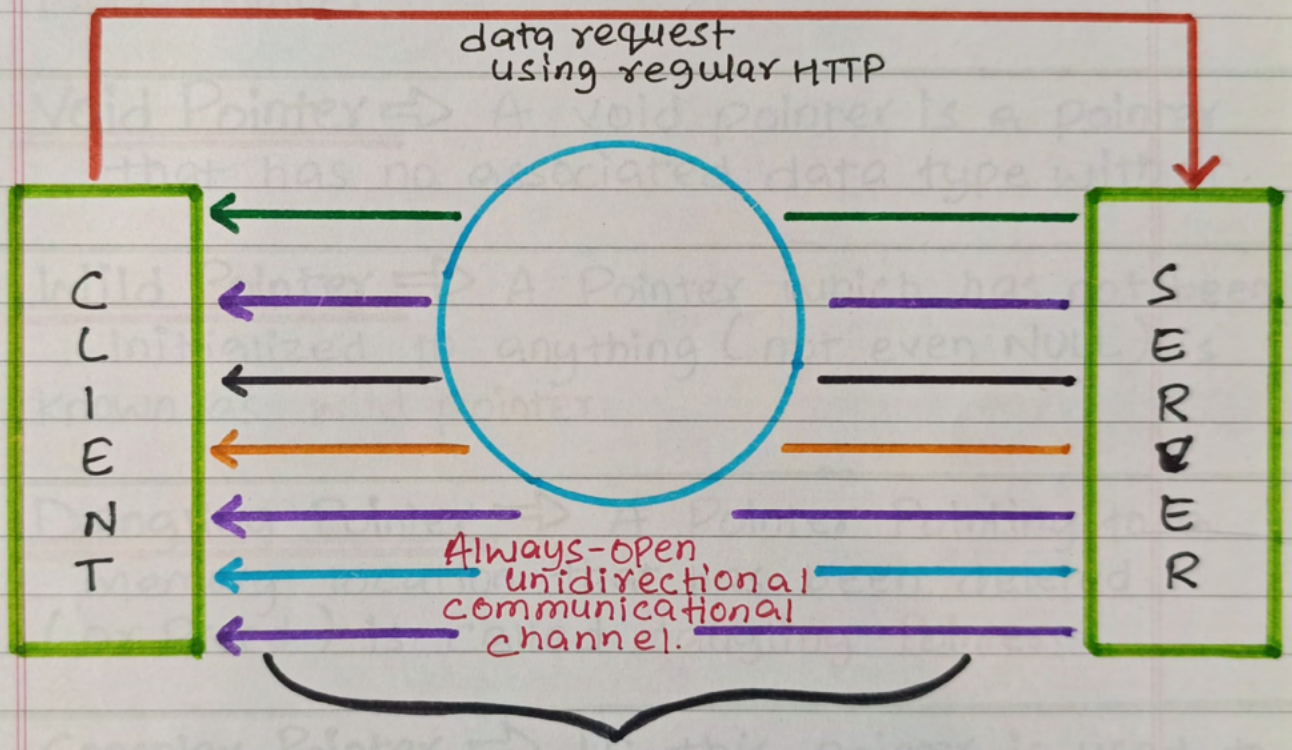
→ Lower Overheads.
→ Real time data transfer.

Server- Sent Events (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 server
↳ 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 events to the client.