Agenda:

→ What is a Database
→ Why we are learning DB at Scaler.
→ Type of Databases.
→ Intro to Relational Databases
→ Intro to Keys.

MySQL Databases.

Install MySQL Workbench + MySQL DB.

What is a Database :-

Data :- Information.

Excel / Word / Notepad / ....

Name	Phone no
Abc	1234
XYZ	5678

⇒ Scaler's DB

→ Students

→ Instructors

→ Students.

Name	email	batch	psp	attendance
Nikhil	abc@—	123	97	95

batch : 123.

↳ Name : Mar 23 Intermediate

→ start_date : 25th Mar.

→ Current Module : SQL.

⇒ Students, batches, Instructors, ...

Students.csv

batches.csv

instructors.csv

Companies.csv.

Students.csv

Name	email	batch	psp	Attendance	coins	rank
→ Parth	abc	123	45	94	1000	5
→ Himanshu	def	432	96	92	2000	4

Ex : Avg psp of batch = 123.

→ Iterate over the file row by row.

⇒ Issues of Storing data in files.

1) Inefficient.

2) Data Integrity.

Student.txt.

Name	email	batch	PSP	Attendance	Coins	Phone
Parth	abc	123	45	94	1000	abc.
Himanshu	def	432	96	92	2000	
—	—	—	—	—	—	

CSV.

Deepak, Kasera, 7015608331, - - - - -

↑ first. ↑ second ↑ Phone . . . . .

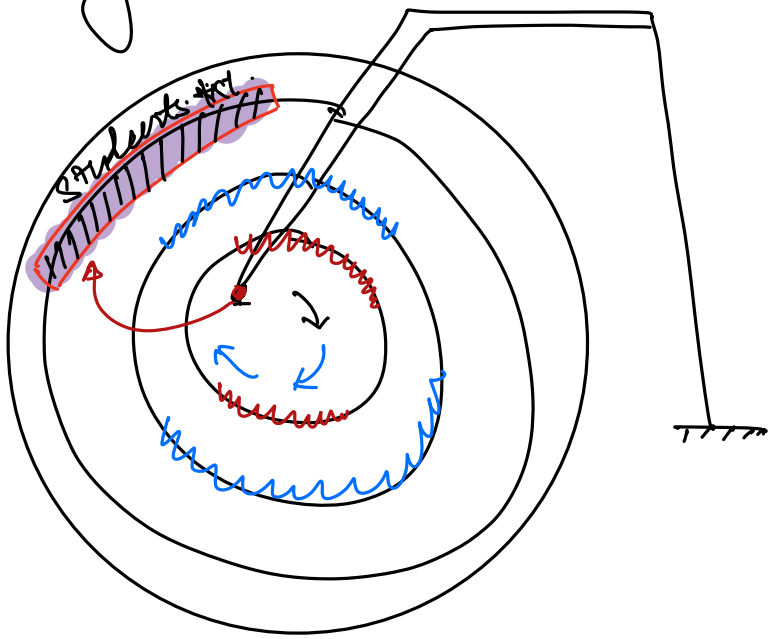
Rahul, Nandanwar, - - -

↑

Second. ? ✗

File → HDD.

3) Concurrency.



→ Only one user can read the data from HDD at one time.

⇒ Concurrency

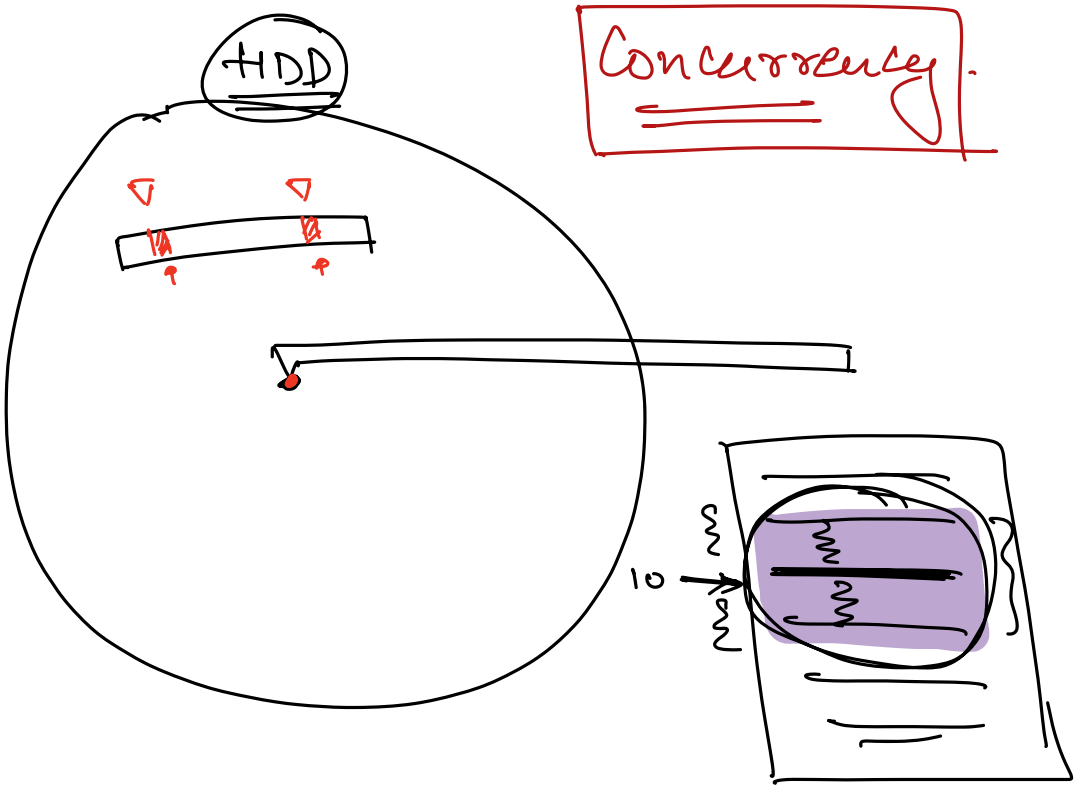
⇒ Not fast enough.

File

IOM

↓

Concurrency.



4) Security.

→ Role based access — Read, Write

⇒ Database

Airbase : Place to Store aircrafts.

Database : Place to Store data.

⇒ DBMS. : Database Management System

Type of Databases.

→ * Relational Databases.

→ Non Relational Databases.] HLD.

- → Graph based (GraphQL).
- → Mongo DB
- → Columnar DB.
- → Key-value DB.

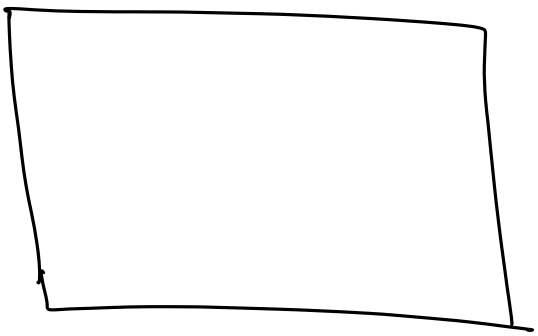
Relational Databases.

→ Collection of Tables

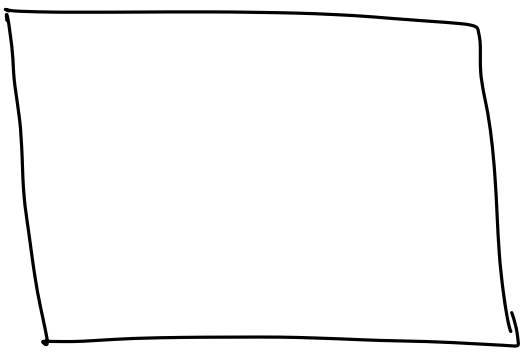
entity

⇒

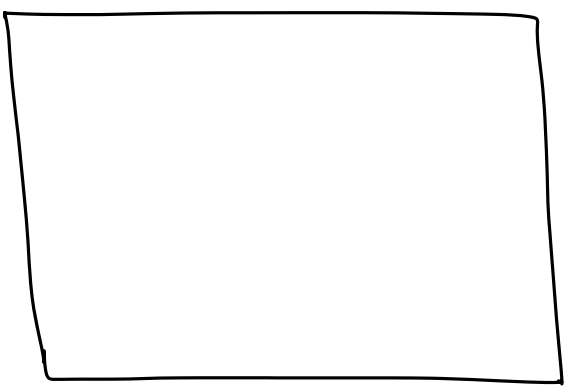
Students



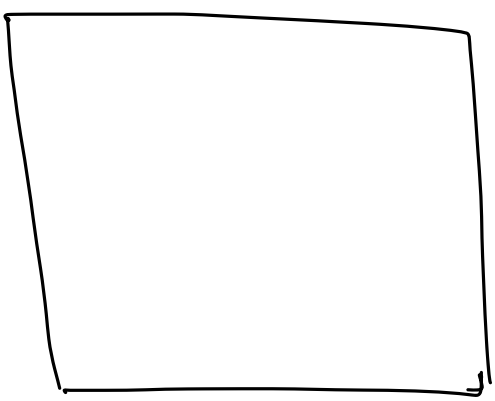
batches



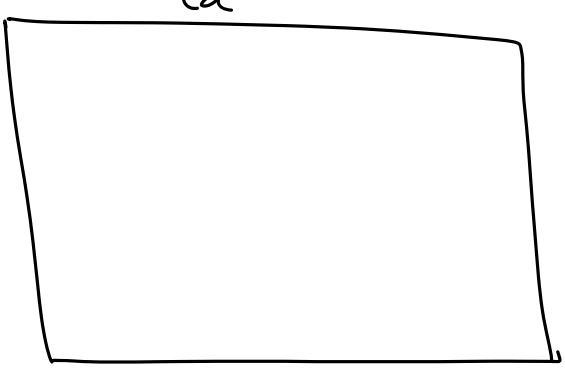
instructors



mentors



ta



. . .

⇒ Mathematics.

⇒ Sets.

⇒ Properties of Relational DB:

1) Every row is unique.

Students

Rollno	Name	PSP
1	Deepak	95
2	Deepak	95
⋮		

2) All the values present in a column should be of same datatype.

3) Every cell should only contain atomic value.

↓

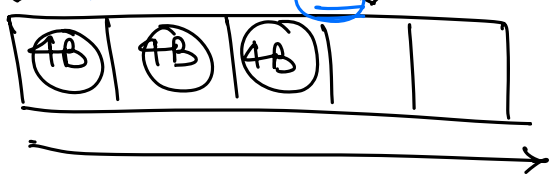
Single / Can't be divided further.

→ Array, List, Map ✗

⇒ Efficiency.

4B.

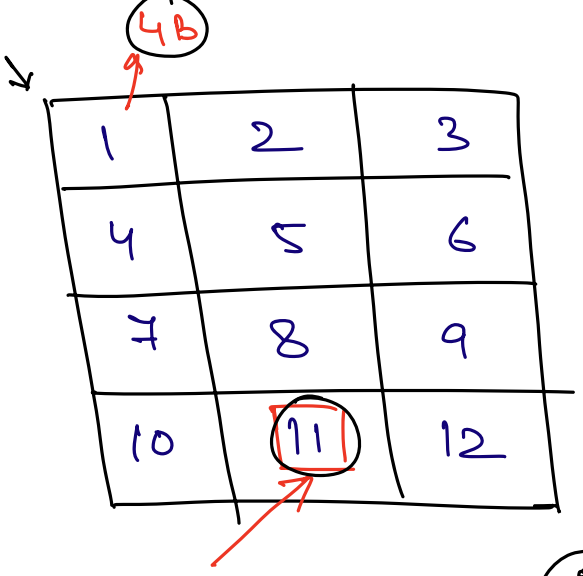
int.

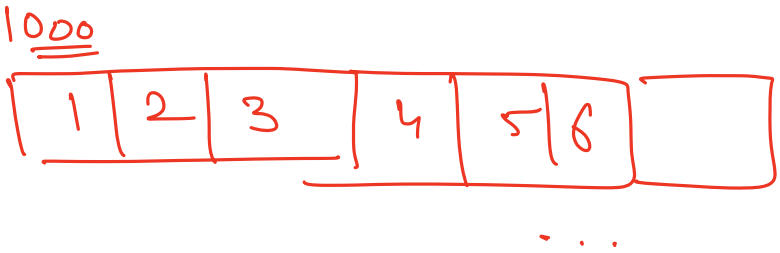


O(1)

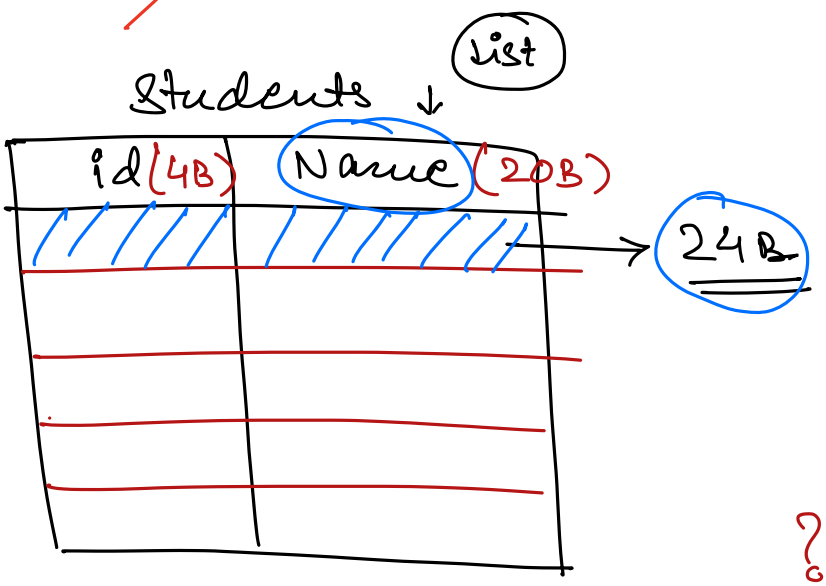
100 + 4 * 3

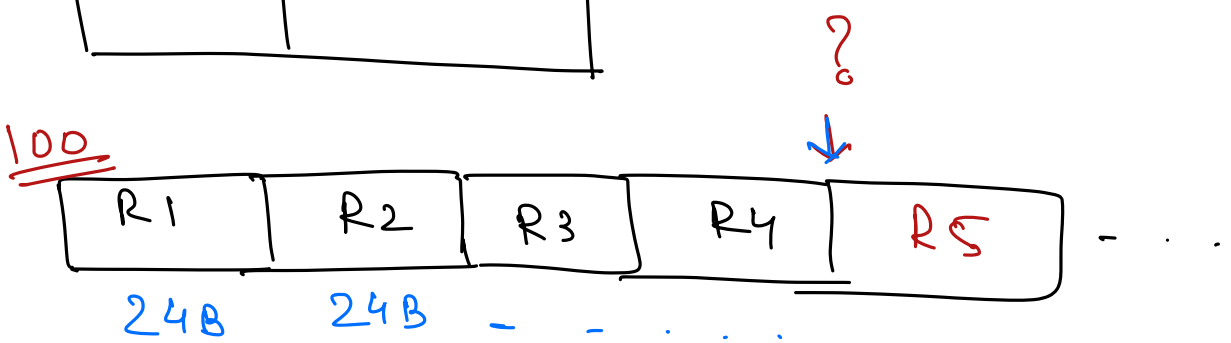
112





1000 + 10×4 ⇒ 1040.





100 + 24×5 = 220

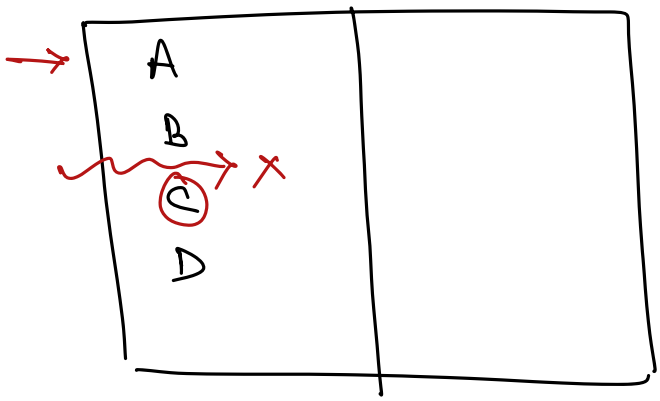
4) Order of columns is not guaranteed.

id	Name	age	batch	rsp.
50	Imran	25	1	95

⇒ Select * from students where id = 50.

⇒ Select Name, age, batch, rsp from students where id = 50.

5) Order of rows is not guaranteed.



ORDER BY. ←

7) Column names should be UNIQUE.

Name	~~Name~~
Ponujal P	Join

Select Name from Students.

⇒ KEYS.

Students

Name	age	PSP
Venkatesh	25	95
Venkatesh	25	95

Key: Column (or) a group of columns that can uniquely identify a row.

→ Super

→ Primary

→ Candidate

→ Composite

→ Foreign.

SUPER KEY :

A column (or) a group of column that can uniquely identify a row.

Students

name	PSP	email	batch	Phone-no

1) name ✗
2) batch ✗
3) email ✓
4) Phone no ✓
5) (email, phone) ✓
6) (name, email) ✓
7) (name, phone) ✓
8) (phone, PSP) ✓
9) (batch, PSP) ✗

Agenda:

=> KEYS

- Candidate Keys
- Primary Keys.
- Composite Keys
- foreign Keys.

=> Introduction to SQL.

CANDIDATE KEYS.

Students

name	PSP	email	batch	Phone-no

1) name ✗
2) batch ✗
3) email ✓
4) Phone no ✓
5) (email, phone) ✓
6) (name, email) ✓
7) (name, phone) ✓

SK

8) (phone, psp) ✓

9) (batch, psp) ✗

⇒ In many of the Super keys, there are columns that don't really play any role in uniquely identifying a row in a table. Even without these attributes we can uniquely identify a row.

Candidate key:

⇒ Super key of minimum size.

⇒ Super key where if we remove any of the attribute, then remaining part will not be a Super key.

⇒ Super key where all the attributes in that are required to uniquely identify a row.

(email, phone) → SK ✓

↓

CK ✗

(name, psp, batch_id) ⇒ SK ✗

Garima, 90, 123

Garima, 90, 123

Ex

instructors

first_name	last_name	salary

Instructors can't have same name.

(first_name, last_name, salary) ⇒ SK ✓

↓

CK ✗

(first_name, last_name) ⇒ SK ✓

↳ CK ✓

⇒ Ex : attendance.

student_id	class_id	attendance
1	12	80
2	12	90
3	12	70
1	14	85
5	90	95
6	91	95
1	20	85
10	20	85

Super keys. SK

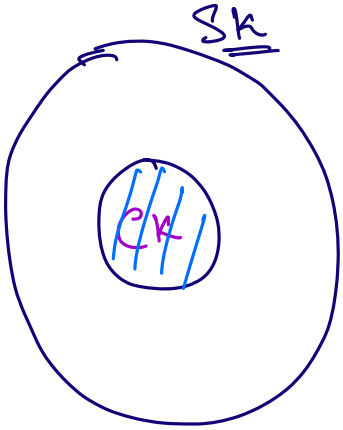
1) Student_id ✗
2) Class_id ✗
3) (Student_id, Class_id) ✓
4) (attendance) ✗
5) (Student_id, attendance) ✗
6) (Class_id, attendance) ✗
7) (Student_id, Class_id, attendance) ✓
8) (Student_id, Class_id, - - - - -) ✓

⇓

Candidate key

(Student_id, Class_id)

⇒ Super key of Min Size.



CK ⇒ Subset of SK.

⇒ EmpId ✓

Email ✓

(first_name, last_name) ⇒ SK ✗

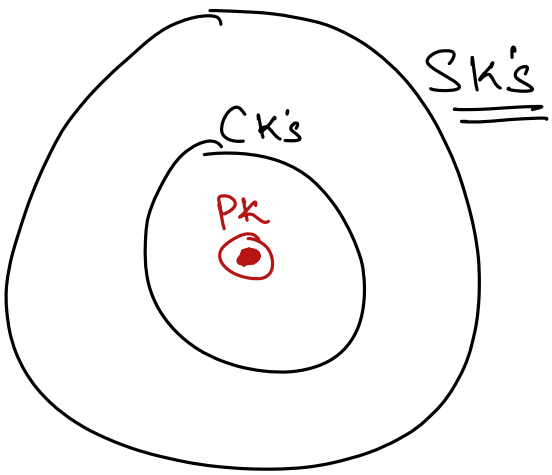
(last_name, department) ⇒ SK ✗ } ✗

⇒ Super Keys ⇒ Party workers.

Candidate Keys ⇒ MLA's.

Primary Key ⇒ CM

} Example.



PRIMARY KEY :

A candidate key selected to be the unique set of columns to help identify a row in the DB.

Students

first_name	last_name	email	batch_id

SK's ⇒ (email, first_name)
(email, batch_id)
(first_name, batch_id) ✗

CK's ⇒ email

⇓

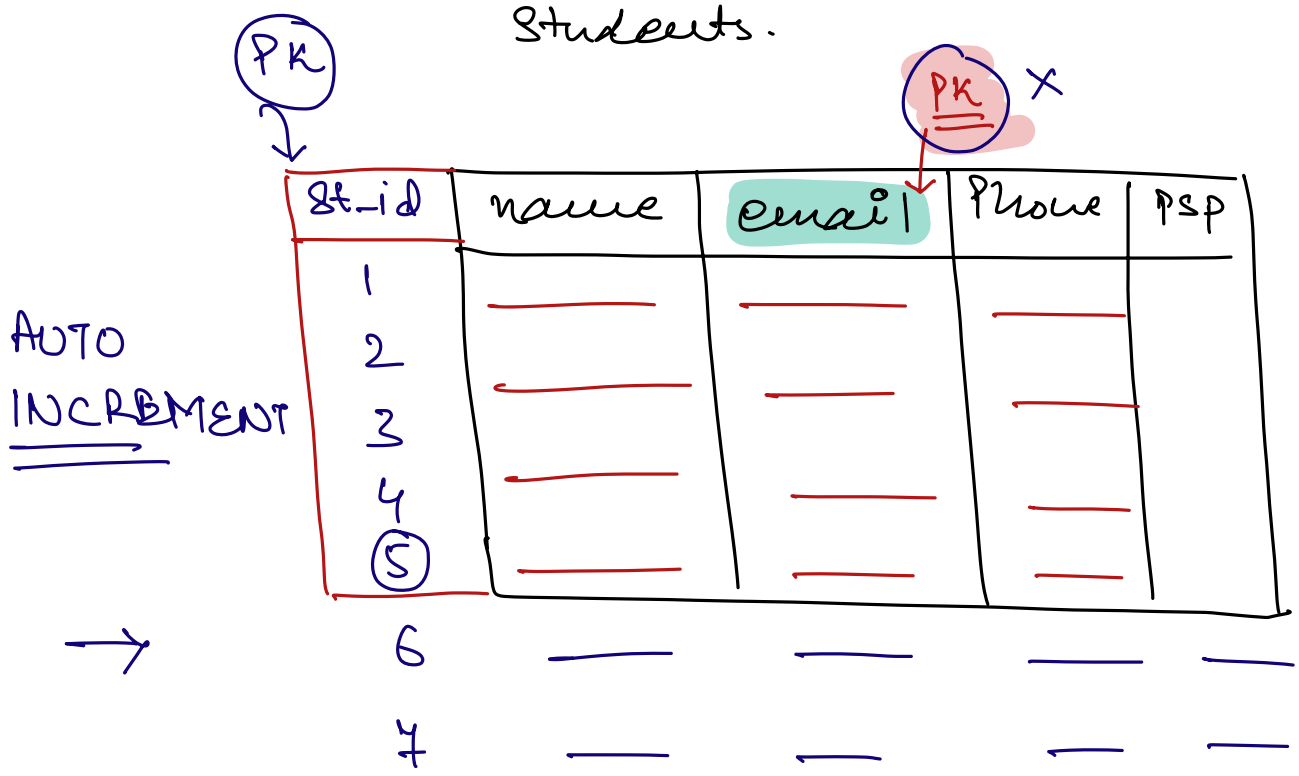
PK.

⇒ When we create a table in the DB, then DB forces us to specify a column or set of columns that will uniquely identify a row in the table.

⇒ PK.

Properties of PK.

1) Database sorts the table on PK.



