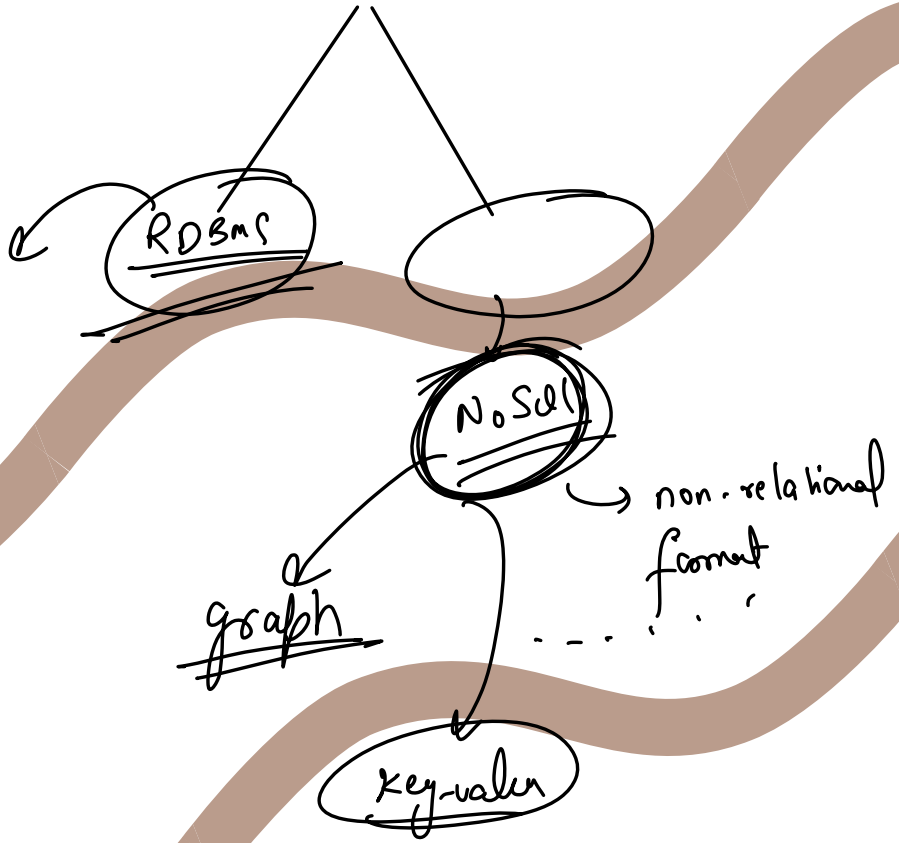

RDBMS
NoSQL
non-relational
format
graph
key-value

Let's Design a Database

→ Initial Requirements:

- → fast reads
- → fast writes (more imp)
- → durability
- → unstructured data
- → handle a few PB of data.

Let's understand how RDBMS works:

1) In RDBMS we have schema pre-defined

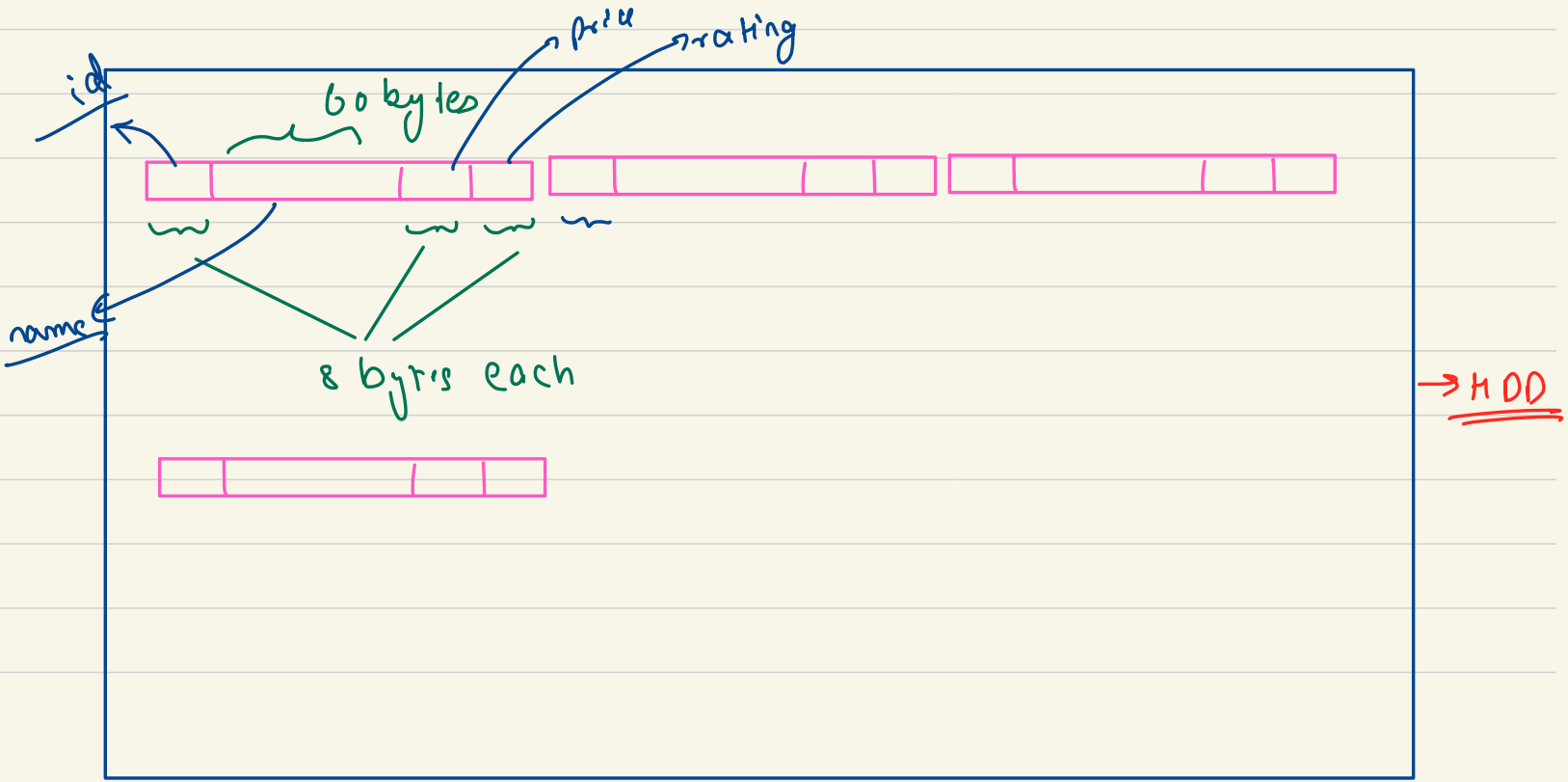
2) for every row we know the exact size & data types.

assume a product table

- id → 8 bytes (primary key)
- name → 60 bytes
- price → 8 bytes
- rating → 8 bytes

Total → 84 bytes

Now how will be this structured data stored on hard disk??



Q=> Now what will happen during the reads??

→ assuming data is ordered by id, because of fixed schema and defined size, the db knows exactly where the next record is.

