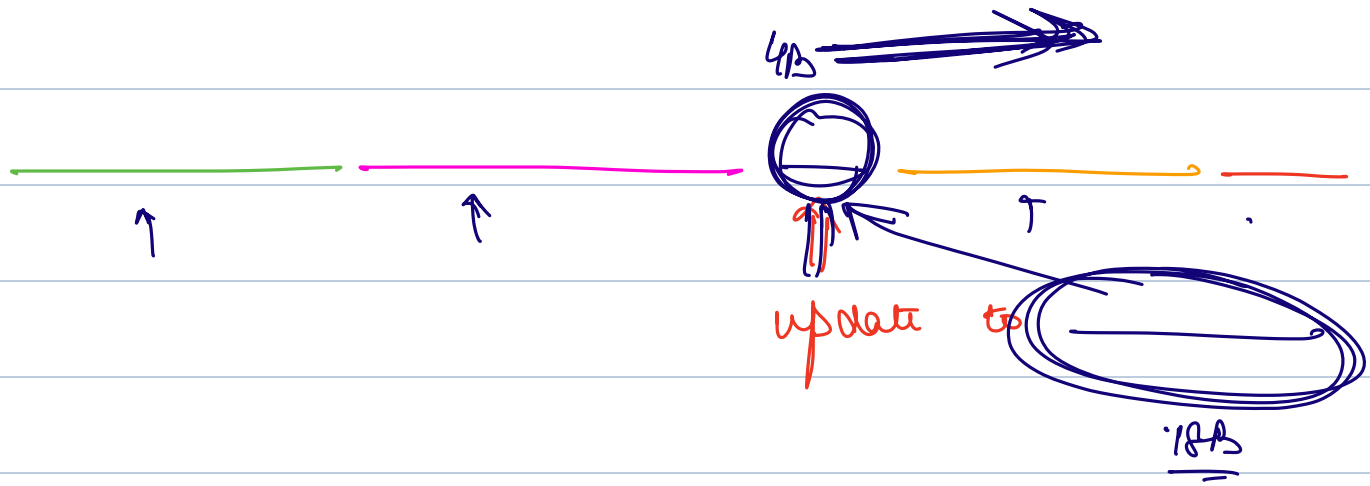
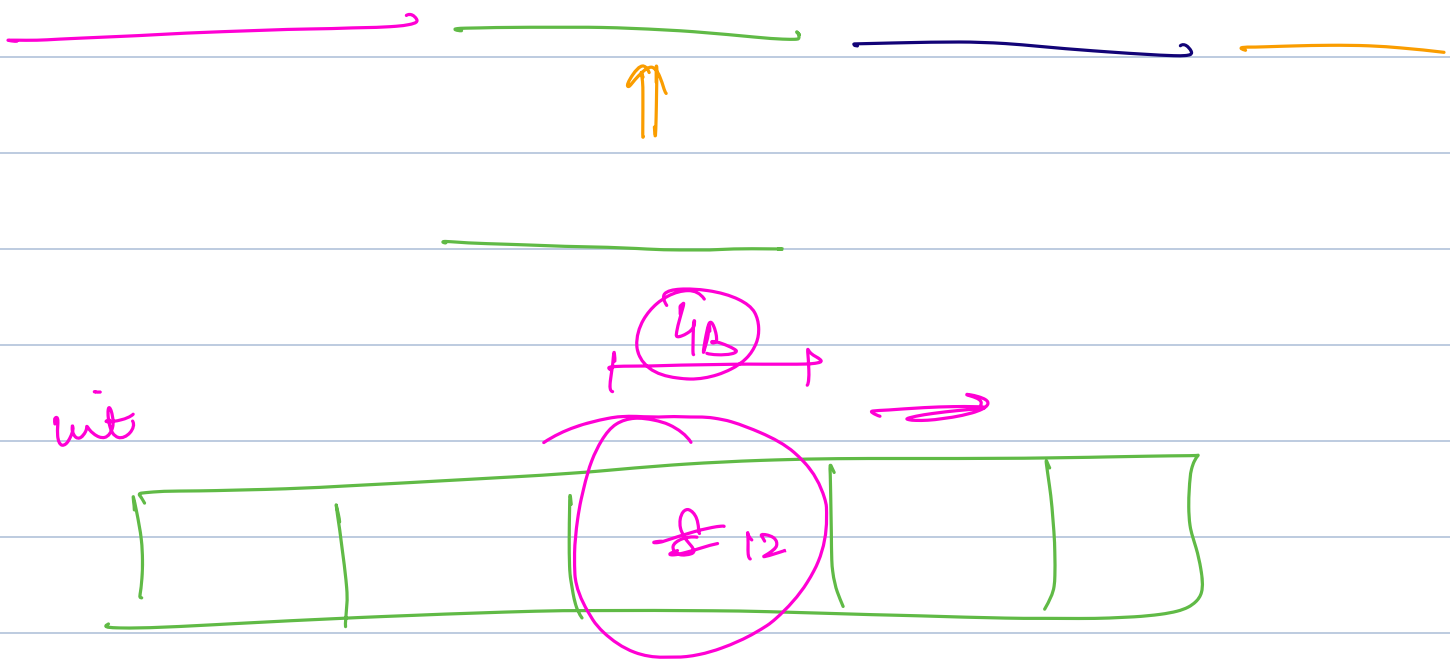
Will start at 9:10 AM

