AGENDA

1.) what is index / why / advantages
2.) working of index (B/B+)
3.) cons of index
4.) index on Multiple cols
5.) index on strings

INDEX:

JOINS:

→

```
for i.....M:
    for j.....N:
        if cond match:
            doSomething()
```

✓ slow → O(N·M)

Assume → 1M Rows ...

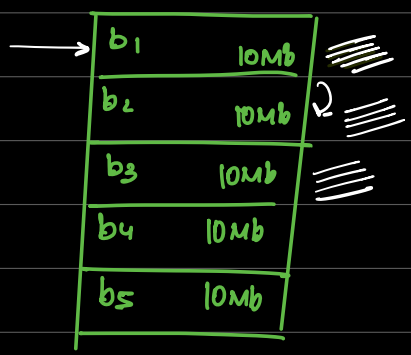
Query is slow.

→ d/b stores data ON Disk.

RAM v/s DISK

✓✓

*) Internal working of DISK:

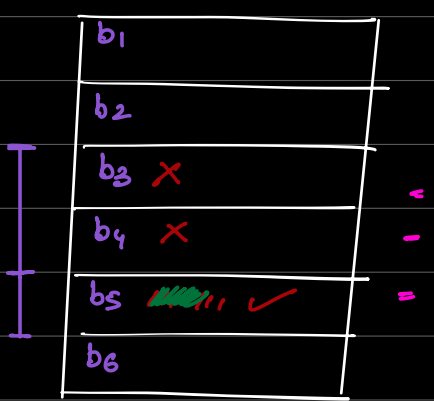


from DISK → MEMORY

MEMORY → executed.

PROBLEMS:

✓ select * from students where i= 100;



d/b → b3/b4/b5

① block contains data.

(DISK)

→ b3 | → b4

MEMORY ← b3 | ←

execute query | execute query

.....

Presence of Idx helps you to speed up work!!

"Indexes sort the data" —— xxxx

Purpose → to reduce No. of DISK Accesses.

*) WORKING OF INDEX:

Assume → 100M Rows (table)

↓

query → select * from stud where id = 100;

yes → PK

id	Name	email
1	A	....
2	B	...
3	C	...
4	D	

(100M) Rows

↑↑

→ using index

DS used ↓

<K,V>

<id, address_block>

<K> <V>

Index → id

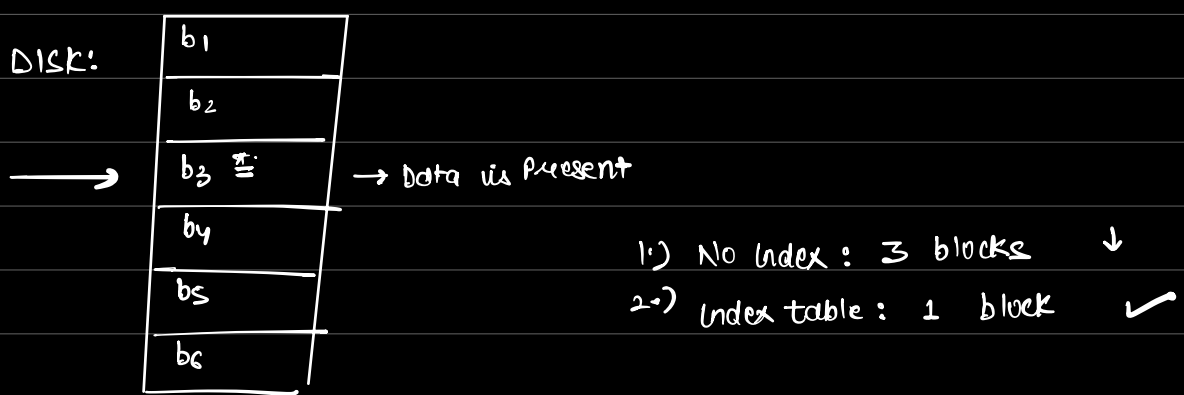
id	value
1	#abc12 → Address of memory block.
2	#abc123
⋮	#91172
100	#aa100bc .....