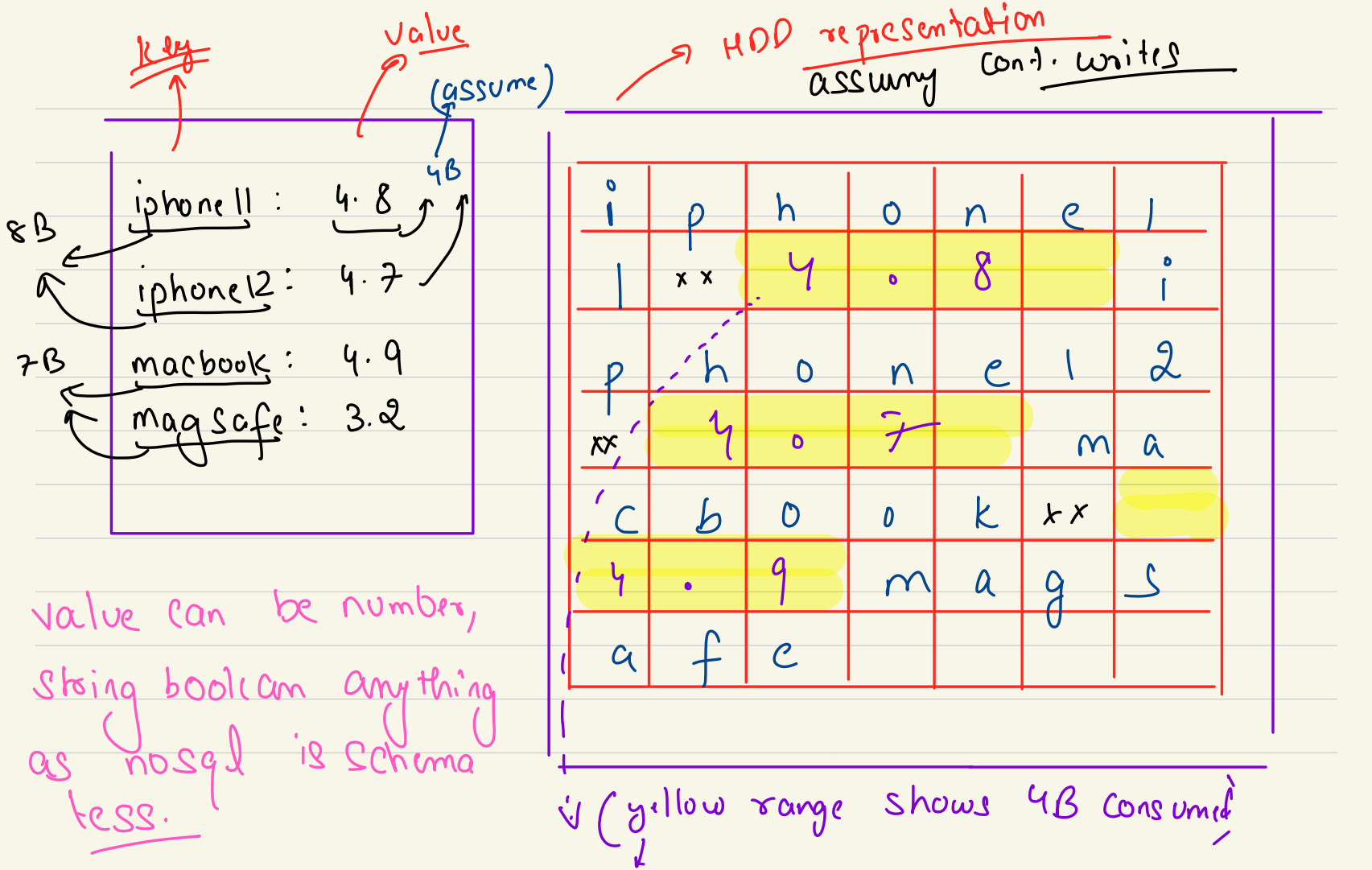
Why?

Because from starting byte of one row we can add 84 bytes & reach next row.

Even for update on a specific col of a row, we just need to go to specific loc & update.

↳ Because updates won't even change size of data hence allocated mem space is same.

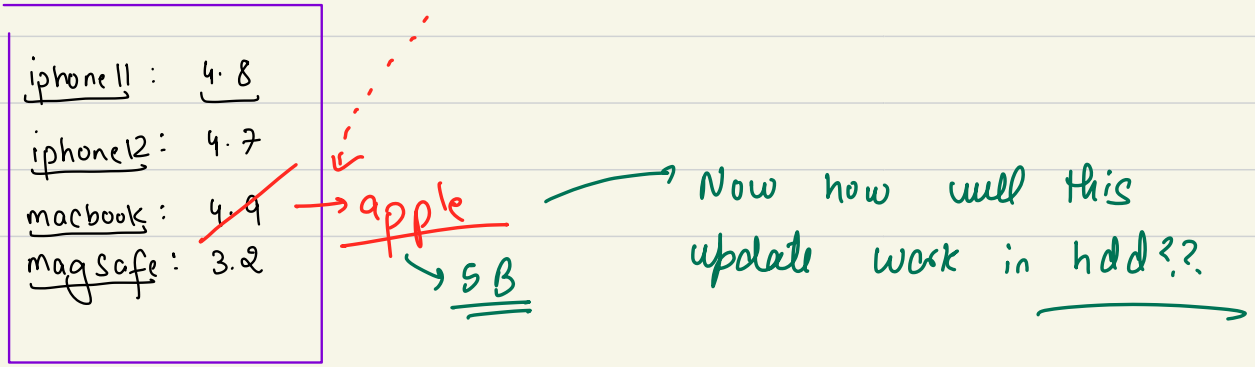
Q. Now can you try to think for a key value based no-sql how data orientation on hdd will look ??

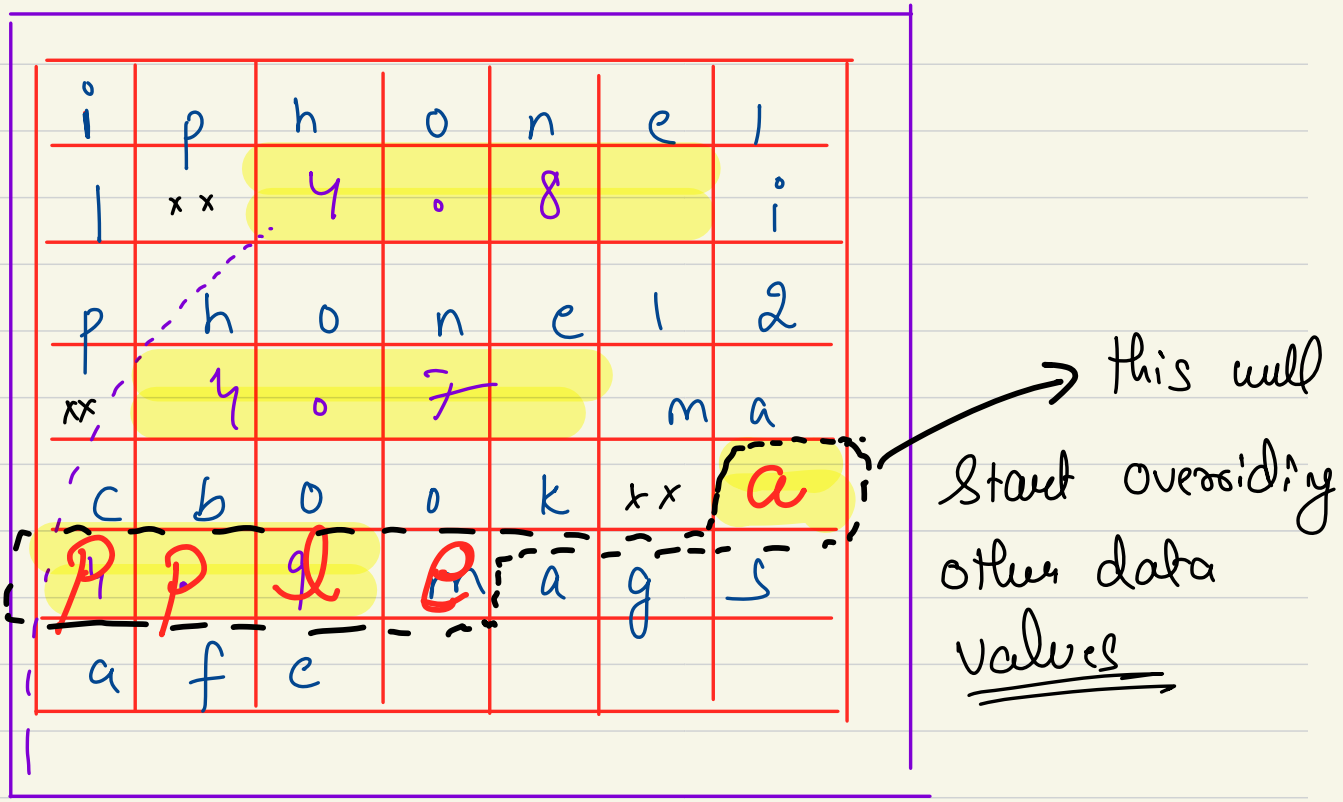

Key
Value
(assume)
HDD representation
assumy cont. writes
iphone11 : 4.8
iphone12 : 4.7
macbook : 4.9
magsafe : 3.2
8B
4B
7B
value can be number, string boolean anything as nosql is schema less.
(yellow range shows 4B consumed)

(for simplicity we assumed decimal no. taking 4B)

Now for a moment assume instead of a decimal no. we updated the value to become a string

Ex: set ("macbook") → "apple"




this will start overriding other data values

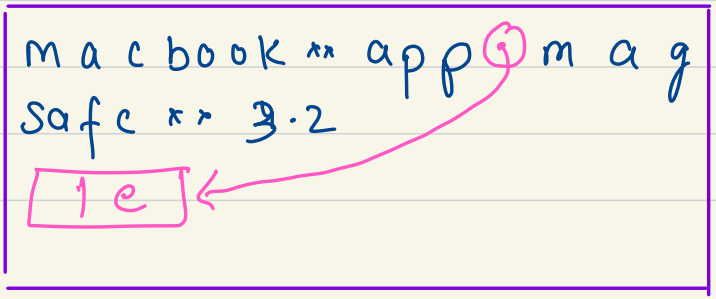
Q. So how the update will work?