NoSQL Internals

Storing data in NoSQL DB is hard

⇒ every entity may have a diff size





Camera

{id: 101} {id: title:} {id: title ram: os}

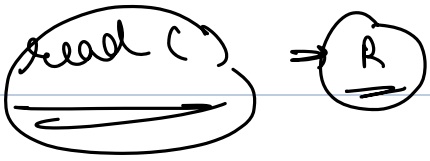
{id: title: fabric wand}

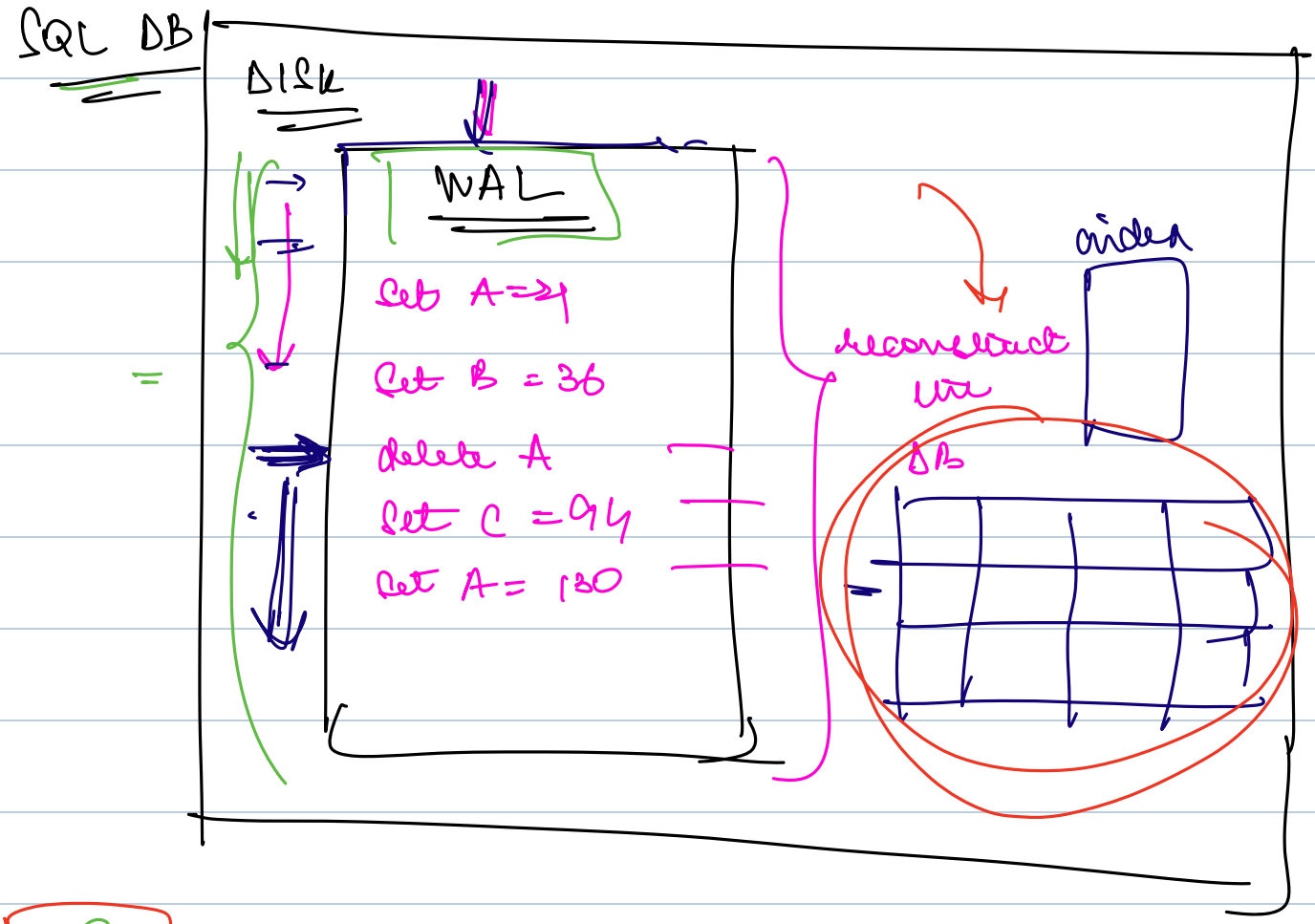
How do NoSQL DB Store Data

(1) every DB (SQL or NoSQL) typically stores data at 2 places:

(i) WAL (Write Ahead Log)

(ii) Storage from where data is queried.





1. WAL: append only file
   - ↳ never editing already written data
   - ↳ only adding new data at end

↳ whenever a write query happens, the details of that query are appended to this file

→ we can reconstruct DB using this file

→ This file is used by replicas to

Sync as well.

→ isn't optimal for reads!!

↳ reads will be VV slow because traverse the whole file.

② Actual Storage

→ data is stored in a way easier to query!

SQLDB ⇒ Tabular Format

⇒ for every entry you reserve a particular amount of space.

⇒ in NoSQL DB this isn't only.

WAL

Set 1 = {id: 1, title: iphone}

Set 1 = {id: 1, title: ram: 8}

Set 2 = {id: , title: , fabric: , color}

HOW EXACTLY DATA IS STORED IN NO SQL

Qn 1 We dont store anywhere

⇒ only rely on WAL

```
write (key, value) {
    wal.append(key, value)
}
```

WAL (Disk)

Set A = 1

~~Set B = 13~~

Set A = 7

Set C = 40

Set B = -1 ← Special value for delete

Set B = 13

delete ⇒ set to a special value

```
read (key) {
    traverse wal from start to end:
        find last occurence of key:
            return that value.

    return -1
}
```

doesn't exist

0 1 0 0 1 0 1 0 1 1

0 1 0 0 1 0 1 1 0 1 0 1 0 0 0 0 1 1 1

Op^n 1 : WAL Only

```
write(key, value) {
    wal.append(key, value)
}
```

$\Rightarrow$ $\boxed{O(1)}$

```
read(key) {
    traverse wal from start to end:
        find last occ of key:
            return val
    return -1
}
```

$\Rightarrow$ $O(N)$ — # of updates

V.V.V. slow

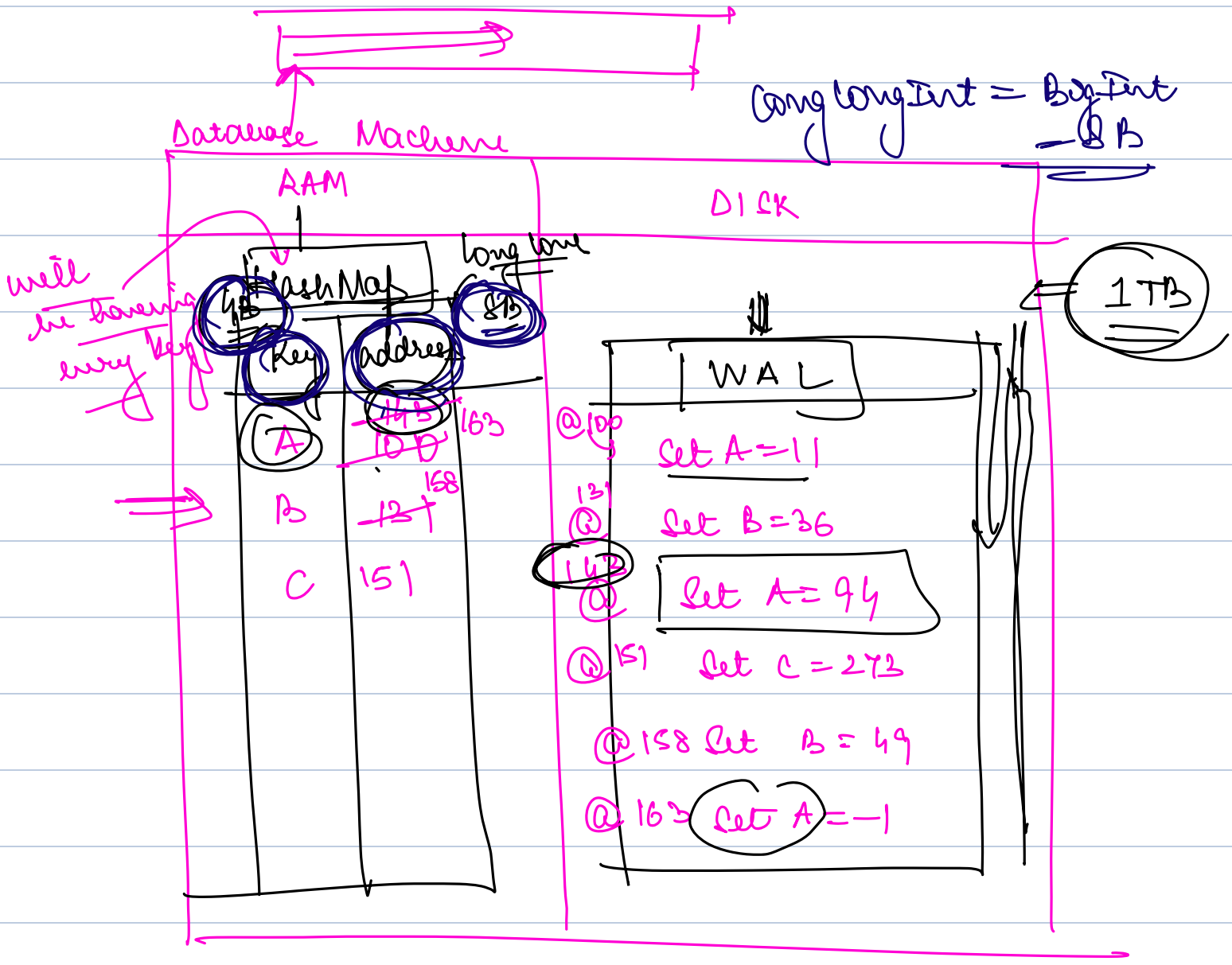
# of entities

$\boxed{\text{W/S in SQL} \Rightarrow O(\log N) \Rightarrow R, \quad O(\log N) \Rightarrow W}$

Op^n 2

WAL + HashMap

Someone who told me for every key what is its latest loc^n in WAL



```
write (key, value) {
    cur_last_add_wal = wal.end()
    => wal.append(key, value)
    hm.put(key, cur_last_add_wal)
}
```

O(1)

```
read (key) {
    add = hm.get(key)
    if (add == null) {
        return -1;
    }
    read disk at that add
    return the value.
}
```

