If you have any questions or suggestions regarding the notes, please feel free to reach out to me on WhatsApp: **Sanjay Yadav Phone: 8310206130**

Install mysql in Ubuntu

- **Step 1: Update system repositories**
  sudo apt update
- **Step 2: Install MySQL on Ubuntu 22.04**
  sudo apt-get install mysql-server
- **Step 3: Verify MySQL service status**
  systemctl is-active mysql
- **Step 4: Configure MySQL server** -> In the configuration manual, you will be first asked to set the level for the password validation policy. Input a number from the given menu according to your password
  sudo mysql_secure_installation
- **Step 5: Log in to MySQL server**-> Next, log in to the MySQL server for setting the default authentication method to “mysql_native_password” and specify a native password for the root:
  sudo mysql
  As you can see, the following query will set the root password to “Password123#@!” and the authentication method to “mysql_native_password”:
  ALTER USER 'root'@'localhost'
  IDENTIFIED WITH mysql_native_password BY 'Password123#@!';

  Reload the grant tables in the MySQL database so that the changes can be applied without restarting the “mysql” service:
  FLUSH PRIVILEGES;
  Lastly, to assure that the MySQL service is active, execute the following command:
  systemctl status mysql.service

- **mysql -u -root -p**

  The command **mysql -u -root -p** is used to access the MySQL database using the command-line interface. Here's what each part of the command does:

  mysql: This is the command to start the MySQL command-line client.

  -u: This option is used to specify the MySQL user you want to log in as. In your command, it's followed by -root, which means you want to log in as the user "root."

  -p: This option tells MySQL to prompt you for the password of the user you specified with the -u option. After you press Enter, MySQL will ask you to enter the password for the "root" user.

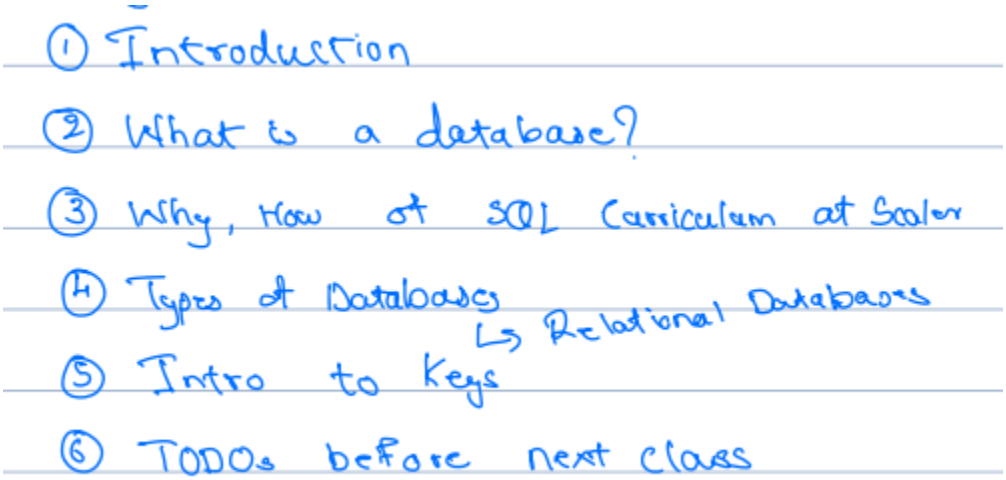
  So, in summary, the command mysql -u -root -p is used to log in to MySQL as the "root" user and prompts you to enter the password for that user.
- **sudo systemctl stop mysql -> stop mysql**
- **Come out from mysql ctrl + D**

Notes ->

1. Using aggregate functions directly in the WHERE clause within the same query is not possible because the WHERE clause is assessed before the aggregate functions. Consequently, you cannot filter or make conditional selections based on the outcomes of aggregate functions in the same query. it is happen because every time if we will use wit where it will confuse because on every row calculation it will change.
2. The HAVING clause is exclusively used in conjunction with the GROUP BY clause, and it comes after the WHERE clause when records are grouped. In this context, you can employ aggregate functions within the HAVING clause, even if aliases are used.
3. shortcut to run a query ctrl + R
4. **Describe table_name** is a command which will tales you all structure of table
5. The "SELECT" statement functions similar to a print command, enabling us to output specific values. For instance, when I write "SELECT 1," it will display the value "1," regardless of whether this particular column exists in the table. In other words, it does not validate the presence of the specified column within the table
6. To retrieve rows for pagination, it's preferable to use OFFSET and LIMIT rather than specifying a range like id > 0 to <= 50. This is because when entries are deleted, the direct range approach may result in returning fewer records. For instance, if you deleted 5 records, it would return 45 records. However, when you use LIMIT and OFFSET, it

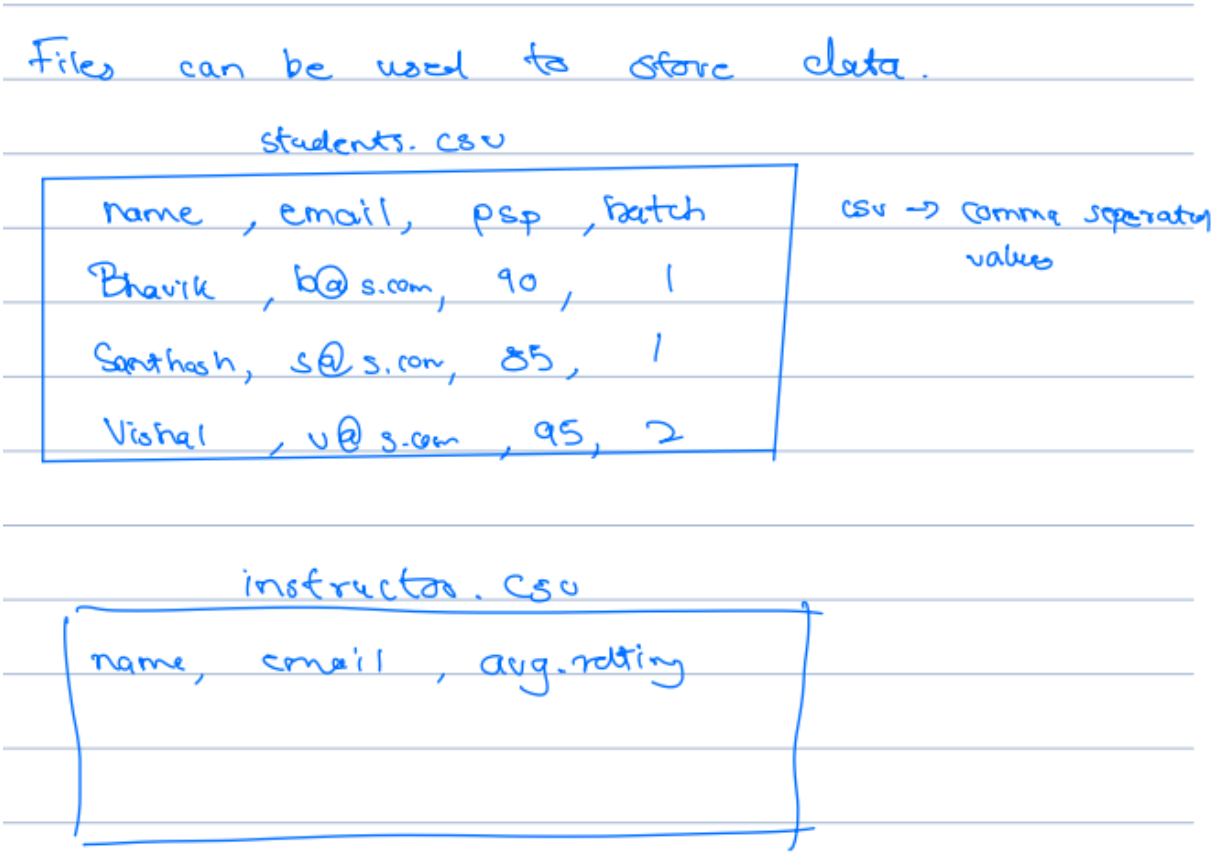
ensures you consistently receive the desired number of records, such as 50, regardless of deletions in the dataset.

SQL Intro - 21-sept-23



Data - Collection of information. Base - Store something

Database - place where information/data to store information about someting.



Problem in filesysystem

1 - Querying Data is Difficult & Inefficient - This problem suggests that it is challenging to retrieve or search for specific data within the file system. Efficient data retrieval is crucial for quick and accurate access to information.

2 - Data Integrity - Data integrity refers to the accuracy and consistency of data stored in the file system. Ensuring that data remains intact and unaltered is essential to maintain the reliability of the system.

3 - Security - Read Access Control - This problem pertains to controlling who has read access to confidential files or information within the file system. Proper security measures are necessary to protect sensitive data from unauthorized access.

4 - Concurrency - Concurrency issues involve multiple users or processes attempting to access and modify the same data simultaneously. Managing concurrent access to files is important to prevent data corruption and ensure consistent behavior.

Actal DBMS come & solve all there problems.

DBMS - Database Management system - A S/W (software) system which helps us manage databases.

- A file system is a primitive version A DBMS

How did actual DBMS soved the problems

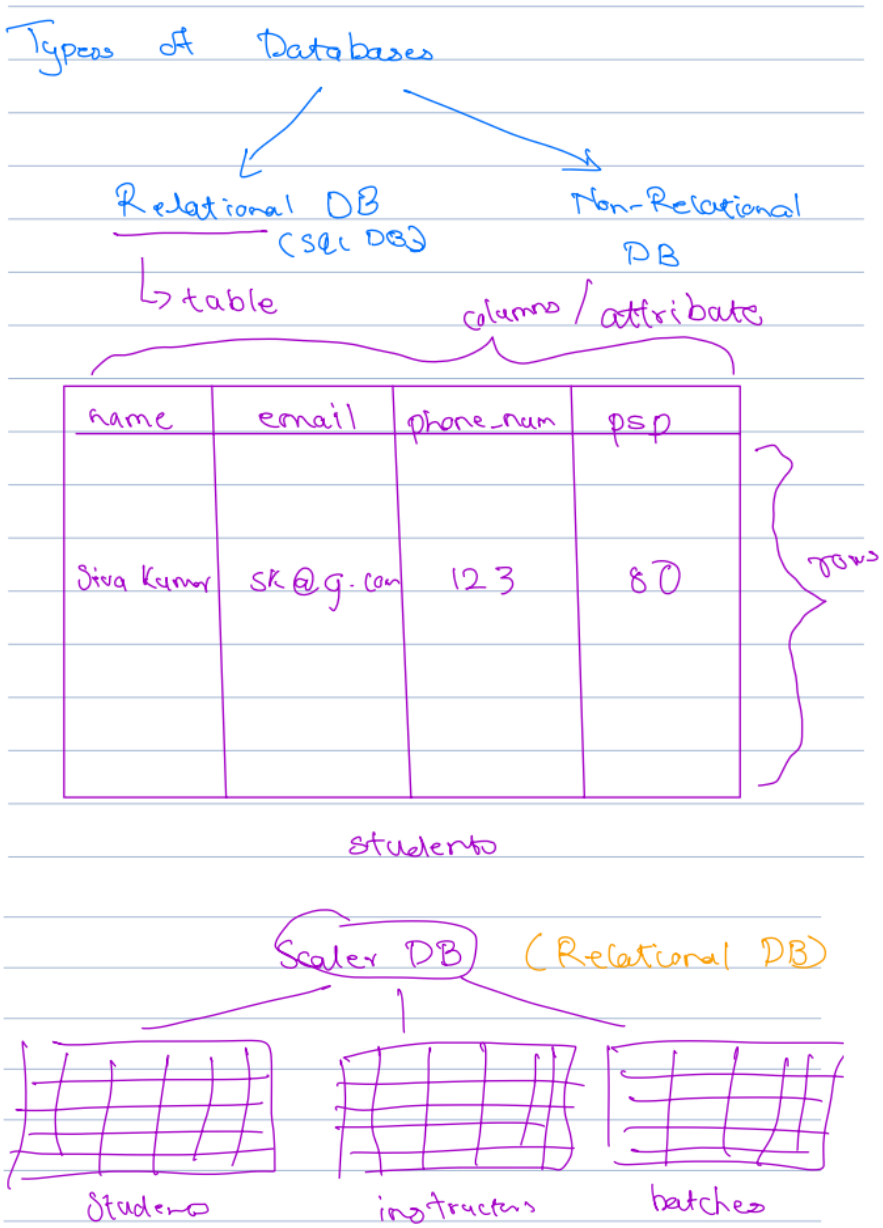
1 - SQL (Structured Query Language): SQL is a domain-specific language used for managing and manipulating relational databases. It provides a standardized way to interact with databases, including operations like querying, updating, and managing data.

2 - Constraints: In SQL, constraints are rules and limitations applied to columns in database tables to maintain data integrity. Common constraints include primary keys, foreign keys, unique constraints, and check constraints (**Check constraints are used to enforce data integrity rules by ensuring that data values in a column or columns meet certain criteria. -> CONSTRAINT CHK_Salary CHECK (Salary >= 0)**). They ensure that data stored in the database meets specific criteria and maintains consistency.

3 - Efficient Way to Store & Retrieve Information: SQL databases are designed to efficiently store and retrieve data. They use indexing, query optimization techniques, and various data structures to speed up data access operations, making them well-suited for managing large datasets.

4 - Introduced Security Features: SQL databases typically offer security features to protect data. This includes user authentication, authorization, and role-based access control (RBAC). SQL also provides features like encryption for data-at-rest and data-in-transit to enhance data security.

Types of Database



Relational - A relation nothing but a Table

- SQL is used to intract with R-DB - MySQL, Pastgre SQL, Oracle, SQL server etc.

- Non-Relational (No SQL DB) | DB (HLD)

They do not store data in table. Instead they store data in key-value pairs. graphs, documents etc.

Non relational Types of Database :

1 - Columns (**Cassandra, Big Table**)

2 - Graph (**Neo4j**)

3 - Key-vaue (**Dynamo DB**)

Relational DB Properties :

#1] We store data in R-DB as a collection of tables / relations.

These tables can have relationships amongst themselves.

#2] Every row is unique

name	psp	email
Bhavik	80	b1@g.com
Ayush	95	a@g.com
Bhavik	~~80~~ 100	b2@g.com
:		

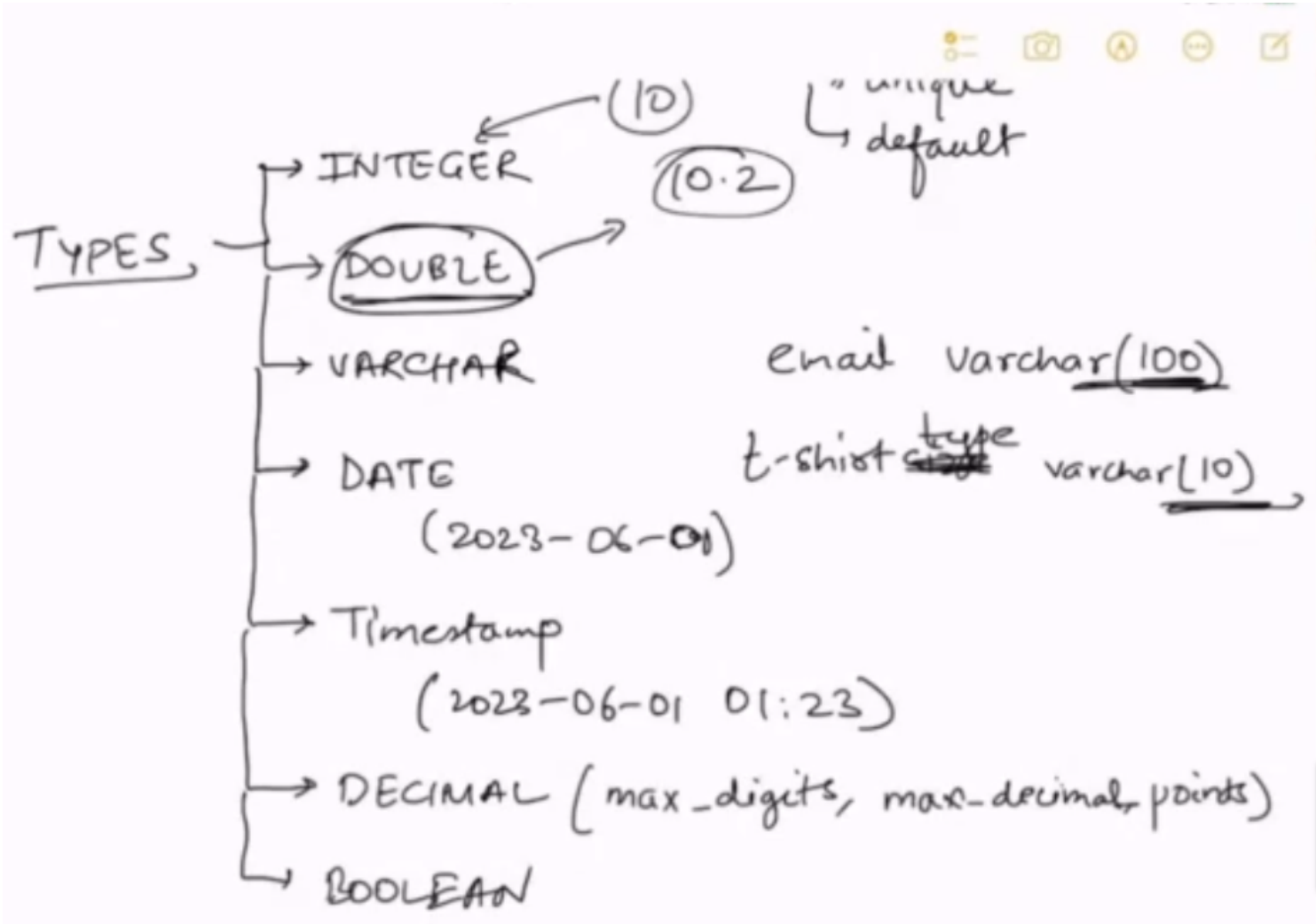
Students

CREATE TABLE

- TABLE NAME
- COLUMNS
  - TYPE
  - NAME (unique for this table)
  - Properties.
    - NOT NULL
    - unique
    - default

Column Properties
- AUTO_INCREMENT
- NOT NULL
- UNIQUE
- DEFAULT

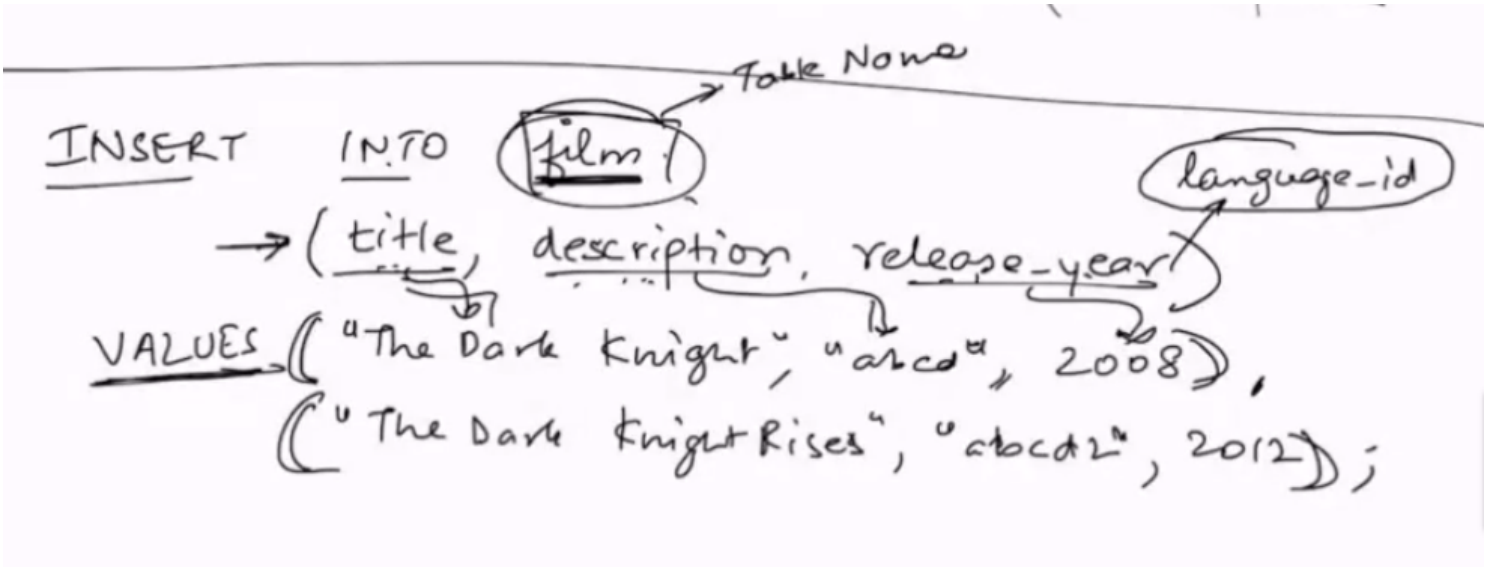
TABLE PROPERTY
- PRIMARY KEY (col-na)
- FOREIGN KEY.