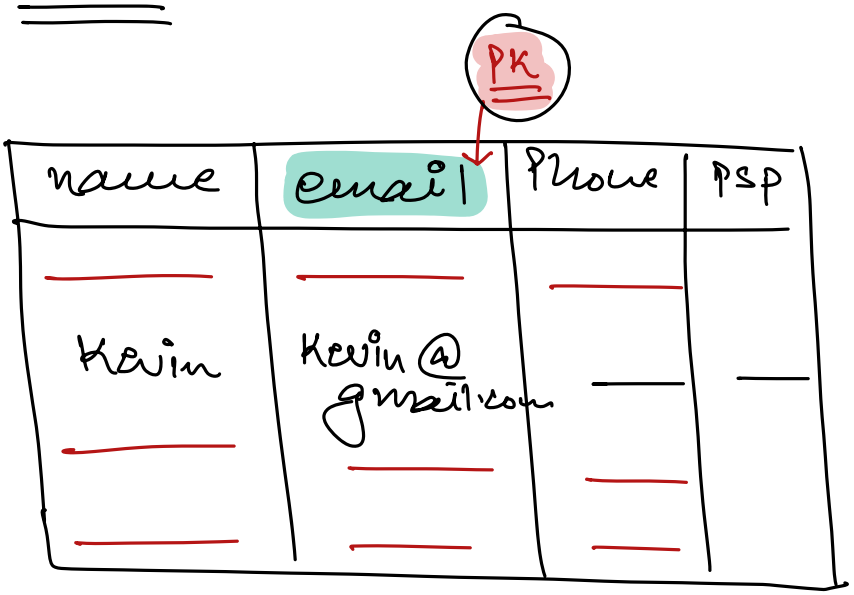
2) Database outputs the data in the sorted order based on the PK.

3) Database creates index on PK by default.

A good PK:

1) Should be easy to sort on.
2) Should be small in size.
3) Should not change.

Students.



Can users change their email_id? Yes.

⇒ Changing the PK isn't good at all for your DB.

⇒ Ideally we shouldn't have user attributes as PK.

⇒ Always it is suggested to have a single integer (defined by the DB owner) as a PK.

Composite Keys.

⇒ Super key with more than one attribute is a composite key

foreign key

Students.

FK.

st_id	name	email	phone	batch_id
1	Parag	—	—	123
2	Venkatesh	—	—	123
3	Sriram	—	—	123
4	Deepa	—	—	456

batches.

batch_id	name	start_date	instructor	no. of students.
123	Morning	—	—	100
	—	—	—	—

A column or a group of columns that helps us to uniquely identify a row in the other table.

⇒

Students

st-id	name	batch_id
1	ABC	100
2	XYZ	10
3	PQR	11
4	DEF	~~12~~ NULL

batches

batch_id	batch_name
100	~~Mar23 Int~~ Mar23 Beg XYZ
11	
12	
13	
14	

(1) If we are adding a student to a batch, the it should exist.

(2) Deleting/Updating a batch.

↓

NO ACTION i) Not allowed.

SET NULL ii) Mark the batch_id for all the students belongs to this batch to be NULL.

CASCADE. iii) Delete/update the students.

⇒ book_marks.

id	name	class_id
1	—	10
2	—	10
3	—	10
⋮	—	10

Classes

id	desc
10	—

Ex

Student

St_id	name	b_id
1	Rahul	10
—	—	10

batches ↓

b_id	b_name
10	~~Mar~~23 Intermediate
	Feb.
11	Mar23
10	
10	
10	

Ex

Student ⇒ 10

St_id	name	b_id
1	Rahul	10
		11

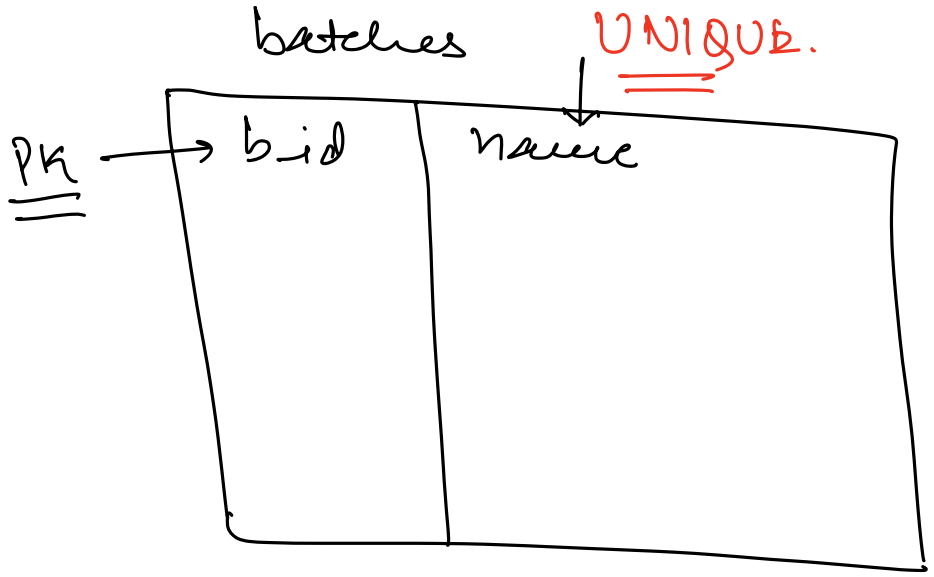
batches ↓

b_id	b_name
10	Mar23 Intermediate
11	—
12	
13	

Delete the batch with batch_id. → Not Allowed (NO ACTION)

→ Do what we did in the other table (CASCADE)

↓ Mark b_id as NULL in Students. (SET NULL)



⇒ (St_id, f_name, l_name) SK ✓

Composite key ✓

(f_name, l_name) ⇒ SK ✗

Composite key ✗

```
SELECT * FROM film;
-- Print
-- 2023-10-16 08:06:20
INSERT INTO film
VALUES
(default, "Tiger", default, 2012, 1, NULL, 3, 15, 120, 10, "G", "Trailers,Behind the Scenes",
default);

-- Error Code: 1048. Column 'last_update' cannot be null
-- Error Code: 1364. Field 'title' doesn't have a default value


select film_id as id, title as film_name
from film
order by film_name;

-- DISTINCT

select distinct rating from film;

select distinct rating, release_year
from film;

select "Hello World!";

select title, "Hello World" as greeting
from film;


-- Print the title and duration of the movie (in HRS)

select title, round(length/60) as length_in_hrs
from film;


create table film_copy (
  title VARCHAR(128) NOT NULL,
  release_year YEAR DEFAULT NULL
);

select * from film_copy;

insert into film_copy
select title, release_year from film;
```

```
select distinct description
from film;

select distinct title
from film;

select * from film
where release_year = 2006 AND rental_duration < 5;


-- NOT
-- Get all the movies which were not released in 2006

select * from film
where release_year <> 2006;

-- Get all the movies with rating "G" OR "PG"
select * from film
where rating = "G" or rating = "PG";

-- IN Operator.
select * from film
where rating NOT IN ("G", "PG");
```

Agenda :

READ.

→ BETWEEN

→ LIKE

→ IS NULL

→ ORDER BY

→ LIMIT.

UPDATE

DELETE.

⇒ BETWEEN.

Find all the students with psp >= 50 && psp <= 70.

```
Select *
from Students
where psp >= 50 AND psp <= 70.
```

Find all the movies released b/w 2010 & 2020.

```
Select *
from film
where release_year >= 2010 and
      release_year <= 2020.
```

⇒ BETWEEN a AND B ⇒ >= a AND <= b.

⇒ Select *
from Students
where psp between 50 AND 70;

⇒ Select *
from film
where release_year BETWEEN 2010 AND 2020;

⇒ BETWEEN 20 AND 50

⇒ 20 ✓
⇒ 19 ✗
⇒ 31 ✓
⇒ 50 ✓
⇒ 51 ✗

⇒ BUILT-IN functions.

- Recordings.
- Typed Notes.
- Google.

⇒

⇒ LIKE.

↳ String pattern matching.

Students.

id	name	batch_name
1	Venkat	Academy Mar22 Evt Morning
2	Priya	Apr22 Beg Academy Morning

⇒ All the academy batches contains Academy in the name.

⇒ Morning batches contains Morning in the name.

Q. Return all the morning batches in Academy.

```
bool checkMorningAcadBatches(String bName){
    if(bName.contains("Morning") &&
       bName.contains("Academy")){
        return true;
    }
    return false
}
```

⇒ LIKE : STRING MATCHING.

2 Wildcard Characters.

'%' : Any character any # of times.
(0, 1, 10, 20, ...)

'_' : Any character exactly 1 time.

⇒
```
Select *
from film
where desc LIKE '%Hello%'
```

⇒ _____ Hello _____ ✓

Hello _____

_____ Hello

⇒ Cat :
- __t ✓
- %t ✓
- % ✓
- %Cat% ✓

Q. Return all the morning batches

```
Select *
from batches
where batch_name LIKE '%Morning%.';
```

Q. Return all the morning batches in Academy.

```
Select *
from batches
where batch_name LIKE '%Morning%.';
      AND LIKE '%.Academy%.';
```

⇒ ____ Morning ____ Academy ✓

____ Academy ____ Morning ____ ✓

⇒ LIKE '%.academy%.'

ACADEMY ✓ ACAdemy ✓

⇒ Students with mes in their names.

```
Select *
from students
where name LIKE '%.mes%.'
```

⇒ Animesh ✓

⇒ Umes ✓

⇒ Ending with son.

%son → Jameson ✓, Samson ✓, Samsons ✗

✗ [%son% → Jameson ✓, Samsons ✓

⇒ Contains the word 'moon'

%moon%

⇒ ___1___2___3___

LIKE '○123○'

○ → Only 1 char

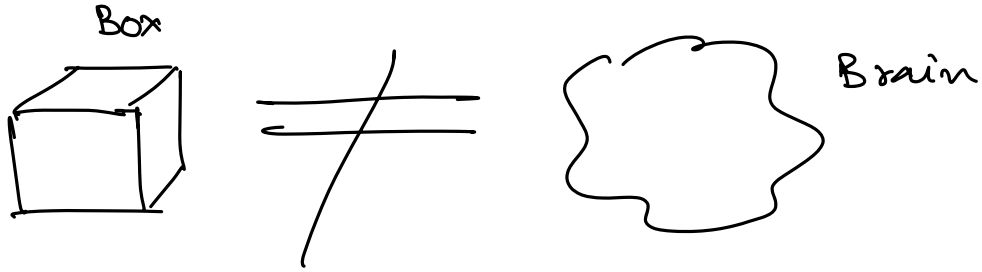
○ → Only 1 char

✗ %123% — abc123xyz ✗

IS NULL

⇒ We can't use (=) operator to match NULL values.

⇒ NULL = NULL ✗



⇒ IS NULL.

```
Select *
from film
where title = NULL
```

✗

```
Select *
from film
where title IS NULL;
```

ORDER BY

⇒ By default the order of data is NOT guaranteed.

⇒ If want the final ans to be ordered by some column

⇒ ORDER BY

```
Select *
from film
Order by title;
```

⇒ ASCENDING.

```
Select *
from film
Order by title Desc;
```

⇒ DESCENDING.

```
Select *
from film
Order by release_year, name;
```

Students.

id	name	yob
1	C	1990
2	B	1991
3	A	1990

```
Select *
from students
Order by yob
```

C 1990
A 1990
B 1991

```
Select *
from students
Order by yob, name;
```

A 1990
C 1990
B 1991

NOTE: Ultimate tie breaker is always PK.

```
table : [{} {} - - - . {}]
ans : [ ]
for each row in table:
    if condition is matched:
        ans.add(row);

ans.sort(sorting cond^n);
```

ORDER BY with DISTINCT keyword

```
Select DISTINCT title
from film
ORDER BY release_year;
```

⇒ Students

Ex

id	name	YOB
1	abhishek	1998
2	abhishek	1990
3	bhim	1995

```
Select distinct name
from Students
order by yob;
```

⇒ Bhim. 1995
Abhishek 1998

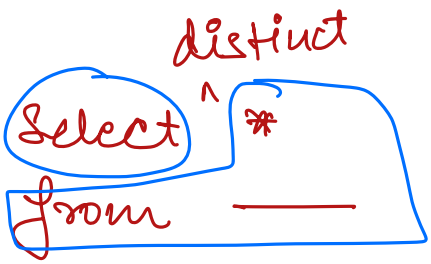
If we have distinct in select clause, then we can only order by the columns present the distinct clause.

```
Select distinct a, b
from table
Order by a
```

✓

```
Select distinct a, b
from table
Order by c
```

✗



where name = ____

Order by ____

FROM
↓
WHERE
↓
ORDER BY
↓
DISTINCT
↓
PRINT

⇒

table : [{ } { } — - - . { }]

ans : [ ]

for each row in table :

if condition is matched : ⇒ where

ans.add(row);

ans.sort(sorting cond^{n})

final_ans = set(ans) ⇒ DISTINCT.

print(final_ans) {Ambiguity}.

id	name	YOB
1	abhishek	1998
2	abhishek	1990
3	bhim	1995

select distinct name
from Students
order by yob;

ans : [{Abhishek, 1998} {Abhishek, 1990}, {Bhim, 1995}]

↓

ans : [{Abhishek, 1990} {Bhim, 1995} {Abhishek, 1998}]

↓

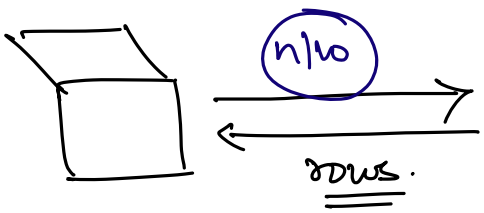
DISTINCT NAMES.

⇒ Set doesn't preserve the order of elements.

⇒ LIMIT.

```
Select *
from Students.
```

⇒ 10^6 rows.



DB ⇒ latency. (time).

⇒ LIMIT.

```
Select *
from Students.
LIMIT 10;
```

} first 10 rows

⇒ 11 to 20.

```
Select *
from Students.
LIMIT 10 OFFST 10;
```

CRUD

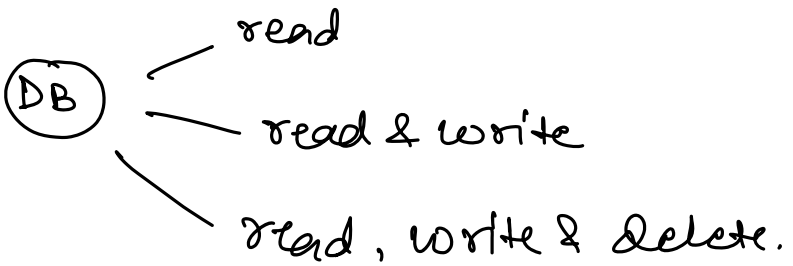
=> Update.

↳ Updating row data.

```
Update {table_name}
Set psp = 99;
```

=> Update the psp for all the students.

```
Update {table_name}
Set psp = 99
Where id = 10.
```



DELETE.

↳ Rows

```
delete from {table}
```

=> Deletes all rows.

```
delete from {table}
where id = 10
```

✓

table : [{ } { } — - - . { }]

for each row in table :
    if row matches cond^{n} :
        deletes row ;

Agenda :

→ JOINS.

→ Self Join.

JOINS.

=> Till now only single table was involved in the queries.

=>

students.

id	name	batch_id
1	Vishal	1
2	Rahul	1
3	Deepak	2
4	Ajay	3

batches

id	name
1	A
2	B
3	C
⋮	
10	

Q. for every student print their batch name.

name	b_name
Vishal	A
Rahul	A
Deepak	B
Ajay	C

⇒ We need to combine data from 2 tables.

↓

JOINS.

⇒ JOINS. : Allows us to combine the data across multiple tables.

SYNTAX.

```
Select  students.name, batches.name
from    students
 join   batches
  ON    student.batchid = batches.id;
```

↓

Cond^n

```
Select  s.name, b.name
from    students as s
 join   batches b
  ON    student.batchid = batches.id;
```

↓

JOIN Condition.

⇒ ans: [ ]

```
for each row in the Students:
    for each row in the batches:
        if JOIN cond^n matches:
            ans.add(row);
print(ans);
```

O(NM)

JOINS: We are combining the data from Students & Batches table based on the batch_id.

⇒ STUDENT BUDDY.

We assign a senior Student as a Buddy to each Scaler Student.

Students

id	name	batch_id	psp	buddy_id
1	Parth	1	85	4
2	Umes	2	90	4
3	Venkata	3	92	4
4	Deepak	2	93	NULL

BUDDY ID : Self referential Column.

⇒ A buddy is nothing but a Student itself.

⇒ Buddy id is an id of another Student.

Q: For every Student, print their name along with their buddy name.

Students

id	name	batch_id	psp	buddy_id
1	Parth	1	85	4
2	Vinci	2	90	4
3	Venkata	3	92	4
4	Deepak	2	93	NULL

buddies

id	name

```
Select  students.name, buddies.name
from    students
 join   buddies
 ON     students.buddy_id = buddies.buddy_id ;
```

```
Select  S1.name as Name, S2.name as Buddy name
from    students S1
 join   students S2
 ON     S1.buddy_id = S2.id
```

⇒ SELF JOIN.

⇒ In self join, table name aliases are mandatory.

Ex: Employees.

employees.

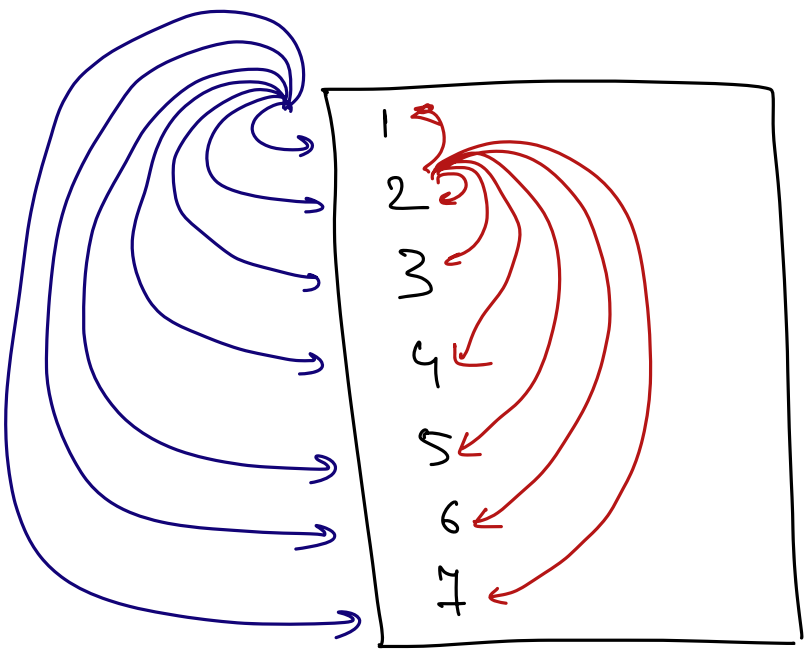
id	name	joining_date	department	manger id

Q: for every employee, print its name & its managers name.

```
Select e.name as Name, m.name as Manager
from employees e
join employees m
 ON e.manger_id = m.id
```

ans: [ ]

```
for each row in the Students:
    for each row in the Students:
        if JOIN cond^ matches:
            ans.add(row);
print(ans);
```



JOINS.

Students

id	name	b_id
1	Rajesh	1
2	Umesh	1
3	Jasid	2

batches

id	b_name
1	A
2	B

JOIN cond^

Students (s)

id	name	b-id
1	Rajesh	1
2	Vinu	1
3	Jasil	2

batches (b)

id	b_name
1	A
1	A
2	B

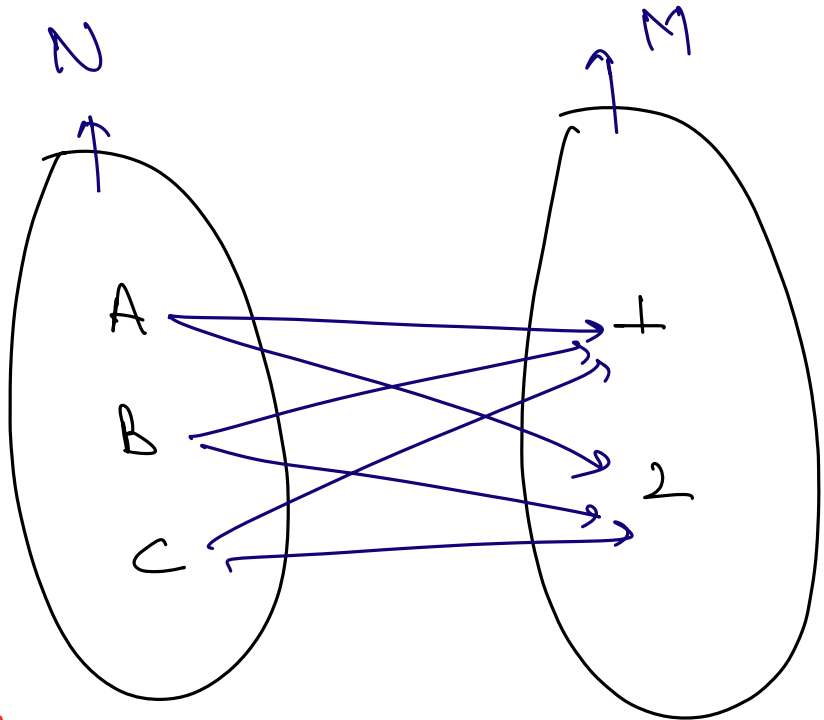
Temp table.

Select s.name, b.name.

```
from students (s)
join batches (b)
ON s.b_id = b.id
```

→ temp table.

⇒ Cross join.



N×M

A1
A2
B1
B2
C1
C2

```
Select ——
from table1, table2
```

Students

id	name
1	A
2	B
3	C

job_referals.

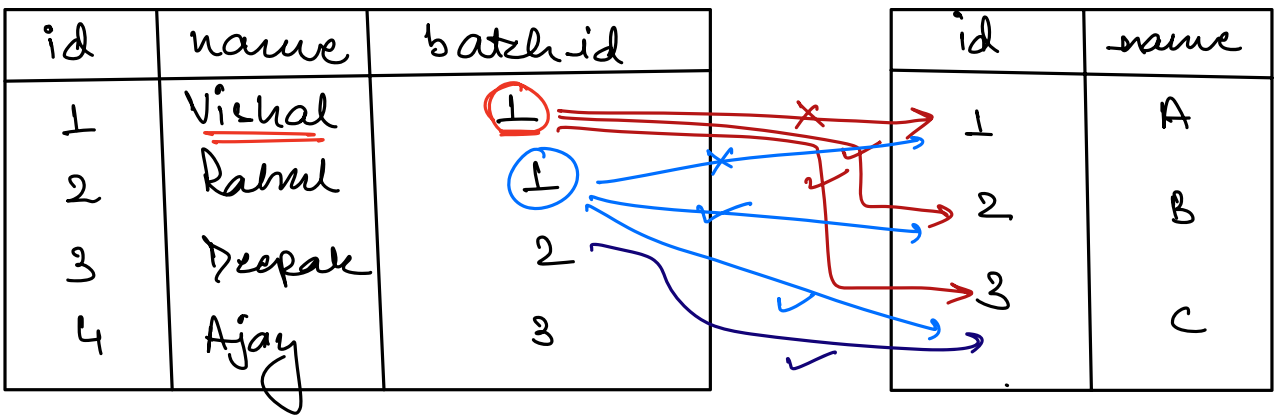
id	referral
1	Amazon
2	Google.

St-name	referral
A	Amazon
A	Google
B	Amazon
B	Google
C	Amazon
C	Google

```
Select ____________
from Students, referrals;
```

→ left Join
→ Right Join
→ Inner Join
→ Outer Join.

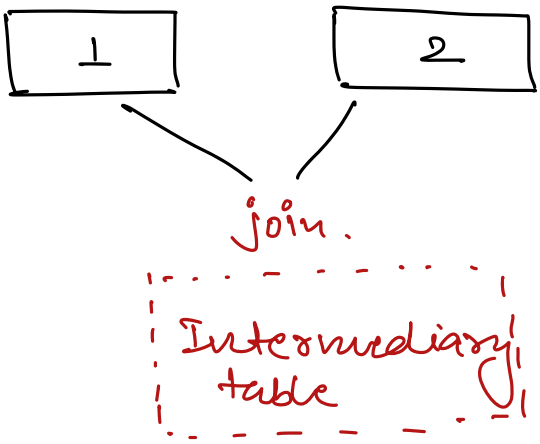
```
Select  s.name, b.name
from    students as s
join    batches  b
ON      s.batchid < b.id;
```



id	name	batchid
1	Vishal	1
2	Rahul	1
3	Deepak	2
4	Ajay	3

id	name
1	A
2	B
3	C

Joining multiple tables



Ex. Students.

id	name	instructor_id	batch_id

instructors

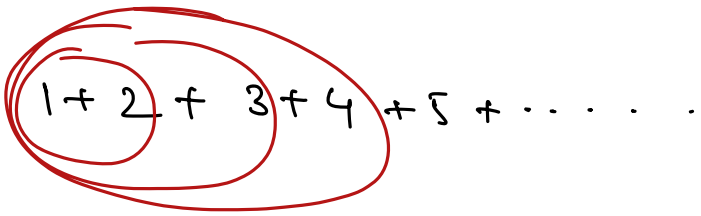
id	name

batches

id	name

Q. For every student, give their name along with the name of their instructor & name of their batch.

Abhishek	Deepak	Mar23 Intr.



