NoSQL Internals

→ SSTables
→ LSM Trees

Structures that are used by most of NoSQL Databases.

B Trees / B+ Trees

Optimize SQL Queries

No class next Saturday
10th Feb

Creating own SQL DB is still easier

↳ Structure

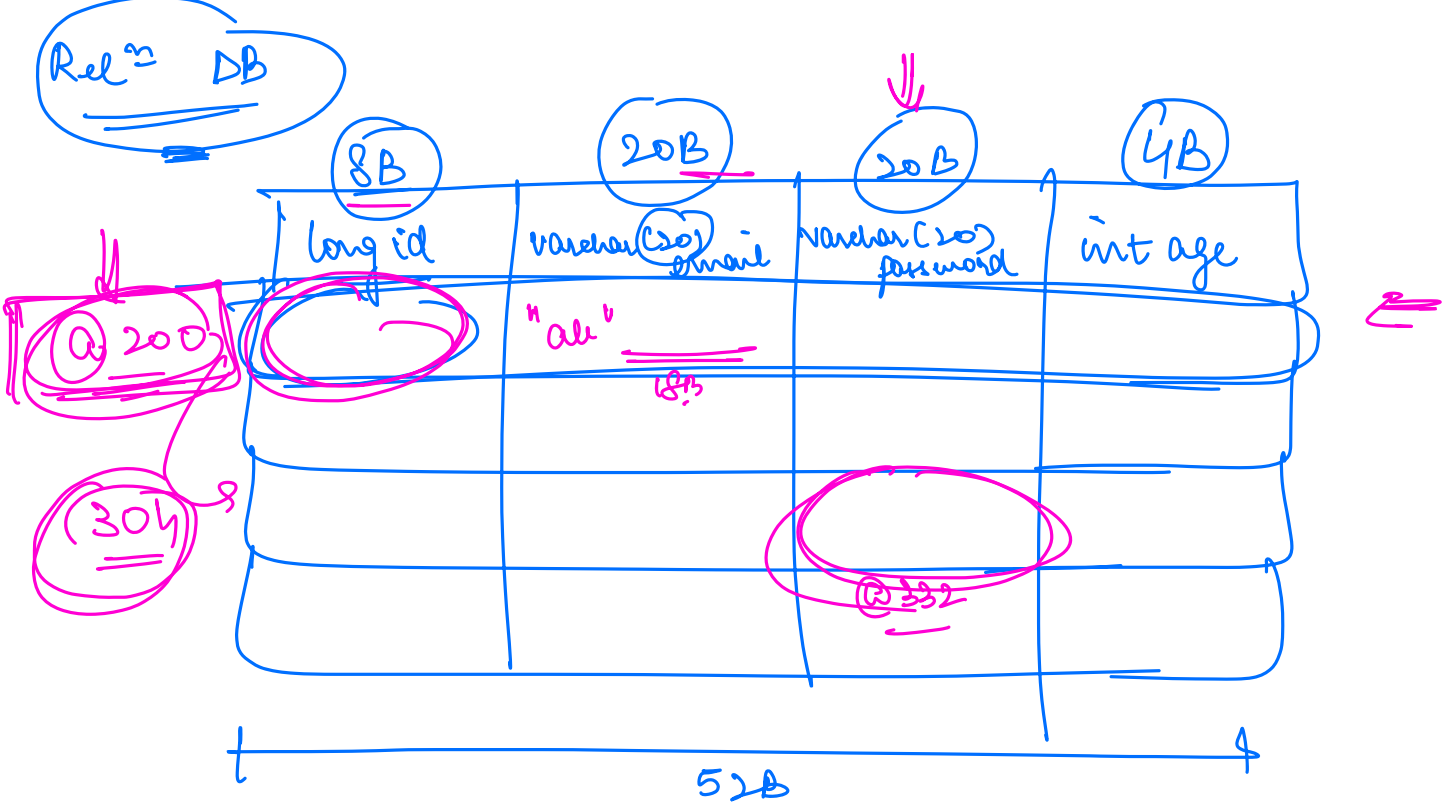
↳ every col^m has a fixed data type.

```
int[] a = new int[] { 1, 6, 3, 5 }
```

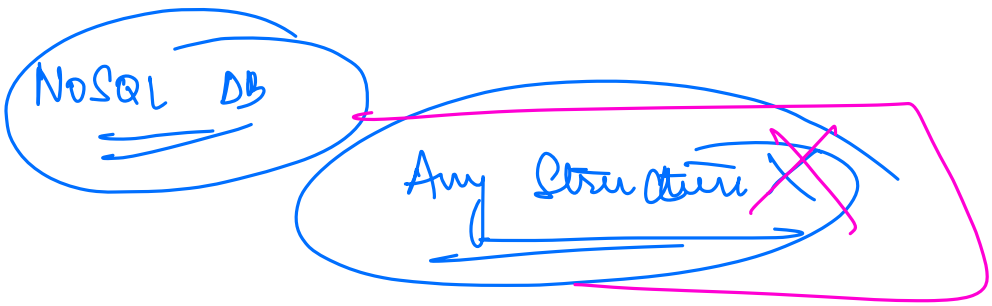
@200 → @204, @208

= Binary Search on an array

↳ ~~LinkedList~~

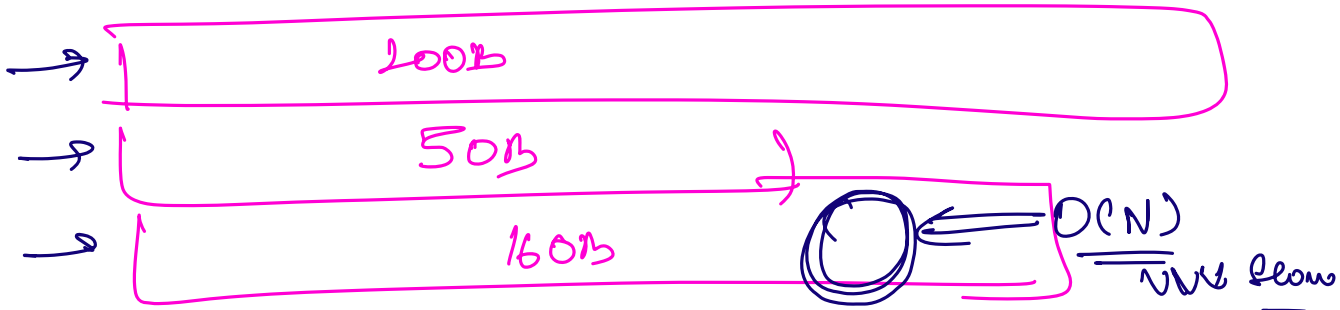


52*2 → 104 + 28 → 132B



↳ → How will I store data internally

→ How will I query data



⇒ As every entity/ attribute can have its own sizi

⇒ isn't possible for me to find value of x attr of iter entity in O(1)

"TOPIC — how is india" ⇒ [, ,]

K-V, Document DB

⇒ O(N)

