Agenda

- → Unique Id Generator
- → Rate Limiter
- → Garbage Collection

Unique Id generator — trillions of ids (no collision)

UUID v4 → 122 bit (0/1) random number.

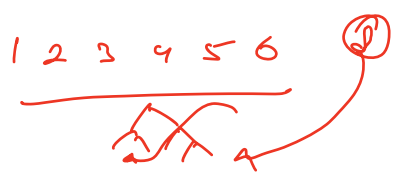
1 billion UUidv4() every second
100 years → 50% chance that there will be atleast 1 collision.

Pros

- → generated on any system (fully distributed)
- → efficient to generate

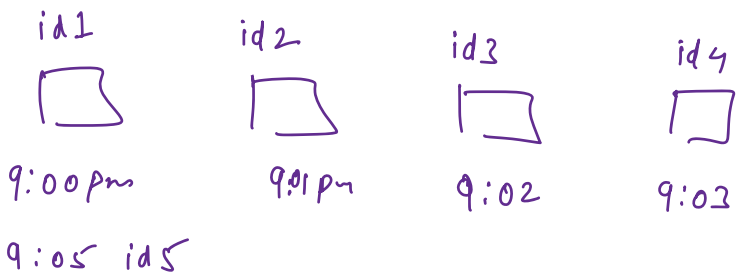
Cons

- → Super long
- → Random → not sorted
  Indexing is slower

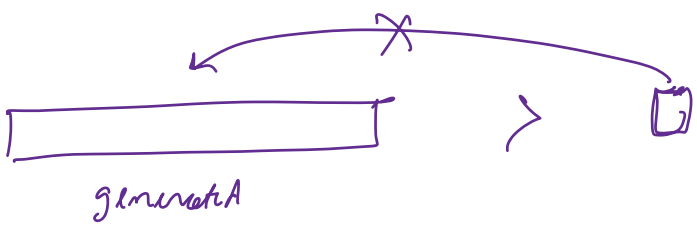


Unique & Incremental ids

↳ monotonically increasing.



id1 < id2 < id3 < id4 < id5

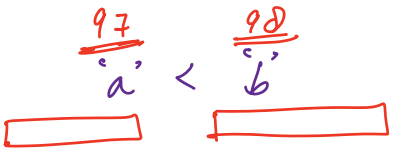


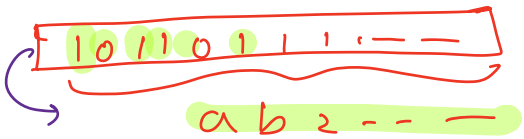
Why monotonically increasing?

→ Indexing is faster

→ useful for figuring out which event happened before.

distributed transactions





Counter Based

```
int id = 0
int unique_increasing_id() {
    return id++;
}
```

— works if there is 1 system

— many systems

if every system has its own counter

→ not increasing.

→ not unique

a central system / service that is in charge of ALL id generations.

Central API

Central Database

Pros

— meets requirements

— very simple to implement

Cons.

— single point of failure

— latency is high

↳ n/w call for each id

↳ single server will be bottleneck.

a central service which is replicated

both read & write heavy.

— predictable → security risk.

---

aadhar.india.gov/view/ 0, 1, 2, ⋮, 3728

aadhar.india.gov/vid/13bc5a---

2^{122} possible uuids

$\approx 10^{36}$

$\frac{10^{26}}{1.5 \times 10^{9}} = 10^{27}$ ids

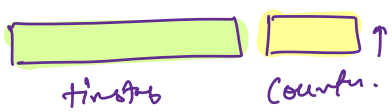
Timestamp → every cpu has an internal Quartz clock

↳ System clock → time in milliseconds → integer

(epoch time → # of millisec that have elapsed since 1 Jan 1970 —)

① Single Machine → id() called twice within same millisecond

have an additional thread-safe counter.



