INDEXES

Agenda.

1. What are indexes
2. How indexes work
   → B and B+ tree
3. Cons of indexes
4. Indexes on multiple columns (How they help)
5. Indexes on strings
6. Indexes PRACTICAL
   ↳ EXPLAIN.

Till now we majorly focused on writing queries or how do they function via pseudo codes.

We always said pseudo code is not the exact working code, just a medium to build intution of how things might be happening behind the scenes.

eg for joins.

```
for i ____ 100 :
    for j ____ 100:
        doSomething();
```

} joins

if the DB was exactly doing this.

⇒ Performance will be very slow.

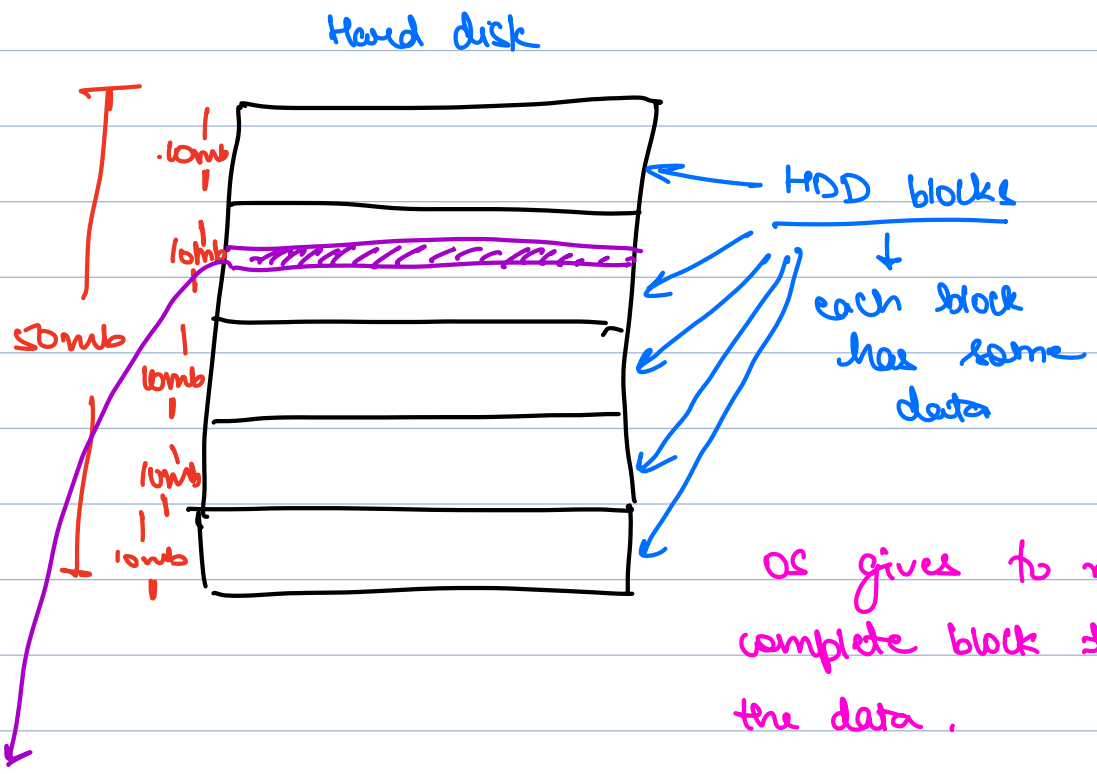
① DB have to do a lot of optimizations to make queries fast.

② DB stores data on disk.

vs Program → stores in memory (faster)

Show table of latency for disk vs memory.

