INDEXING

SELECT * FROM students WHERE id = 100.

O(N) operation

Hashmap

id		value
1	⇒	(1, Ajay, ...)
2	⇒	(2, Saurabh, ...)

~~Score~~

~~id~~ → duplicates?

roll number ✓

(SELECT * FROM students WHERE id > 100 AND id < 10,000)

① Sorted order → Binary search

1
2
5
8

id = 9

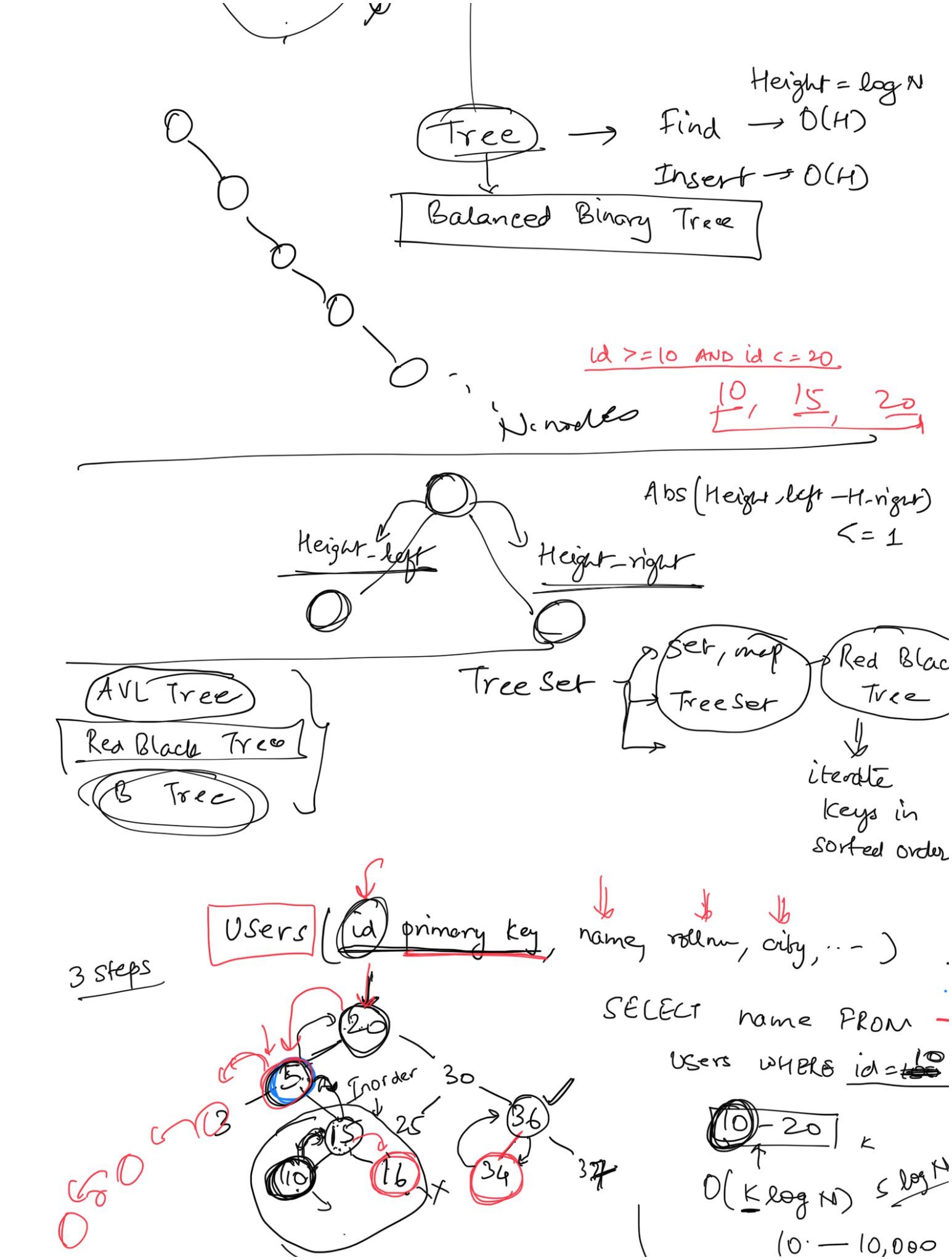
→ insert becomes expensive

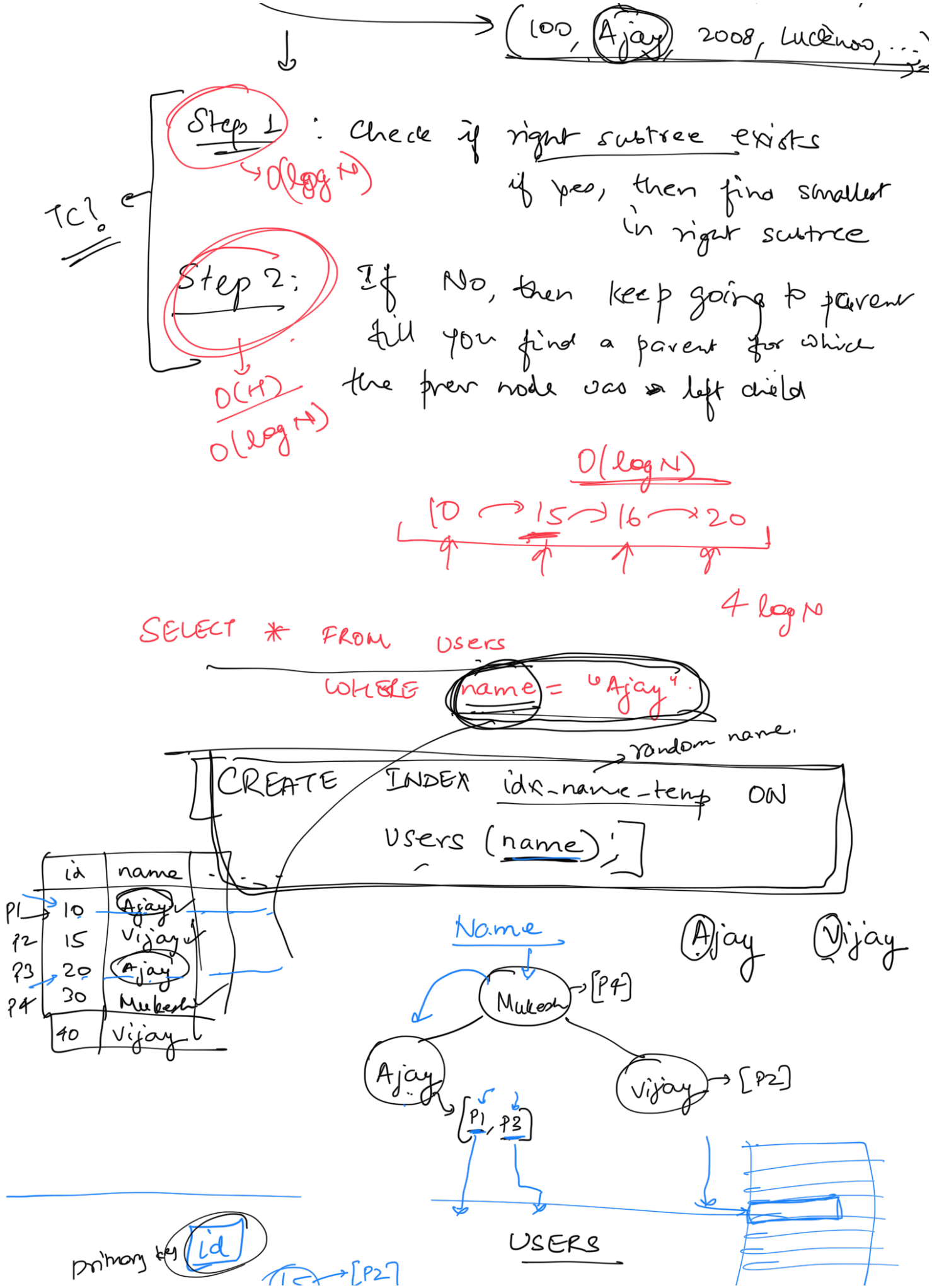
id → Tree → B Tree

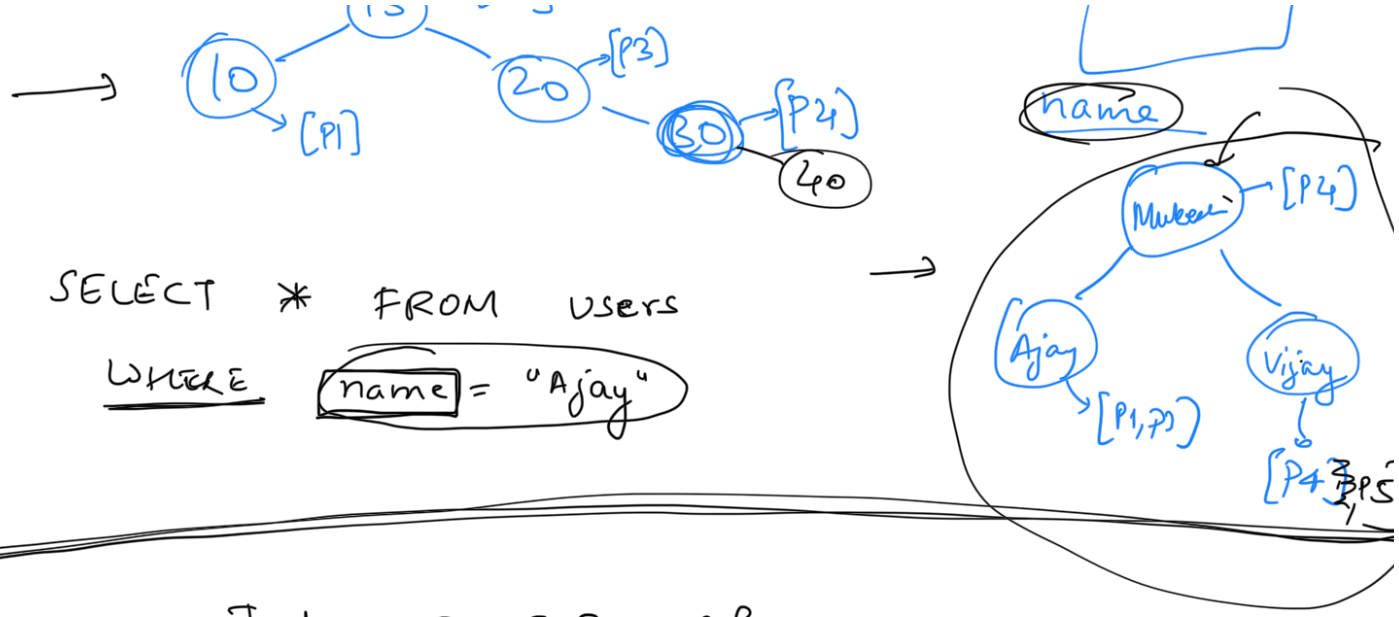
8:1
2:2

✓ Hashmap → X (Range Query)
✓ Sorted (id) → X (Insert)

— TREE


Height = log N
Tree
Find → O(H)
