Agenda

- ✓ Problem Statement
- ✓ Write Ahead Log (WAL)
- ✓ LSM Tree
- ✓ Bloom Filter

} overview / intuition

NoSQL → Sharding automatically
→ managing servers

Today → how is data stored on disk.

Problem Statement

↳ SQL → well defined schema
I know the exact size of each row
col → data type

Users
id name
8 byte varchar(50)

~~20~~ 30	Rohan
5	~~Nikhil~~

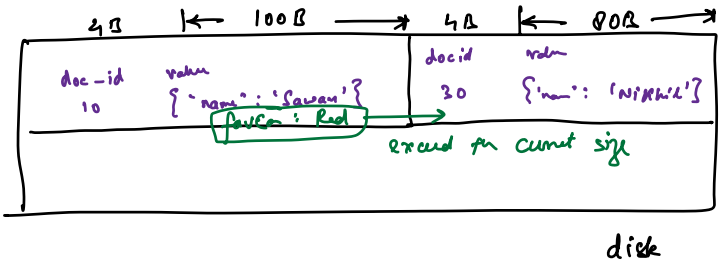
Nikhil Agrawal.

Size of row remains same in changes

↳ NoSQL → different

Key-value store
{ string : string }

document store
{ json object }



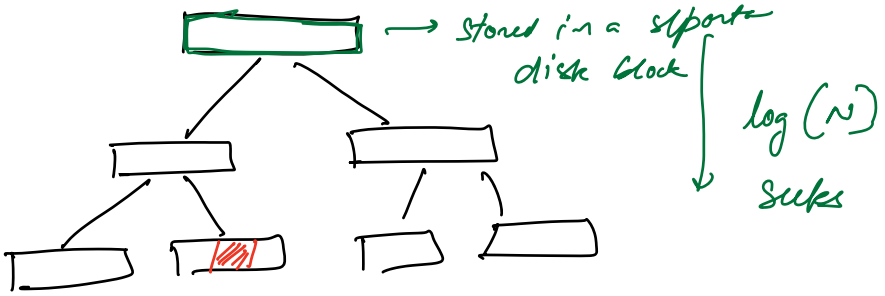
db.update(10, {'value': 'Saveen', 'favColor': 'Red'})

① Cavities → over writing

② Split the document across disk → fragmentation
a lot of disk seeks → very expensive
(100 ms)

even for SSD
↳ sequential I/O is 100x faster than random I/O

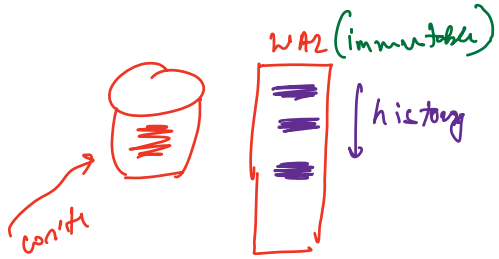
SQL → Indexes → B+ Trees



Goal — NoSQL → optimized for both read & write loads

For any database (Especially NoSQL) — in SQL you can turn it off

1. Write Ahead Log (WAL)
   - ↳ append only
   - ↳ every write → appended to WAL
   - ↳ allows you to replay the entire history of DB.



   - ↳ power failure recovery is done via WAL

↳ replication

↳ db Backups

↳ Point of Time Recovery

WAL can also be read from

read the tail of the file

look at the last 2Mb / 200 entries

of the file

② Current state of data

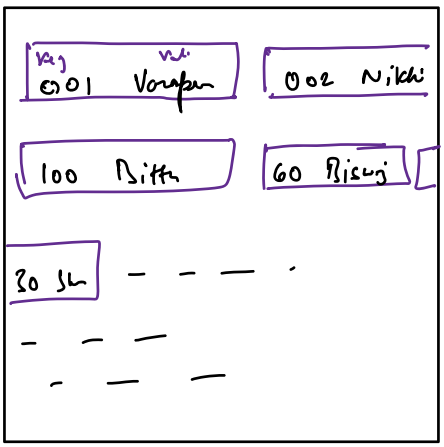
discuss how to implement efficiently.