HDD ⇒ for slower ⇒ RAM



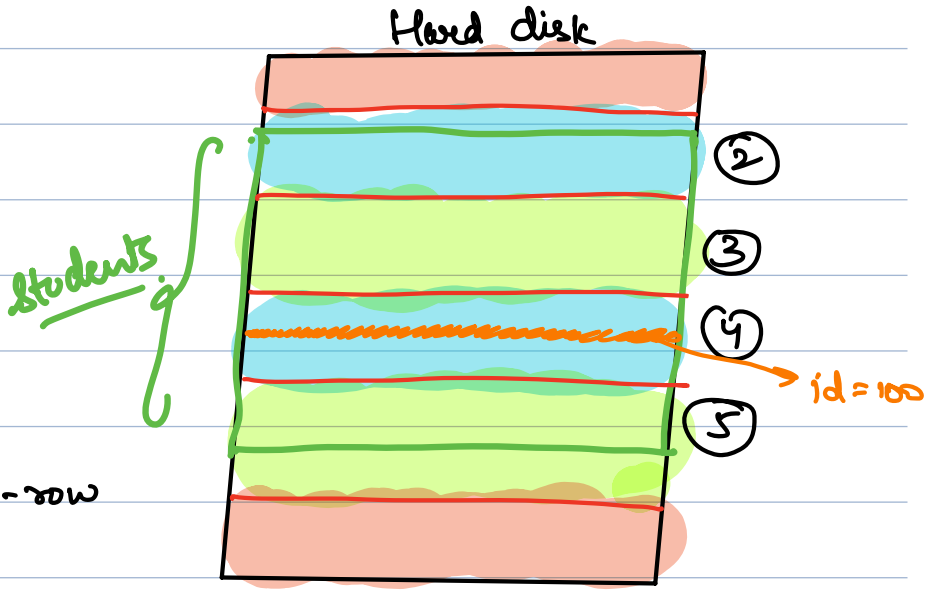
Memory (entire block)

↑

CPU → always fetches data from RAM.

If system wants data from disk. It is first loaded into the memory and then read by CPU.

Select *
from students
where id = 100



Brute force way for this?
Cant traverse row-by-row on disk

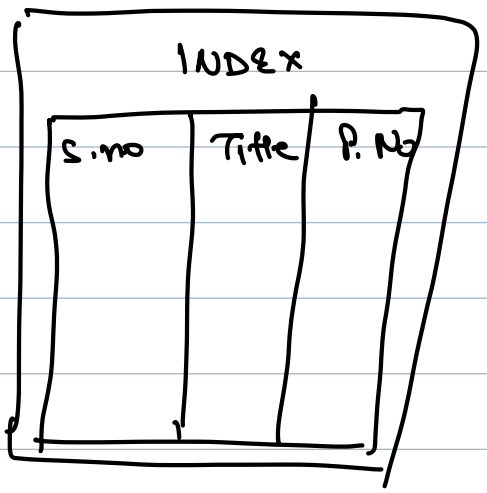
(1) bring block 2 to memory
(2) check all rows if they contain id = 100;

next block

NOT OPTIMAL

⇒ Unnecessary blocks that are being fetched are a waste of performance.

where do you see index in day-to-day life?



CLRS

800 pages

⇒ At the end of a good book, there is an index page, which contains where exactly topics are present.

Index in a books helps find page faster
Index in dB ⇒ finds blocks of disk faster

↓

blocks that contains the rows that we desire.