Q. Also is this good for fast searches in our key value store?

HDD → optimized → sequential

→ What if when we update the data, then we keep writing & when no space is left then

1. Don't override existing data
2. Split the data & store remaining data some where else on the disk.
3. And maintain pointer for this splitted data.



→ Can lead to slow reads due to high fragmentation.

(Not good)

Any other ideas ??

What if for updating we just insert a new value.

→ 1) when update comes don't override

2) append new data in hdd

3) for reads, do linear search from the end

O(n) T.C. (Not fast)

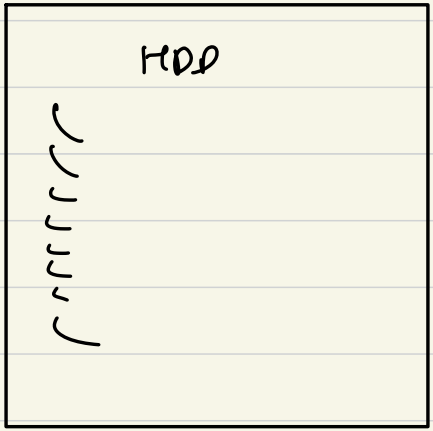
(Any optimizations on reads will be great)

How can we optimize this ??

↳ The idea can come from lru cache.

↳ for lru cache to find a node fast, we maintain a hashmap of data & addr node.

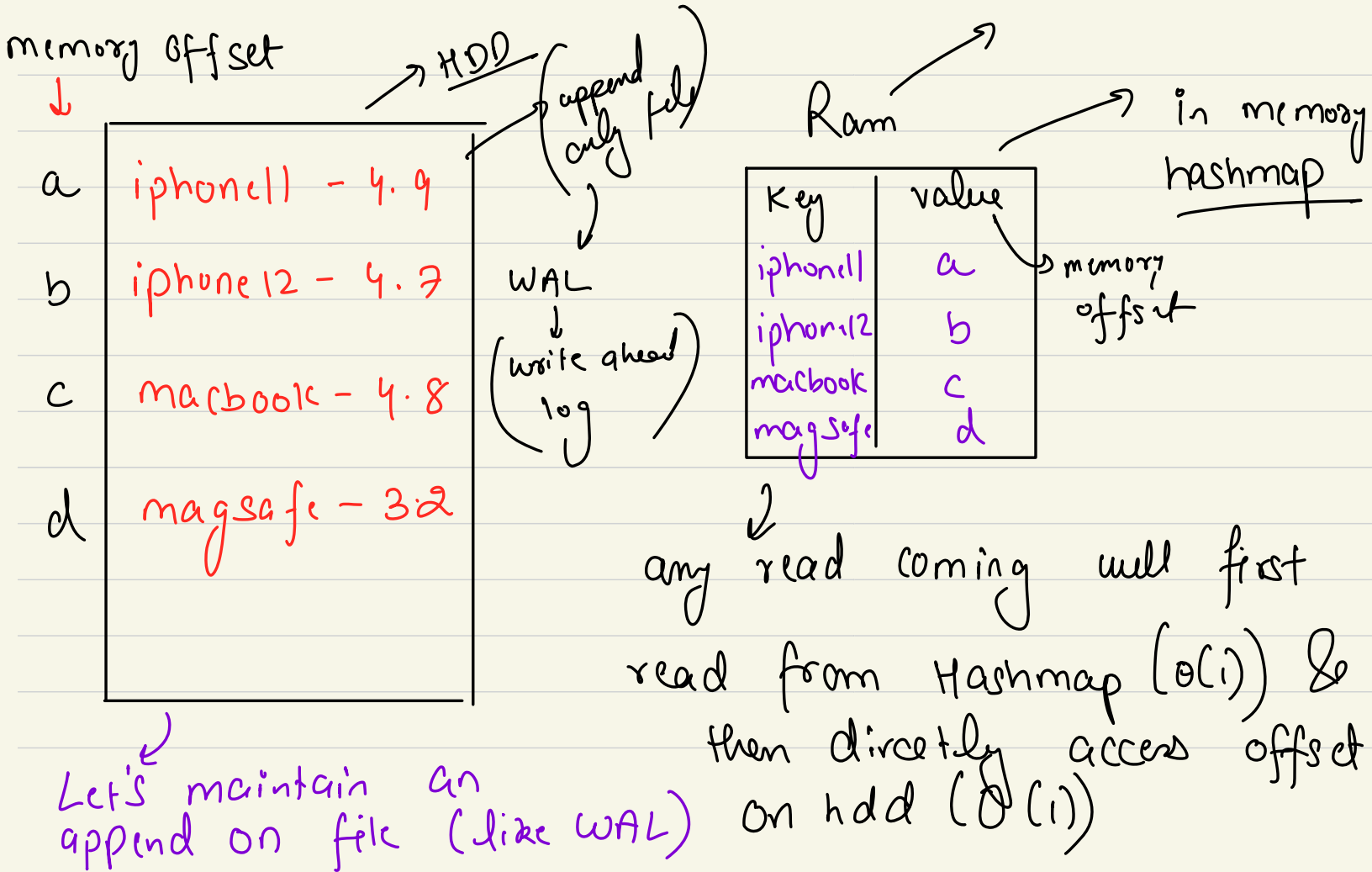
Similarly, what if we maintain a hashmap of the data and the address on disk where latest data is stored ??



key - value

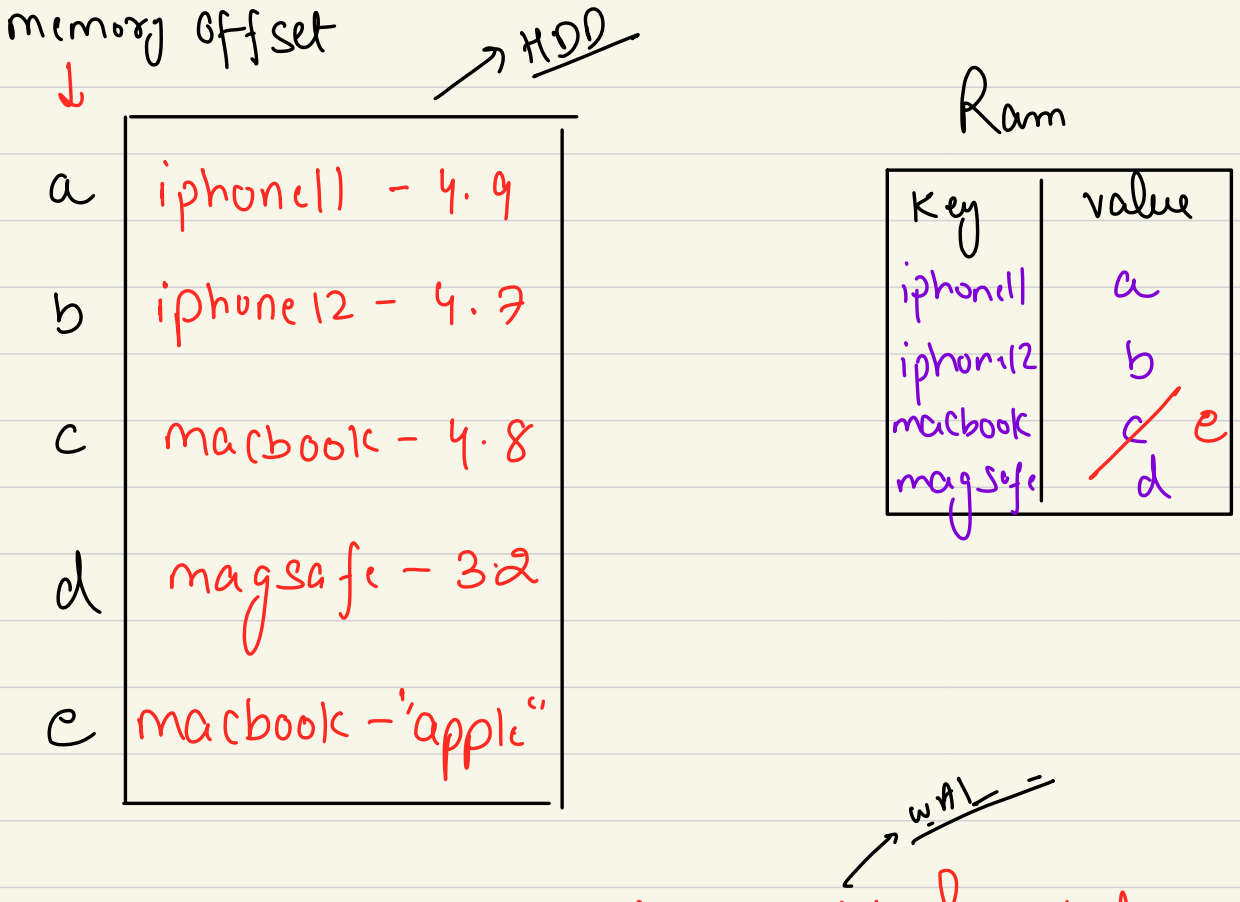
H.M → RAM

key → addr → O(1) Searching


memory offset
HDD
a
iphone11 - 4.9
b
iphone12 - 4.7
c
macbook - 4.8
d
magsafe - 3.2
append only file
WAL
write ahead log
Let's maintain an append on file (like WAL)
Ram
in memory hashmap
Key
value
iphone11
a
iphone12
b
macbook
c
magsafe
d
memory offset
any read coming will first read from Hashmap (O(1)) & then directly access offset on hdd (O(1))