Key-value Store → Design how the data is written on disk

if each entry was fixed size → B^+ tree

↳ not the case for NoSQL dbs

Brute force



write values to a file

(1) Read key (60)

Linear Scan entire file

(2) write key (60) → Risuajit

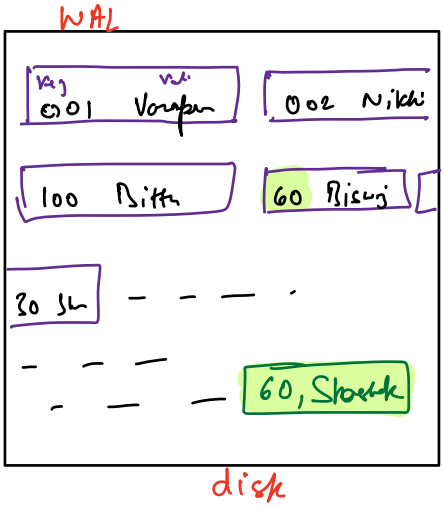
Linear Scan & update

↓

tricky ∵ value can be ↑ varied size

Use WAL

Every write → appended to the file



(1) read key (60)

Linear scan from last to first (take most recent write)

(2) write key (60) → Shoelok

append a new entry to file

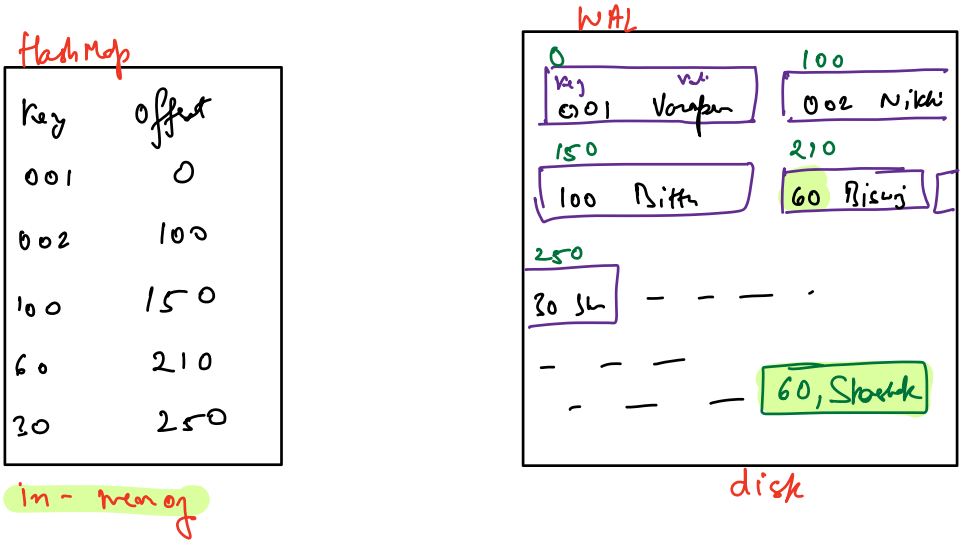
O(1) writes extremely fast

There will be duplicate entries!

writing → super fast read → still super slow.

Option → B tree → sorted data → can't scan slower
↳ varied sized entries

HashMap → given a key it location in file
(memory) (byte offset)



def read (key):

0.1 µs { offset = Hmap.find (key)
10,000x
10 ms { buffer = read the next n bytes from file start from 'offset' O(1)
key, value = parse (buffer)
return value

```
def write(key, value):
    offset = WAL.size()
    append(file, [key, value])
    Hmap.set(key, offset)
```

O(1)

Still duplicates in WAL

The reason we're still appending in WAL for writes is because data has varied size.