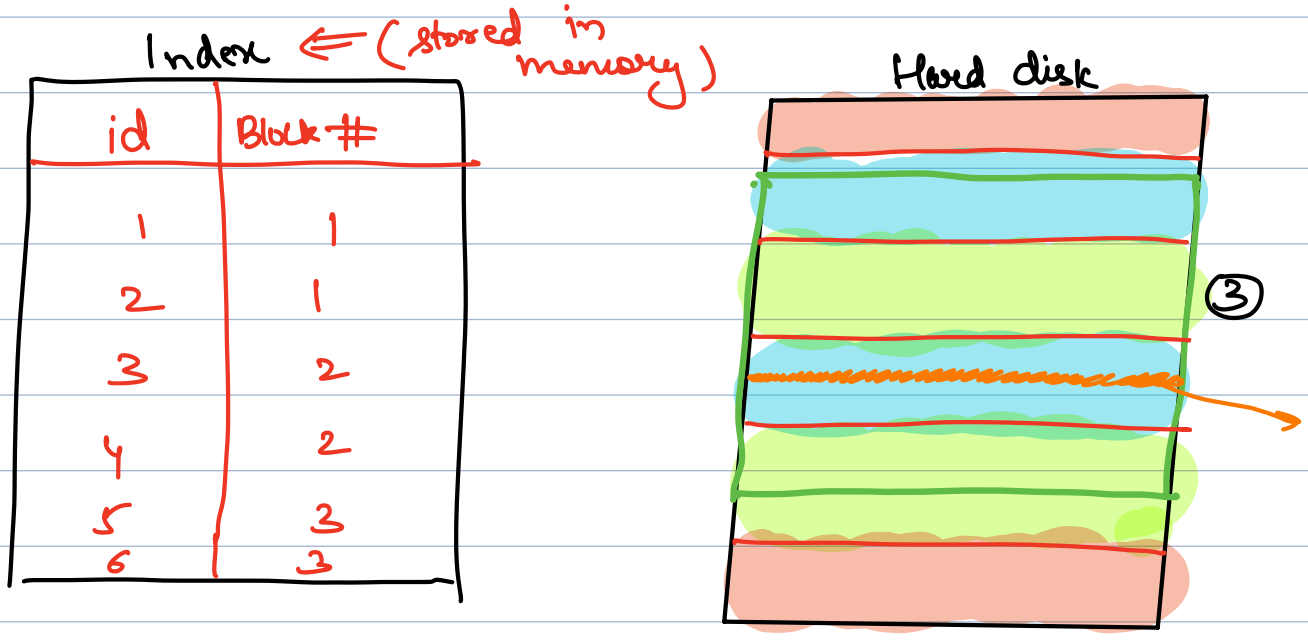
Myth - Indexes dont sort data on the disk

Purpose of indexes : Reduce no. of disk block accesses to fetch data.

HOW INDEXES WORK

Huge table 100M rows.

select * from students where id = 100 ;



Map / HashMap

database that tells me student with id = 100 is in this data block directly.

(key / value pair)

without hashmap → 4 blocks

with hashmap → 1 block

Q Do indexes help with non-unique columns?

```
Select * from students
    where name = 'Ujjwal';
```

students

id	name	psp	
1	Ujjwal	80	Block 1
2	Amit	70	
3	Ujjwal	100	Block 2

Index

name	~~block #~~ list (block)
Ujjwal	(1, 2)
amit	1
alok	(1, 3)
deepak	[2, 4, 6]

```
Select * from students
    where name = 'Ujjwal';
```

without index → 6 blocks

with index → 2 blocks

Till now we have done indexes for queries like

```
where x = y;
```

⇒ there are also queries about ranges

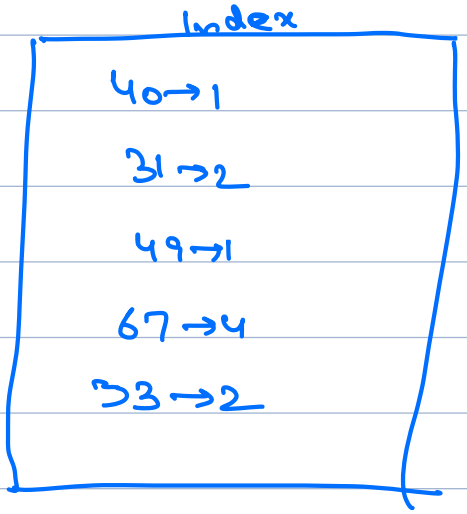
eg find all students with name >= 'M' and name <= 'O';

Starts with M, N, O

eg. find all students with psp between 60 & 70.

↓

Can a hashmap optimize it.



hashmap <int, string>

get all keys with value ≥ 20 and $<= 40$

```
for i =20 →40 :
    if i in map:
        map.get(i);
```

since psp is float type

we miss values like

20.1, 31.2 etc.

With a hashmap it is very tough to get all values within a range.