Select Database Structure Content Relati

```sql
CREATE TABLE students (
    id INT AUTO_INCREMENT,
    firstName VARCHAR(50) NOT NULL,
    lastName VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    dateOfBirth DATE NOT NULL,
    enrollmentDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    psp DECIMAL(3, 2) CHECK (psp BETWEEN 0.00 AND 100.00),
    batchId INT,
    isActive BOOLEAN DEFAULT TRUE,
    PRIMARY KEY (id)
);
```

```
Step 1: Look at tables.
Step 2: Apply the filters (WHERE clause)
Step 3: Print (in printing, we can have aggregate functions (min, max, sum, avg, DISTINCT))
```



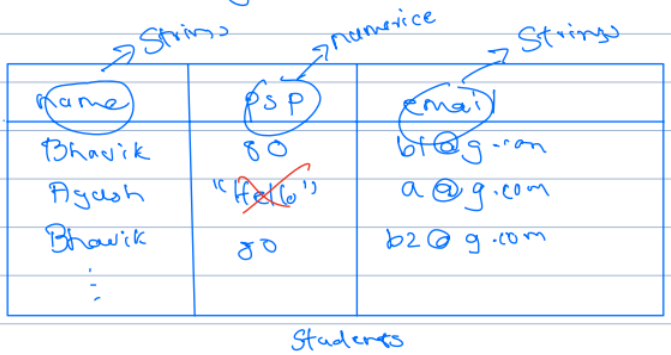
Bulk insert from 1 table to another -> In below query it is not required to write a values we can directly get value from other table using select with condition and put in another table.

```
INSERT INTO film_copy (title, description, release_year, language_id, rental_duration, rental_rate, length,
replacement_cost, rating, special_features)

SELECT title, description, release_year, language_id, rental_duration, rental_rate, length, replacement_cost,
rating, special_features
FROM film WHERE release_year = 2006;
```

INSERT INTO target_table (column1, column2, column3, ...)
SELECT source_column1, source_column2, source_column3, ...
FROM source_table
WHERE your_condition;

INSERT INTO target_table (column1, column2, column3, ...)
SELECT * // add all columns sequentially in the top including id.
FROM source_table
WHERE your_condition;

#3] All the values present in a column should hold the same data type



Students

#4] Values in a cell should be atomic.

Atom → Indivisible

name	psp	phone numbers
Bhavik	80	[123, 456]
Ayush	95	789
Bhavik	80 (60)	1234
⋮		

Students

#5] Column sequences are not guaranteed

name, email, phone_number, psp

↑ User given column list

name, psp, email, phone_numb

↑ R-DBMS storage

#6] Row sequence is not guaranteed unless you use dedicated sequencing

name	psp	email
Bhavik	80	b1@g.com
Ayush	95	a@g.com
Bhavik	80	b2@g.com
⋮		

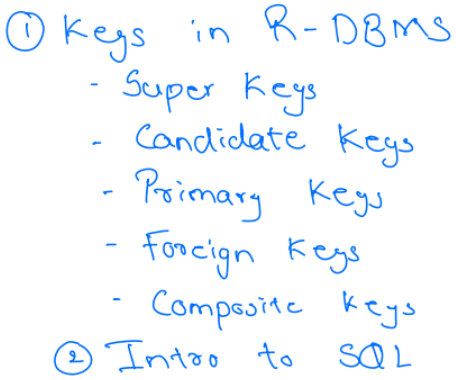
Students

#7] Every column name must be unique

name	psp	psp
Bhavik	80	
Ayush	95	
Bhavik	80	
⋮		

Students

Keys - 23-sept-23



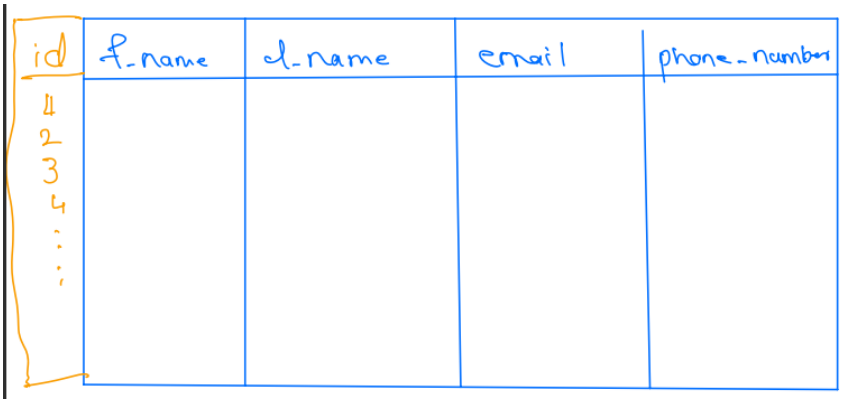
Kegs in R DBMS

Lets say we have a table of students data if I ask you to update psp for student named Bhavik to 100 what will you do if your data looks like this?

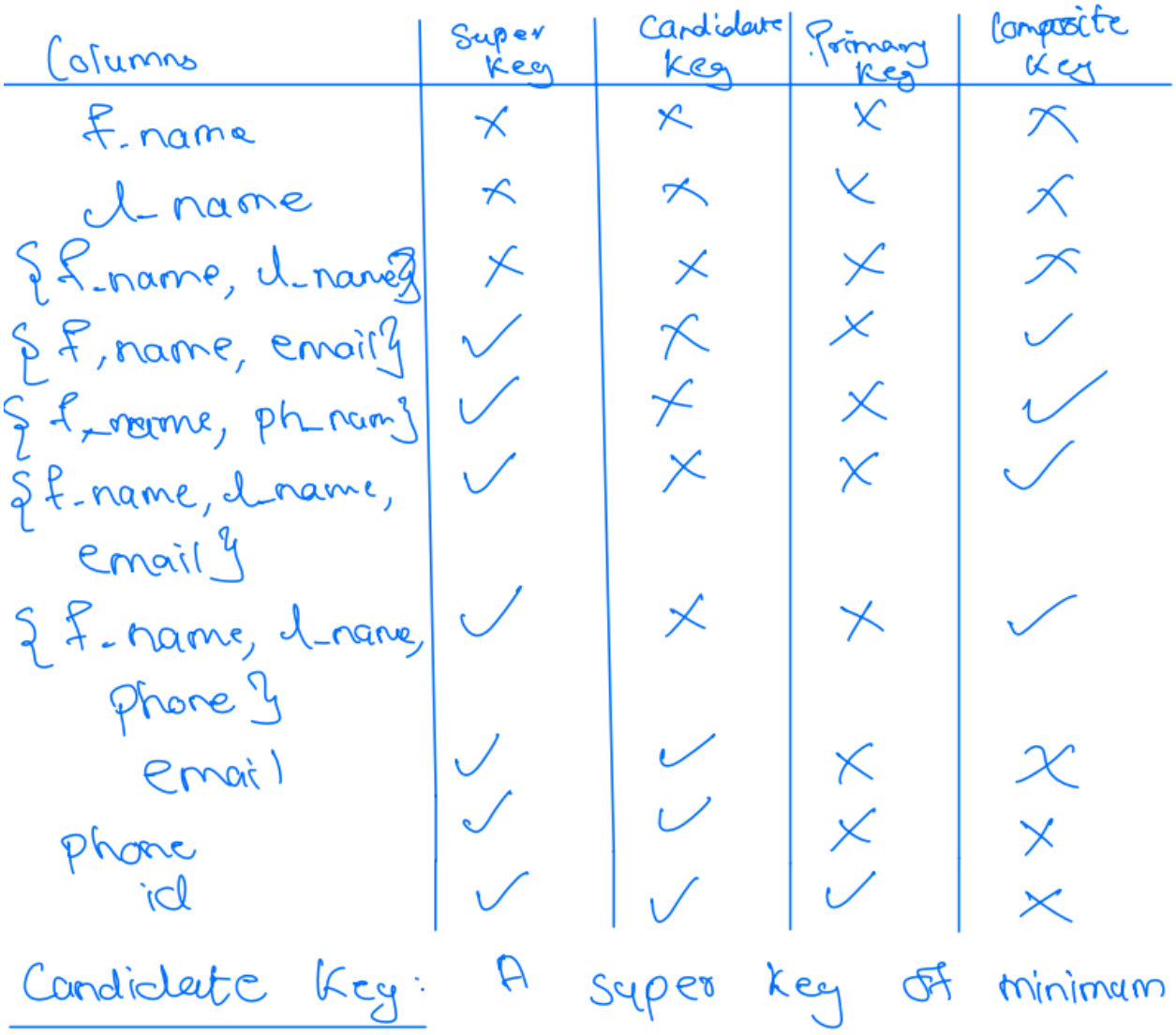
name	psp	batch
Bhavik	65	1
Shubham	85	2
Punit	90	1
Bhavik	80	1
Pooja	75	3

- Keys help us identify 1 row in a table uniquely

- Lets look at different types of keys for this table



- Super key : Any set of columns which can uniquely identify a row in a table is a super key.



- **Primary key** : one of the candidate keys chosen by database architect become primary key

Note => I will not choose user attributes which can change like email or phone as my primary key.

We can use autogenrated ids which is not possible to modified from user. We will introduce a new column called as **“id”** or **“student-id”**.

Composite Key : A composite key, also known as a composite primary key or compound key, is a key in a relational database that consists of two or more columns (attributes) used together to uniquely identify a row (record) within a table

Foreign Key:

id	name	psp	batch_id
1	A	80	1
2	B	60	1
3	C	70	2
4	D	100	1

Student (child)

id	batch_name	batch_schedule
1	~~Feb23 Beg~~	
2	Feb23 Adv	
3	Feb23 Int	
4		

batches (parent)

Foreign key : A foreign key (FK) is a relational database constraint that enforces referential integrity between two tables in a database. It establishes a link or relationship between two tables by ensuring that the values in one table's foreign key columns match the values in another table's primary key or a unique key column. Foreign keys help maintain the consistency and integrity of data in a relational database by enforcing rules about relationships between tables.

Constraints for Deletion

- **Cascade:** When we delete a batch, cascade mode deletes all the student records in the student table linked to that batch.
- **Set Null (Empty - No value):** If we delete a batch, this mode replaces the batch ID in student records with nothing (null) instead of deleting them.
- **No Action (Default Mode):** In this mode, nothing happens to the student table when a batch is deleted.

- **Set Default:** Instead of making it empty, set the batch ID to a default value if we delete the batch.

Queries

- Lets see how we can create a database in MySQL.
  Syntax : CREATE DATABASE DB_NAME

- Lets see how to create a table in MySQL
  Syntax : CREATE TABLE table_name
  call1_name data_type constraints
  call2_name data_type constraints

Constraints : These are rules and restrictions applied to a column in a table. These are crucial to ensure data integrity.

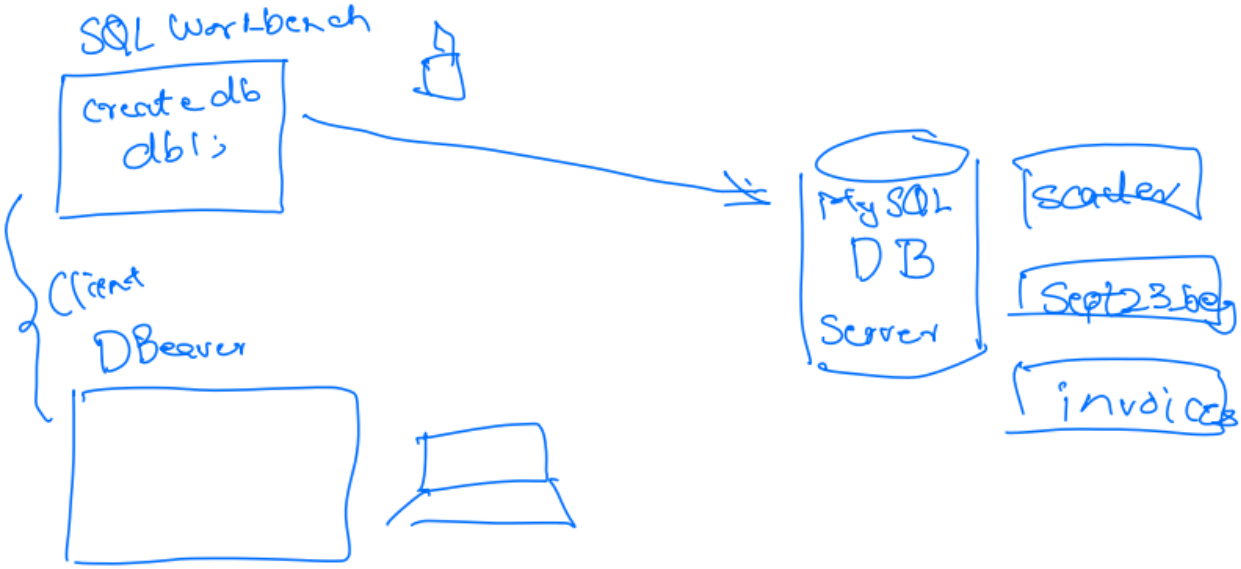
1 - Not Null Constraint : When applied ensures that null cannot be stored in that column.
2 - Unique Constraint : When applied to a column ensures that only unique values can be stored in that column.
Note: It allows only 1 NULL.

3 - Primary key constraint : Combination of unique + not null constraint.

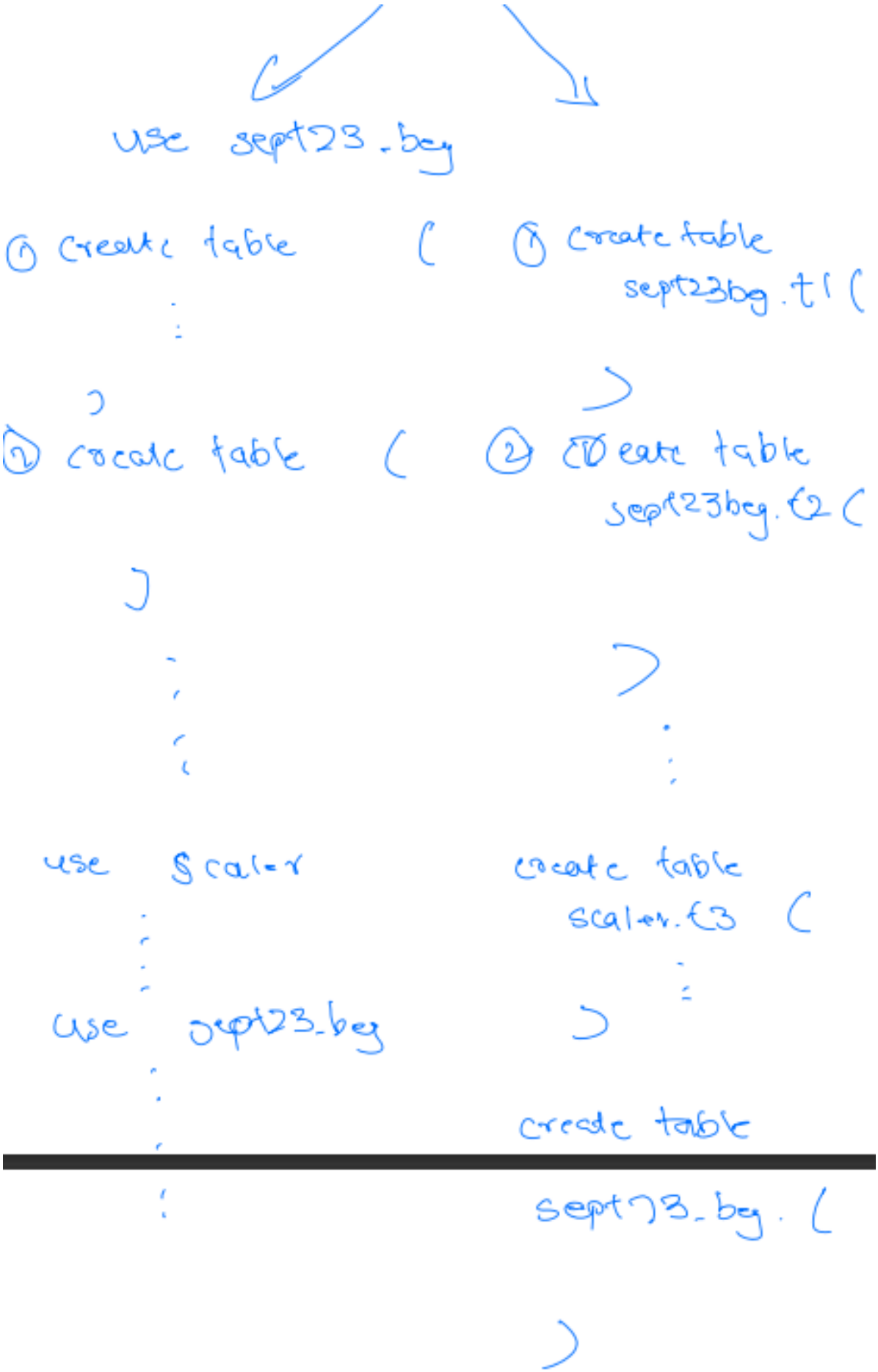
4 - Foreign Key Constraint : Links two tables together ensures that value in column must match a value in table's primary key. (FOREIGN KEY (batch_id) REFERENCES batches(batch_id))

5 - Default Constraint : Used to give a default value to a column, when no value is specified while insertion of new data, the default value is used.

6 - Check Constraint : Enforces that the value in a column meets a specific condition
Example : age column should only have values greater then 8.



w/o specifying table name:
No database selected.


use sept23.beg
① create table (
:
)
② create table (
:
)
use $calar
:
use sept23.beg
:
:
create table
① create table
sept23beg.t1 (
)
② create table
sept23beg.t2 (
)
create table
scalar.t3 (
:
)
create table
sept23_beg. (
)

```
CREATE DATABASE sept23_beg;

-- Option #1
USE sept23_beg;

CREATE TABLE batches (
        batch_id INT PRIMARY KEY,
        batch_name VARCHAR(20) UNIQUE NOT NULL
)

-- Option #2
USE scaler;

CREATE TABLE sept23_beg.batches1 (
        batch_id INT PRIMARY KEY,
        batch_name VARCHAR(20) UNIQUE NOT NULL
)

CREATE TABLE students(
        student_id INT PRIMARY KEY,
        f_name VARCHAR(20) NOT NULL,
        l_name VARCHAR(20),
        batch_id INT,
        FOREIGN KEY (batch_id)
        REFERENCES batches(batch_id)
        ON DELETE CASCADE
        ON UPDATE SET NULL
)
```

SQL:3 - CRUD_1 - 26 - Sep 2023

- **What is CRUD**
- **Sakila Database**
- **Create**
- **Read**
- **Selecting Distinct Values**
- **Select statement to print a constant value**
- **Operations on Columns**
- **Inserting Data from Another Table**
- **WHERE clause (AND, OR, NOT)**
- **IN Operator**

What is CRUD -> CRUD is an acronym that stands for Create, Read, Update, and Delete. It is a set of basic operations that are commonly used in the context of database managemen

Sakila Database -> It is a MySQL-provided demo database containing sample data, primarily designed to represent a video library
open click below link find Sample database and download Sakila
: https://dev.mysql.com/doc/index-other.html

Extract a zip and open It has 3 files **sakila-data.sql, sakila-schema.sql and sakila.mwb** open **sakila-data** in vs code editor and copy all content and pest it in dbever by clicking new query and click on run all query icon. same do for **sakila-data to import demo.**

Create —----------

Syntax for insert

INSERT INTO table_name(col_1, col_2,...) **VALUES**(val_1, val_2,...);

for multiple queries–

INSERT INTO table_name(col_1, col_2,...) **VALUES**(val_1, val_2,...), (val_1, val_2,...)

-- Option #1

INSERT INTO sakila.film
(film_id, title, description, release_year, language_id, original_language_id, rental_duration, rental_rate, `length`, replacement_cost , rating, special_features, last_update)

VALUES (1001, "Jawaan", "Shahrukh Khan + South Indian Cinema", 2023, 1, NULL, 3, 10.99, 180, 29.99, 'PG-13', "Behind the scenes", '2023-09-25')

-- Option #2

INSERT INTO sakila.film
VALUES (1003, "Jawaan - 2", "Shahrukh Khan + South Indian Cinema", 2023, 1, NULL, 3, 10.99, 180, 29.99, "PG-13", "Behind the scenes", '2023-09-25')

-- **Option #2** is not readable and can lead to human errors because if we will change the order of values then it will insert wrong data or gerate error.

Note -> We also can add **default** instead serial because it is **auto_increment,** If we want we can skip column and value both.

