Agenda

pragy@scaler.com

- Working with large files
- Nearest Neighbour Queries

Quad Tree

Amazon S3 / HDFS → Hadoop Distributed File System.

Simple Storage Service

Images / Videos → User generated content ⇒ few Mbs, few Gbs.

App server logs → append only files → 50 Tbs, 100 Tbs

Stock broker / Financial Institute

↳ mandated by govt to maintain logs.

**Requirements**

→ store extremely large files (100 TB)

→ Storage must be reliable & durable

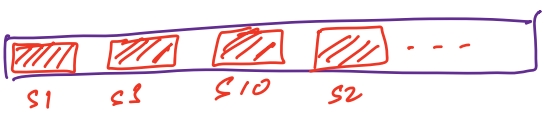
→ Both read & write must be possible

↳ read → download/ file on demand stream

→ perform analytics on top of the file.

How to store large file → split it into **chunks**

& distribute chunks across servers

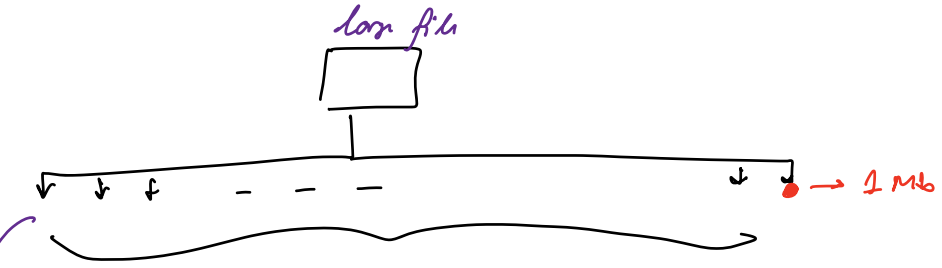



**Chunk Size** → 128 Mb in HDFS

1 Mb file of 50 TB

# of chunks $\frac{50 \times 10^{12} \text{ bytes}}{1 \times 10^{6} \text{ bytes}} = 5 \times 10^{7}$

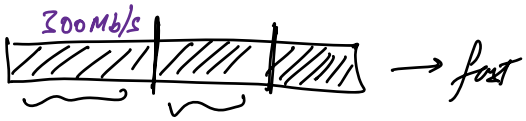
= 50 Million chunks




① Merging / collation → if client wants to download we have to merge these chunks.

↓

overhead grows as chunk size gets smaller.




System 32 dll's in Windows
node_modules/...
} → same 3GB in total.
1Mb/sec

② Store metadata about chunks

every chunk → ① file it is part of → 8 bytes
② offset → 8 bytes
③ server stored on. → 4 bytes

20 Bytes / chunk.

50M * 20 bytes = 1000M byte = 1GB
of metadata.