⇒ VVVV Bad

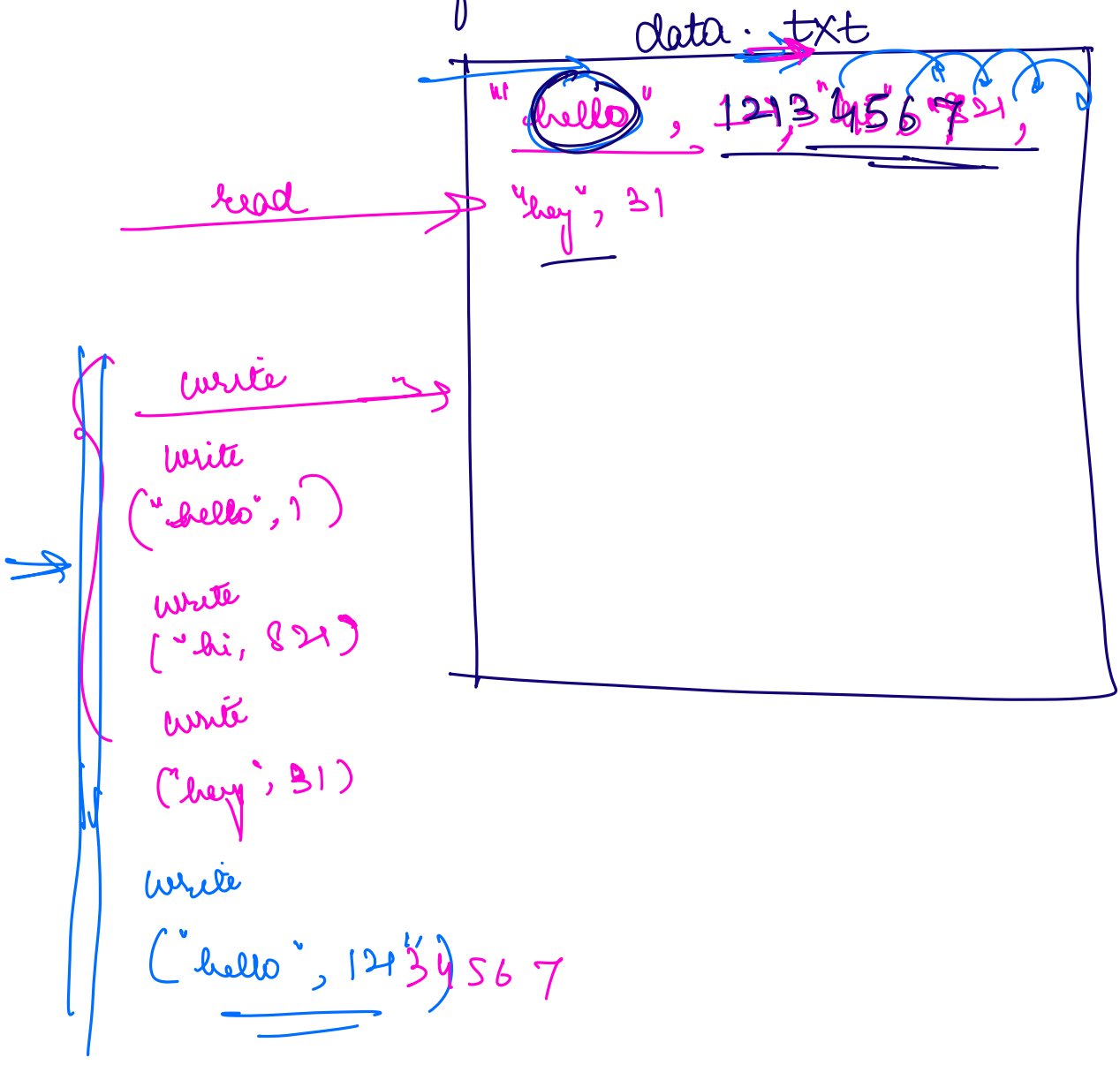
How to store data in NoSQL DB

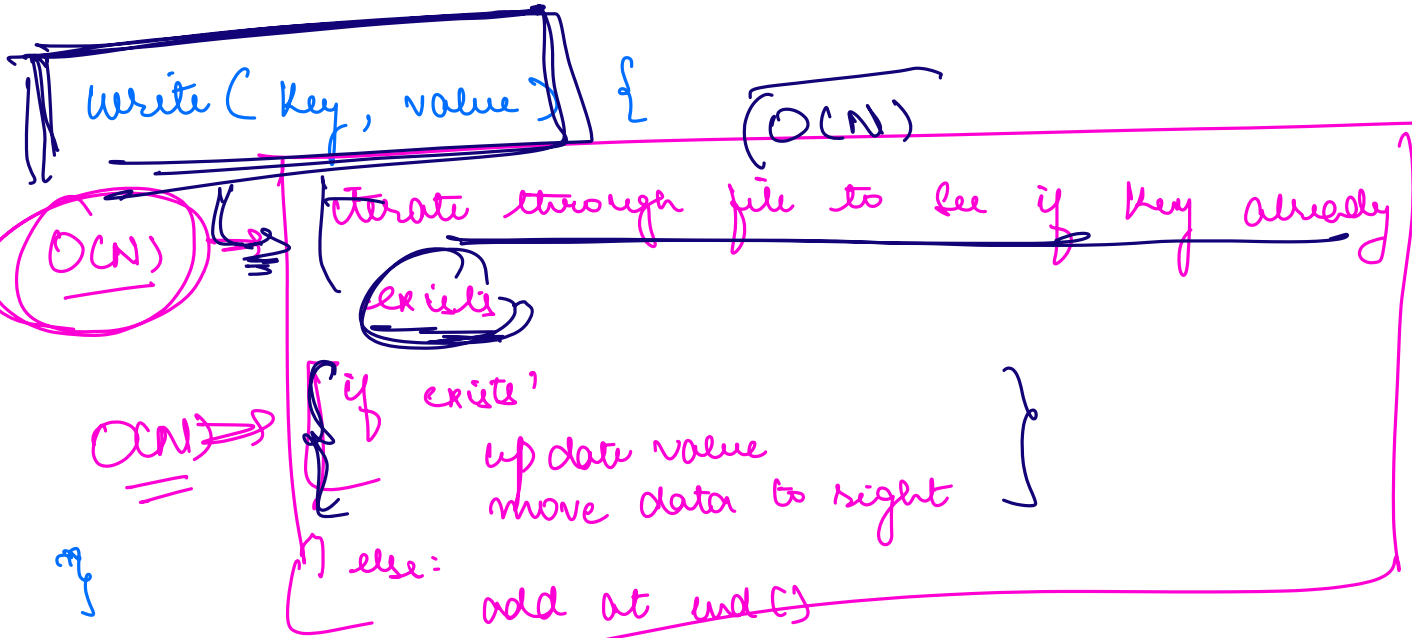
On disk

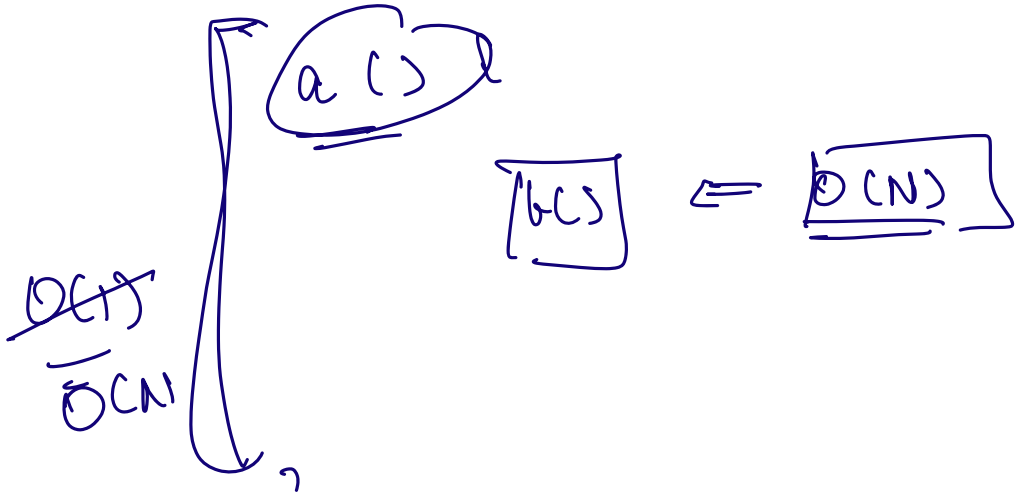
Brute Force

"\n"

⇒ Create a file to store all data

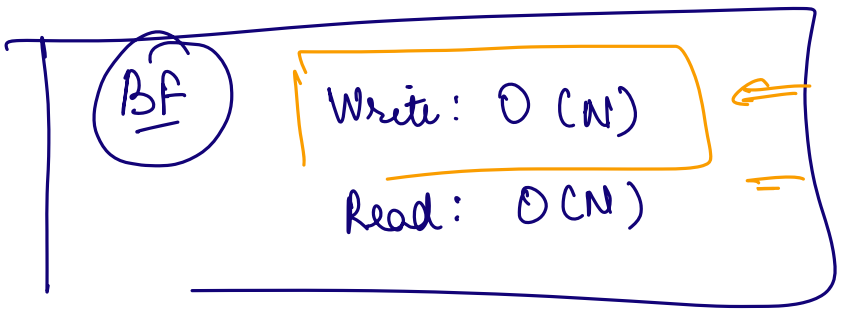






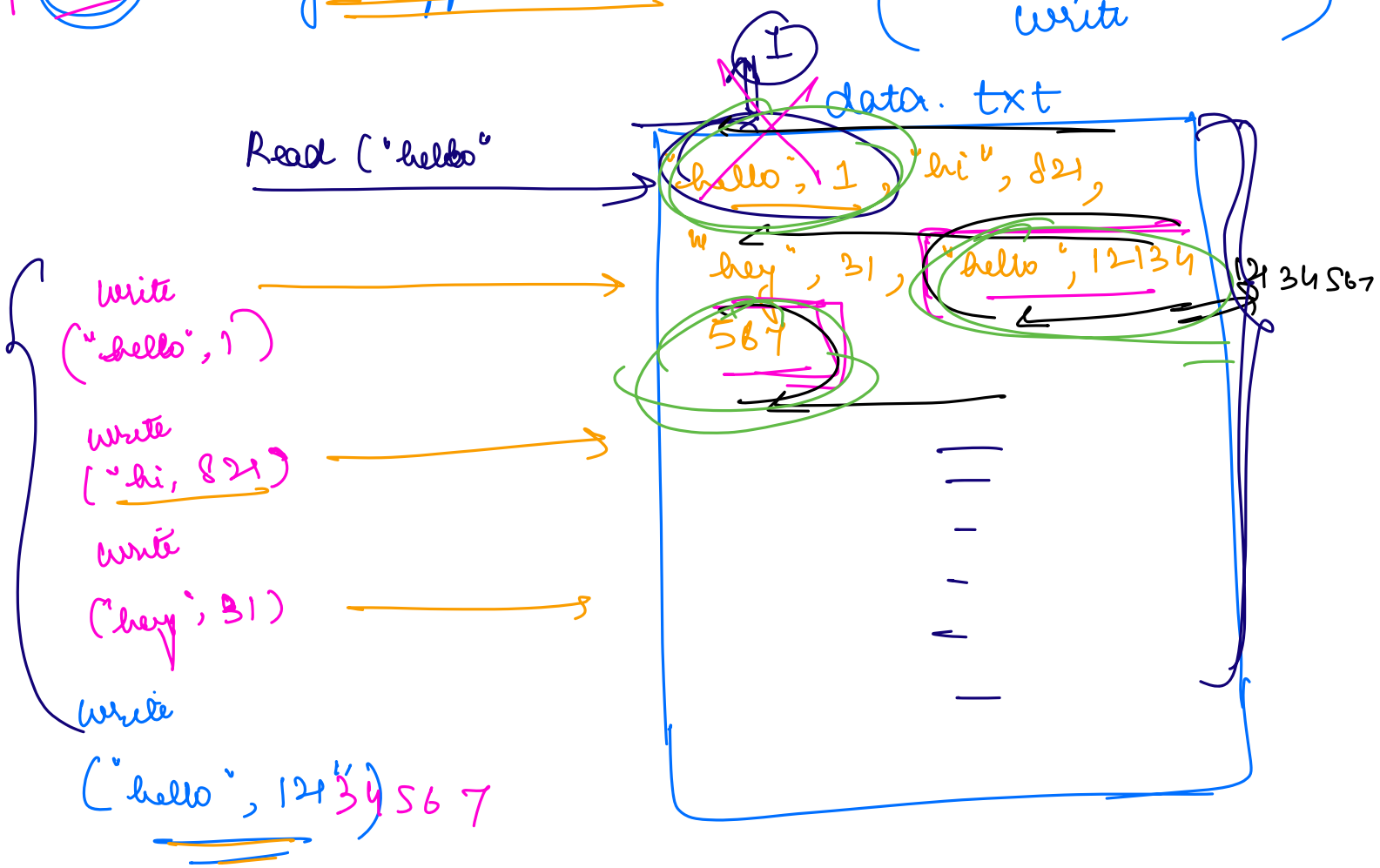
read(key)
iterate through file
find key
return if exists

O(N)



II

Only append on write (no update on write)



Now, on write, we are just appending to file and not checking if key already exist

TC

Write ⇒ O(1)

Read ⇒ O(N) in worst case

write (key, value)

append to file

⇒ O(1)

⇒ read (key)

start reading file from end

to find key

if exists:

return

else: error

O(N)

Problems

① ⇒ O(N) is still bad — VVV slow

② A lot of duplicates ⇒ ?

