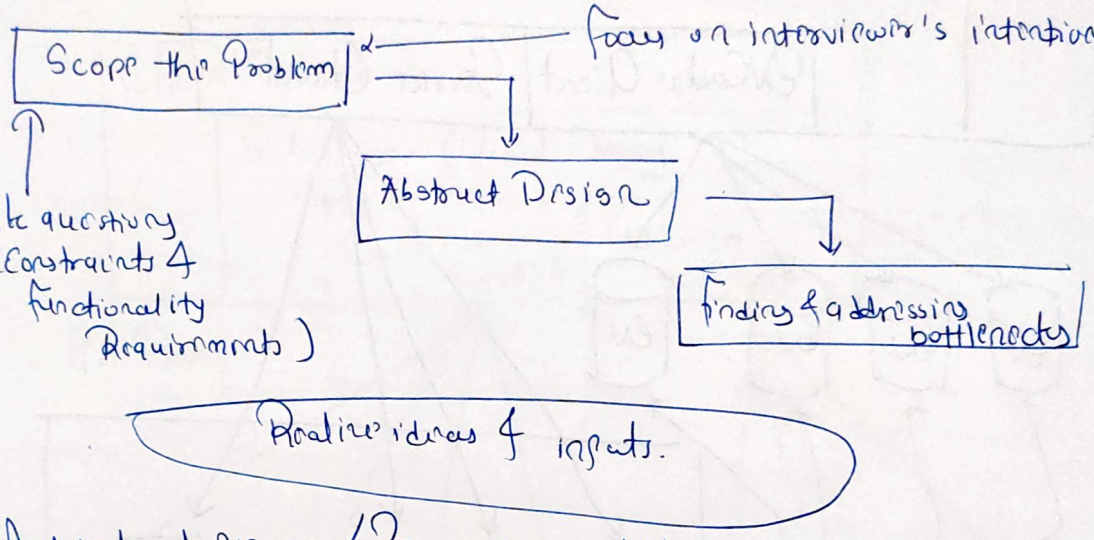
System Design Basics Amazon SDE-2

1) Try to Break the Problem into simpler modules (Top down approach)
2) Talk about trade offs
(NO solution is perfect)

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



1) Architectural pieces / Resources available
2) How these Resources work together
3) Utilization & Trade offs

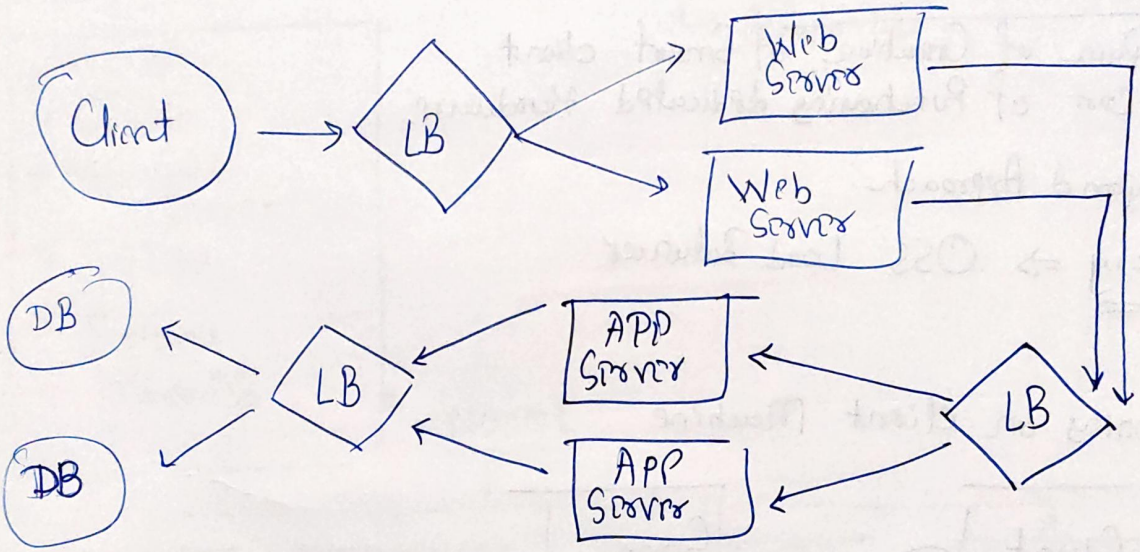
- Consistent Hashing
- CAP theorem
- Load balancing
- Queues
- Caching
- Replication
- SQL v/s NO SQL
- Indexes
- Proxies
- Data Partitioning

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

Takes a pool of Service hosts & balances load
- ↳ detects host 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. as use it as 1st point of Contact to their System to serve user requests.

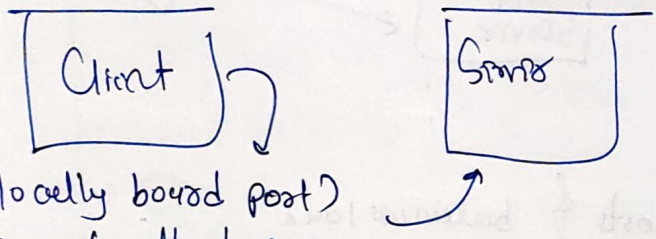
Intra Network uses Smart Clients / hybrid solution for load balancing traffic.

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)
eg. localhost:9000

↳ Managed by HAProxy
(With efficient management of Requests on the Port)

2) Running on intermediate Server: Proxies running b/w different Server side Components.

HAProxy
- Manages health Checks
- Removal & addition of Machines
- Balances requests a/c pools.

World of databases

SQL v/s 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 store
- Document DB
- Wide column DB
- Graph DB

NOSQL

key value store

Data ⇒ Array of key-value pairs
key ⇒ Attribute
Value ← dictied to

Redis
Voldemart
Dynamo

Document DB

Data ⇒ Documents
⇓ grouped into
Collections
Each doc can be different.

CouchDB
MongoDB

Wide-column DB

Instead of tables, Column families
↳ Containers for rows
No need of knowing all columns upfront.
Each Row ⇒ different 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 b/w Entities)

Neo 4J
Infinite graph

High level differences b/w 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, columnar)
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 row to contain data.
Querying	SQL	UnSQL (Unstructured Query Language) Queries focused on Collection of Documents. Diff. DB ⇒ Diff. UnQL
Scalability	Vertically Scalable (+ horsepower of hardware) Expensive Possible to Scale across multiple Servers ↳ Challenging & time-consuming	Horizontally Scalable. Easy addition of Servers. Hosted on cloud or cheap Commodity hardware. → 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

Reasons to Use SQL DB

1) You need to ensure ACID Compliance:
   ACID Compliance
   → Reduces anomolies
   → Protects integrity of the Database.
   for many E-commerce & financial Applications
   ↳ ACID compliant DB is the first Choice.
2) Your data is Structured & unchansing
   If your business is not Experiencing rapid growth or sudden changes
   ↳ no Requirement of more Servers
   → Data is Consistent

then there's no reason to use System Design to support variety of Data & high ~~Performance~~ traffic.