O(1)

k-v DB

key : int (4)

val : int (4)

→ 1 k-v pair = 8B

→ in 1 Machine

$\Rightarrow 10^{12}$ B $\quad 1TB = 10^{12}$ B

$\cancel{8B}$ 10

$\Rightarrow 10^{11}$ B

$\Rightarrow 100 \times 10^{9}$ entries

$\Rightarrow$ 100 · Billion

12B key in our HM

$\Rightarrow$ our HM will take

$\Rightarrow 100 \times 10^{9} \times 12$ B

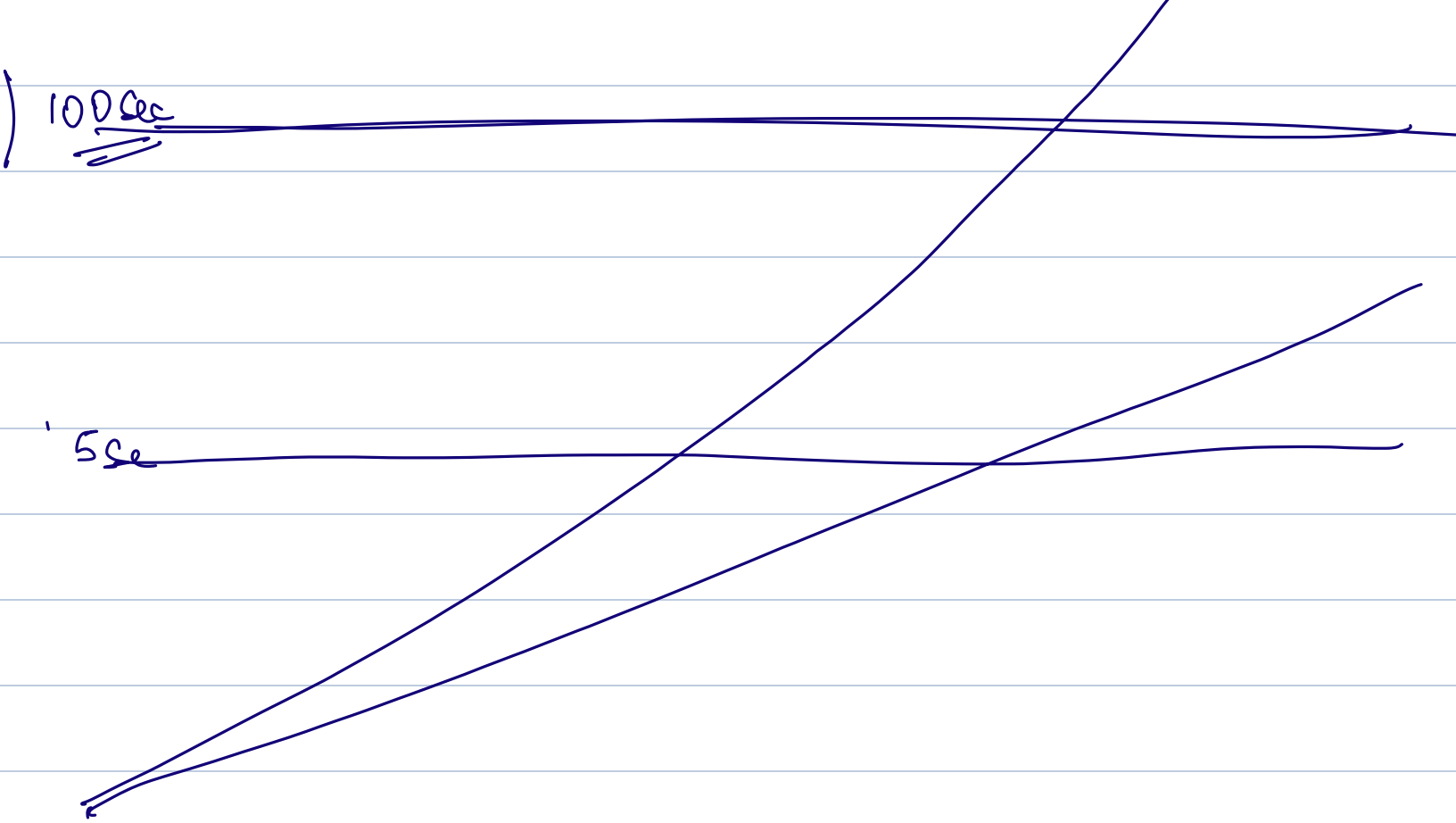
$\Rightarrow$ 1200 GB

$\Rightarrow$ 1.2 TB

we don't have this much of RAM :(

CON :

1.) we might not be able to store info about every key in the HM.



At the time of restart:

HM is reconstructed from WAL

	Writes	Reads	CON
Only WAL	$O(1)$	$O(N)$	reads are v slow
WAL + HM	$O(1)$	$O(1)$	can't have everything in HM