HashMap → check a value ⇒ O(1)

get all values in a range → O(N)

range queries are like [L ..... R]

① Sorted

② Key : Value → Block ID

Key → Sorted

Value → Map

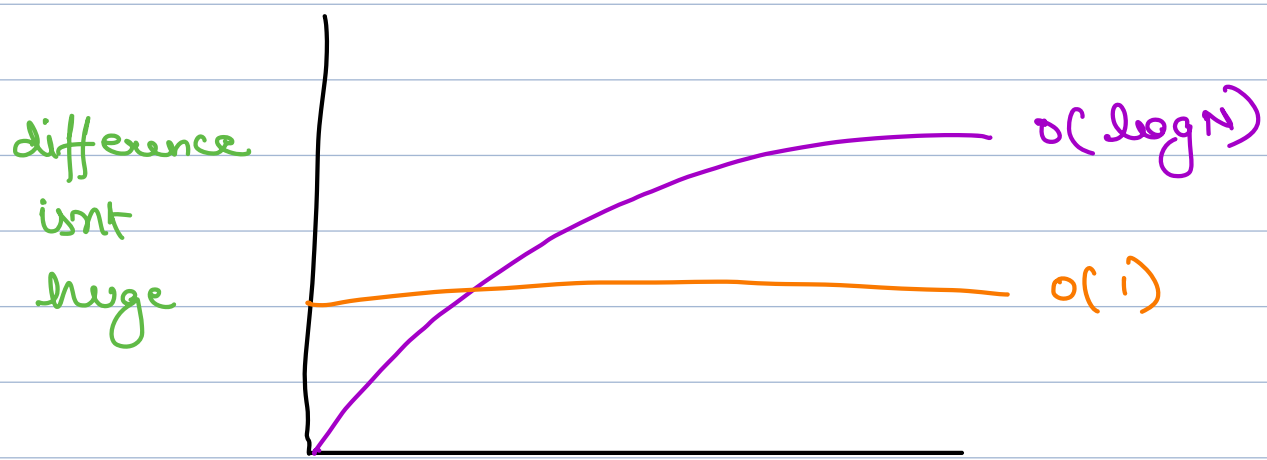
order_map

Treemap

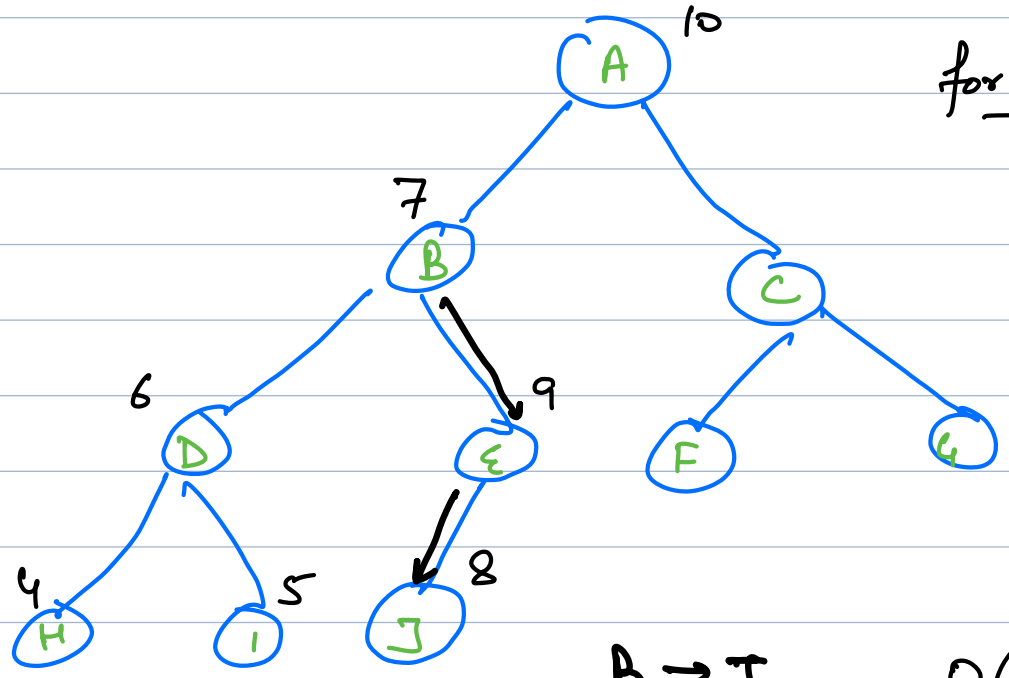
Sorted map

They work on Balanced BST

Check a value → O(log N)



get all values in a range.



for BST.

given a node get to the next bigger node

B → J O(log N)

40 → 60

① go to 40

② Keep going to next node till you go > 60.