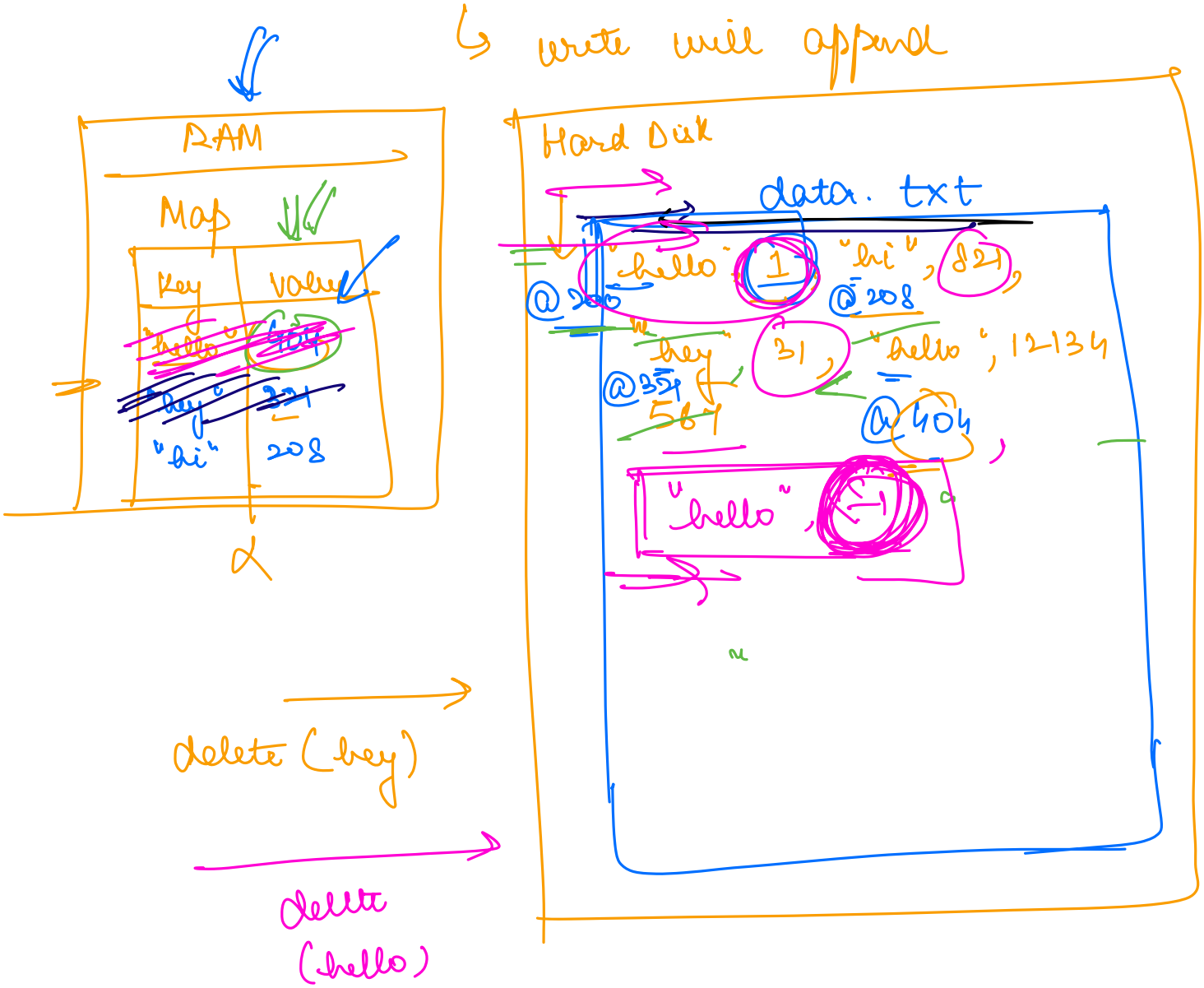
⇒ Wasted Storage

$O(N^2 + N) \Rightarrow O(N^2)$

V3 Also try to optimize reads somewhat

→ In disk we have same thing as in V2

↳ write will append



delete (bey)

delete (hello)

→ In memory, I create a Map

O(1) ⇒ write (key, value)

append to file → O(1)

update hm with the latest → O(1)
address of that key

read (key) ⇒ O(1)

if hm.contains(key) ⇒ O(1)

go to address

return value

else: error

Write ⇒ O(1)

Read ⇒ O(1)

What if computer restarts?

- ⇒ db will go through the file
- ⇒ rebuild the map
- ⇒ db will start

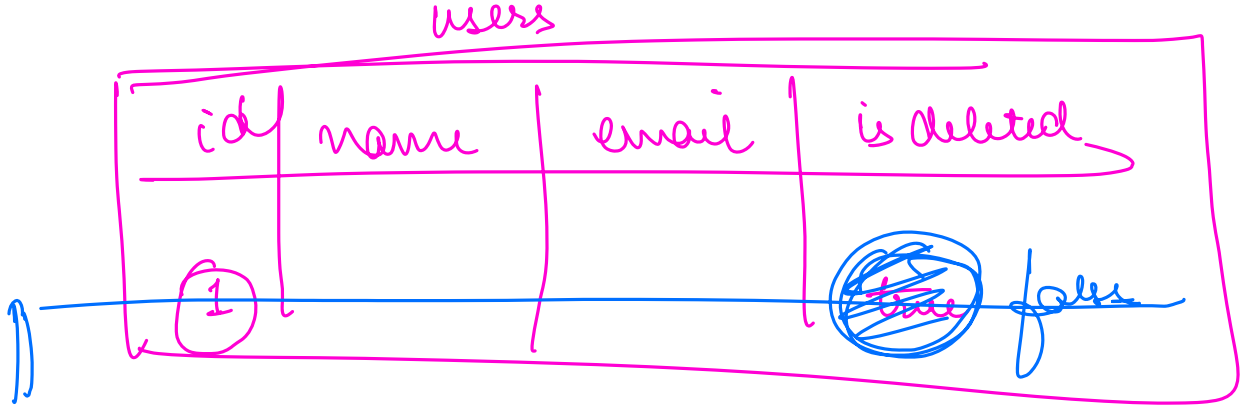
How to handle delete ?

Sol^{n} 1: just delete from HM

⇒ But that will not work because when computer restarts, while rebuilding HM, it will find key in file and bring it back.

Sol^{n} 2 ⇒ delete from file

⇒ will have to iterate over complete file ⇒ O(N)



Tombstone Method

lol ☺ Tombstone

O(1)

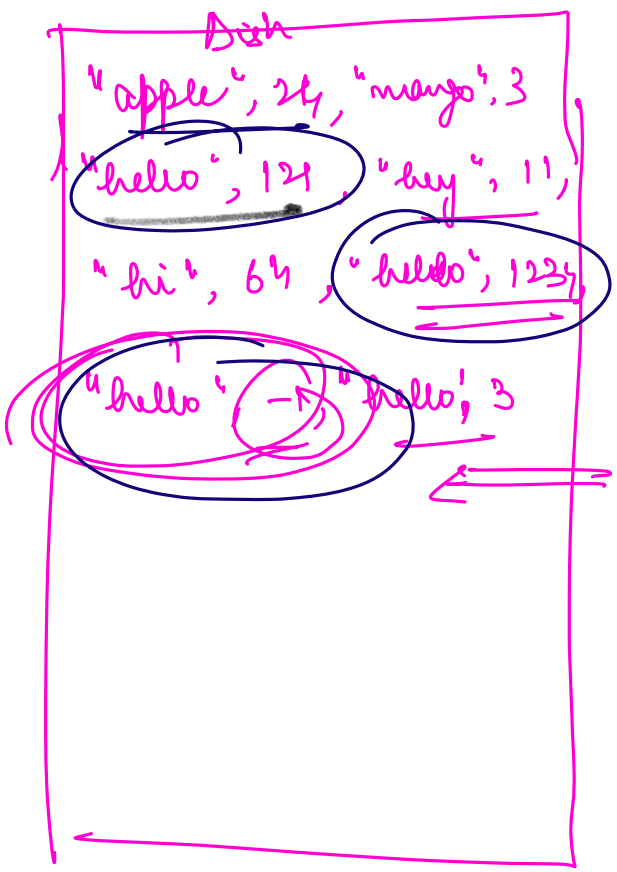
delete (key) {

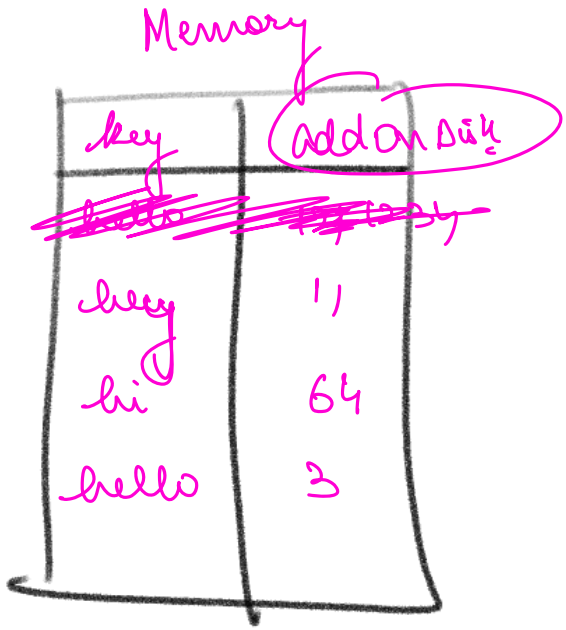
remove key from hm ⇒ O(1)