If I try to modify existing value → overflow

---

(1) Still duplicate entries → try to remove duplicates periodically

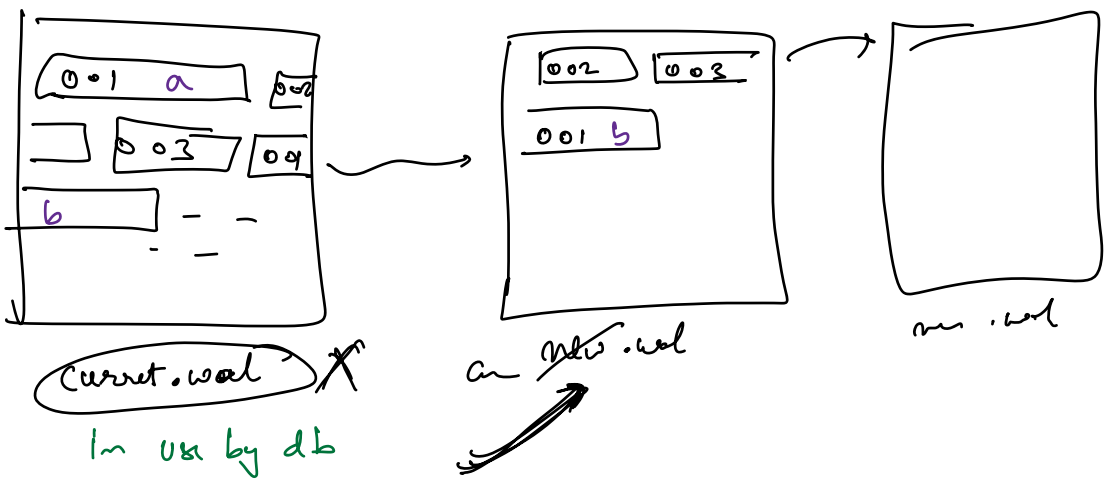
(2) In a DB, there might be billion / trillion keys

Key – offset Hashmap → might not fit in RAM

We have a cron job / background process

WAL —(read)→ finds duplicate keys → discards old value → writes WAL again

update HashMap



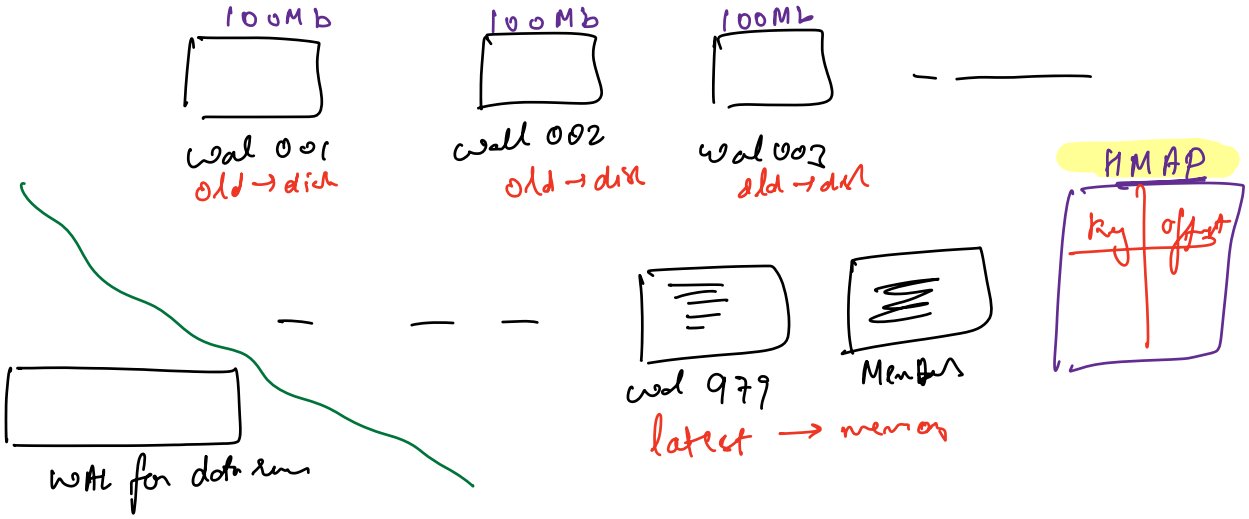
WAL are pretty large

↳ ① can we find duplicates efficiently?

↳ ② we can't read entire WAL in one go

Can we process the WAL in chunks? ✓

Can we also just store WAL in chunks?



① Any write that happen → latest WAL file

② each chunk of WAL is small → store latest (current) chunk in memory instead of Disk??

↓

write a now even faster!!

to ensure that data is not lost in case of power failure

any write → changed in memory
↳ appended to separate WAL on disk → very small ~100 Mb

latest WAL → in-memory

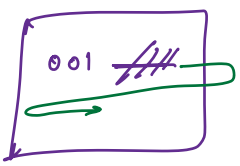
↳ doesn't need to be append only

[- - - -]

MemTable

↳ does not need to have duplicates

assume: this is a hashmap



this Memtable will also have latest data

↑ write 001

any read for which key found in WAL

↳ read can be fetched from Memory

↳ doesn't have to touch disk!!