Read —----------
Syntax:
SELECT {columns} FROM table_name;

SELECT * FROM sakila.film;

-- Print film_id, name, description, rental_rate
SELECT film_id, title, description, rental_rate **FROM** sakila.film;

-- Rename the title column as movie_name
SELECT film_id, title as 'movie_name', description, rental_rate **FROM** sakila.film;

-- Selecting distince values
SELECT DISTINCT rating **FROM** sakila.film;

-- Print distinct rating + release_year
SELECT DISTINCT rating, release_year **FROM** sakila.film;

Valid queries —---
select 'Hello world'; **select 'Hi'**; // this is also valid queries it will print same string as column and data

Here it will print tile from table and ‘HI’ as string because it is not present in column.
select title, 'Hi' from sakila.film;
select title, 10 from sakila.film;

Operations on Columns—----------

select title, `length`/60 as 'duration_in_hrs' from sakila.film ;
select title, ROUND(`length`/60) as 'duration_in_hrs' from sakila.film ;

-- Inserting rows from 1 table to another

CREATE TABLE film_copy(film_id INT PRIMARY KEY, title VARCHAR(128), release_year INT, rating varchar(10))

INSERT INTO film_copy (film_id,title, release_year, rating)
SELECT film_id, title, release_year, rating from sakila.film;

select * from sakila.film_copy;

Delete table
DROP TABLE sakila.film_copy;

-- Select with filtering
select * from sakila.film where rating = 'PG-13';
select * from sakila.film where `length` >= 180;
select * from sakila.film where rental_rate > 5;

-- AND, OR, NOT
select * from sakila.film where rental_rate < 8 and rating = 'PG-13';
select * from sakila.film where rental_rate < 8 or rating = 'PG-13';
select * from sakila.film where NOT rating = 'PG-13';
select * from sakila.film where rating != 'PG-13';

-- != and <> are the same
select * from sakila.film where rating <> 'PG-13';
select * from sakila.film where rating = 'PG-13' or (release_year < 2023 AND rental_rate < 5);

-- print movies which are G or NC-17 or PG-13 rated
select * from sakila.film where rating = 'G' or rating = 'NC-17' or rating = 'PG-13';

-- IN is used when we are doing multiple ors on the same column
select * from sakila.film where rating IN ('G', 'NC-17', 'PG-13');

SQL:CRUD_2 - 30 - Sep 2023

Print all film names which were released after and including 2005 and before and including 2008
Select title from sakila.film where release_year >= 2005 and release_year <=2008 ;

BETWEEN Operator
-- syntactical sugar it will not anything in optimisation just little less code
-- between will include upperlimit and lower limit in the output
Select title from sakila.film where release_year between 2005 and 2008;

Like operator

- Find all the movie where the name contains the word "love" see notes
SELECT * from sakila.film where title like '%Love%'

-- Title ends with love
SELECT * from sakila.film where title like '%Love'

-- Title start with love
SELECT * from sakila.film where title like 'Love%'

All the moves where 2nd char is a 'A' we will use _ to match how many charecter we want to skip from start or end with like query.

-- title has 2nd char as A
SELECT * from sakila.film where title like '_A%'

-- title has 3 char with first word
SELECT * from sakila.film where title like '___ %'

-- IS NULL operator
-- select all movie where description has null
select * from sakila.film where description = null;

-- **we cannot use =, <>, >=, <= to compare with NULLS**
Note -> Please note that when a column contains no value, using 'column = null' will not yield results because the database does not match null to no value. In such cases, it is necessary to use the '**IS NULL**' keyword.

select * from sakila.film where description IS Null;

-- where description exists
select * from sakila.film where description IS NOT Null;

-- ORDER BY→ By default ORDER BY will use ascending order for sorting

-- In the case of multiple column sorting first will sort all data but second colun only sort that data which is duplicate after first sorting and it will sort onlt duplicate records not all data and if there is no duplicate then second column sorting will not effect on data sorting.

SELECT * from sakila.film ORDER BY title;
SELECT * from sakila.film ORDER BY title ASC;
SELECT * from sakila.film ORDER BY title DESC;
SELECT * from sakila.film ORDER BY 'length' DESC, title ASC;

Steps

1. which table has ny information
2. write the filters (WHERE clause)
3. Print (SELECT columns)
   - Maybe sort? (ORDER BY)
4. Paginate (LIMIT, OFFSET)

-- I can order by a column that is not going to be a part of my output
SELECT title from sakila.film ORDER BY 'length' DESC;

-- select movie where name start with love and order them by length
SELECT title, 'length' from sakila.film Where title like 'love%' ORDER BY 'length' ASC;

-- LIMIT clause -> Instead of pulling all the rows from a table, we can use limit clause to pull only specified amount of rows
SELECT * from sakila.film LIMIT 10;

-- Skip first 10 rows and then fetch next 10 rows. It is set where do you want to start from
SELECT * from sakila.film LIMIT 10 OFFSET 20;

-- Select movies where name contains with love and order them by length desc and limit 2 rows and skip 2 rows.

SELECT * from sakila.film where title like %love% order by 'length' desc limit 2 offset 2;

Update -> Where clause is optional is update query, but ideally everytime we sholud use where clause with update, else all the rows will be impacted but it will give warning to you.

```
/**
UPDATE table_name
SET col1 = val1, col2 = val2,
WHERE conditions
**/


UPDATE sakila.film SET release_year = 2007, rental_duration = 100 Where film_id = 1;
```

-- Delete -> Delete will remove rows from the table, if where is mentioned then only those rows will be remvoed which staisfy the where clasues else all the rows will be deleted
-- **Delete doesnt resets the auto increment counter.**

```
/*
DELETE FROM table_name
WHERE conditions
*/

DELETE from sakila.film_copy WHERE film_id = 2;
```

-- TRUNCATE -> Truncate will first delete the entire table structure (i.e. all the rows and cols) and recreate the table structure. Auto increment will reset and start from 1. This operation cannot be undoed

```
/*
TRUNCATE table_name
*/
```

-- DROP

```
/*
DROP TABLE table_name;
It will delete the table structure (i.e rows and cols) and thats it.
This operation cannot be undoed
*/
```

SQL:JOIN - 3 - Oct 2023

- **Joins**
- **Self Join**
- **compound joins**

Join -> Join is used to get data from two table with the help of foreign key and primary we will get it.

```
select film.title, `language`.name
from film
join `language`
on film.language_id = `language`.language_id;
```

- alias -> We will use alias to rename our table name

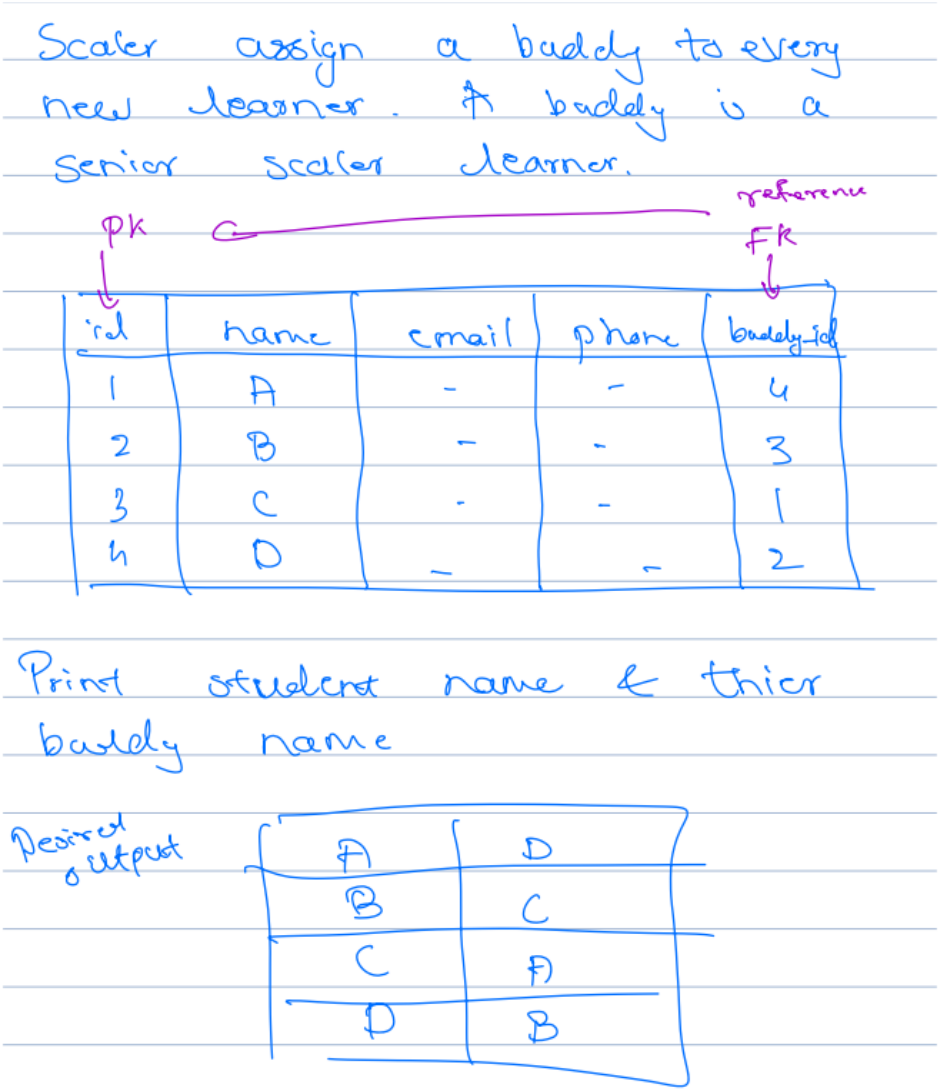
```
select f.title, l.name
from film f
join `language` l
on f.language_id = l.language_id ;
```

-- Print city name and country name

```
select city.city, country .country
from city
join country
on city.country_id = country.country_id;
```

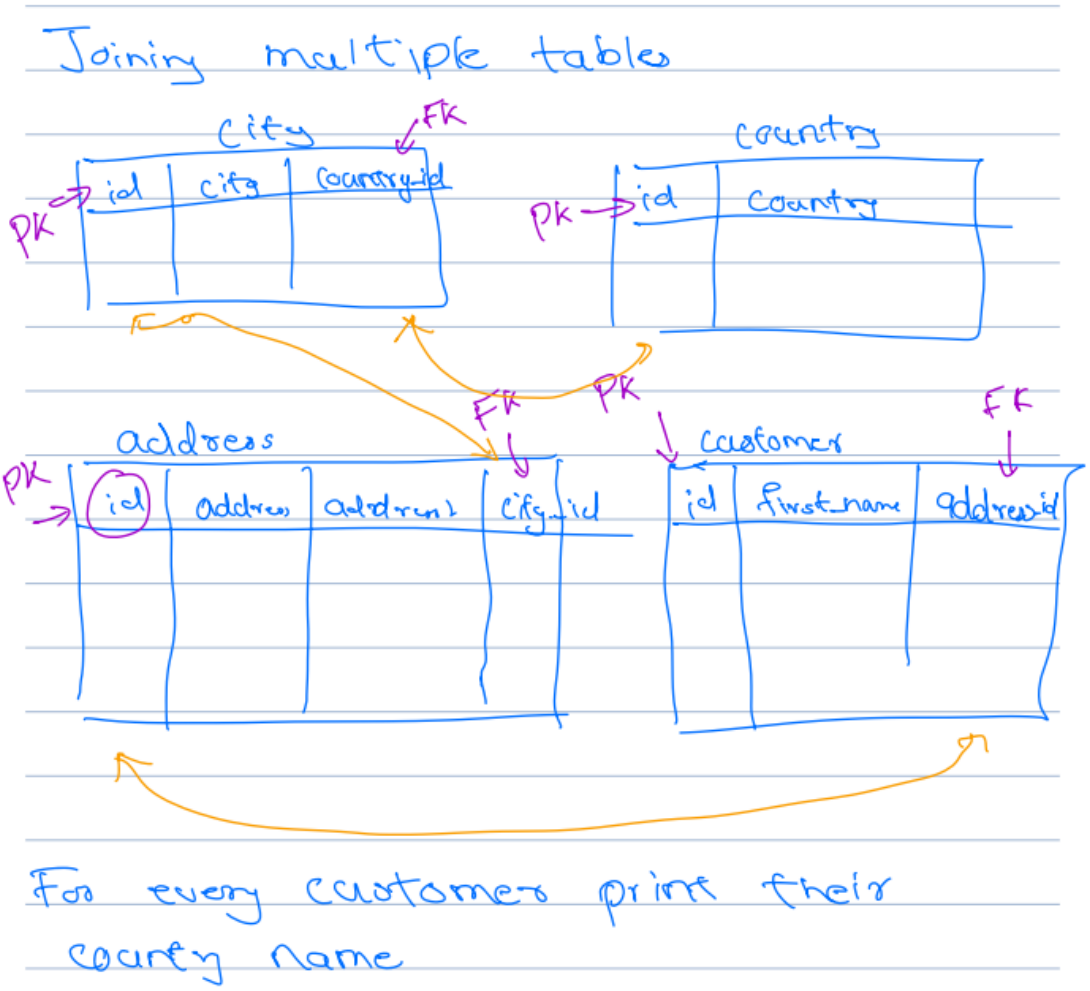
-- with alias -> The use of aliases in a SELECT statement is made possible by an internal algorithm, as aliases are defined after they are used. In MySQL, the process involves first combining all the data, followed by applying the WHERE clause to filter the selected data.

```
select c.city, co .country
from city c
join country co
on c.country_id = co.country_id;

-- self join
```




```
select * from sql_hr.employees ;

-- print employee name and manager name

select e.first_name, e.last_name, m.first_name, m.last_name
from sql_hr.employees e
join sql_hr.employees m
on e.reports_to = m.employee_id;
```

-- joins with multiple tables




```
select c.first_name, c.last_name , co.country
from customer c
join address a
on c.address_id = a.address_id
join city ci
on a.city_id = ci.city_id
join country co
on ci.country_id = co.country_id ;
```

-- compound joins

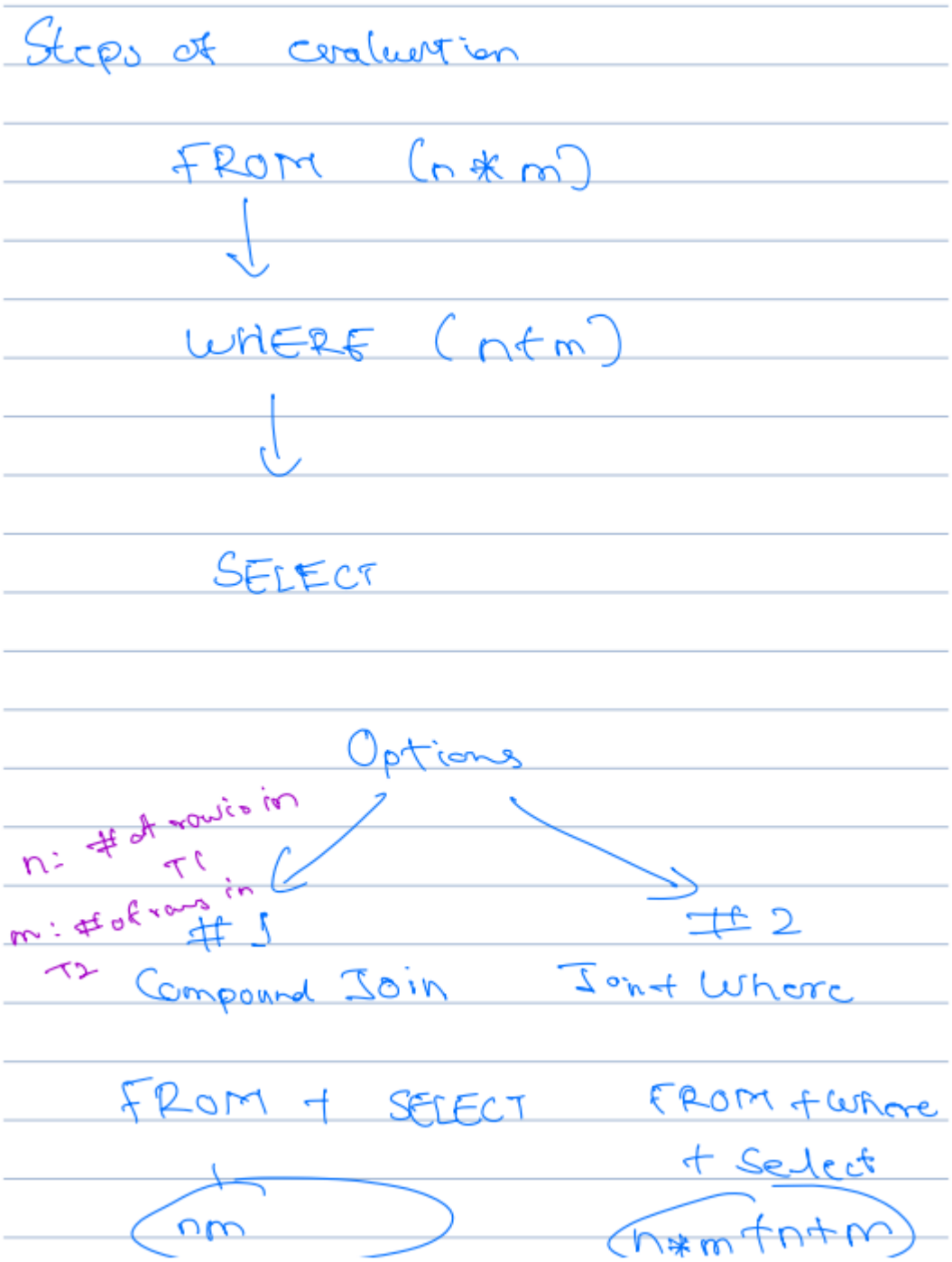
Are joins where joining condition has multiple condition.

Print country name & city name where country name starts with 'I'

```
select ci.city, co.country
from city ci
 join country co
 on ci. country_id= co.id
        and co.country LIKE 'I%'
```

```
select ci.city, co.country
from city ci
join country co
on ci.country_id = co.country_id
AND co.country LIKE 'I%';
```

-- same query using where -> If we will use where clause instead compound then our query will take more time. Good idea to use compound instead where

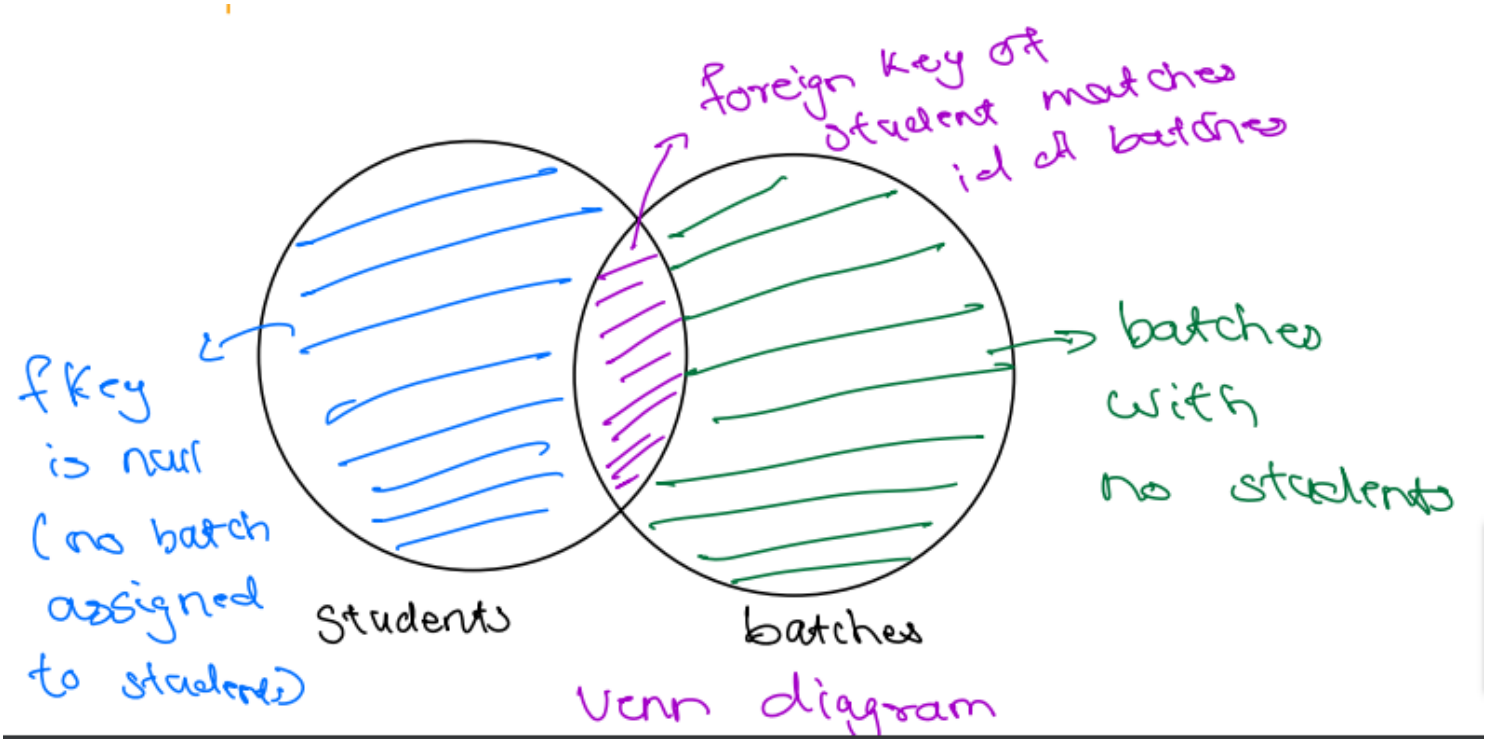



```
select ci.city, co.country
from city ci
join country co
on ci.country_id = co.country_id
where co.country LIKE 'I%';
```

SQL:JOIN 3 - 5th Oct 2023

- Types of Joins
- USING
- NATURAL JOIN
- CROS JOIN
- IMPLICIT JOIN
- UNION

Types of Joins



- **Inner Join (Join) ->** An Inner Join, sometimes just called a Join, gives you only the rows that match between two sets of data. Imagine you have a list of students and a list of batches, and you want to find where they overlap or match. That's what an Inner Join does; it finds the common ground or intersection between them

  select s.fname, s.lname, b.batch_id
  from student s
  join batch b
  on s.batch_id = b.id

```
select s.fname, s.lname, b.batch_id
from student s
inner join batch b
on s.batch_id = b.id
```

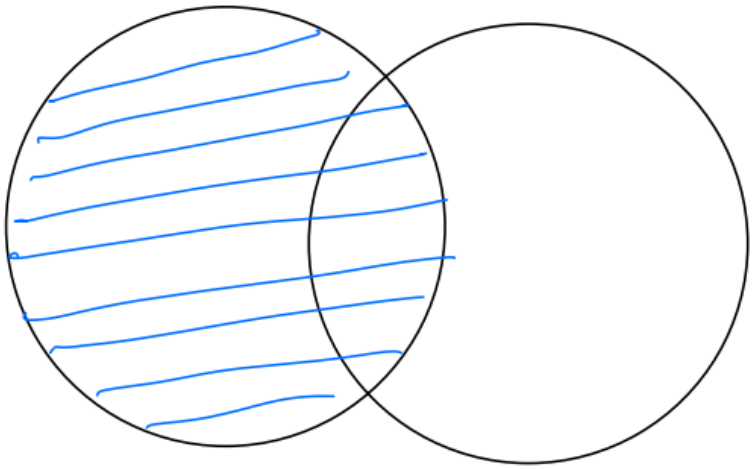
- **Left Join ->** Print name of students and their batch name if a student is not assigned any batch then that student should also be part of the output.

```
select s.fname, s.lname, b.batch_id
from student s
left join batch b
on s.batch_id = b.id
```

Note : All the rows from left table will be part of the output

O/P :

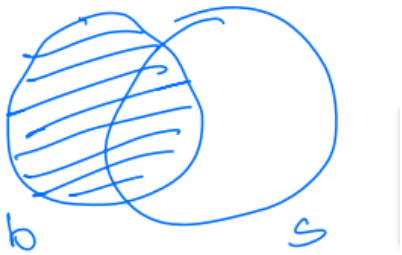
John	Doe	Batch A
Jane	Doe	Batch A
Jim	Brown	NULL
Jenny	Smith	NULL
Jack	Johnson	Batch B



Left Join

- **Right Join ->** Print name of students and their batch name also if a batch has no students assigned, include that as well in the output.