⇒ Join the tables in Pairs.

```
select s.name , i.name , b.name
from students s
join instructors i
ON s.instructor_id = i.id
join batches b
ON s.batch_id = b.id
```

Students (s)				instructors (i)		batches (b)	
id	name	instructor_id	batch_id	id	name	id	name

table1 : [ ]
table2 : [ ]
table3 : [ ]

ans : [ ]

ans1 : [ ] // Intermediary.

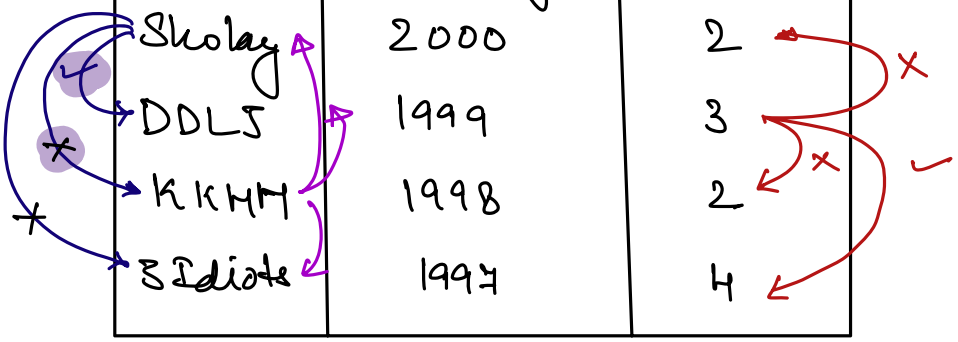
```
for each row1 in table1:
    for each row2 in table2:
        if cond^n is TRUE:
            ans1.add(row1 & row2)

for each row in ans1:
    for each row in table3:
        if cond^n is TRUE:
            ans.add(_____);

print(ans);
```

COMPOUND JOINS.

Q. For every film, name all the films that were released with ±2 years of that film & their rental rate > rental rate of the movie.

film

name	release_year	rental rate
Sholay	2000	2
DDLJ	1999	3
KKHH	1998	2
3Idiots	1997	4



⇒ Output

Sholay | DDLJ.

DDLJ | 3 Idiots.

KKHH | DDLJ

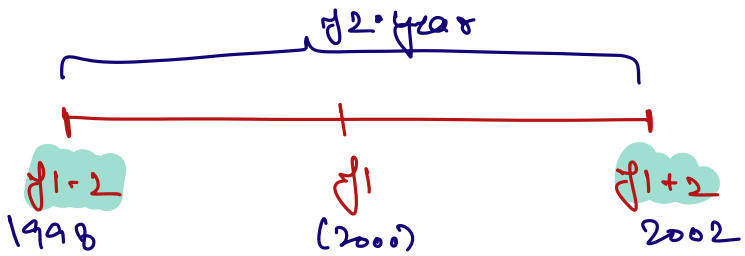
KKHH | 3 Idiots.

film (f1)

name	release_year	rental rate
Sholay	2000	2
DDLJ	1999	3
KKHH	1998	2
3Idiots	1997	4

film (f2)

name	release_year	rental rate
Sholay	2000	2
DDLJ	1999	3
KKHH	1998	2
3Idiots	1997	4



```
Select f1.name, f2.name
from film f1
join film f2
ON (f2.year >= f1.year - 2 AND
    f2.year <= f1.year + 2) AND
    f2.rental_rate > f1.rental_rate ;
```

BETWEEN.

f2.year BETWEEN (f1.year - 2, f1.year + 2)

(1) Join need not to have always equality cond^.

(2) Join can also have multiple conditions in ON clause. ⇒ Compound Join.

⇒ Type of Joins.

Students

id	name	buddy_id

Q. for every student, print their name along with their buddy name.

Students (s) ⇒ left

id	name	buddy_id
1	A	2
2	B	NULL
3	C	2
4	D	1

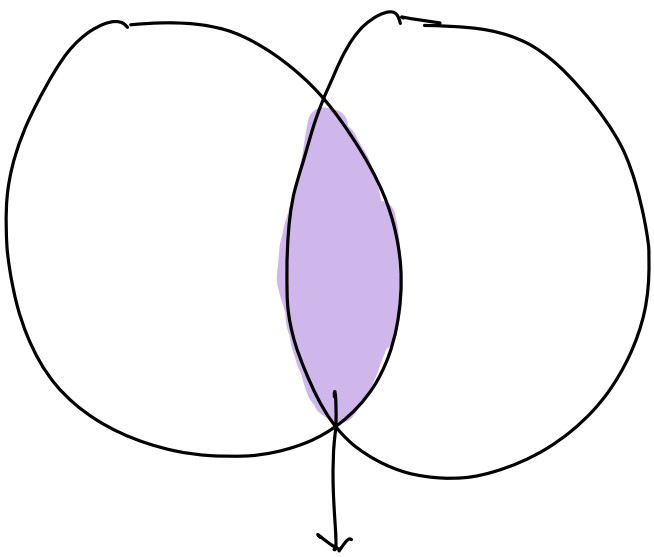
Students (b) ⇒ Right

id	name	buddy_id
1	A	2
2	B	NULL
3	C	2
4	D	1

```
Select ____________
from Students s
join students b
ON s.buddy_id = b.id;
```

inner → Optional.

Student	Buddy
A	B
C	B
D	A



Inner Join.

⇒ When do a join b/w 2 tables Left & Right, if a row of left table doesn't match with a row of right table, then we'll not have that row in the output & Vice-versa.

Q. for every student print their names along with names of their batches.

Students:

id	name	batch_id
1	John	1
2	Jane	1
3	Jim	Null
4	Jenny	Null
5	Jack	2

batches

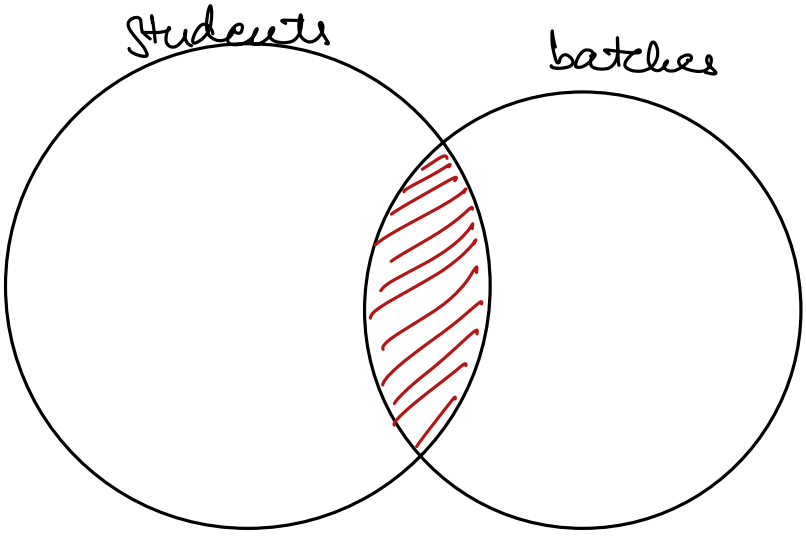
id	name
1	A
2	B
3	C

⇒ John | A
Jane | A
Jack | B

```
Select S.name, b.name
from Students s
join batches b
ON S.batch_id = b.id;
```

⇒ INNER JOIN.

INNER JOIN : Join that includes only rows from both the tables that matches the cond^{n}.



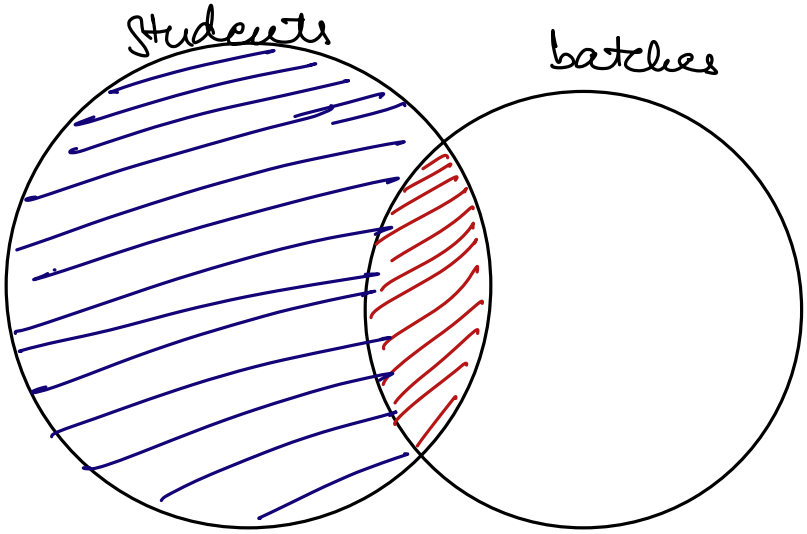
OUTER JOINS

⇒ Allows us to get the rows which don't matches the cond^{n}.

1) Left JOIN. / Left Outer Join.
2) Right Join / Right Outer Join.
3) full Outer Join.
   ↳ MySQL doesn't support.

LEFT JOIN.

→



→ Includes all the rows matches the cond^

+

Includes all the rows from the left table that doesn't match the cond^. for these rows we'll put NULL for all the columns of right table.

Students.

id	name	batch_id
1	John	1
2	Jane	1
3	Jim	Null
4	Jenny	Null
5	Jack	2

batches

id	name
1	A
2	B
3	C

John | A
Jane | A
Jack | B
Jim | NULL
Jenny | NULL

```
Select s.name, b.name
from students s
left join batches b
ON s.batch_id = b.id;
```

RIGHT JOIN.

Includes all the rows matches the cond^

+

Includes all the rows from the right table that doesn't match the cond^. for these rows we'll put NULL for all the columns of left table.

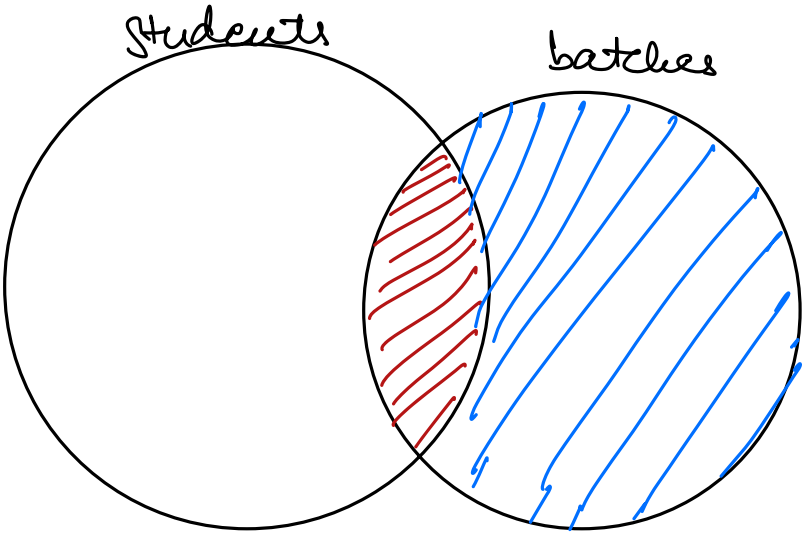
Students

id	name	batch_id
1	John	1
2	Jane	1
3	Jim	Null
4	Jenny	Null
5	Jack	2

batches

id	name
1	A
2	B
3	C

John | A
Jane | A
Jack | B
Null | C



FULL OUTER JOIN.

Includes all the rows matches the cond^

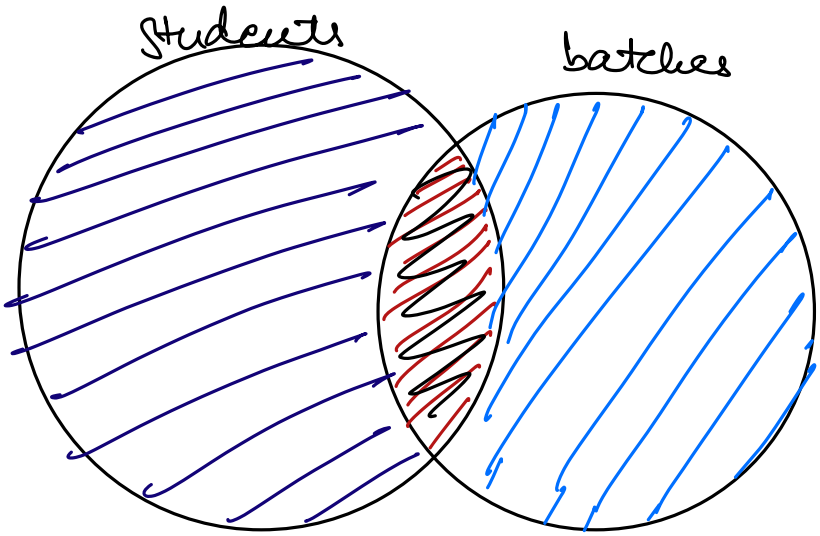
+

Includes all the rows from the left table that doesn't match the cond^. for these rows we'll put NULL for all the columns of right table.

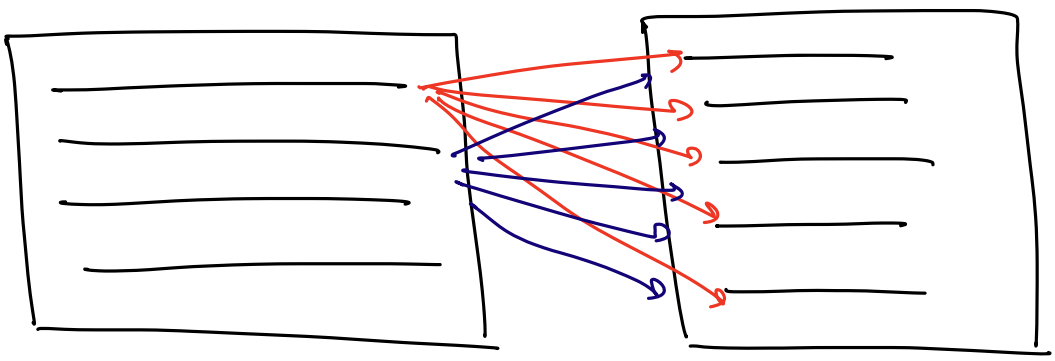
+

Includes all the rows from the right table that doesn't match the cond^. for these rows we'll put NULL for all the columns of left table.

John | A
Jane | A
Jack | B
Jim | NULL
Jenny | NULL
Null | C



Cross Join.



Select ——

from table1 ⇒ 9

join table2 ⇒ 4

ON (true)

9 × 4

⇒ 36.

⇒ O(N × M)

⇒ No cond^{n}.

```
Select ____________
from table1, table2;
```

Q. full Outer Join ≠ Cross Join.

↓ A+B. ↓ A×B.

USING.

→ joining based on the equality cond^{n}.

→ Name of the column is same.

```
Select ____________
from student s
join batches b
ON s.batch_id = b.batch_id
```

↓

```
USING (batch_id)
```

NATURAL JOIN.

⇒ When we join tables based on equality cond^ for all the same columns (name), that is natural join.

table (1)

a	b	c	d	e

table (2)

b	c	x	y	e

```
Select ————————
from table1
join table2
ON table1.b = table2.b AND
   table1.c = table2.c AND
   table1.e = table2.e
```

⇒

```
Select ————————
from table1
join table2
USING (b, c, e);
```

#

```
Select ————————
from table1
natural join table2
```

↓

Only joins based on the same columns.

IMPLICIT JOIN. = Cross Join.

```
Select *
from table1, table2;
```

ON (vs) WHERE.

(1)

```
Select ————
from A → (N)
join B → (M)
ON A.id = B.id
```

⇒

```
A: [ ] , B: [ ]
ans: [ ]
for row1 in A :
    for row2 in B:
        if Cond is TRUE :        (ON)
            ans.add(row1, row2);
print(ans)
```

} N×M.

⇒ Select ——

(2) from A

Cross Join B

Where A.id = B.id;

```
A: [ ] , B: [ ]
temp: [ ], final_ans: [ ]
for row1 in A :
    for row2 in B:
        temp.add(row1, row2);
```

Cross Join table.

```
for row in ans :
    if cond^n is TRUE || WHERE.
        final_ans.add(temp[row])
```

⇒ 1st Approach is better.

———— * ————

Aggregate Queries.

→ Aggregate functions ⇒ COUNT, MIN, MAX, SUM..

→ Group By

→ Having

Aggregate Queries.

Till now whatever queries we have written, they were returning some rows.

→ find students ——————

→ find batches ——————

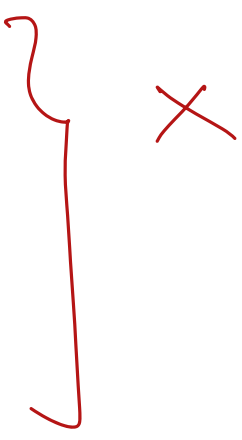
Q. find the avg rep of all the batches at scaler.

Output:

batch_id	avg rep
1	90
2	70
3	75

batches

batch_id	name	rep.



Students

id	name	batch_id	DSP
1	A	1	90
2	B	1	80
3	C	1	70
4	D	Null	—
5	E	2	90

go through the Students table.

⇓

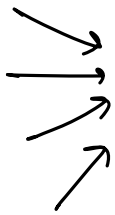
Combine rows of Students for each batch.

⇓

find the Avg batch.

Aggregate functions

↳ COUNT / SUM / AVG / MAX / MIN ....



COUNT( ) ⟶ 1 value.

Q. COUNT.

Students

	id	name	psp	batch_id
→	1	A	80	1
→	2	B	70	1
→	3	C	90	Null
→	4	D	80	2

⇒ Find the count of students that have a batch.

COUNT: takes a lot of values in the input, combines them into a single value which is equal to count of non-null values.

COUNT(1, 2, 3, 4, 5) ⟶ 5

COUNT(1, 2, NULL, 4) ⟶ (3)

Select COUNT(batch_id)

for loop [from Students.]

⇓

COUNT(1, 1, NULL, 2) ⇒ 3

?

Print.

```
Select COUNT(batch_id)
from Students.
Where batch_id IS NOT NULL;
```

from → Where → Select.

⇓

COUNT(1, 1, 2) ⇒ 3

Note:- All aggregate functions ignores the NULL values.

Students

id	name	psp	batch_id	age
1	A	80	1	22
2	B	70	1	21
3	C	90	Null	24
4	D	80	2	23

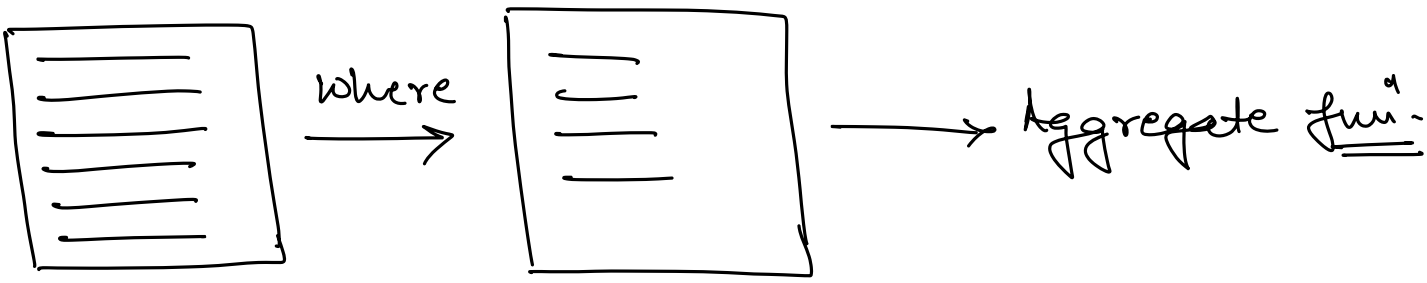
⇒ Count of all the students that have a batch and age < 23.

```
Select COUNT(batch_id)
from Students.
Where age < 23.
```

⇒ 2

id	name	psp	batch_id	age
1	A	80	1	22
2	B	70	1	21

COUNT(1, 1) => 2




```
Select
from  Students s
join batches b
ON  s.batch_id = b.id;
```

→ Nested for loop.

from → Select.

Q.

Students

id	name	batch_id
1	A1	1
2	B1	1
3	C1	2

batches

id	name
1	A
2	B

get the count of students with batch name (A).

```
Select COUNT(S.id)
from Students S
join batches b
ON S.batch_id = b.id;
Where b.name = 'A';
```

Intermediate table

Students		batches	
id	name	b.id	id	name
1	A1	1	1	A
2	B1	1	1	A
3	C1	2	2	B

JOIN

ans

Where.

ans2.

Students			batches	
id	name	b_id	id	name
1	A1	1	1	A
2	B1	1	1	A

⇒ Select Count(id)
from A
join B
ON Cond'1 (1)
Where cond2 (2)

A: [ ]

B: [ ]

ans: [ ]

```
for row1 in A:
    for row2 in B:
        if cond1 is TRUE:
            ans.add(row1, row2)
```

} JOIN.

ans2: [ ]

```
for row in ans:
    if cond2 is TRUE:
        ans2.add(row);
```

Where.

```
Count_b = 0
for row in ans2:
    if row[id] != NULL:
        Count_b ++;
print (Count_b)
```

⇒ AVG (1, 2, 3, NULL) ⇒ 2, 3, Null, 6, 4

Aggregate fun".

$\frac{1+2+3}{3} = \frac{6}{3}$

$= (2)$

Students.

id	psp
1	1
2	2
3	3
4	Null

6/4 = 1.5.

SUM(PSP) / Count(id) ≡ AVG(PSP) ✗

↳ $\frac{1+2+3}{3} = (2)$

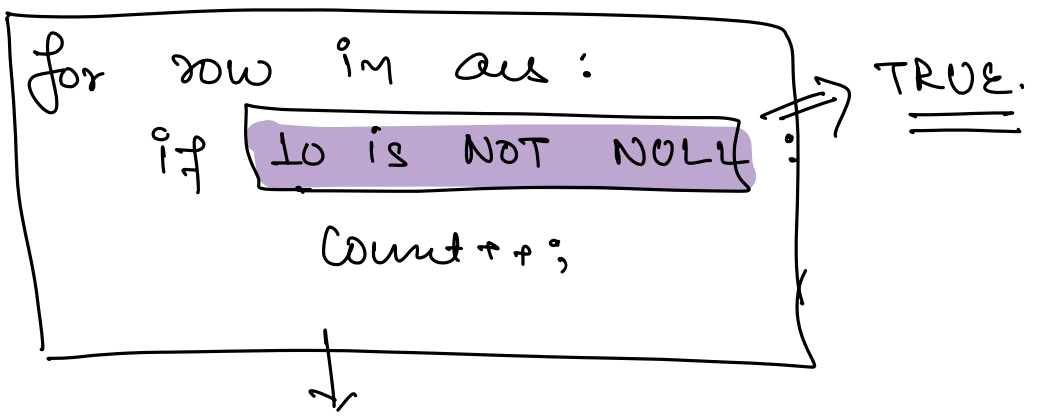
(*) ⇒ Complete row.

Count(id)
Count(psp)
Count(b_id)

⇒ COUNT(*)

COUNT(row1, row2, row3, - - - -)

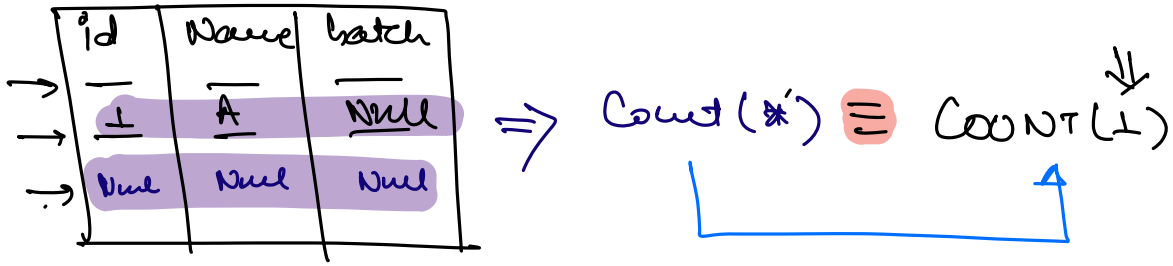
⇒ Select Count(10) ⇒ DUMMY. (Always count all the rows).
from Students.