# HDFS

① Name Node → also replicated (Master - slave)
↓
Immediate consistency

↳ Only 1 name node

↳ Stores the metadata

② Data Node

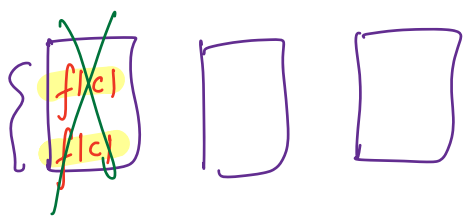
↳ stores chunks

Every chunk is replicated

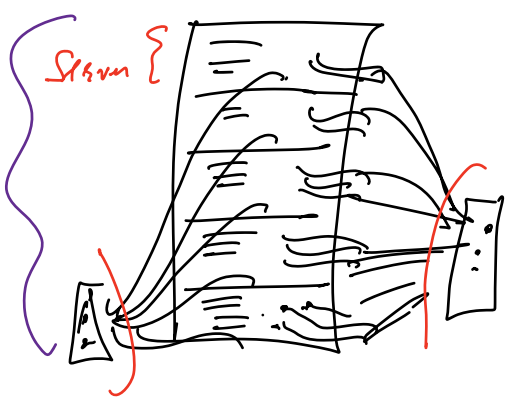
& stored in multiple data Nodes


↳ Rack - aware algorithm

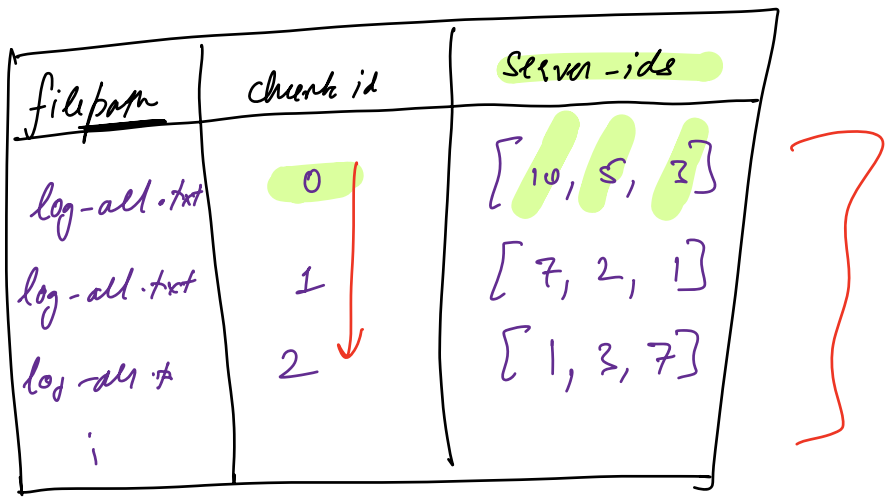
Replicas should be on same machine



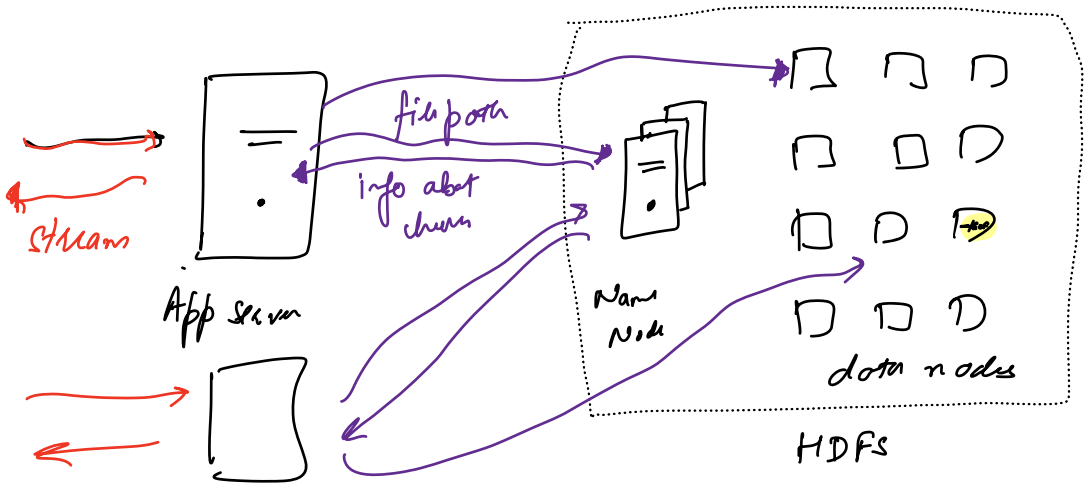

Rack.



Name Node


filepath
chunk id
server-ids
log-all.txt
log-all.txt
log-all.txt
0
1
2
[10, 5, 3]
[7, 2, 1]
[1, 3, 7]

# Read / Write in HDFS




(1) Read

↳ go to nam node & find metadata about file's chunks

↳ for each chunk (sequentially in order of chunks)

↳ goto servers containing chunk

↳ load the chunk in ram (128 MB)

↳ Stream to the client

Map - Reduce

aggregate function

② Write

↳ client will start sending large data stream to app server.