Indexes work very similar to TreeMap.

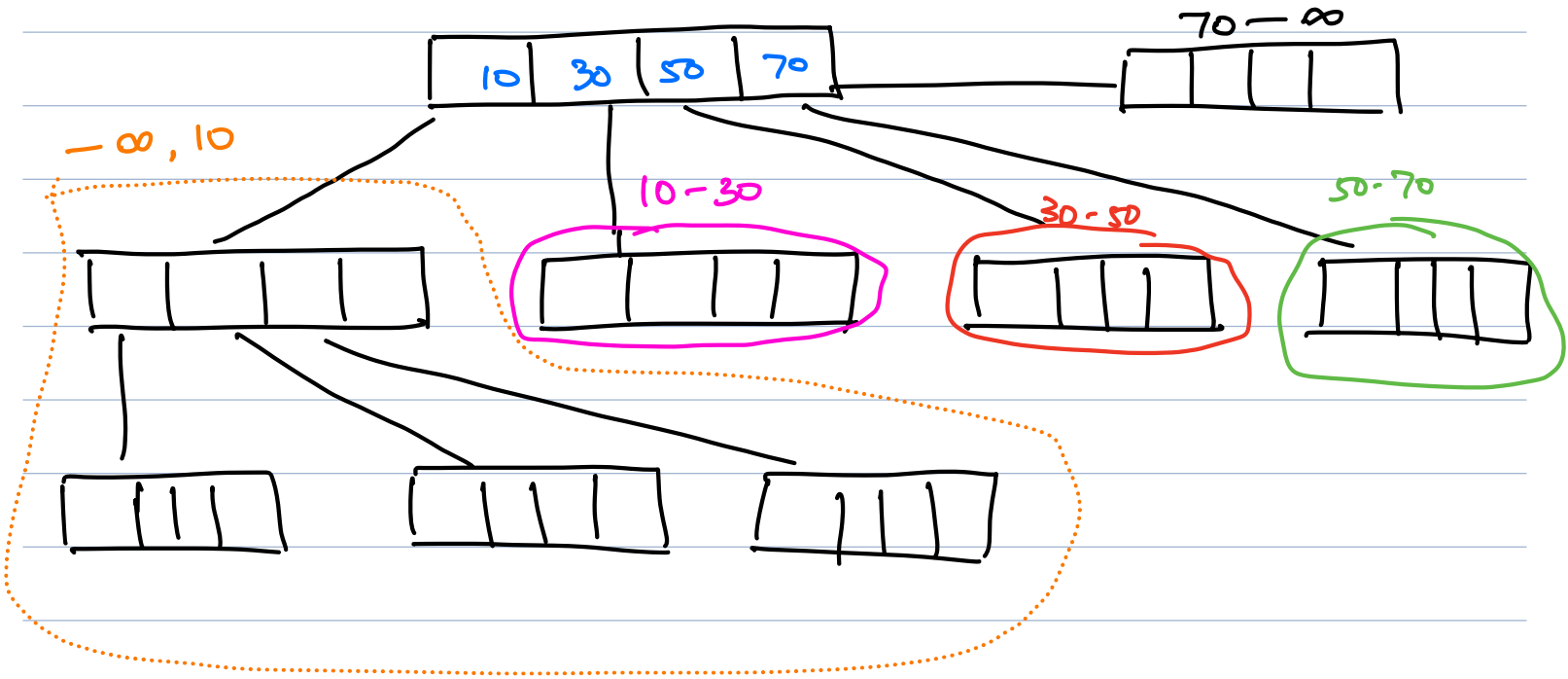
↓

DS that is used to reduce no. of disk access to increase performance of SQL query.

Tree maps use → AVL or RBT (eg of BBST)

★ Indexes use data structures called B|B+ Trees.

→ instead of every node having ≤ 2 children
every node can have ≤ x children.



⇒ Because a B tree has many children
⇒ The height of B tree will be lesser.