Count(*)

Efficient.

Count(*) == Count(1)
Count(2)
Count('SCALER')
Count('Umes')

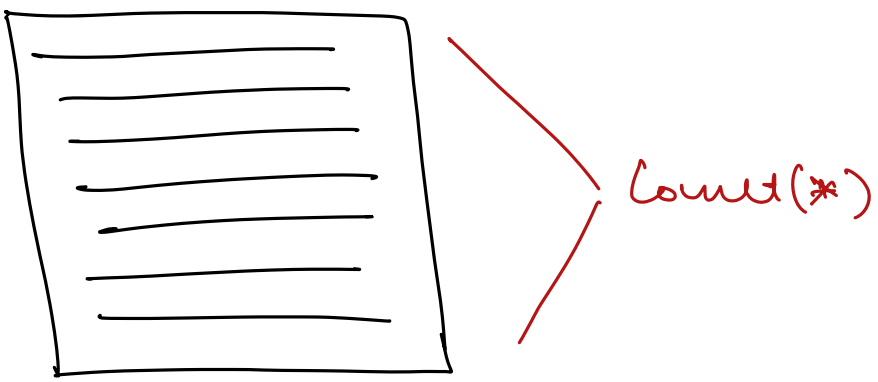


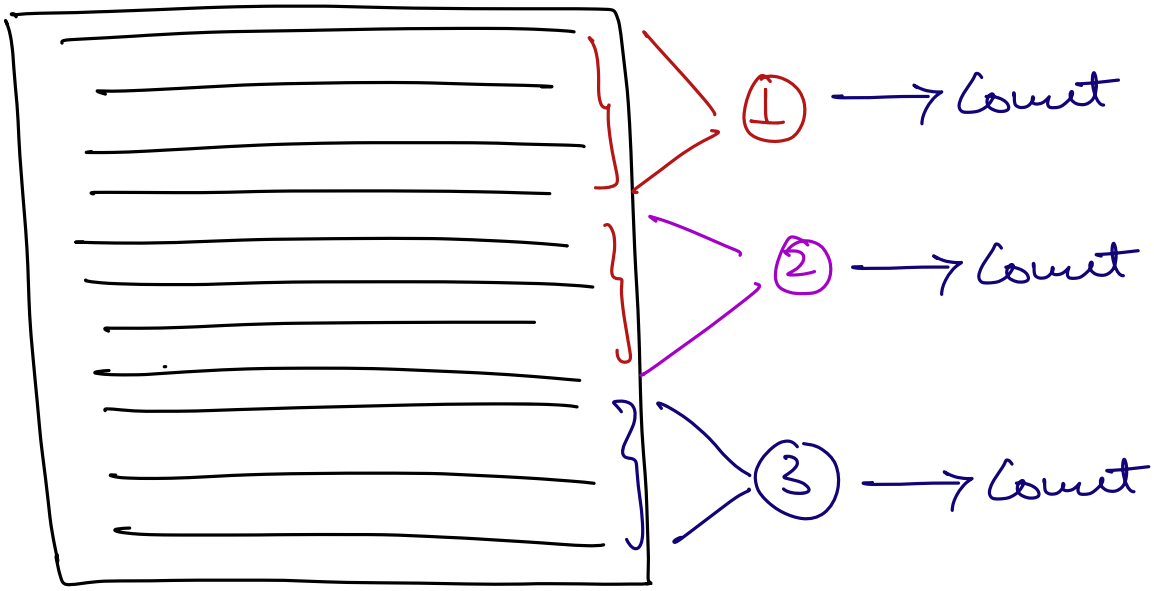

```
if (true) {
    ===
    ===
    ===
```

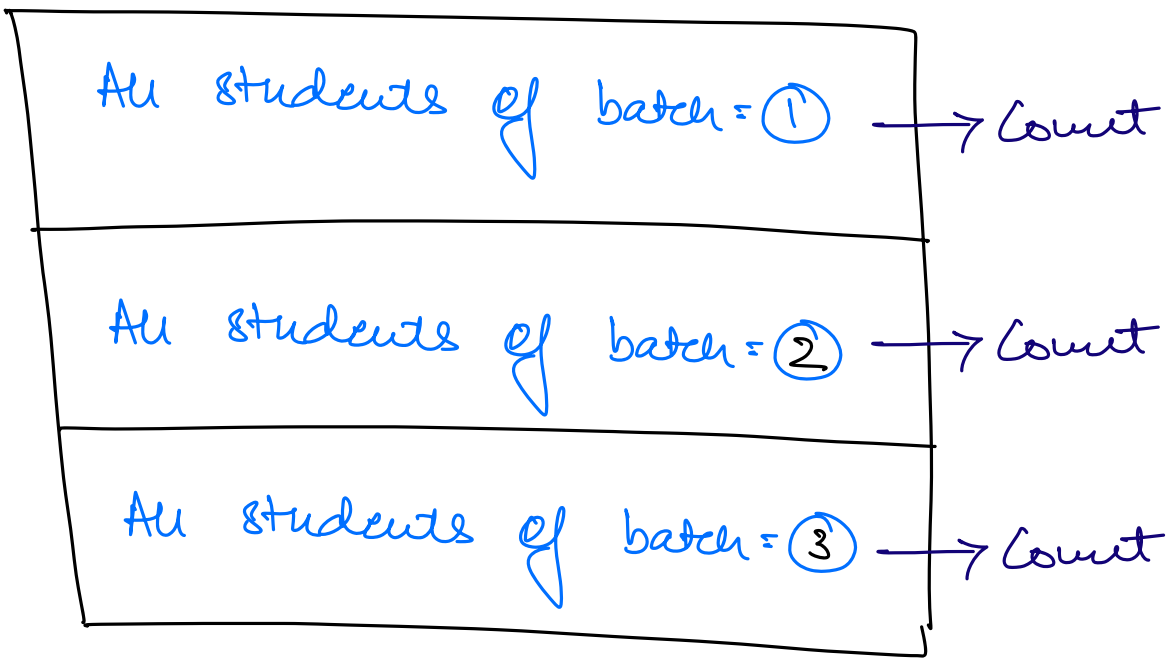
3

Get the count of students for every batch.

b_id	count
1	50
2	70
3	40
4	60





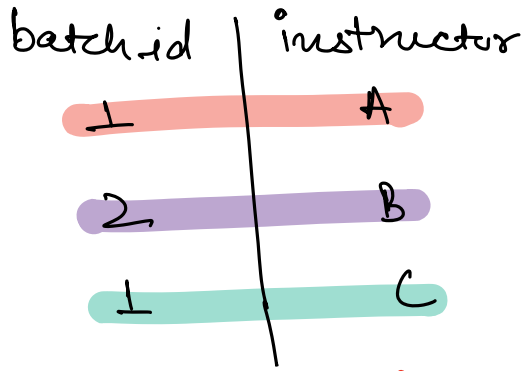


⇒ GROUP BY. : Allows us to split the table into multiple groups and apply the aggregate fun' on groups.

⇒ group by batch_id

↳ brings all the Students of a batch together.

⇒ group by batch_id, instructor ;

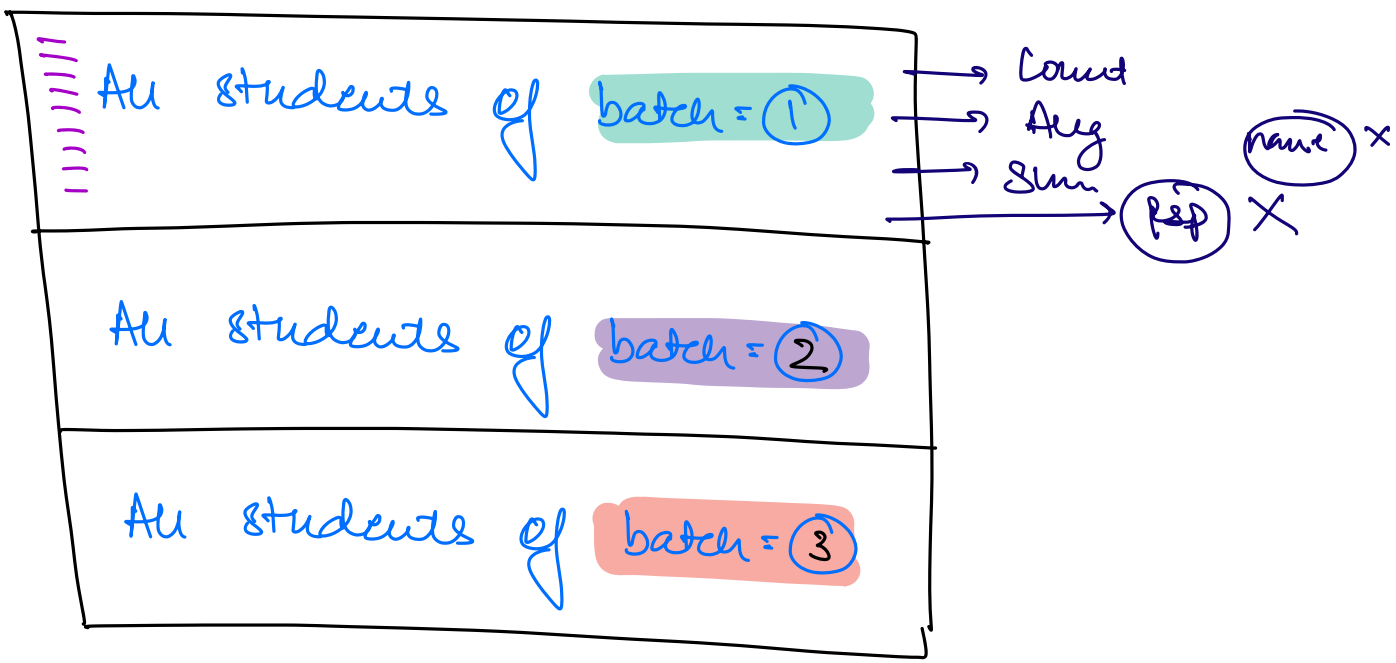


Count students in each group.

```
Select  COUNT(1)
from    student
group by  batch_id
```

⇒

50
70
40
60




```
Select  COUNT(1), batch_id
from    student
group by  batch_id
```

⇒

50	1
70	
40	3
60	4

```
Select COUNT(1), batch_id, P, name, avg(PSP)
from student
group by batch_id
```

⇒ If we are using group by clause then we can only use aggregate queries (or) Columns in group by in Select statement.

Q. Print the batch names along with no. of students having more than 100 students.

Students

id	name	b_id

batches

id	name

```
Select batch_name, count(*)
from students s
join batches b
ON s.b_id = b.id
group by b.name;
HAVING count(*) > 100;
```

① ② ③

FROM
↓
group by
↓
HAVING
↓
Select.

temp-

student (s)			batches (b)	
id	name	b_id	id	name

batch names
A
B

temp

group (batch_name = A) → count = 100

group (batch_name = B) → count = 120

group (batch_name = C) → count = 70

⇒ Where Clause :- Used to filter rows

⇒ Filter groups : HAVING clause.

⇒ After creating groups can we have where clause ?

NO.

⇒ We can't add where clause after group by.

Q. Print the batch names along with no. of students which have psp > 80. having more than 100 students

```
Select batch_name, count(*)          (5)
from students s
join batches b                       (1)
ON s.b_id = b.id
where s.psp > 80.                    (2)
group by b.name;                     (3)
HAVING count(*) > 100;               (4)
```

ans:

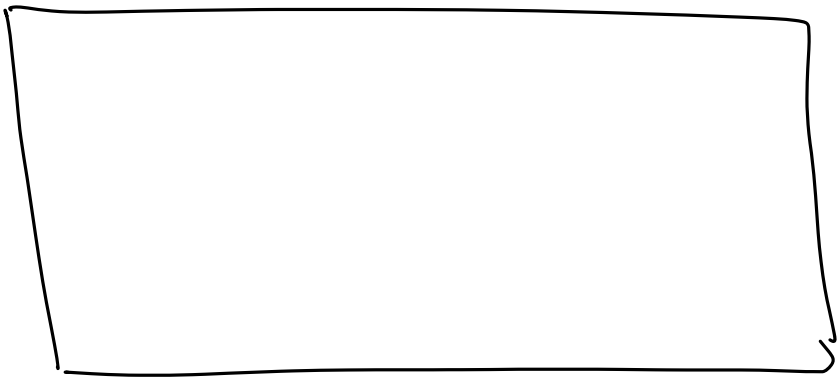
student (s)			batches (b)	
id	name	b_id	id	name

→

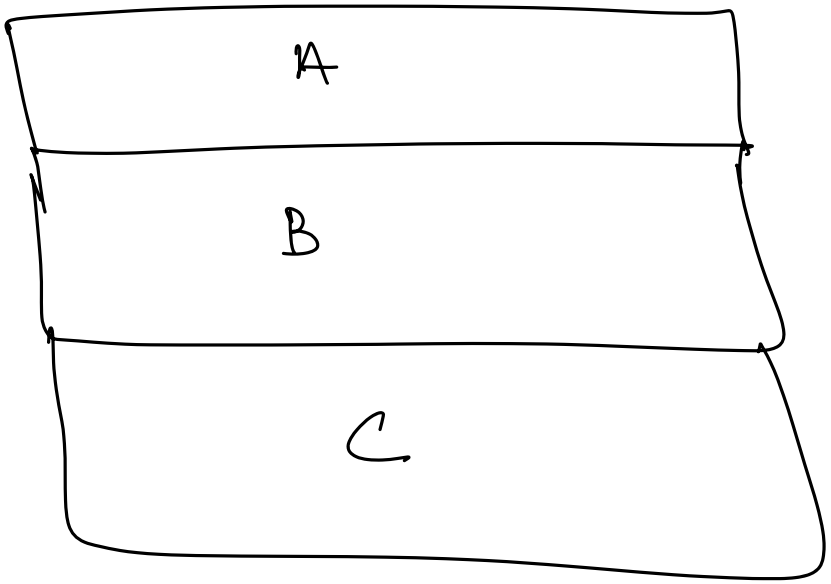
→

↓ 780 (where)

Case2



↓ Group by.



↓ HAVING Count > 100

↓ Select.

From
↓
Where
↓
Group by
↓
HAVING
↓
Select

Subqueries.

- → Subqueries.
- → IN.
- → ALL
- → ANY
- → Correlated Subqueries.
- → EXISTS.

Q. Students

id	name	psp	batch_id

Find all the students with psp > maximum psp of batch_id = 2.

students : [. . . . .]

① find the MAX psp of batch_id = 2;

↳ (x)

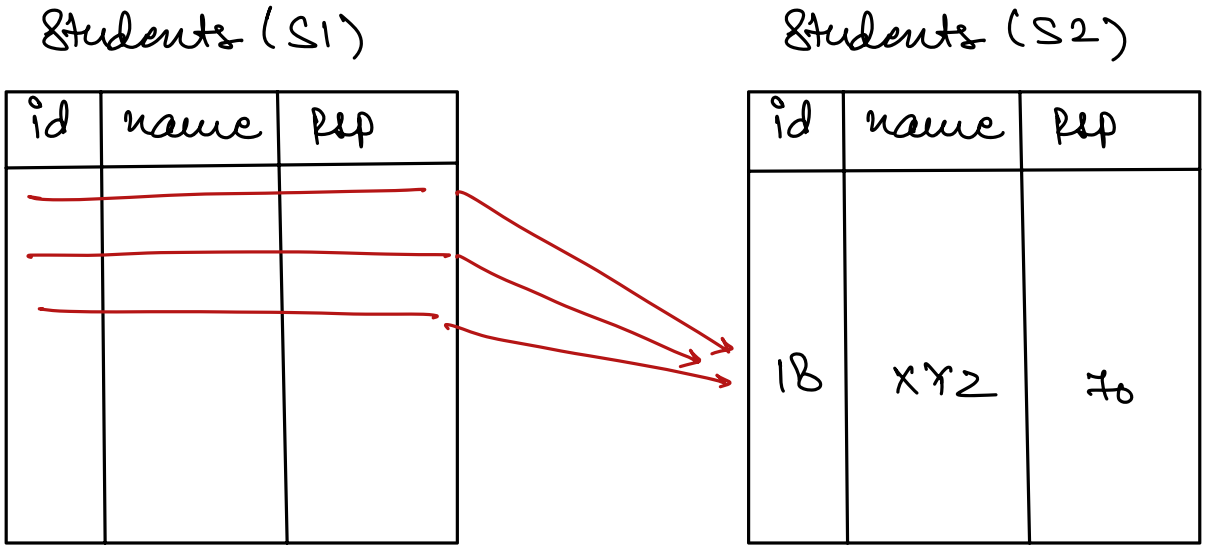
```
Select max(psp)
from students
where batch_id = 2
```

⇒ (x)

(II) Find all the students with psp > x.

```
Select *
from Students
where psp > x
```

Q: Find all the students whose psp > psp of student with id = 18.




```
Select  S1.id , S1.name
from    Students  S1
join    Students  S2
ON      S2.id = 18  AND  S1.psp > S2.psp
```

Self Join.

① Find psp of id = 18. ⇒ (x)

② Find students with psp > (x).

SUBQUERY : When we break a problem into smaller subproblems and use their ans to find the final ans.

⇒ Most of the times problems that we solve via subqueries can also be solved by some other type of queries but generally subqueries are easier to understand.

① Find psp of student with id = 18. ⇒ (x)

② Find students with psp > x.

① Select psp
from students ⇒ (x)
where id = 18;

② Select *
from students
where psp > x.

(+)

```
Select *
from Students
where psp > (Select psp
             from Students
             where id = 18)
```

⇒ Subqueries are easier to understand.

⇒ Subqueries :- Bad performance.

TC : $O(N^2)$

Above subquery is returning a single no. (psp of id = 18).

Q. Fill all the students with psp > MAX psp of batch_id = 2

```
Select *
from Students
where psp > (Select max(PSP)
             from Students
             where batch_id = 2)
```

↓

Subquery is returning a single value.

Subqueries can return multiple values as well

No. of rows	No. of columns	Output
1	1	Single value
1	M	Single row.
M	1	Single column table.
M	M	

>, <, >=, <=, = : Single value.

IN.

Q. users.

id	name	is_student	is_ta.
1	AB ✓	T	f
2	XY		f
3	Pranav	T	T ✓
10	AB	f	T

⇒ Get the name of students that are also name of TA's.

(st)
users

id	name	is_student	is_ta.
1	Vishal	T	f/T
2	Deepak	f	

(ta)
users

id	name	is_student	is_ta.
10	Vishal		(T)

```
select ____________
from students st
join tas ta
ON st.name = ta.name;
```

⇒

```
select distinct(St.name)
from users st
join users ta
ON st.is_student = true AND
   ta.is_ta = true AND
   St.name = ta.name.
```

(1) find name of all the TA's. ⇒ [ ]

(2) find all the students with name ↴

(I)

```
Select name
from users
Where is_ta = true;
```

⇒ [ ]

(II)

```
Select distinct (name)
from users
Where is_student = true AND
   name IN (Select name
            from users
            Where is_ta = true);
```

['Deepak', 'Akhilesh', 'Pranav']

Q: Get the name of all the students whose psp is not less than smallest psp of any batch.

batchid	min psp
1	40
2	30
3	50

psp = 50 ✓
20 ✗
32 ✗
45 ✗
52 ✓
70 ✓

Not less than any value.

OR

greater than / equal to all values.

nγ = Max of the min psp's of all the batches.

(I) Select MIN(psp)
from students
group by batchid

40
30
50
⇒ ✗

(II) Select MAX(psp) } ⇒ (Y) ⇒ 50
from (X)

(III)
```
Select *
from students
where psp >= (Y)
```
50

(3)
```
Select id, name
from students
where psp > (Select MAX(min_psp)
             from (Select MIN(psp) as min_psp
                   from students
                   group by batch_id) minPspBatches
             (2)
             )
                   (1)
```

SUBQUERY inside FROM.

1. We can give subquery inside from clause as well.
2. Output of the subquery will be considered as a table.
3. We need to give a name to the subquery table if we are using with in from.

#

x >/ ALL (10, 20, 30, 40)

x should be greater than (or equal to) 10, 20, 30, & 40.

ALL ⇒ AND.

```
Select *
from students
where psp >= ALL(Select MIN(psp)
                 from students
                 group by batchid)
```

ANY. ⇒ OR.

Q: Get all the students with psp greater than avg psp of their batch.

student, b_id = 2, psp > Avg psp of b_id = 2

bid = 3, psp > Avg psp of b_id = 3.

① Get students with psp > (X)

② X = Avg psp of student's batch.

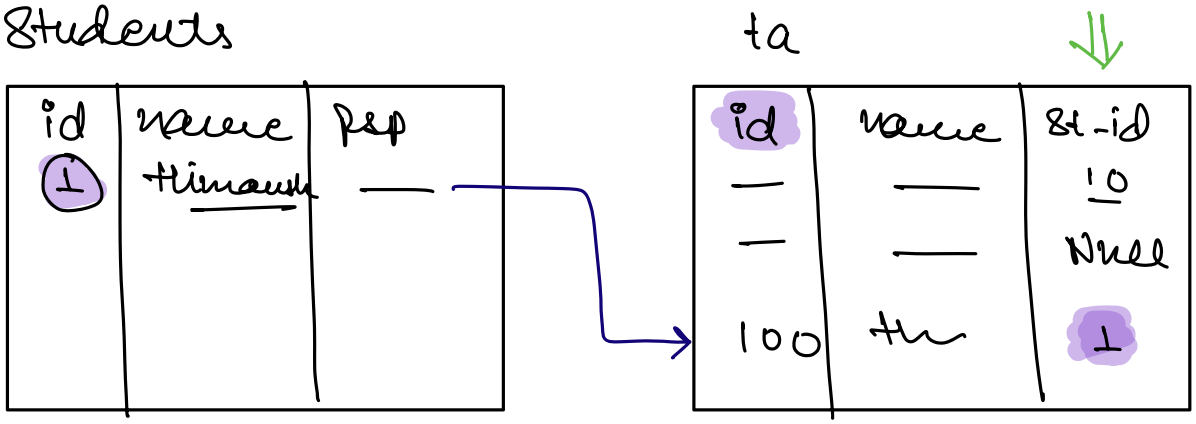
```
Select *
from student st
where psp > ( Select avg(psp)
              from students
              where batch_id = st.batch_id)
```

⇒ Correlated Subqueries.

```
                                    => st
for student in Students:
    for row in Students
        if (row.b_id = st.b_id) {
            avg_psp = ————————
        }
    }
    if (st.psp > avg_psp) {
        // Add in the ans
    }
}
```

EXISTS

Q. Get all the students who are also TA.




```
Select st_id
from ta
where st_id IS NOT NULL;
```

=> select id
from Students
where id IN (select st_id
from ta
where st_id IS NOT NULL);

O(N)

IM
(Linear search)

[1, 10, 12, 15].

IM.

TC:

select *
from Students (s)
where EXISTS (select st_id
from ta (t)
where t.st_id = s.id)

This query will be faster

#

Students

id	st_name
1	Ranjal ✓
2	

PK

mentor_sessions

s_id	s_id	mentor_id
100	1	
101	2	
102	1	
	1	
	1	
	1	

⇒ Get all the students names those have taken a mentor session.

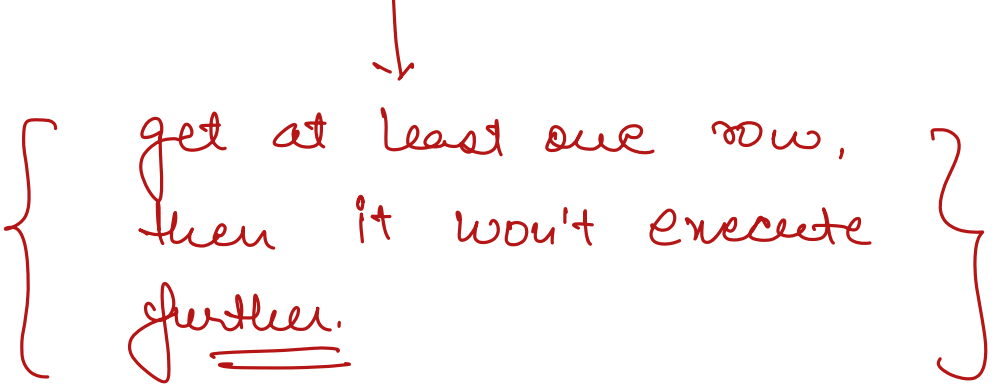
```
Select *
from Students
where id IN ( select st_id
              from mentor_sessions );
```

IN.