Opⁿ 3

Prev Solⁿ

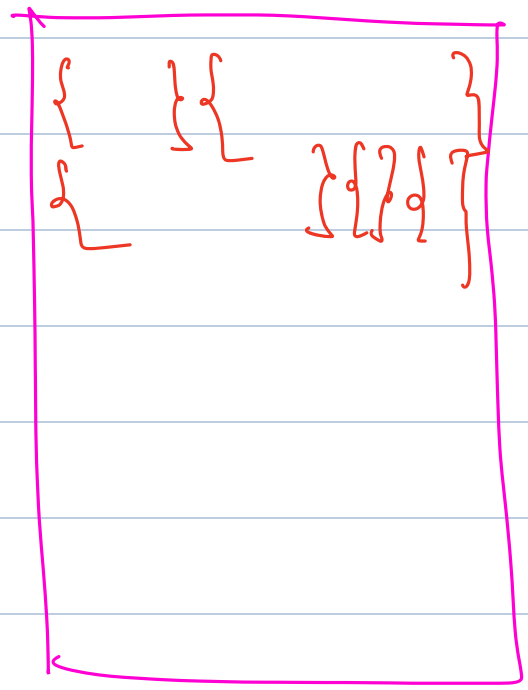
(i) huge RAM

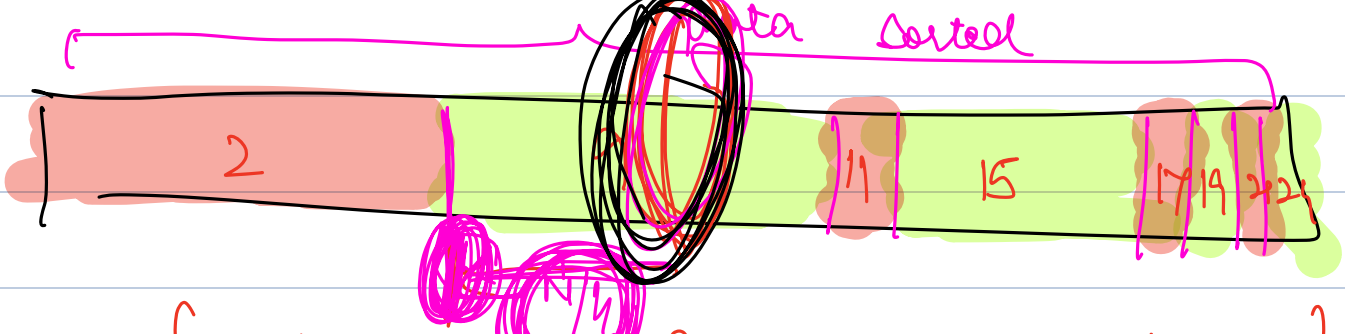
(ii) many duplicates
⇒ increase size

I am trying to somehow make queries directly on disk in faster way

Sorted

order can help

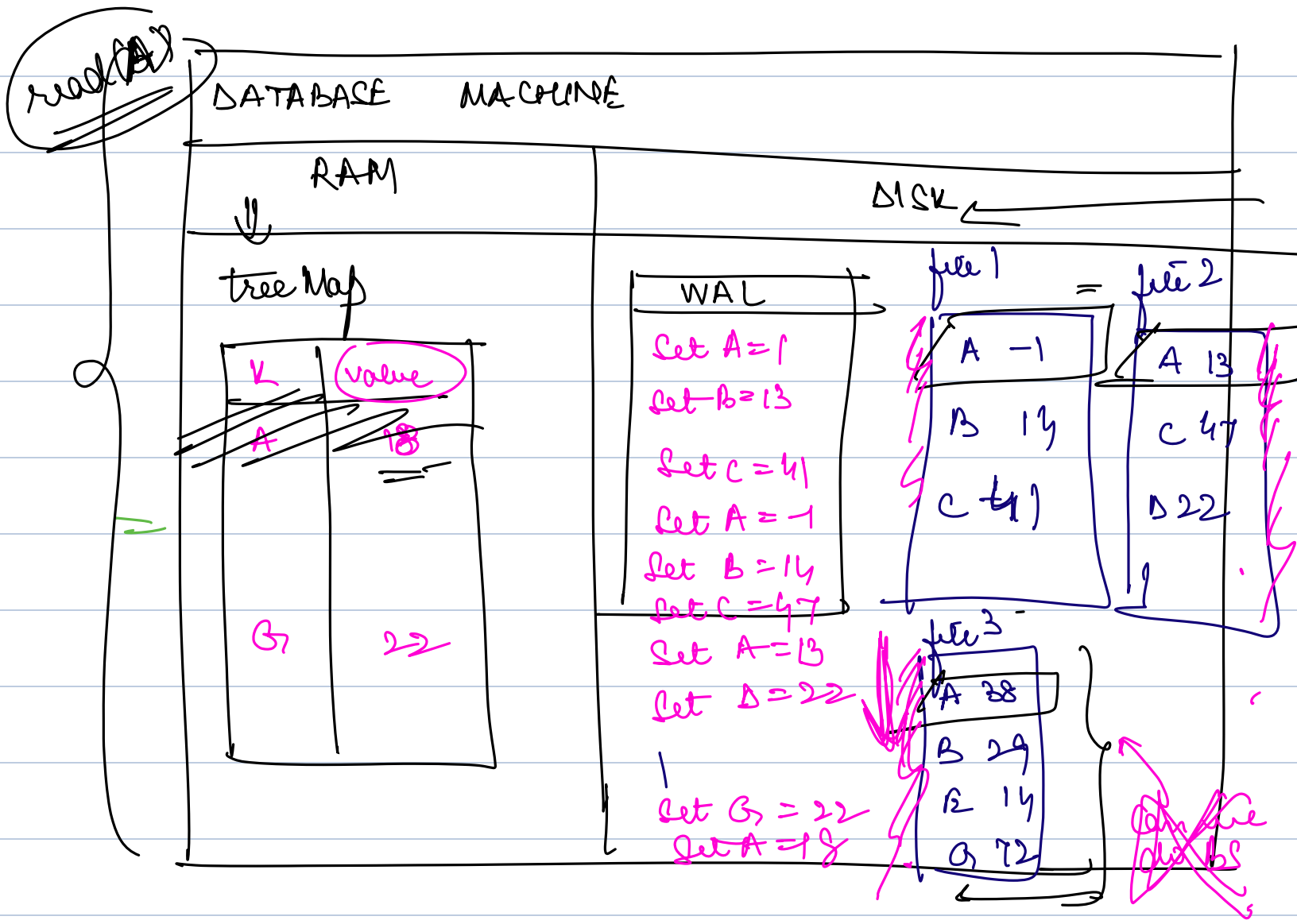




{ unfortunately BS can only be applied if data is of equal sizes }

$$\boxed{i \times (\text{size of elem})}$$

Let's try to get data in sorted way on disk



treeMap ⇒ allows us to traverse in sorted order.

(1)

```
write (Key, value) {
    O(1)        wal.append(Key, value)
    O(logN)     treeMap.put(Key, value)
}
```

O(logN)

BACKGROUND PROCESS

(2) every 1 hr,

→ convert the treeMap into a new file

→ Empty the treeMap.

(3) read(Key) {

if treeMap.contains(Key) {
return treeMap.get(Key)
}

O(log N)

for file from latest to oldest:

O(N) ⇒ if file contains Key:
return value

return -1

}

$\Rightarrow O(\log N) + (\#\text{ files}) \cdot O(N)$

$\Rightarrow$ (# files) $\times$ $O(N)$

SUMMARY

	Writes	Reads
Only WAL	$O(1)$	$O(N)$
WAL + HM	$O(1)$	$O(1)$
TreeMap + files	$O(\log N)$	$O(N)$ (# files)

opⁿ 4 : TreeMap + files + (Merge files every 1 day)

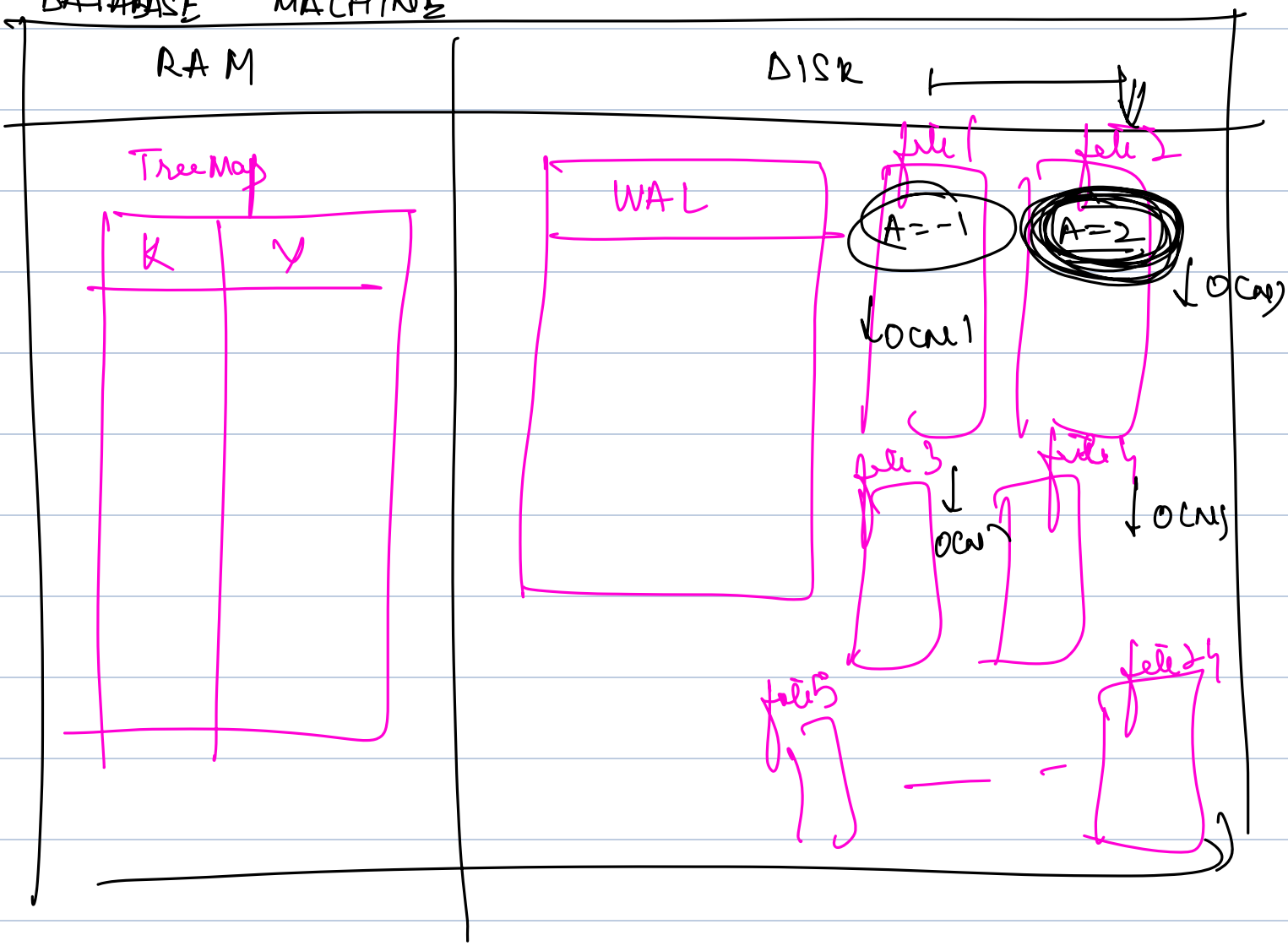
arr1 [ ]

arr2 [ ]