After some writes the Memtable will get large

>100 Mb (threshold)

↳ dump it on disk as a WAL file

→ Any WAL chunk will not have any duplicates

→ Across different WAL chunks → duplicates

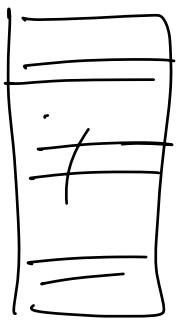
So we still need a background process to compact data

---

(1) Memtable → in memory → random order

↳ HashMap → Indexing is O(1)
↳ wastage of space
↳ dumping on disk loses the HMap property

↳ BBST



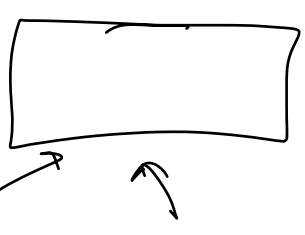
→ Operation is O(lg n)

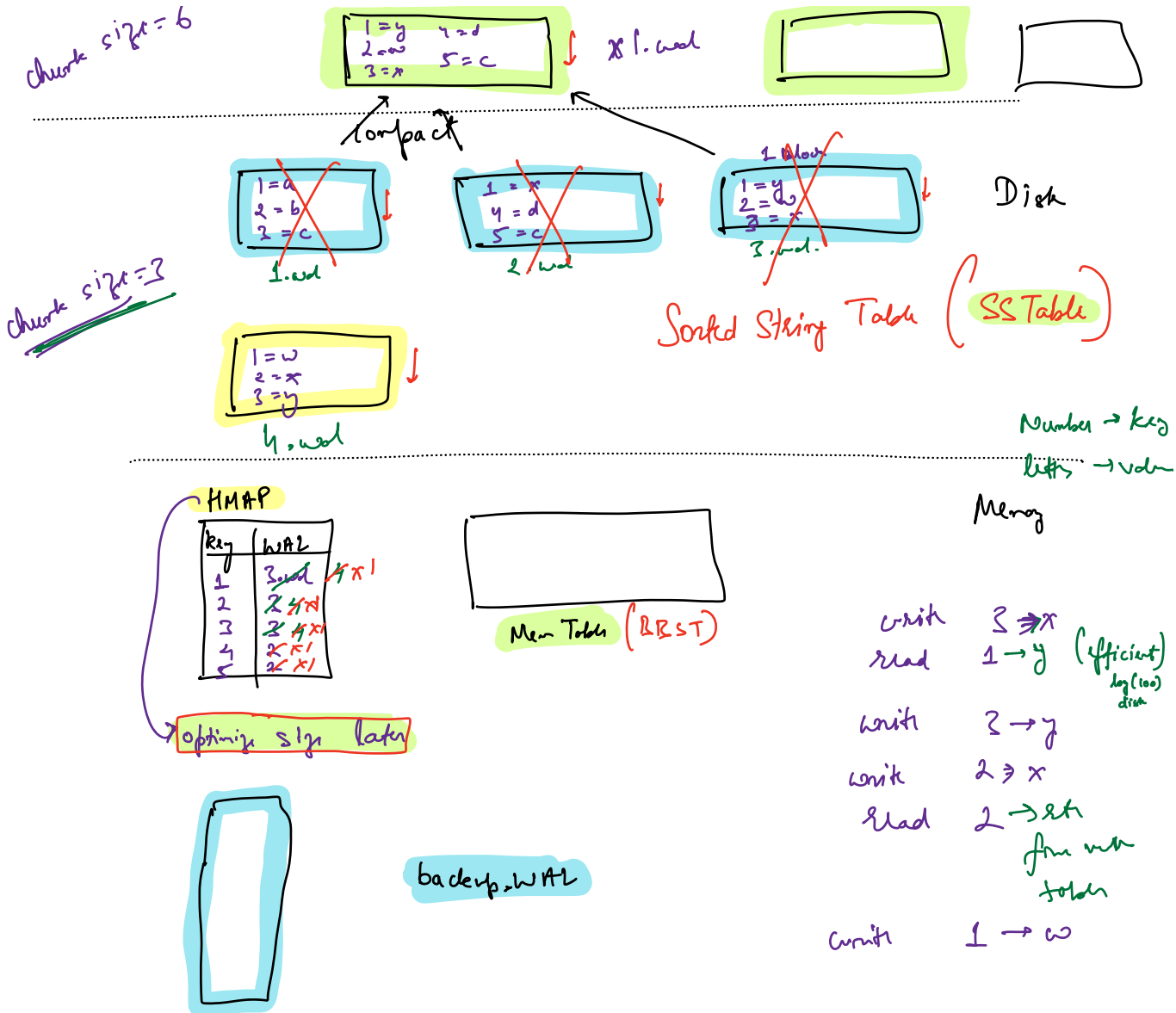
→ no space wastage

→ dump it on disk → sorted order

Every WAL chunk will be automatically sorted !!

