Agenda

What & why of Rate Limiting

Client side

Server side

Design a distributed Rate Limiter

Various Rate Limiting algorithms

Proxy

Rate Limiter

limit the rate of requests / usage of a service / # of calls per hour

Safeguarding / DoS attacks / QoS

We want to limit access to some resource

API / software service

Why?

① DDoS prevention (Distributed Denial of Service)

900 Million requests / sec

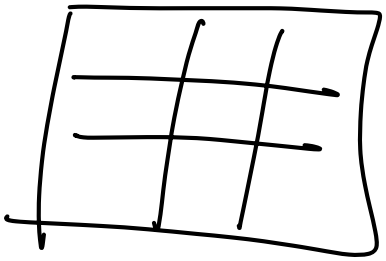
② fair usage (FUP limit)

↳ fair usage policy.

we have some public resource — prevent some bad users from 'hogging' up all the resources.

③ Prevent Scraping. — Using a program to download & process data from a website.

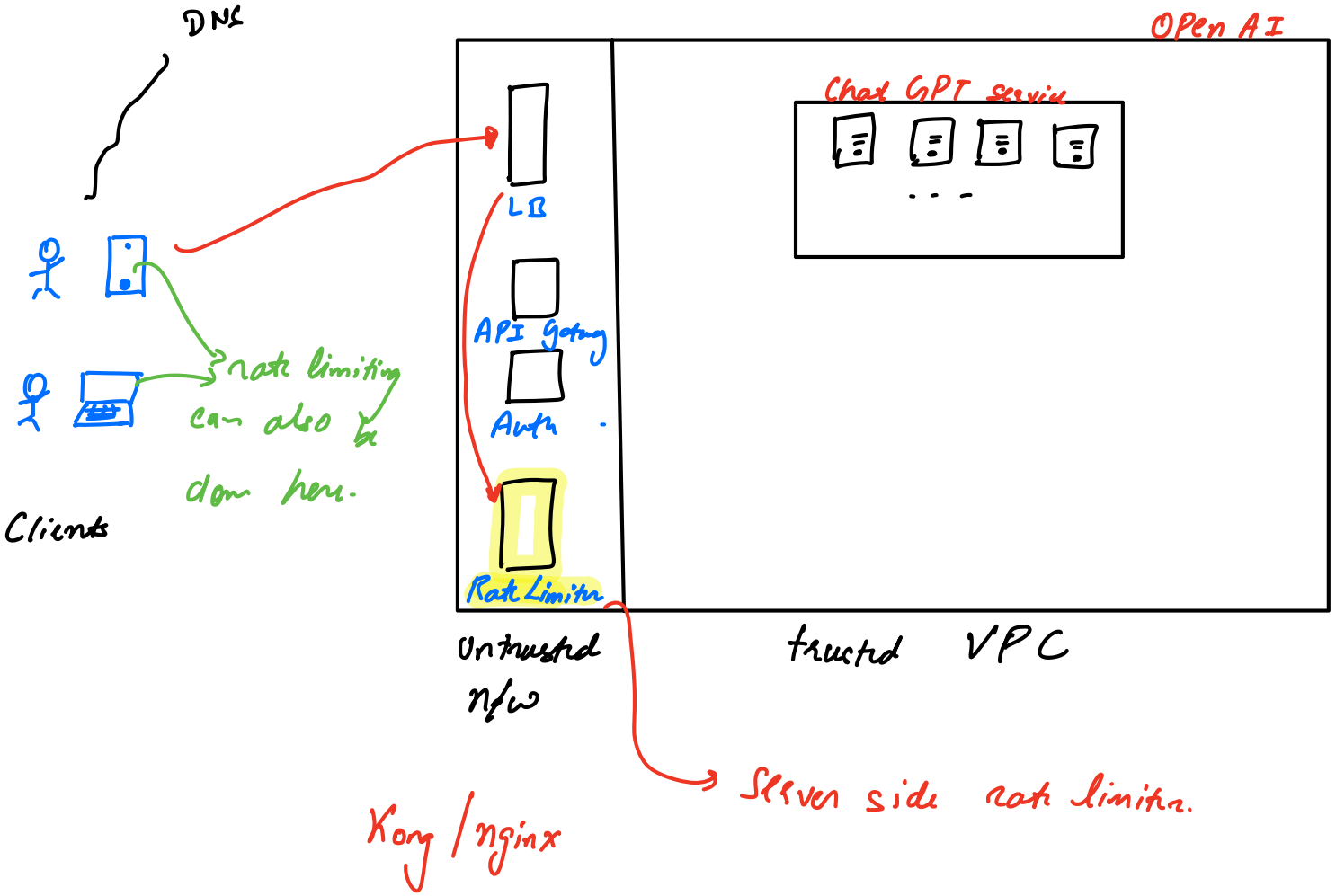
Google ☐ I'm not a robot



Bing

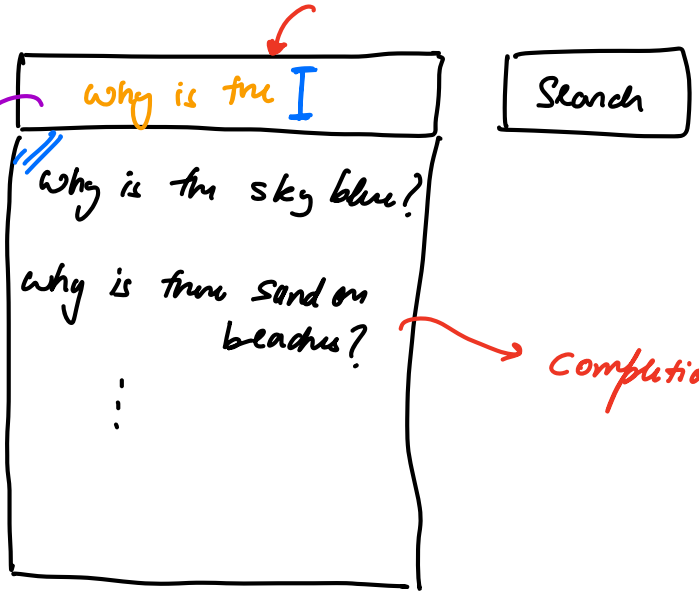
④ Regulatory Compliance — limit the no of stock trades/second on some instrument.

System Architecture

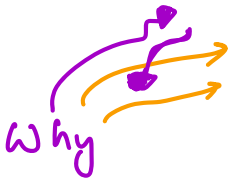


Client Side Rate Limiting

① Debouncing → every time you type a letter the auto complete results update.



type 10 letters in 1 second.



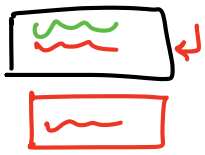

```
#("cc.input").keydown((e) => {
    if (e.keycode != = ENTER){
        // make an API req &
        Populate autocomplete suggestions
    }
})
```

Only send the request once user stops typing!

① Sending request on user input

② auto saving

② Throttling → manually add a delay b/w any two requests



↲ → error: posting too quickly.

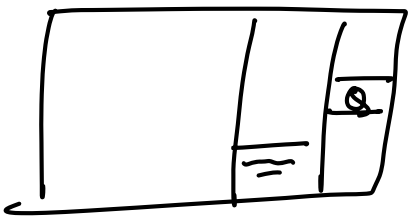
Exponential Backoff / Queuing / Caching / Batching

Purpose of all client side rate limiting → to reduce the no of API calls that are generated.

Issues with purely client side rate limits

① It can always be bypassed.

