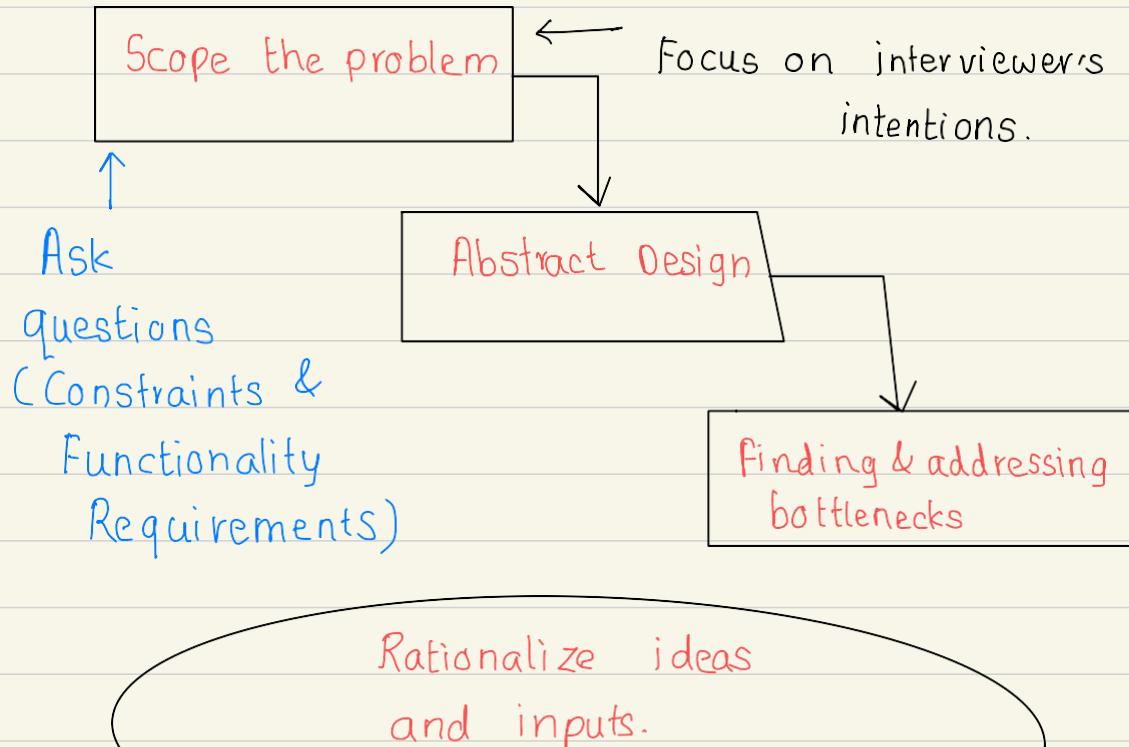


# System Design Basics

①

- 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.



# System Design Basics

(Contd.)

②

- 1) Architectural pieces / resources available
- 2) How these resources work together
- 3) Utilization & Tradeoffs

Consistent Hashing	
CAP Theorem	✓
Load balancing	✓
Queues	
Caching	✓
Replication	✓
SQL vs No-SQL	✓
Indexes	✓
Proxies	
Data Partitioning	✓

# Load Balancing

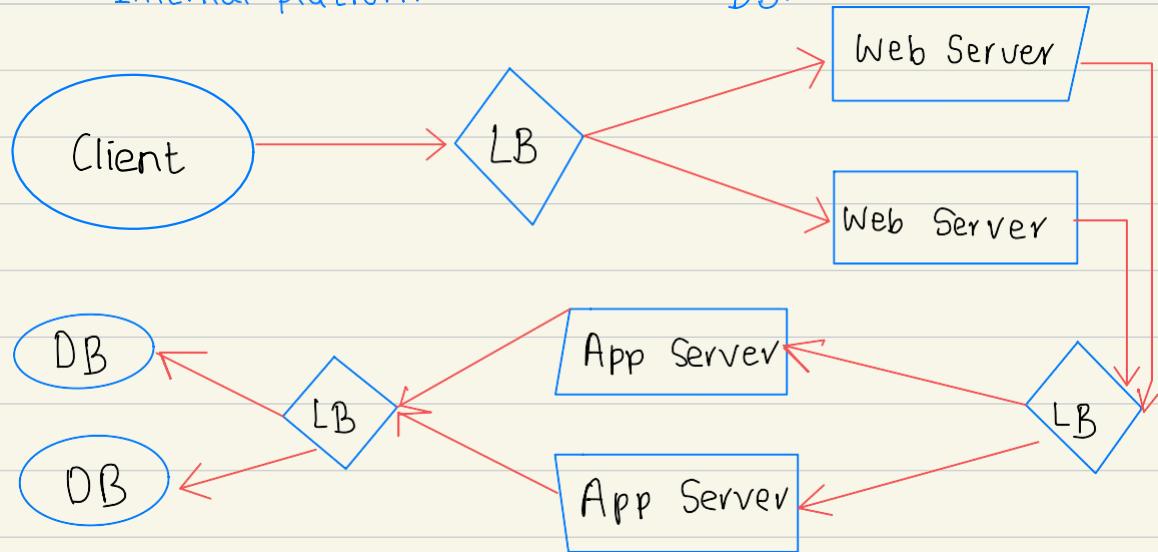
(Distributed System)

Types of distribution

- Random
- Round-robin
- Random (weights for memory & CPU cycles)

To utilize full scalability & redundancy, add 3 LB

- 1) User  $\xleftarrow{LB1}$  Web Server
- 2) Web Server  $\xleftarrow{LB2}$  App Server / Cache Server  
(Internal platform)
- 3) Internal platform  $\xleftarrow{LB3}$  DB.



## Smart Clients

Takes a pool of service hosts & balances load.

- detects hosts that are not responsive
- recovered hosts
- addition of new hosts

Load balancing functionality to DB (Cache, Service)

\* Attractive solution for developers

(Small scale systems)

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 1<sup>st</sup> point of contact to their system to serve user requests &

Intra network uses Smart clients / hybrid solution → (Next page) for

load balancing traffic.