→ allows us to avoid the key → (val, offset) hashmap partially

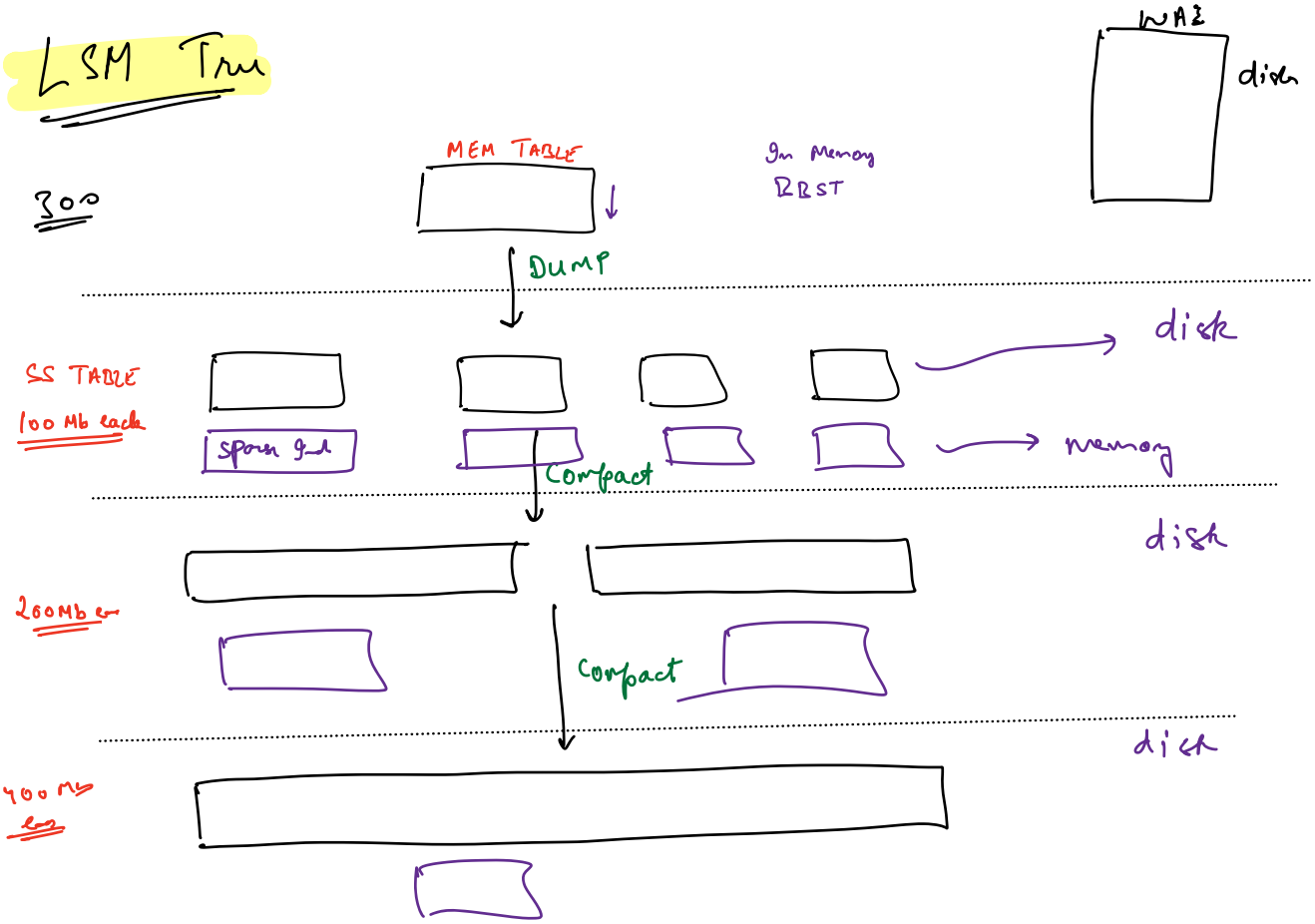




1. power failure → recover from backup.wal
2. Mem table is full
3. Background compaction process