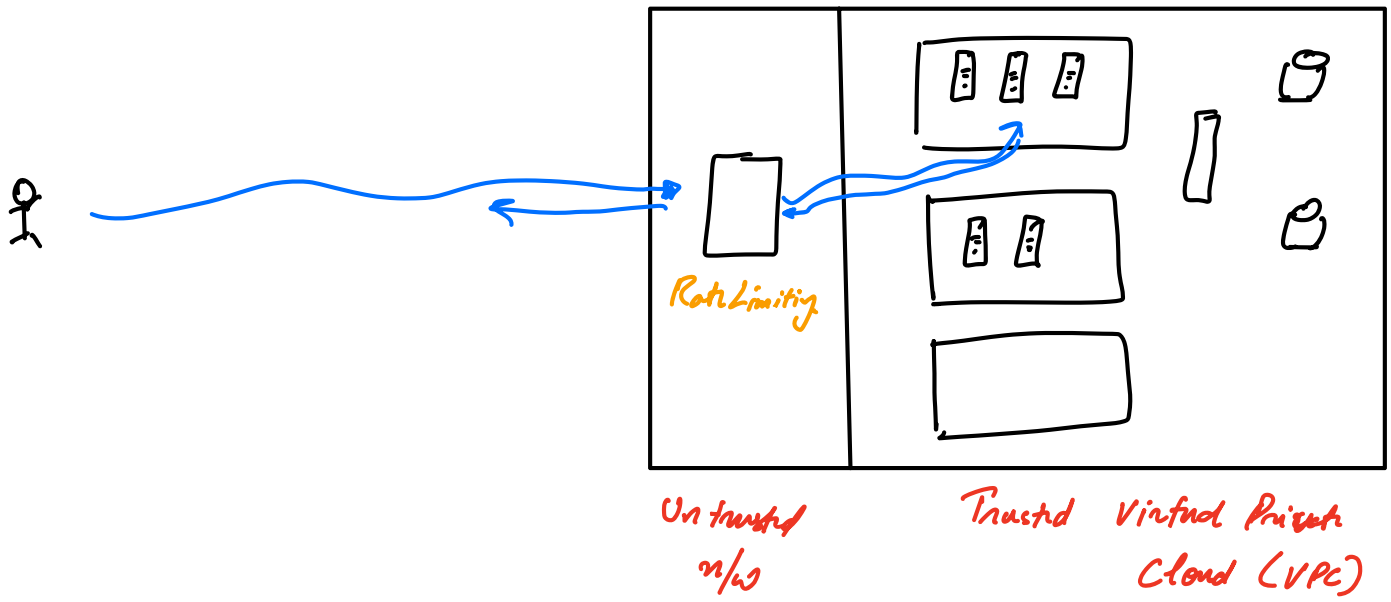
② If your clients have direct API access → no frontend.



Zerodha → stock broker

Kite API → to build your own stock trading platform.
→ to build your custom trading algorithm.

Always need Server Side Rate Limiting as well!



If rate limiter decides to not accept → Queue it for later
→ drop/deny request.

Work flow

1. Client request arrives at Load Balancer.
2. Load Balancer checks with Rate Limiter about whether it should accept/reject it.
   IP/user-id/geographic location/...
3. Rate Limiter will retrieve information from some "data store" to store how many req have we seen in past
4. Decision Making → Accept
   → Reject → Queued
   → Denied → proper HTTP error code
   429 : Too many Requests

⑤ Request Processing → let the request go through.

⑥ Logging & Monitoring.

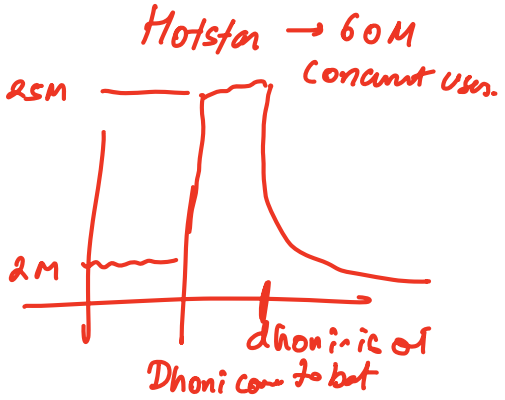
↳ irrespective of whether the request is accepted or denied we should log it!!

↳ Metrics Collection : % of acc/rej / Top 10 Users with most req ..

↳ Monitoring Tools

↳ Alerting

↳ Capacity Planning.



Prometheus / Grafana / Load Balancing / API Gateway / DNS. ..

High Level Design / System Design → how to scale a system.

Google → Staff Engineer (Bengaluru 1.5 - 2 Cr / annum)

V. simple: given a list of strings → sort them alphabetically.

V. difficult twist: 50 Petabytes of data

KB MB GB
TB PB

50,000,000 GB of data

@ Scale → 1.5 months on High Level Design

↳ 2.5 months on Low Level Design

9:02 → 9:15

SDE 1 (fresher) → SDE-2 → SDE-3

SDE-3 → Individual Contributor

SDE-3 → Managerial role

SDE-3 → Staff Curriculum

Pretty much every one at SDE-2+ is involved in architectural decision.