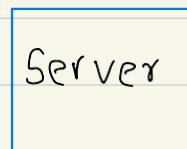
# Software Load Balancers

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

HAProxy ⇒ OSS Load balancer



1) Running on client machine



(locally bound port)

e.g. localhost :9000



↑ managed by HAProxy

(with efficient management  
of requests on the port)

2) Running on intermediate Server: Proxies running betn  
HAProxy diff. server side components

manages health checks

removal & addition of machines

balances requests a/c pools.

# World of Databases

## SQL vs. NoSQL

Relational  
Database

Non-relational  
Database

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

Row  $\Rightarrow$  One Entity Info

Column  $\Rightarrow$  Separate data points

MySQL  
Oracle  
MS SQL Server  
SQLite  
Postgres  
MariaDB

- 1) Unstructured,
- 2) distributed
- 3) dynamic schema

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

# NoSQL

Key - Value Store

Document DB

Wide - Column DB

Graph DB

Data  $\Rightarrow$  array  
of key - value pair

Key  $\Rightarrow$  attribute

Value  $\leftarrow$  linked to

Data  $\Rightarrow$  documents  
 $\downarrow$  grouped  
into

Collections

Each doc can be  
different.

Container  
 $\hookrightarrow$  for rows

No need of  
knowing all

columns upfront.

Each row  $\Rightarrow$   
diff. no of columns.

Analysis of large  
datasets.

Redis

Voldemort

Dynamo

Couch DB

MongoDB

Neo 4J  
Infinite Graph

Cassandra  
HBase

Data whose relations  
are best represented  
in graphs.

$\Rightarrow$  Nodes (Entities)  
 $\Rightarrow$  Properties (Information  
of entities)  
 $\Rightarrow$  Lines (Connections  
between entities)

# High Level differences betn SQL & NoSQL

Property	SQL	NoSQL
<u>Storage</u>	Tables ( Row → Entity , Column → Data point) e.g. Student ( Branch, Id, Name)	Diff. data storage models . ( Key Value, document, graph, Columnar )
<u>Schema</u>	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 row to contain data.
<u>Querying</u>	SQL	UnQL ( Unstructured query language ). Queries focused on collection of documents. Diff. ØB ⇒ diff UnQL.
<u>Scalability</u>	Vertically Scalable ( + horsepower of hw ) Expensive possible to scale across multiple servers, ⇒ challenging & time - consuming.	Horizontally Scalable: Easy addition of servers. Hosted on cloud or cheap commodity hw. → Cost effective
<u>Reliability</u> or <u>ACID</u>	ACID * Compliant ⇒ Data Reliability ⇒ Guarantee of transactions	Sacrifice ACID Compliance for scalability & performance. ⇒ Still a better bet.
<u>Compliance</u>		( ACID - Atomicity, Consistency, Isolation, Durability )

# Reasons 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 app<sup>n</sup>

→ 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 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



Every request gets response (success / failure)

Achieved by replicating data across different servers

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

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

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

We cannot build a datastore which is :

- 1) Continually available
- 2) Sequentially consistent
- 3) partition failure tolerant.

Because,

To be consistent  $\Rightarrow$  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

$\hookrightarrow$  client reads data from out-of-date partition

After having read from up-to-date partition.

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

$\hookrightarrow$  Service is no longer 100% available.

# Redundancy & Replication

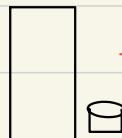
⇒ Duplication of critical data & services

↳ 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.

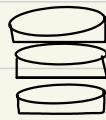


Primary Server

→ Failover

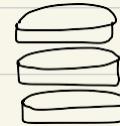


Secondary Server



Active data

→ Data  
Replication



Mirrored data

# Service Redundancy: Shared-nothing architecture.

Every node ⇒ independent. No central service managing state.

More resilient  
to failures

New servers  
addition without  
Special conditions

No single point of failure



Helps in  
Scalability

# Caching

Load balancing  $\Rightarrow$  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 requests randomly