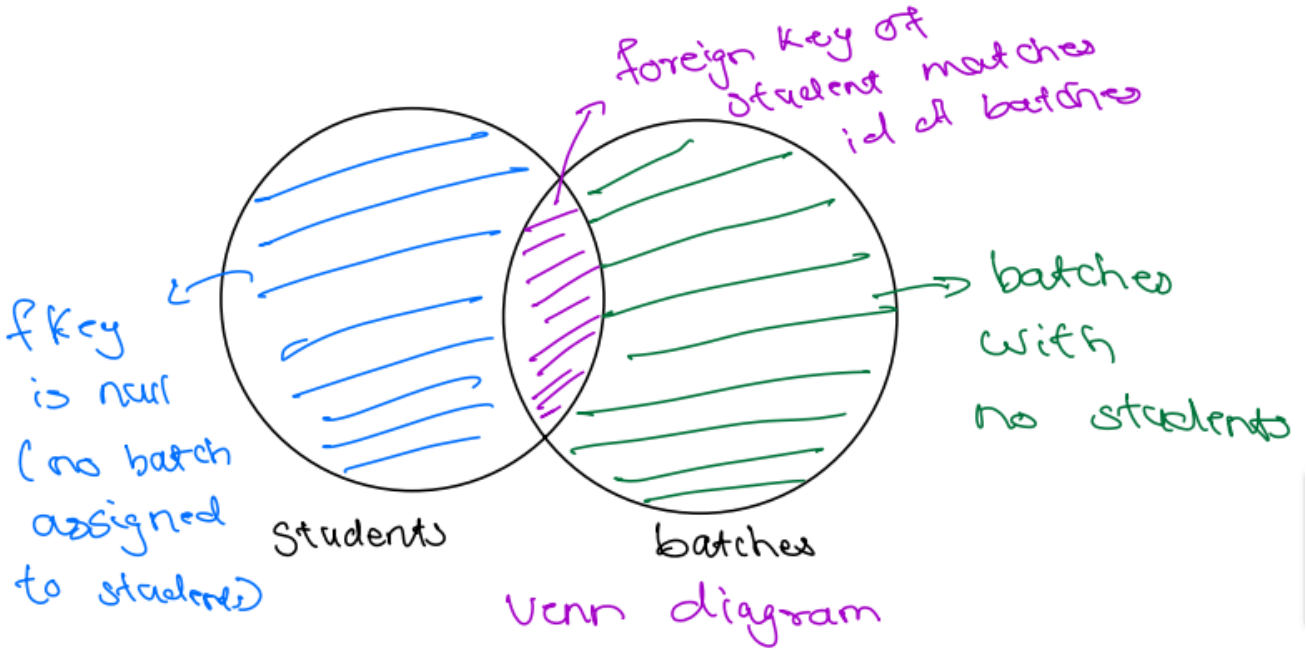
select s.fname, s.lname, b.batch_id **from** student s **right join** batch b **on** s.batch_id = b.id	

select s.fname, s.lname, b.batch_id **from** batch s **left join** student b **on** s.batch_id = b.id	

Note : All the rows from right table will be part of the output. above both query same mirror to each othor.

John	Doe	Batch A
Jane	Doe	Batch A
Jack	Johnson	Batch B
NULL	NULL	Batch C

- **Full Outer Join ->** A Full Outer Join is used to display both student names and batch names. It includes students who are not part of any batch and also lists batches with no students in them. So, if a student doesn't belong to any batch, we'll still print their name, and if a batch has no students, we'll include it in the result.

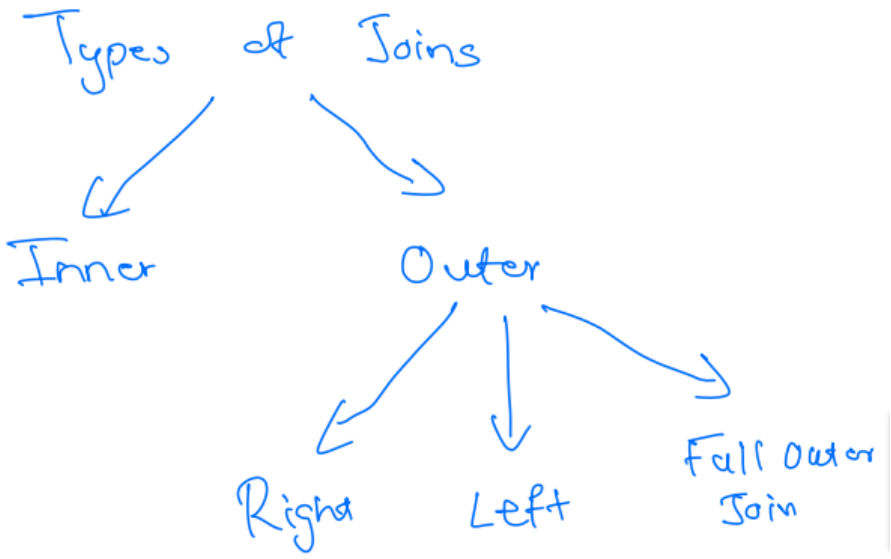



```
select s.fname, s.lname, b.batch_id
from student s
full outer join batch b
on s.batch_id = b.id
```

O/P:

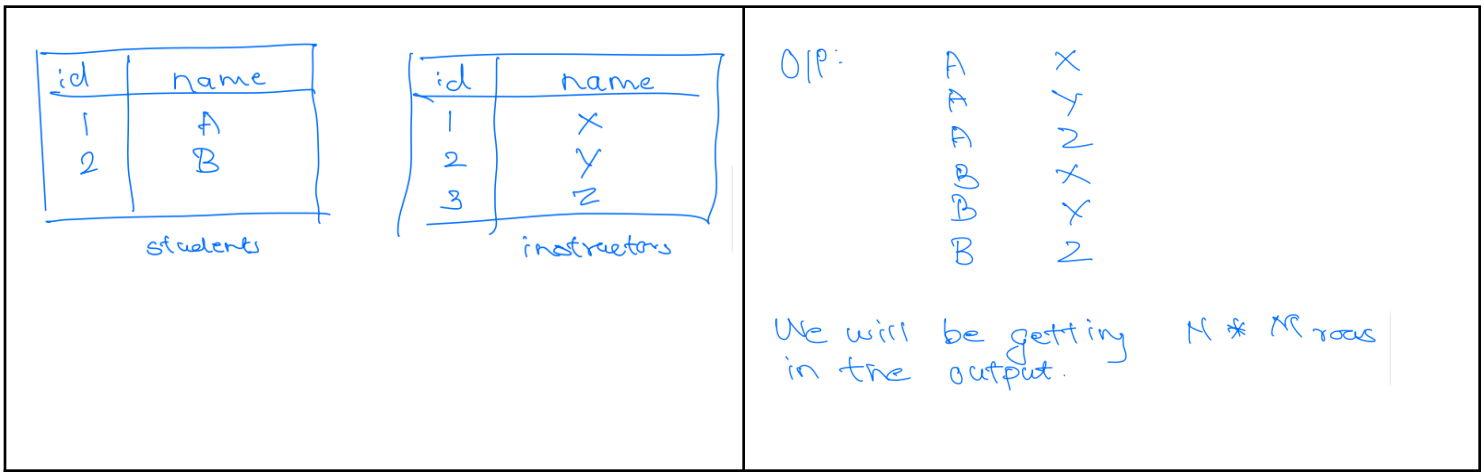
John	Doe	Batch A
Jane	Doe	Batch A
Jim	Brown	NULL
Jenny	Smith	NULL
Jack	Johnson	Batch B
NULL	NULL	Batch C

Full Outer Join is not supported in MySQL



- **Cross Join ->** A Cross Join, or Cartesian Join, combines each row from one table with all the rows from another table, resulting in a combination of all possible pairs of rows between the two tables.

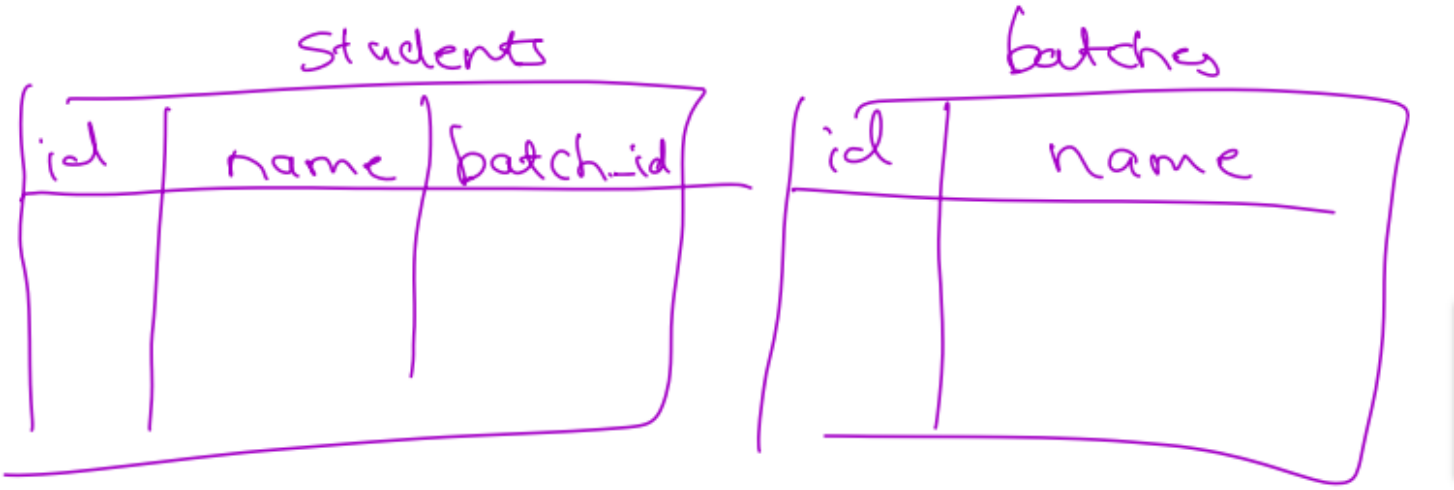
select s.name, i.name
from students s
cross join instructors



Syntactic Sugars

Using -> Using is use to inplace of join condition if

- For it to work column name must be the same in both tables
- All the types of join ar esupported.|



Using will not work in the above table senario

Below Both are same

select s.name, b.name from student s join batch b on s.batch_id = b.batch_id	select s.name, b.name from student s join batch b uisng (batch_id);

Natural Join -> A Natural Join specifically performs an Inner Join, and it doesn't support other types of joins.

select s.name, b.name from student s join batches b on s.batch_id = b.batch_id	select s.name, b.name from student s natural join batches b;

Implicit Join -> It will help us do cross joins

select * from students s cross join batches b;	select * from students s, batches b;

TC for below queries:

```
Select  s.name, b.name
from    student s
join    batches b
on      s.batchid =
        b.batch_id
```

```
Select  s.name, b.name
from    student s,
batches b
where   s.batch_id
        = b.batch_id
```

O/P of both queries is same.

Q1

nm TC to filter x rows (intersection part)

FROM

Q2

nm rows

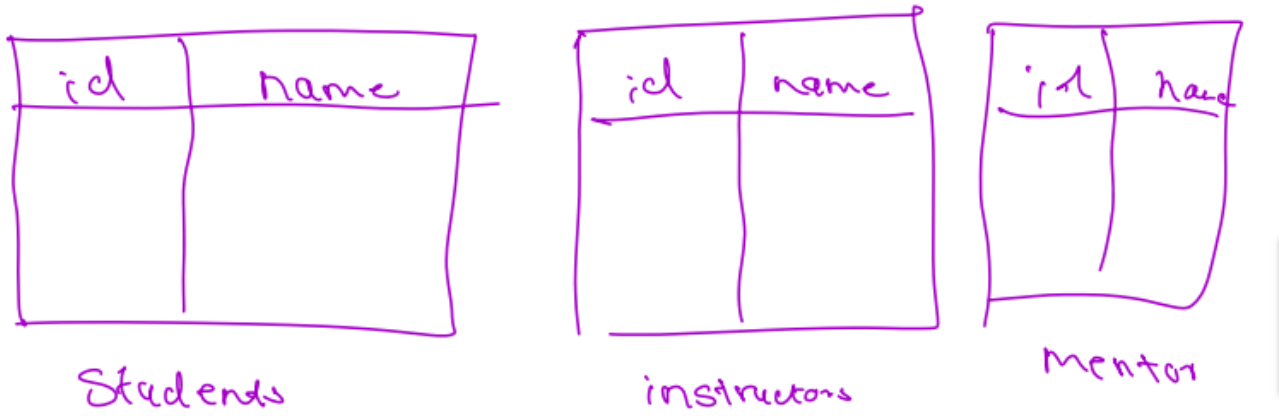
1

WHERE

nm tc to apply where

o/p will be x row.

Union -> A UNION query in SQL combines the result sets of two or more SELECT statements into a single result set. It removes duplicate rows by default and returns a distinct set of rows from all the combined SELECT statements. Each SELECT statement within a UNION must have the same number of columns, and the columns must have compatible data types.



Print name of all the people involved with scaler.

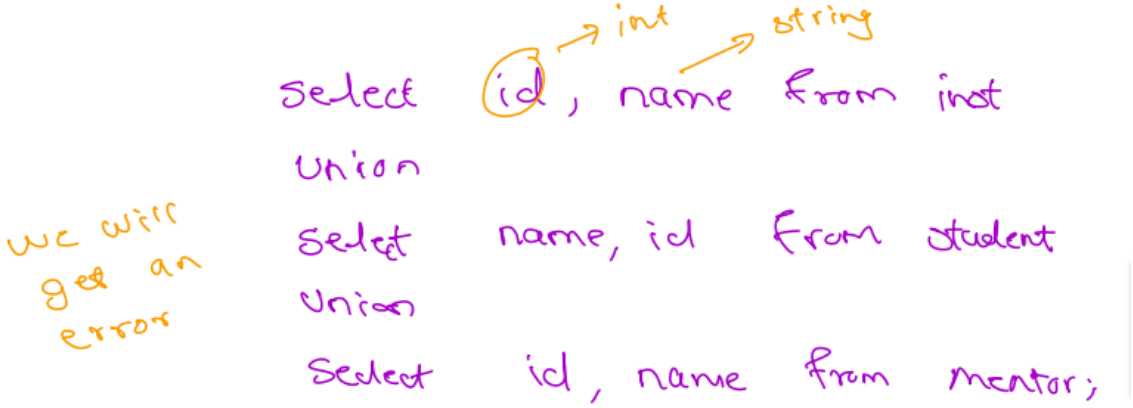
Opt #1

Write 3 seprate queries. Problem here is we will got 3 seprate output.

Opt #2

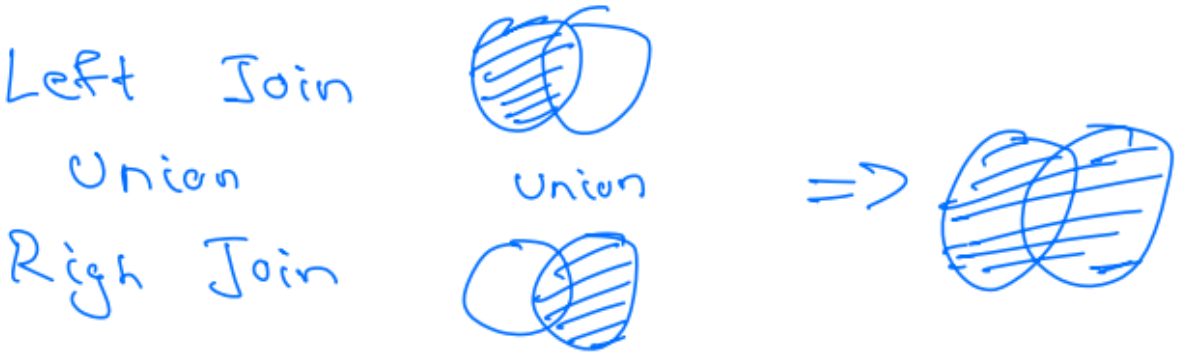
select name from students
union
select name from instructors
union
select name from mentor

O/P of union will be always unique
We sholud do a union with correct data type



What if I want duplicate values?
use "UNION ALL" instead of "union"

Full Outer Join is not supported in MySQL. How can we get o/p of FO join in MySQL?



All queries

```
-- Use the 'sql_store' database
USE sql_store;

-- Select all records from the 'customers' table
SELECT * FROM customers;

-- Select all records from the 'orders' table
SELECT * FROM orders;

-- Print customer name and order date using an inner join
SELECT c.first_name, c.last_name, o.order_date
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;

-- Print customer name and order date. Include customers with no orders using a left join
SELECT c.first_name, c.last_name, o.order_date
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

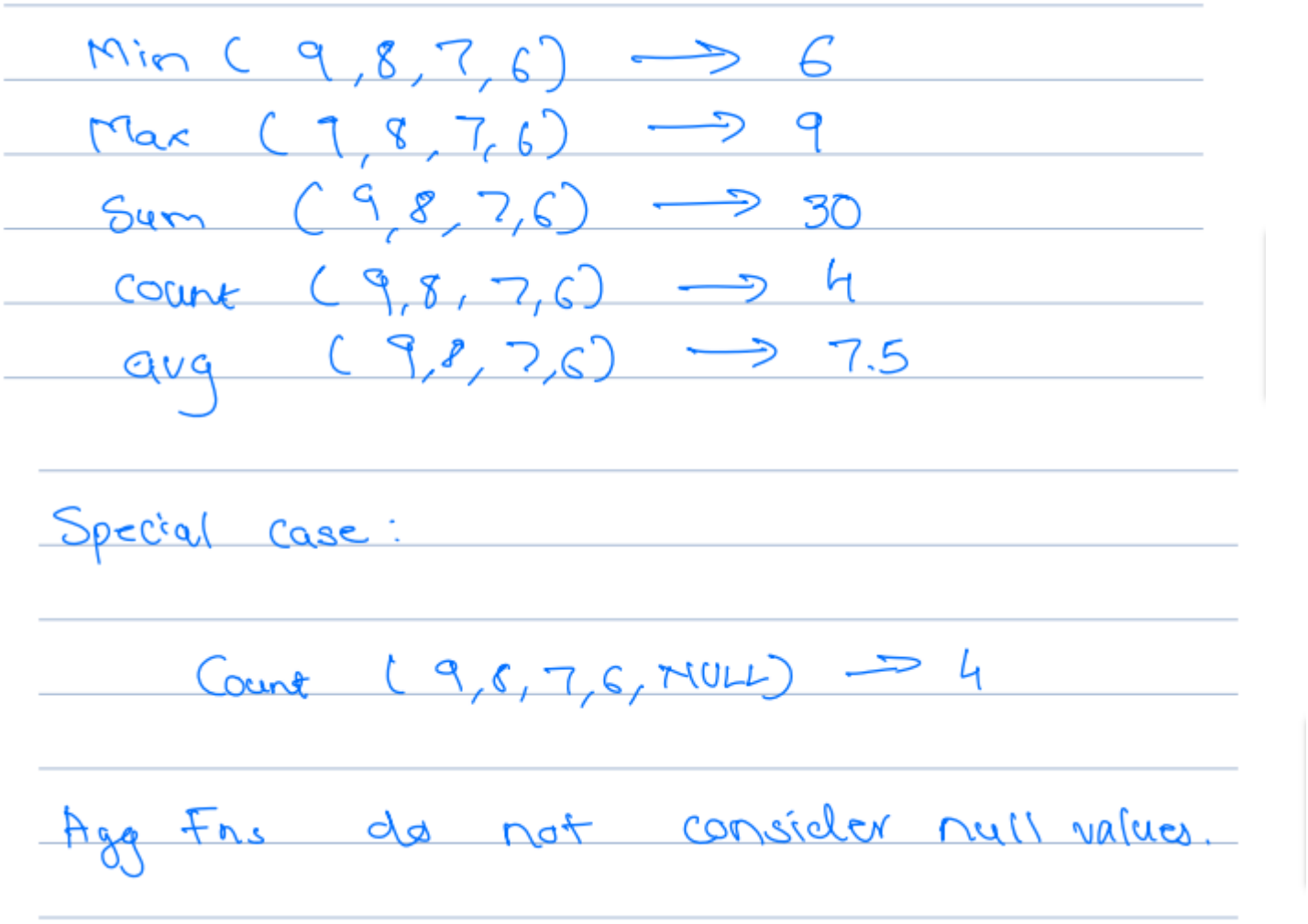
```
-- Print customer name and order date using a right join
SELECT c.first_name, c.last_name, o.order_date
FROM orders o
RIGHT JOIN customers c ON c.customer_id = o.customer_id;