↳ app server will go to Name Node & find chunk size

↳ the moment app server got 128 MB in buffer

① go to NameNode
where should I store this new chunk?
← server id

② go to server-id & write this 128 MB chunk

③ Name Node → I have written if this chunk_id at server-id

→ clear the buffer & continue receiving from client

App Server

```
byte [128*1000*1000] buffer;  → in-memory.
int count = 0;

void recieve() {
    byte data;
    while ( data = recieve from Client() ≠ EOF){
        buffer [count ++] = data;
        if ( count == 128*1000*1000){
            store (buffer) //async
        }
        count = 0;
    }
}
```

What if a file is not in multiples of 128Mb

5Mb

200Mb

1 client

C0 → 5MB

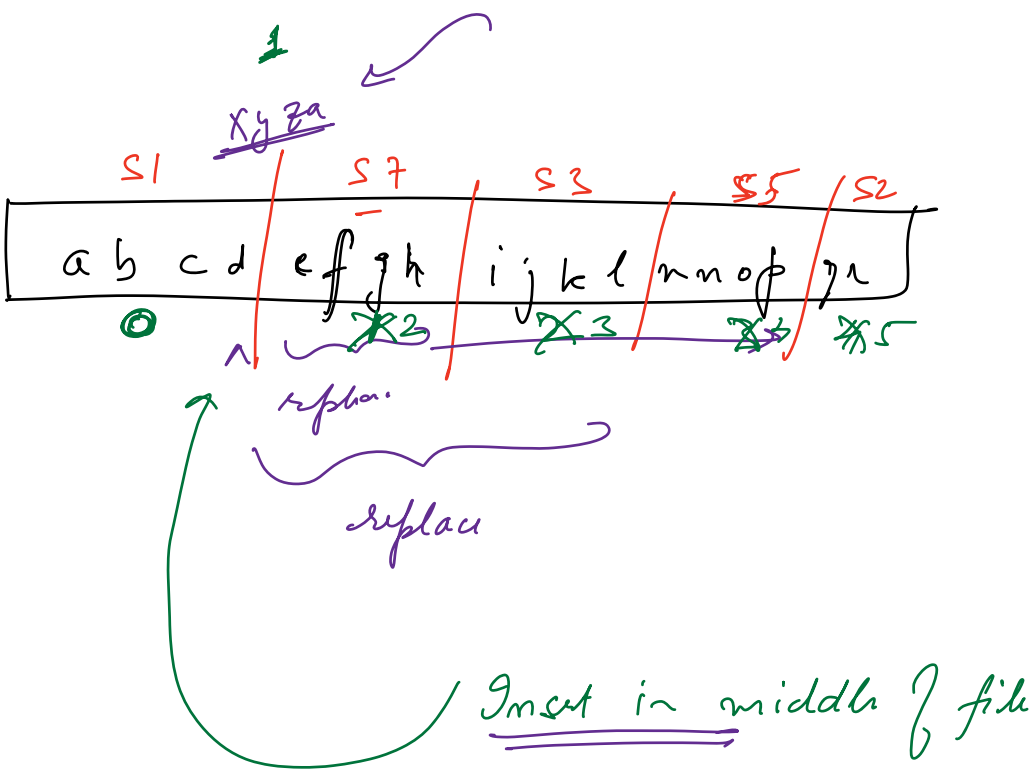
2 clients

C0 → 120 MB

C1 → 72 MB

file parts

10:21 → 10:35



## Nearest Neighbour

Uber → location of user
→ location of driver

find all drivers near me

near → mean diff thing based on situation

Zomato → location of user

find all restaurants near me.

→ location of delivery partner
→ location of restaurant

find all delivery partners near the restaurant

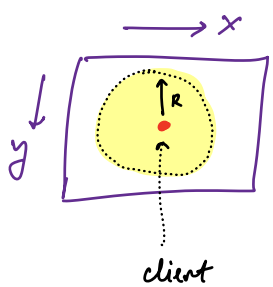
Google Maps → location of user
→ location of a place

find all pharmacies near me

## SQL

restaurants

id name _ _ lat lon




## Circle Based

query for find all restaurants near me.

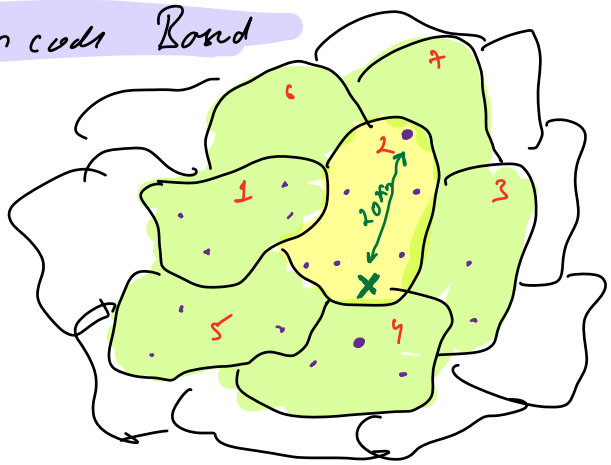
```
Select * from restaurants
where euclid_distance_2d(lat, lon, user_lat, user_lon)
<= R;
```