↳ Same request  $\Rightarrow$  different nodes

More Cache miss

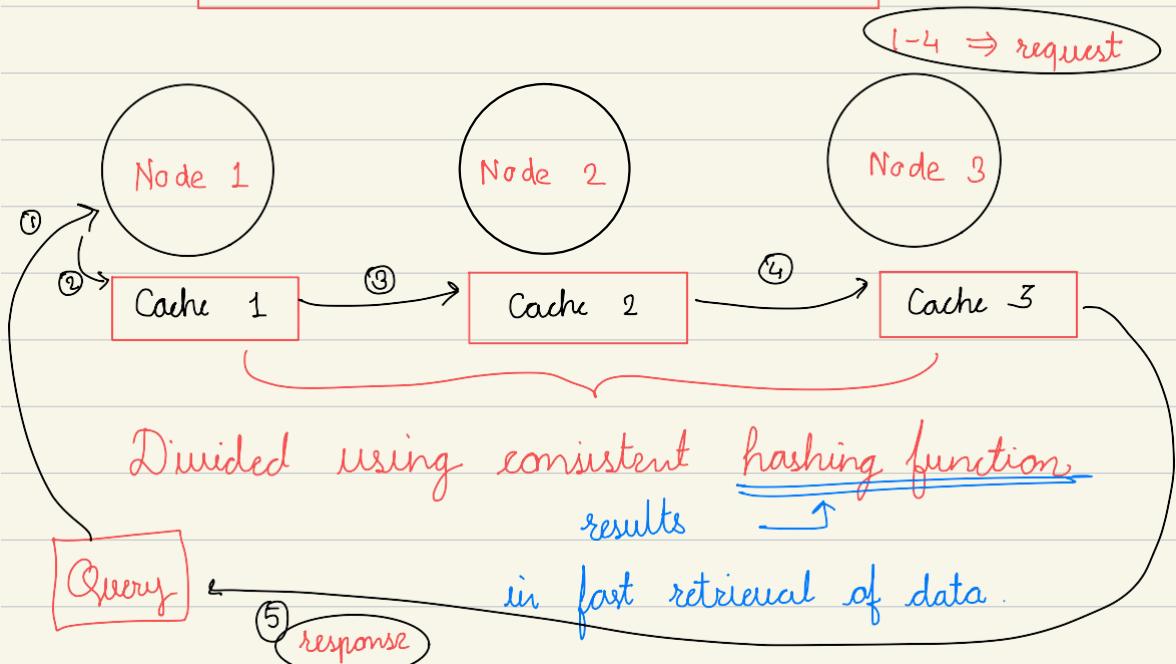


1) Global Caches

2) Distributed Caches

can be  
overcome  
by

## 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.

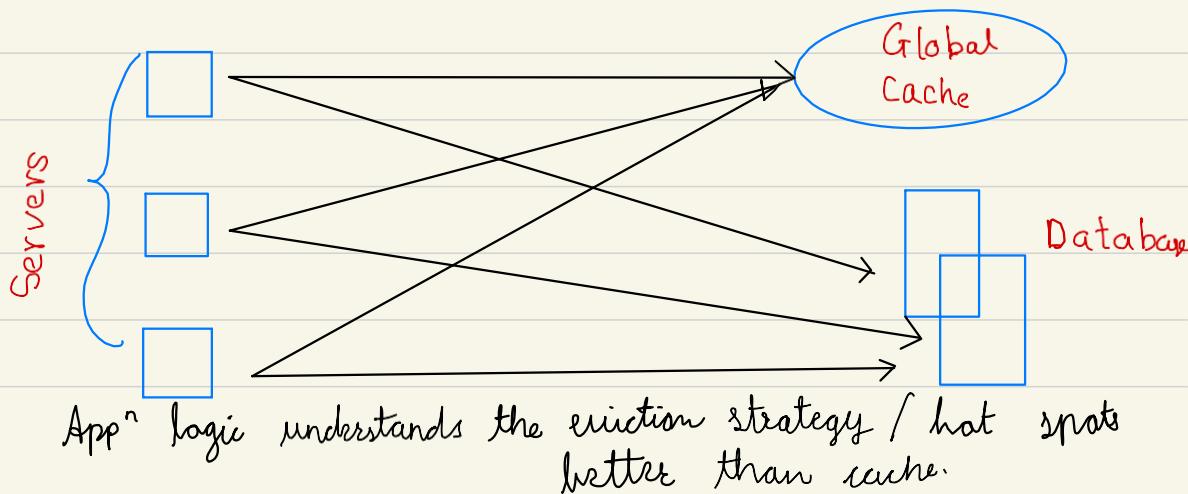
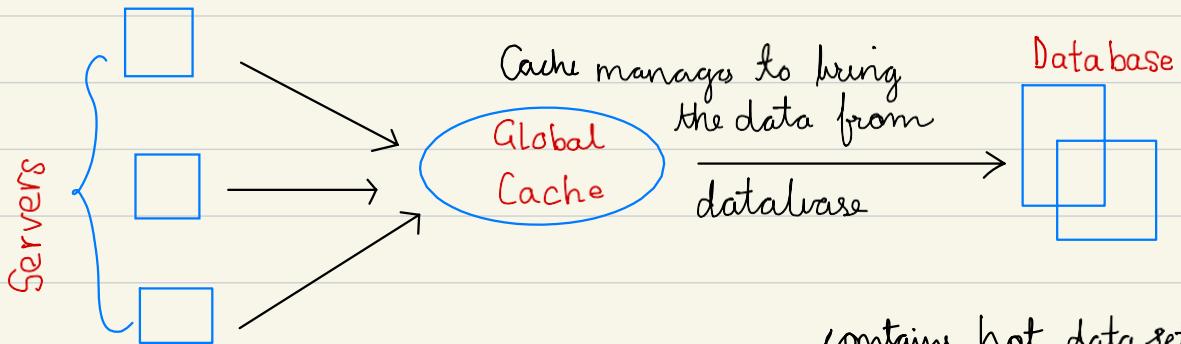
↳ Adding a cache server / file store (faster than original store)

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

Effective if

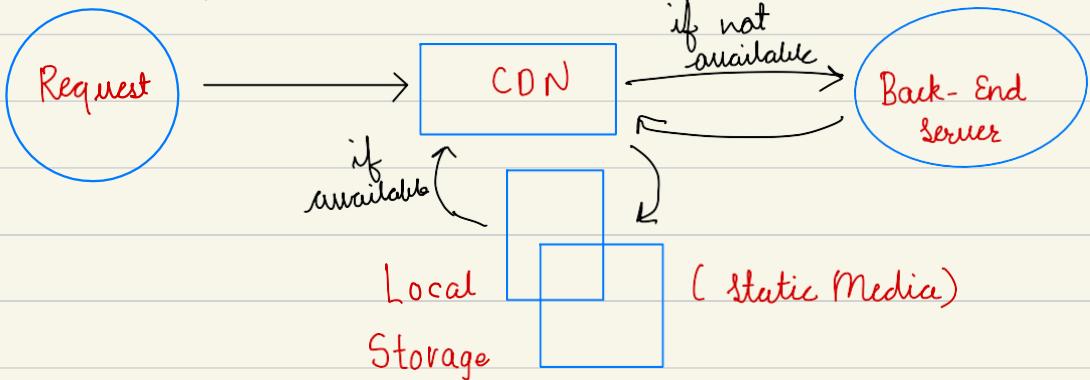
- 1) fixed dataset that needs to be cached
- 2) special H/w  $\Rightarrow$  fast I/O.