add tombstone (key, -1) at NULL

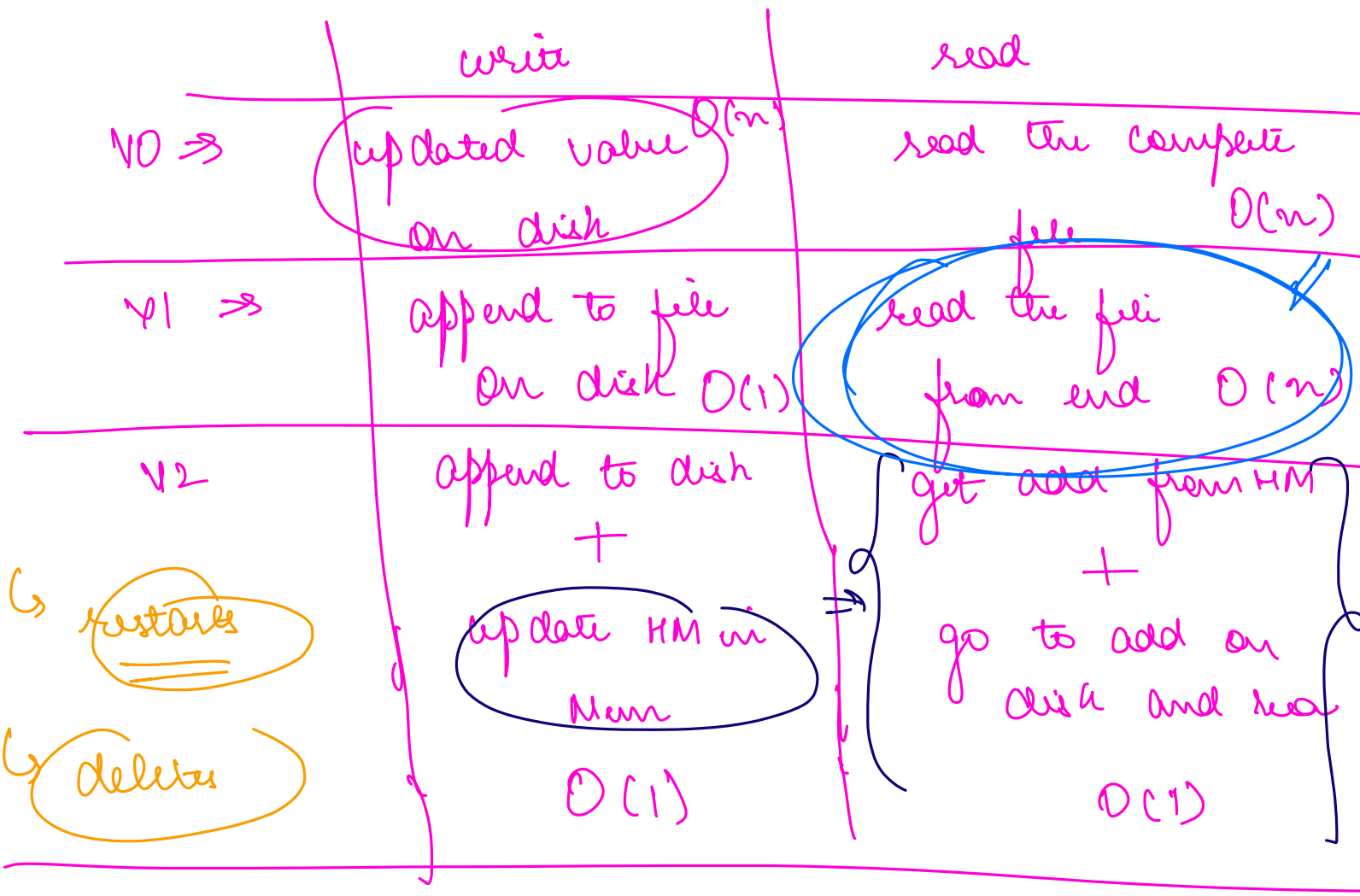
the end of fcn ⇒ O(1)

}





hm.put(k, v)

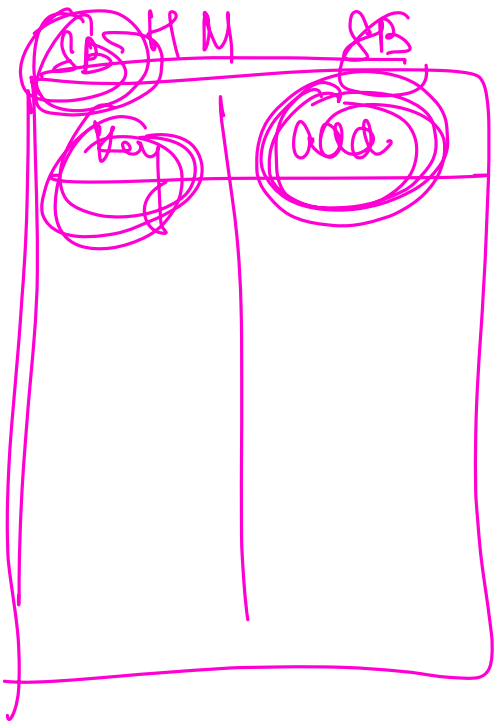


The problem still remaining :

⇒ a lot extra disk storage is being used.

1 TB disk ⇒ every entry on avg takes 50B to store

⇒ # keys = $\frac{10^3 \times 10^9 \text{ B}}{50}$



$\Rightarrow 10^2 \times 10^{10}$ B

50%

$\Rightarrow 20 \times 10^9$ keys

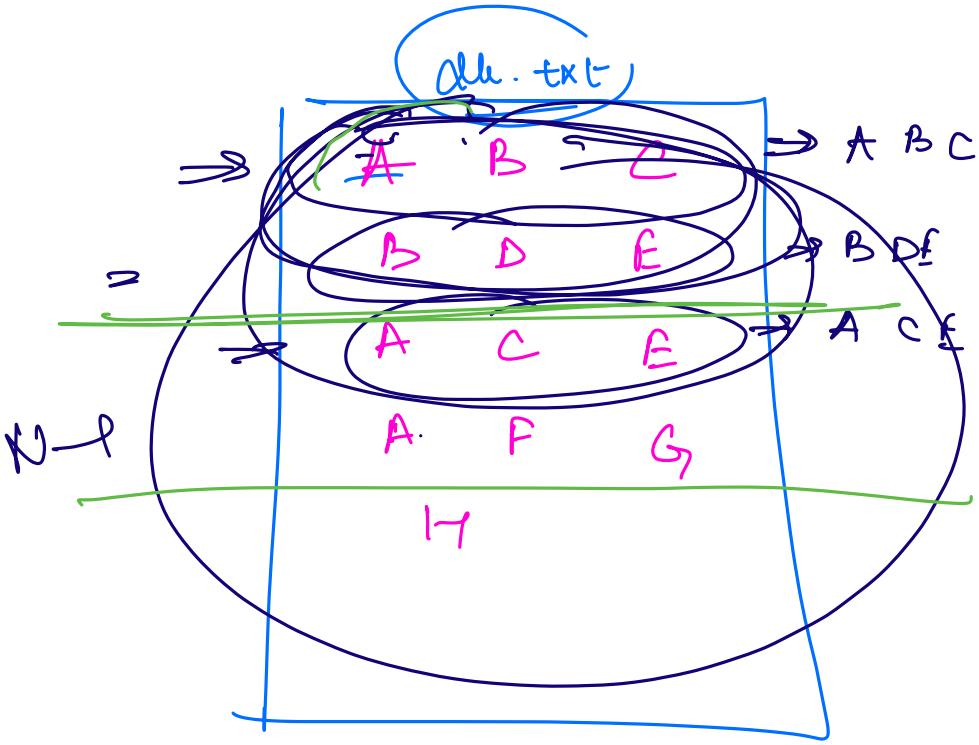
$\Rightarrow 16 \times 20 \times 10^9$ B

$\Rightarrow$ 320 GB

- Our RAM might not be able to store the value of all keys

Problems

1. Too much storage $\Rightarrow$ duplicate
2. RAM won't be able to fit complete HM



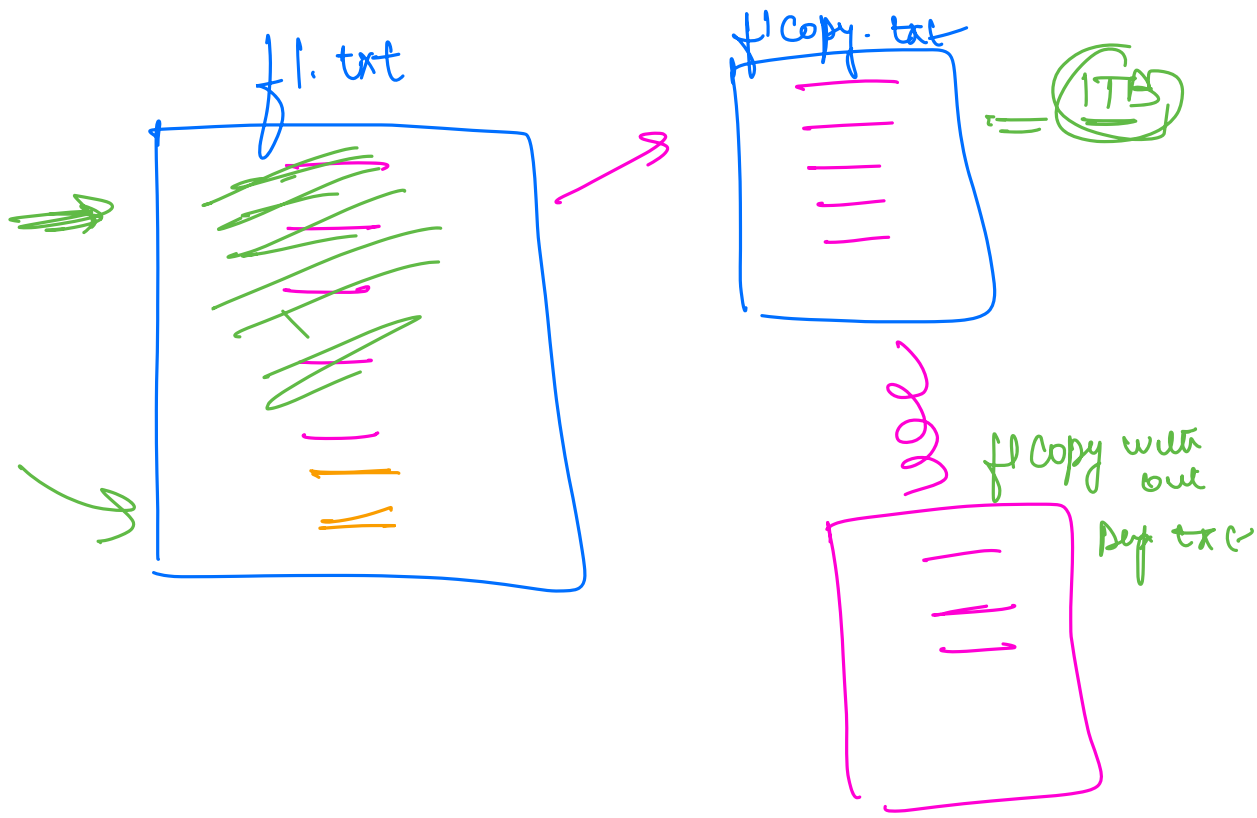
How to remove duplicates

Sol^{n} 1 : at the time of insertion → writes will be slow

Sol^{n} 2 : at restart

Sol^{n} 3 : A background job running every X hrs.