Reasons to use NoSQL DB

When all other components of System are fast
↳ Querying & Searching for data → bottle neck
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)

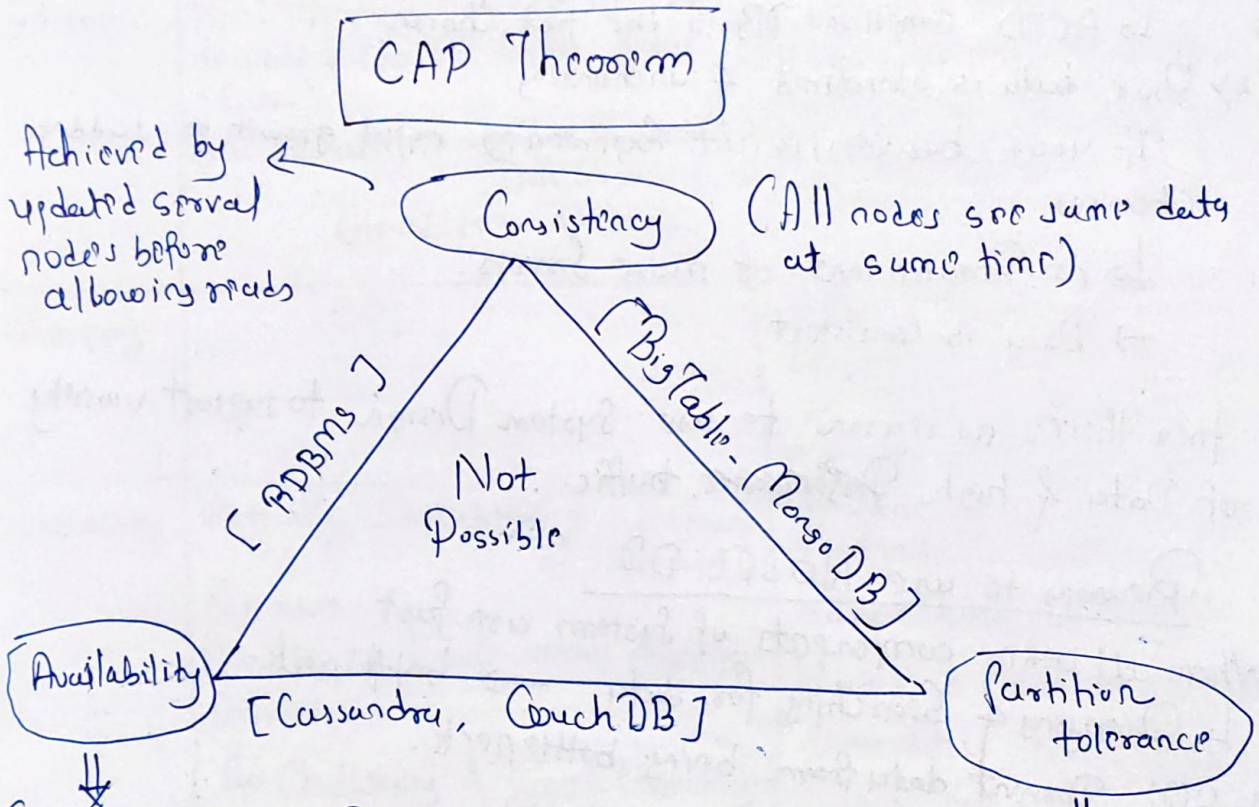
2) Using Cloud & storage to fullest.
   Excellent cost saving solution. (Easy spread of Data across multiple Servers to Scale up)
OR Commodity hardware on Site (affordable, Smaller)
⇒ No headache of additional software

4 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..



Every request gets Response (Success/failure)

Achieved by replicating data across 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 sufficiently replicated across Combination of nodes / networks 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 can not 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,
Updates 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 Requests from out-of-date Partition.

↳ Service is no longer 100% available.

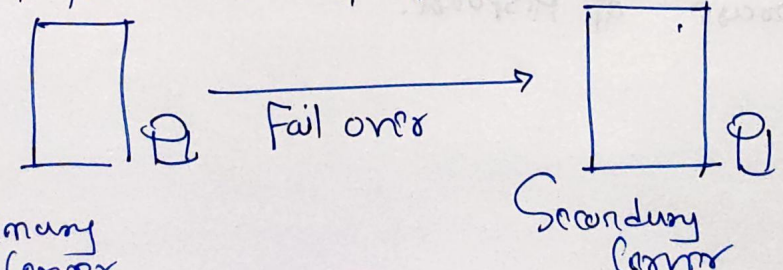
Redundancy & Replication

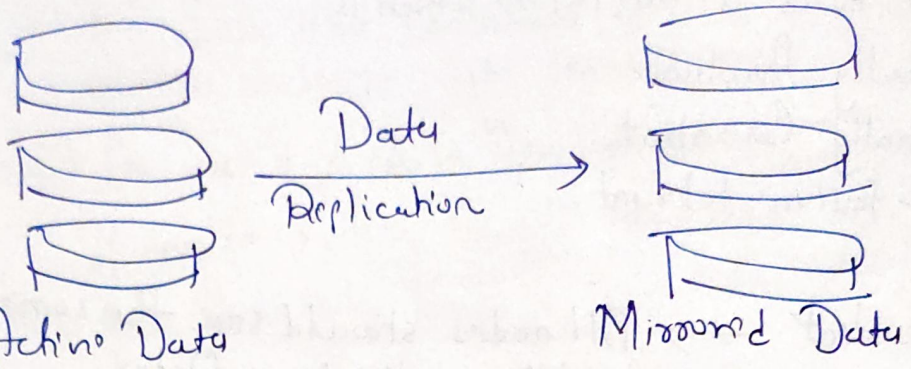
→ Duplication of Critical Data & Services.

↳ 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 Condition ← More Resilient to failures

No Single Point of failure

Caching

load balancing ⟹ Scales Horizontally

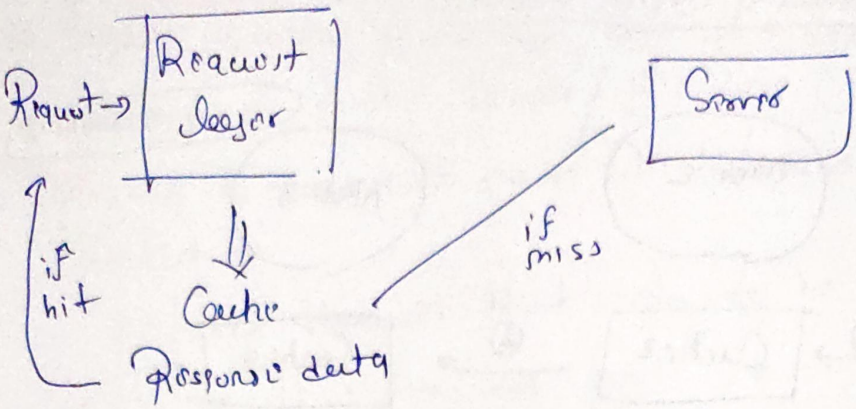
Caching: Locality of Reference Principle.

[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 load Balancing Distributes Request Randomly.