-- Perform a cross join between 'customers' and 'orders'
SELECT *
FROM orders o
CROSS JOIN customers c;

-- Print customer name and order date using a right join with 'USING' clause
SELECT c.first_name, c.last_name, o.order_date
FROM orders o
RIGHT JOIN customers c USING (customer_id)
WHERE c.first_name LIKE 'E%';

-- Print customer name and order date using a natural join
SELECT c.first_name, c.last_name, o.order_date
FROM customers c
NATURAL JOIN orders o;

-- Perform a cross join via implicit join
SELECT *
FROM customers c, orders o;

-- Switch to the 'sakila' database
USE sakila;

-- Print customer names and actor names together without duplicates
SELECT c.first_name, c.last_name FROM customer c
UNION
SELECT a.first_name, a.last_name FROM actor a;

-- Print customer names and actor names together with duplicates
SELECT c.first_name, c.last_name FROM customer c
UNION ALL
SELECT a.first_name, a.last_name FROM actor a;
```

SQL:Aggregate Queries - 5th Oct 2023

- **Aggregate Queries**
- **GROUP BY clause**
- **HAVING Clause**

Aggregate Queries - > aggregate functions are functions that operate on a set of values, combining them to produce a single result
Sql has 5 aggregate functions

Min, Max, Count, Sum, Avg



Note: To obtain the count, we won't consider any specific column due to the possibility of containing null values. Therefore, we will use "COUNT(*)" as it directly counts unique rows, providing a reliable result.

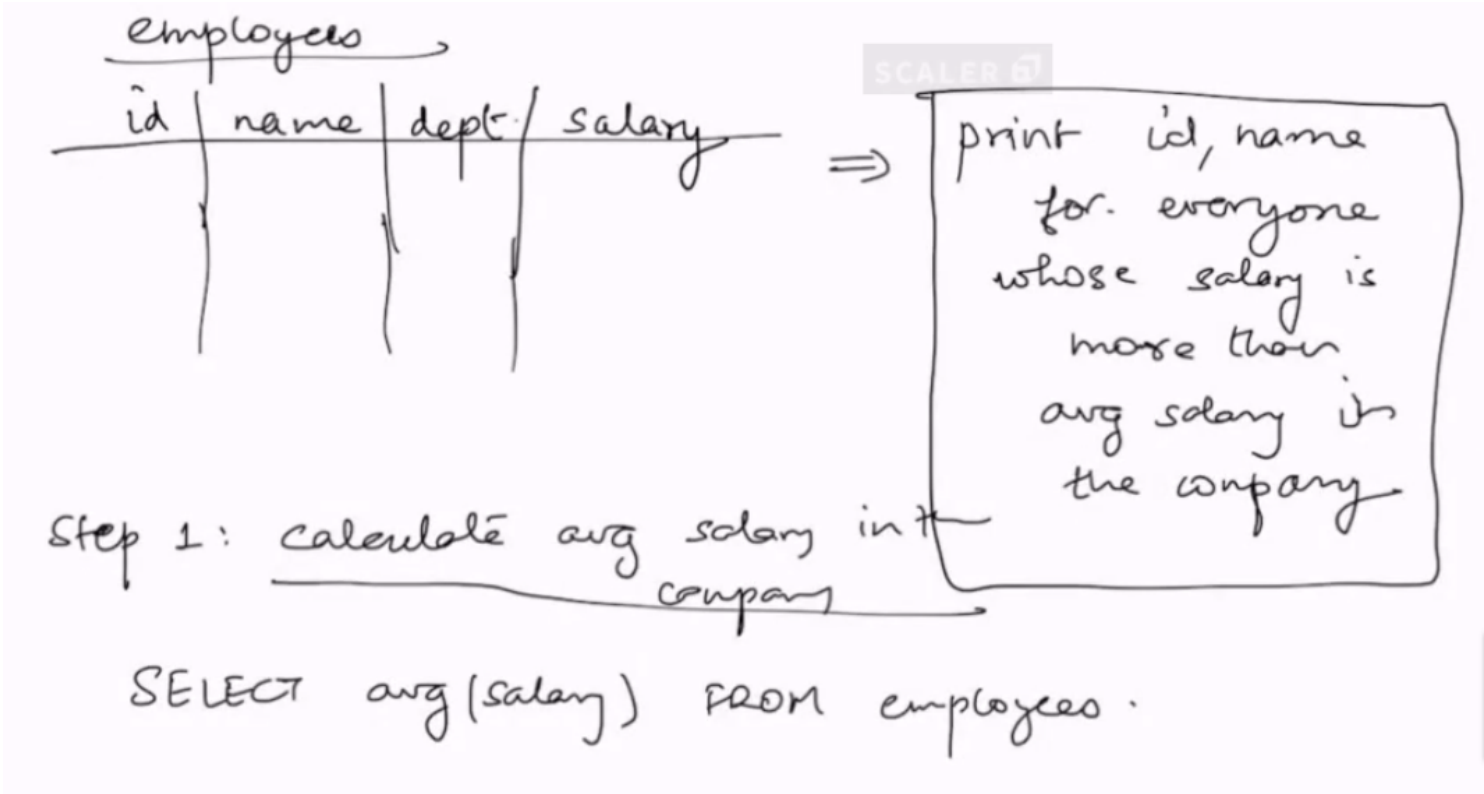
```
select count(*) from customers;

select count(customer_id) from customers;
```

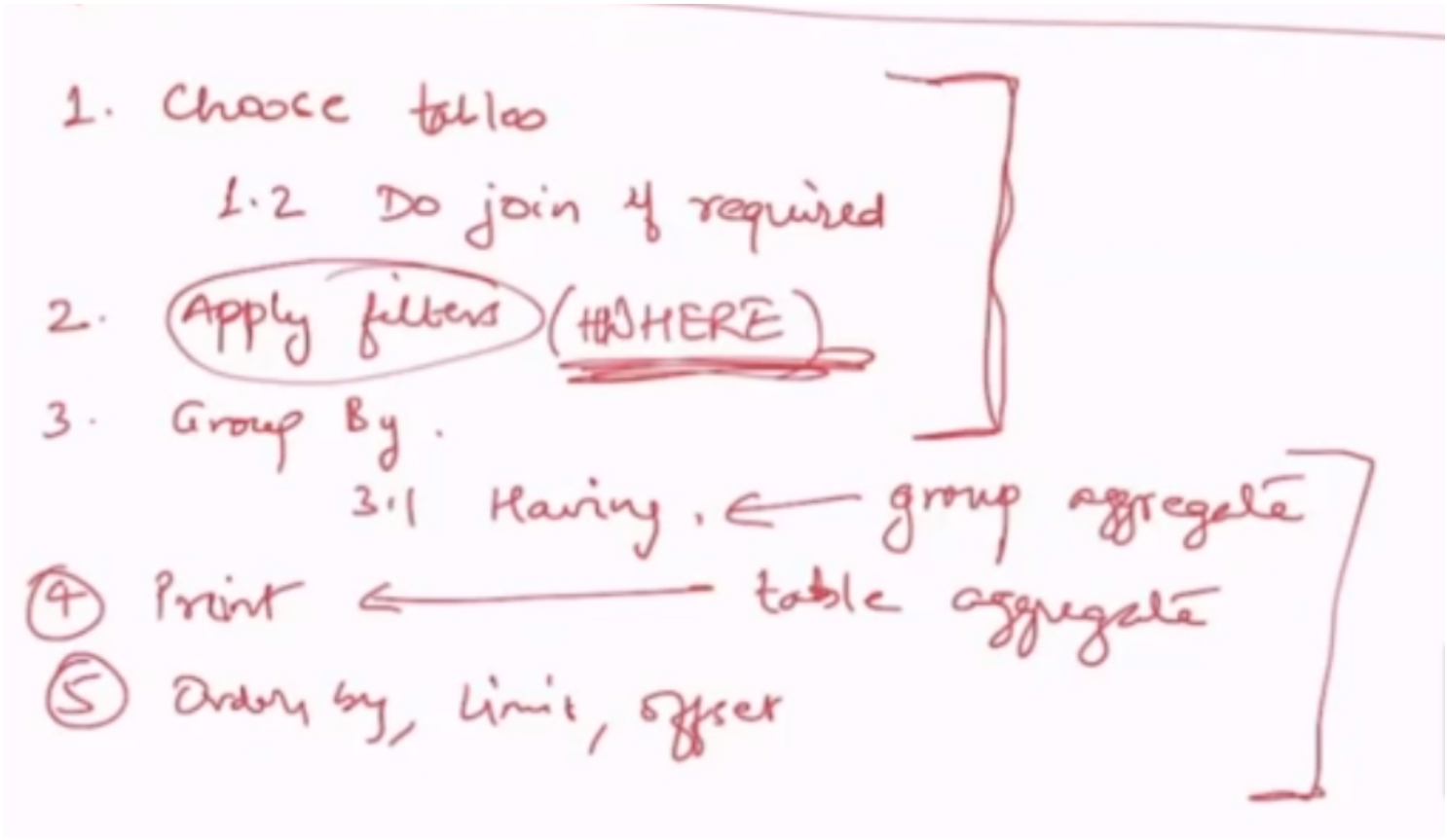
select min(points) from customers c;

select max(points) from customers c;

select avg(points) from customers c;

select sum(points) from customers c;





Aggregate function with filter

```
select avg(psp)
from students where batch_id = 1;
```

Print avg (psp) of batch with id 1.

id	name	batch_id	psp
1	A	1	60
2	B	1	20
3	C	2	80
4	D	2	90

Expected output: 65

Group by -> When using the "GROUP BY" clause in SQL, you are instructing the database to group rows with the same values in specified columns together. This is typically followed by aggregate functions such as "SUM," "COUNT," or "AVG" to perform calculations on each group of rows separately.

Note:

1. If columns are included in the `SELECT` statement (except when used with aggregate functions), they must also be part of the `GROUP BY` clause.
2. There is no limit to the number of aggregate functions you can use; you can use as many as needed.
3. Avoid including the 'id' column in the `GROUP BY` clause, as doing so may result in multiple rows in the output.

Print batch_id & its avg psp

id	name	batch_id	psp
1	A	1	60
2	B	1	70
3	C	2	80
4	D	2	90

Desired o/p :

batch_id	avg(psp)
1	65
2	85

Steps to get the output

1. Break the rows into groups depending upon their bhatch_id
2. Apply the avg fn for every group.
3. Print the output

```
select batch_id, avg(psp)
from students
group by batch_id
```

Print avg(psp) for batch_id & grand year

```
select branch_id, grand_year, avg(psp)
from students
group by batch_id, grand_year
```

id	name	batch_id	psp	grad_year
1	A	1	60	2020
2	B	1	70	2021
3	C	1	80	2020
4	D	2	90	2022

O/P:

batch_id	grad_year	avg(psp)
1	2020	70
1	2021	70
2	2022	90

① Group Creation

G1	G2	G3
(1, 2020)	(1, 2021)	(2, 2022)
(R1, R3)	(R2)	(R4)

② Apply agg f^n to all the groups.

batch_id	grad_year	avg(psp)
1	2020	70
1	2021	70
2	2022	90

Having -> the HAVING clause is used to filter the results of a GROUP BY query based on a specified condition. It is similar to the WHERE clause, but while the WHERE clause filters rows before they are grouped, the HAVING clause filters the results after they have been grouped. Here's how it works:

1. First, you use the GROUP BY clause to group rows that have something in common, typically using an aggregate function like SUM, COUNT, AVG, etc., to summarize data within each group.
2. Then, you can use the HAVING clause to filter the groups based on a condition. Only the groups that meet the condition specified in the HAVING clause will be included in the result set.

Note -> that whenever you use the HAVING clause, you should remember to include an aggregate function in the SELECT statement and also in the HAVING clause.

Here's an example to illustrate how the HAVING clause works:

Q. Suppose you have a table named orders with columns customer_id and total_price, and you want to find the average total price of orders for customers who have placed more than one order:

```
SELECT customer_id, AVG(total_price)
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 1;
```

Q. Print batch_id and avg(psp). only include those batches in the output whose avg(psp) > 70;

id	name	batch_id	psp
1	A	1	95
2	B	1	10
3	C	2	60
4	D	2	90

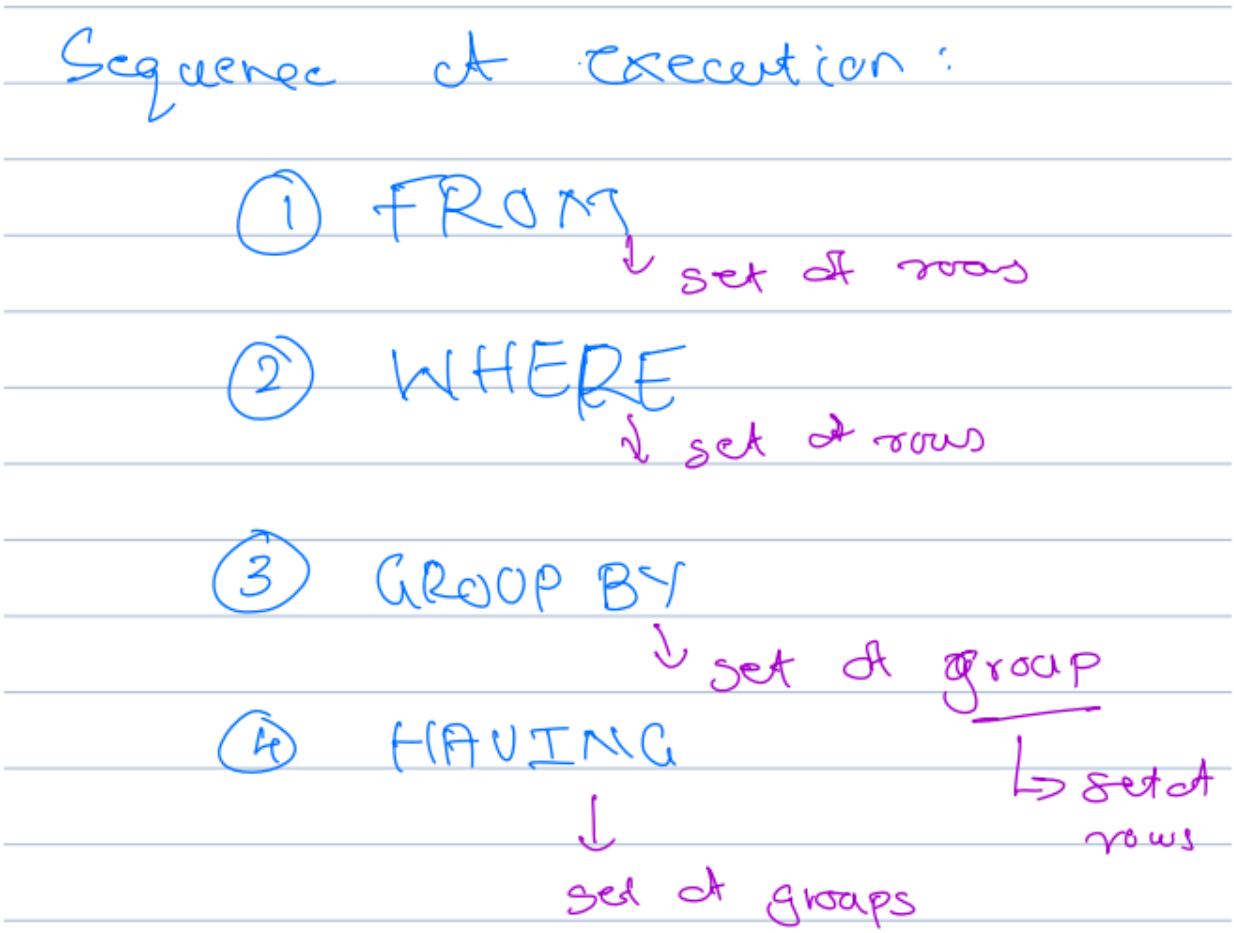
O/p:

batch_id	avg(psp)
2	75

Select batch_id, avg(psp)
from students
group by batch_id
having avg(psp) > 70

Having v/s where

Having	Where
(1) Used to filter groups	(1) used to filter rows
(2) Happens after group by.	(2) Happens before group by.



Select the tables.

1.2. Do the joins if rwquired

1. **Apply the filters**
2. **Group by**

   3.2: Filtering happens with having clause
3. **Select what to print including aggregates**
4. **Orde by, limit, offset.**

```
select a.name as actor_name, count(fa.actor_id) as number_of_film
from actor a join film_actor fa
on a.id = fa.actor_id
// name is not requreid we can print it directly because it is present in join table
group by fa.actor_id,
having count(1) > 25 // we can use both count(*) or count(1) both same
order by count(1) desc // we also use aliases like number_of_film
limit 5
```

Aggregate queryies

Note- >

1. Aggregate functions cannot be used directly in the WHERE clause within the same query because the WHERE clause is evaluated before the aggregate functions. This means that you cannot filter or conditionally select rows based on the results of an aggregate function within the same query.
2. Having only works with group by and available after where clause whenever records grouped also use aggregation function in having instead aliases even we can use aliases

```
select count(*) from customers;
select count(customer_id) from customers;
select count(phone) from customers c ;
select min(points) from customers c;
select max(points) from customers c;
select avg(points) from customers c;
select sum(points) from customers c;

-- take sum of points of people living in FL
select sum(points) from customers where state = 'FL';



-- aggregate avg points per state
select state, avg(points) from customers group by state;

-- aggregate avg points per state and per city
select city, state, avg(points) from customers group by city,state;

-- groups will be created on the basis of city and state combination
-- KA, Bangalore
-- KA, Mangalore
-- Imagine group by city, state to result into a hashmap
```

-- where the key is {city_state} and value is aggregated value for this key aggregate avg points, and count of people per state and per city

select city, state, avg(points), count(*) **from** customers **group by** city,state;

-- print state, avg points and only include those states whose avg points > 1000

select state, avg(points) **from** customers **group by** state having avg(points) > 1000;

select state, avg(points) as average **from** customers **group by** state **having** average > 1000;

select state, avg(points) **from** customers **group by** state **having** state = "KA";

-- print state, avg points and only include those states whose avg points > 1000 and state name starts with C

select state, avg(points) **from** customers **where** state like 'C%'
group by state having avg(points) > 1000;

select state, avg(points) **from** customers **group by** state
having avg(points) > 1000 AND state like 'C%';

-- print state, avg(points) where state has more 2 people living in it

select state, avg(points) as avg_points **from** customers **group by** state having count(*) > 2;

SQL: Subqueries and Views - 17th Oct 2023

- **Subqueries**
- **Subqueries and IN clause**
- **Subqueries in FROM clause**
- **ALL and ANY**
- **Correlated subqueries**
- **EXISTS**

Subqueries -> A subquery, also known as an inner query or nested query, is a SQL query embedded within another SQL query. Subqueries are used to retrieve data from one or more tables based on the result of another query. They are commonly used for various purposes, including filtering, retrieving specific data, or performing calculations within a larger query.

Here are some key characteristics and use cases for subqueries:

Subquery Types: Subqueries can appear in various parts of a SQL statement, including the SELECT, FROM, WHERE, and HAVING clauses.

Note ->

1. When your query result relies on aggregate functions like MAX or is closely tied to the outcome of an existing query, using subqueries is often a better approach than joins to obtain the desired results.

2. Indexes play a significant role in optimizing queries. When indexing is absent, the performance of subqueries and joins can be similar. However, when indexing is in place, joins tend to be more optimized compared to subqueries.

3. Every query involving joins can be expressed using subqueries, but not every subquery can be rewritten using joins.

4. Subquery can we written as result single value, row, column, table

Students

id, name, psp, b_id

I

Find all the students who psp is greater than the max psp of batch 2.

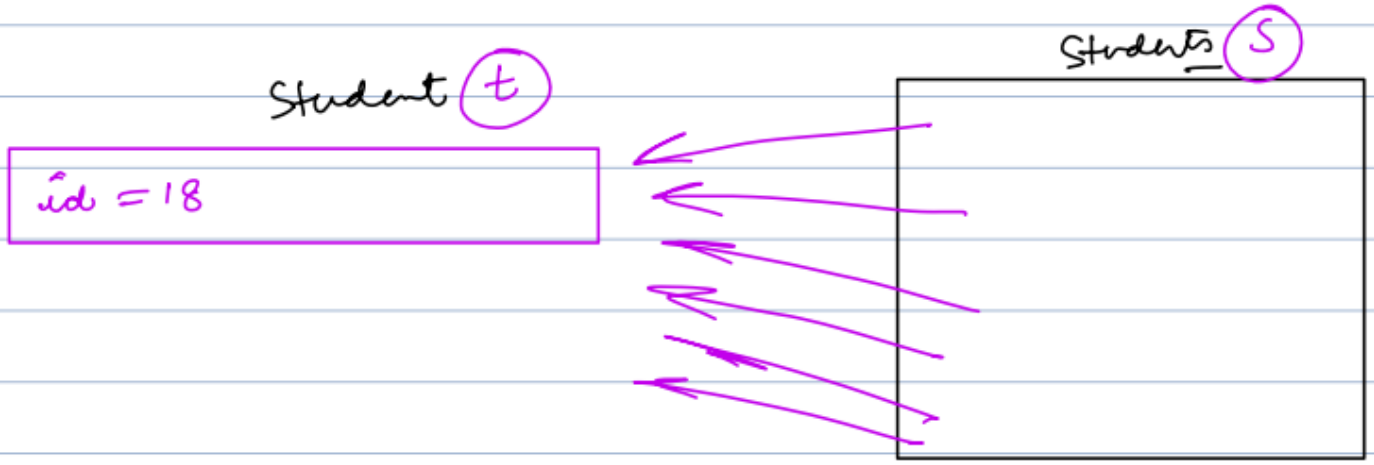
II

A	60	1
B	70	1
C	45	2
D	52	3
E	39	2

```
x_max_psp
for all students (x)
    if x.bid = 2
        x_max_psp = max(
            x_max_psp, x.psp)
```

```
ans = []
for all students (x)
    if ( x.psp > x_max_psp)
        ans.insert(x);
```

- Find the students whose psp is greater than psp of student with id = 18.