9:00pm

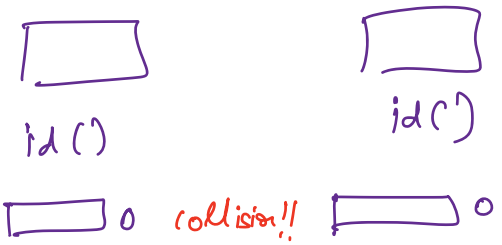
id1 = id() ← few nanosec

id2 = id() ←

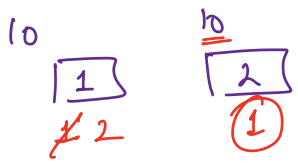
millisec changes

reset counter = 0

(2) Multiple machines



guarantees 0 collisions



majority of the time Order is maintained ∵ timestamp.

Sometimes order can be violated → violations.

1011 1021 1012

Achieving truly monotonically increasing unique ids in a distributed setting without a central authority

→ IMPOSSIBLE

Clock synchronization

NTP → it can rewind time!!

start-ts = sys()

end-ts →

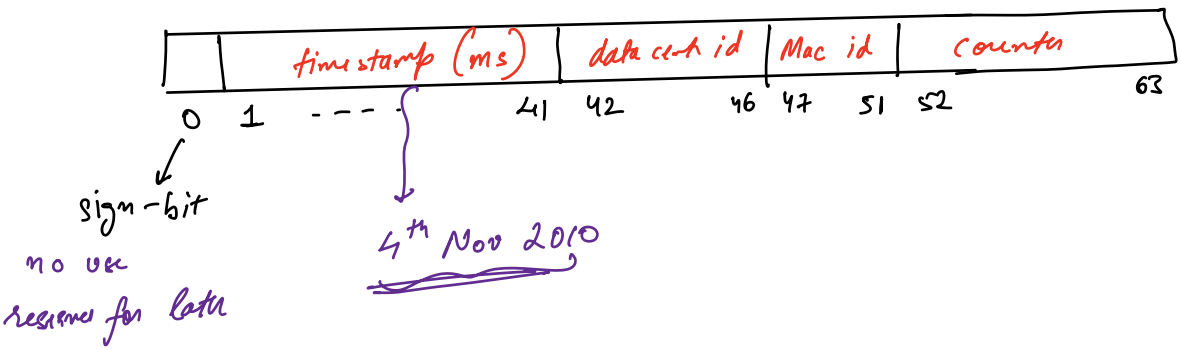
elapsed-ts = end-ts()

'ʃ
-ve

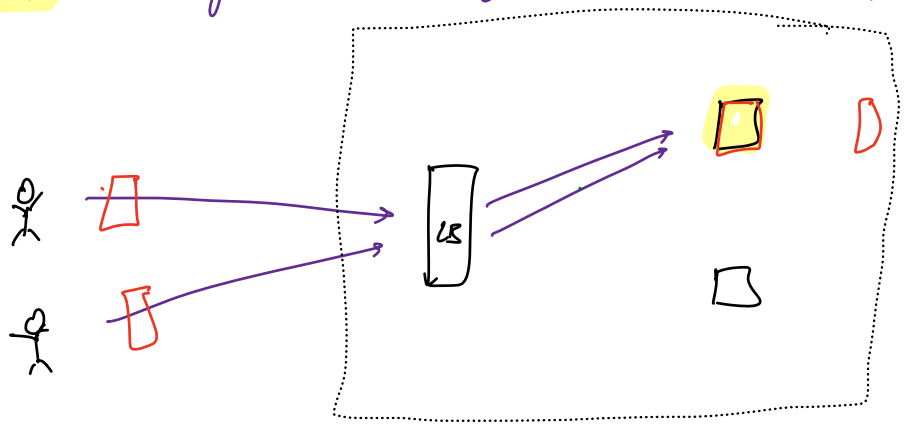
Snowflake IDs — Twitter

Instagram / Discord

64 bit integer



Rate Limiter → any user can only make k requests / seconds



Reasons

- → prevent DDoS attacks
- → avoid abuse
- → ensure fair usage

Each user has a unique ip address or some other id.

Design rate limiter for user.