Ram read & writes are faster	HDD read & writes are slower

Reads on Ram are 100x faster & writes 10^5x faster than HDD.



for updates, we append in hdd & update offset in

hashmap

Quick question →

why are we even writing in hdd if it's that slow & only write in RAM hashmap ?

→ storing only in RAM lacks durability.

→ value can be a big complex object leading to fast filling of RAM if we only rely on that.

Are there problems with this approach??

1) Old data is, never deleted so how to handle this ever growing data ??

a) for billions of keys (unique) the RAM won't be able to fit the entries (ideally 10^9 keys ~ 1B is 1gb)

3) What about support for partitioning & sharding ?
(because file can become huge)

4) if system crashes, then data in RAM will be lost, what about that ??

↳ using the data on hdd we can recover the data & reconstruct the RAM offset hashmap.

LSM Trees

(Log structured merge tree)

used by

- → Cassandra
- → Scylla DB
- → DynamoDB
- → Big table
- → Couch Base
- and more ....

(With the current setup of a file (append only) & a hashmap we can make a few changes & get LSM)

The append only file we have is called WAL → in HDD

↓ write ahead log

↓ used by kafka, nosql, sql

Q→ why RDBMS uses WAL ?

To speed up writes. how ?? → rdbms writes data on hdd. To improve reads they support indexes (B+tree)

#fact → On a HDD, sequential read is way faster than random read. (Read happen in blocks)

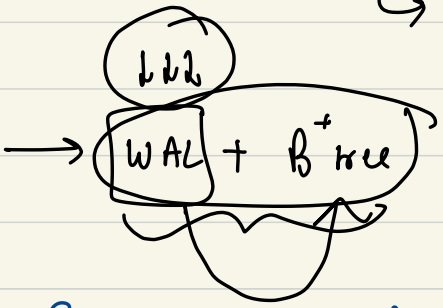
at every write on sql db,

↳ we update data → 1 disk seek

↳ update B^+ tree index → $\log n$ disk seek

↳ & rebalance

total $\log n$ on disk



So instead of rebalancing on every write sql uses WAL

In this WAL it stores these recent writes (maybe 100 ↙ config batch size)

and then when our batch is full, then we move it from WAL to B^+ tree index.

→ WAL file is also used for master slave replication.

2

There's a little bit of overhead to updating entries in an index, but it's reasonably low cost. Most indexes are stored internally as a B+Tree data structure. This data structure was chosen because it allows easy modification.

MySQL also has a further optimization called the Change Buffer. This buffer helps reduce the performance cost of updating indexes by caching changes. That is, you do an INSERT/UPDATE/DELETE that affects an index, and the type of change is recorded in the Change Buffer. The next time you read that index with a query, MySQL reads the Change Buffer as a kind of supplement to the full index.

→ similar work around like WAL

A good analogy for this might be a published document that periodically publishes "errata" so you need to read both the document and the errata together to understand the current state of the document.

Eventually, the entries in the Change Buffer are gradually merged into the index. This is analogous to the errata being edited into the document for the next time the document is reprinted.

The Change Buffer is used only for secondary indexes. It doesn't do anything for primary key or unique key indexes. Updates to unique indexes can't be deferred, but they still use the B+Tree so they're not so costly.

If you do OPTIMIZE TABLE or some types of ALTER TABLE changes that can't be done in-place, MySQL does rebuild the indexes from scratch. This can be useful to defragment an index after you delete a lot of the table, for example.

28.3. Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) is a standard method for ensuring data integrity. A detailed description can be found in most (if not all) books about transaction processing. Briefly, WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, after WAL records describing the changes have been flushed to permanent storage. If we follow this procedure, we do not need to flush data pages to disk on every transaction commit, because we know that in the event of a crash we will be able to recover the database using the log: any changes that have not been applied to the data pages can be redone from the WAL records. (This is roll-forward recovery, also known as REDO.)

> **Tip**
>
> Because WAL restores database file contents after a crash, journaled file systems are not necessary for reliable storage of the data files or WAL files. In fact, journaling overhead can reduce performance, especially if journaling causes file system *data* to be flushed to disk. Fortunately, data flushing during journaling can often be disabled with a file system mount option, e.g., `data=writeback` on a Linux ext3 file system. Journaled file systems do improve boot speed after a crash.

Using WAL results in a significantly reduced number of disk writes, because only the WAL file needs to be flushed to disk to guarantee that a transaction is committed, rather than every data file changed by the transaction. The WAL file is written sequentially, and so the cost of syncing the WAL is much less than the cost of flushing the data pages. This is especially true for servers handling many small transactions touching different parts of the data store. Furthermore, when the server is processing many small concurrent transactions, one `fsync` of the WAL file may suffice to commit many transactions.

WAL also makes it possible to support on-line backup and point-in-time recovery, as described in Section 25.3. By archiving the WAL data we can support reverting to any time instant covered by the available WAL data: we simply install a prior physical backup of the database, and replay the WAL just as far as the desired time. What's more, the physical backup doesn't have to be an instantaneous snapshot of the database state — if it is made over some period of time, then replaying the WAL for that period will fix any internal inconsistencies.

Note → we can only do insert at end & get operation on WAL file.

Q: Why NoSql use WAL?

Nosql use WAL as a part of data storage, mostly facilitated by LSM.

→ Now coming back to our data structure we will use WAL in following way:

1) We will use WAL to keep appending the data.

2) We maintain a hashmap to track for a key what was the latest offset for that key in WAL.

→ called as Compaction

Q: How can we remove duplicates from WAL ?

→ how about we run a periodic job to remove old entries.
But with this deletion will be happening in btw the file, so to reclaim that space we might need to shift remaining data.

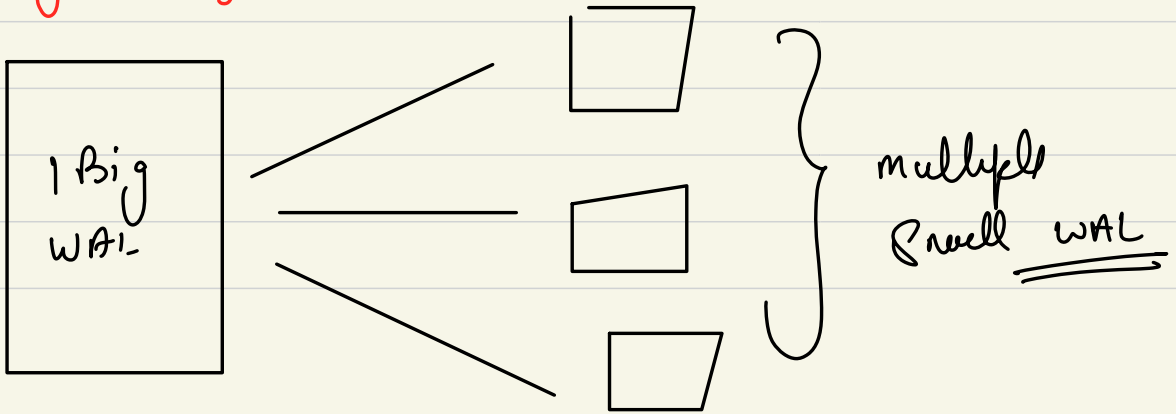
Also when we delete then we won't be able to take new writes until delete finishes.
& reads