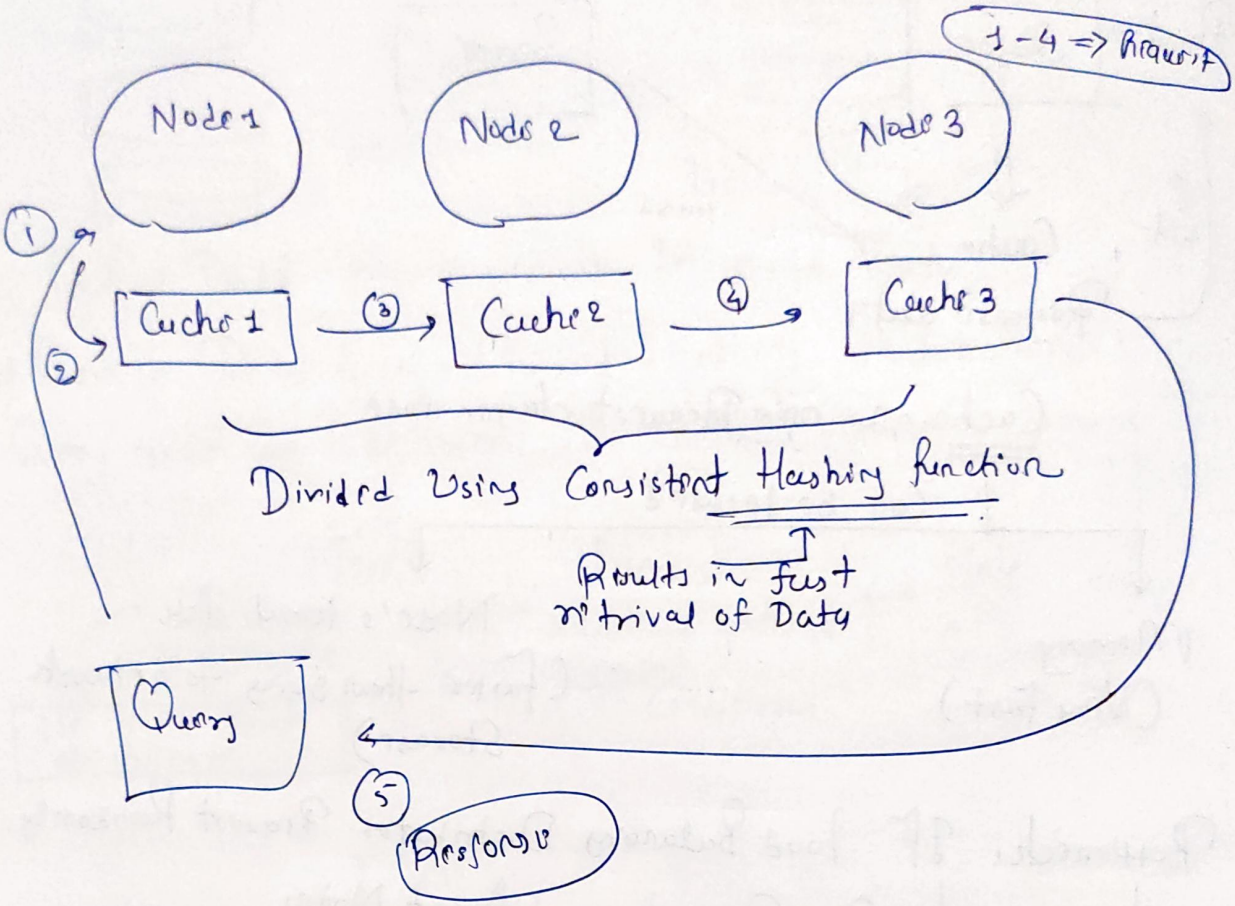
↳ Same Request ⇒ different Nodes

More cache Miss

Can be overcome by

1) Global Caches
2) Distributed Caches

Distributed Cache



Easy to increase cache by Space by adding more nodes

Disadvantage : Resolving a missing node

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

↳ Where making it more Complicated.

Even if node disappears ⟹

Request can Pull data from Origin

Global Cache

Single cache space for all the nodes.

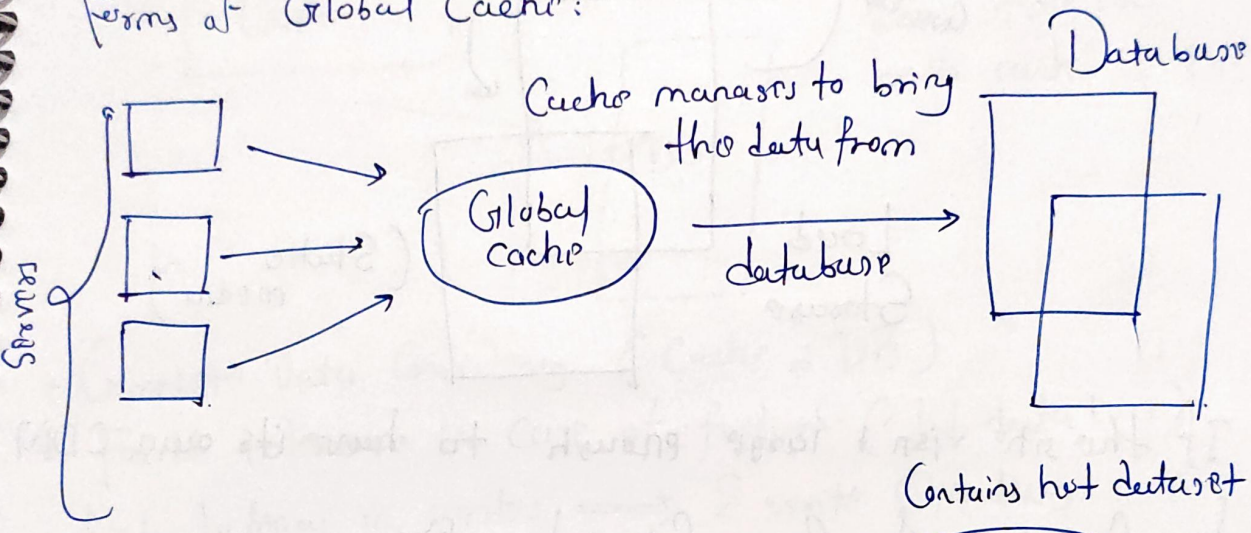
↳ Adding a cache server / file store (faster than original store)

Difficult to manage if no of Clients / request increases

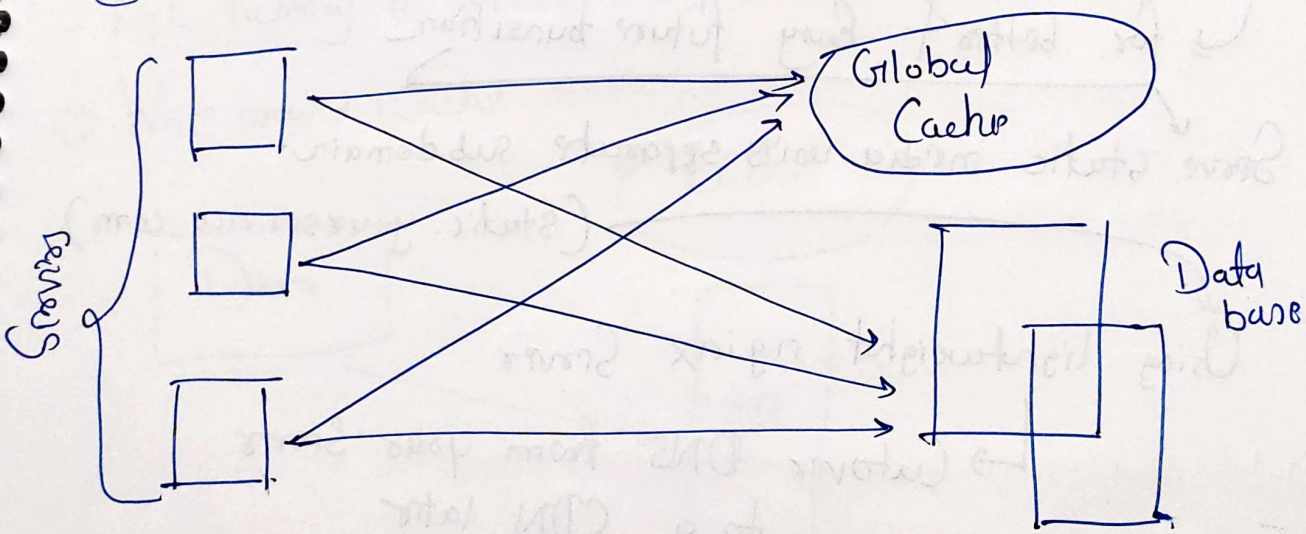
Effective if

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

Forms of Global Cache:



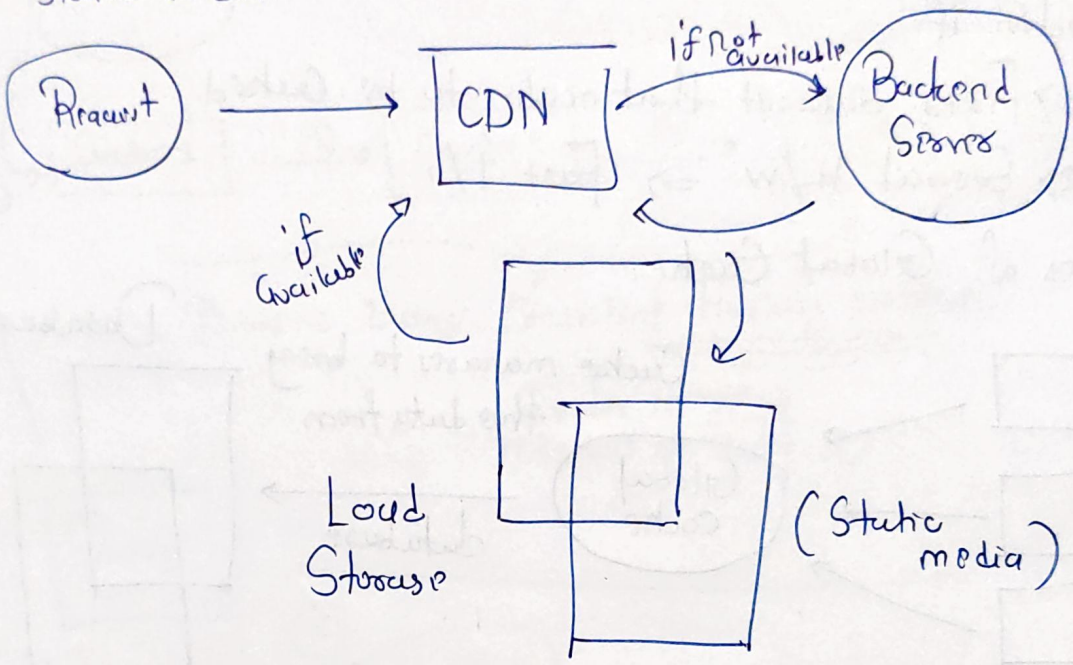
Contains hot dataset



Application logic understands the eviction strategy / hotspots better than cache.

CDN: Content Distribution Network

↳ Cache store for sites that serve 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

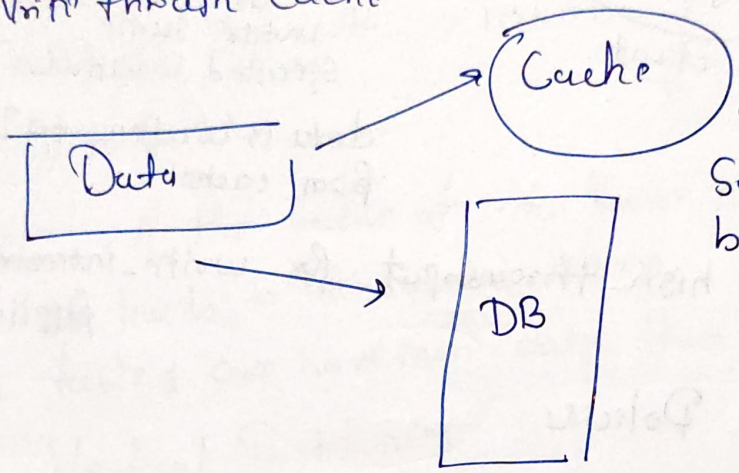
↳ Cutover DNS from your Server to a CDN later

Cache Invalidation

Cache data ⟹ needs to be coherent with the database if data in DB modified ⟹ Invalidate the cache data.

3 Schemes

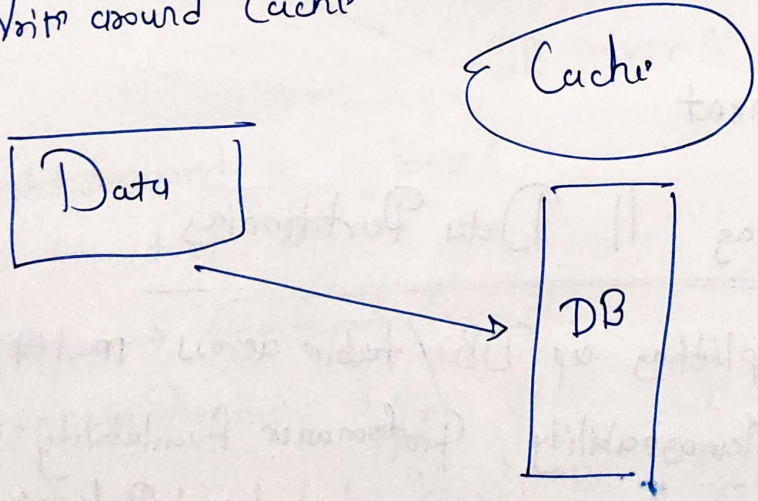
1> Write through Cache



Data is written same time in both cache & DB.

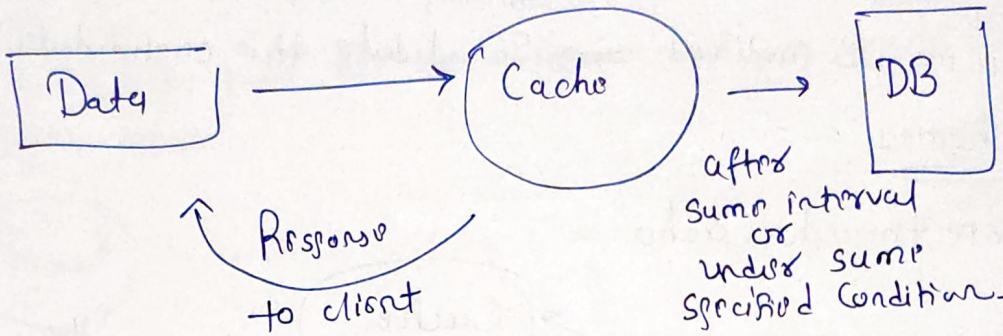
+ Complete Data Consistency (Cache = DB)
+ Fault tolerance in case of failure (↓↓ data loss)
- high latency in writes ⟹ 2 write Operations

2> Write around Cache



+ no cache 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 Application

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 machine ⟹ Manageability, Performance, Availability & load Balancing

After a certain scale point, it is cheaper and more feasible to scale horizontally by adding more machines instead of Vertical Scaling by adding bigger Servers.

Methods of Partitioning:

1) Horizontal Partitioning: Different rows into different tables
range based Sharding.

E.g. Storing locations by Zip

Table 1: Zips with < 100000
Table 2: Zips with > 100000

⟹ different ranges in different tables

and so on

Cons: If the value of the Range not choosen carefully
↳ leads to unbalanced Servers.
Eg 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

Straight forward to implement
low impact on Apps.
if app → Additional Growth
need to Partition feature specific DB across various servers.
i.e. 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~~

Directory Based Partitioning

↳ A loosely coupled approach to work around issues mentioned in above two Partitionings.

Create lookup Service ⟹ Current Partitioning Scheme & Abstract 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 attributes of the data → [Hash function] → Partition Number

Efficiency fixes the total number of Servers / Partitions

So if we add new server / Partition → Change in hash function

downtime because of Redistribution ← Change in hash function

→ 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: Enforce Constraints on the different Operations
⇓
Operations across multiple tables or multiple Rows in the same table ⟶ No longer running in single Server.

4> Joins & Denormalization:
Joins on table on single Server ⟹ Straight forward
not Feasible to Perform joins on sharded tables
↳ less efficient (data needs to be compiled from multiple servers)

Workaround ⟶ Denormalize the DB.
So that the queries that Previously required joins can be performed from a single table.
↳ Cons: Perils of Denormalization
↳ Data inconsistency

Referential Integrity: Foreign keys on Sharded DB
↳ Difficult

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

If an application demands Referential integrity on sharded DB
↳ Enforce it in application code (SQL jobs 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)

Indexes

→ Well known because of databases.
→ Improves speed of Retrival.
→ Increased Storage overhead
- Slower writes
↳ Write the data
↳ Update the index

→ Can be Created using one or more Columns.

+ Rapid Random lookups.
  + Efficient across of Ordered Records.

#Data Structure

Column → Pointer to Whole Rows

→ Create different views of the same data.

↳ Very good for filtering / Sorting of large data Sets.