(i) create a copy of file
(ii) iterate over copy, find duplicates
(iii) create a new version with all dup removed



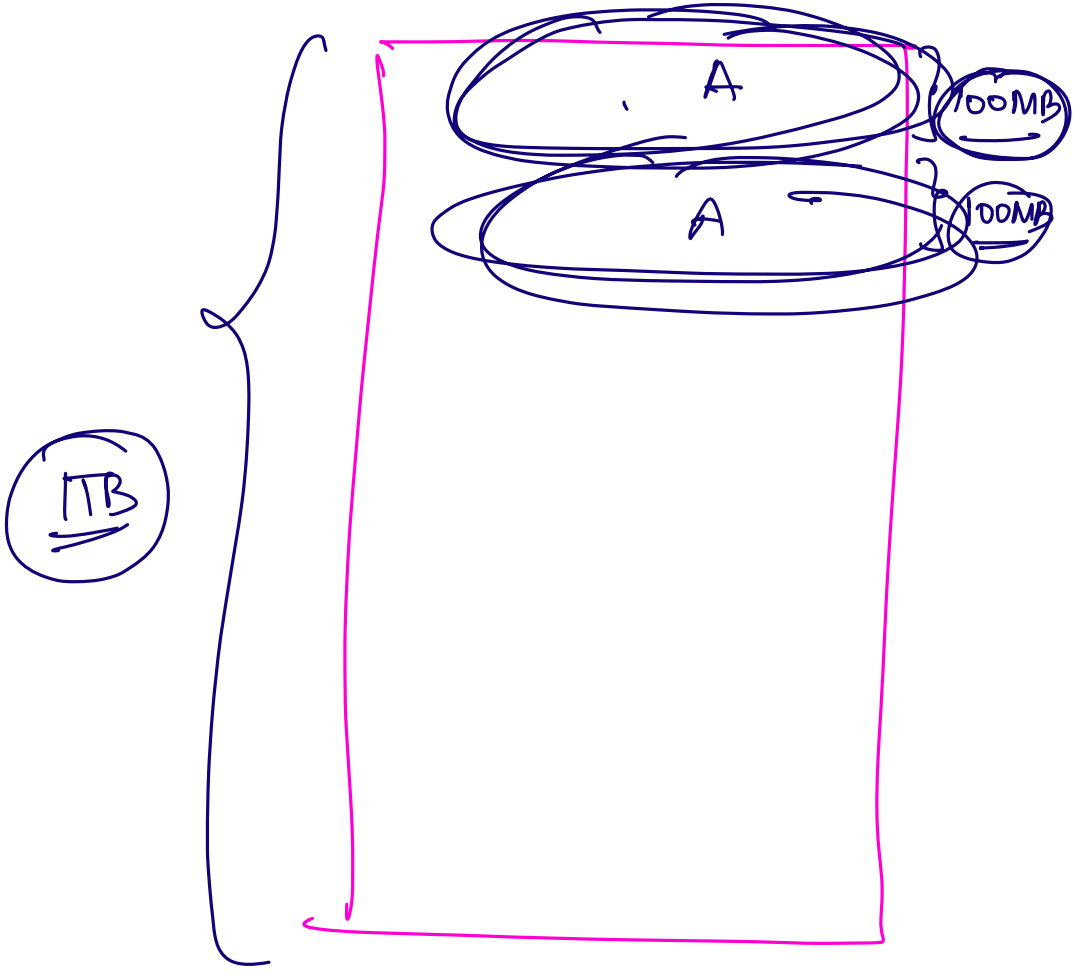
⇒ Implementation is very troublesome.

removeDuplicates(List<String>) {

HSet

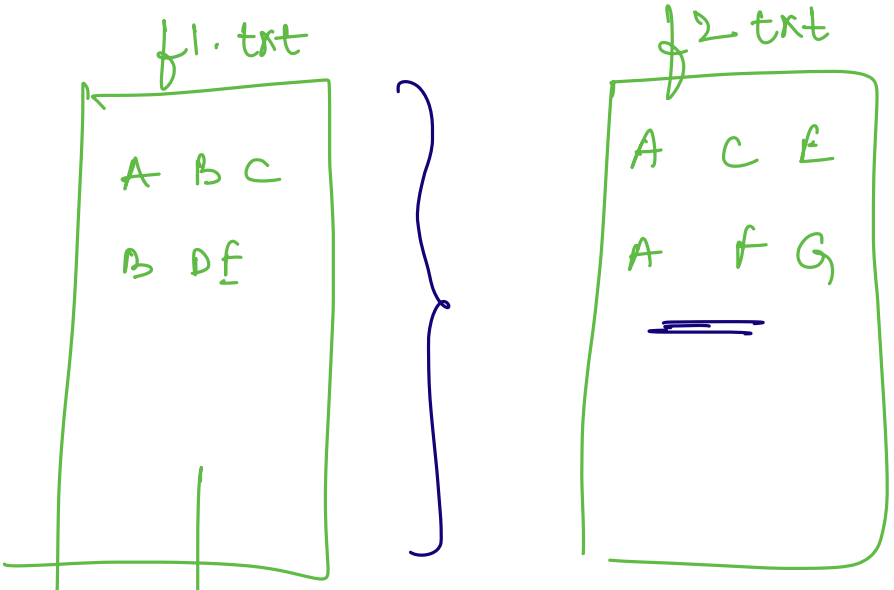
}

⇒ for a 1TB file putting all keys in a HashSet is not doable.