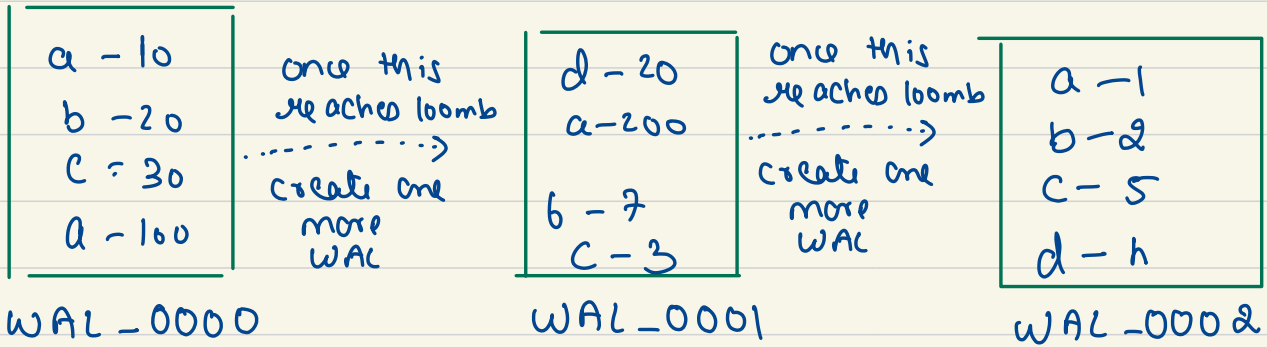
Also RAM hashmap will undergo a lot of updates.

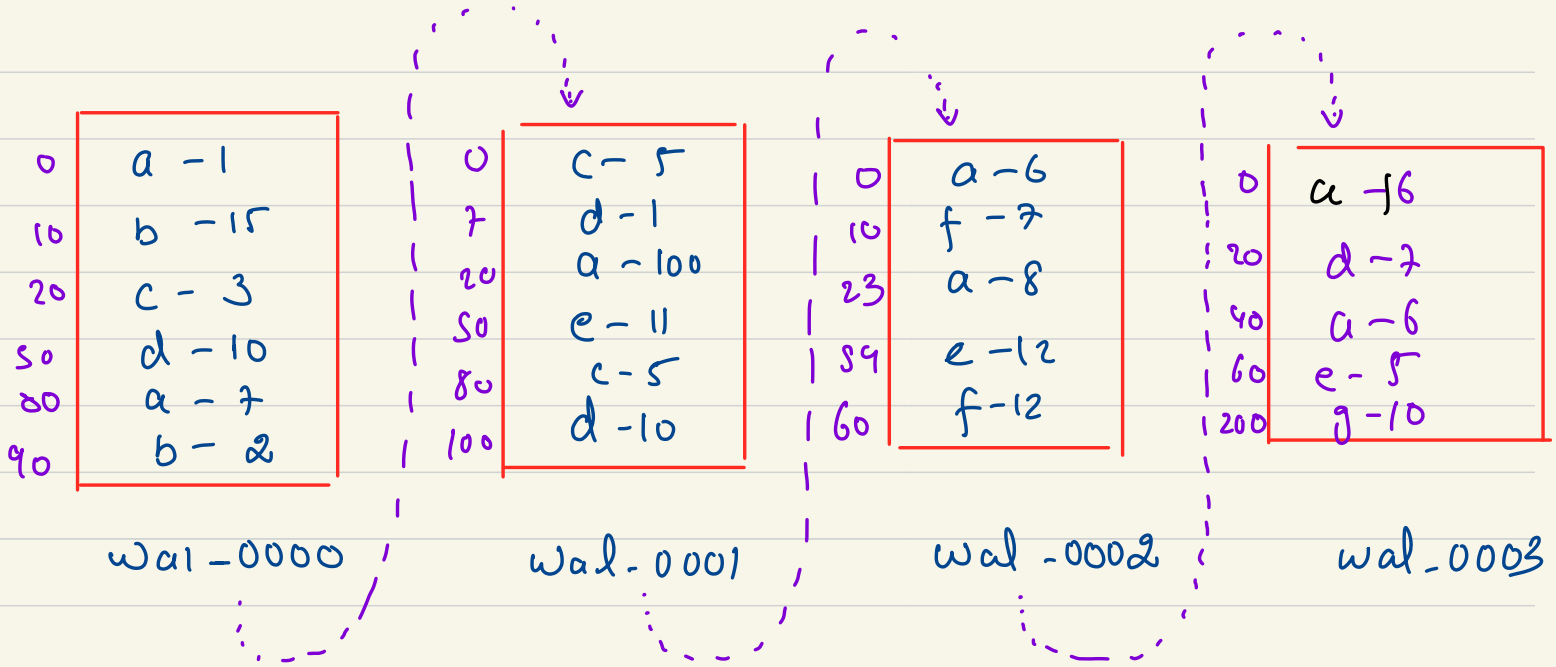
Now because WAL can be heavy, so this deletion needs to handle file chunk by chunk. as the complete file is heavy & big.

Q:- Now if we are already suggesting to read file chunk by chunk → why not maintain multiple smaller size WAL files?

also in the hashmap now instead of just storing key & offset, we store key, wal file & offset.




a - 10
b - 20
c - 30
a - 100
once this reaches 100mb
create one more WAL
d - 20
a - 200
b - 7
c - 3
once this reaches 100mb
create one more WAL
a - 1
b - 2
c - 5
d - 4
WAL_0000
WAL_0001
WAL_0002



hashmap →

key	file	offset
a	0003	40
b	0003	0
c	0001	80

d	0003	20
c	0003	60
f	0002	60
g	0003	200

Read → get (key) → find in hashmap O(1) RAM
↳ get file + offset O(1) disk

put (key, value) → append to latest WAL file → O(1) disk
↳ set in hashmap → O(1) RAM
↳ if WAL file size > 100mb:
create new WAL file → O(1) disk

So now both get and put are efficient.

Note → all writes are happening in active WAL file

reads can happen in any WAL file.

→ Now because of this, Compaction process can work on old files

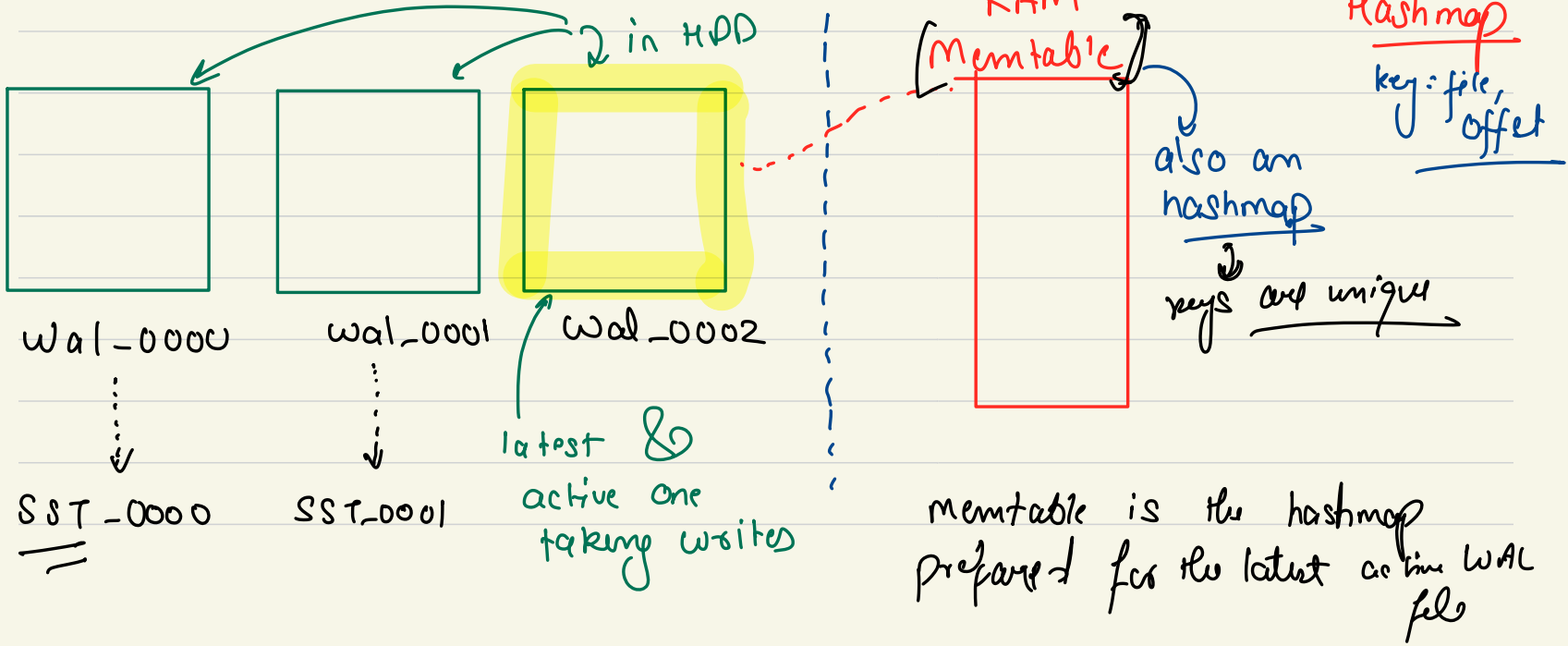
It will not interfere with writes, & writes won't be blocked.