Safe
slow
contain table
linear scan.

$$euclid\_distance\_2d(x1, y1, x2, y2) = \sqrt{(x1-x2)^2 + (y1-y2)^2}$$

1.4 Million restaurants

## Pincode Based




Pro → easy to query.

restaurants

| id | name | — | pincode |
|---|---|---|---|

select * from rest
where pincode = user_pincode

Cons → other restaurants in nearby pincodes that are closer.

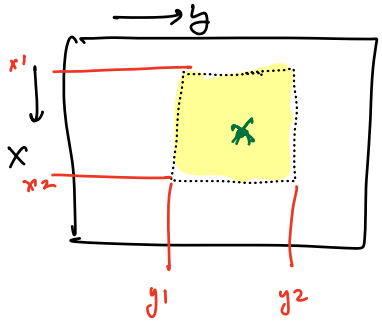
Pincode neighbors

| pincode | neighbor |
|---|---|
| 2 | 1 |
| 2 | 3 |

→ what if a pin code is large / neighbors
distance

→ what if there are 100 many restaurants in the pincode?
slow queries

# Rectangular / Box Bound

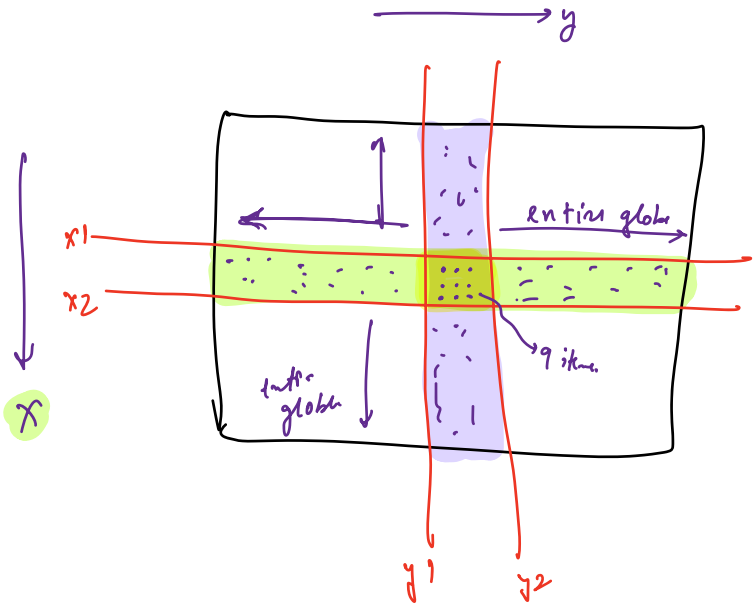



slow but fast put circle

Select * from restaurant where lat between x1 and x2 and lon betwn y1 and y2

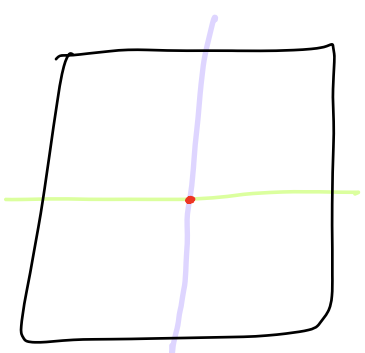
index index

index scan on lat
linear scan on lon




A) find ▭ via index
linear search for $y1 \le lon \le y2$

B) find ▭ via index
linear scan for $x1 \le x \le x2$



C) find ▭ via index
find ▭ via index
intersect.

K-d tree → generalization/DS that allows you to use multiple indexes at the same time.