CTO ← Architectural roles ← Staff / Principal Engg.

Architectural roles → Software Architect

Mentors → CTO of Groww

→ Bhavin Turakia

Various Rate Limiting Algorithms

① Fixed Window Counter

limit to 1000 requests / hour (window)

Max count

- for every window of T time we will store an integer count

Product team
Infra team
Stakeholders

- when a new req comes
  - find the window that it falls under.
  - retrieve the count for window
  - if count is within the limit (1000)
    - accept req
    - increment count
  - else
    - reject request.

10am – 11am | 11am – 12noon
0
12noon – 1Pm | 1Pm – 2Pm ...

req 1 (10:20am)
↓
10am – 11am

Data Structure

① On what "entity" am I putting the rate limit on?

- user_id → simple limiting
- access_token / API Key → exposed APIs
- IP address → anonymous users
- geographic → DDos prevention.

Key

Attempt 1 : Store a count for each key
↳ # of req for this key for this window.

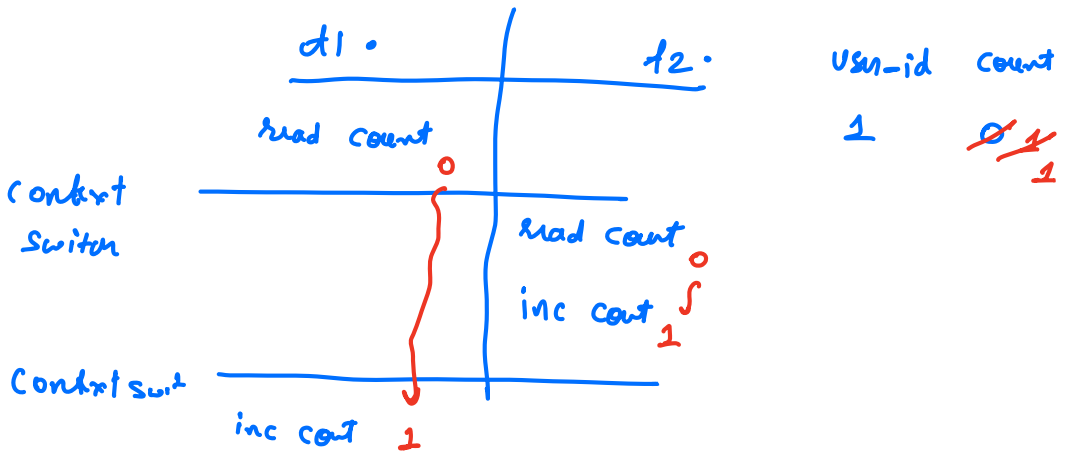
```
Map < Key, Integer > counts;
Map < Key, Integer > limits;
t1 ↳ void process (Request r) {
t2 ↳
    Key k = r.key;
    int count = counts.get(k);
    if (count < limit) {
        // accept
        counts.set(k, count + 1);
    } else {
        // reject
    }
}
```

→ when will we store these? (in RAM? Rate Limiter server) (what if we have a dist system?)

? → global variable (same limit for everyone) / loaded from some config (custom limits for each key)

RACE Condition!

'fetch then set' pattern



Topics for later

① Data Store ② Config Store ③ Mutual Exclusion

How to reset the count at start of next window?

① every 1 hour, go through all counts & set them to 0.

Eager

```
= for (Key k : counts)
=         counts.set(k, 0);
```

2 Billion users.
0.1 ms / Key
$2 \times 10^9 \times 0.1 \times 10^{-3}$ seconds
$= 2 \times 10^5$ seconds
≈ 60 hours

② Map < Key, Pair < Timestamp, count >>

Key: user id / IP addr / access token.

Timestamp: current start of window

count: count of current window

Lazy ✓

when req comes & check curr time.
if curr time is in the same window
then ✓ use it
else ✓ update the window & reset count.

Properties / Behavior of Fixed Window Counters

① v. simple to implement

② allows short bursts

③ succesfully prevents long bursts.