[1, 2, 10, 5, ——]

IN

```
Select *
from Students S
where EXISTS ( select *
               from mentor_sessions ms
               where ms.st_id = S.id )
```



IN (VS) EXISTS.

Agenda:

→ Indexes.
→ How indexes work?
→ Indexes for range queries.
→ Cons of Indexes
→ How to create indexes!

⇒ Indexing.

```
[for i → 1 to N
   [for j → 1 to M
      ans.add(row)
```

} O(NM) ⇒

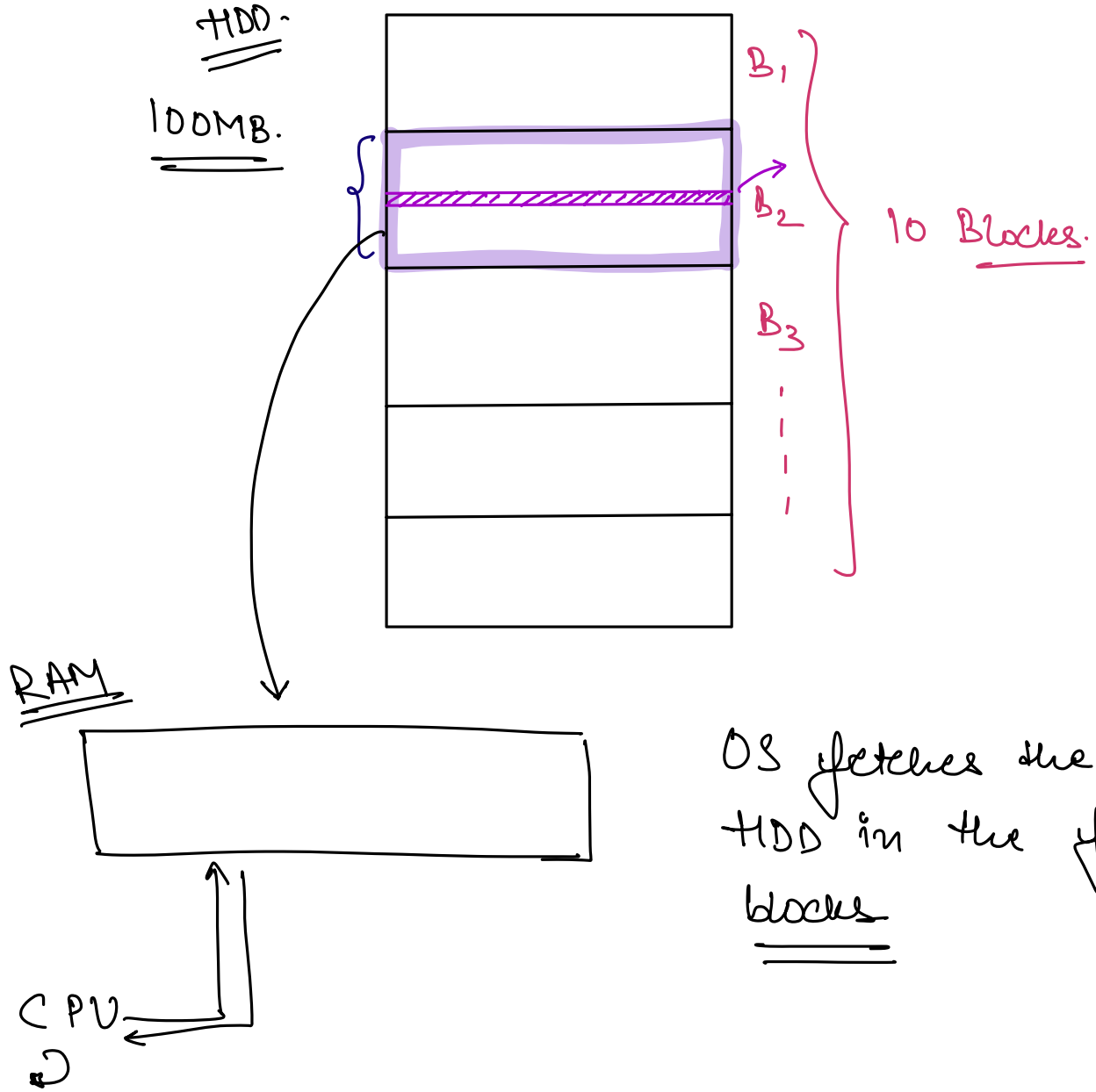
1B rows ⇒ 10^{18}

⇒ Performance will be very very slow.

(1) DB needs to do lot optimizations to make our queries faster.

(2) DB stores the data on disk.

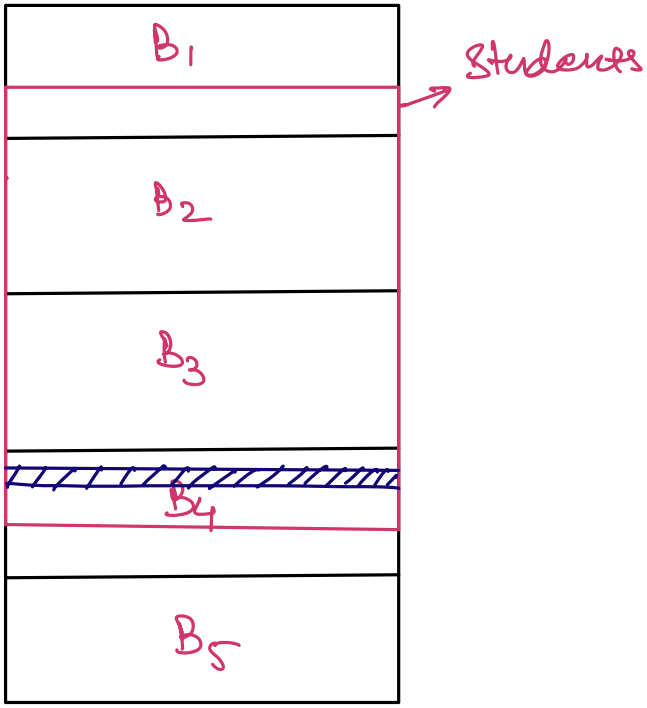
⇒ CPU can't fetch data directly from disk.
⇒ CPU can fetch data from RAM.



OS fetches the data from HDD in the form of blocks

⇒

```
Select *
from Students
where id = 100
```



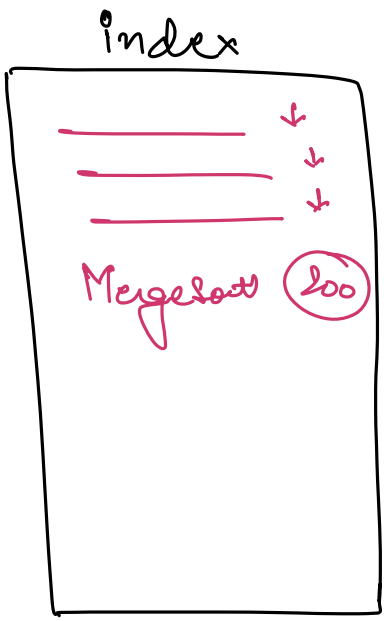
① Bring ~~B1~~ ~~B2~~ B3 to the RAM:

② Check if id = 100 is present in ~~B1~~ ~~B2~~ B3

Time ⇒ # of block accesses from disk.

If we want to optimise the overall performance of DB, then we need to optimise # of Block accesses from HDD.

INDEX.



CLRS. ≃ 1000 Pages

Index table can reduce no. of false page accesses.

index in Book ⇒ find pages faster.

index in DB ⇒ Optimises the # of block accesses.

Purpose of Indexing : Reduces the disk accesses

⇒ How Indexes work ?

Table : 100M rows.

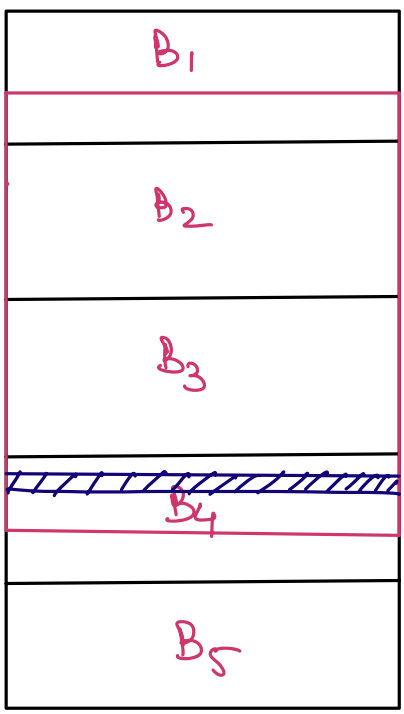
```
Select *
from Students
where id = 100
```

Index (HashMap | Map)

table

id	block
1	B_1
2	B_1
3	B_1
4	B_2
5	B_3
⋮	
100	B_4

RAM & HDD



⇒ TC: O(1) Avg.

Select *
from Students
where name = 'Abhishek';

Index => HashMap.

name	List<Block>
'Abhishek'	<1, 2, 3, 5, 10, 18>
'Priya'	<5, 6>
'Sonam'	<2, 4, 10>
'Deepak'	<6, 9>
=	=

without index => All the blocks.

with index => only 6 blocks.

Index table ≡ HashMap

↓

<key, value>

#

```
Select *
from Students
where PSP between 40 AND 80;
```

PSP → double.

40	B_1
45	B_2
55	B_3
41	⋮
⋮	
60	
65	
59	
52	
47	

40.591 50.86 70.68

Range Queries.

HashMap<Double, String>

```
for i >= 40 && i <= 80:
    if (i in HM) {
        map.get(i);
```

O(N)

HashMap / unordered_map.

$40 \le PSP \le 80$ ⇒ Ordering

↳ TreeMap | ordered_map.

Balanced BST.

Height : O(logN) ⇒ Always

10^6 ⇒ O(1)

10^6 ⇒ $\log 10^6 \approx 20$

I) Indexes work very similarly to TreeMaps.

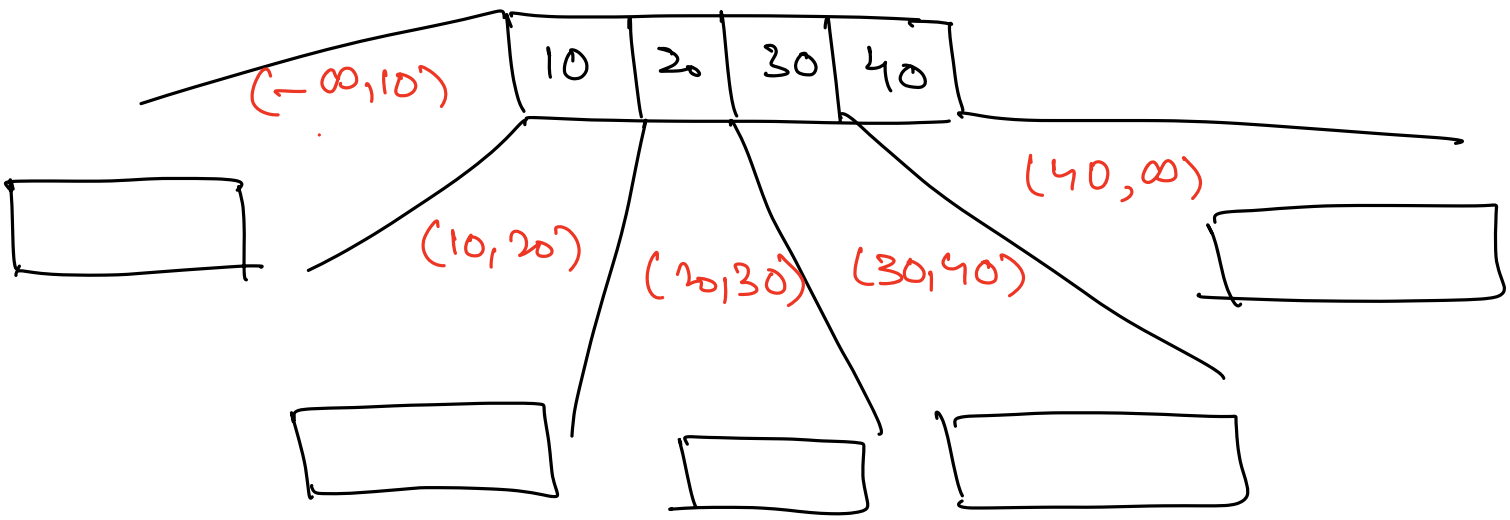
II) Index: DS that reduces the no. of disk accesses to optimise the queries.

⇒ Self Balancing BST (<= 2 children)

- AVL
- Red Black Tree.

⇒ logN

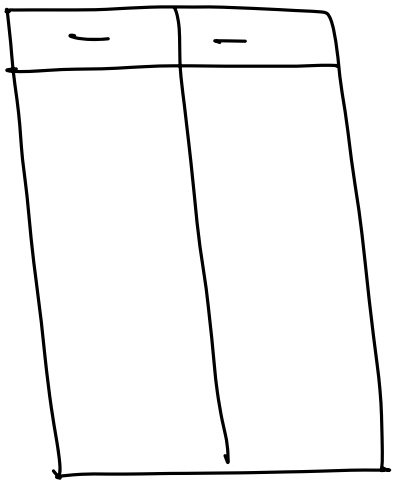
⇒ B | B+ Trees. (<= x children)

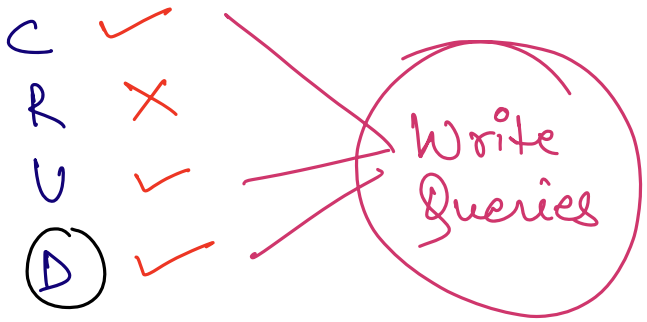


H < logN.

CONS of indexing.

Index table





⇒ Writes becomes slower.

⇒ Storage also increases.

Note: Don't create indexes blindly.

Create an index only when we start getting frequent queries on any table for a column.

Index of Multiple columns.

(id)

(name)

(name, psp) ✓

I) (name, psp)

II) (psp, name)

(name, psp) ⇒ TreeMap

Sorted Map

Key for Map

(name, psp)	Block
('Abhishek', 80)	1
('Abhishek', 90)	1
('Abhishek', 92)	2
(Ajay, 20)	3
(Bijoy, 80)	1
(Deepak, 60)	1
(==)	1
(==) 80	1

```
select *
from Students
where name = 'Abhishek'
```

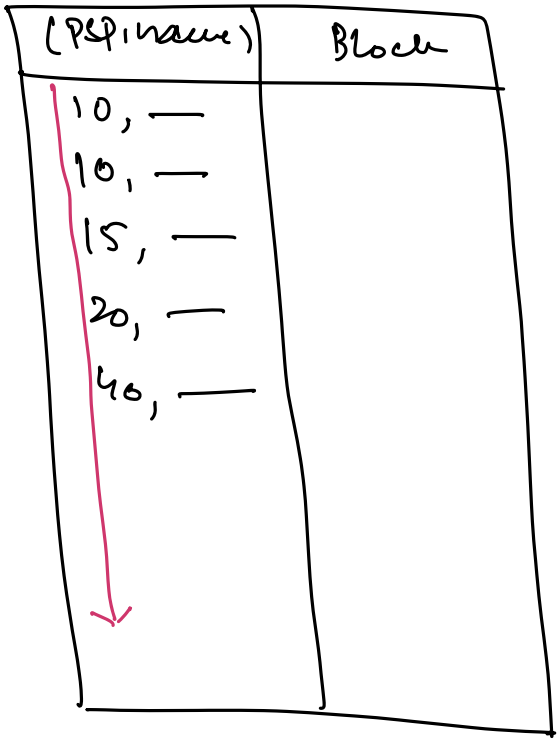
✓

```
select *
from Students
Where psp = 80;
```

✗

Index on (name, psp) won't optimise queries based on psp.

(psp, name)




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

```
Select *
from students
where name = Abhishek AND
psp = 80;
```

⇒ (name, psp)

500

⇒ Index	Query	Help / Not Help.
name	psp	✗
(name, psp)	name	✓
(name, psp, attendance)	name	✓
(name, psp, attendance)	(name, psp)	✓

(name, psp, attendance)	(psp)	X

If a query is on column (X) any index with prefix (X) would optimise the query

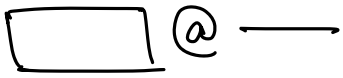
⇒ Index on Strings.

```
Select *
from Students
where email = ___
```

email	Block
deepak@scaler.com	1
tirnavsen@gmail.com	2
soham@yahoo.com	3
___	
___	
___	

1M entries

B|B+ Trees.



email	Block
deepak	[2,8,5,4]
himanshu	[1,4]
Soham	[2,3]
gaffar.	⋮
⋮	

⇒ < 1M entries.

⇒ Space (VS) Optimization.

⇒ (name)

```
Select *
from Students
where name = ____________;
```

⇒ Index on name Column

⇒ Ideally when we have to create index on strings Column, it's better to create index only on first few characters.

prefix.

⇒ first (5) Characters

⇒ (2B) users.

index on how many Characters	# of users	# of unique entries	No. of Block accesses.
1	2×10^9	26	$2\times10^9 \mid 26$ (Avg).
2	2×10^9	26×26	$2\times10^9 \mid 26\times26$
3	2×10^9	$26\times26\times26$	$2\times10^9 \mid 26\times26\times26$
⋮	⋮	⋮	⋮

ab → 2B | 26×26
ac →
ad →
⋮

a → 2B | 26
b →
c →
d →

abc
abd
abe
|

Agenda:

→ Transactions.

→ Properties of Transactions:

A

C

I

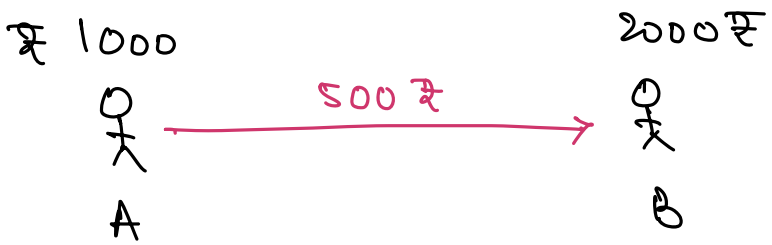
D

→ Commit & Rollback

→ Isolation Levels.

⇒ What are Transactions?

Bank Server.



accounts.

id	name	balance	type	- - - - - - -
1	A	1000 500	Savings	
2	B	2000 2500	Saving	

Transfer Money :

i) Get balance of A from DB ⇒ Select

ii) Check balance >= 500 ⇒ Application.

iii) Reduce balance of A by 500₹ ⇒ Update

iv) Add 500₹ to (B). ⇒ Update

A $\xrightarrow{500₹}$ B

C $\xrightarrow{500₹}$ B

Transfer Money operation might be getting executed for multiple users at the same time.

`x += 10` ⇒ Not a single op^n.

$x = x + 10$.

```
temp ← x
temp ← temp + 10
x ← temp.
```

A: ~~1000~~ 500

B: 5000 ~~5500~~ 6000

C: ~~4000~~ 3500

A $\xrightarrow{500}$ B

1000
(1) X ← read A
500
(2) X ← X − 500
500
(3) A ← X

(4) X ← B
5000
(8) X ← X + 500
5500
(9) B ← X

C $\xrightarrow{500Z}$ B

4000
(5) X ← read C
3500
(6) X ← X − 500
(7) C ← X

5500
(10) Y ← B
(11) Y ← Y + 500
6000
(12) B ← Y

A: ~~1000~~ 500
B: ~~5000~~ ~~5500~~ 5500
C: ~~4000~~ 3500
} Inconsistent.

A $\xrightarrow{500}$ B

1000
(1) X ← read A
500
(2) X ← X − 500
500
(3) A ← X
5000
(4) X ← B
5500
(8) X ← X + 500
(11) B ← X

C $\xrightarrow{500Z}$ B

4000
(4) Y ← read C
3500
(5) Y ← Y − 500
3500
(6) C ← Y
5000
(9) Y ← B
5500
(10) Y ← Y + 500
5500
(12) B ← Y

⇒ If multiple operations happens on same data at the same time it can lead to inconsistent.

Transaction: Set of DB operations logically grouped together to perform a task.

transferMoney :-

i) Get balance of A from DB ⇒ Select
ii) Check balance >= 500 ⇒ Application.
iii) Reduce balance of A by 500₹ ⇒ Update
iv) Add 500₹ to (B) ⇒ Update

→ transaction.

⇒ Expections | Properties of a Transactions :

A : Atomicity

C : Consistency

I : Isolation

D : Durability

ATOMICITY.

↓

Smallest indivisible unit.

⇒ Transaction is the smallest unit.

⇒ To the end user transaction should appear single indivisible operation.

⇒ Either the complete transaction should happen or nothing should happen.

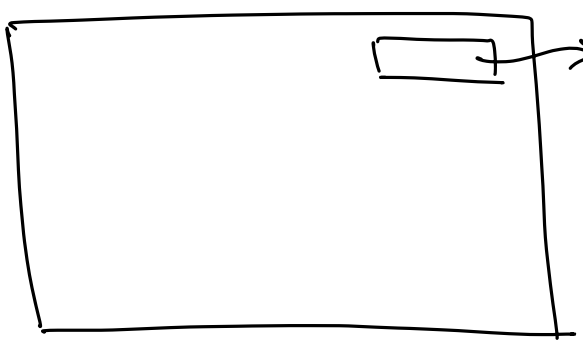
⇒ Any transaction should not stop at any intermediary step.

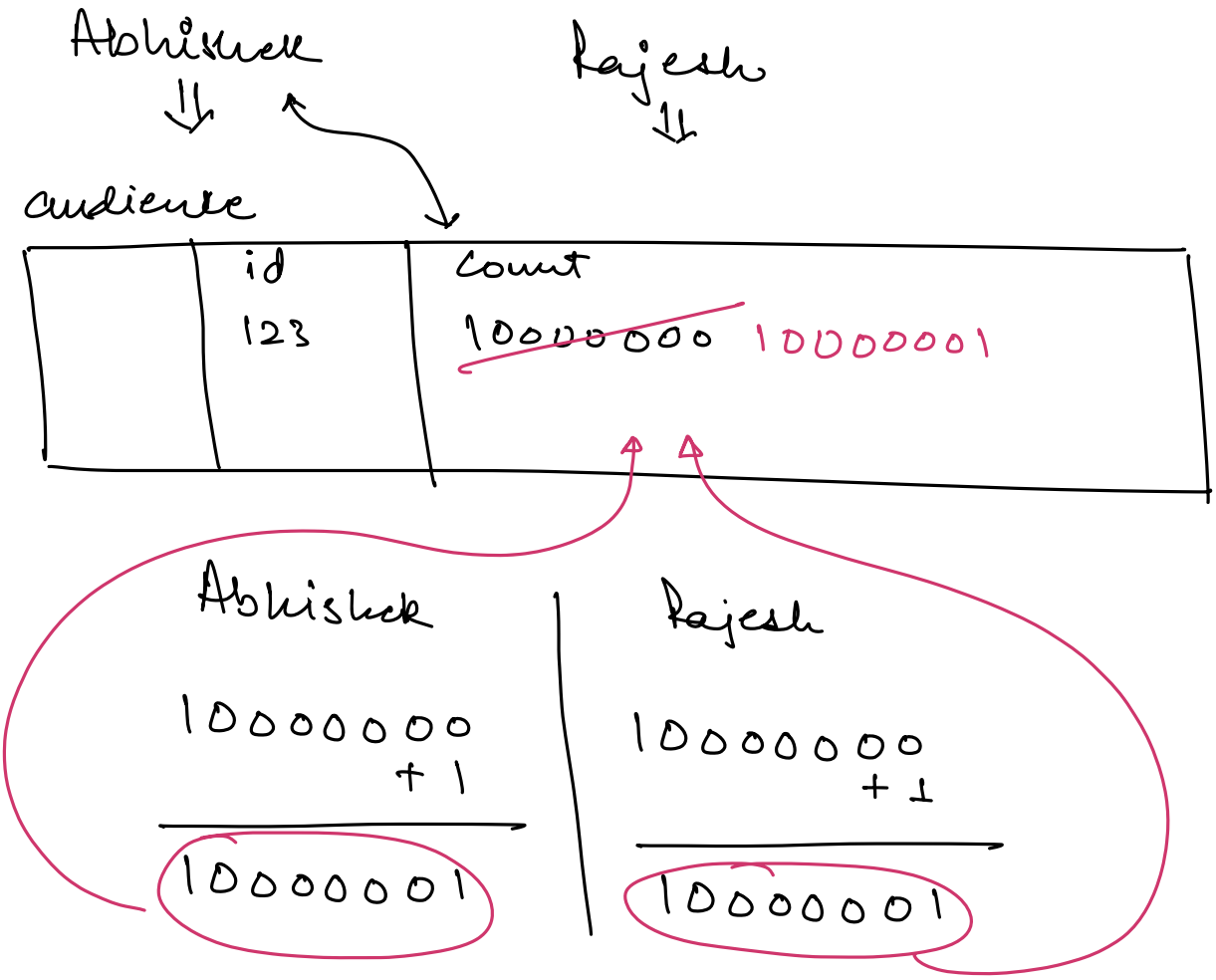
CONSISTENCY.