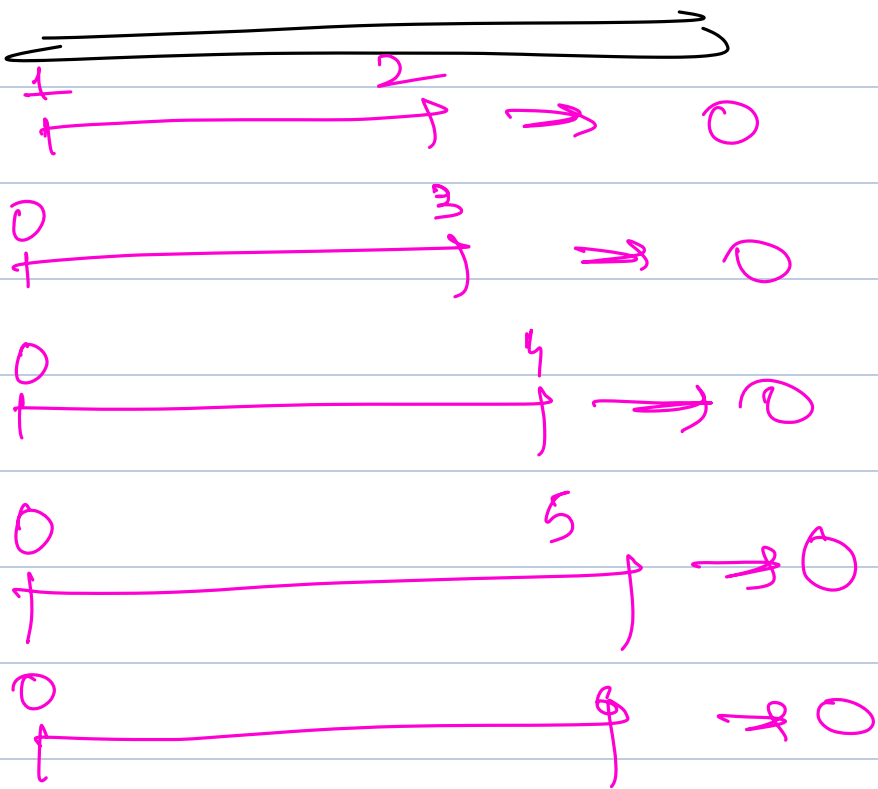
$O(M+N)$

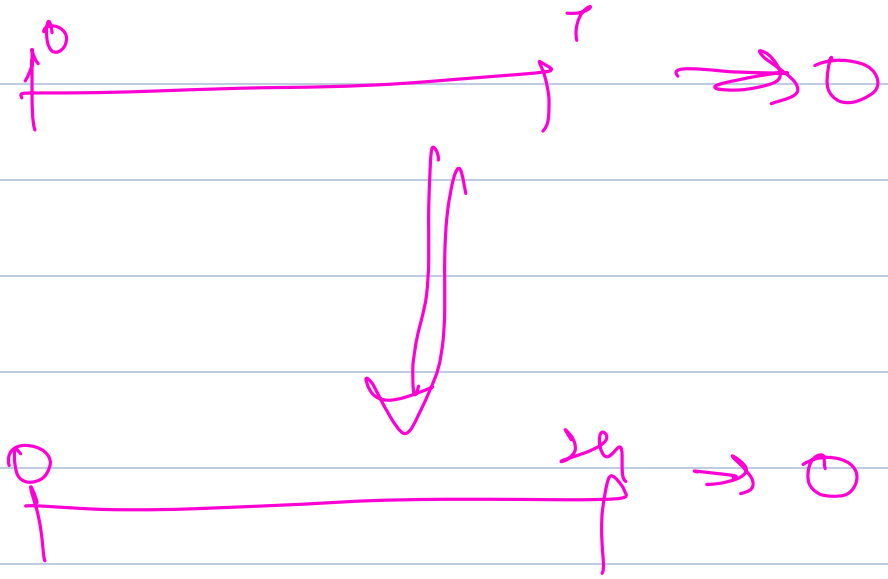
Merge 2 sorted Arrays $\Rightarrow$ 2 Pointer