(stored in memory.)

select * from stud where id = 100;

s1.) before Query execution → Go to Map, Get address of block.

s2.) d/b Gets data from block.



1.) No index : 3 blocks ↓

2.) index table : 1 block ✓

'id' ⟶ <K> index table.

| id | Name | email_id | .... |

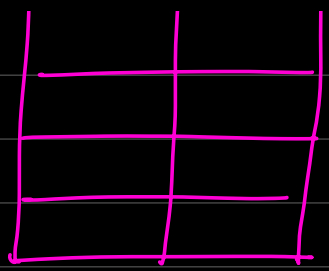
Name→

" select * from stud where name = 'yash';

10 yash

key	value
yash	#1, #2, #3
xxx	

value: list<string> = address blocks.



<Yash, [odd1, odd2, ddd3 ...] >

→ still Not fetching all blocks ....

<key, [] > → work for every query ??

3.) Range based queries

id	Name	Psp (*) → index
1	A	10.0
2	B	10.1
3	C	10.2
4	D	10.3
		⋮

Index table

10.0	b1
15.2	b2
21.7	b3
70.1	b3

select * from stud where
Psp > 10 and Psp < 60;

for i = 10 .... 60 :
// Go to index table
⋮

10 – 60

10.6 10.6

10.0
10.1
10.2
10.3
⋮

10:7 11:0

this DS: <key, [values]>

✗ won't work.

**) Reasons:

→ HashMap based indexes won't work.
→ fail on Range based queries.

solutions:

Requirements for index:

*) data to be ordered. → (index table)
i.e. (key)

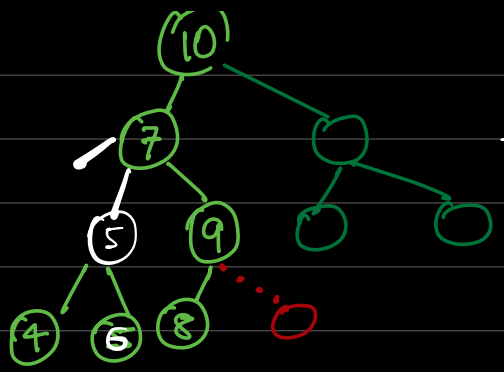
value → block_address.

<k, v>
↓
(sorted)

Treeset / ordered_map → solve
✓ keys sorted
✓ stores <k, v>
TC: O(logN)

O(1) v/s O(logN)

Internally they are BST. (balanced bst)



7-10

log(N)

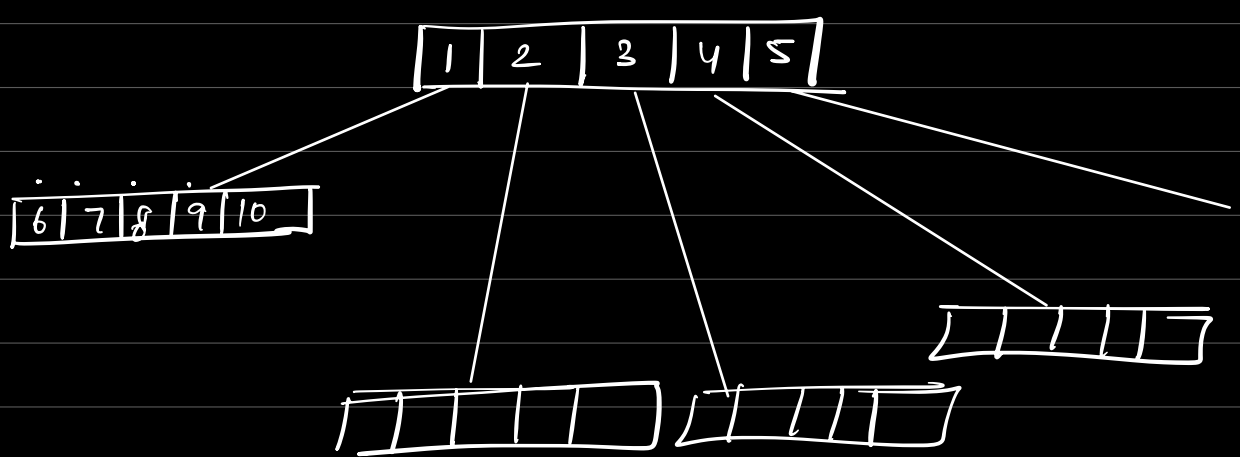
(height)