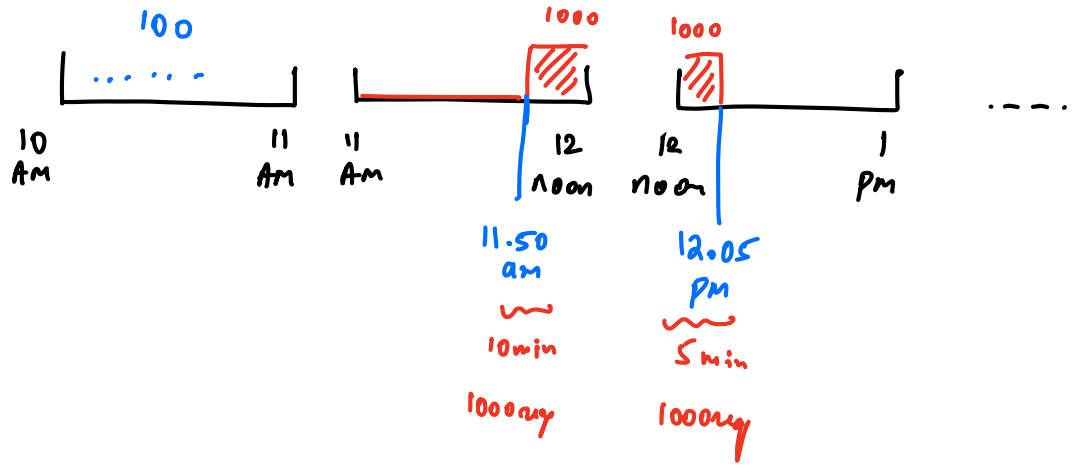
④ allows high throughput across windows

Burstiness
if suddenly a lot of req can is a short period (within rate limit)

long burst { continued high req above the rate limit.

short burst { 1000 req / hour.
in a 5 sec window user makes 999 req.

Boundary Case → 1000 req / hour



→ | 15 min | ←

2000 req

violate our rate limit

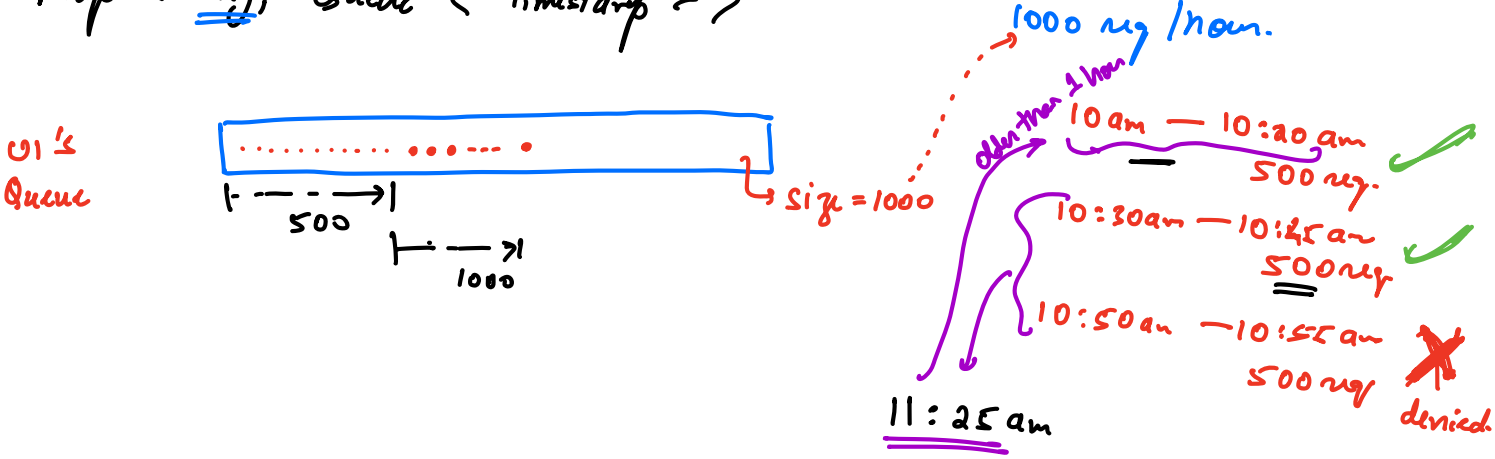
our algo lets this pass!