User hashmap map < user-id, ——— >

Store ~~count~~

map < user-id, {int, first-call} → count, → timestamp

void handle-request (req) {

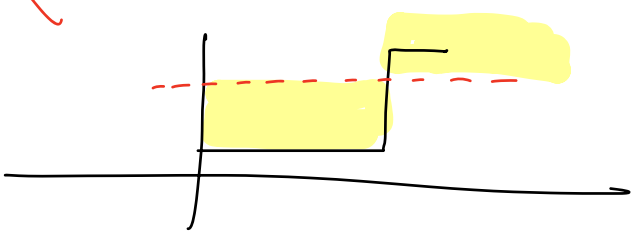
now = curr-ts()

diff = now - first-call

count/diff → rate of this user's req

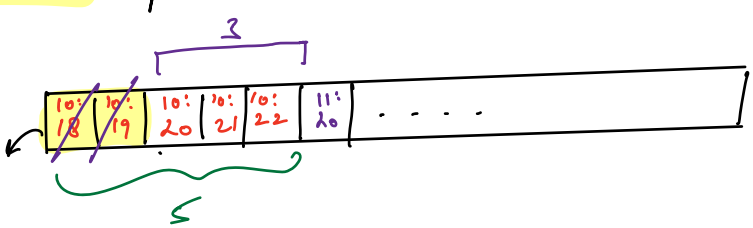
reject()

process(req)



Constraint: 5 req/hour

Request Queue per user



timestamp of req

10:19

10:20

10:23 reject

(10:23 − 1 hour) → any req older than this can be removed from queue.

current req timestamp

= 9:23

10:30 - 1 hour = 9:30

11:20 - 1 hour = 10:20

Challenges

→ High memory → maintain a queue for each user.

→ High latency for certain request

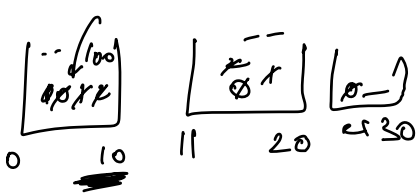
50K req/hour

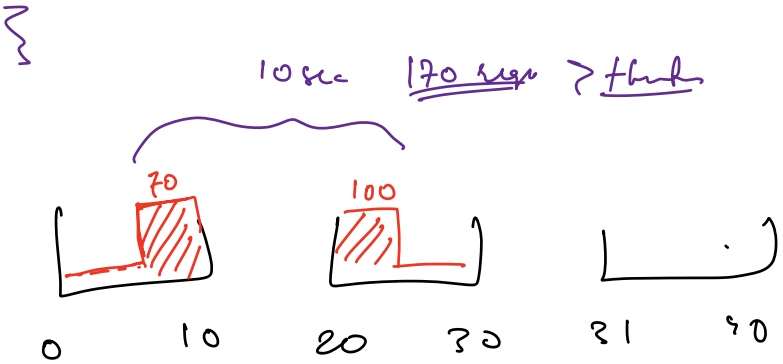
Bucketing time & use counter

100 req / 10 seconds

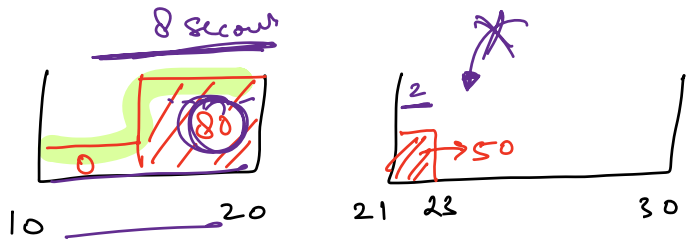
```
start_time = now()
counter = 0

void handle_request (r) {
    curr_time = now()
    if (curr - start > 10) {
        counter = 0
        start = curr
    }
    counter ++
    if (counter > 100)
        reject
    else
        accept
```





approximate (Sliding Window Counter)



$50 + 64 \geq 100$

64 req/s

last bucket cnt

can be a cnt.