⇒ O(H)
⇑
all of them will be faster for Btree → algorithms like height, find next etc

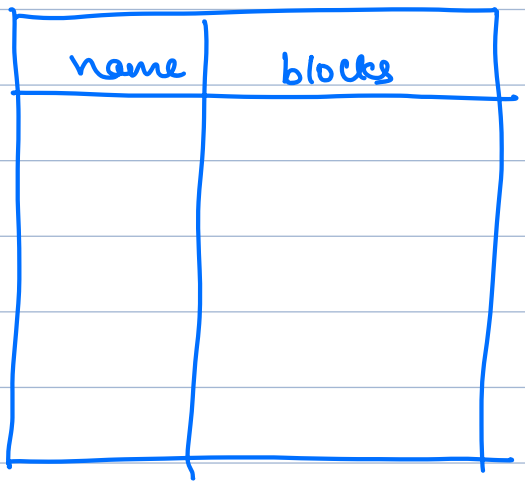
checking a value → O(H)
getting in a range → O(H)
} faster

What you need to know :-

1. How indexes work
2. What is benefit of index
3. Why hashmap is not good.

Cons of indexes

When will you put data in a map.



index.

C → ✓
R → X
U → ✓
D → ✓

which operations will require some changes / updation in the index.

Write operations
(C, U, D) → Change the data on the disk

1. Writes are slower

we wont keep indexes only in memory.
Typically a copy of index is also stored on disk.
↳ Now for a write you will also need to update this index, making writes even slower

② Storage requirement increases.

★ Dont create indexes prematurely.
→ Create an index only when you see the need.

↓

Check performance metrics
⇒ avg time of exec^n of queries.

Indexes on multiple columns

(id)
(name)

(name, psp)
(name, psp, rank)

} We can create an index on multiple cols as well.

Assume I have an index on (name, psp)

Following query.

```
select *
from students
where psp = 80;
```

Will the above index help? NO.

★ If you created an index on (name, psp)

#

creating an index on (name)

and

creating an index on (psp)

You sort by 2 columns means

Sorted by name

↓ psp → only tie breaker

Index (name, psp)

(Abhishek, 70)
(Abhishek, 90)
(Abhishek, 100)
(Bipin, 30)
(Biswa, 71)
(Ujjwal, 20)

sort by name & resolve clashes by psp.

(psp, name)

(20, Ujjwal)
(30, Bipin)
(71, Biswa)
(70, Abhishek)
(90, Abhishek)
(100, Abhishek)

If you have an index on (name, psp) it won't help a query on only psp.

(psp, name)

→ v/s if you have an index on (psp, name) will help.

SCENARIOS.

query on	index on	Help	Not help
① name	psp		✓
② name	(name) (psp)	✓	
③ name	(psp, name)		✓
④ name	(name, psp)	✓	
⑤ name = x and psp = y	(psp, name)	✓	
⑥ name = x or psp = y	(psp, name)	✓	
⑦ name = x			

★ If a query is on colm (x) any index with the x in the prefix will help.

(x, a, b, c) ✓
(x) ✓
(x, a) ✓
(a, x) ✗

Scenario
x = a and y = b

Indexes.
(x) (y)
(x, y)

If this is not there either of them are equally same.

=> SQL finds index with biggest overlap on prefix

X = a and Y = b and Z = c

(X)
(X, Z)
(X, Y)
(X, Z, Y) ✓

★ AND query multiple coln index will help.
OR query multiple coln index WONT help.