Reads might be impacted.

Can we improve further?

Now our latest WAL file is only 100mb.

So how about we keep the latest WAL file in disk & RAM both.



Memtable refers to the RAM version of actual WAL.

Why we want to keep it?

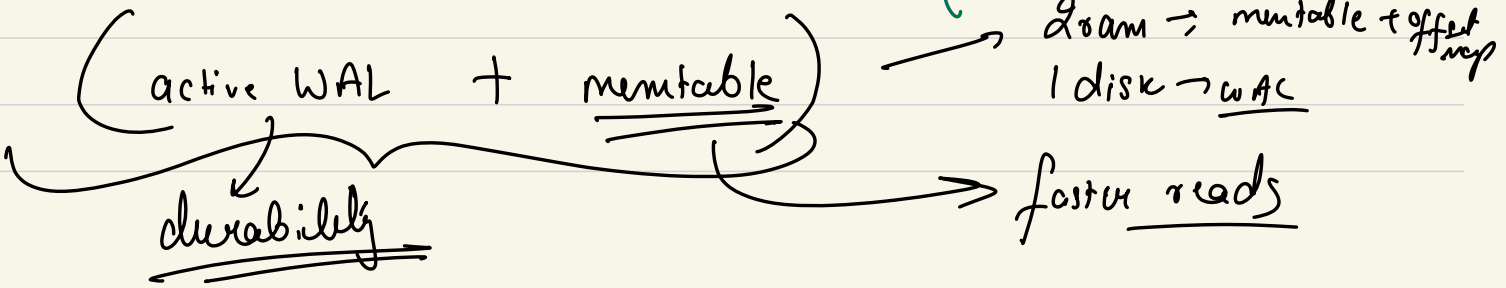
1) On disk updating is issue. as size keep on changing
2) On disk pointers are bad as random access is very slow.

So memtable will not keep a duplicate as it will be also a hashmap.

also we can agree that data on memtable is always latest as it's representing active WAL.

Now if we do read, & key is in memtable then read is going from memory, hence its very fast.

So when we write, we update memtable & also append to WAL (to ensure durability)



Now what if while appending to WAL we don't immediately write to file offset hashmap?

This would be nice as latest data is already coming from memtable.

Q> So when should we add data to hashmap of offset?

→ So once our WAL file (active one) is full, then we dump it's data in offset hashmap.

Q: Can we optimize what we do once a WAL file is full?

1) So when WAL (active) is full, we create a new file.

2) But what if we dump the active WAL in a new file in a way that we store data in arranged form. ?

3) So we can use the memtable (which has all unique value) & take all key value, sort it in memory, and dump these unique keys in sorted form to a new file.

4) This file will act as a newly created dump of the last active WAL but in sorted form with all unique keys.

5) Instead of keeping exact WAL file, we store this new file as it has lesser data (as only unique keys from memtable is added) & sorted data (so searches are faster)

6) This new file is called as SST (sorted set table)

keys are stored in sorted ways

because all unique values

So here are the steps for write once WAL file is full:

1) flush memtable on disk as a new SSTable
2) SSTable will be sorted by key
3) Update offset hashmap.

4) Create a new WAL file & dump old one (as data is in new SSTable in persistent fashion)

Q: Should we clear memtable ?

→ No, we can keep an upper limit to size of memtable.

→ Once we reach limit, we can have LRU or LFU or fifo eviction (implement like in mem cache).

Q. Is the problem of duplicates solved?

→ In a single SS Table we don't have duplicates

→ But between 2 SSTables, duplicates can be there.

Q. Is the problem of partitioning solved?

Yes. These SS tables doesn't need to be on the same server.
Also we can easily replicate it.
Also reads and writes are also faster.
↓ how?

↓

as memtable is acting as a cache, & if the hit ratio is good, the data reads will be way faster.

Now as majority of the reads are gonna be handled by memtable & it definitely contains latest data, compaction can happen on WAL files without hampering reads.

↓

SST files

Compaction → procedure to remove duplicates

→ operate in background on SS Table.

(merge sort)

→ SS tables are small, so we can easily load 2-3 tables in RAM.

→ Say we take 2 SS tables, load in RAM & remove the duplicates and create a new de-duplicated WAL.

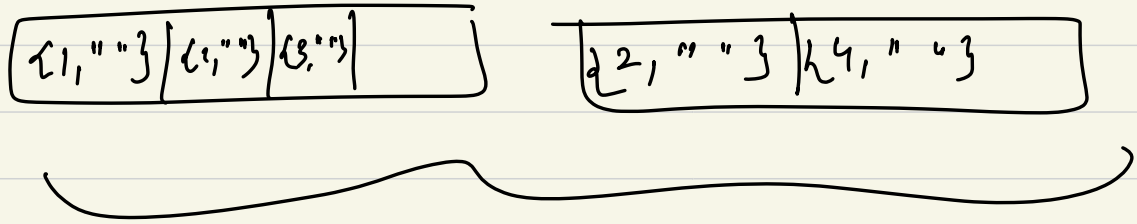
→ Very similar to merging 2 sorted arrays.

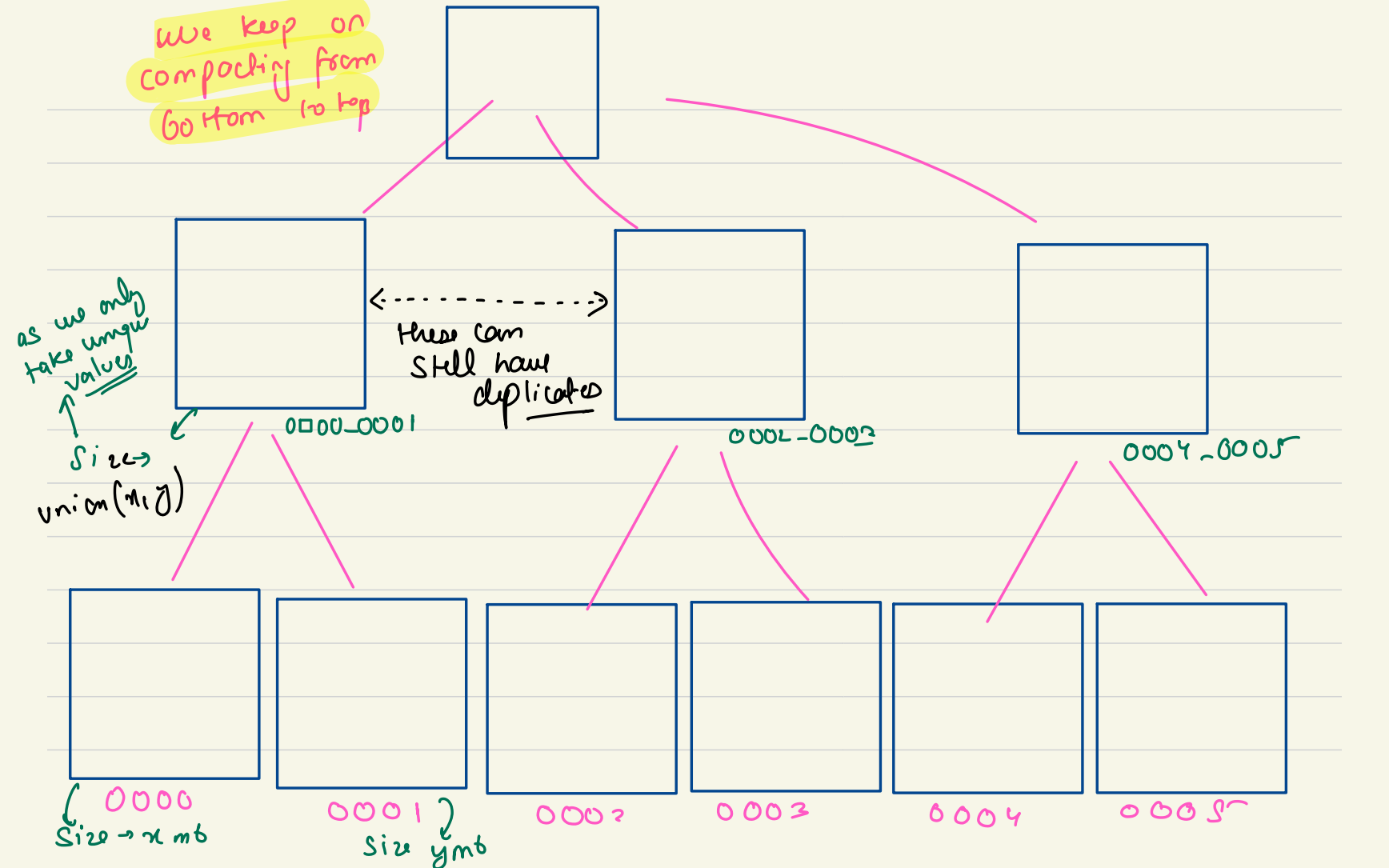
→ So just like in merge sort, we can combine 2 SS tables into 1 with only unique entries.