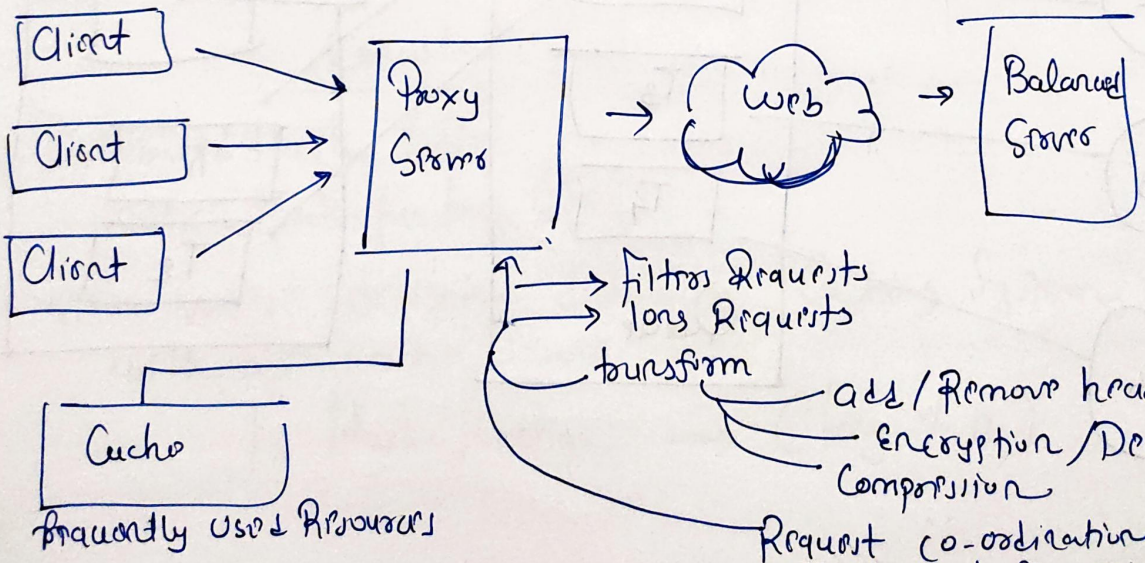
↳ no need of to create additional copies.

↳ Small Payload (1KB)

#Used for datasets (1TB in size)

Spread over Several Physical Devices → We need some way to find the Correct Physical location. i.e. Indexers

Proxies
- Useful under high load situations
- if we have limited Caching
  ↳ batches Serval Requests into one



We can also use spatial locality

↳ Collapsing Requests for data that is spatially close

→ Collapse same data queries request into one.

→ Collapsed forwarding → Minimize Read from Origin.

Queues

Effectively manages Requests in large-scale distributed Systems.

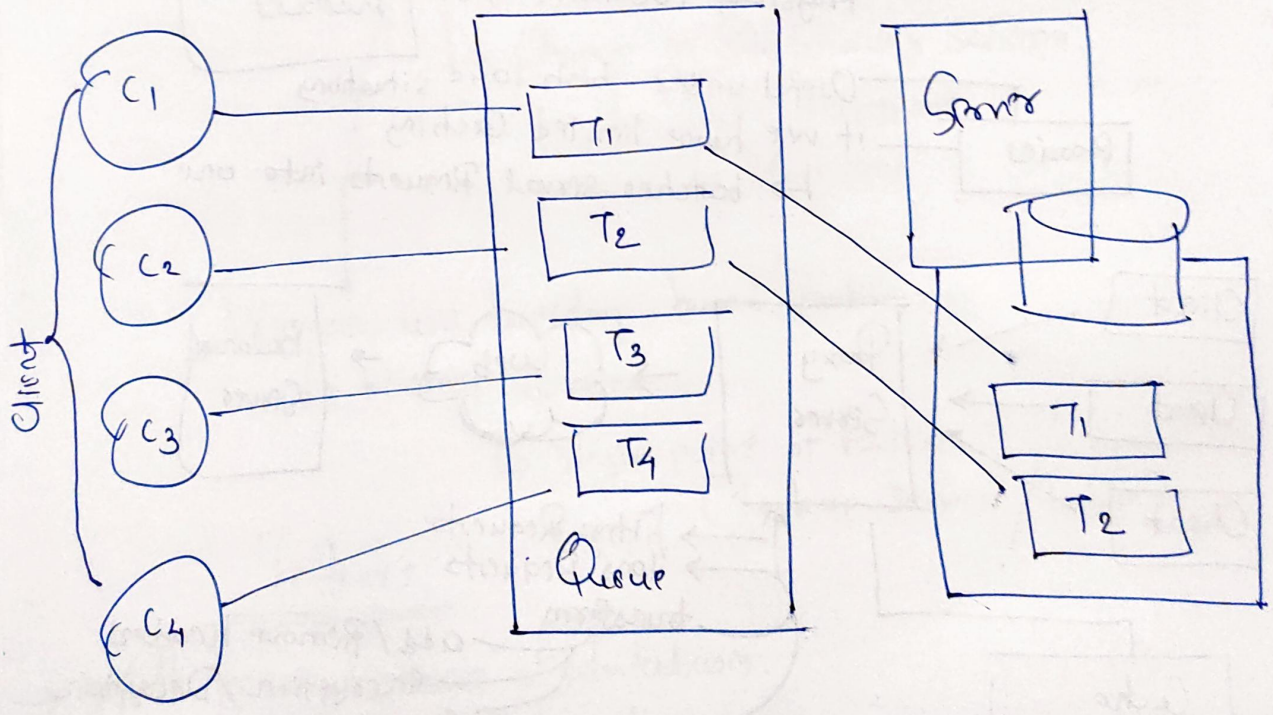
→ In Small Systems ⟶ Writes are fast.

→ 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 Request in Queue.

#Queues: 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 implementation

↳ 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 ⟹ (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?

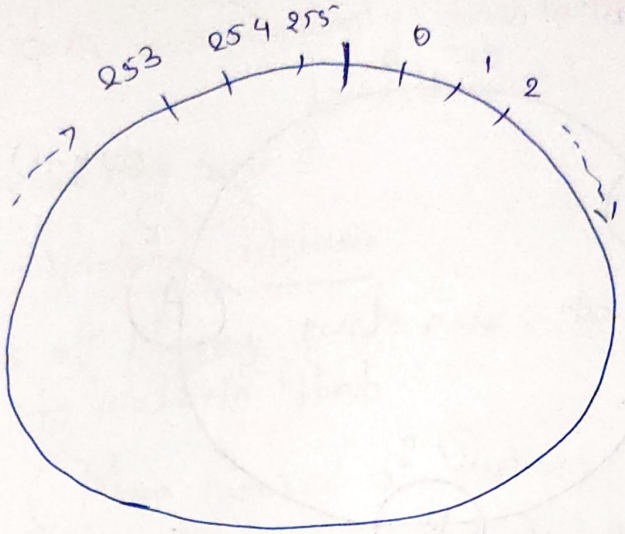
→ Very useful Strategy for distributed Caching & DHTs.
→ Minimizes Reorganization in Scaling up / Down.
→ Only k/n keys needs to be remapped.

k → total number of keys
n → Number of Servers

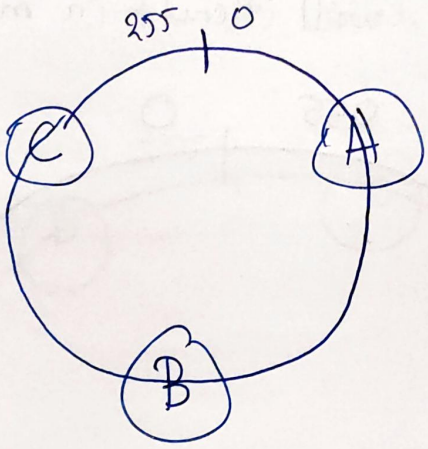
How it Works?

Typical Hash function suppose outputs in $[0, 256)$
In Consistent Hashing.

Imagine all of these integers are placed on a Ring.
Eg We have 3 Servers: A, B & C



1> Given a list of Servers, hash them to integers in the Range.