map < user id, pair < int, int >>

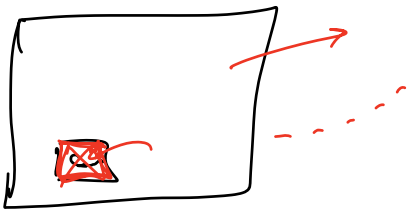
Token

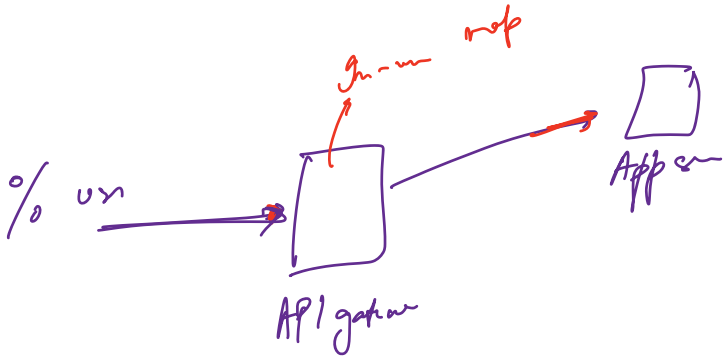
token count

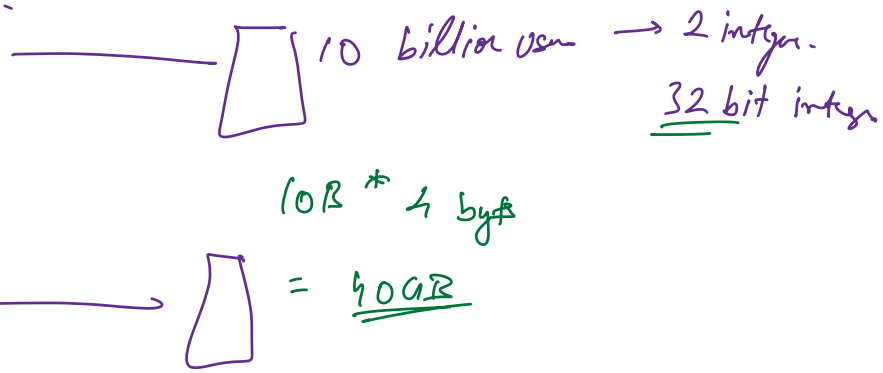
map < user_id, int >

7Bs / week

10:40 → 10:50




% users
API gateway
App servers

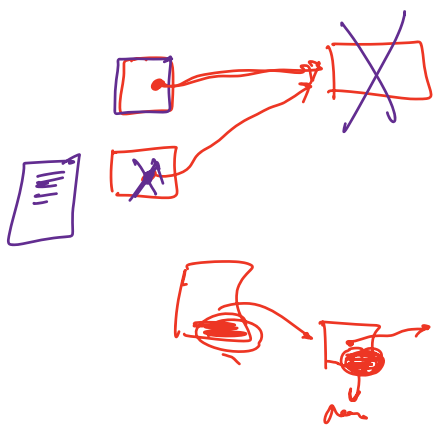

10 billion users → 2 integers.
32 bit integer
10B * 4 bytes
= 40GB

Garbage Collection

Why? if we allocate memory & never de-allocate it

Memory Leak

1. Manual Memory Mangement (C)
   - ↳ 80% bugs are due to MM
   - ↳ leaks
   - ↳ dangling pointer → pointer to memory that does Not exist
   - ↳ double free



2. Program to do memory mangt for us
   - Garbage Collector.
3. Rust → Borrow Checker

Common Issues with GC

- → overhead
- → Some GC algo can lead to unpredictable pause!!