Sliding Window Log

- Store / Log every req that comes
- Aim : 'Strictly' enforce the rate limit

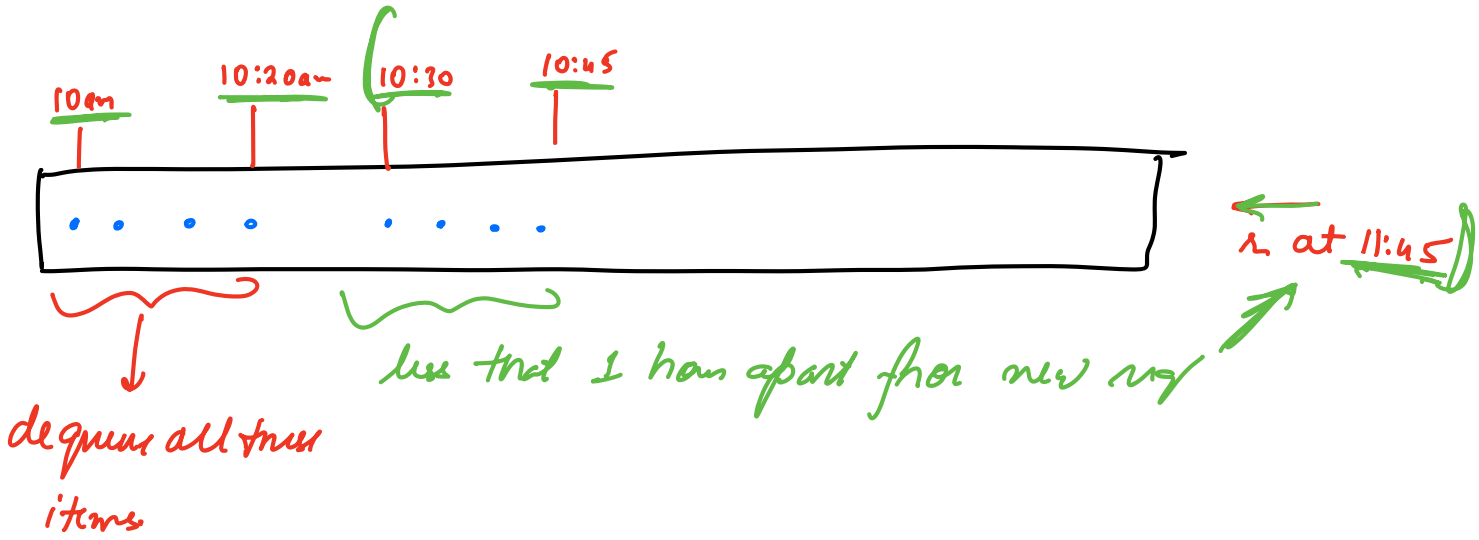
not a req Queue → it is a log queue

Map < Key, Queue < Timestamp > >



① when a new req comes.

- remove all items from queue that are outside curr window.
- check how many items are in log Queue
  ∵ all these are in the same hour.
  - if # items < rate limit
    - accept & add to Queue
  - else
    - reject