→ Data correctness.

→ Database should be in consistent state before & after the transaction.

Hotstar.





35000000 ⇒

35000100 ⇒

ISOLATION.

One transaction should not affect the execution of other transactions running at the same time on same data.

A $\xrightarrow{x}$ B ⇒ T1

C → B ⇒ T2

} isolated.

A → B ⇒ T1

C → D ⇒ T2

DURABILITY.

↳ Persistence.

⇒ Once a transaction has completed we would want its changes to persisted in the DB permanently.

⇒ Write the changes to disk.

Commit & Rollback.

Students.

id	name	psp	- - -
1	Deepak	70	

update Students
set psp = 80
where id = 1;

⇒

Select *
from Students
where id = 1;

⇒ 80

autocommit. → 0
→ 1

Every SQL query starts a transaction by by default and it auto commits the changes

Starts query
executes
Commit

↳ We'll have to Commit the changes done by a transaction on our own.
Keyword.

Transaction Isolation levels.

⇒ Most relaxed

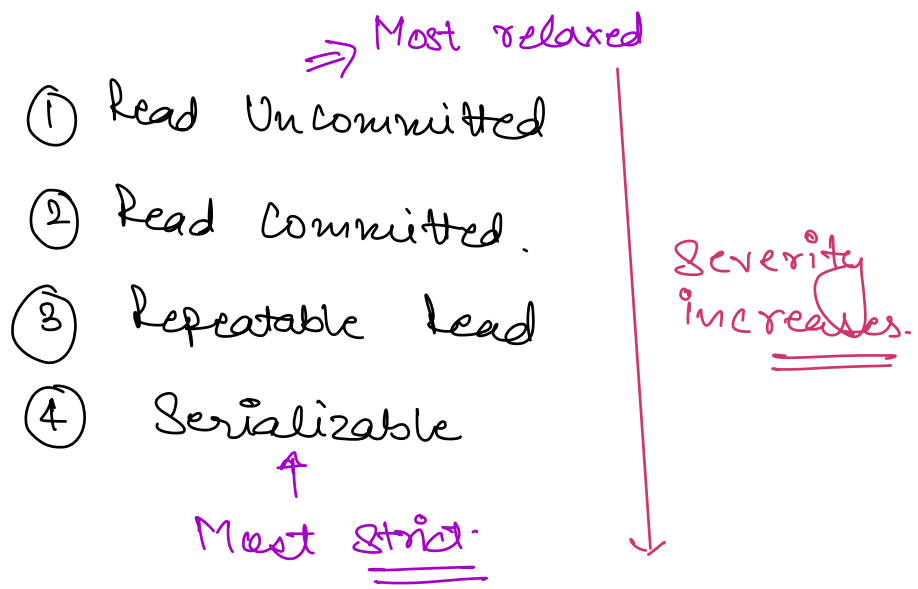
1. Read Uncommitted
2. Read Committed.
3. Repeatable Read
4. Serializable

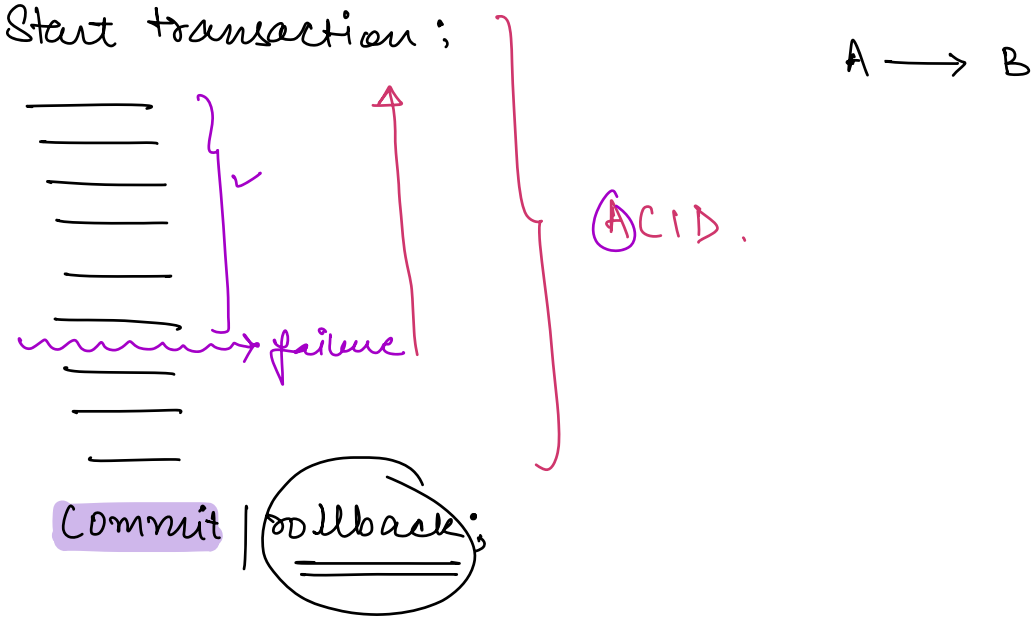
Severity increases.

↑
Most strict.

*

Transaction Isolation levels.

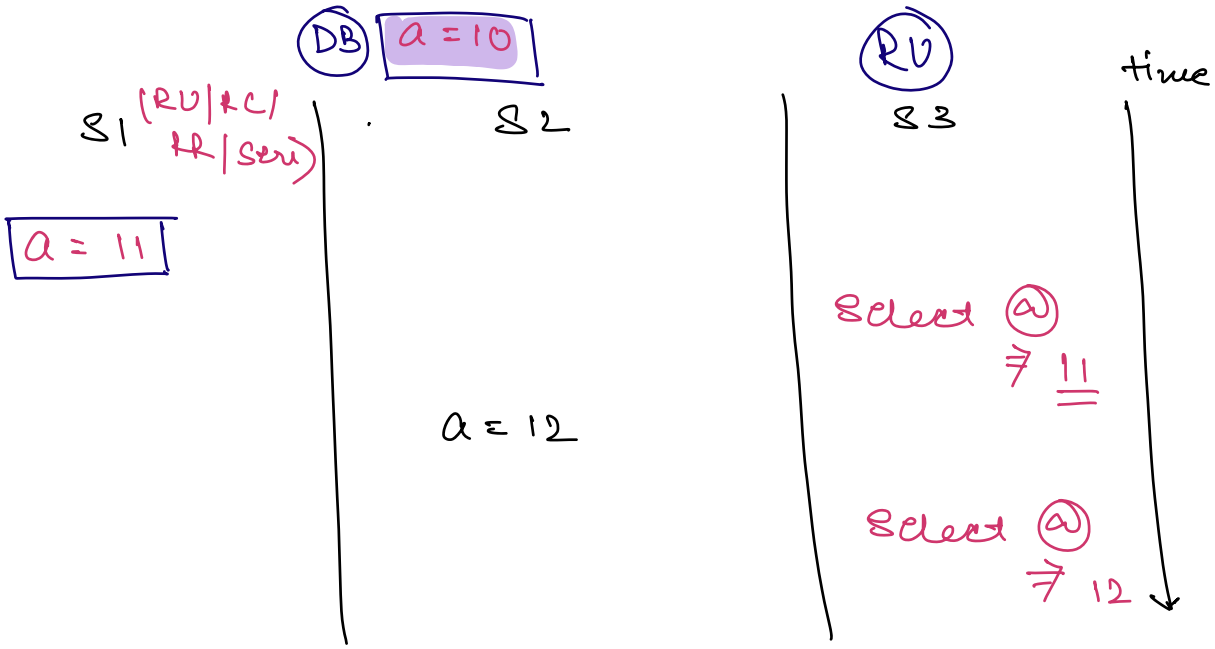




1. By default autocommit = true
2. If we start a transaction then we'll have to commit on our own.

Read Uncommitted :-

⇒ Allows us to read the latest uncommitted.



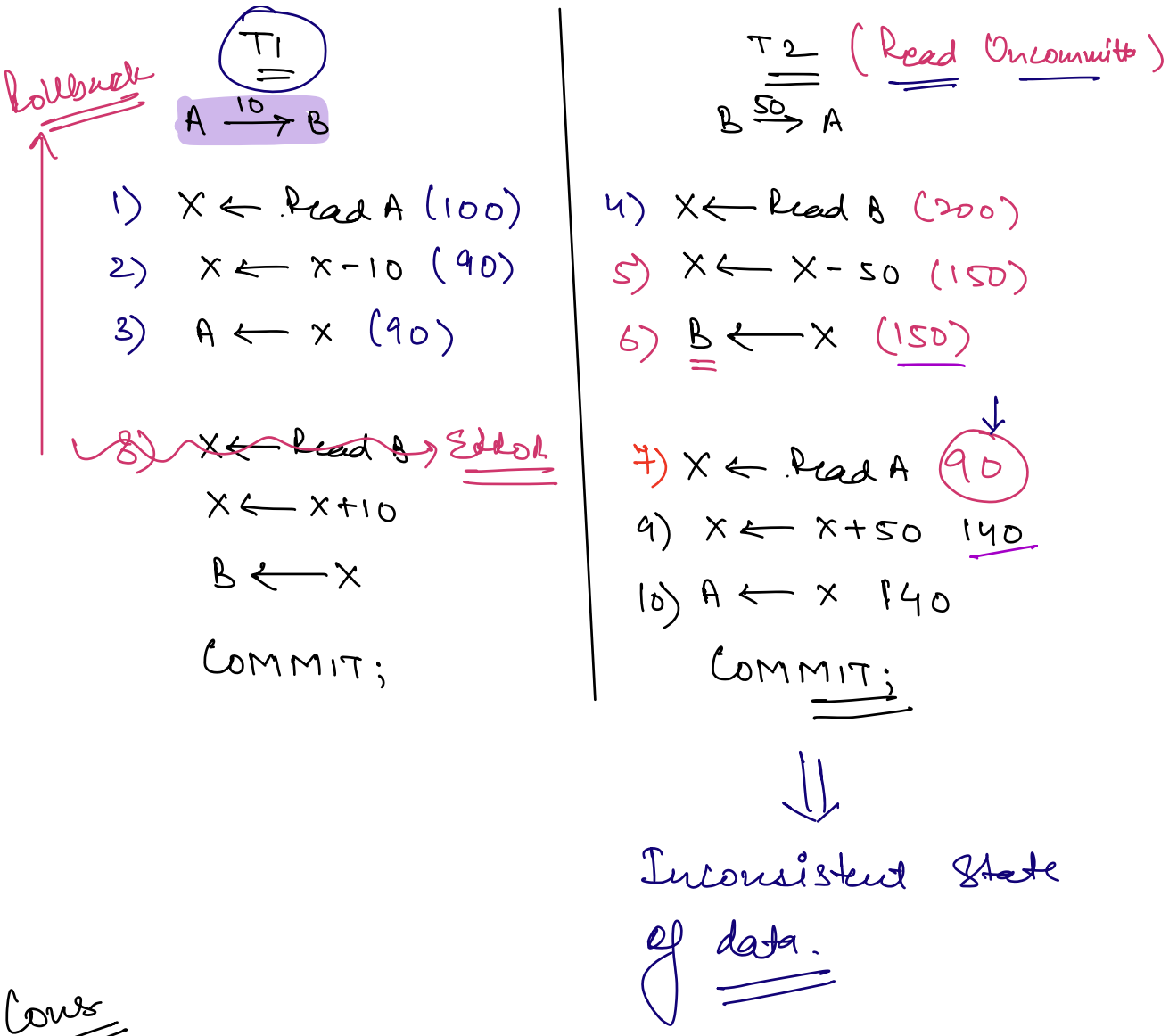
Note: The value that we are going to read will depend on our isolation level & not on other's isolation level.

⇒ MySQL has Repeatable Read isolation level.

ACID.

A = ~~100~~ 140

B = ~~200~~ 150



Cons

⇒ DIRTY READ.

It can lead to inconsistent | wrong DB state if we read uncommitted data.

Pro: fast

Read Committed.

⇒ Transaction will always read the committed data from disk.

⇒ No dirty read problem.

⇒ Stricter than Read Uncommitted.

Students

id	name	Psp	voucher_sent
1	A	~~82~~ 79	false
2	B	75	false
3	C	85	false
4	D	40	false
5	E	60	false

Q. Send voucher to all the students with psp > 80.

List<Students> ⇒ Select *
from Students
where psp > 80;

Vouchers Sent. ⇒ 20 mins

```
Update Students
set Voucher_sent = true
where Rsp > 80;
```

Read Committed. ⇒ Student (A) won't come in the output of T2 because Rsp < 80.

⇒ At both the places we should read the same data.

⇒ We should read the Repeated Data.

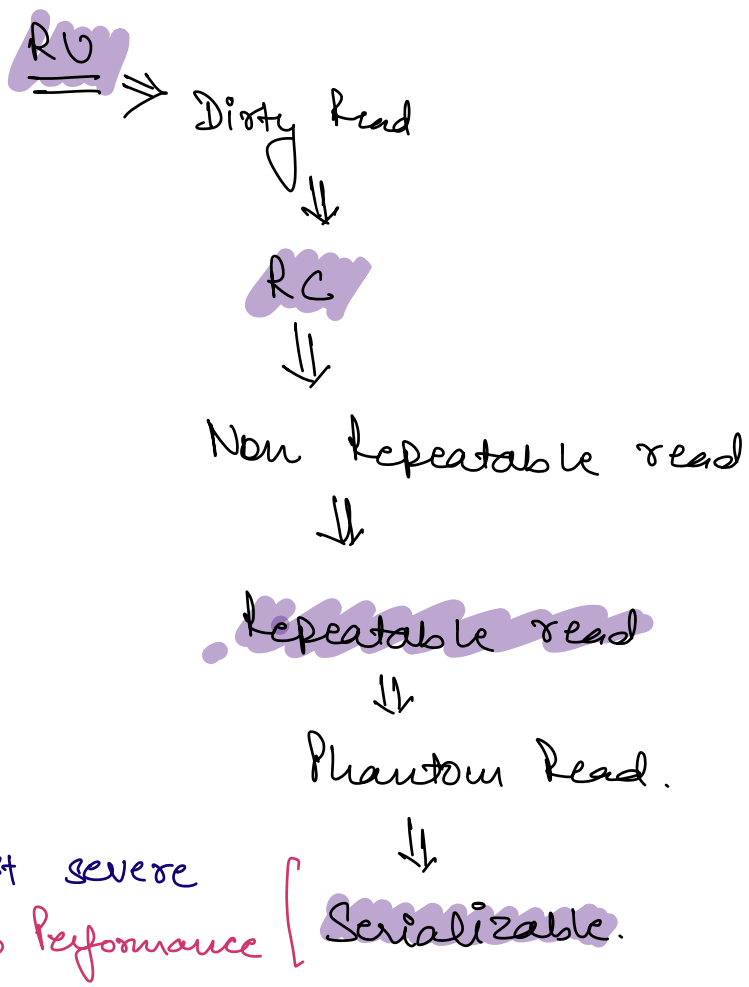
⇒ Problem : Non Repeatable Read.

Repeatable Read.

⇒ In a transaction once we have read that we should read the same data again

Problem.

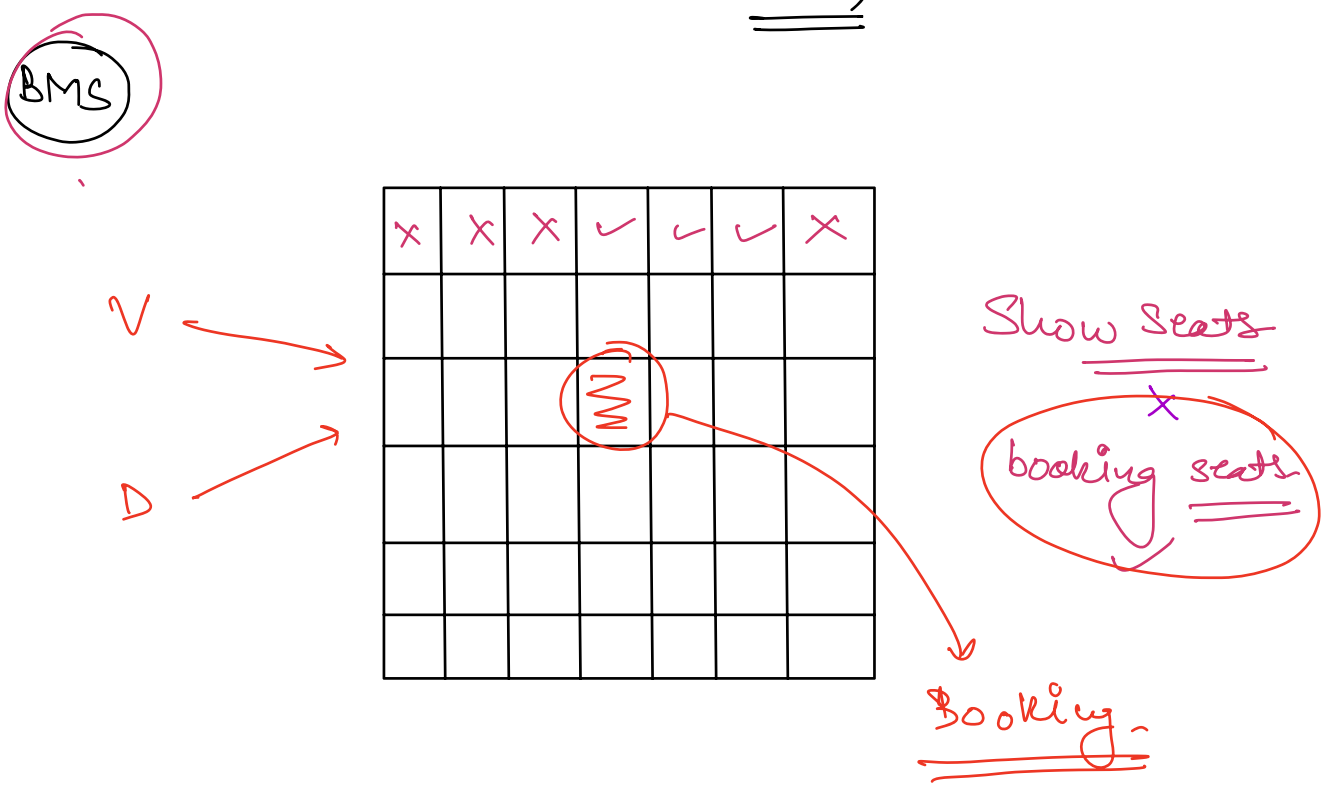
PHANTOM READ. : A row came magically from nowhere at the time of Update.



Serializable.

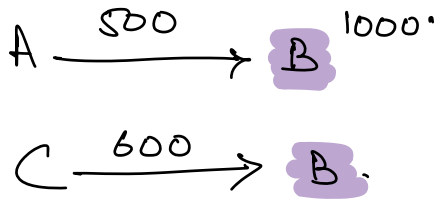
Repeatable Read + Checks for locks even for read operation.

(If lock is present while reading then others won't be even able to read the data).



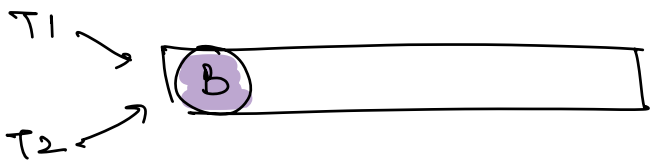
⇒ A —500→ B (T1) | C —100→ D (T2)

⇒



B ⇒ ~~1000~~ 1600

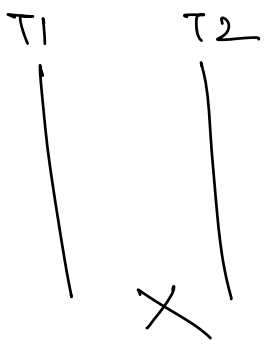
Serializable

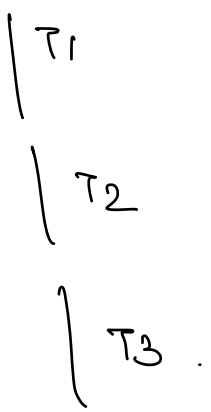


B = 1000

(T1) A —500→ B ⇒ B = 1500

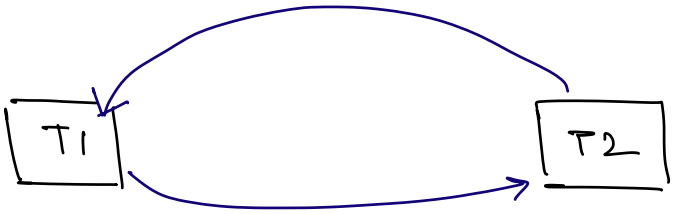
(T2) C —600→ B ⇒ B = 2100.





⇒ One transaction can only happen on any set of data any given point of time.

#



- Lock on id = 1
- Trying to read id = 2
- Trying to read id = 1
- Lock on id = 2

⇒ Deadlock.

⇒ MySQL will handle deadlocks by cancelling one of the transaction.

T1

```
Select *
from ——
where id = 1
for update;

Select *
from ——
where id = 2
```

T2

```
Select *
from ——
where id = 1

Select *
from ——
where id = 2
for update;
```

Deadlock

Agenda:

1. What is Schema Design.
2. How to approach Schema Design.
3. Cardinalities
4. How to find cardinalities.

Schema Design. ≡ Database Design.

- Structure of tables.
- Which all the tables.
- Columns in a tables
- Primary Keys
- foreign Keys.

⇒ Benefits of Schema Design.

- Performance
- Reduce redundancy.
- Reduce Inconsistency

How to Approach Schema Design.

SCALER.

1. Scaler will have multiple batches, for each batch we'll store its name, start month and current instructor.
2. Each batch will have multiple students.
3. Each batch will have multiple classes.
4. for each class we need to store their name, date & time, instructor.
5. for each student we'll store their name, email, phone, grad Year, university name, psp.
6. Every student will have a buddy, who is another student.
7. A student can be moved from one batch to another batch.
8. for each batch a student goes to we'll store the start date of the batch.
9. Every student can choose their mentor, for mentor we'll store their name, email & current company.
10. We'll store the information regarding mentor sessions like session_id, time, duration, student, mentor, student rating, mentor rating

11. for every batch we'll store if it is Academy & DSML

STEPS.

(1) Create the tables.

⇒ How to identify tables.

→ Find out all the Nouns in the requirements & check if we need to store data.

Conventions about names

1) Table names should be plural.

Students
mentors
batches.
mentor_sessions.

|

MentorSessions.

2) Column name ⇒ Singular.