Insert → O(H)
Balanced Binary Tree
id >= 10 AND id <= 20
10, 15, 20
N nodes
Abs (Height_left − H_right) <= 1
Height_left
Height_right
TreeSet
set, map
TreeSet
Red Blac
Tree
iterate
keys in
sorted order
AVL Tree
Red Black Tree
B Tree
Users (id primary key, name, rollno, city, ...)
3 steps
20
5
30
3
25
15
36
10
16
34
Inorder
SELECT name FROM
Users WHERE id = 10
10 — 20
k
O(k log N) ≤ log N
10 — 10,000


(100, Ajay, 2008, Lucknow, ...)
Step 1 : Check if right subtree exists
O(log N)
if yes, then find smallest in right subtree
TC!
Step 2:
If No, then keep going to parent till you find a parent for which the prev node was a left child
O(H)
O(log N)
O(log N)
10 → 15 → 16 → 20
4 log N
SELECT * FROM users
WHERE name = 'Ajay';
random name.
CREATE INDEX idx_name_temp ON users (name);
id
name
P1 10 Ajay
P2 15 Vijay
P3 20 Ajay
P4 30 Mukesh
40 Vijay
Name
Ajay
Vijay
Mukesh
[P4]
Ajay
Vijay
[P2]
P1, P3
primary key
id
[P2]
USERS



- Index on every column

Pros	Cons
1. ~~Search~~ SELECT will become faster	1. Space 2. update → insert, update, deletion

User

id | fna | lna | role | city | email | score | ...