# Forms of global 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 separate subdomain

(static.yourservice.com)

Using lightweight Nginx server

↳ cutomer DNS from your server  
to a CDN later

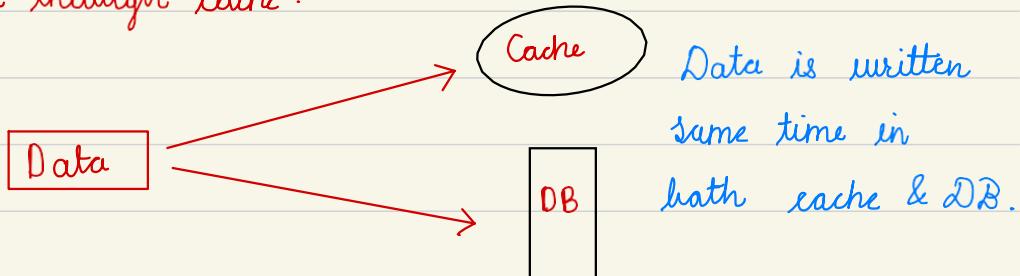
# Cache Invalidation

# Cached data  $\Rightarrow$  needs to be coherent with the database

If data in DB modified  $\Rightarrow$  invalidate the cached data.

# 3 schemes:

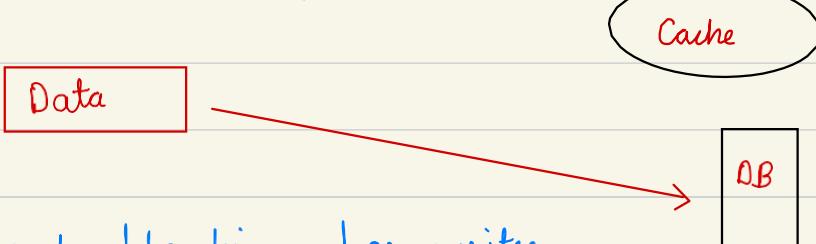
1) Write-through cache:



Data is written same time in both cache & DB.

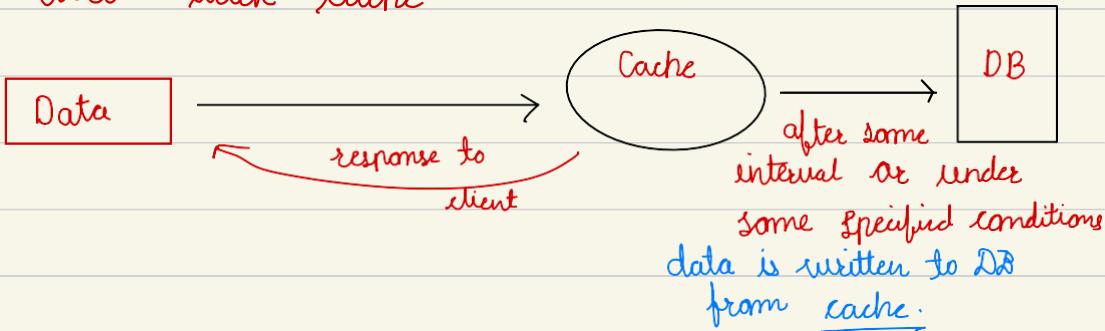
- + Complete data consistency (Cache = DB)
- + fault tolerance in case of failure (↓↓ data loss)
- high latency in writes  $\Rightarrow$  2 write operations

2) Write-around cache



- + No cache flooding for writes
- read request for newly written data  $\Rightarrow$  Miss  
higher latency  $\leftarrow$

3) Write back cache:



- + low latency & high throughput for write-intensive app
- 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  $\Rightarrow$  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 beefier servers.

# Methods of Partitioning:

1) Horizontal Partitioning : Different rows into diff. tables

Range based sharding

e.g. storing locations by zip

Table 1 : Zips with  $< 100\,000$

Table 2 : Zips with  $> 100\,000$

and so on

different ranges in different tables

\*\* Cons: if the value of the range not chosen carefully

$\Rightarrow$  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 billion photos by 140 mill users)

## Directory based partitioning

⇒ A loosely coupled approach to work around issues mentioned in above two partitionings.

\* \* Create lookup service ⇒ current partitioning scheme & abstracts it away from the DB access code.

## Mapping (tuple key → DB service)

Easy to add DB servers or change partitioning scheme.

# Partitioning Criteria

1) Key or Hash based partitioning :



# Effectively fixes the total number of servers/partitions

So if we add new server/partition →

change in hash function

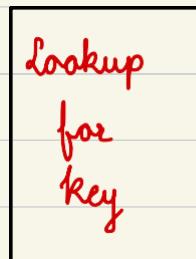
downtime [ ] of ←

redistribution ↗

Solution : Consistent hashing

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

new record →



→

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  
 $(i \bmod n)$

### 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 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  $\Rightarrow$  straight forward.

\* Not feasible to perform joins on sharded tables

$\hookrightarrow$  less efficient (data needs to be compiled from multiple servers)

# Workaround  $\Rightarrow$  Denormalize the DB

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

cons: Perils of denormalization

$\hookrightarrow$  data inconsistency

## 2) Referential integrity : Foreign Keys on sharded DB

$\hookrightarrow$  difficult

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

# If app" demands referential integrity on sharded DB

$\hookrightarrow$  enforce it in app" code (SQL jobs to clean up dangling references)

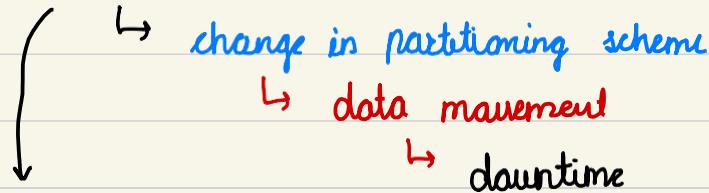
### 3) Rebalancing:

Reasons to change sharding scheme:

- Non-uniform distribution (data wise)
- Non-uniform load balancing (request wise)

Workaround: 1) add new DB

2) rebalance



We can use directory-based partitioning

↳ highly complex

↳ single point of failure  
(lookup service / table)