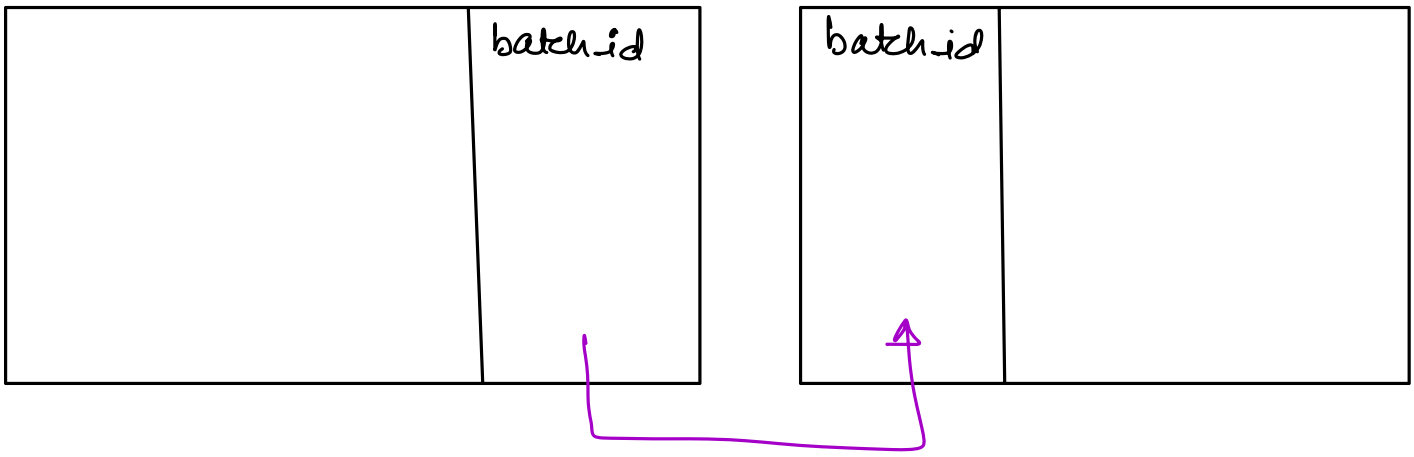
3) Primary keys.

batches.

id	batch_id

Students.

batches



Tables.

batches

batch_id	name	start month	

instructors

instructor_id	name	email	avg_rating	..

Students

Student_id	email	phone	grad_year	univ_name

Classes.

class_id	name	date	

mentors

mentor_id	name	email	company	

mentor_sessions

mentor_session_id	time	duration	mentor_feedback

Student_feedback.

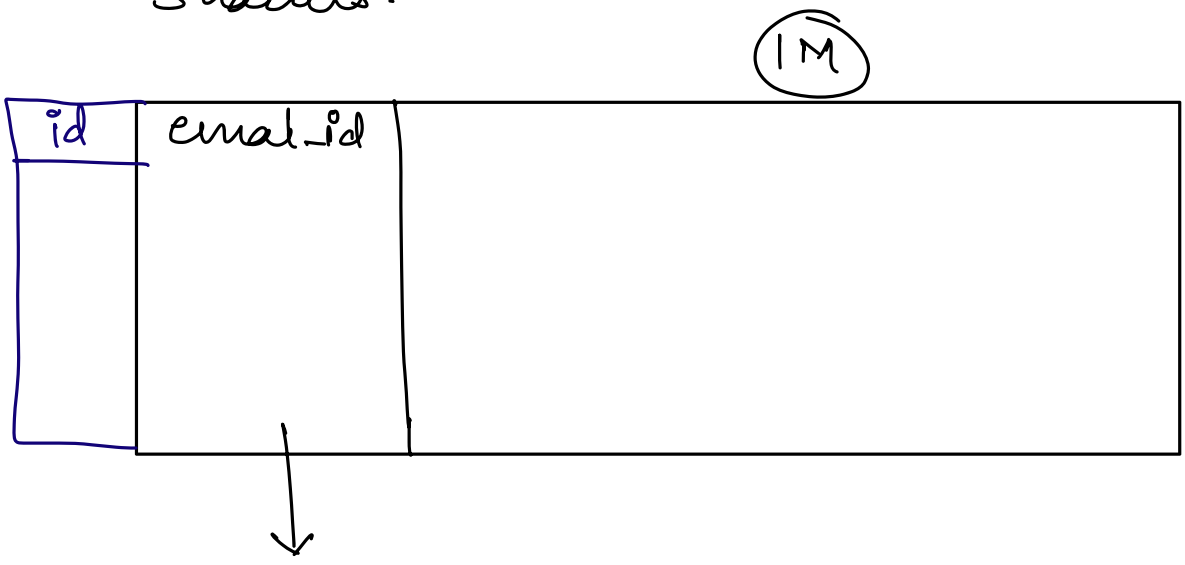
Constraints.

⇒ email_id ⇒ PK.

↳ Can be a PK ✓

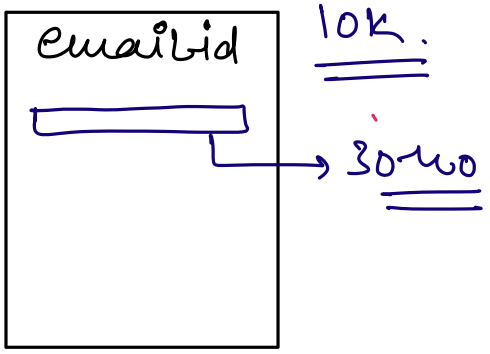
Reasons for why not to choose email as a PK

① Students.



email_id's

⇒ Space Constraint



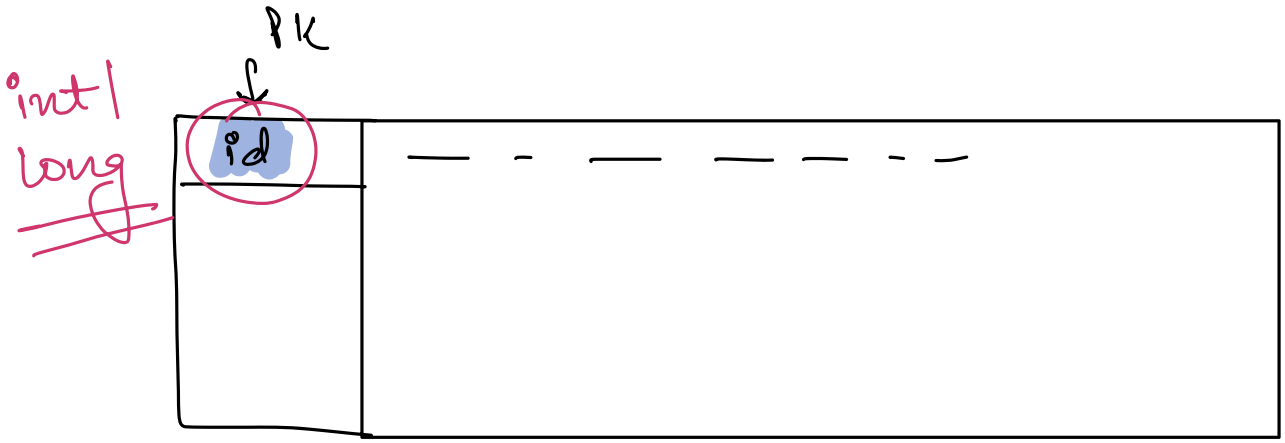
⇒ Rearrangement.

⇒ User's data can change.

(By default DB creates an Index on PK).

Sorted.

⇒ Strings expensive compared int.



⇒ How to represent relations.

CARDINALITY

A B

⇒ How many A's are related to how many B's.

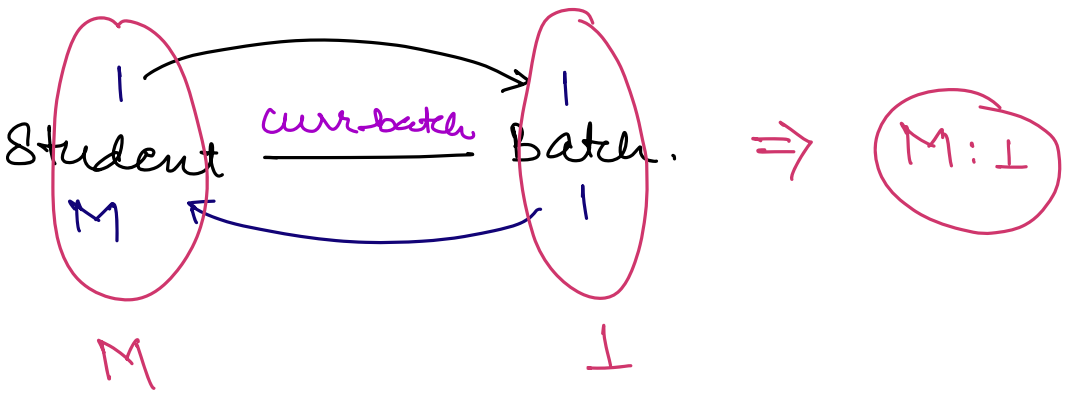
1 : 1 ⇒ 1 (A) is related to 1 (B)

1 : M ⇒ 1 (A) is related to Many (B's)

M : 1 ⇒ 1 (B) is related to Many (A's)

M : M ⇒ M (A's) are related to Many (B's)

Eg



⇒ Which relation?

(1): An entity can be associated to [0,1] of other entity

M: (>1) [2, ∞)

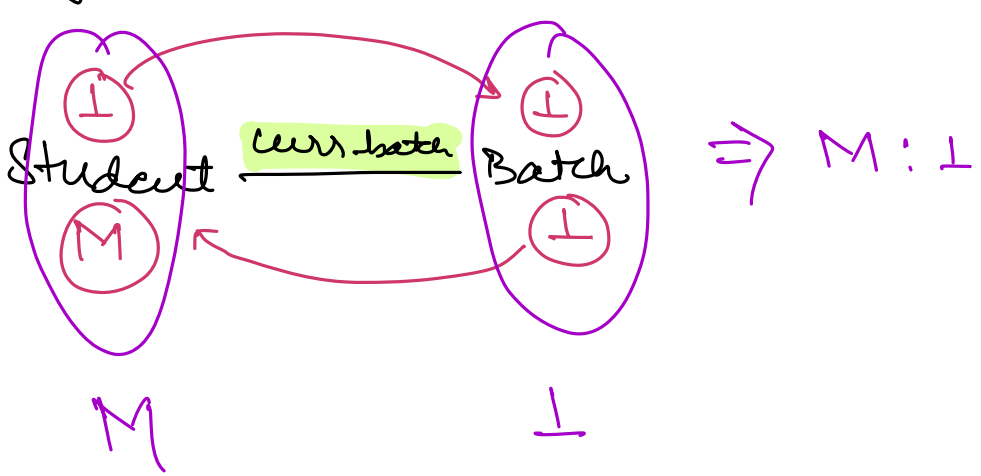
⇒ How to find cardinality

(1) Define the relation b/w the entities for which we want to find cardinality

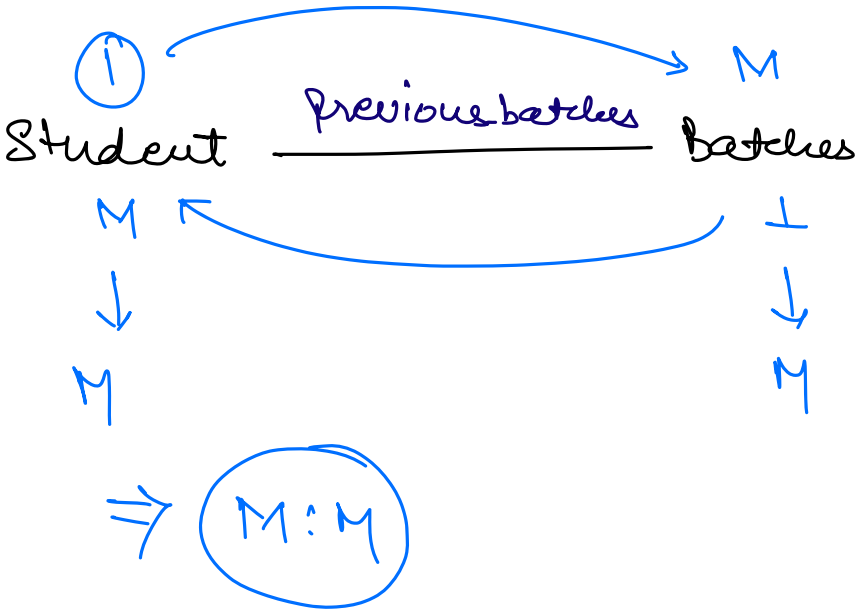
(2) 2 step approach

i) Write (1) on left side & find how many of right side entities are associated to (1) entity of left side

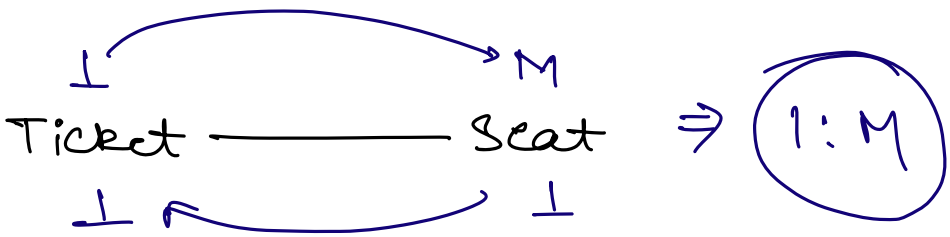
ii) Write (1) on right side & find how many of left side entities are associated to (1) entity of right side

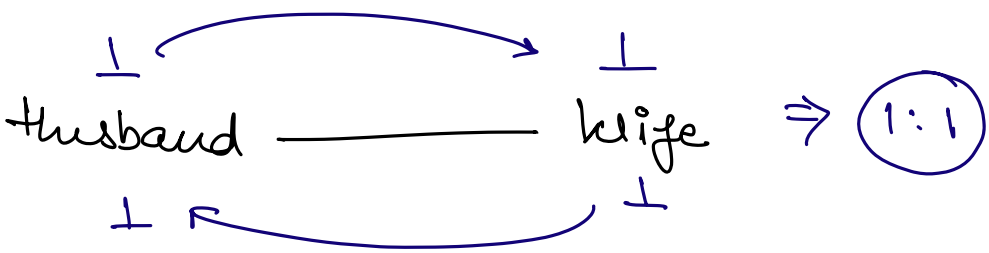


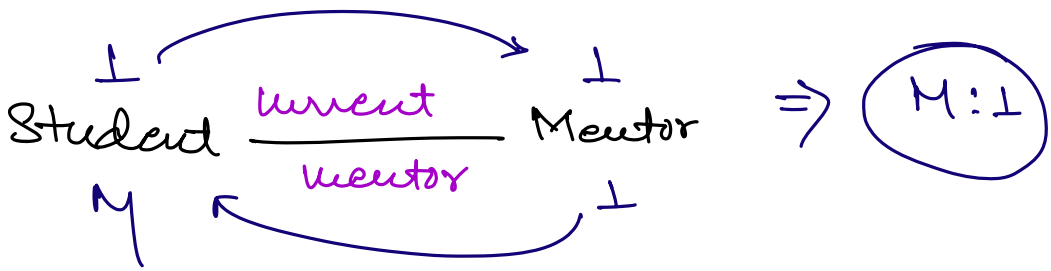
⇒ If there's any (M) on a side then take (M) else take (1)

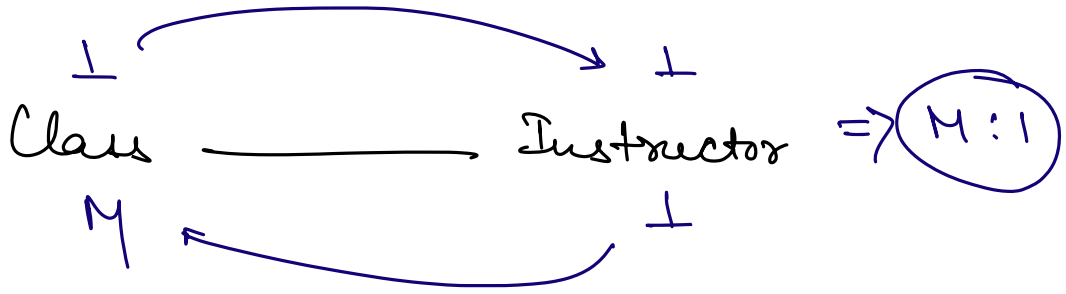


BookMyShow.



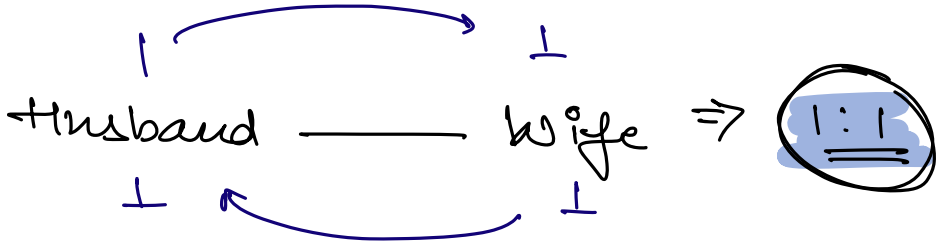


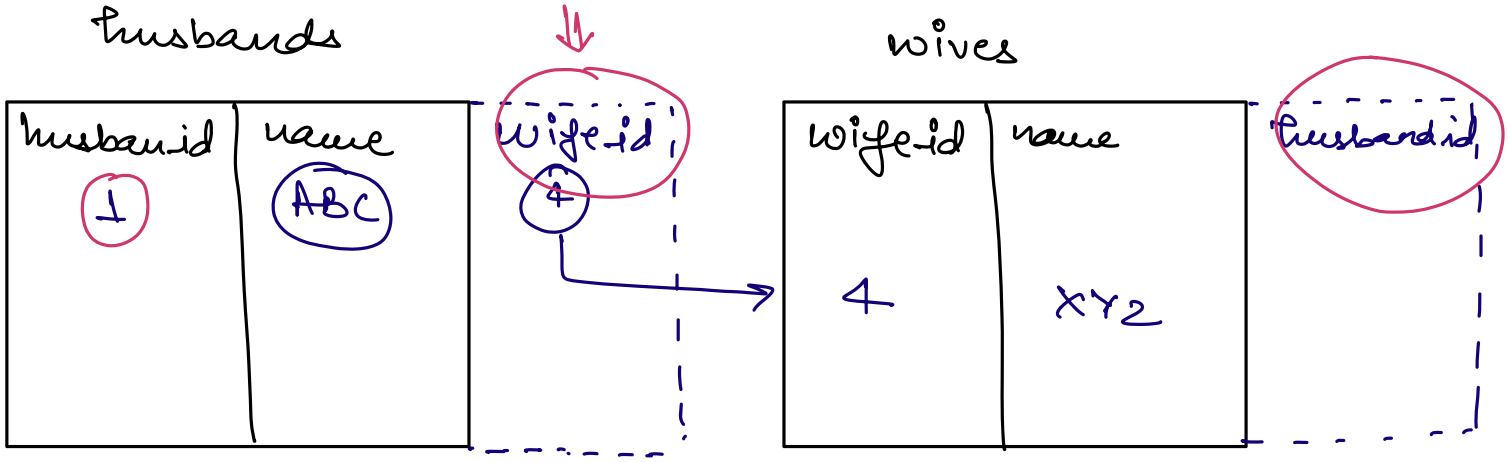




⇒ how to represent cardinalities.

1) 1:1





husband_id in wives table.

OR

wife_id is husbands table.

⇒ Can we do both? Yes

⇒ Should we do both? ⇒ NO

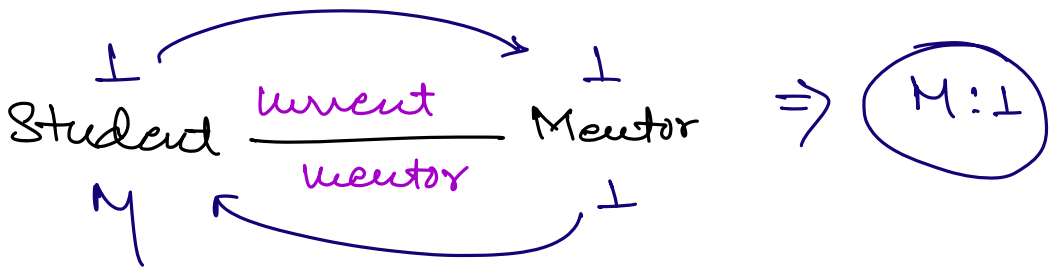
→ Space.

→ Redundancy.

⇒ If we are storing same data at 2 different places then it can lead to inconsistency

⇒ Update Anomaly

2) 1 : M / M : 1



Students

id	name	mentor id
1	Vinu	10

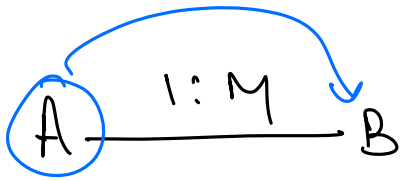
mentors

mentor id	Student ids
5	[2,4,6,7,8]

⇒ Every cell should be Atomic.

Student — M:1 — Mentor

⇒ Id of (1) side on (M) side.



⇒ Id of (A) in (B) table.

3) M : M.

Orders

id	Product_ids.
1	[10, 11, 12]

Products

id	Order_ids
5	[5, 6, 7...]

⇒ Mapping table. | lookup table.

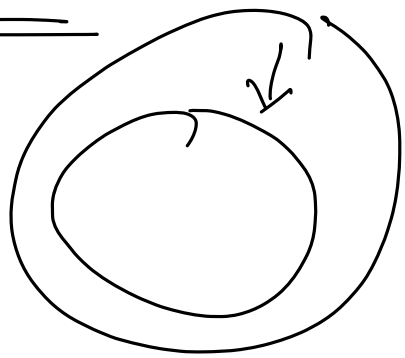
Orders_products.

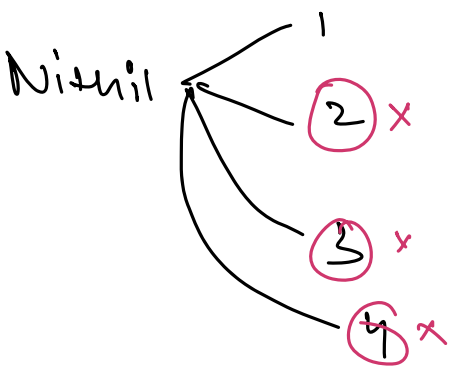
order_id	product_id
1	10
1	11
1	12
2	10
3	10
4	10

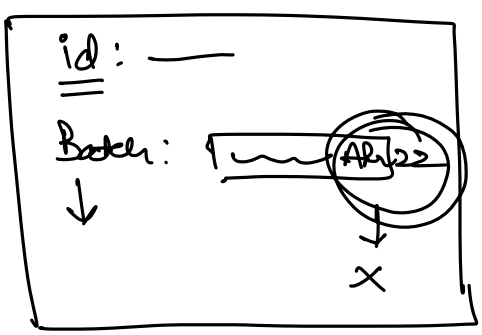
—— * ——

Happy Diwali

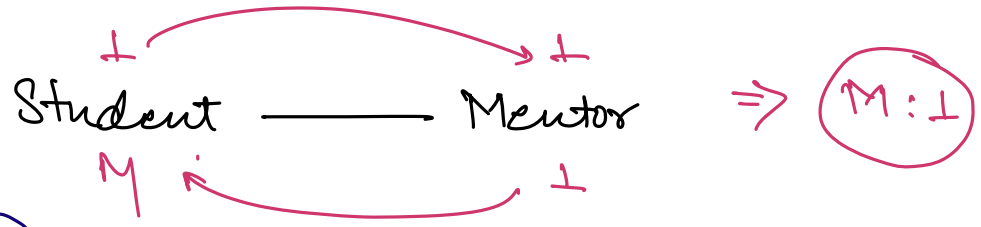
draw.io.