SELECT * FROM film → 100,000

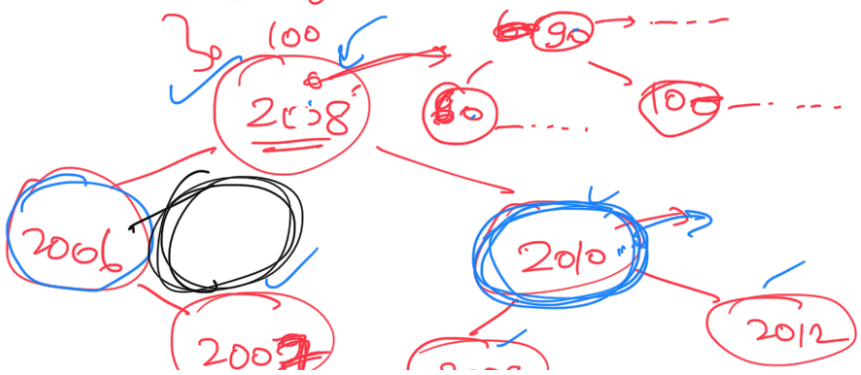
WHERE release_year = 2010 → 50,000 → 2006

AND length > 75

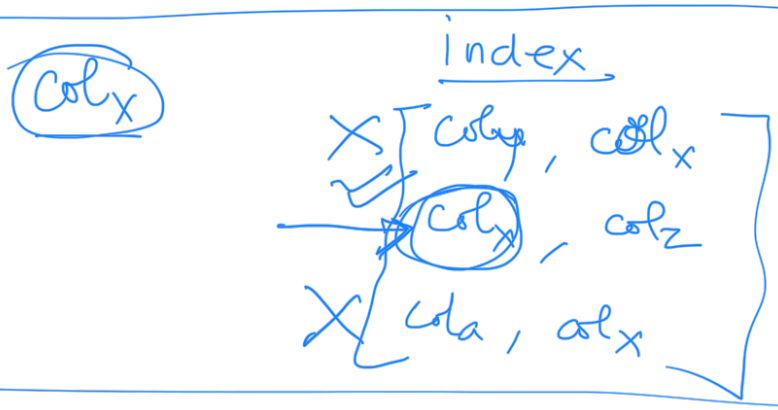
release_year, length.

O(N) query

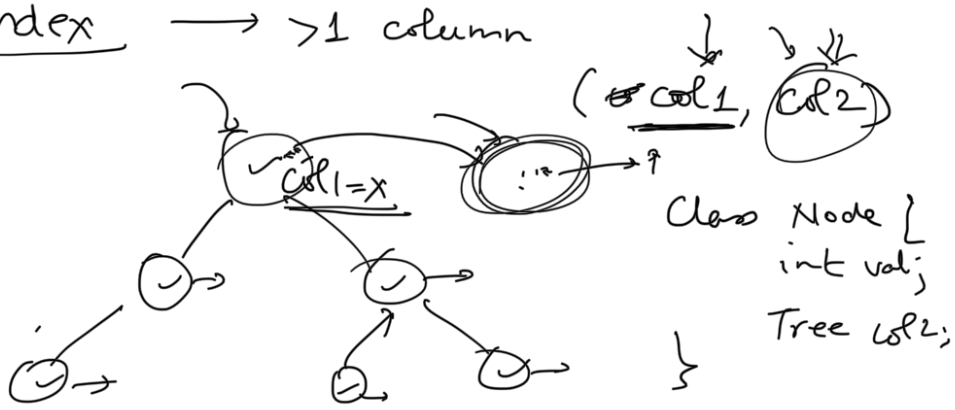
2-D

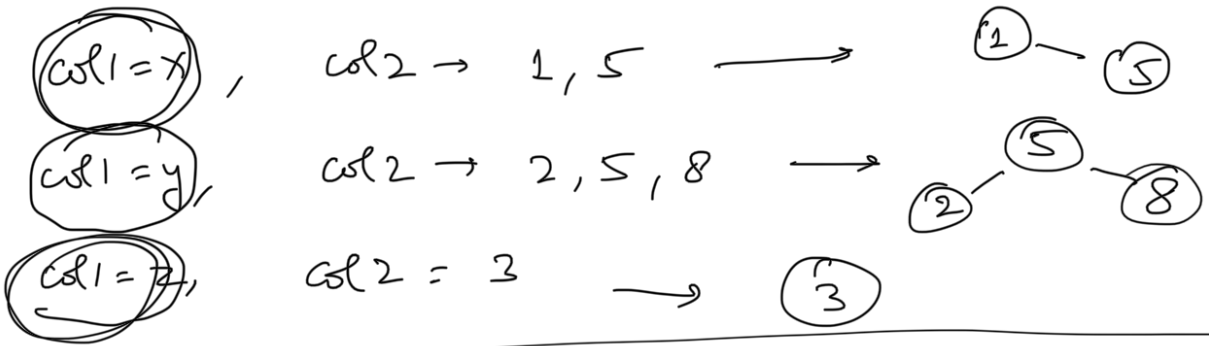



```
SELECT * FROM
film WHERE
release_year = 2006
```



Composite index ⟶ >1 column





① Index ⟶ B+-Tree
↳ Why?