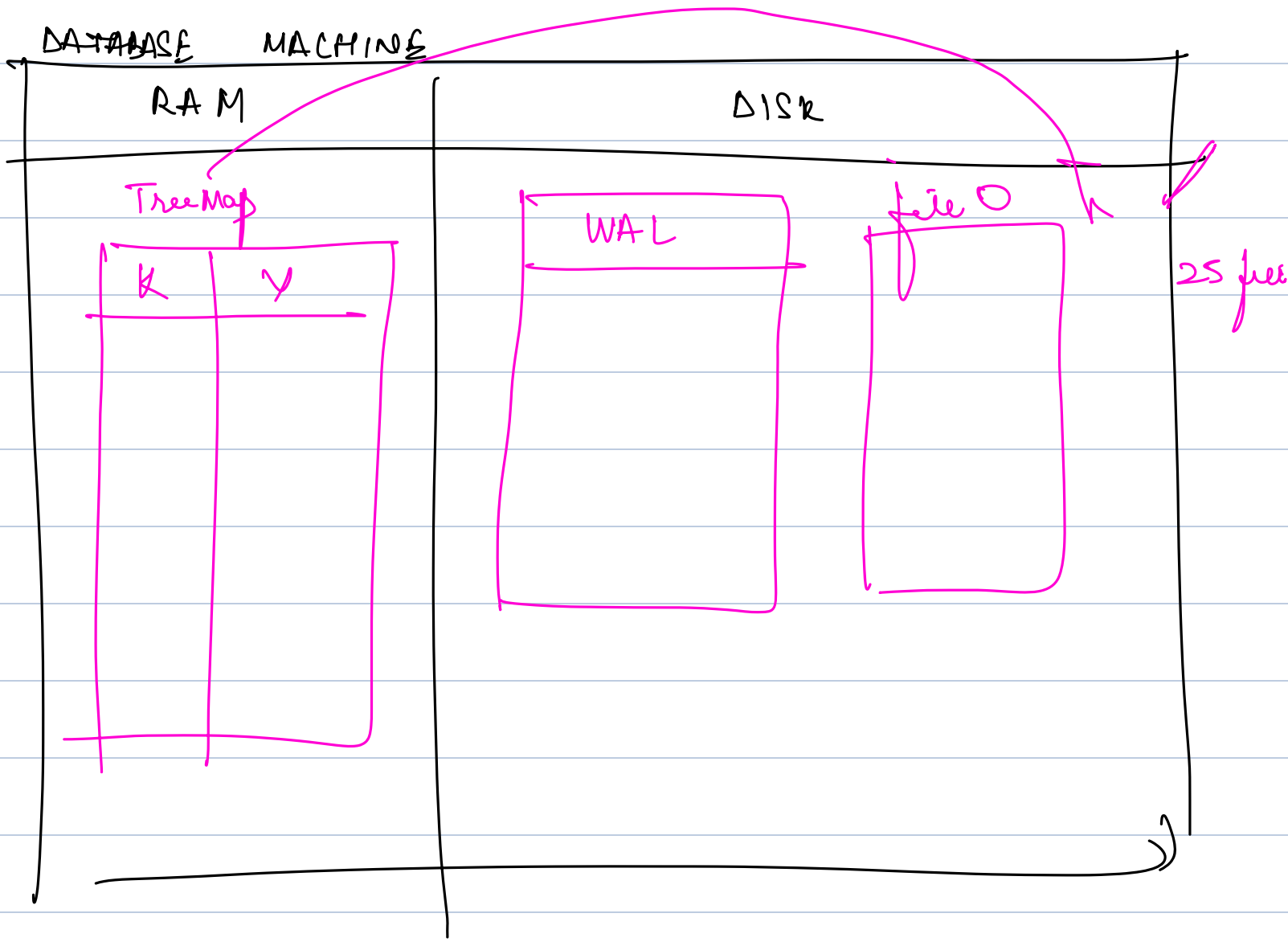
DATABASE MACHINE



BACKGROUND PROCESS 2

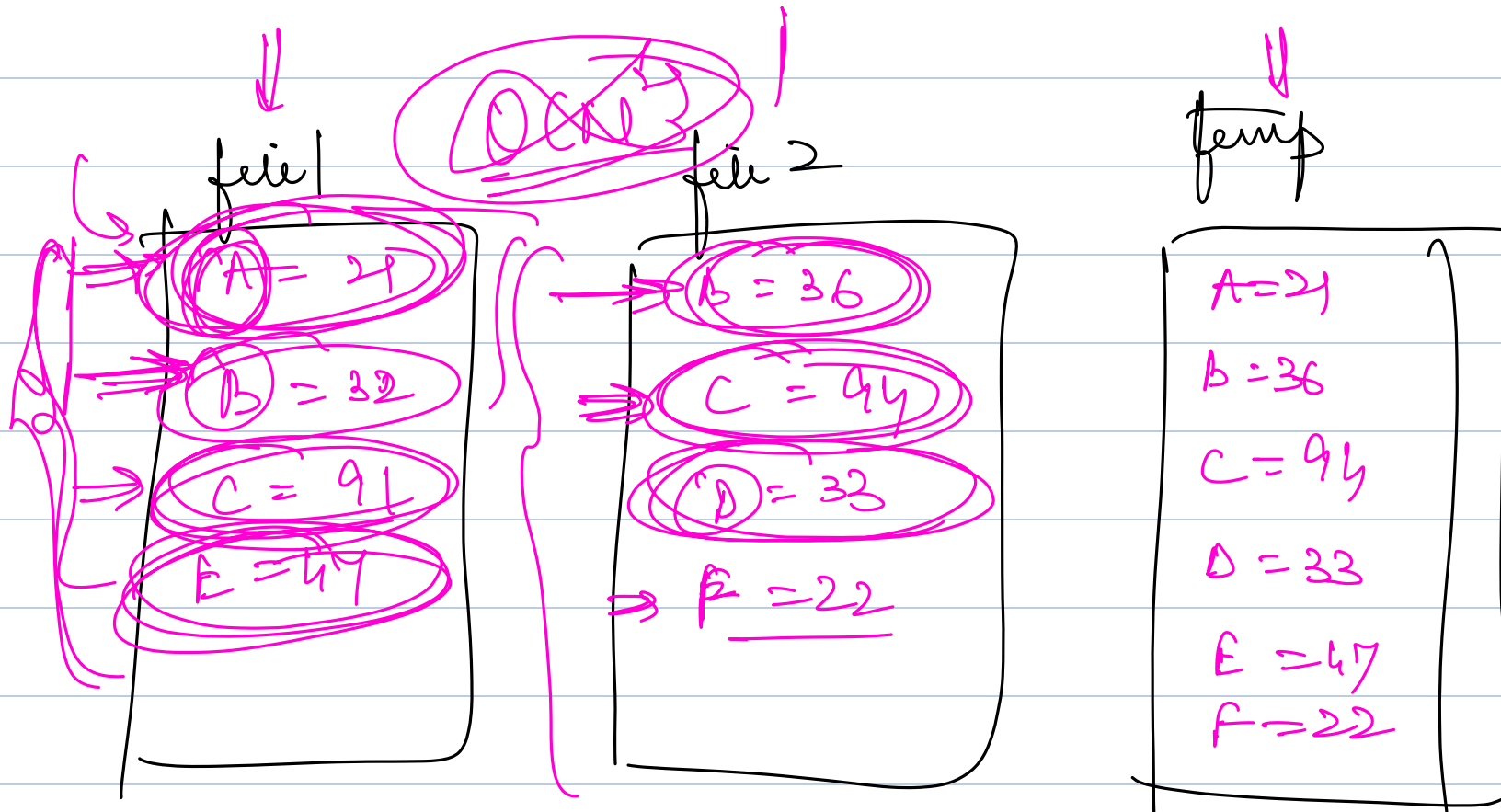






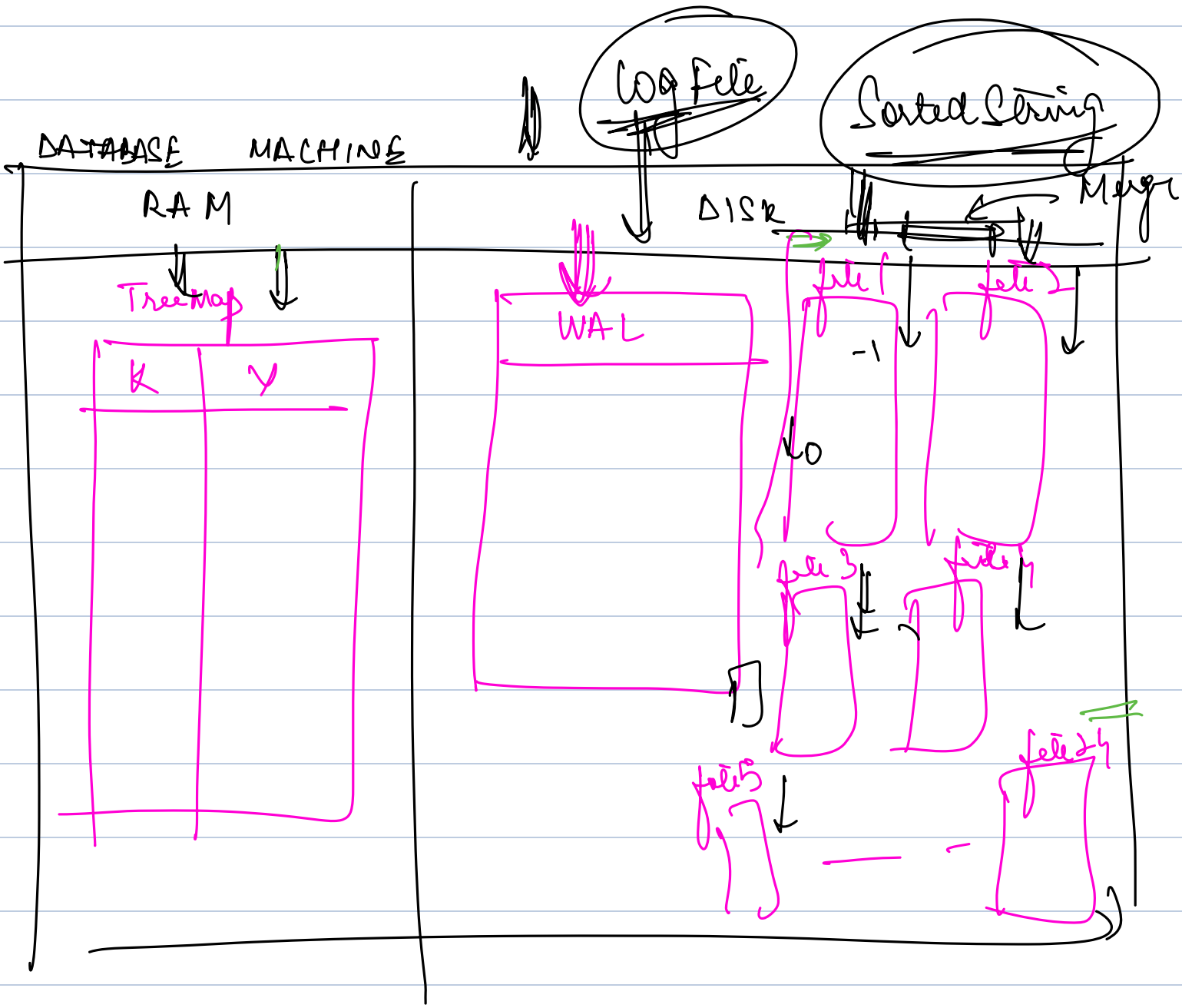
⇒ every day at end of day we will be left with only 1 free