Functional Requirements

1. system finds & removes objects that are no longer in use
2. correctness → it should not release any object that is in-use

Non-functional Requ

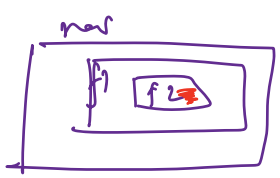
1. High performance / Low overhead
2. Predictable

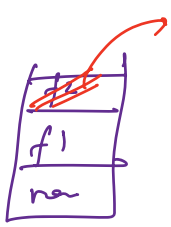
Stack Memory & Heap Memory

Call stack for each thread
- ↳ functions currently running
- ↳ variables have been declared & are in scope.

any primitive variables / any static variables

int/bool/--





all variables → Stack

↓

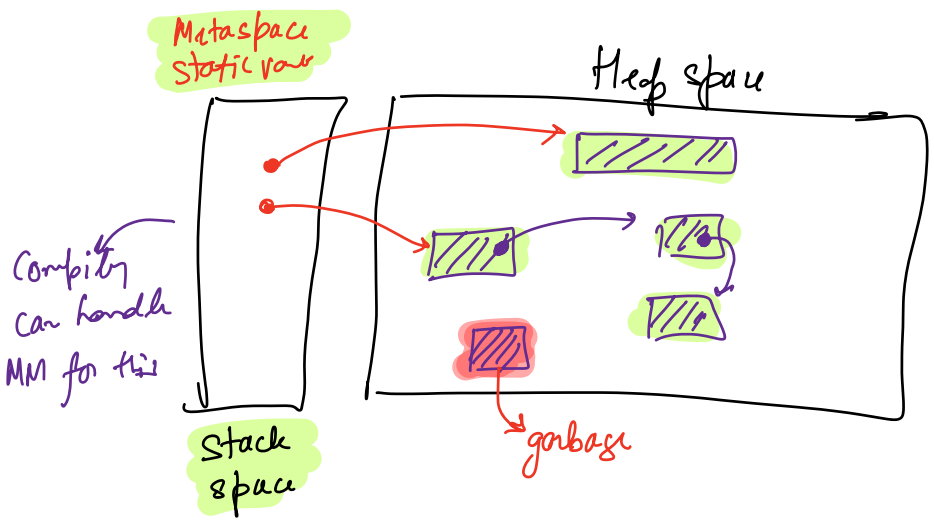
Primitive type

int a; → 4 bytes

bool b; → 1 byte

object Person p; → pointer 8 bytes

object List<Integer> l; → points 8 bytes




```
if(a){
    pr(a);
}
a=10;
```

Mark & Sweep

(1) stops the world (Pause any mutator threads in your program)

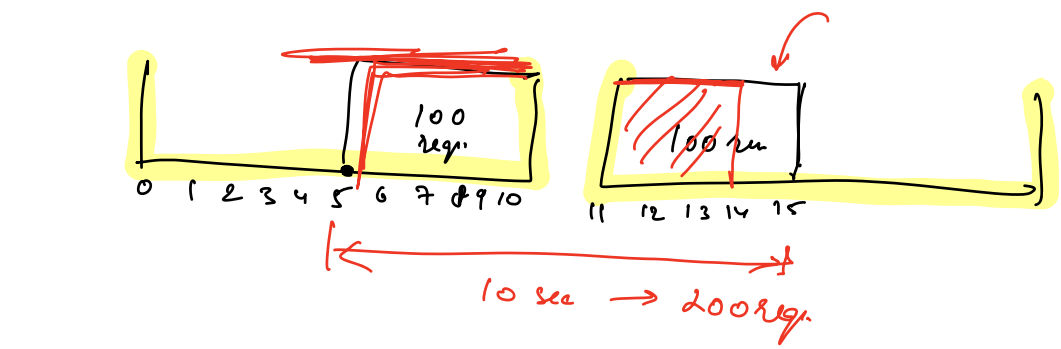
(2) start from (stack + static + metaspace)