LSM Tree

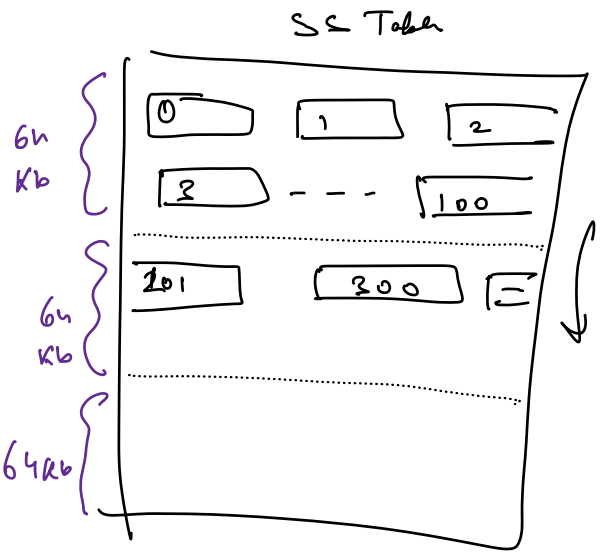
LSM Tree



SS Table → 1 GB

(sorted)

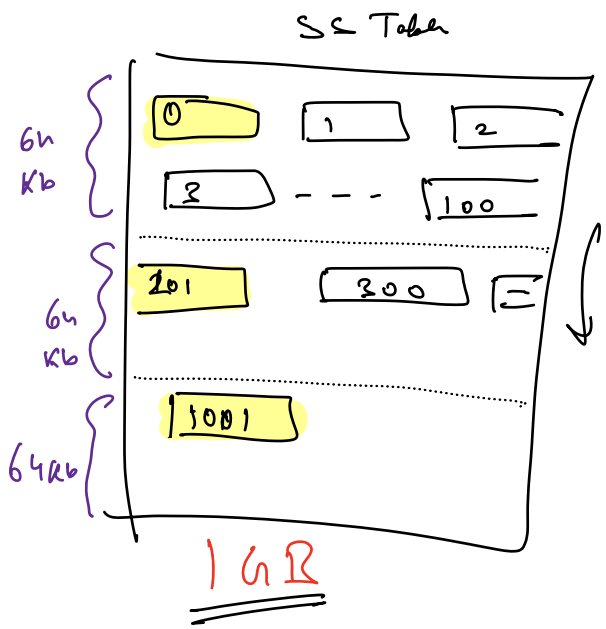
64 Kb block



how many chunks? $\frac{1 \times 2^{30}}{64 \times 2^{10}} = 2^{14}$ chunks

= 16 K chunks

Sharad



Index

0, offset = 0

201, offset = 2000

1001, off = 5000

}

16 K entries

≈ 1 MB

→ no values (only keys)

→ on avg each key ≈ 64 bytes

$16 \times 2^{10} \times 64 = 2^{20} B$

$2^4 \quad 2^6$

$= 1 MB$

Problem If key is not there → not slow

→ but still we doing a bunch of work to return nothing !!

Bloom filter → probabilistic data structure

insert (key) → does Not increase size at all !!

contains (key) → true/false

if key is present → guaranteed to return true

is absent → return false with high probability ← turn this probability.

(false positive)

Bloom filter

insert ⇒ Nikhil → H1 H2 H3 → 2, 3, 9

contains (Nikhil) → H1 H2 H3 → 2, 3, 9 if all are set

insert → Vishal → 2, 5, 11

contains → Varun → 1, 3, 7

contains → Akash → 3, 9, 11 → false +ve

0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	1	0	1	0	0	0	1	0	1

M = 12 (size of bloom filter)

K = 3

Hash functions (k)

H1(s) → INDEX(s0) % 12

H2(s) → ≠ (s1) % 12

⋮

K → Hash funct

N → # of values inserted

M → # bits in Bloom filter

Probability of false +ve $\approx \left(1 - e^{-kn/m}\right)^k$

M = 10 MB = 10×2^{20}

N = 1 billion keys = 10^9

k = 10

NoSQL → Bloom filter to store which keys are present

Akamai / Cloudflare → CDNs (70% of URLs are only visited once)

↓

Cache a URL only if it has been visit 2 times

Trillion URLs → Each URL path by ~ 500B

500 TB of data → Store

Bloom Filter

Delete (key) ⟶ write (key, Tombstone)