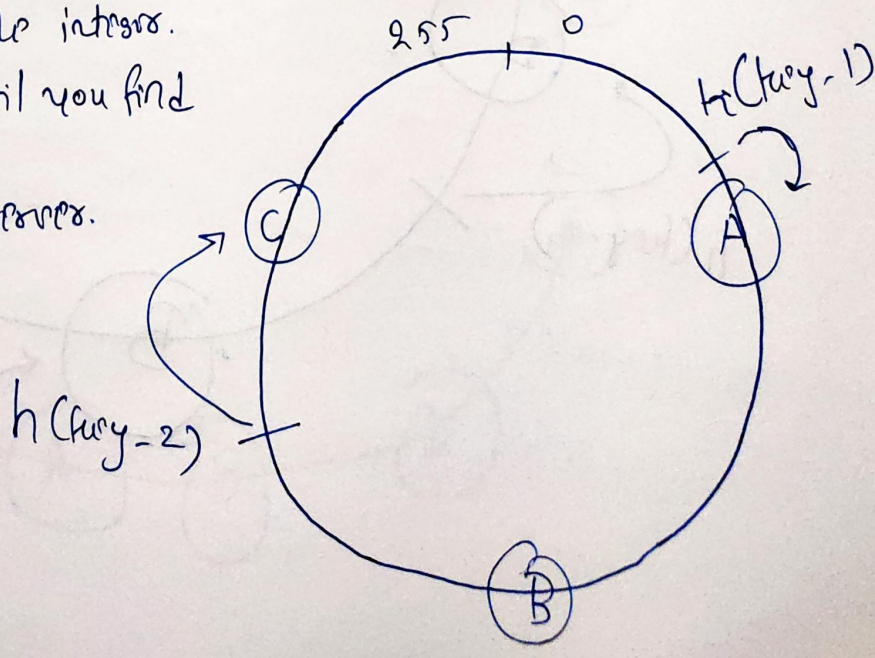


2> Map key to a server:

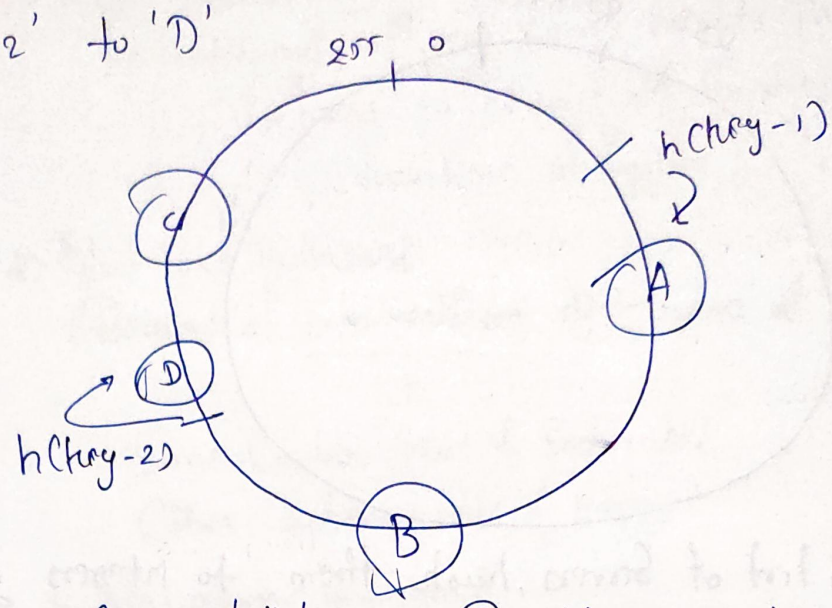
a> Hash it to single integer.

b> Move CLK wise until you find Server.

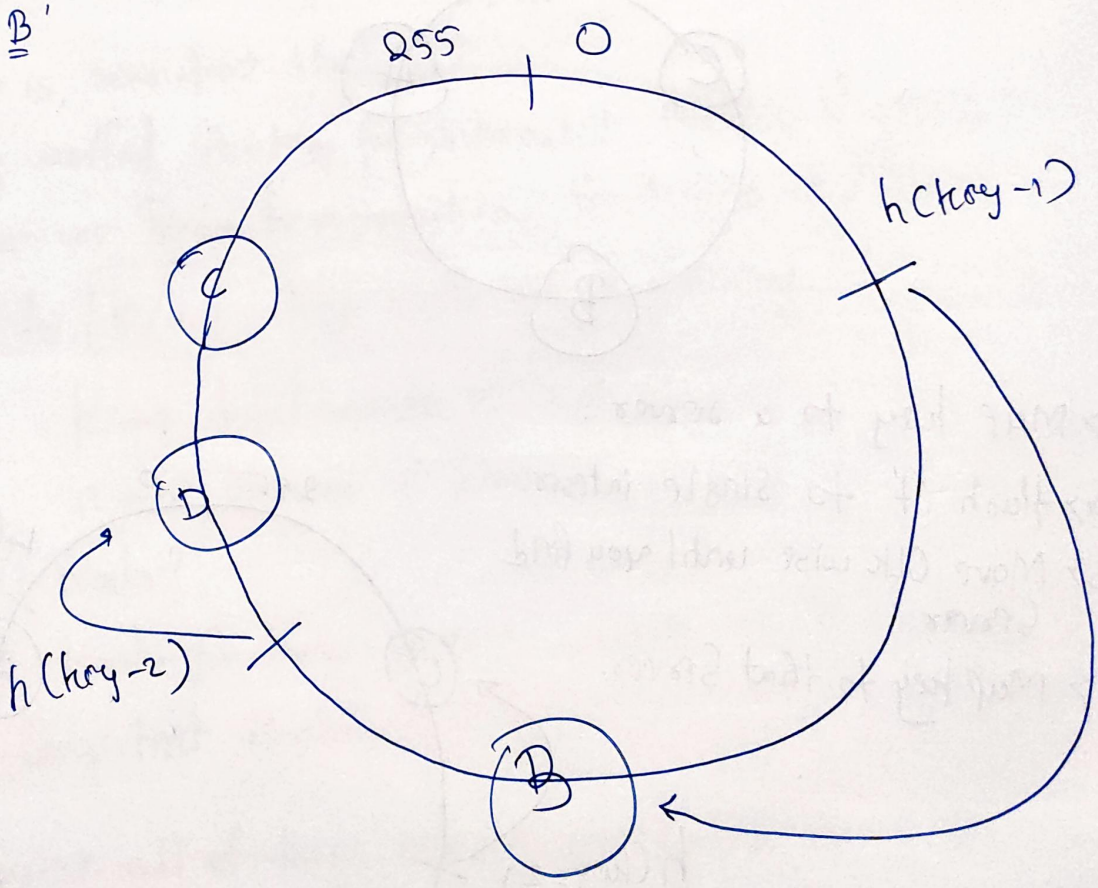
c> Map key to that Server.



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

data → Randomly distributed

↳ Unbalanced Caches.

How to handle this issue ?