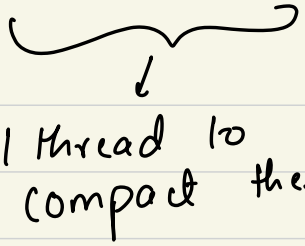
→ To know which SS table has new value we can use the file name.

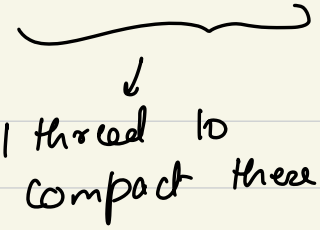
→ while merging we keep latest entry & discard old one.

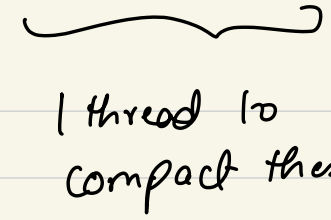
So what if keep on taking 2 SS tables, merge them & keep doing it recursively ??




we keep on compacting from Bottom to top
as we only take unique values
Size → union(x,y)
These can still have duplicates
0000_0001
0002_0003
0004_0005
0000
Size → x nb
0001
Size y nb
0002
0003
0004
0005







Once we have compacted some files, we can remove them from HDD.

Look good. Can we now eliminate the offset hashmap?

→ Currently we use the offset hashmap to find an entry.

→ This hashmap can become too large in size.

→ If we know which SSTable to search we can use binary search to get the data.

Q. → how to know the correct SSTable to search for?

→ Because of compaction, no. of SSTable will be less. (but of larger size) as we get rid of lower SS tables

→ Because of tree structure, # of SSTable is approx $\log n$

There are $\log(n)$ disk access, size of table.

We should reduce these $\log n$ disk reads as they are expensive.

Sparse Index

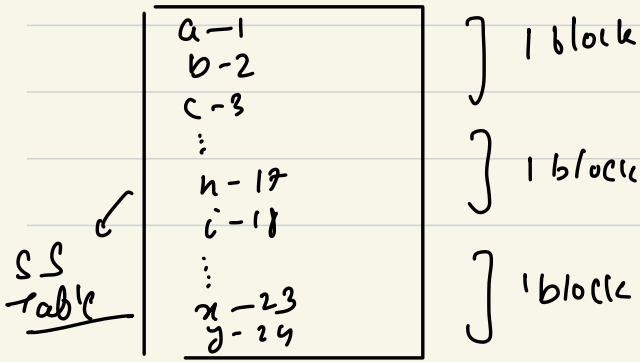
Let's divide our WAL file in small chunks such that each chunk can go on a single block of HDD.

Now if we know the right chunk, then in one HDD read we get complete block & we can binary search on it.

WAL → SST → Block

☆ So for each SSTable we maintain a sparse index.

This sparse index contains starting value of each block.



Sparse Index (RAM)

a-0
n-50
x-200

probably 100 entries

Note → SS tables are immutable, so we can only create it & search on it, no CRUD.

So how do we delete a key?

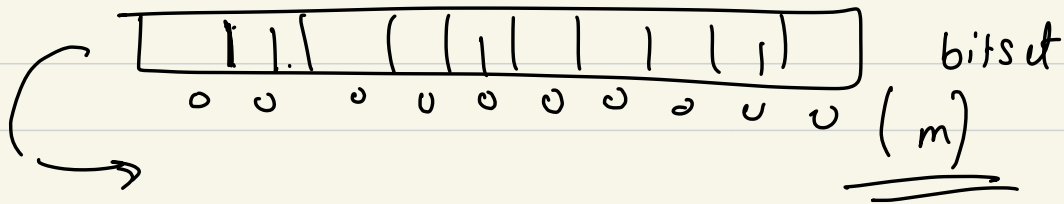
→ We make a new entry of key with a Sentinel value.

(Sentinel value is a special value to mark end of a sequence or data structure)

Also to check if a key is present in LSM or not we should

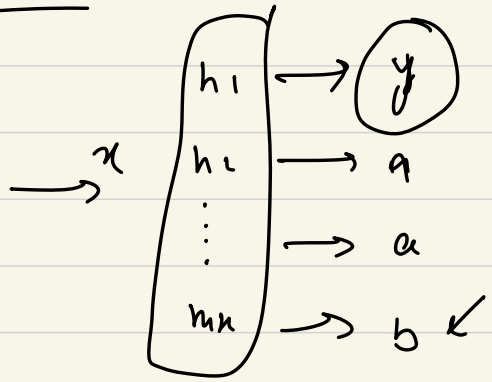
not go to all SS table, instead we can use Bloom filters.

↳ probabilistic, space-efficient data structure to test if an element is a member of a set of values or not.

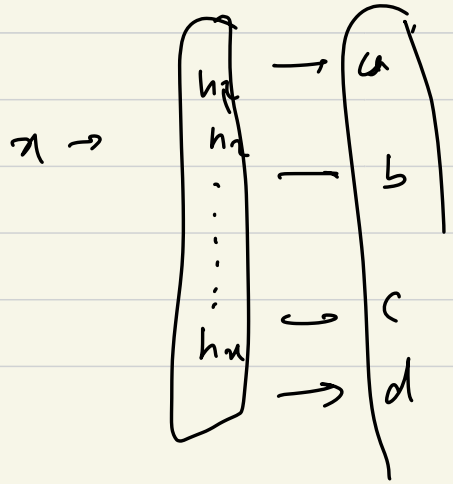


K – hash func^n

add (x)

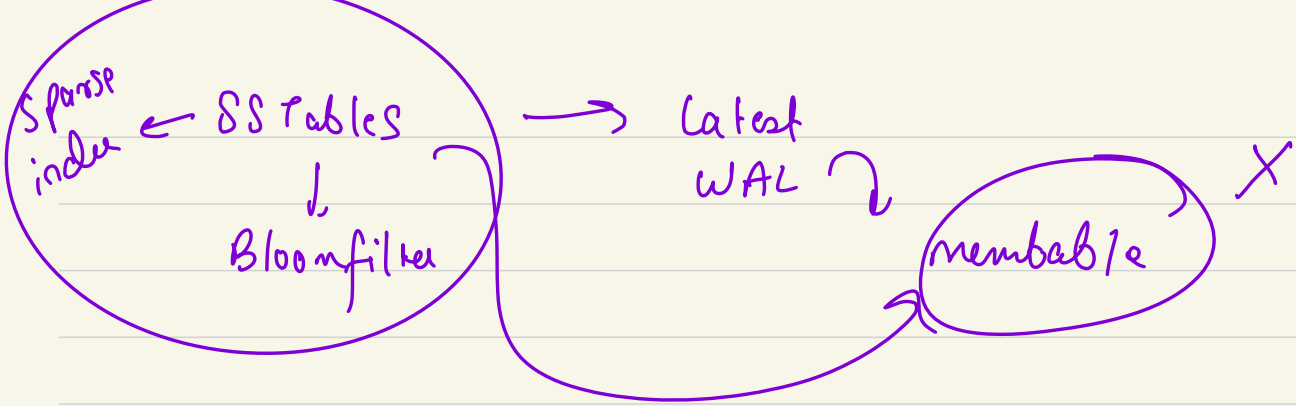


check (x) →

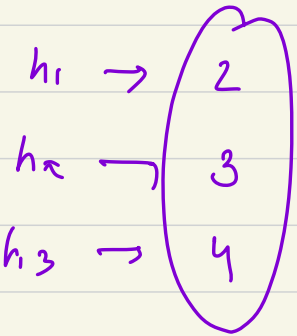


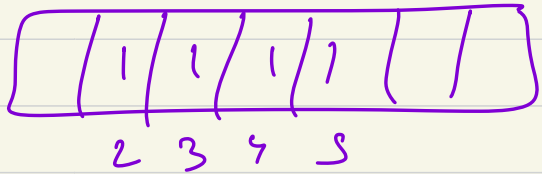
if all bits are set, then this value might be present

if some are not set in the bitset, then defaultly it is a no.