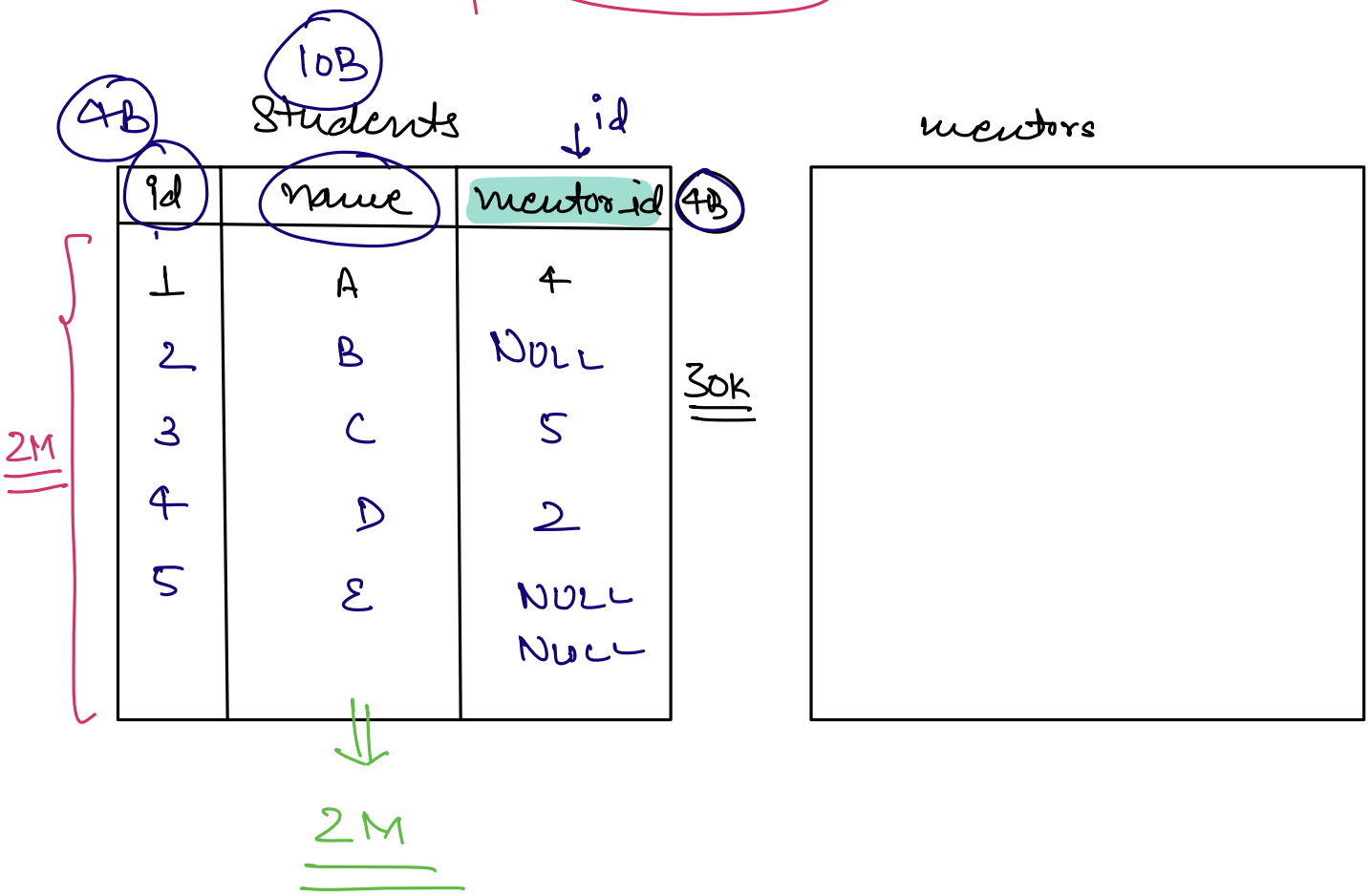






* 1:M | M:1.





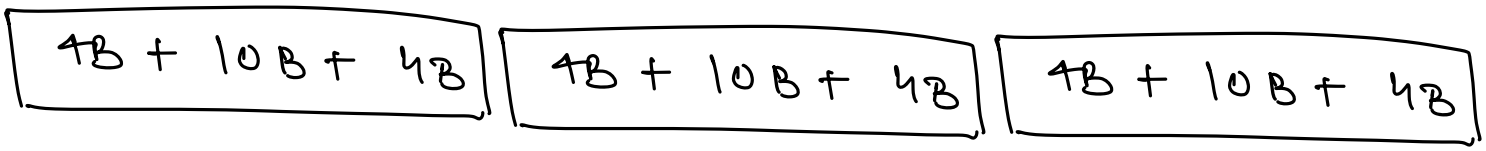
⇒ 30K.

⇒ 1970000.

⇒ 1.97M rows will have mentor_id as NULL.

⇒ Sparse relation.

⇒ Space wastage.



⇒ Mapping / lookup table.

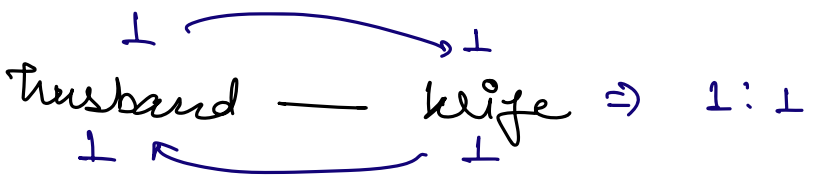
Students_mentors

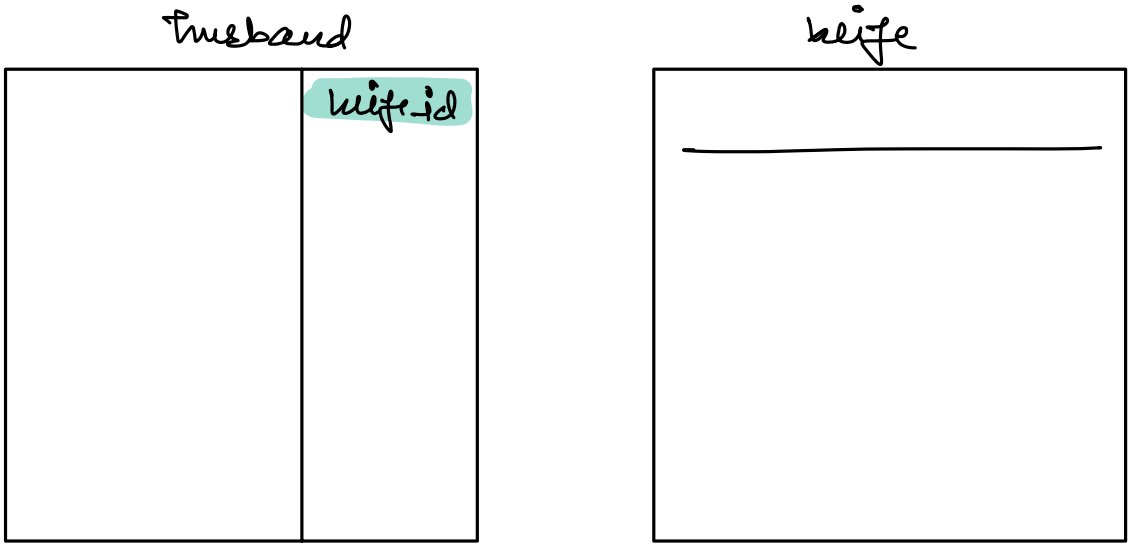
Student id	mentor id
1	10
2	5
6	7
—	—
—	—
—	—

⇒ Pro.: No space wastage.

Cons: Joins.

⇒ (1:1)





husband

mid

id	name	wife_id	marriage_date	eng_date	Status	...
1	A	10				
2	B	4				
3	C	5				
4	D	6				
10						

SRP : Single Responsibility Principle

⇒ Mapping table.

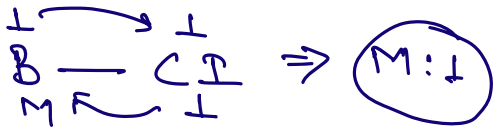
marriage

id	h_id	w_id	marriage_date	—	— — —
1	10	5	2021		
		?			

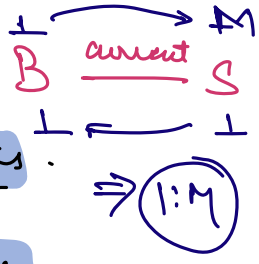
Mapping / lookup / Extra.

Cardinalities	Normal	Sparse	Rel^n has attributes
1 : 1	Id of one side on other side	Mapping Table.	Mapping Table.
1 : M / M : 1	Id of one side on (M) side.		
M : M	Mapping table.		

SCALER.



1. Scaler will have multiple **batches**, for each batch we'll store its name, start month and current **instructor**.



2. Each batch will have multiple **students**.
3. Each batch will have multiple **classes**.
4. for each class we need to store their name, date & time, instructor.
5. for each student we'll store their name, email, phone, grad Year, university name, psp.
6. Every student will have a buddy, who is another student.
7. A student can be moved from one batch to another batch.
8. for each batch a student goes to we'll store the start date of the batch.
9. Every student can choose their **mentor**, for mentor we'll store their name, email & current company.
10. We'll store the information regarding **mentor sessions** like session_id, time, duration, student, mentor, student rating, mentor rating

11. for every batch we'll store if it is Academy & DSML

Tables.

batches

batch_id	name	start month	current_ins_id

instructors

instructor_id	name	email	avg_rating	..

Students

student_id	email	phone	grad_year	univ_name	batch_id

buddy_id | mentor_id

Classes.

class_id	name	date	instructor_id

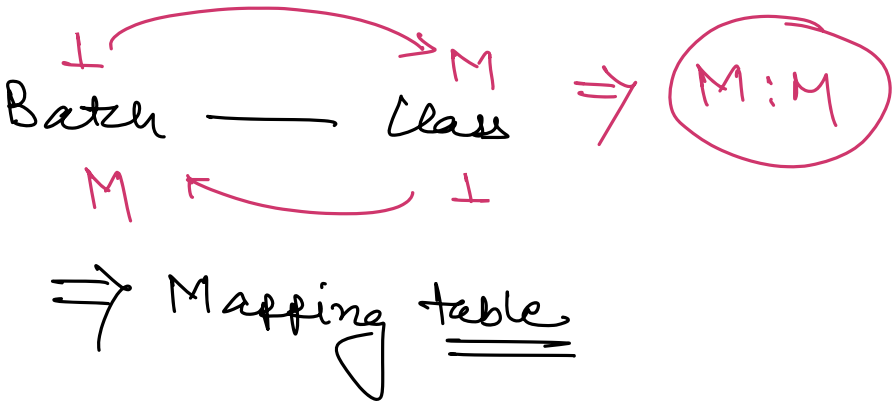
mentors

mentor_id	name	email	company	

mentor_sessions

mentor_session_id	time	duration	mentor_feedback

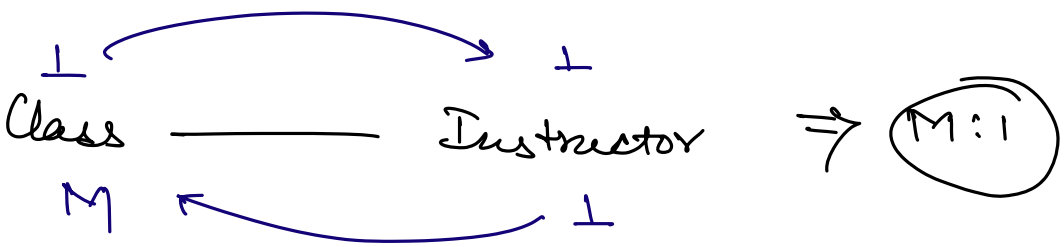
Student_feedback.

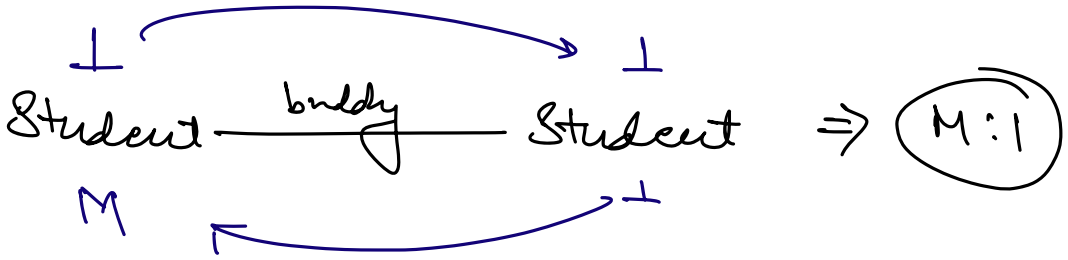


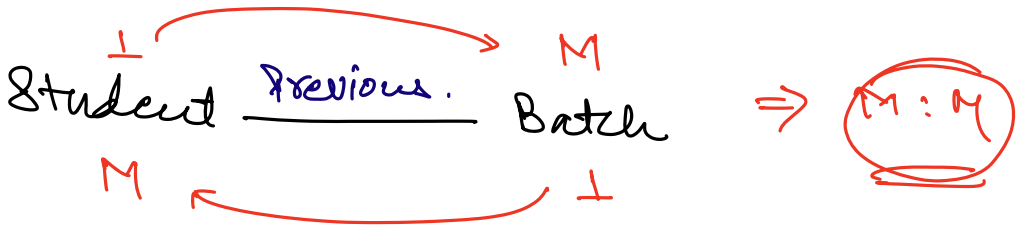
⇒ Mapping table

batches_classes

batch_id	Class_id

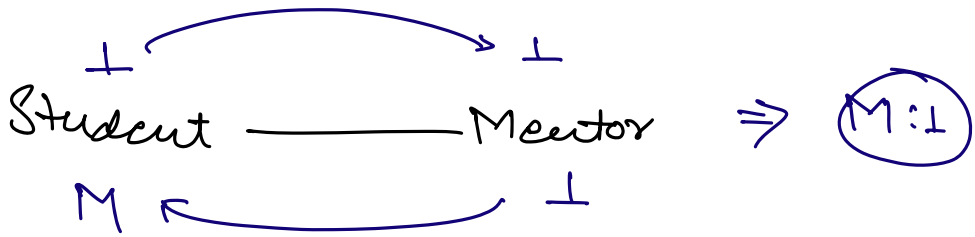






Students_previous_batches.

st_id	b_id	Start_date.	batch_level_attr
1	10	3/01/2022	80
1	12	3/01/2022	90



⇒ Drop Down.

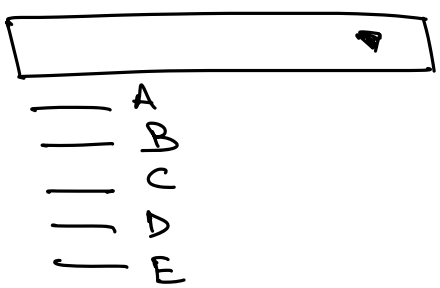
Enum

Enum Gender {

MALE, 110

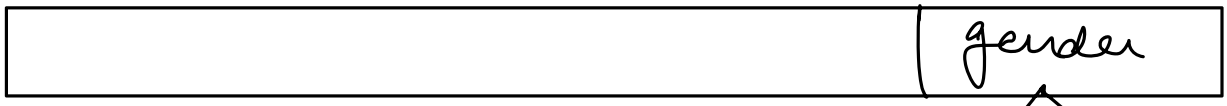
FEMALE 111

}



3

Users.



I String
II int

MALE → 0
FEMALE → 1

⇒ Space ✓
⇒ String Comparison ✓

```
Enum BatchType {
    A ,110
    B , 111
    C , 112
    D , 113
    E , 114
}
```

3

```
Enum BatchType {
    A 110
    B 111
    D 112
    E 113.
}
```

3

ENUM : Predefined set of values

batch_types.

lookup table

id	type
1	A
2	B
3	C
4	D

Students

	batch_type
	1
	3
	4
	~~2~~ NULL

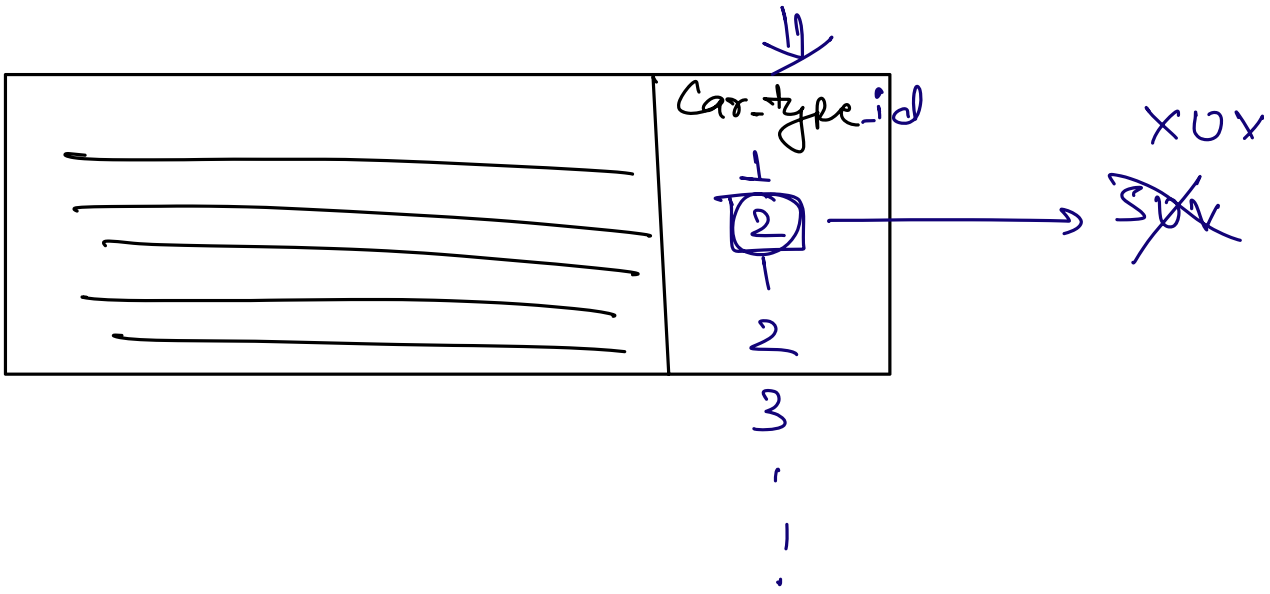
Users.

1M

	Car_type
	SEDAN
	SUV
	HB
	SUV

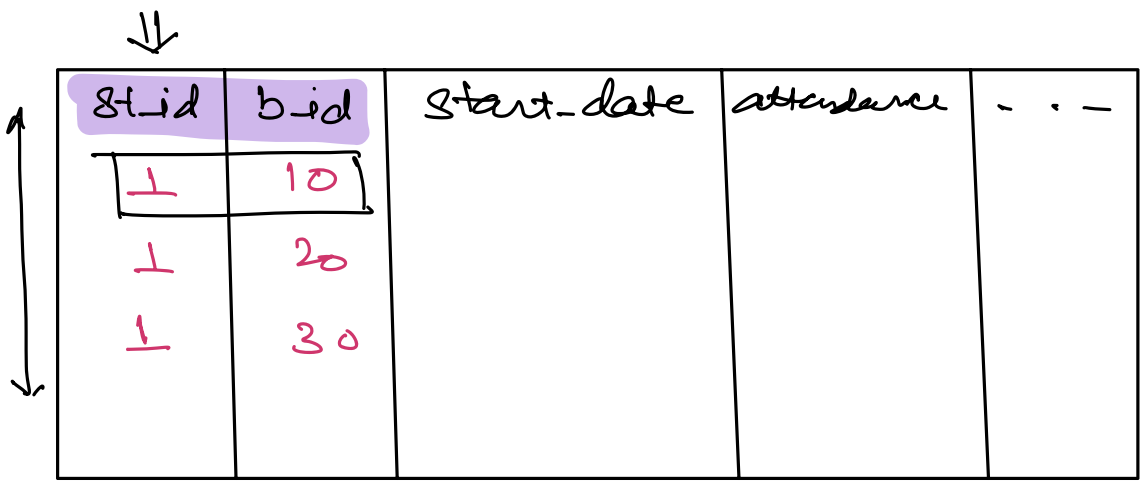
Car_type

id	type
1	SEDAN
2	~~SUV~~ XUV
3	HB



How to decide PK of Mapping table.

Students_batches.



(I) (St_id, b_id)

(II) Create a new id column

id	st_id	b_id	start_date	attendance	psp
1					
2					
3					
4					
5					
...					

⇒ Indexing.

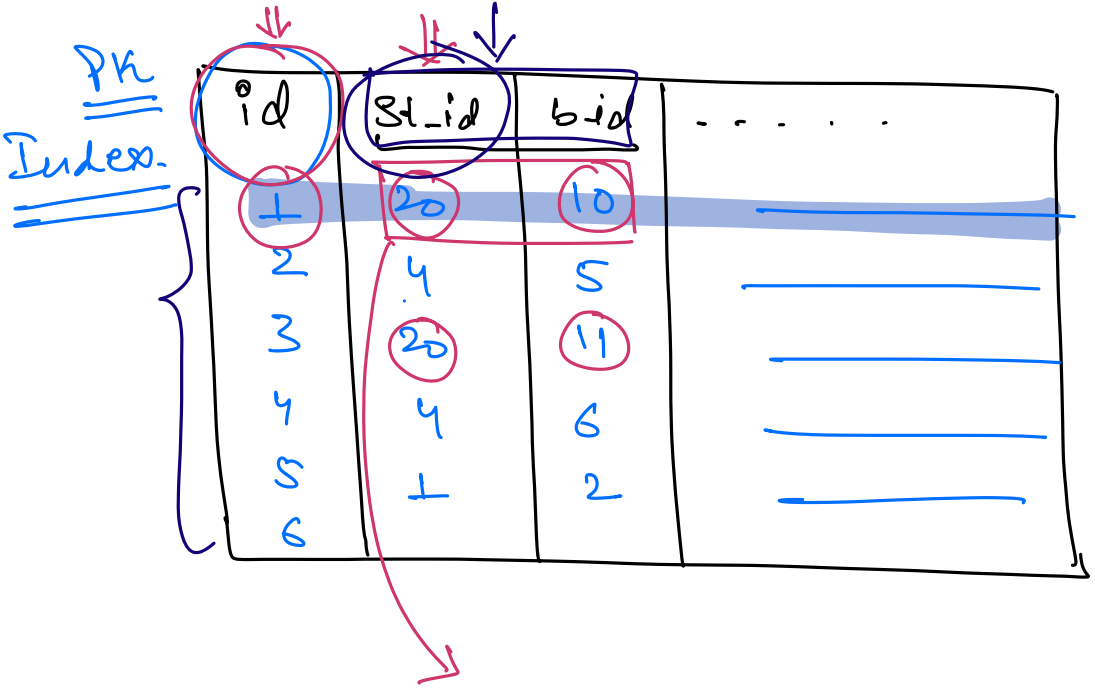
⇒ Index on 2 columns will be bigger than index on 1 column

⇒ If we are taking composite key as the primary key then it sorts the data based on it, the query to fetch data will be faster.

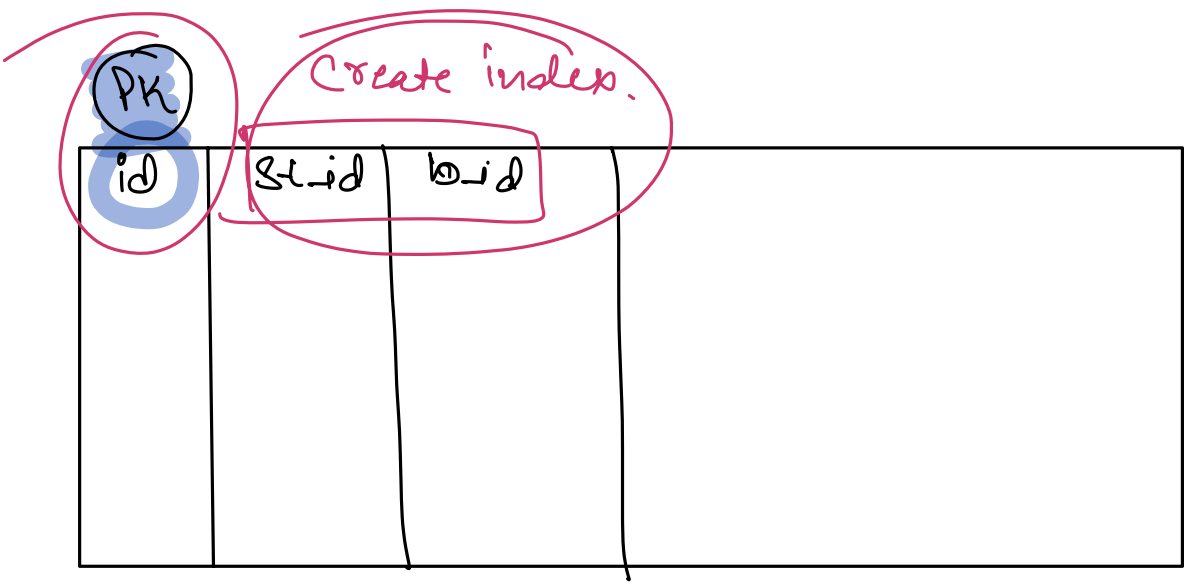
⇒ Get all the students with psp > 80 in their previous batches.

```
Select *
from ___
where st_id = 10
```

⇒ Composite key


PK
Index.
id
st_id
b_id
. . . . .
1
20
10
2
4
5
3
20
11
4
4
6
5
1
2
6
Attribute that uniquely identifies a row.

(A, B, C)


PK
id
Create index.
st_id
b_id