Indexes on strings

Users

id	name	email

```
select *
from users
where email = ' '
```

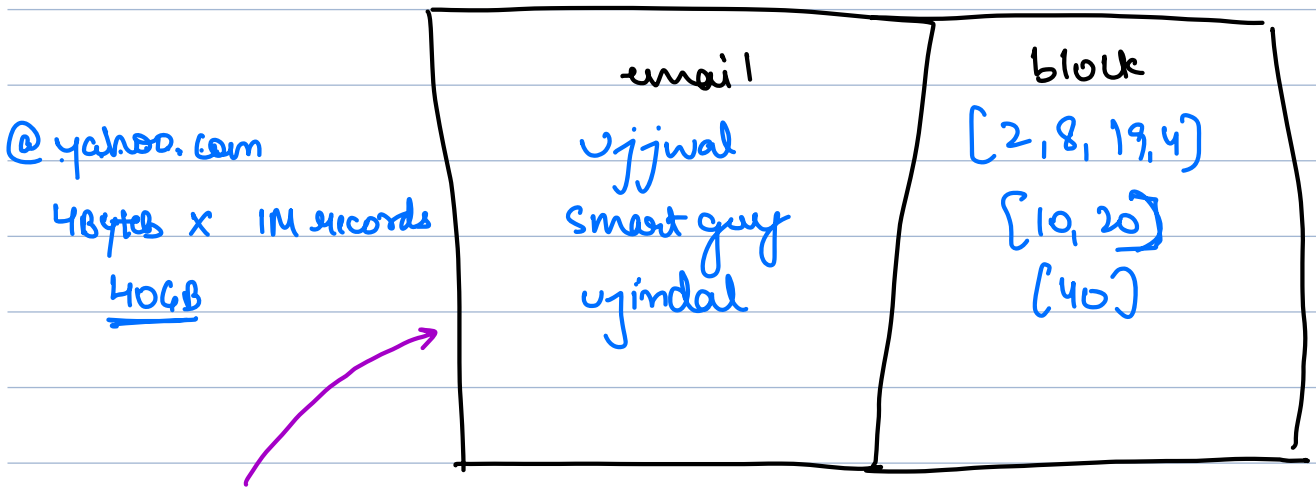
index on email col

email	block
vjjual@scaler.com	2
smartguy@gmail	10
vjindal@gmail	40

① Size of index will be huge
② String matching will be slow

How can we solve this?

Instead of creating an index on complete email, we create index only on part before '@'.



→ Size of the index is reduced.

① Space is saved

select * from users
where email = ujjwal@scaler.com ⇒ will have fetch 4 blocks now as supposed to 1 block when we had index on the whole.

without Index → 250 M
with index on complete → 1
with index on first half → 4

Trade off → Most optimal read or saving 40GB space.

Typically when you have string columns, index is created on a prefix of the coln instead of entire column.

↓

Typically that is enough

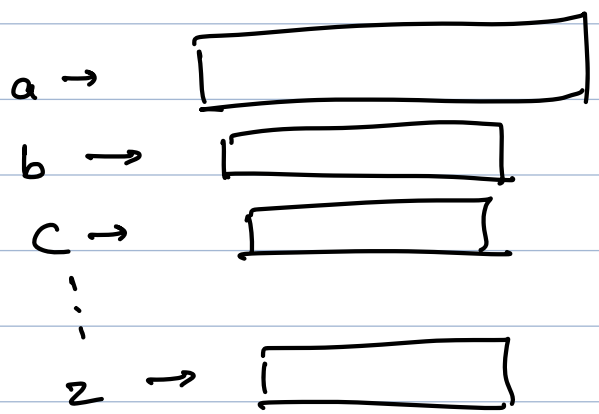
Index on

eg address → address [:5]

countries → countries [:5]

Scaler
⇒ Currently has 2M users / Google 2B users
2×10^9

index on how many characters	# users	# distinct keys in index	disk access
1	2×10^9	26	$2 \times 10^9 / 26$
2	2×10^9	26×26	$2 \times 10^9 / 26 \times 26$



[CODE]

```
for i in range(20):
    print((2 * 10**9)/26**i)
```

if you create an index on the first 7 characters of address, you will have to access 1 block on average.
Reduced space required & equally performant.

Q
```
select *
from users
where address like '%cambala%';
```

↓

will an index on address column
IT WILL BE OF NO USE.

An index on string doesn't help in pattern matching.

↓

out of scope... For this you have FULL TEXT INDEX

How to create index

[CODE]

good naming practice

```
create index idx_film_title_release_year
on film (title, release_year);
```

```
explain analyse select * from film
where title = 'ADAPTATION HOLES';
```

↓

with & without index

```
create index idx_film_title on film (title(7));
```

(slower than complete index
but still faster than no index)