Sparse index
SSTables
Bloomfilter
Latest WAL
memtable
X

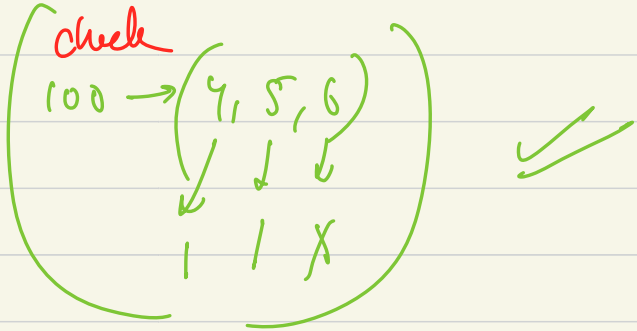
add 10 → 2, 3, 4

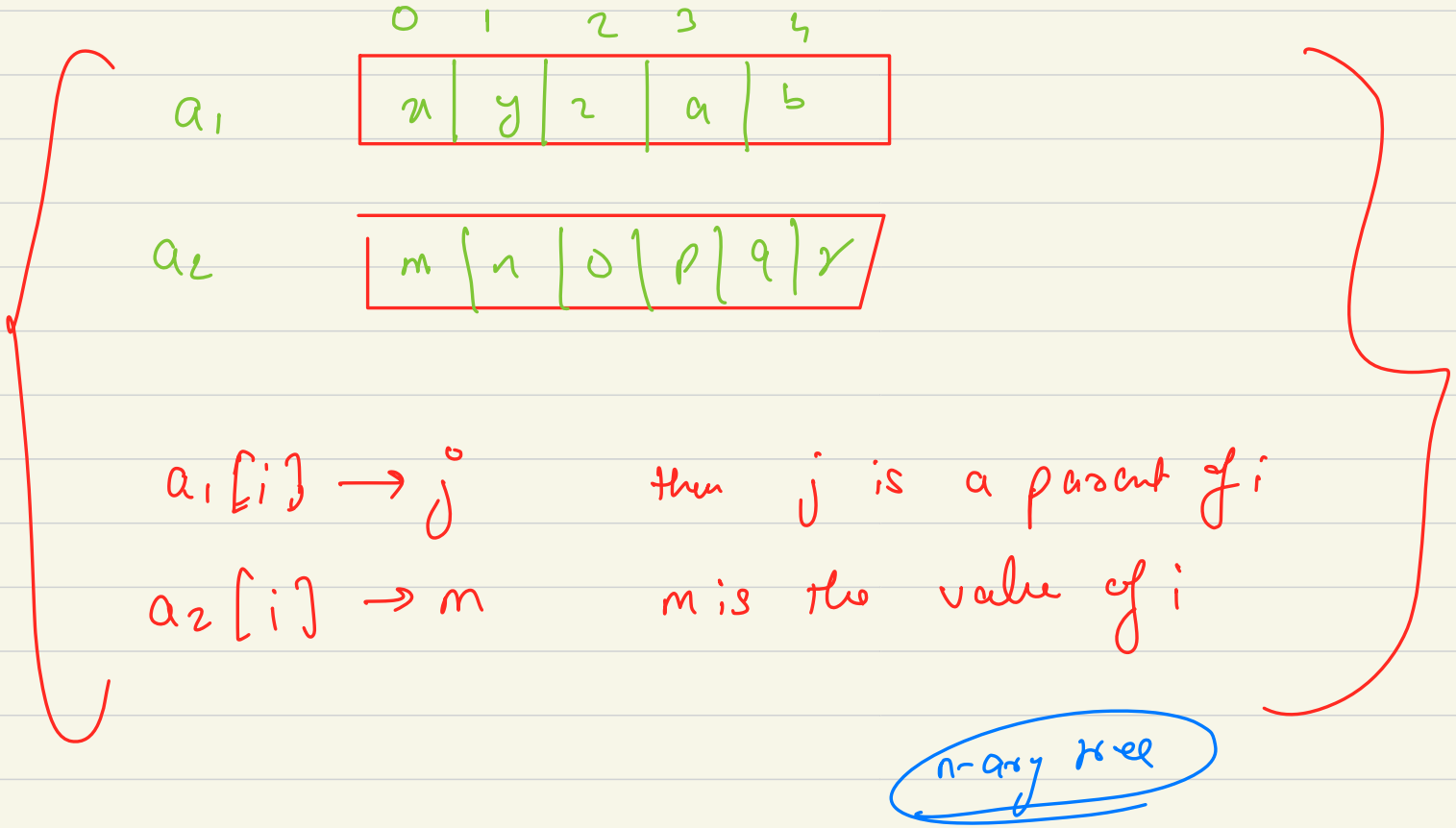


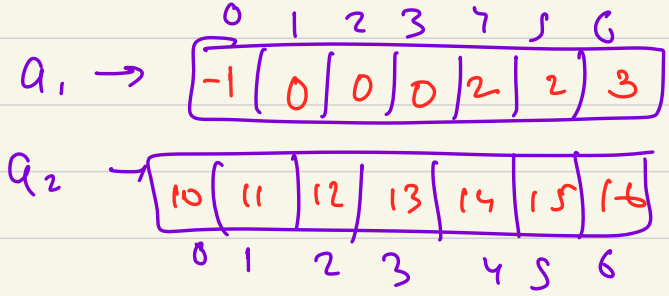


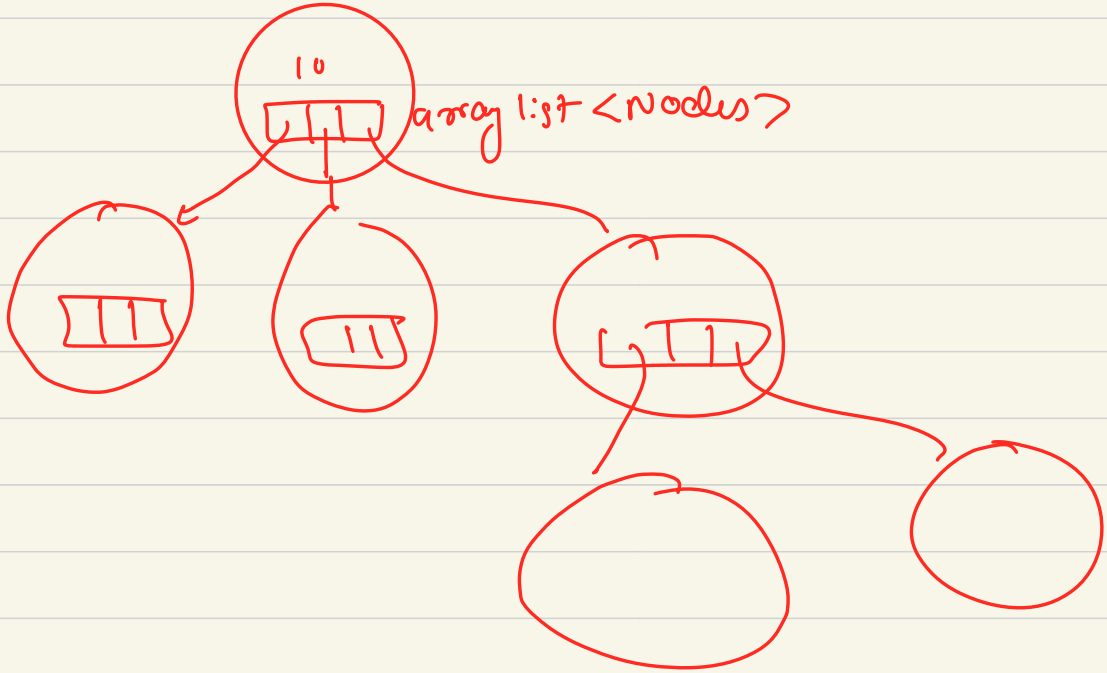
add 20 → (3, 4, 5)

check 30 → (2, 4, 5)




0 1 2 3 4
a_1
x y z a b
a_2
m n o p q r
$a_1[i] \rightarrow j$ then j is a parent of i
$a_2[i] \rightarrow m$ m is the value of i
n-ary tree


0 1 2 3 4 5 6
q_1 →
-1 0 0 0 2 2 3
q_2 →
10 11 12 13 14 15 16
0 1 2 3 4 5 6


10
array list <Nodes>