## 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

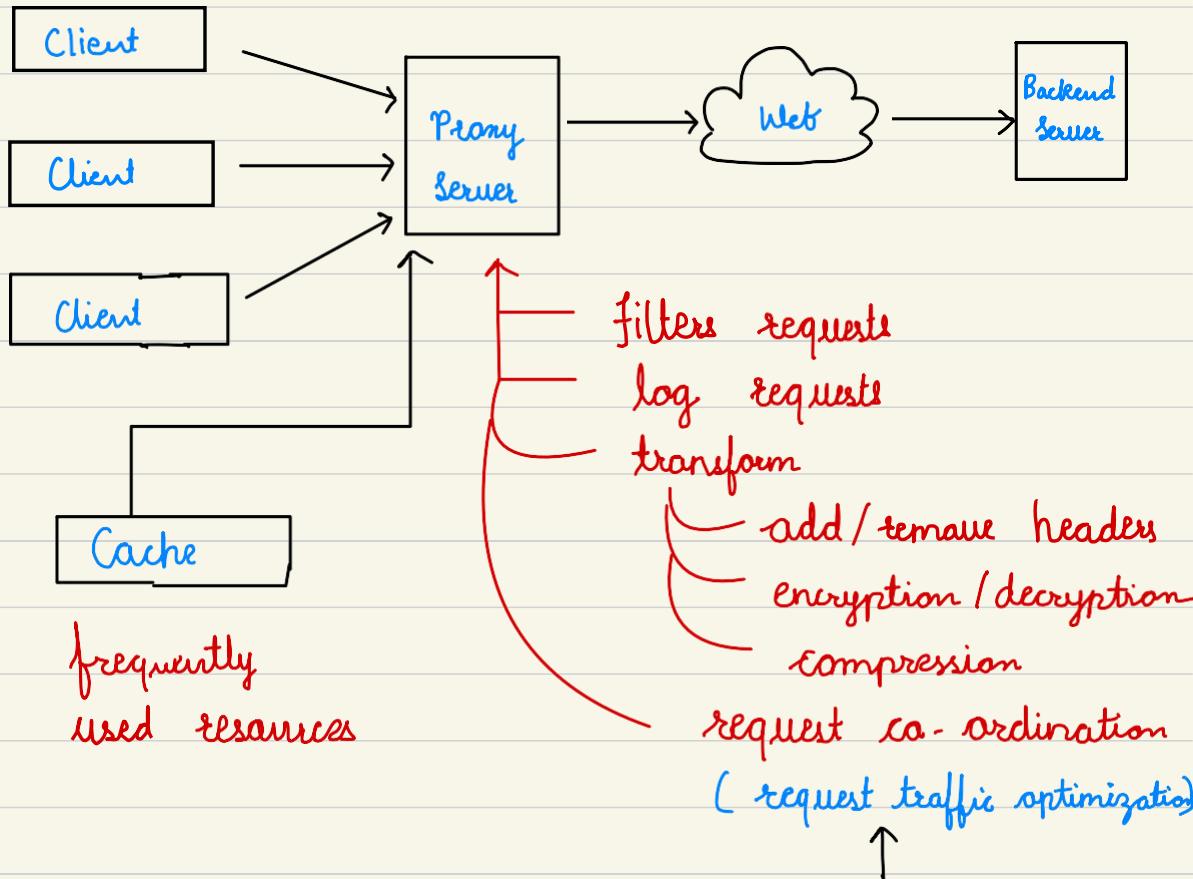
- Column → Pointer to whole row
- Create different views of the same data.
  - ↳ very good for filtering / sorting of large data sets.
  - ↳ no need to create additional copies.
- # Used for datasets (TB in size) & small payload (KB)
  - ↑
- spread over several physical devices → We need some way to 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 requests

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 systems → writes are fast.

→ In complex systems → high incoming load

& individual writes take more time

\* To achieve high performance & availability

↳ system needs to be asynchronous

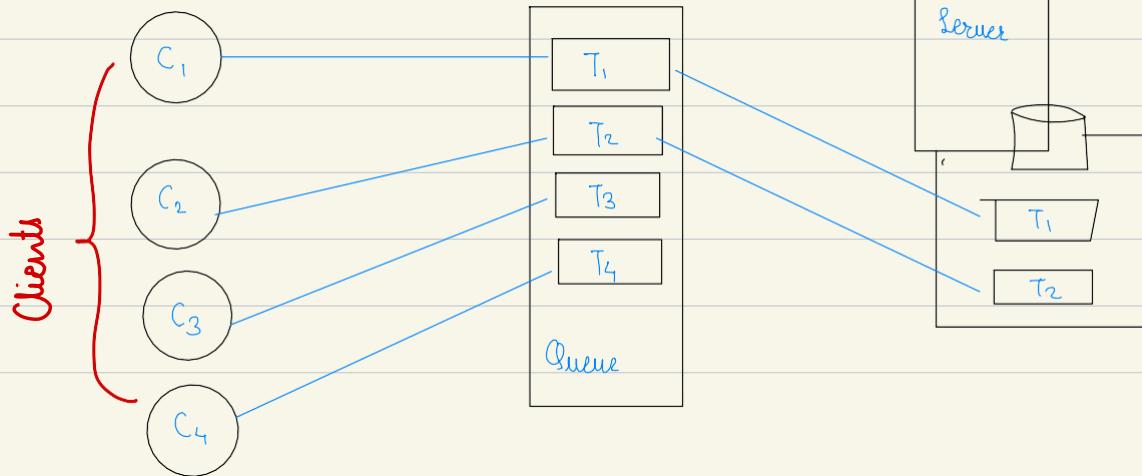
↳ Queues

# Synchronous behaviour → 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

    ↑                          ↳ protection from service outage/failure

    highly robust

    ↑                          ↳ retry failed service request

    Enforces Quality of Service guarantee

    ( Does NOT expose clients to outages )

# Queues : distributed communication

↳ Open Source implementations

    ↳ RabbitMQ, ZeroMQ, ActiveMQ, BeanstalkD.

## Consistent Hashing

# Distributed Hash Table

index = hash-function (key)

# Suppose we're designing distributed caching system with n cache servers

↳ hash-function  $\Rightarrow$  (key % n)

Drawbacks:

1) NOT horizontally scalable

↳ addition of new server 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?