→ it builds a k-dimensional index

bad worst case $O(N)$

v.good avg case $O(\lg N)$

K-d tree K = 1 → BST

K-d tree K = 2 → Quad Tree

K-d tree K = ? → Oct Tree

$k << N$

(if $k \not\ll N$ → curse of dimensionality)

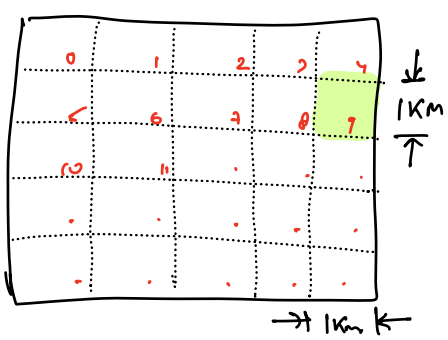
k: dimensionality.

N: # of items (dim locations, rest locat)

Grid Based → basically pin-code on steroids

we control the shape

we control the size

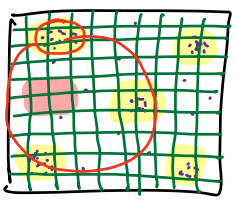


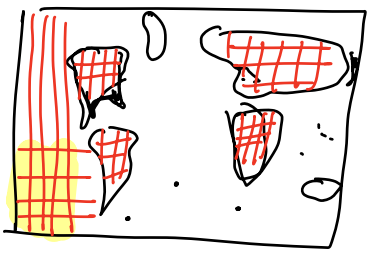

rest | na | cell-id

id

## challenges

- high density hubs
- low density areas



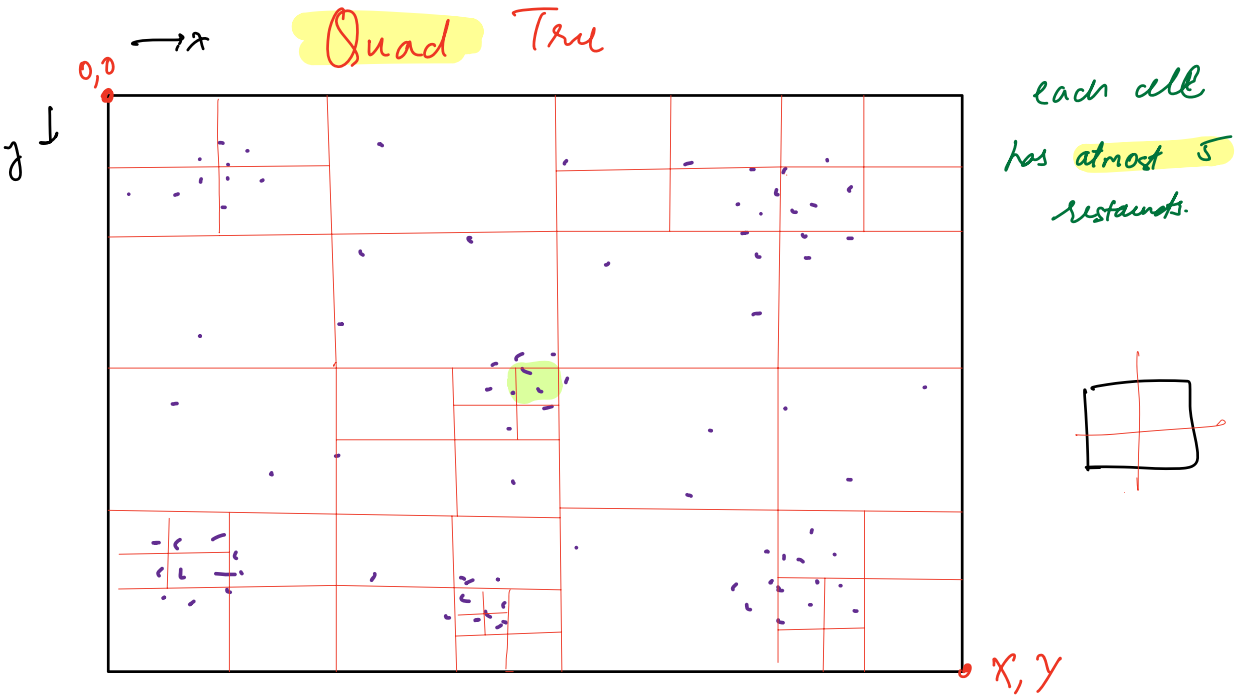


## No optimal grid size

- high density
  - search smaller neighborhood
- low density
  - expand search to wider area.