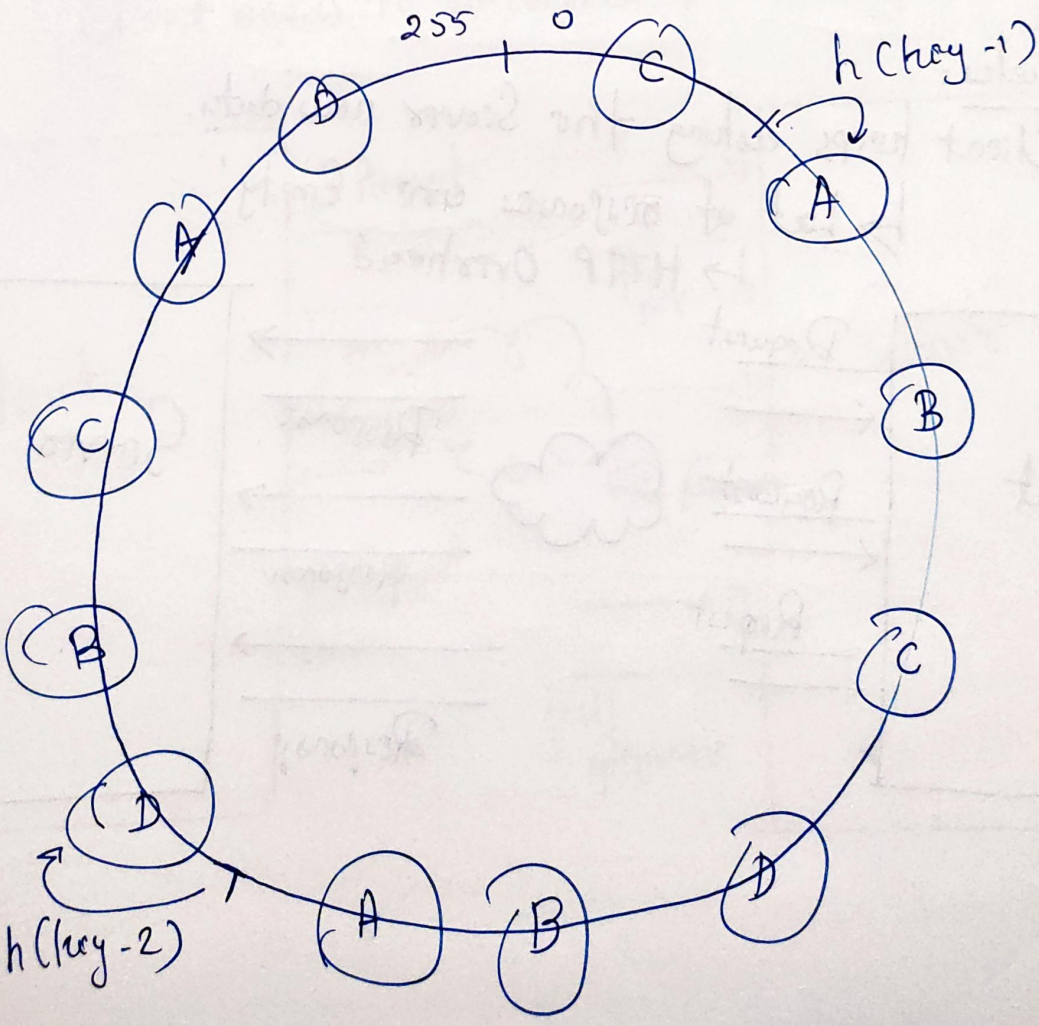
Virtual Replicas

→ Instead of Mapping each node to a single point we map it to multiple Points.

↳ (More number of Replicas

↳ More equal Distribution

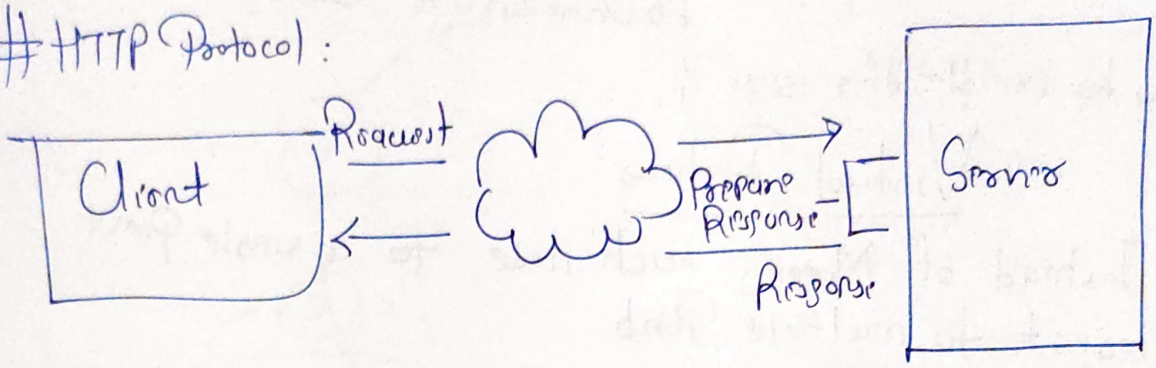
↳ Good load Balancing)



Long Polling v/s Web Sockets v/s Server Events

↳ Client Server Communication Protocols.

#HTTP Protocol:



#AJAX Polling:

Clients Repeatedly polls servers for data.

Similar to HTTP Protocol.

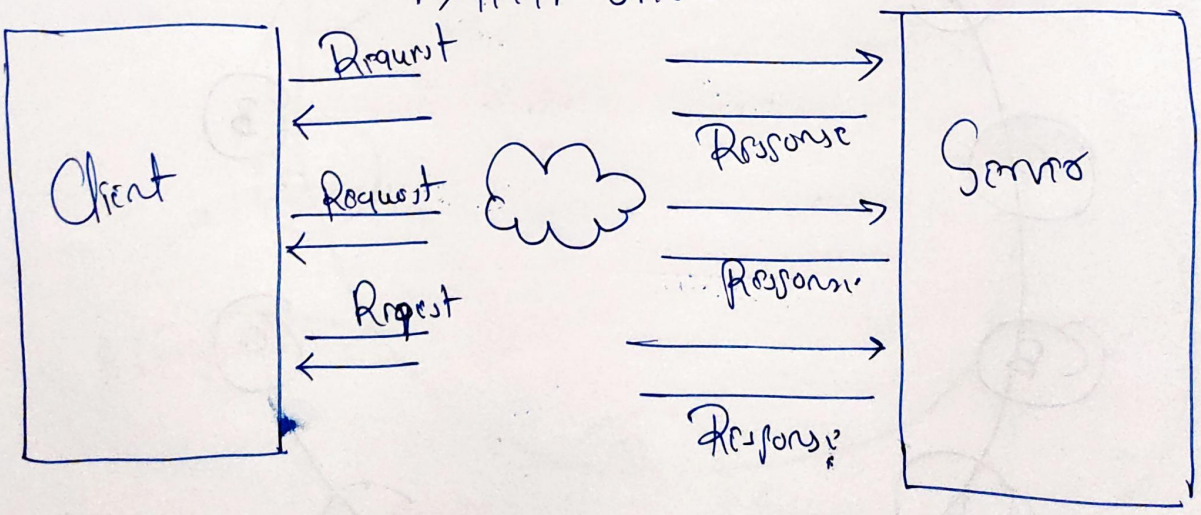
↳ Requests sent to Server at Regular intervals. (0.5 sec)

Drawbacks:

Client keeps asking the Server new data.

↳ Lot of responses are 'Empty'