```
select *
from students s
join students t
on t.id = 18 and t.psp < s.psp;
```

Find the students whose psp is greater than psp of student with id = 18

(I)

(I) Find the psp of student id = 18.

```
select psp from students
where id = 18;
```

(II) Find all the students whose psp is greater than x.

```
select * from students
where psp > (x);
```

```
select * from students
where psp > (select psp from students
where id = 18);
```

Find all the students who psp is greater than the max psp of batch 2.

II

I Find all the students whose psp is greater than x

```
select * from students
where psp > (x);
```

II Find max psp of batch 2

```
select max(psp) from students
where b_id = 2;
```

```
select * from students
where psp > (select max(psp) from students
              where b_id = 2);
```

Subqueries and IN clause -> Subqueries and the IN clause are closely related in SQL and are often used together. The IN clause is used to compare a value to a set of values produced by a subquery. It allows you to filter rows based on whether a value matches any value in a set produced by a subquery.

users

IN

id , name , is_ta, is_stud

Tell the name of the students which are also the name of TA's.

			is_ta	is_stud
1	Ashutosh		1	0
2	Vidushi		0	1
3	Ayushi		1	0
→	4	Ayush	1	1
5	Vidushi		1	0
6	Ayushi		1	0

student

id, name

TA

id, name

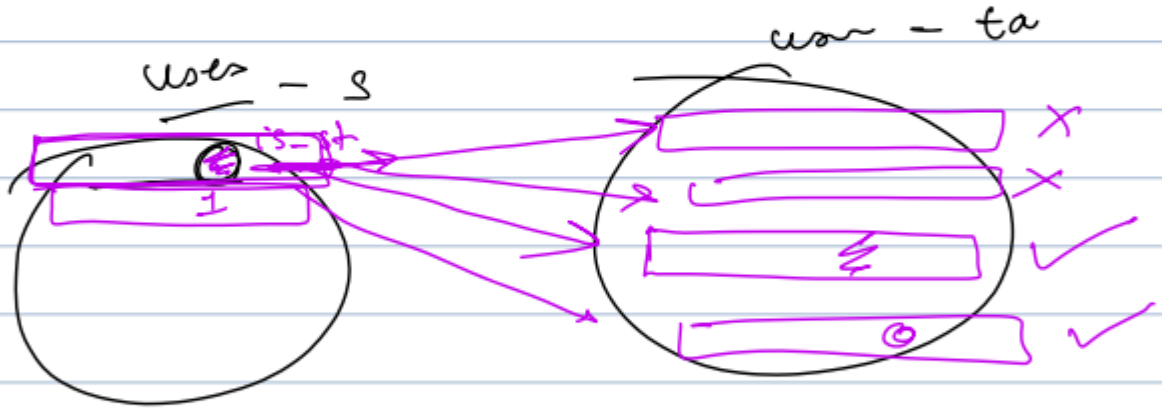
↓

join

```
select * from
  students s
join TA t
  on s.name = t.name;
```

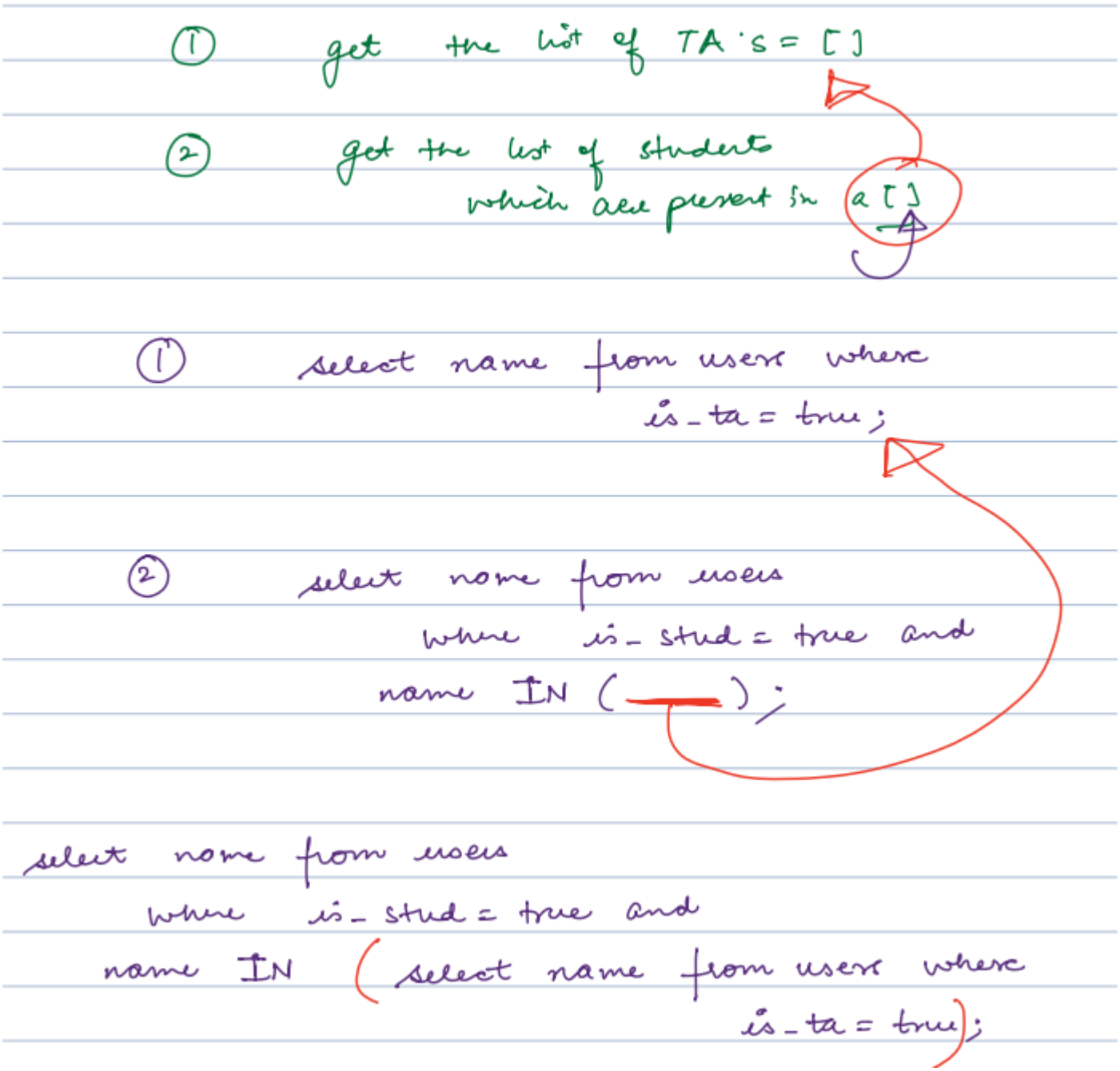
id , name , is_ta, is_stud

```
select *
from   users  s
join   users  t
on     s.is_stud = true
   and t.is_ta = true
   and s.name = t.name;
```



Tell the names of students which also the name of TA's.

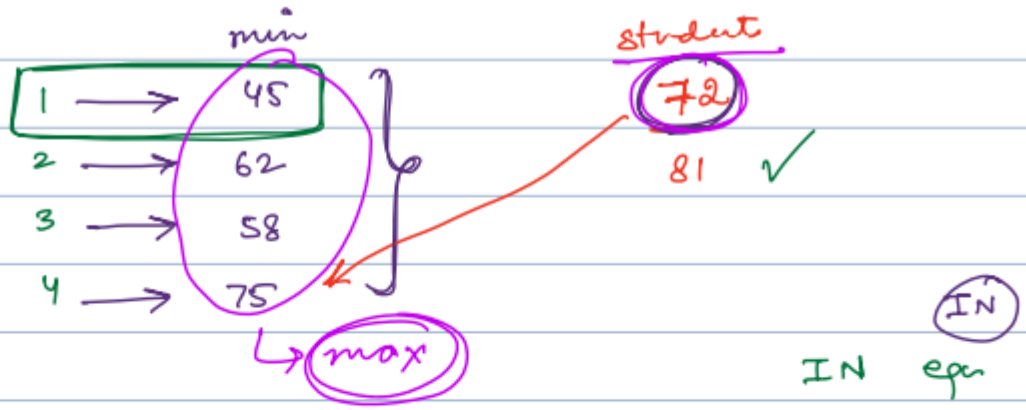
IN (_, _, _, _, _)



Subqueries in FROM clause -> Subqueries in the FROM clause are used to treat the result of a subquery as a derived table or a temporary dataset. This derived table can then be used as if it were an actual table within your SQL query. The primary purpose of using subqueries in the FROM clause is to simplify complex queries by breaking them into smaller, more manageable parts.

```
SELECT department_name, AVG(salary) AS avg_salaryFROM (
SELECT d.department_name, e.salary
        FROM employees e
 JOIN departments d ON e.department_id = d.department_id
) AS employee_department
GROUP BY department_name;
```

Q Find all the students whose psp is greater than min psp of every batch.



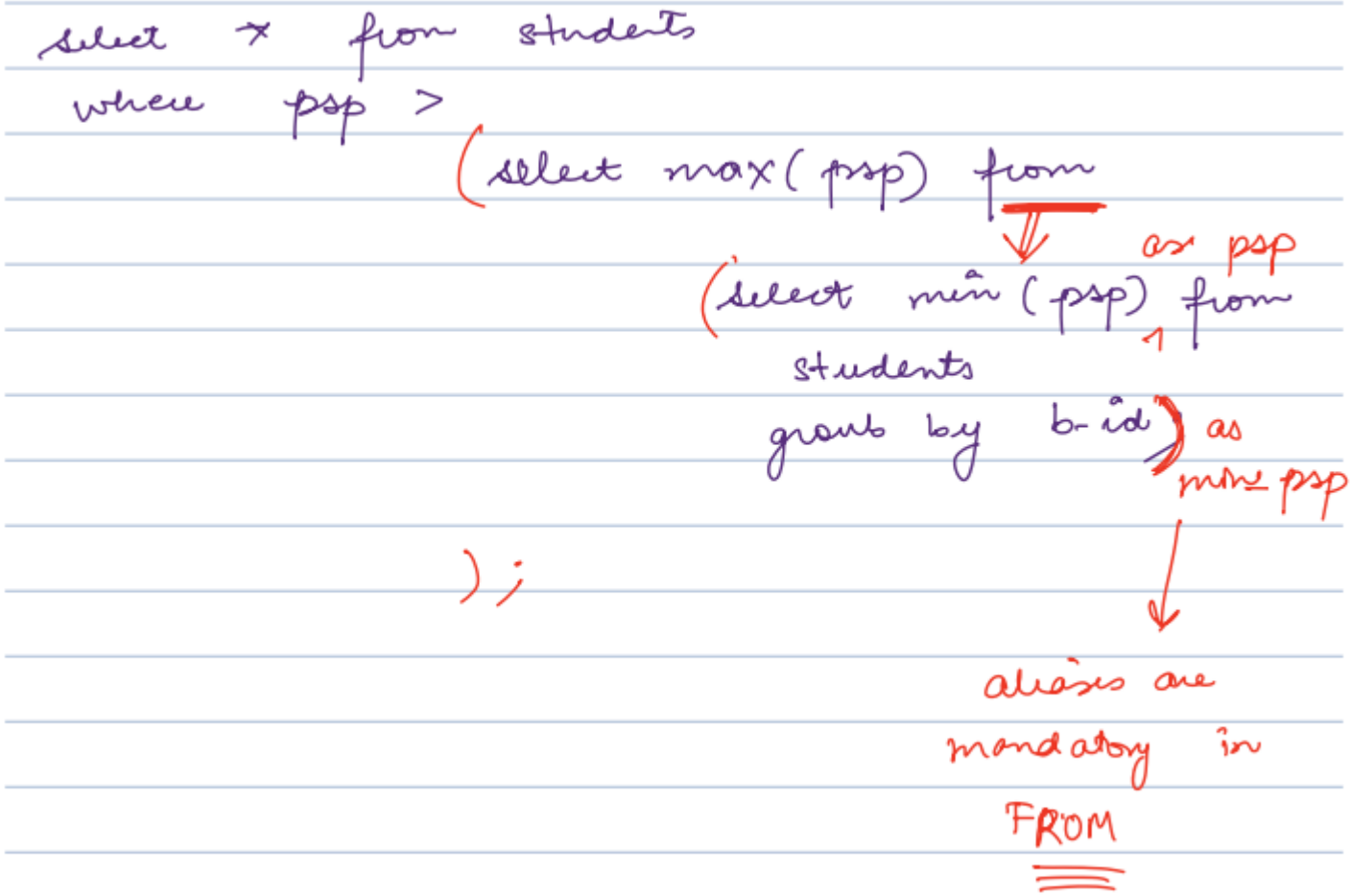
s.psp > MAX (MIN of all batches psp)

III II I

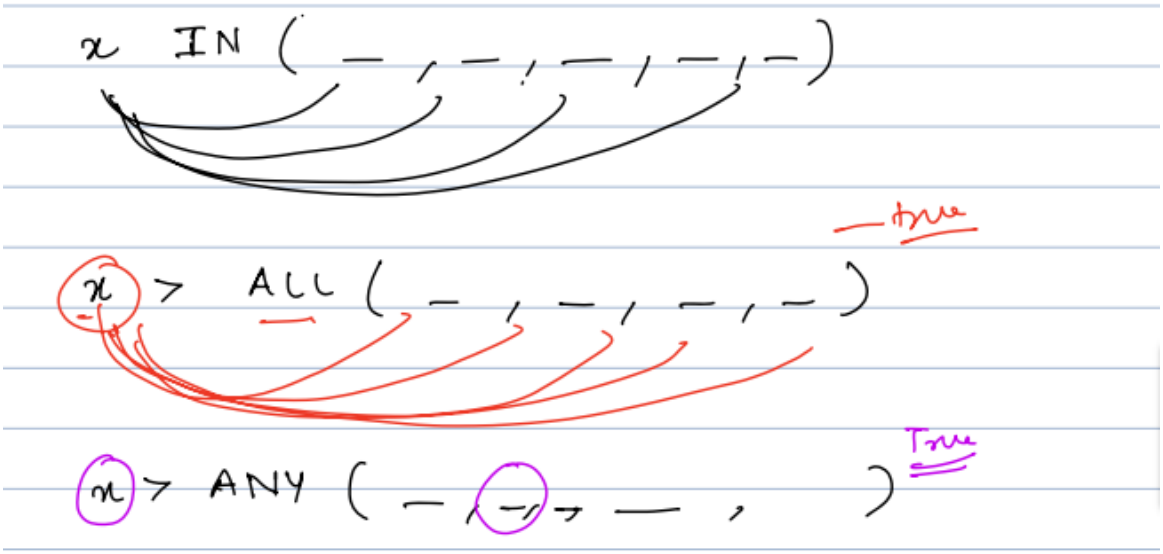
I select min(psp) from students group by b_id;

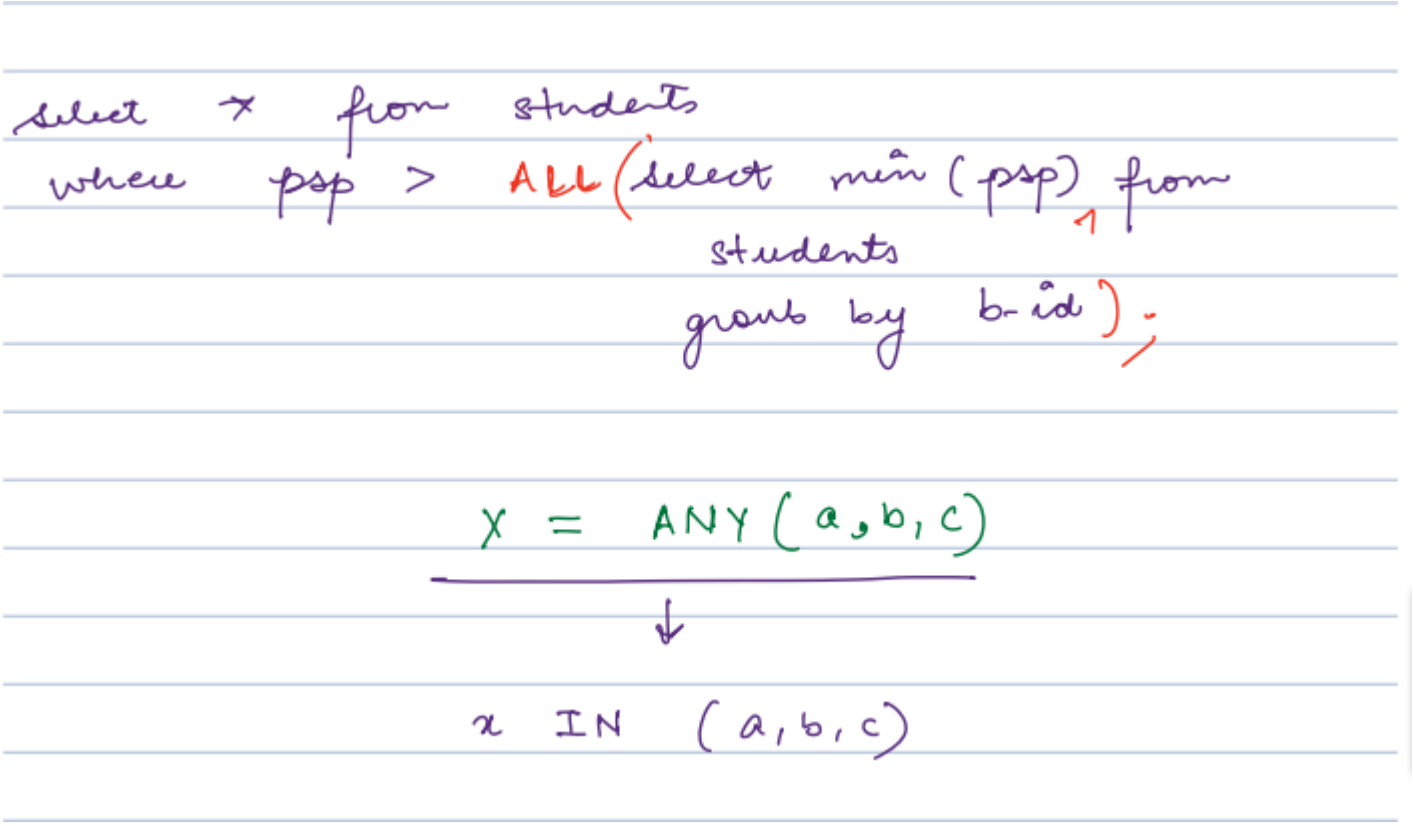
II select max(psp) from x;

III select * from students where psp > y;



ALL and ANY -> ALL and ANY are SQL operators used in combination with a comparison operator (such as <, >, =, <=, >=, <>, etc.) to compare a value to a set of values produced by a subquery. They are often used in the WHERE clause to filter rows based on specific conditions.





Correlated subqueries: A correlated subquery is a type of subquery in SQL that references columns from the outer (main) query within its own query. The outer query and the subquery are connected in such a way that the subquery's result is dependent on the current row being processed by the outer query. Correlated subqueries are used to filter, retrieve, or perform calculations on a per-row basis.

Key characteristics of correlated subqueries:

1. The subquery's result depends on the values of one or more columns from the current row of the outer query.
2. Evaluation for Each Row: For each row in the outer query's result set, the subquery is executed separately, taking into account the specific values of the correlated columns for that row.
3. Used in Filtering and Comparison: Correlated subqueries are often used in the WHERE clause to filter rows based on specific conditions, typically comparing values from the correlated columns.

SELECT employee_name **FROM** employees e1
WHERE salary > **(SELECT** AVG(salary) **FROM** employees e2 **WHERE** e2.department_id = e1.department_id**);**

Get all the students whose psp is greater than avg psp of their batch.

(II) (I)

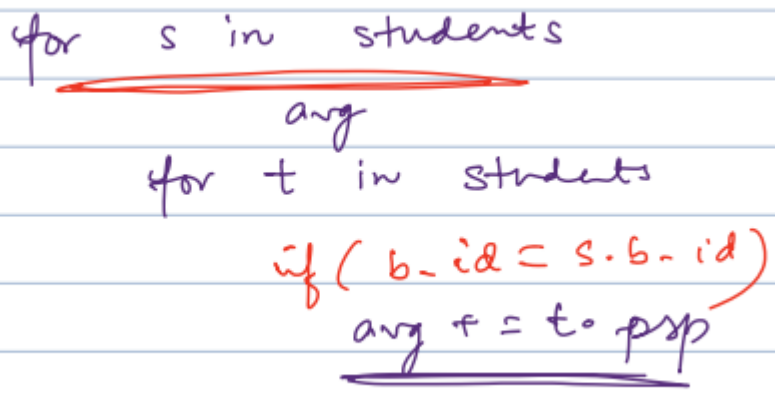
1	A	1
2	B	2
3	C	1
4	D	1
5	E	3

(I) select AVG (psp) from students where b-id = y;

(II) select * from Students where psp > x;

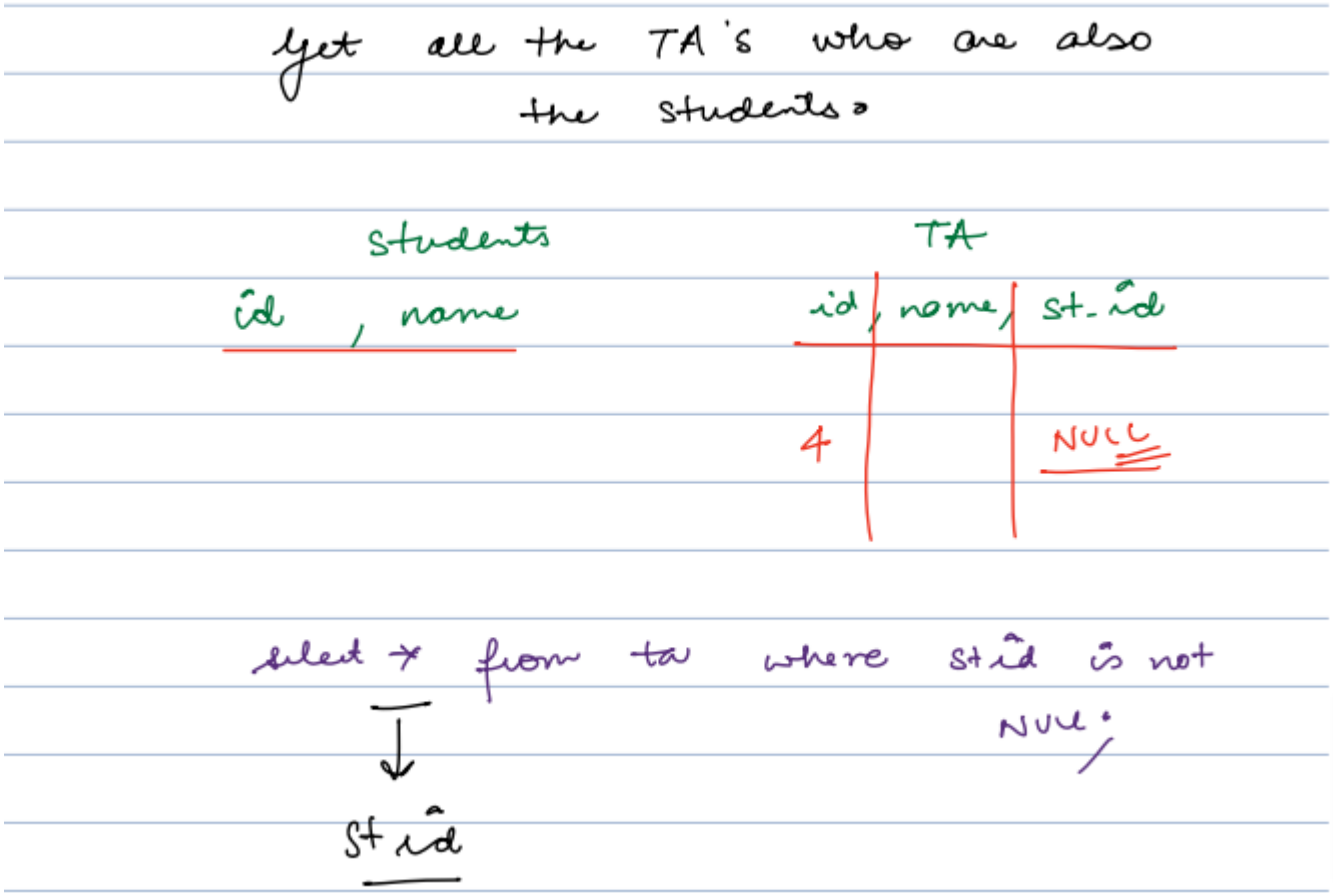
```
select * from
Students S where
psp > (select AVG (psp) from students
       where b-id = S.bid);
```

correlated subquery



EXISTS -> The EXISTS operator is used in SQL to determine whether a subquery returns any rows. It returns a Boolean value, typically TRUE or FALSE, based on whether the subquery's result set is empty or not. The EXISTS operator is often used in the WHERE clause to filter rows in the main query based on the existence of rows in the subquery's result set.

It will optimize the query by braking loop on find first occurrence of record.



- Get all the students who are also a TA.

Students

id	name
2	Mohit

TA

id	name	st_id
4		NULL