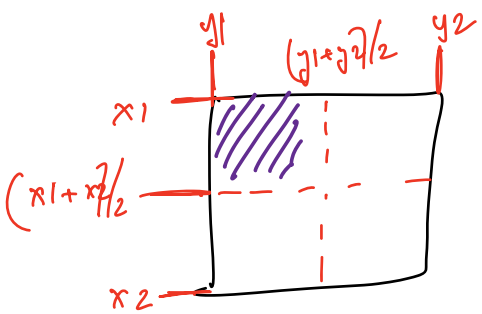
scaled out not in grid cell (x)

**Magic Solution** → a dynamic grid that auto adjusts to density of a region.




Start with the entire world as a node

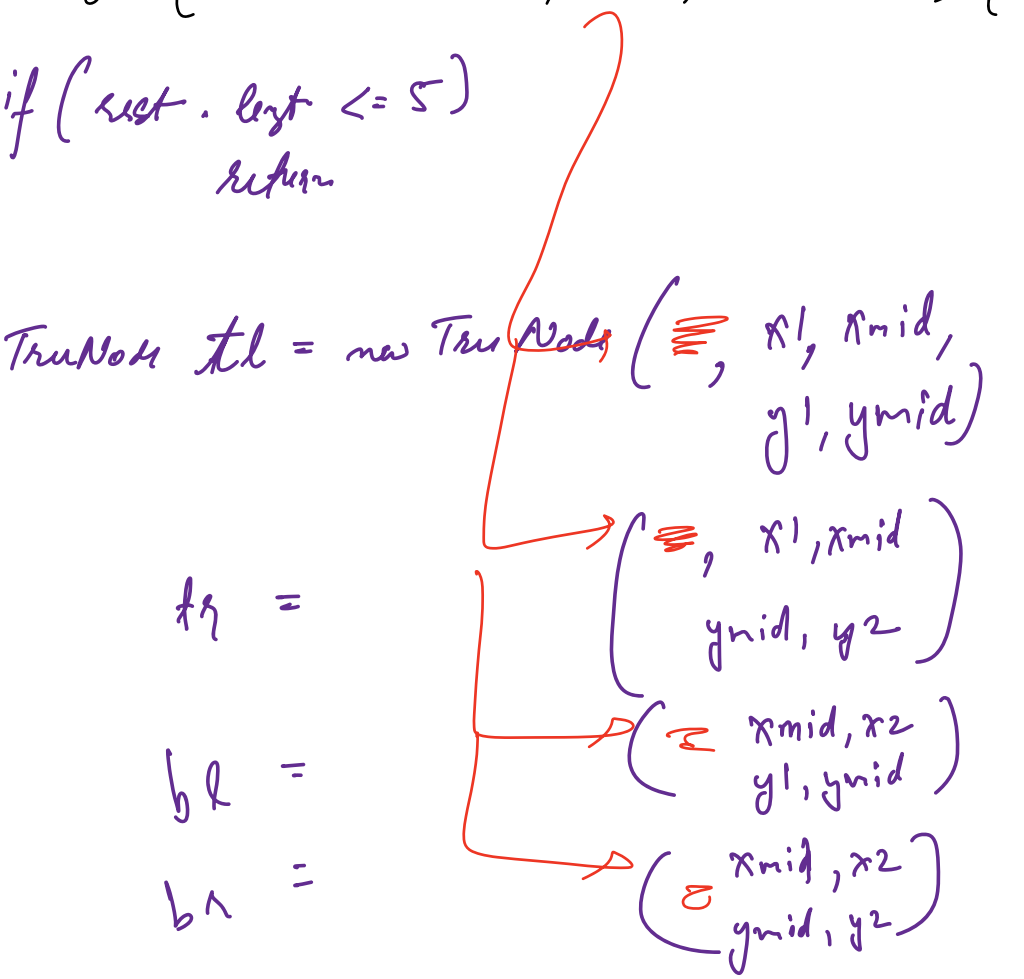
- if node has > 5 restaurants
  - split into 4 quadrants of equal size
    - recurse on each quadrant.



dimensions of current cell

```
TreeNode create_grid(List<Point2D> restaurants, x1, x2, y1, y2) {
    if (rest.length <= 5)
        return

    TreeNode tl = new TreeNode(_, x1, xmid, y1, ymid)
    tr =          (_, xmid, x2, y1, ymid)
    bl =          (_, x1, xmid, ymid, y2)
    br =          (_, xmid, x2, ymid, y2)
}
```