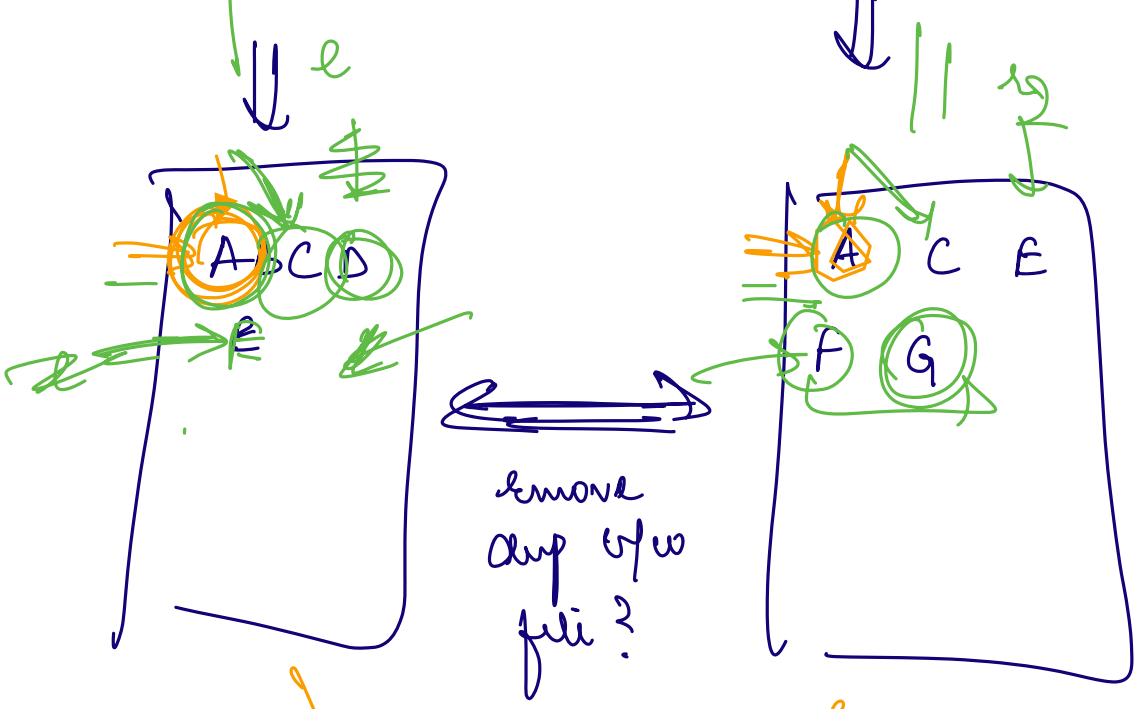


Soln

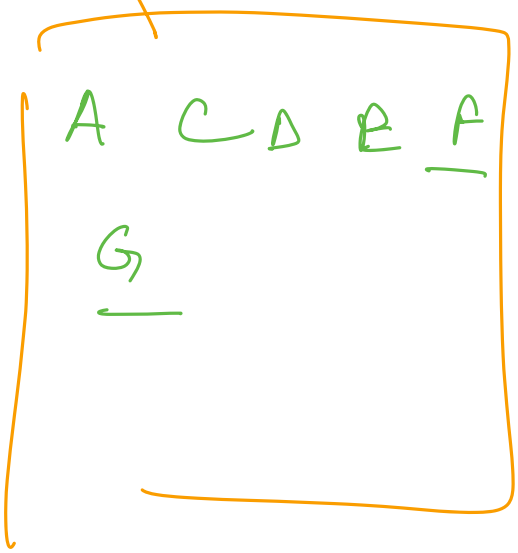
⇒ Split the data amongst multiple files



each file is small enough to be brought into

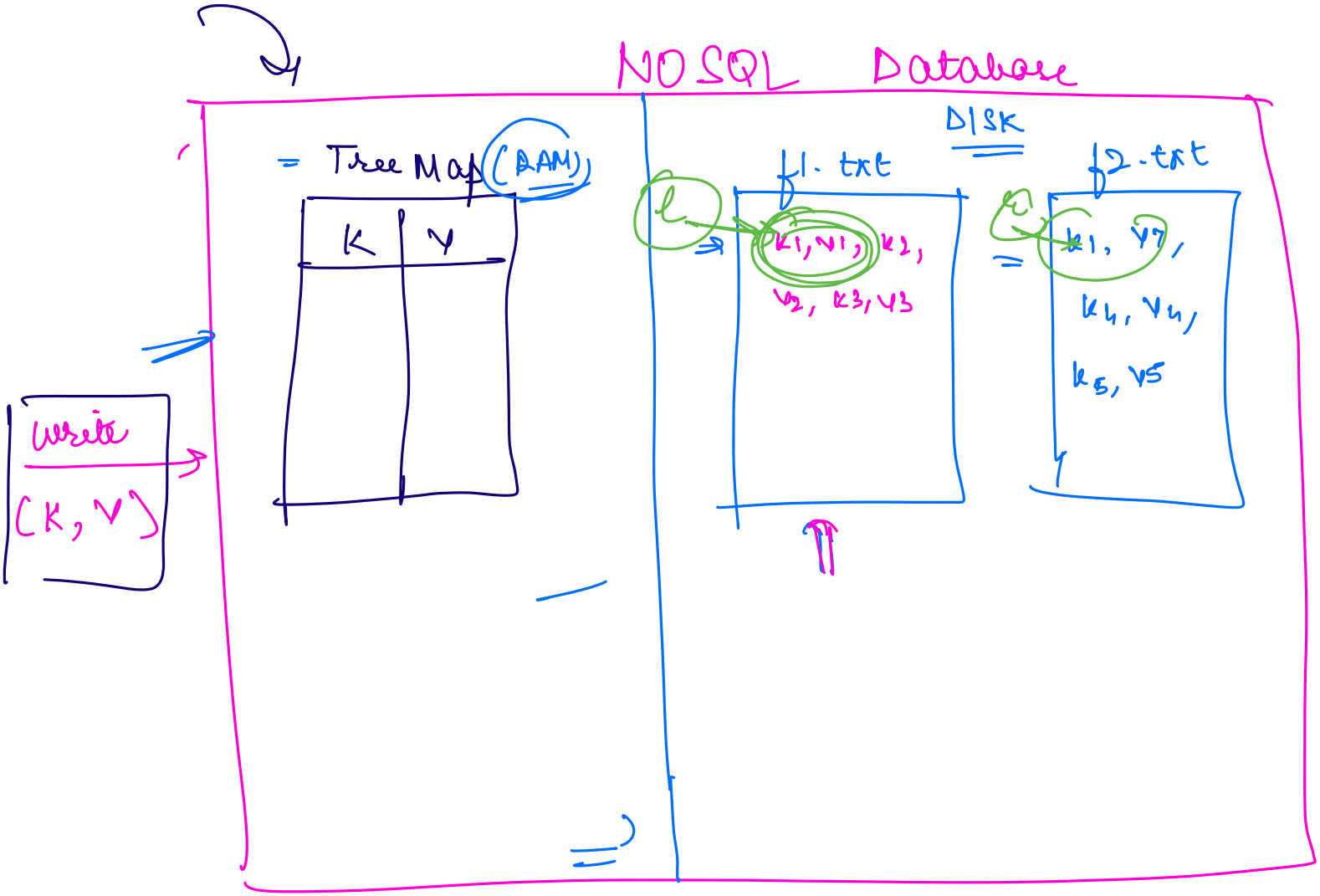


is there a technique using which I dont bring the whole file into RAM but still I can find dup b/w L



⇒ If somehow I am able to get my data into multiple file, and each file is sorted, I can

remove dups with ease




```
write (K, V) {
    treemap.put (K, V);
}
```

⇒ O (log N)

```
jobRunning Every X Mins () {
    create a copy of treeMap
    clear current treeMap.
    tc = treeMap
```

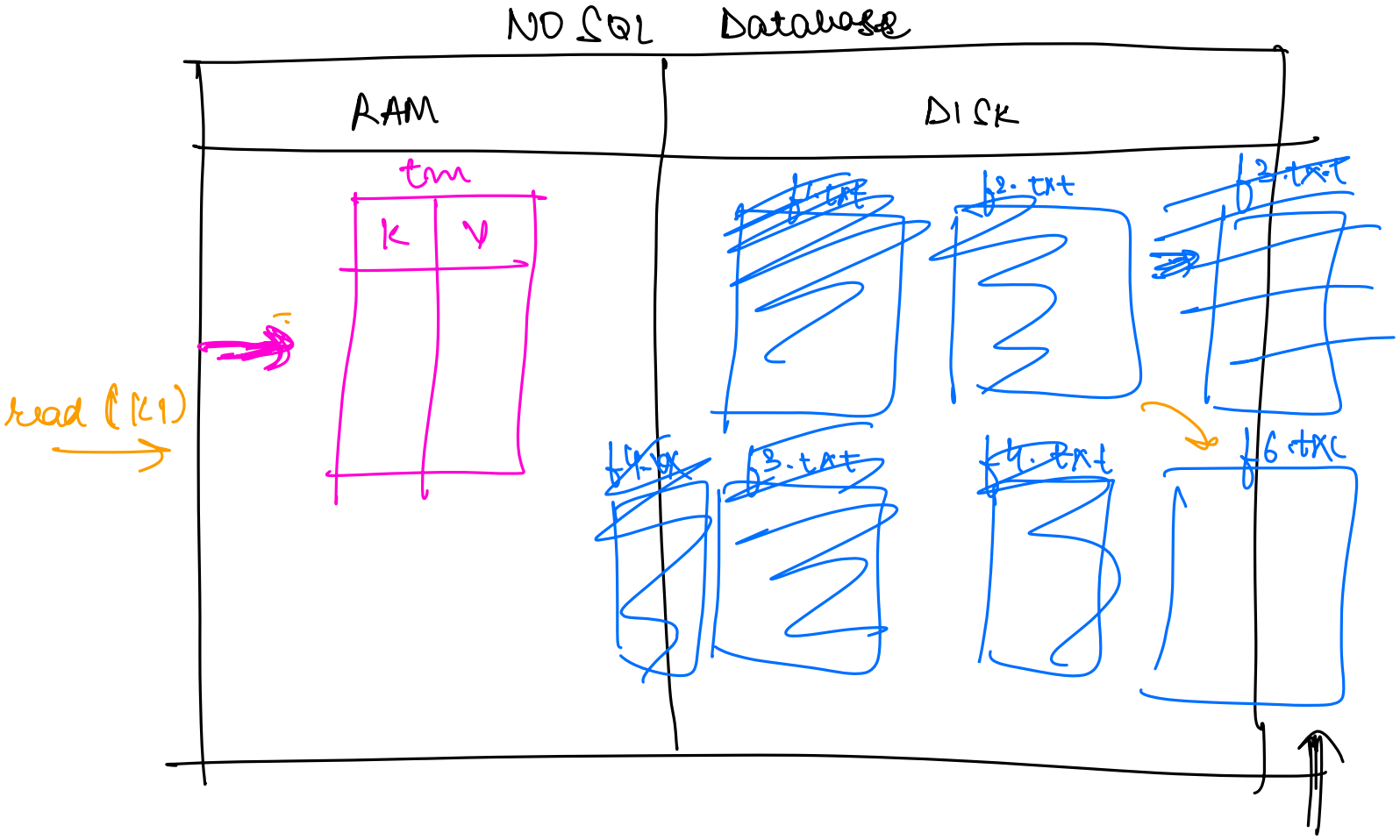
within a Map no dups are possible

treeMap = new TreeMap<>();

Iterate through copy of tc:
put the k, v on a new file

}

① within a file ⇒ no dups
② each file will be sorted.
⇒ if dups are now there on my disk
they are b/w 2 files.
⇒ As both files are sorted I can remove dup without needing to bring everything in mem.




```
write (K, V) {
    treemap.put(K, V)
}
```

VVV fast

```
convertTreeMapToNewFile() {
```

Create a copy of tm and empty current tm.
create a new file:
iterate over the copy of treeMap:
put key, value on file.

} → within each file, no dups and sorted data

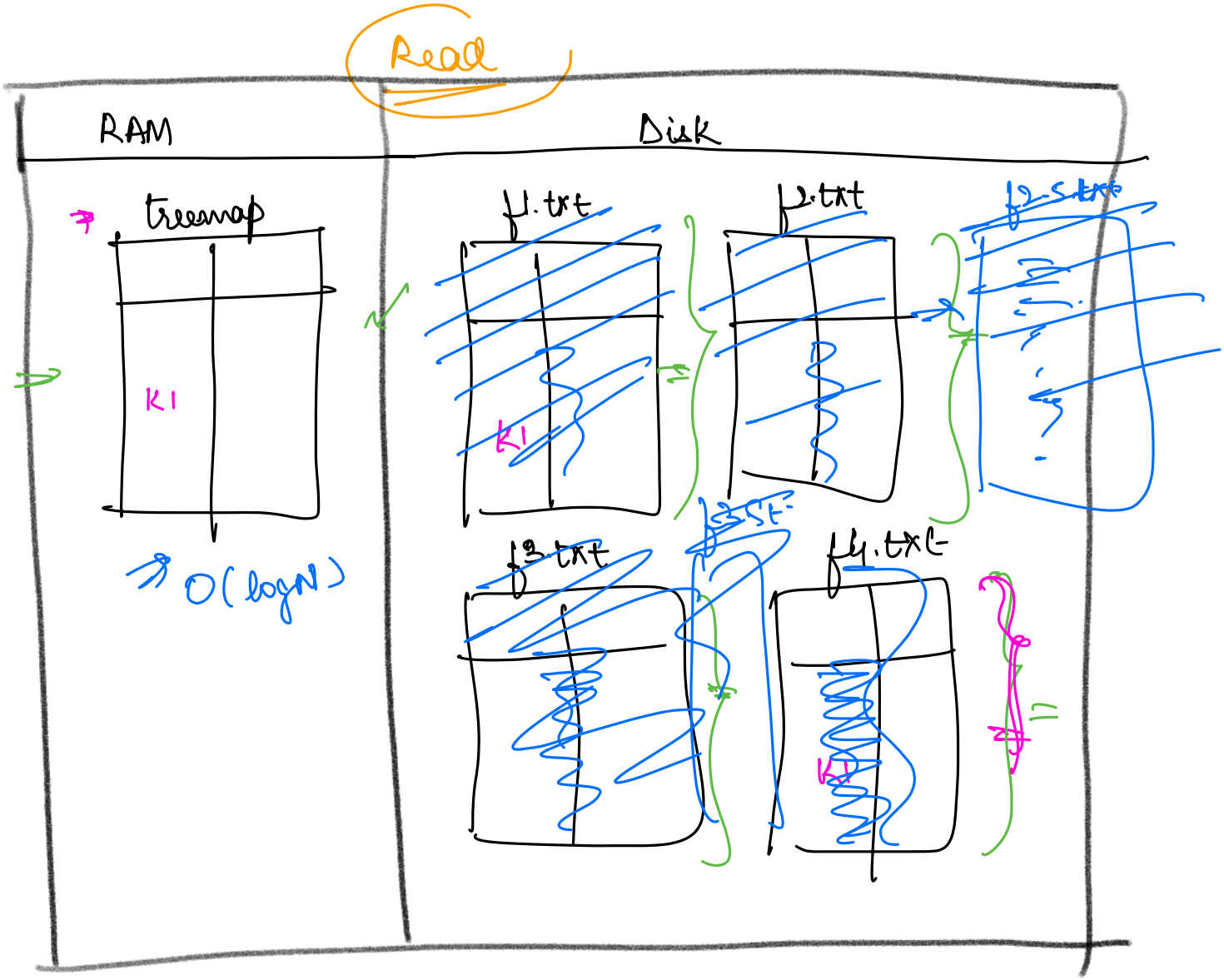
```
TreeMap<> copy = treemap;
treemap = new TreeMap<>();
```

```
mergeFiles() {   // also called compaction

    while there is more than file:
        pick 2 files
        merge them

}  ⇒ at end 1 file with no dup.
```