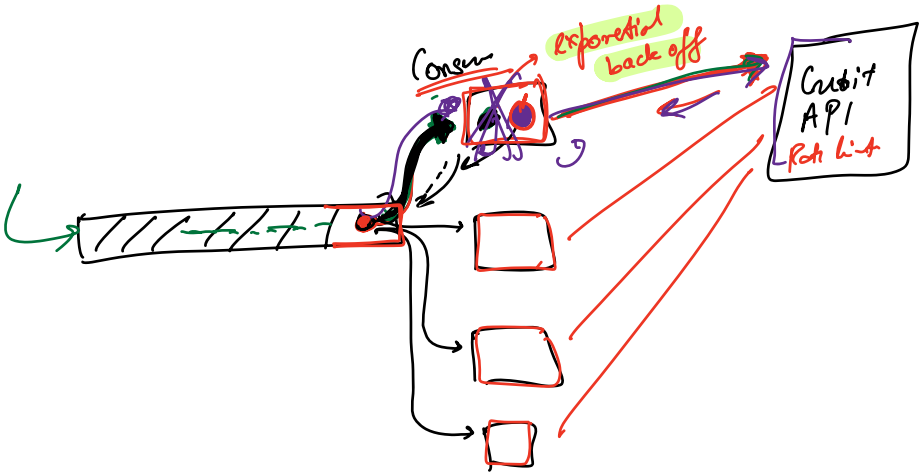
(3) DFS → marks all reachable objects

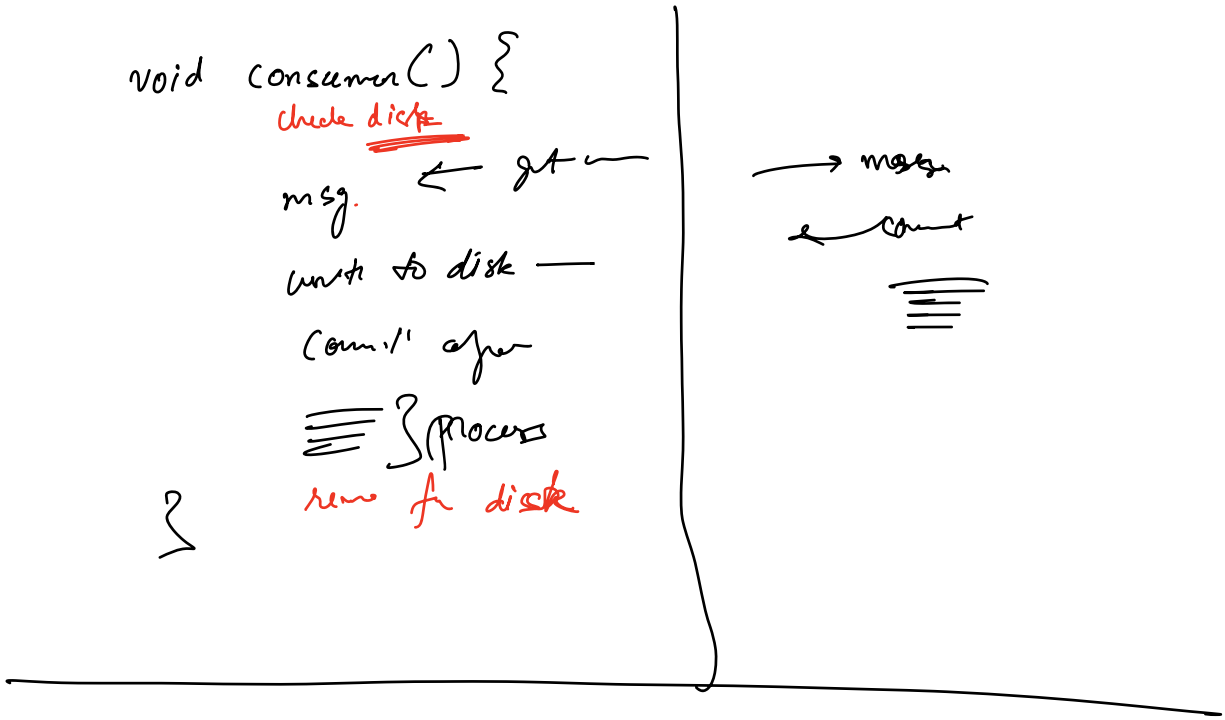
5 seconds (4) Sweep entire heap & remove any unmarked objects

(5) Defragment memory

Generation GC, Reference Counting, Tri-color marking scheme, G1







	Anurag	Kaushik	Ashwith	Bhushan
Age	25	30	20	26
Height	6'	12'	5"	5'

Primary Index on Age