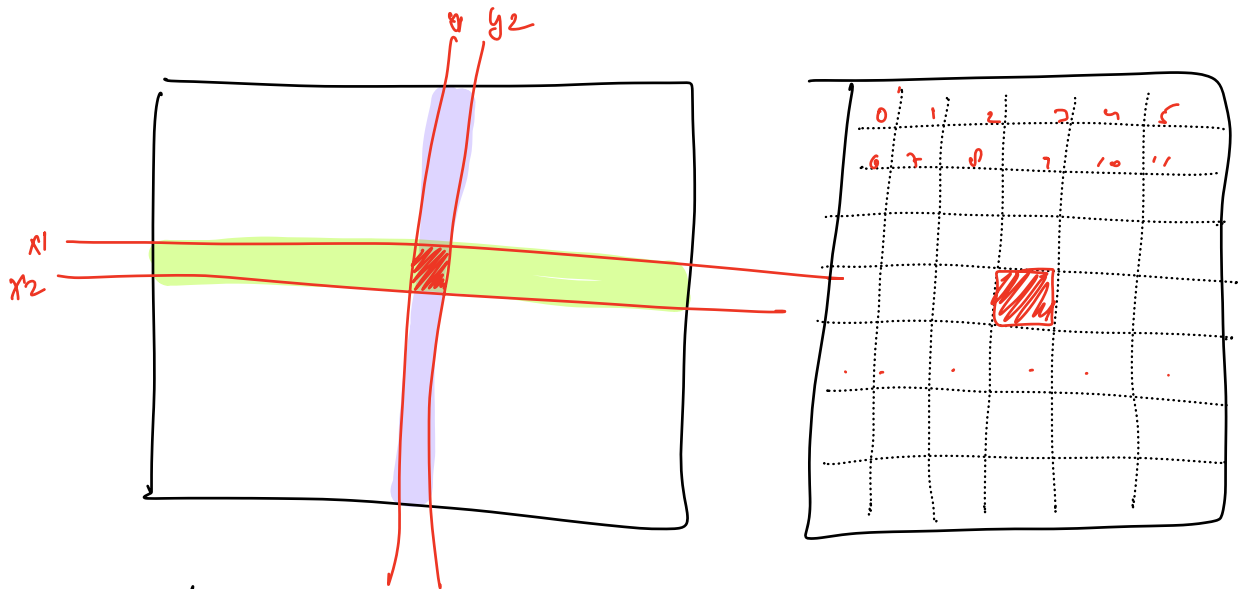


upload to GDrive ⟶ 10 MBPS

↳ 200 KBPS for small file

Big file ⟶ 700 MBPS

10 MBPS




id / name / x / y

id / name / cell-id

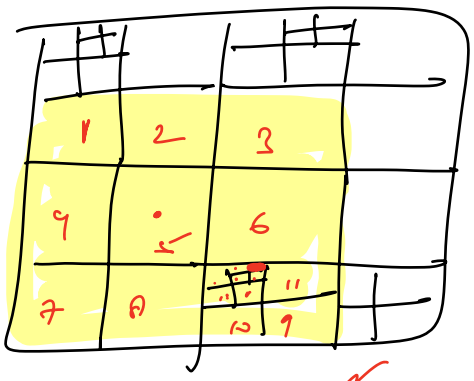
select ——

where x between x1 & x2

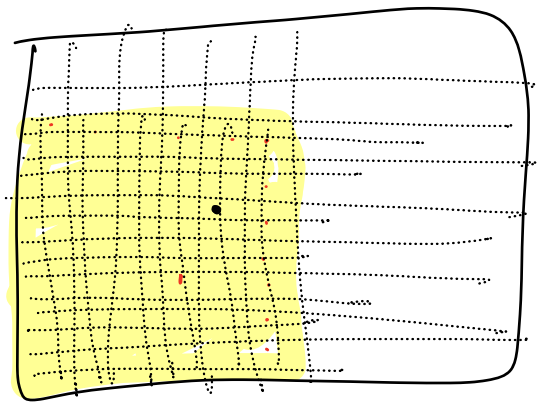
y in y1 & y2

or ——

where cell id = ——




18 cells



168 cells

pragy@scaler.com