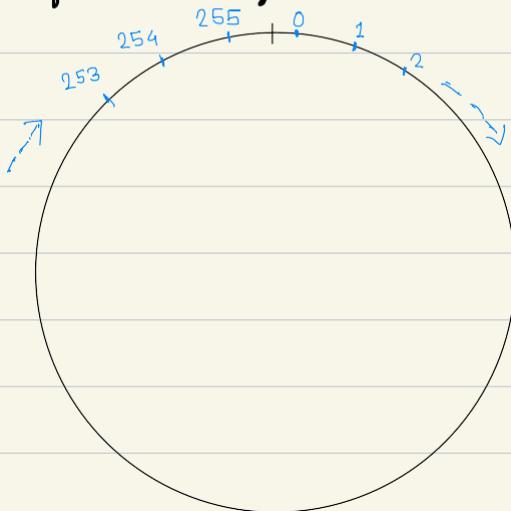
- Very useful strategy for distributed caching & DHTs.
- minimizes reorganization in scaling up / down.
- only  $\lceil k/n \rceil$  keys needs to be remapped.  
 $k \Rightarrow$  total number of keys  
 $n \Rightarrow$  number of servers

## How it works?

Typical hash function suppose outputs in  $[0, 256)$

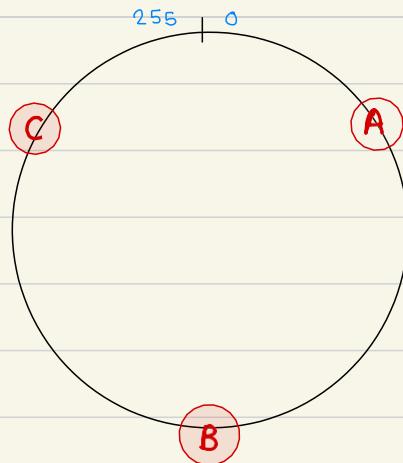
In consistent hashing,

imagine all of these integers are placed on a ring.



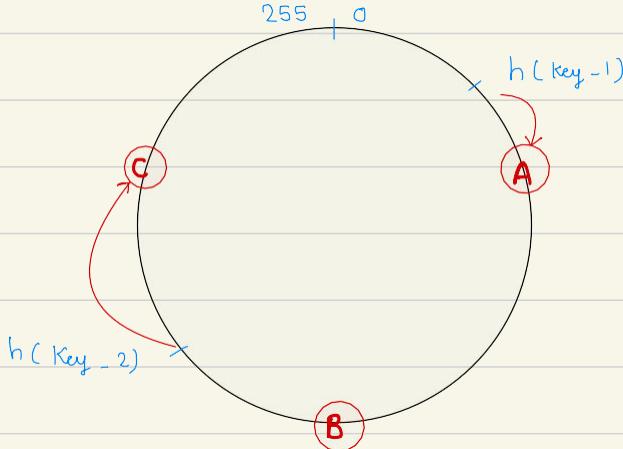
& we have 3 servers : A, B & C.

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

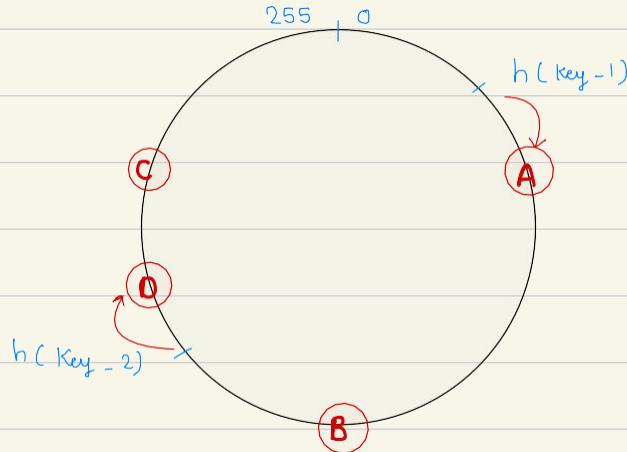


2) Map key to a server :

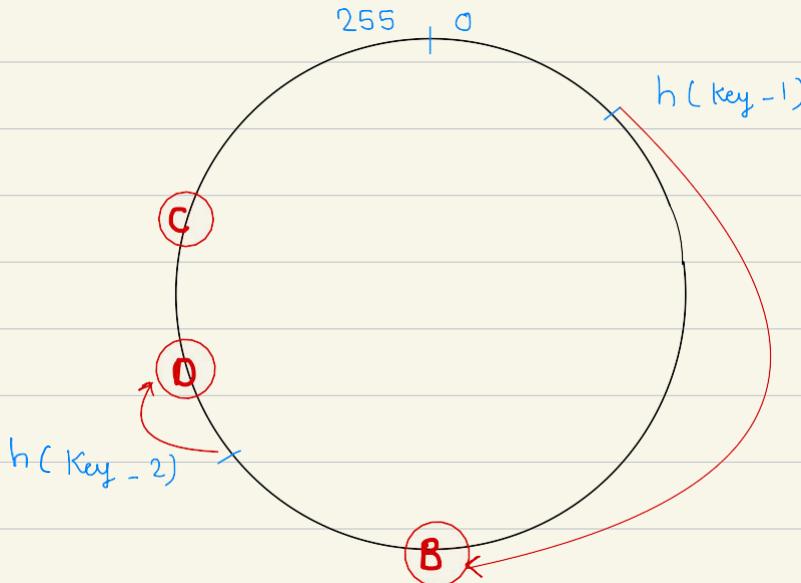
- Hash it to single integer
- Move CLK wise until you find server
- map key to that server.



Adding a new server 'D', will result in moving 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?