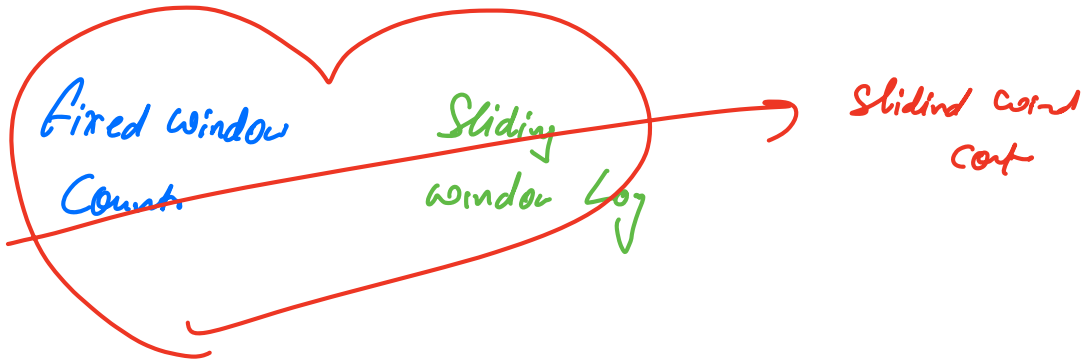


Pros ① absolute correctness

Cons ① takes more memory ∵ Queue

② for some req the latency is v. high

Sliding Window Counter → Approximate algo



Token Bucket

Leaky Bucket

---

① How is data stored → Replication
→ Resiliency

② How is data being read → CAP theorem
→ Read Replicas
→ Caching

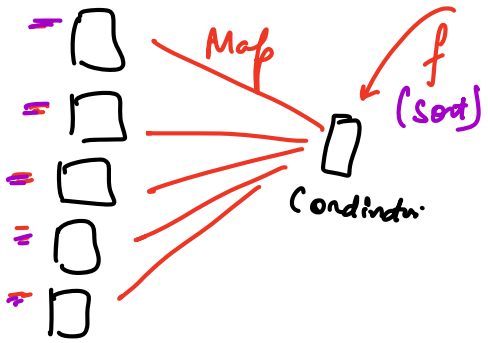
③ How do we sort it?