BUT!!! How will I read data???




```
read (key) {
    if (treemap.contains(key)) {
        return treeMap.get(key)
    }
    read each file from latest to oldest:
        if file contains key:
            return key
    return -1
}
```

// for unlikely key read will be very slow:

⇒ reading non existing key will be slowest

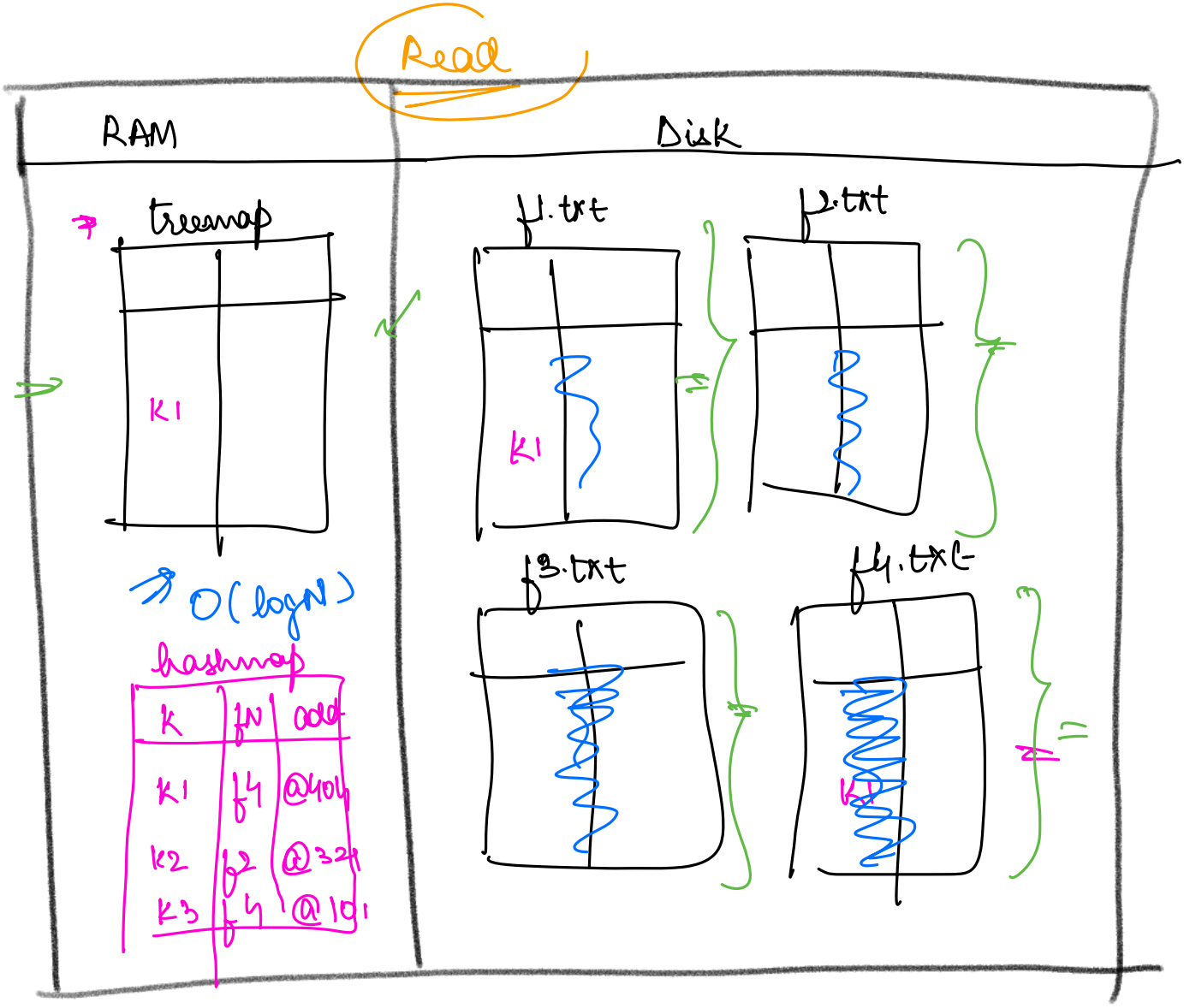
⇒ worst case I have to read all files.

⇒ PROBLEM: reads have now again become slow.

---

what if I again create a HM, which tells for every key, what file and writing that file at what addr that key exists

Slow Reads v/s I can't store HM in Mem.

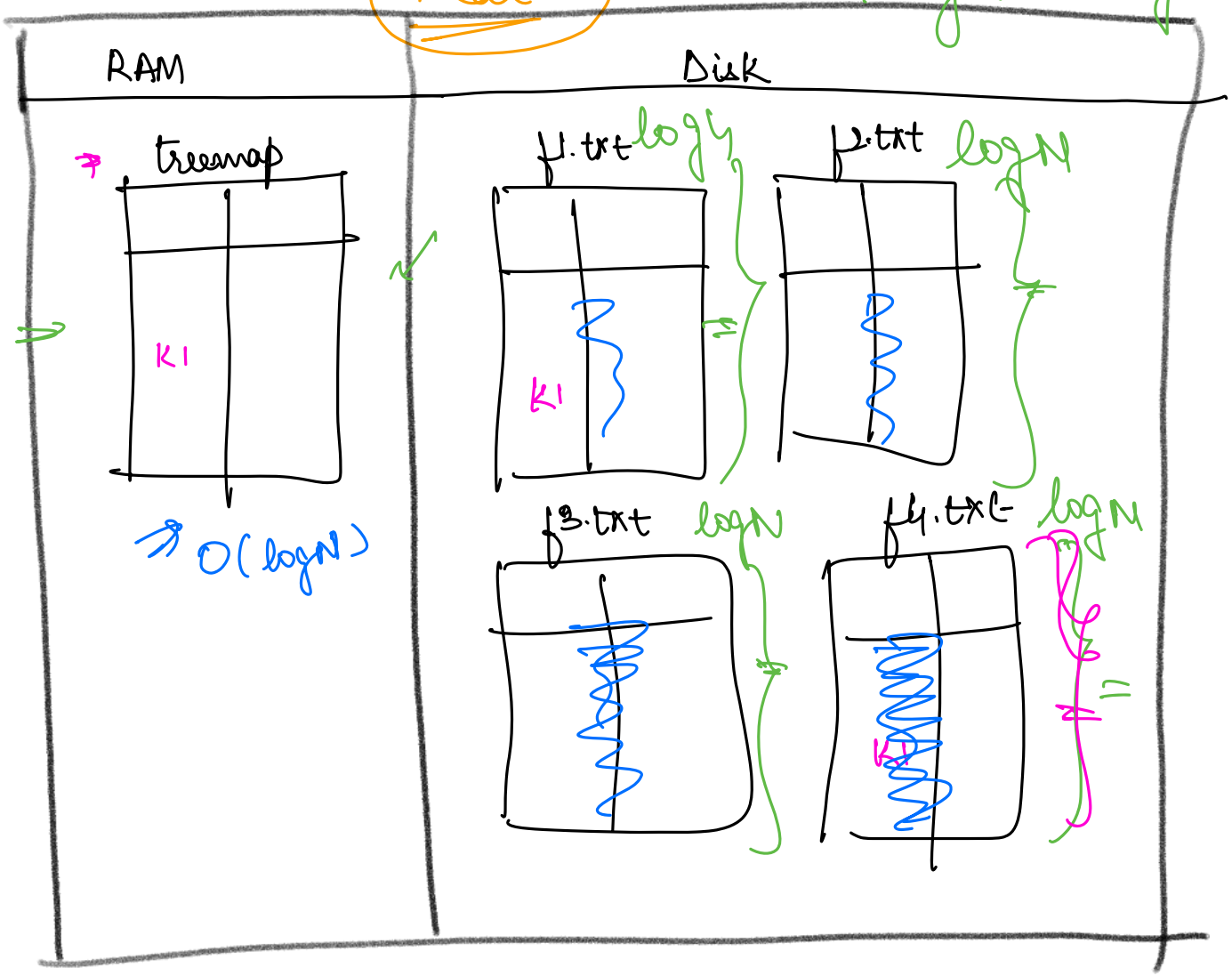


⇒ BUT!!! Data within each file is sorted.