② How to create your own index in a table

③ How does DB decide which index to use?

④ What is composite index?
and given a query, can I use that composite index?

Table

↓

list of index

index1 → [id] ✓

index2 → [name] ✓

index3 → [email] ✓

index4 → [rollno, seatno] ✓

index2 ✓
index4 ✓ ⇒ index2

SELECT - - -

WHERE

id = xyz

SELECT - - -

WHERE

name = - -

AND rollno =

=

Database

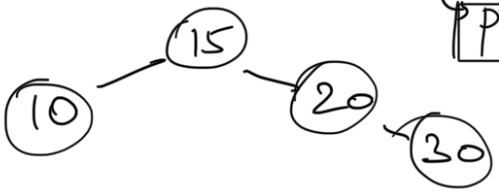
EXPLAIN (- - - - →)

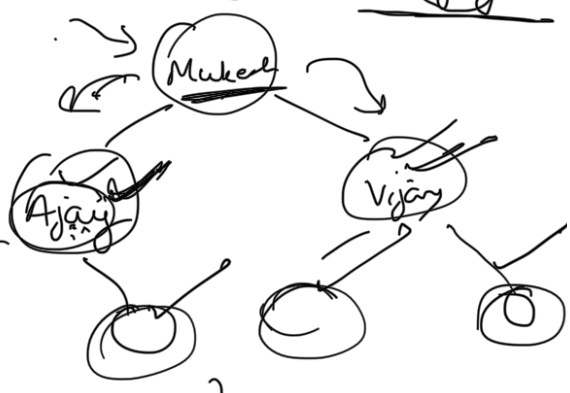
id | name = "Jay%"

BST name

prefix search

name LIKE "%jay%"





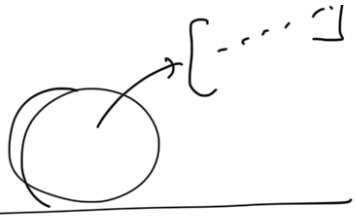
LIKE "%...%"

→ Full Text search

→ O(N)