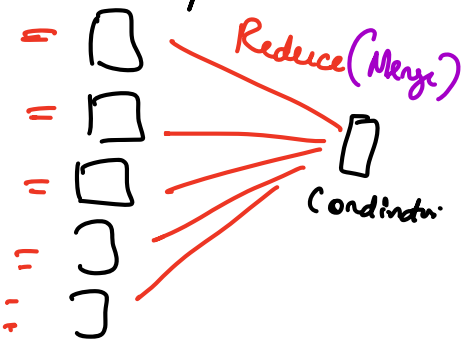
↳ usecase: archiving → ---

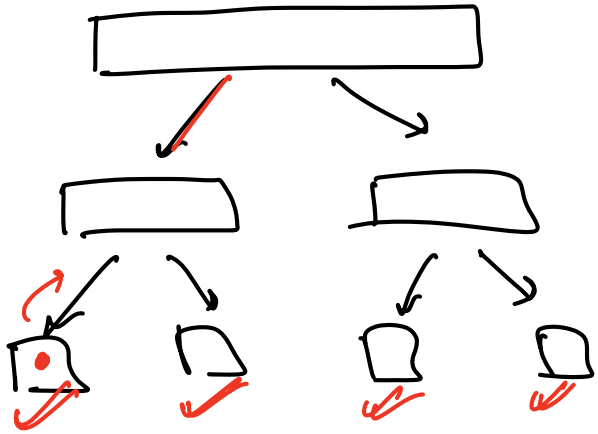
↳ usecase: analytics → Map-Reduce

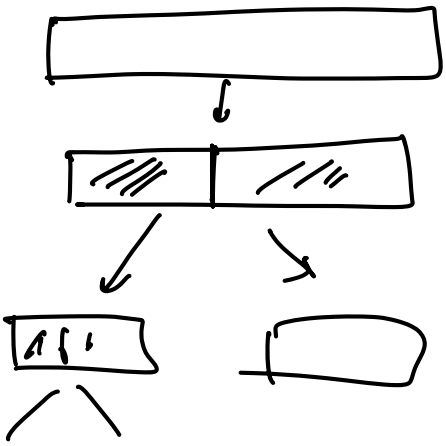
Apache Hadoop