	Writes	Reads
Only WAL	O(1)	O(N)
WAL + HM	O(1)	O(1)
TreeMap + Files	O(logN)	O(N) ^ #files
TreeMap + Files + Merging	O(logN)	O(N) ^ 25 → O(N)



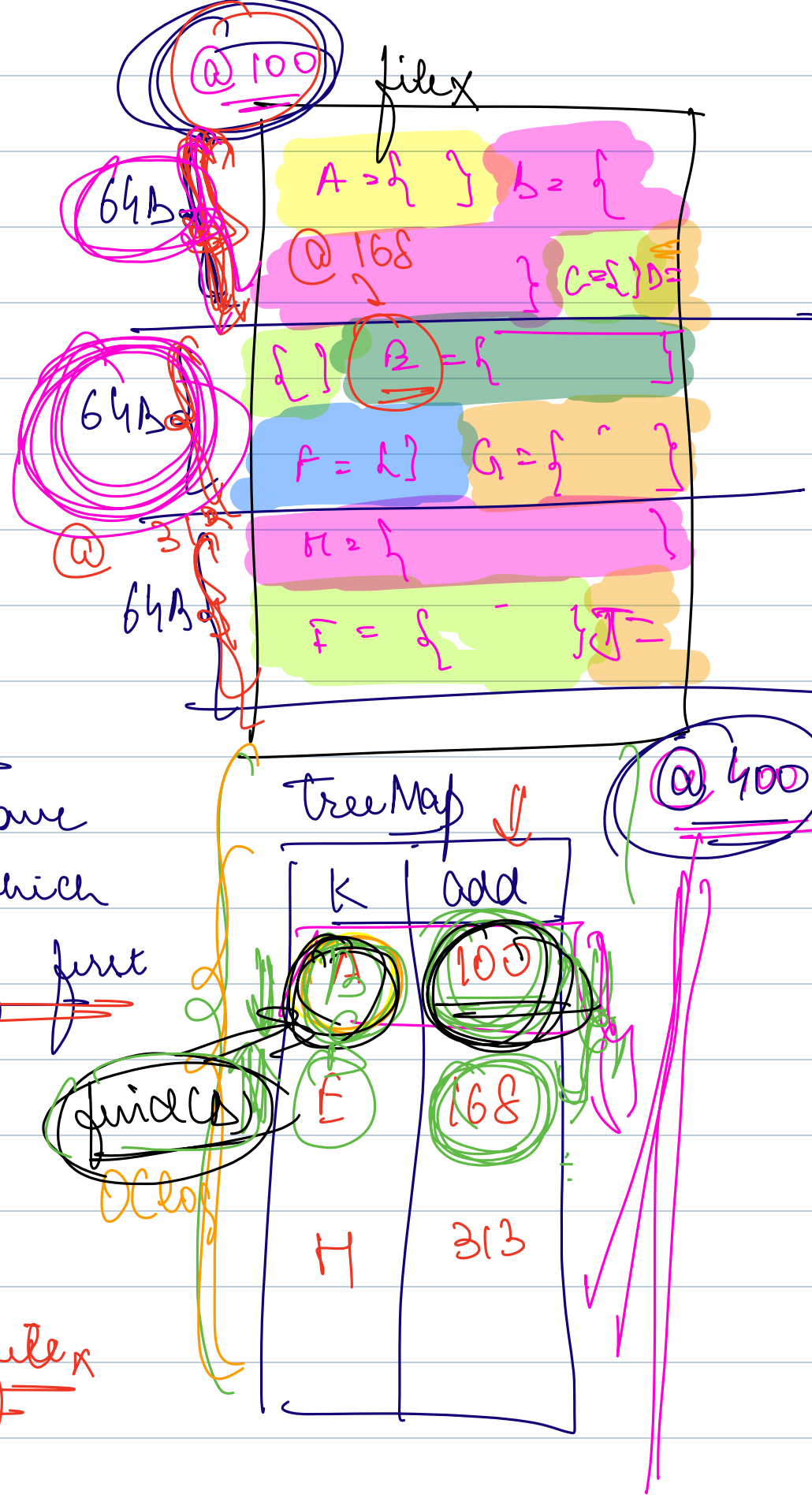
Op^n 5 : if somehow I can read a file in O(logN) times; I have my sol^n !

⇒ Similar TC as SQL



① When I am creating this file, I divide some blocks of equal size

② Alongwith each file I also have a Tree Map which stores add of first elem in each block.



find D in file x

find (elem, file) {

find add of key just <= elem in treeMap.

read at Max next 64B and if found return

}

-1

O(log N)

read (Key) {

if treeMap.contains(Key)

return treeMap.get(Key)

for each file from latest to oldest (24th to 1st):

O(log N)

from treeMap of that file find last key <= key:

read at Max 64B

if key found: return

O(1)

}

return -1

$$TC = \boxed{O(\log N) + \left(\frac{\#\text{levels}}{25}\right) * O(\log N) + 6}$$