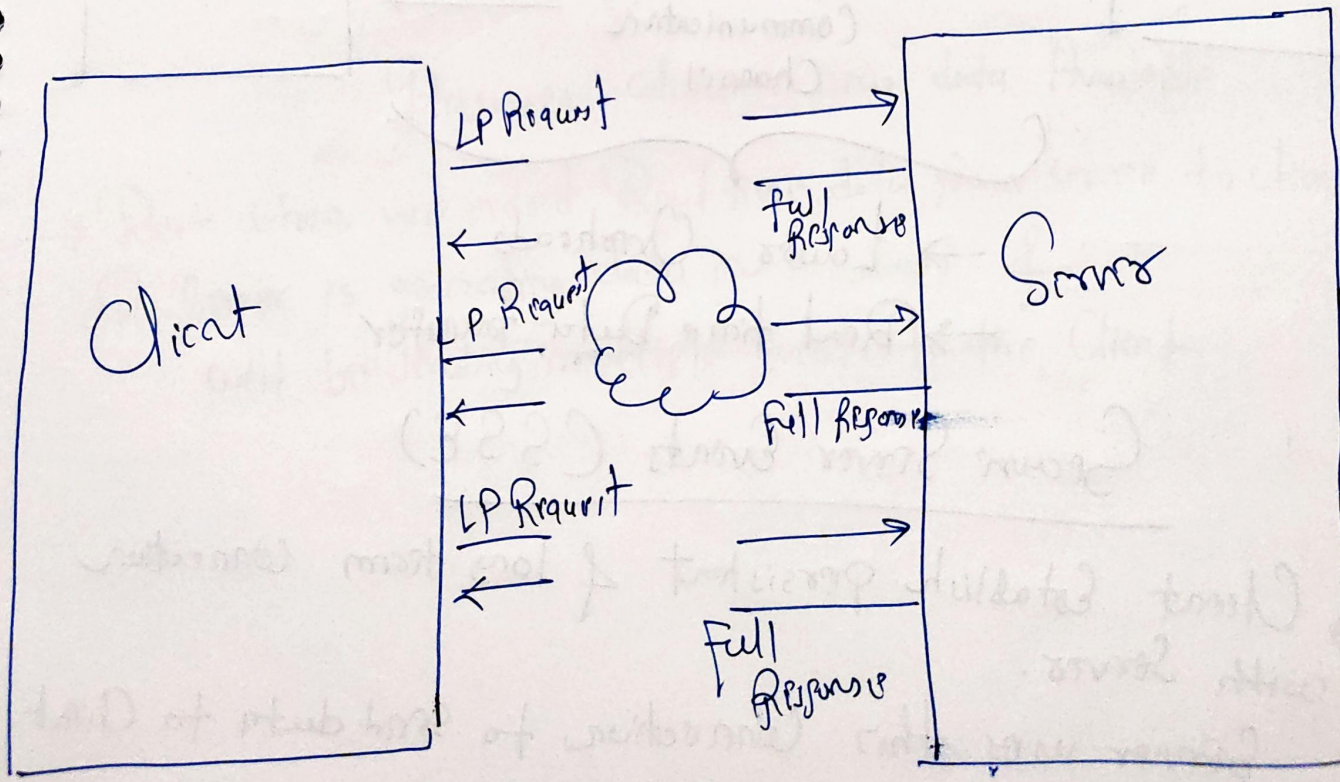
↳ HTTP Overhead



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.

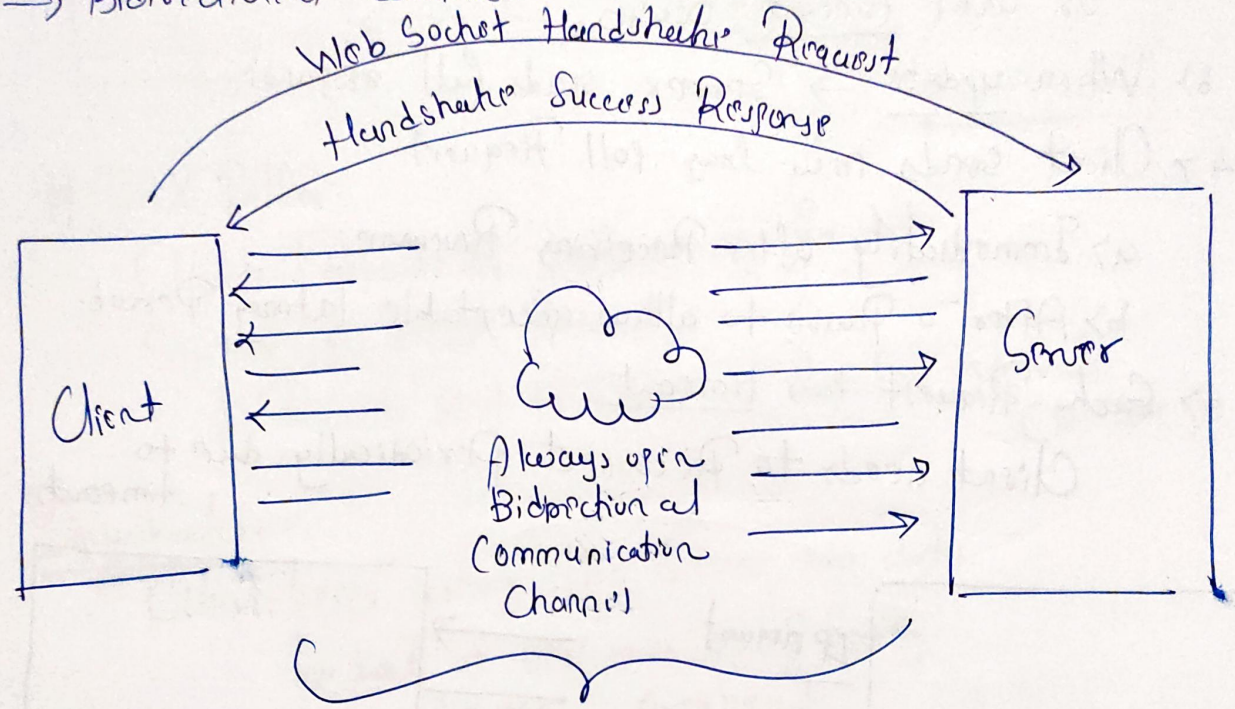


WEB Sockets

→ Full duplex communication channel over single TCP connection.

→ Provides "Persistent Communication"
(Client & Server can send data at Anytime)

→ Bidirectional Communication in always open channel.



→ Lower Overheads

→ Real time Data transfer

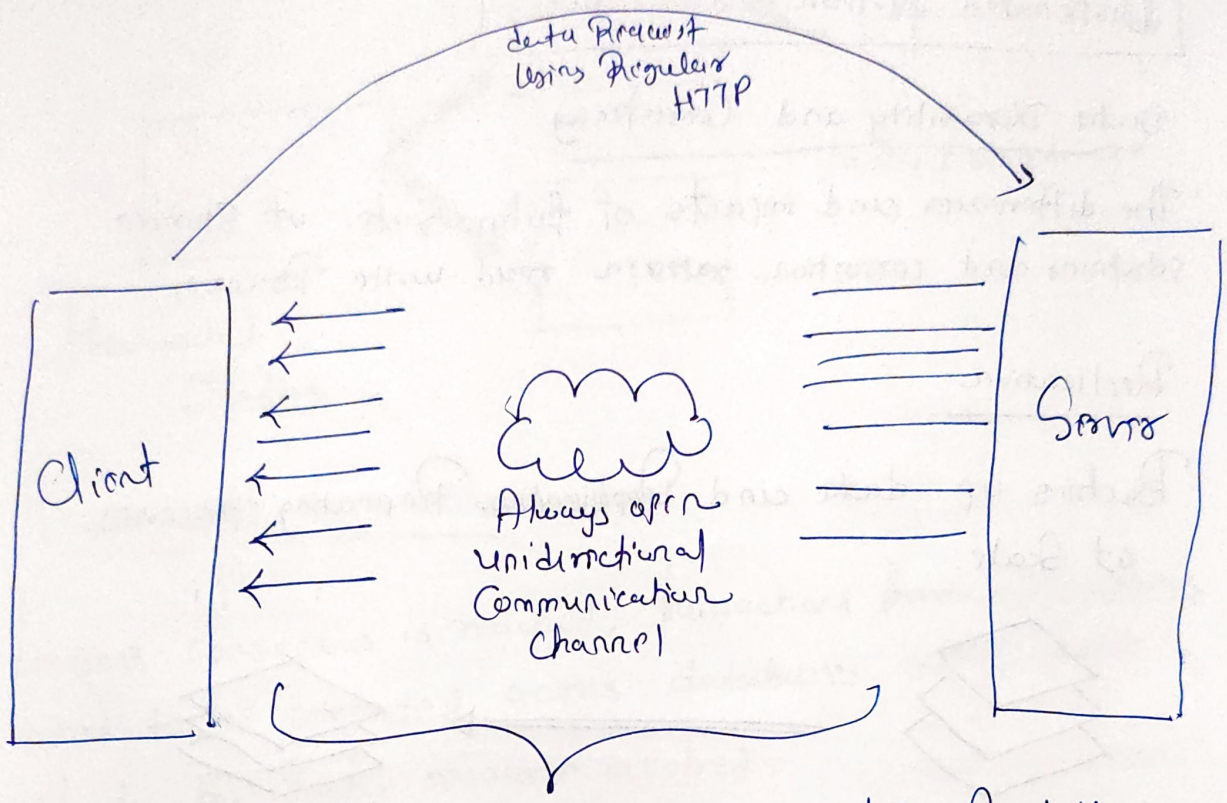
Secure Server Events (SSE)

→ Client Establish 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.