LSM

↓

new for a value special significance

Read (key) ⟶ if val = Tombstone

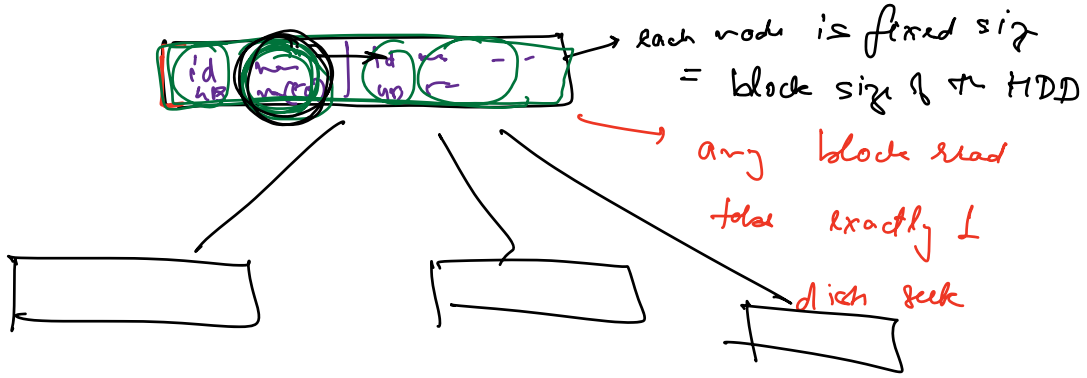
⟶ not found

or

⟶ val value

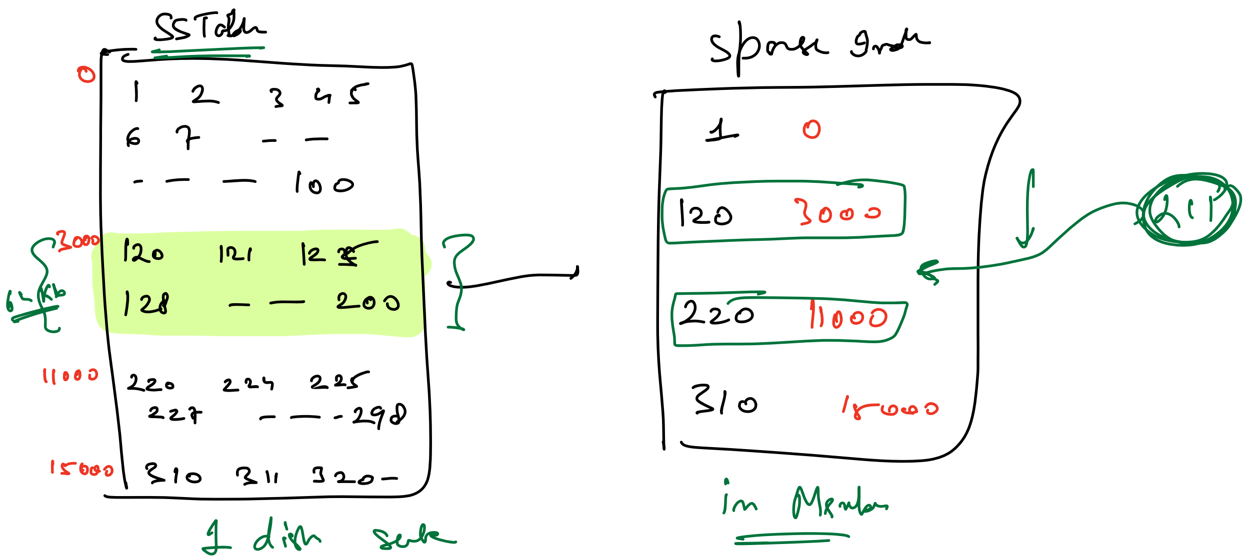
T1 ✓: "Vanaprasad's LSM Tree Delete" value ⟶ ✓

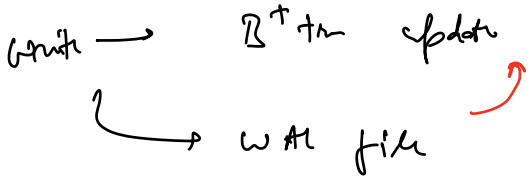
T2 ✓: ~~OK~~



1. duplicate → remove duplicates.
2. Mem

WAL vs




write
B+ tree
update
WAL file