$$= O(\log N) + 25 * O(\log N)$$

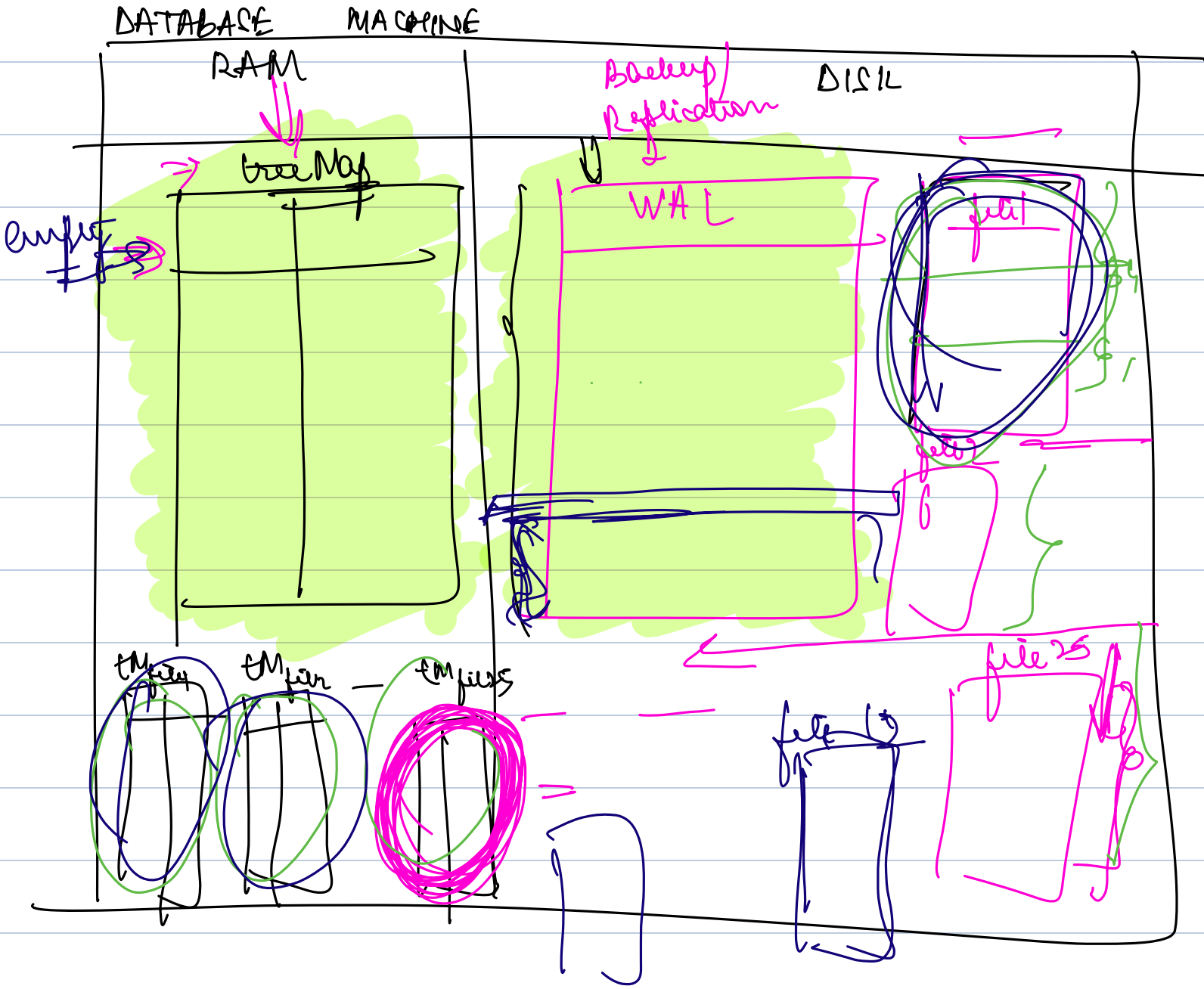
$$= \cancel{26} * O(\log N)$$

$$\Rightarrow \boxed{O(\log N)}$$

Log | Sorted Strings | Merging | Trees

$\Rightarrow$ **LSM Trees**

way to store data in NoSQL DB




```
write (key, value) {
    vec.append(key, value)   ] O(1)

    treeMap.put(key, val)    ] O(log N)

}
```

read (key) {

O(logN) if (treeMap. contains (key))
return treeMap.get (key)

#files for file from latest to oldest file:
O(log(N)) using tm of that file find block where key can be there.

O(64)
read that block
if key present:
return.

return -1

}

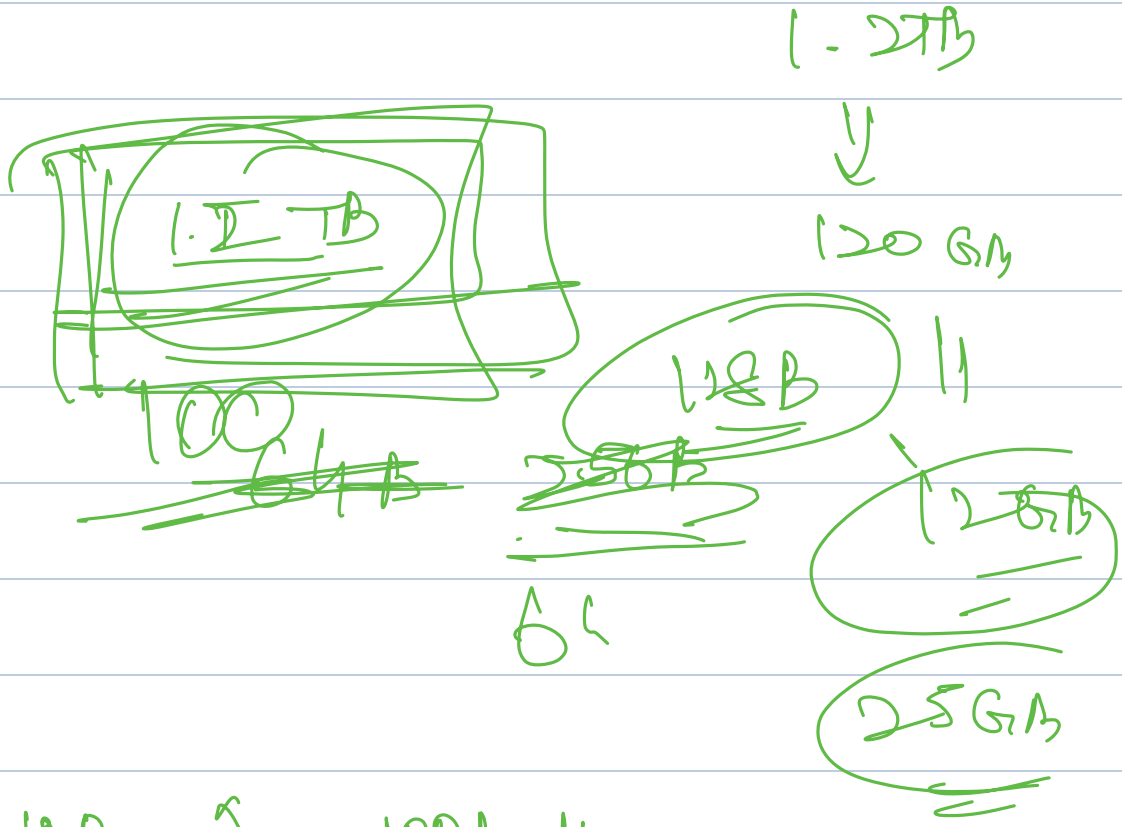
→ O(logN) + (#files) * O(logN)

→ O(logN)

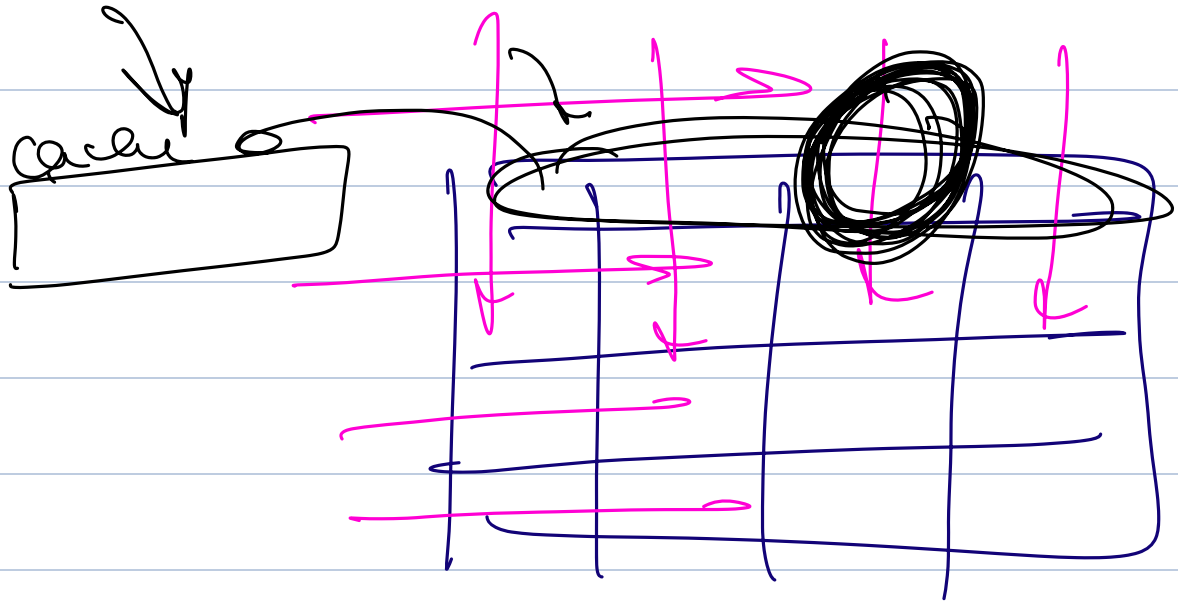
25

2 BG Processes

1. every hr: convert TM into new file
2. every day: merge all existing files into 1



12B < 100B keys


cache