Apache Spark

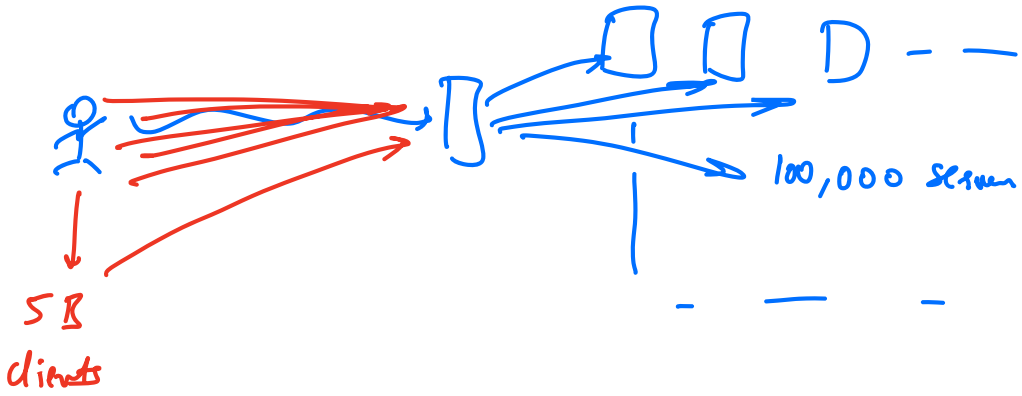


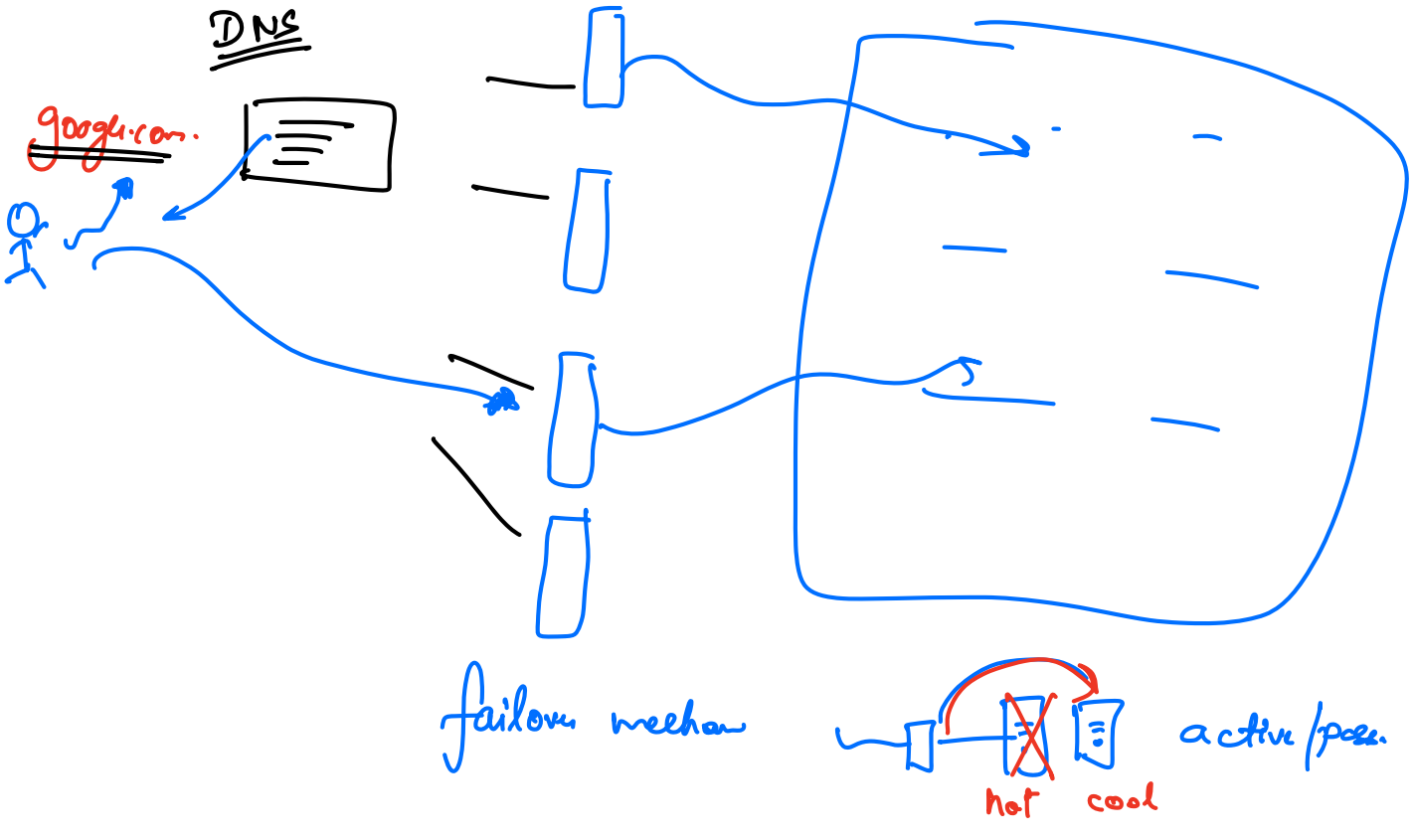






Any Cast


100,000 servers
5 B
clients


DNS
google.com
failover mechan
hot
cool
active/pass