Indexes ⟶ internally work on B. BST.

TreeMap ⟶ internally uses B | B+ trees

exact B. BST ; just ①

difference: $<x$ Nodes.



B | B+ ⟶ BBST ⊕ every Node can have $<= x$ Nodes.

here $x = 5$

- How index work
- benifits
- Why hashMaps x

10:29 PM IST

*) CONS:

when filled ::

Insert
delete
update
} → write Queries

Read → Read Query.

1.) write Queries are slower
2.) Index table is stored on RAM & copy is stored on DISK.
↓ Extra space.
storage increases.

Main point → Not always use indexing.

*) INDEXING ON MULTIPLE COLS°

id ✓

(id, name) ✓

(id, name, psp) ✓

99% make mistakes.

" select * from stud where psp = 80; "

Idx ON: (name, psp)

⟶ (X) NOT be faster (Not used)

created a index ON (NAME)

select * from stud where name = "yash";

·) sorting names
·) storing index table

	Aman	#1
	Payal	#2
⟶	Yash	#3

v/s

(name, psp) ≠ name and psp ××××××

Group by b.id, psp
↓
??

Similarly in indexes psp → acts as tiebreaker

Name	psp
AMan	70
AMan	90
AMan	80
NaMan	20
Yash	70

Index ON (Name, psp)

AMan	70
AMan	80
AMan	90
NaMan	20
Yash	70

Index (Name, psp) = Index (psp, Name)

20	NaMan
→ 70	AMan
70	Yash
80	AMan
90	AMan

Query ON	Index - ON	WORK / NOT WOR
① Name	PSP	✗
② Name	Name	✓
③ Name	(PSP, Name)	✗
④ Name	(Name, PSP)	✓
⑤ name = x AND psp = 10	(psp, Name)	~
⑥ name = x OR psp = 10	(PSP, Name)	✗✗

Rule of thumb →

Query on col = x → fast

if index starts with (x)

- Idx (Name)
- Idx (PSP)

*) INDEX ON STRING:

users

id email Phone

MOST COMMON → select * from users where email_id = '.....';

email is indexed ✓

email-id	address
abc@scaler.com	#1
yashjainsupercool@scaler.com	#1
yash199@scaler.com	#2

users = 1 BILLION = 10^9

1.) SPACE

2.) search on string.... ×

1B → 10^9

1B Records

40 char × 4 = 160 bytes

4 bytes × 10^9 =

11 char × 4 bytes × 10^9 = 44 × 10^9 = 4.4 GB

whenever working with indexes ON strings →

try to use indexes on 1st 5-6 characters....

Google → 2B users

Indexing on Name

Index on 1st 7-8 charac....

Yash Jain
Yash Jainn
Yash Jaiswal

Name	address
Yash Ja	(#1, #2, #100)
Aman	(#2, #110, #41, #22) → 10 blocks/10

else:

Name	address	
Yash Jain	#1	xxxx
Yash Jainn	#2	too much space
Yash Jaiswal	#3	

a-z 10000

index_on_chars	users count	# distinct keys	# disk access
1	2×10^9	26	10K/26
2	2×10^9	26 26	10K/26x26
3			
4		----	10K/26^4

Query →

" select * from users
where name like %yash% ;

(full text index.)