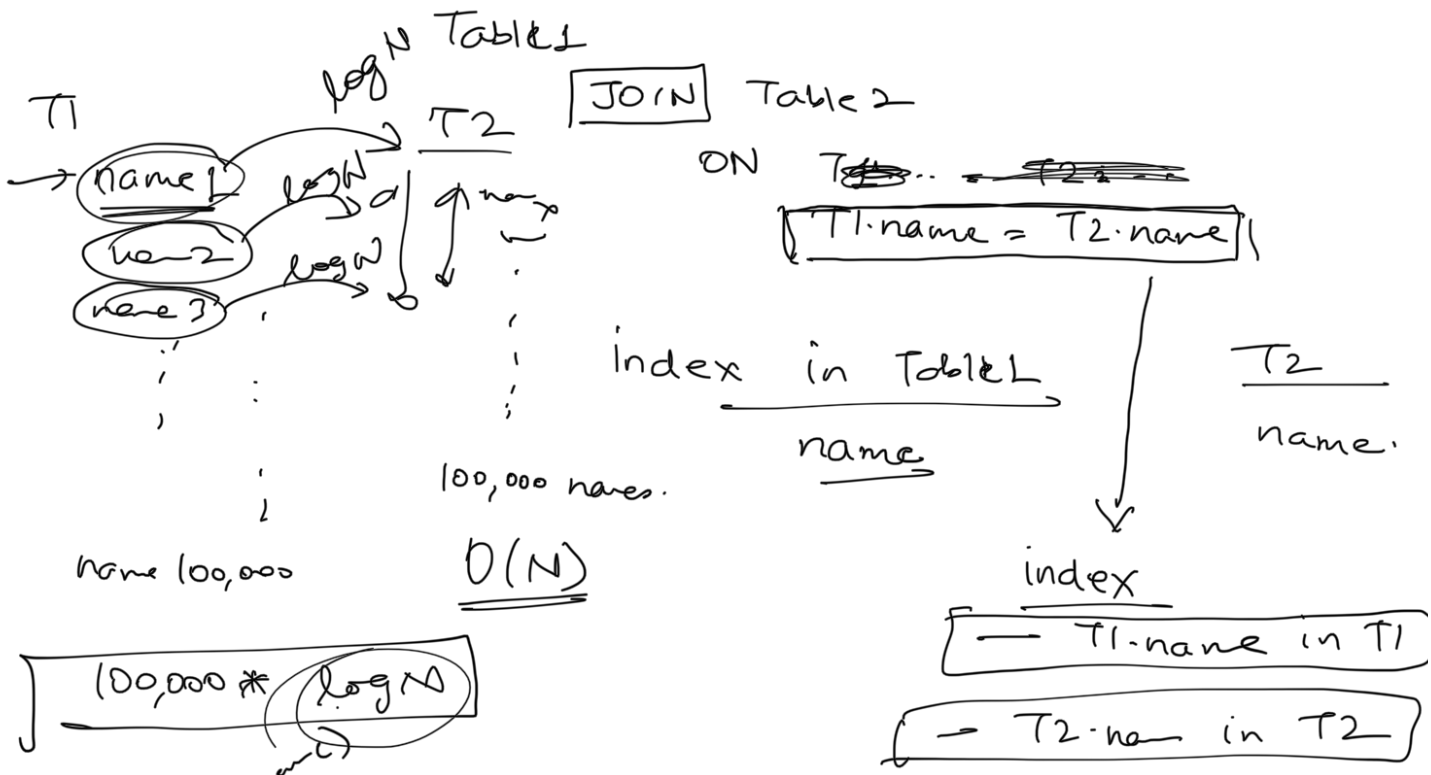
→ SLOW

name LIKE "a%"



HD ⟶ Query Optimisation

↓



for i in left_table.names: ↙ index or

find i in right_table

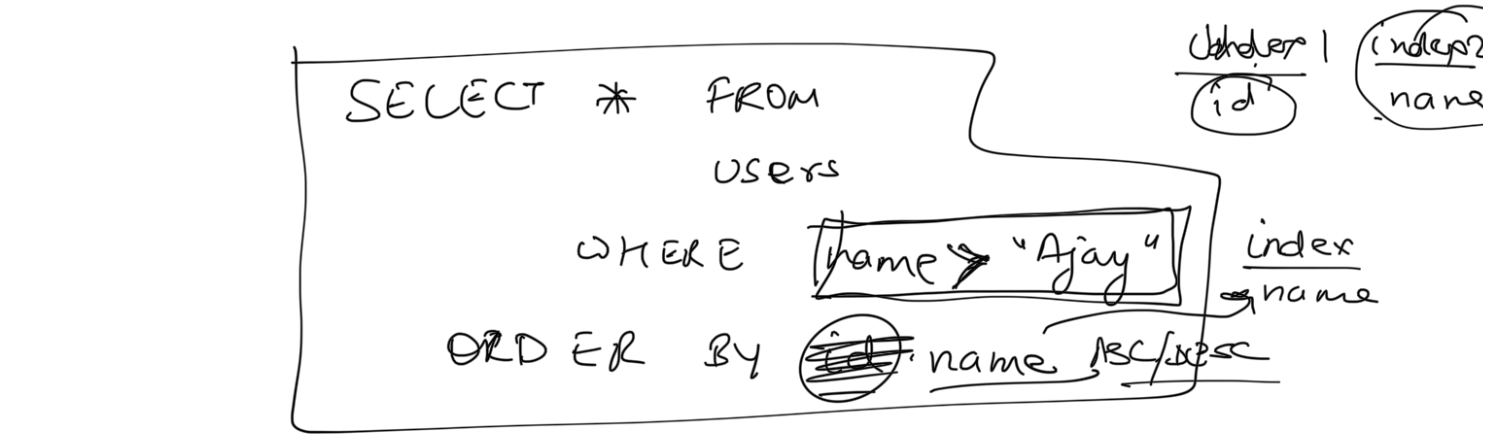
① Joins are fast if index is present.
Subqueries cannot use index and are hence slow.