```
select * from student s where
  id IN ( select st_id from ta where
          st_id is not null ) ;
          and st_id = s.id
```

```
select * from student s where
  id EXISTS ( select st_id from ta where
              st_id = s.id )
```

```
for s in students
    for t in ta
        if( t.stid = s.id)
        {   flag = true;
            break;
        }
    if( flag == true)
        ans.insert(s);
```

```
for s in students
        temp=()
        for t in ta
            if( t.s_id is not null)
                temp.insert(s.e'd)
```

nested subq

```
IN {  for x in temp
        if (s.id = x)
            ans.insert(x) break;
```

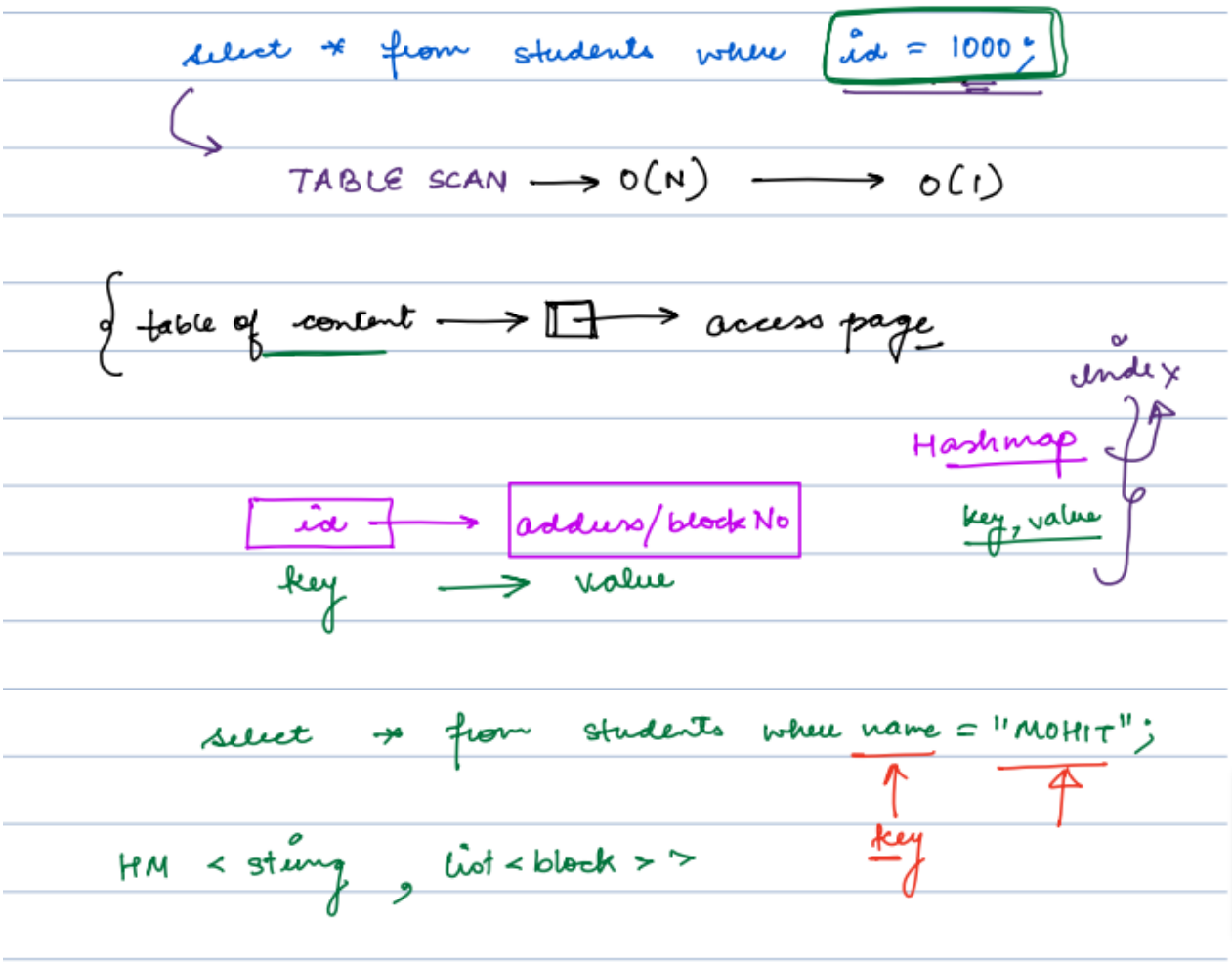
SQL: Indexing - 19th Oct 2023

- **Introduction to Indexing**
- **How Indexes Work**
- **Indexes and Range Queries**
- **Cons of Indexes**
- **Indexes on Multiple Columns**
- **Indexing on Strings**
- **How to create index**

Introduction to Indexing

An index is a data structure that provides a quick and efficient way to access rows (records) in a database table. It works like a table of contents in a book, allowing you to find the location of specific data quickly without having to scan the entire dataset.

Indexes are used to speed up data retrieval operations, such as SELECT queries in a database, by reducing the number of records that need to be examined. Without indexes, the database system would have to perform full table scans, which can be very slow for large datasets.



How Indexing Works:

Index Key: An index is created on one or more columns of a table. The column(s) used for indexing are referred to as the index key(s).

Data Structure: The index key values are stored in a data structure that allows for efficient searching, such as B-Trees, Hash Tables, or Bitmaps, depending on the database system and the use case.

Pointers: The index structure also contains pointers to the actual data rows. These pointers help the database system locate the corresponding records quickly.

Sorting: Indexes are typically sorted, which allows for binary search operations, making data retrieval faster.

Types of Indexes:

B-Tree Index: This is the most common type of index, suitable for range queries and exact match queries. It maintains data in a balanced tree structure.

Hash Index: Hash indexes are optimized for fast exact match queries and are particularly useful for large datasets with unique keys.

Bitmap Index: Bitmap indexes are used for columns with low cardinality (a small number of distinct values), where each bit in the index represents the presence or absence of a particular value for a record.

Full-Text Index: These indexes are used for searching within large text fields and provide support for full-text search capabilities.

Indexes and Range Queries:

Note:

1. It employs B+ Trees (Balanced Trees), which are more efficient than BBST (Balanced Binary Search Trees). The primary distinction between the two lies in the fact that B+ trees are not confined to having just two child nodes. Instead, they do not have a limitation on the number of 2 child nodes. This characteristic aids in reducing the height of the tree, resulting in significantly fewer iterations.
2. By default on primary key indexing create by database
3. Avoid creating indexes prematurely

key (name)	list <block>
Mohit	3, 7, 8
Murali	1, 9
Sanjay	2
⋮	⋮

select * from students where psp >= 50.1 and psp <= 70.9 ;

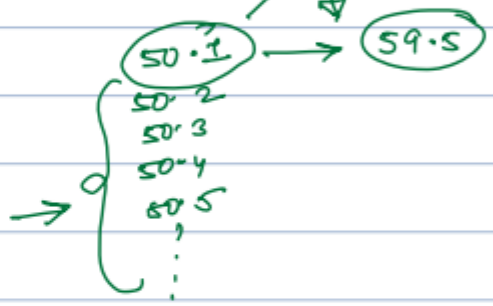
key

are HM's helpful ?

ordered ?

⇓

unordered



check a value equal → HM

Hashmap + ordered

key stored in sorted manner

TreeMap / ordered_map

Balanced Binary Search Trees

search

O(H)

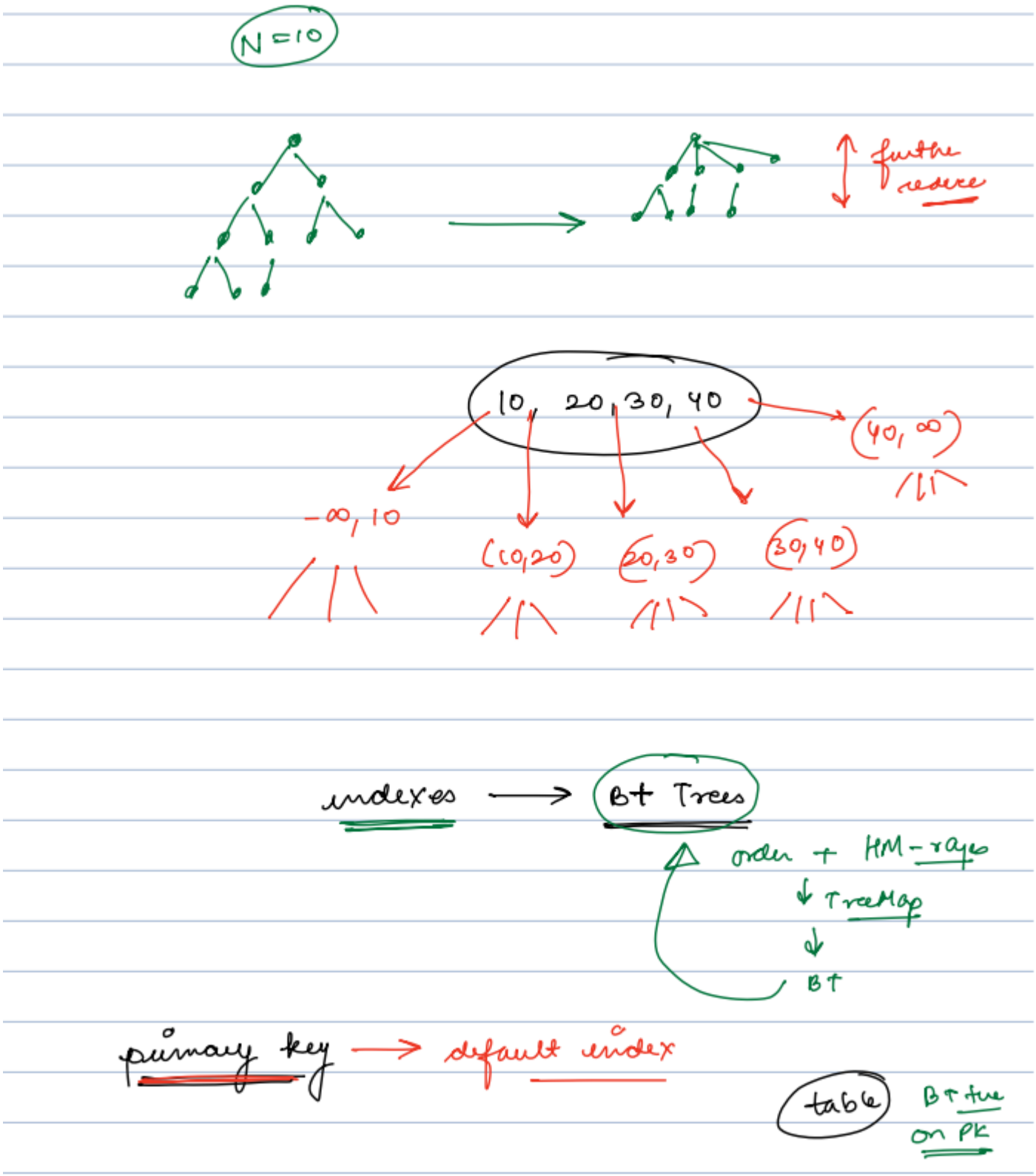
H = $(\log_2 N)$

no of nodes

B+ Trees

<= x children

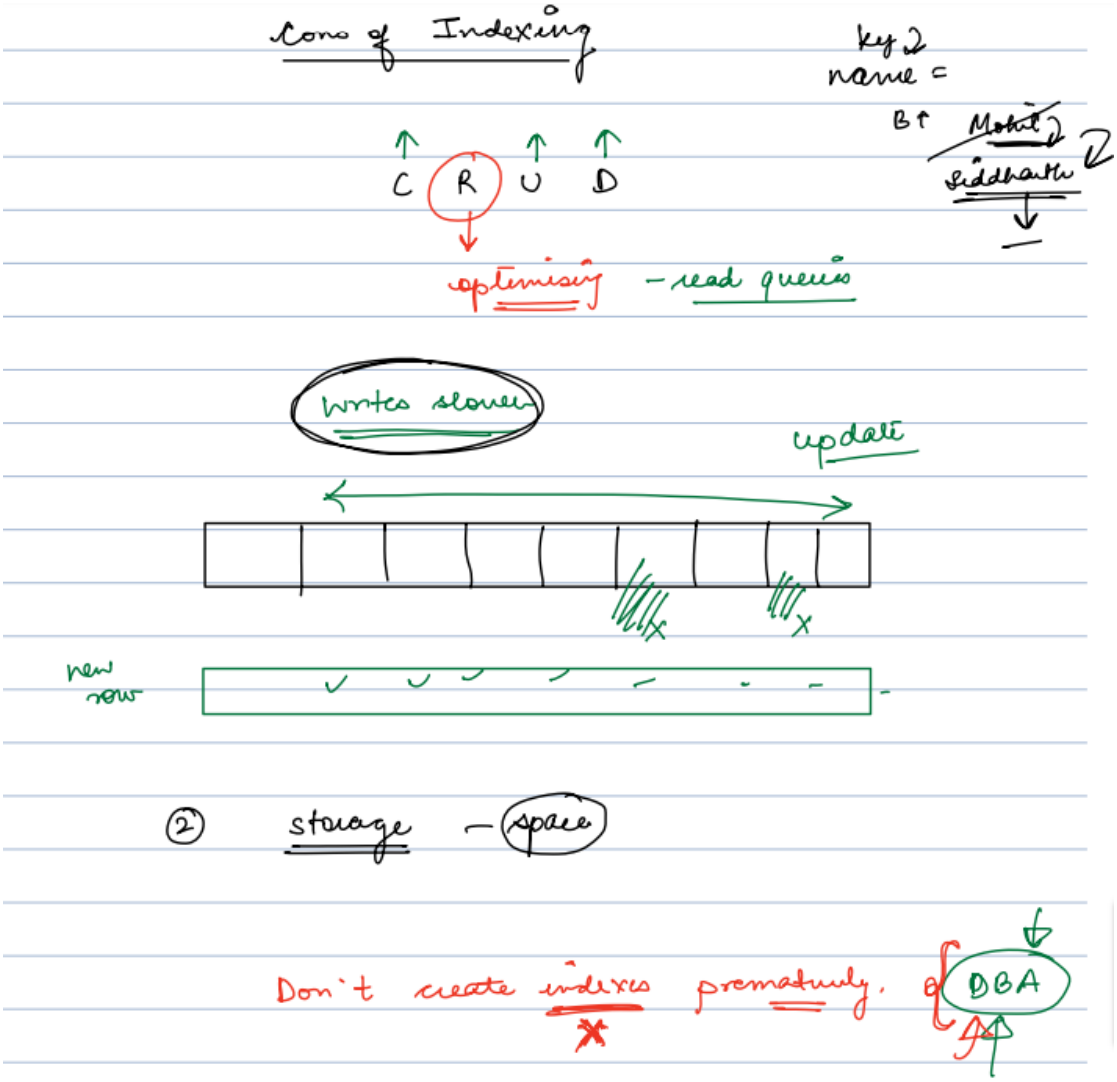
Balance + Tree structure



Cons of Indexes

While indexing is essential for improving data retrieval performance, it also has some disadvantages and trade-offs that need to be considered. Here are some of the cons of indexing:

- **Increased Storage Requirements:** Indexes consume storage space, and in large databases with many indexes, the storage overhead can be significant. This can lead to increased storage costs.
- **Slower Data Modification:** When data in a table is **inserted**, **updated**, or **deleted**, the associated indexes must be updated as well. This can slow down data modification operations and lead to increased write times.
- **Complex Maintenance:** Indexes need to be maintained as the underlying data changes. This maintenance can involve reorganizing the index structure, which requires additional processing and can impact overall system performance.
- **Choice Overhead:** Deciding which columns to index and how many indexes to create is not a trivial task. Poorly chosen indexes can lead to unnecessary overhead without providing significant benefits.



Indexes on Multiple Columns -> "Indexes on Multiple Columns" refers to the practice of creating indexes that involve more than one column in a database table. This can significantly impact query performance and allows for efficient retrieval of data based on combinations of values in those columns. Here's an overview of the concept:

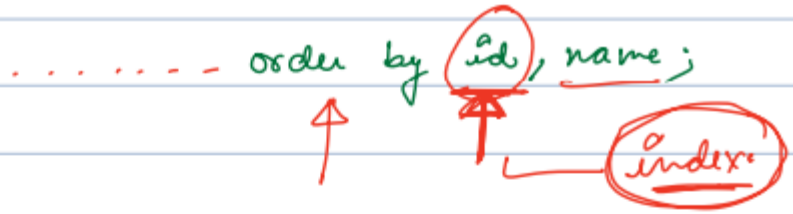
Composite Index: An index on multiple columns is often referred to as a composite index or a multi-column index. This type of index is created by specifying two or more columns when defining the index.

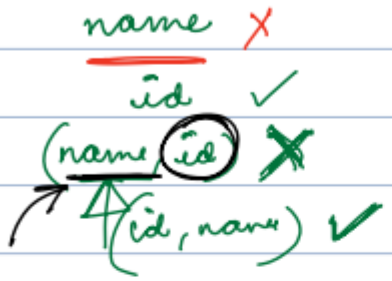
Benefits:

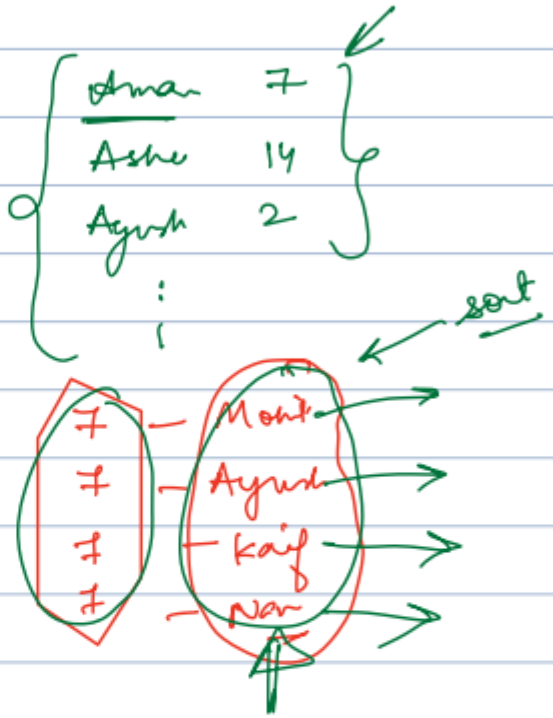
Improved Query Performance: Composite indexes are useful when you need to retrieve data based on specific combinations of column values. They can speed up queries that involve conditions on multiple columns.

CREATE INDEX idx_employee_name_dept ON employees (last_name, department);

Note -> Whenever create index keep prefix for idx_

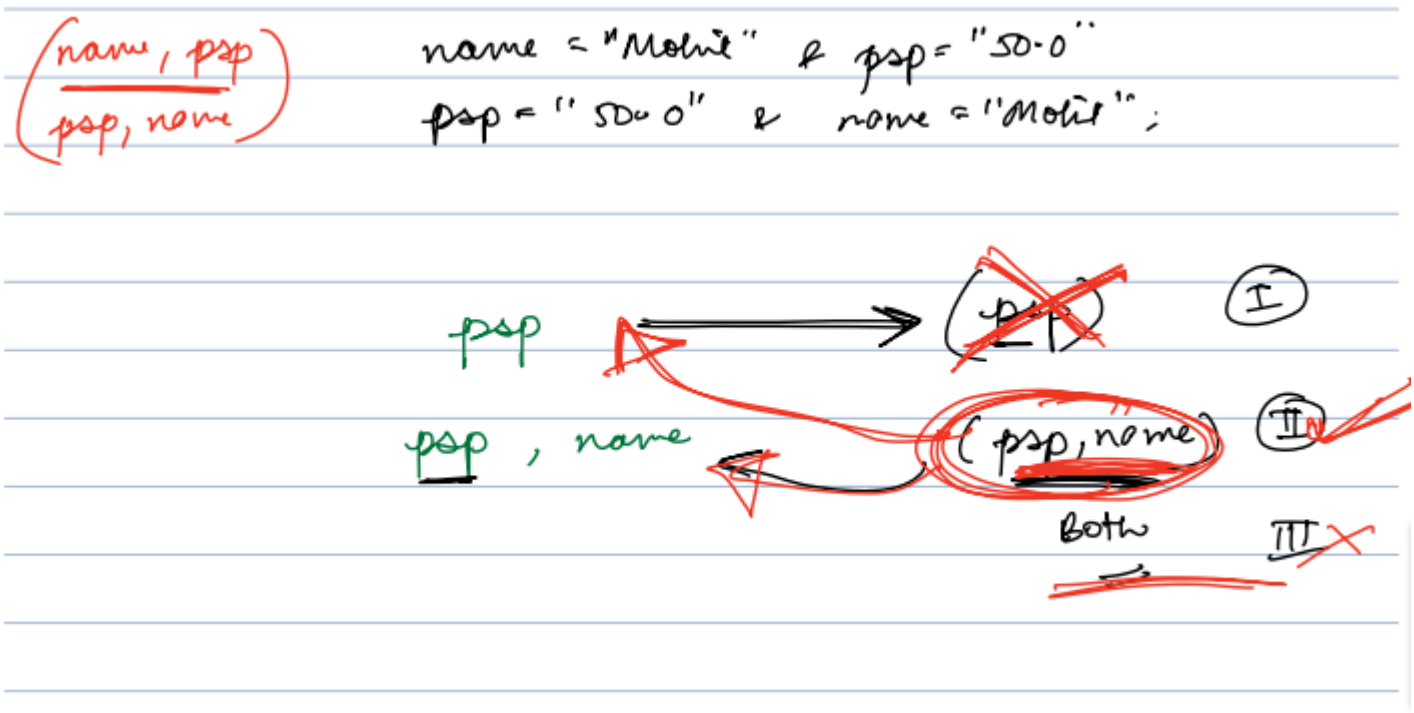






query	index
name	psp ✗
name	(name) ✓ (psp)
name	(psp, name) ✗
name	(name, psp) ✓
name = ____ & psp = ____	(name, psp) ✓
	(psp, name) ✓

In most cases, the order of conditions in a query does not affect the result. Consequently, the order of columns in a composite index may also be insignificant. For instance, consider the conditions on "name" and "psp"; whether the composite indexing is "psp, name" or "name, psp," it will have the same effect.



Indexing on Strings

"Indexing on Strings" refers to the process of creating indexes on columns or fields that contain string (text) data in a database. String indexing is a critical aspect of database management, as it allows for efficient retrieval of data based on text values. Here's an overview of indexing on strings:

String Data Types: String indexing typically involves columns with data types such as VARCHAR, CHAR, TEXT, or other string-related types. These columns may store a wide range of textual information, including names, descriptions, addresses, or any text-based data.

Benefits of String Indexing:

- **Faster Data Retrieval:** String indexes accelerate the retrieval of data when you need to search or filter records based on text values. Without indexes, the database would have to perform full table scans, which can be slow for large datasets.
- **Optimized for LIKE and Text Searches:** String indexes are especially useful when you perform operations like pattern matching (using the LIKE operator) or full-text searches within the text data.

- **Support for Ordering:** String indexes can enhance the sorting of textual data, making it faster to retrieve data in sorted order.

Creating String Indexes: String indexes are created using SQL statements like "CREATE INDEX" in relational databases. When defining the index, you specify the column containing the text data that you want to index.
Types of String Indexes: String indexes can be created as single-column indexes or as composite indexes (covering multiple columns). The specific type of index and indexing method used may vary depending on the database system and requirements.
Full-Text Indexing: Full-text indexes are a specialized form of string indexing designed for efficient full-text searches. They often involve advanced indexing techniques, such as **tokenization**, **stemming**, and **stop-word** removal, to improve search accuracy.

When ever we will create full-text indexing we have to use max 7-8 charr indexing it will cover most of the text area

How to create index

Create index index_name
on table_name(list of column);

explain -> It provides information about the query execution plan, including how indexes are used