Can I make use of this observation

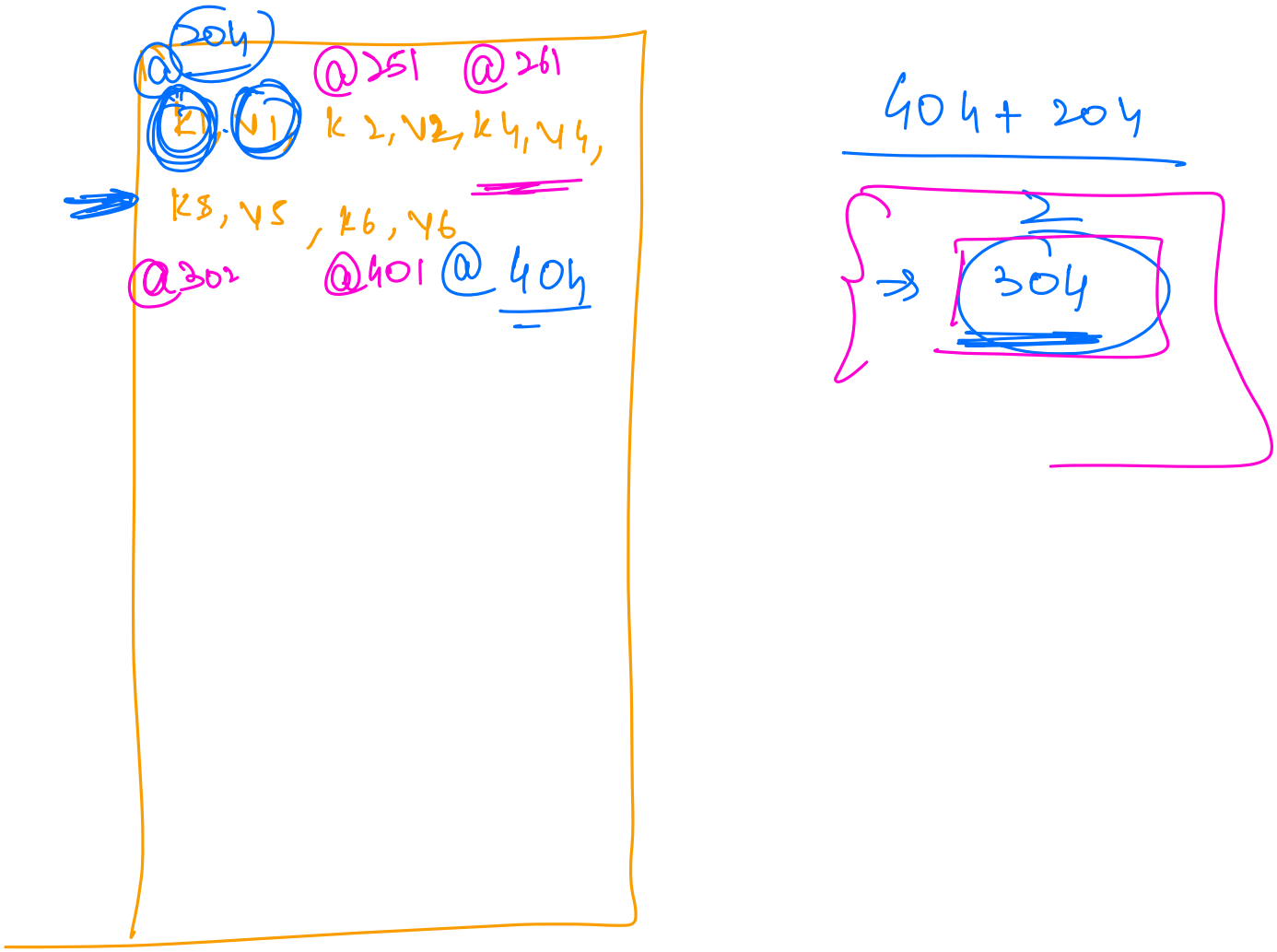
Read

$4 \log N \approx \log N$



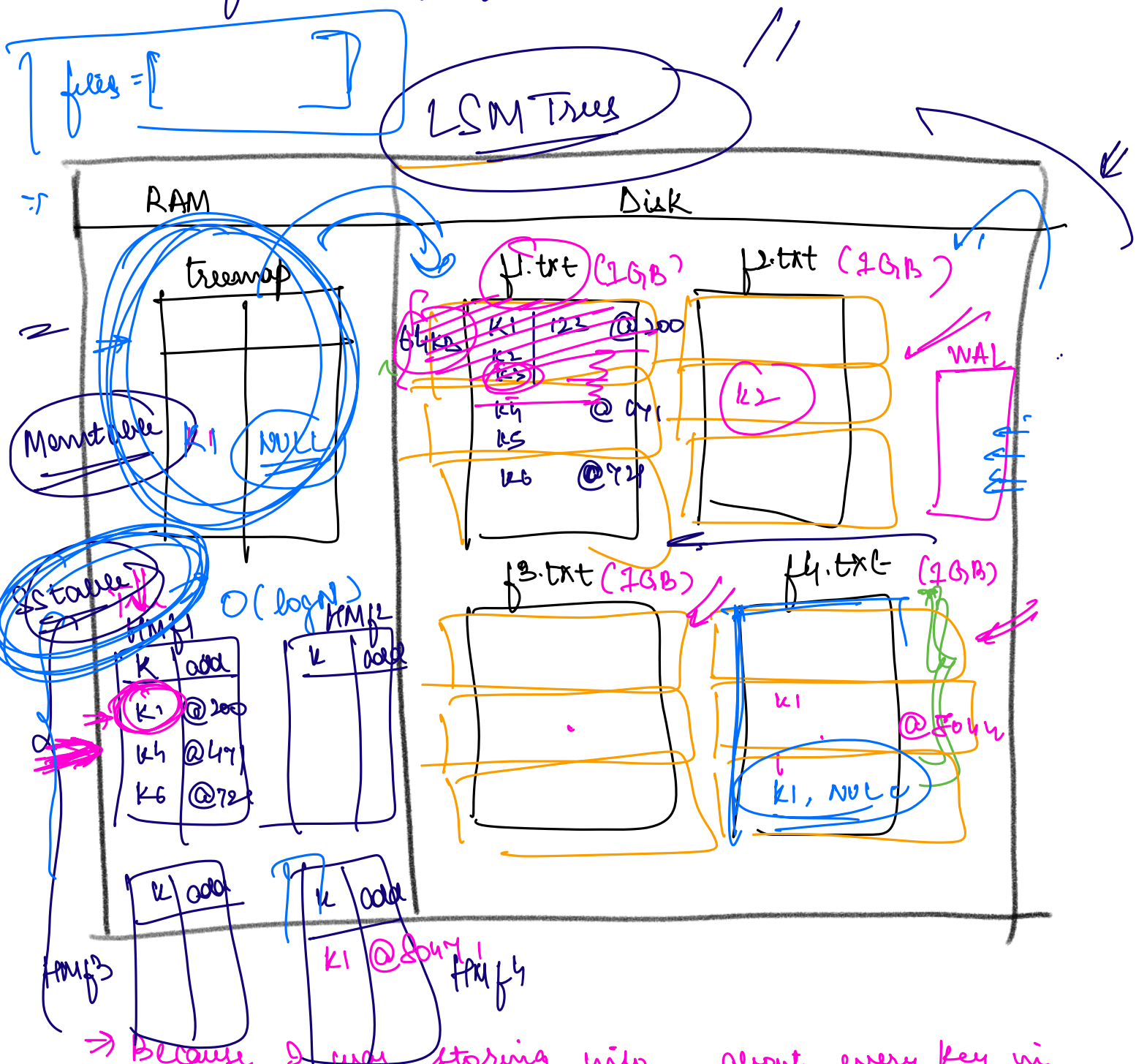

```
read (key) {
    if (treemap.contains(key)) {
        return treeMap.get(key)
    }
    read each file from latest to oldest:
        if file contains key: ⇒ find the key within file using BS
            return key
    return -1
}
```

→ O(N)



Only Problem: Find a key in logN ci fele

Soln ⇒ Create another additional HashMap for every file.



⇒ Because I was storing info about every key in HM, the HM was becoming huge.

to find a key in a file

(i) Binary Search

(ii) Linear Search ⇒ v. slow.

⇒ 1 GB

Divide file into multiple blocks:

(i) Create a HM associated to every file

(ii) In HM, don't store info about every key but some keys (first key of every block)

Now to find a key in a file:

eg find k3 in f1:

⇒ I will go to the bookmap of f1

⇒ I will the key just smaller than k3

O(log N)

⇒ I will just read that block

⇒ 64 KB

```
read (key) {
    if (treemap.contains(key)) {
        return treeMap.get(key);
    }
    for each file from latest to oldest:
        find first key smaller than k
            in TreeMap of that file:
        read that complete block into
        if k exists
            return
    return -1
}
```

Writes are fast b/c ⇒ put in "memtable"

reads are slower ⇒ you may have to read some part of multiple files.

But can we lose writes

Why: I am only writing to memtable (treemap in mem)

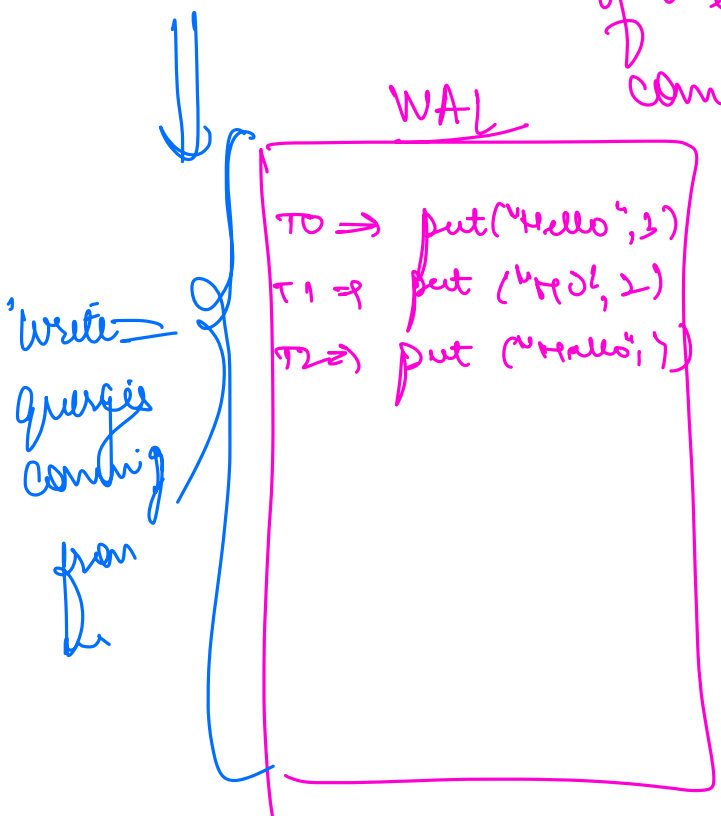
Ans: No

Every DB has 2 things that it stores.

(i) Data

(ii) Write Ahead Log (WAL) / Commit log / Bin log

a file where we keep a note of every write query that is coming to my system



⇒ file that has history of everything in DB.