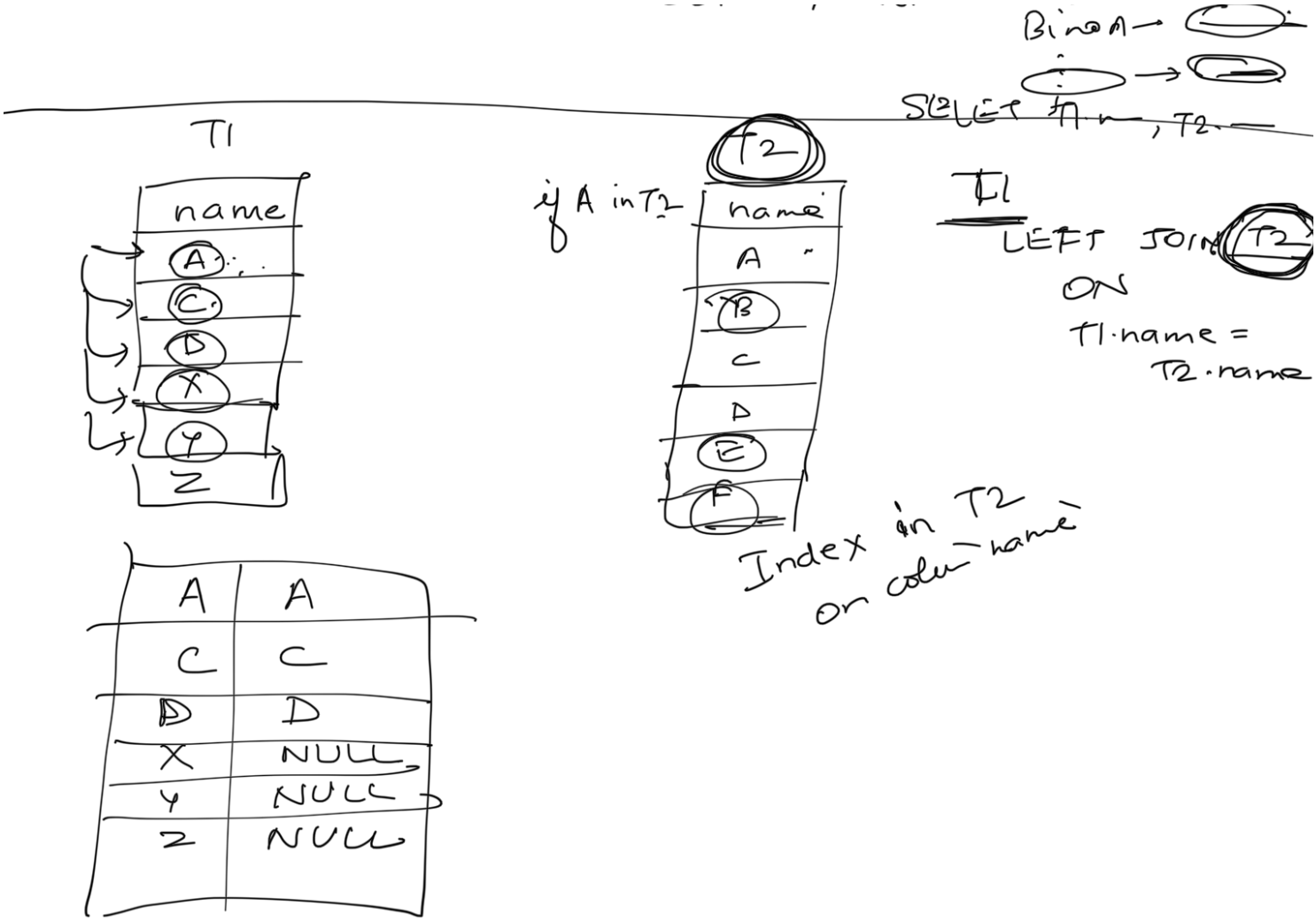
② Create right index

1. Subqueries → Joins which use existing index
2. Re-write query to use index
3. Create index which makes query faster

```
SELECT * FROM
    users
WHERE name > "Ajay"
ORDER BY name ASC/DESC
```

index: name




Bino A
SELECT T1.name, T2.
T1
name
A
C
D
X
Y
Z
T2
if A in T2
name
A
B
C
D
E
F
T1
LEFT JOIN T2
ON
T1.name =
T2.name
Index in T2
or colu-name
A A
C C
D D
X NULL
Y NULL
Z NULL