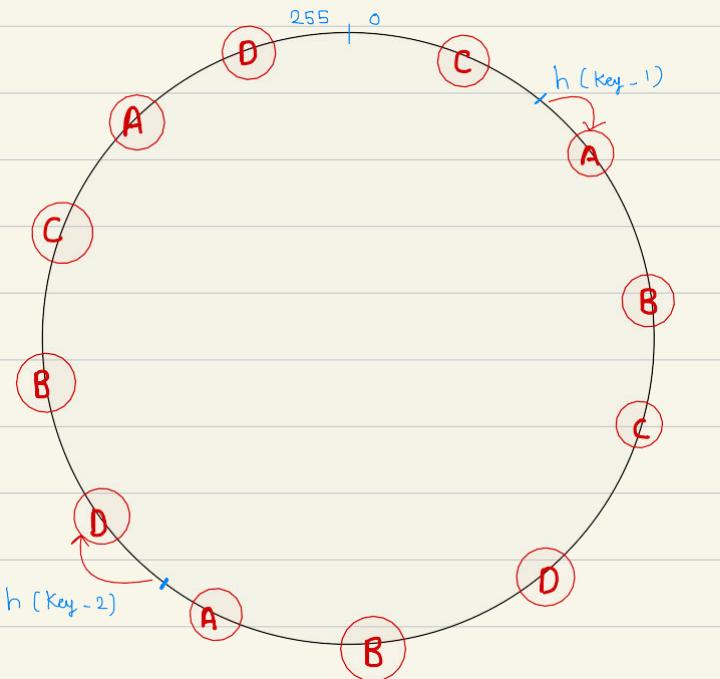
### Virtual Replicas

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

↳ (More number of replicas)

↳ more equal distribution

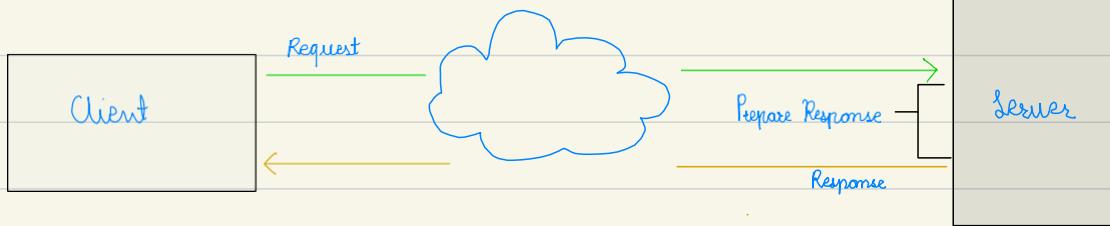
↳ good load balancing.)



# 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

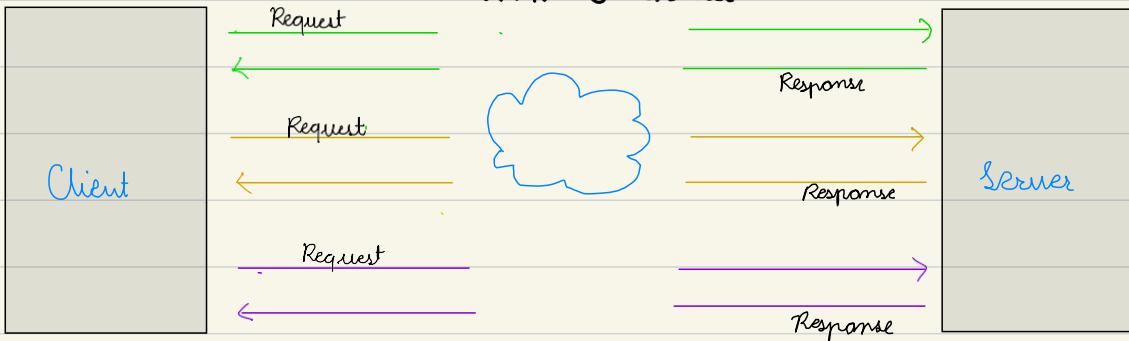
↳ requests sent to server at regular intervals (0.5 sec)

#### Drawbacks:

Client keeps asking the server new data

↳ Lot of responses are 'empty'

↳ HTTP Overhead.



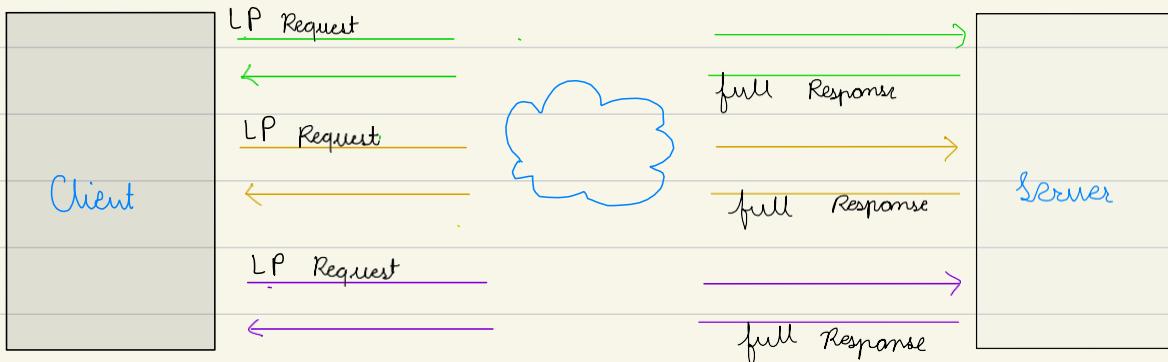
## # HTTP Long Polling: 'Hanging GET'

Server does NOT send empty response.

Pushes response to clients only when new data is available

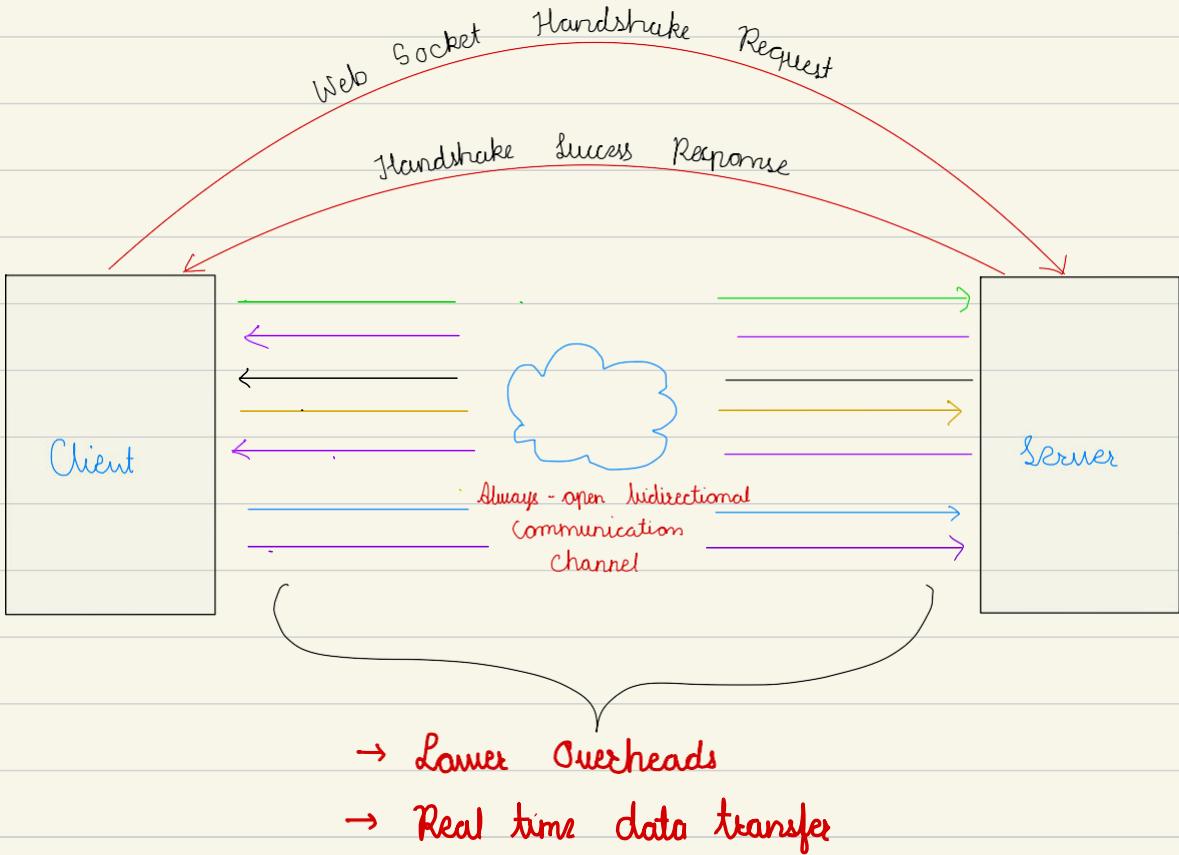
- 1) Client makes HTTP Request & waits for the response.
- 2) Server delays response until update is available  
or until timeout occurs.
- 3) When update → Server sends full response.
- 4) Client sends new long-poll request
  - a) immediately after receiving response
  - b) after a pause to allow acceptable 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.



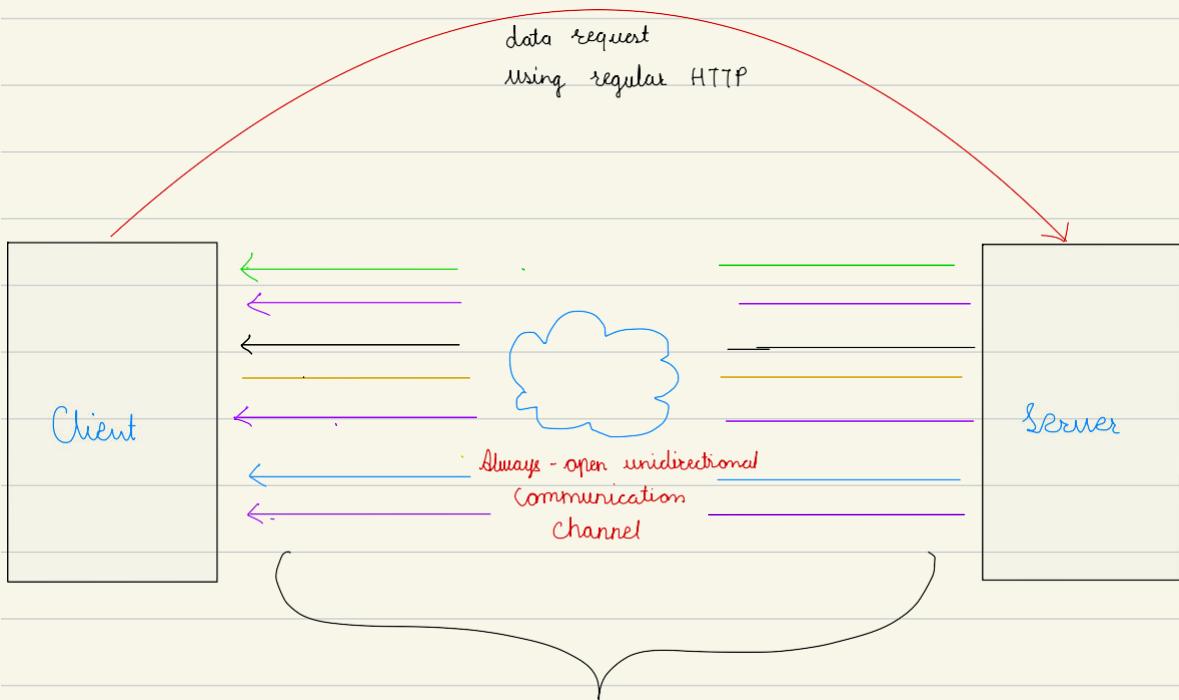
## Server - Sent Events (S S E)

Client establishes persistent &  connection with server

Server uses this connection to send data to client

\*\* If client wants to send data to server

↳ Requires another technology / protocol.



responses whenever new data available

→ 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 .