```
explain select rental_duration, count(film_id)
from film
where rental_rate = 4.99
group by rental_duration
having count(film_id) >= 70;
```

Column as index —----------

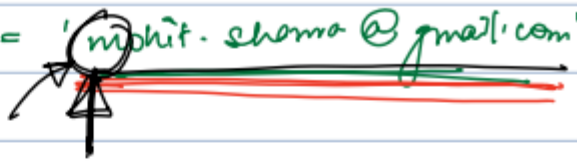
```
create index idx_rental_duration
on film(rental_duration);
```

If we are making large text index keep character 7 to 8. It will not allowed full description

```
create index idx_description
on film(description(7));
```

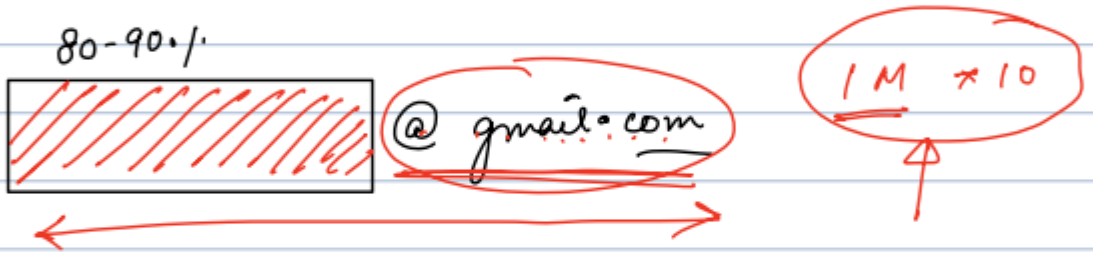
select * from users where

email = 'mohit.sharma@gmail.com'



index

- size of string ↑
- string comparison is slow



2B records $2*10^9$

26 $\quad$ 26*26 = 676

= 17576 $\quad$ $2*10^9$

$= 26^4 = 4*10^5$

$= 26^5 = 1*10^7$

$= 3*10^8$

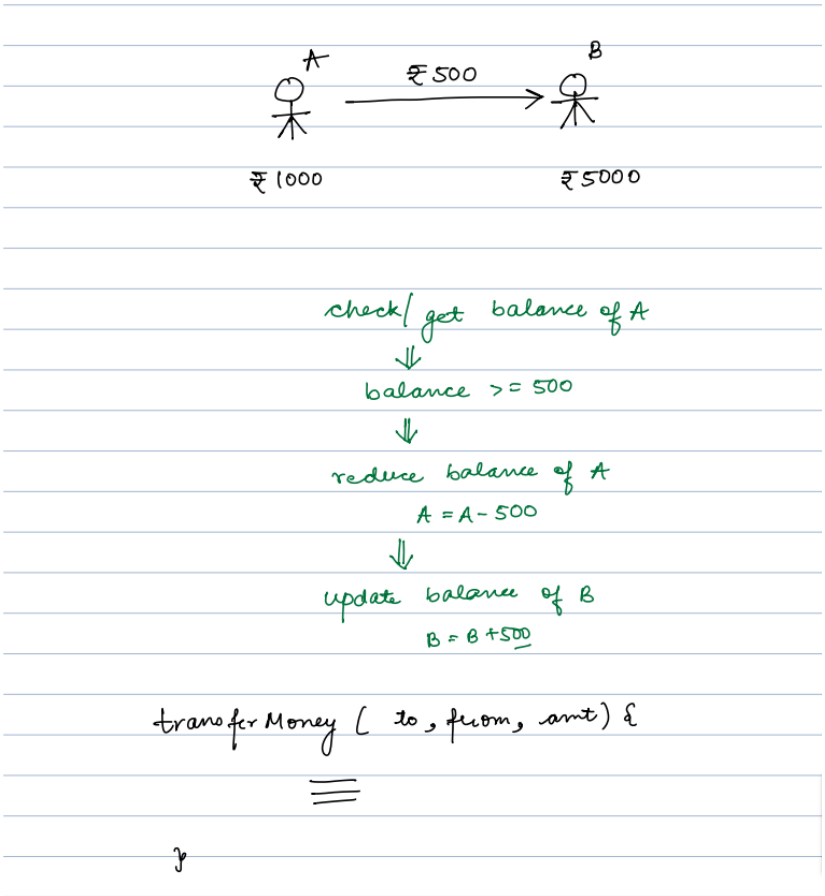
$8*10^9$

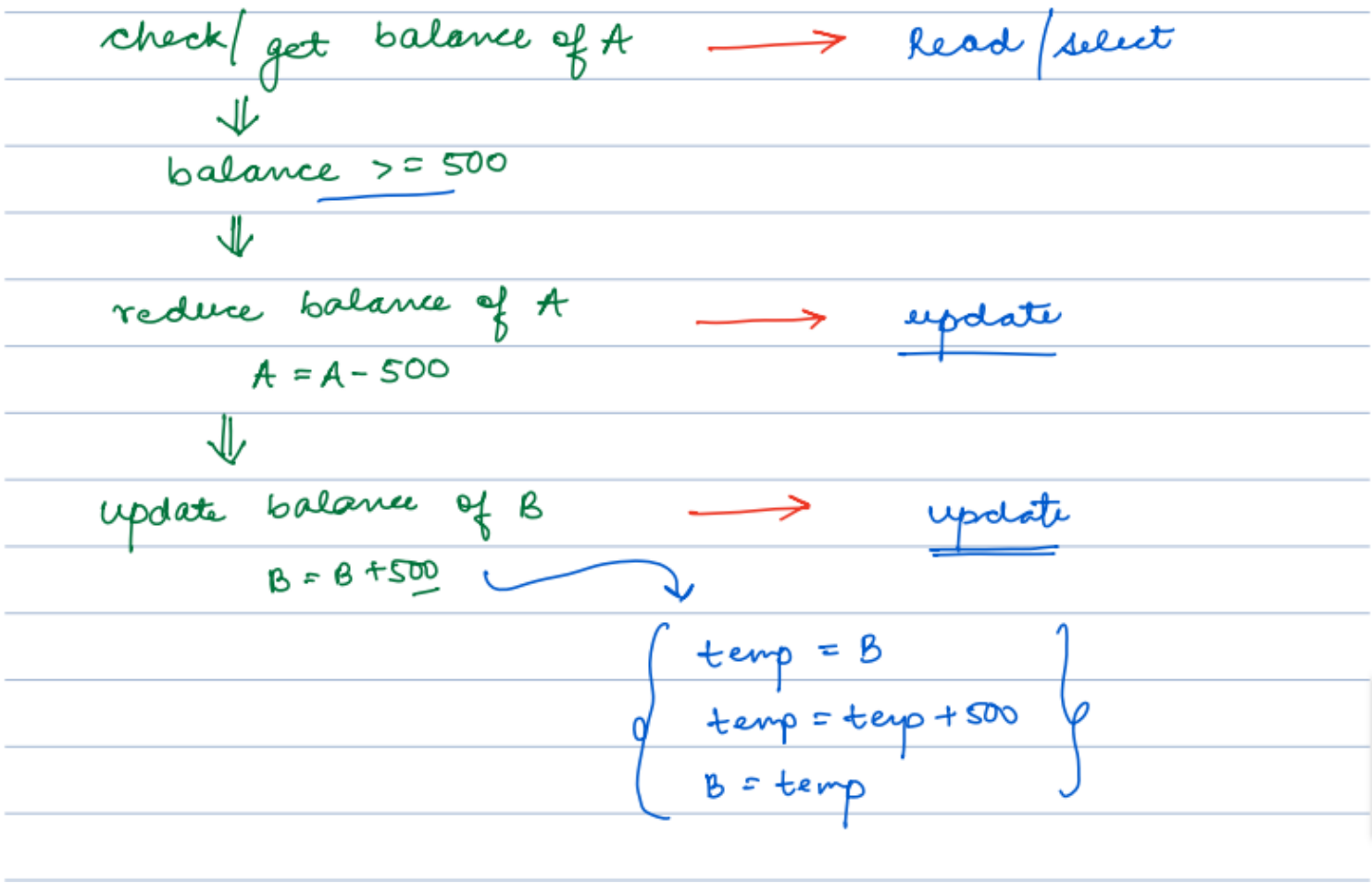
SQL: Transactions - 21st Oct 2023

- **What are transactions?**
- **ACID Properties**
- **ACID Properties - Atomicity**
- **ACID Properties - Consistency**
- **ACID Properties - Isolation**
- **ACID Properties - Durability**
- **ACID Properties - Summarized**
- **Commits and Rollbacks**
- **Transaction Isolation Level**
- **Transaction Isolation Level - Read Uncommitted**

What are transactions?

a "transaction" refers to a logical unit of work that consists of one or more operations. Transactions are used to ensure the integrity and consistency of data. Or **A set of DB operation logically put together to perform a certain task like a money transfer called transaction**





Let see the below scenario

As we can see in below scenario two people A and C transferring money to B on same time and both check balance of B which intilay 5000 and both added there transferred to it using initial balance which is wrong because if we will see A is trasfering 500 and which makes B balance 5500 and C is transferring 10000 which makes B balance 15000 but expected balance of B is 15500. Problem is here both transferring on same time and there is no interaction to each other and transaction overridden happen This happened because we came up in inconsistent and illogical state and one more issue arise if order got canceld and money transferred failed to solve this issue we will use transaction.

A → B

A = ~~10,000~~ 9,500

b = ~~5000~~ ~~5500~~ 15,000

amt = 500

C → B

C = ~~15,000~~ 5,000

b = ~~5000~~ 15,000

amt = 10,000

x ← read A

x = 10,000

if(x >= 500)

↓

A = A − 500

= 9500

y = read(B)

y = 5000

y = y + 500

= 5500

B = y

= 5500

x ← read C

x = 15,000

if (x >= 10,000)

C = C − 10,000

= 5000

y = read(B)

y = 5000

y = y + 10,000

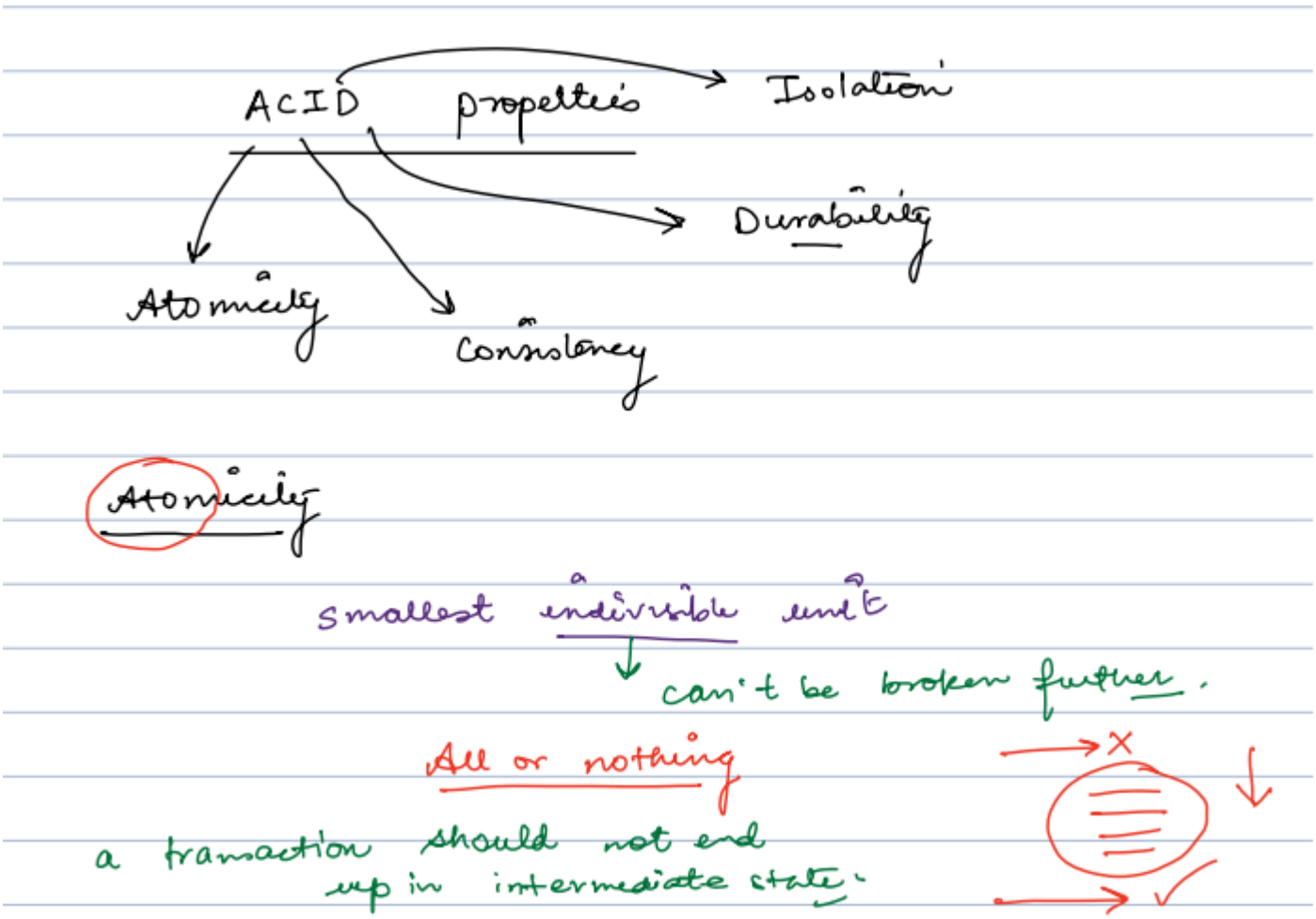
= 15,000

B = y

= 15,000

Expected	15500
actual	15,000

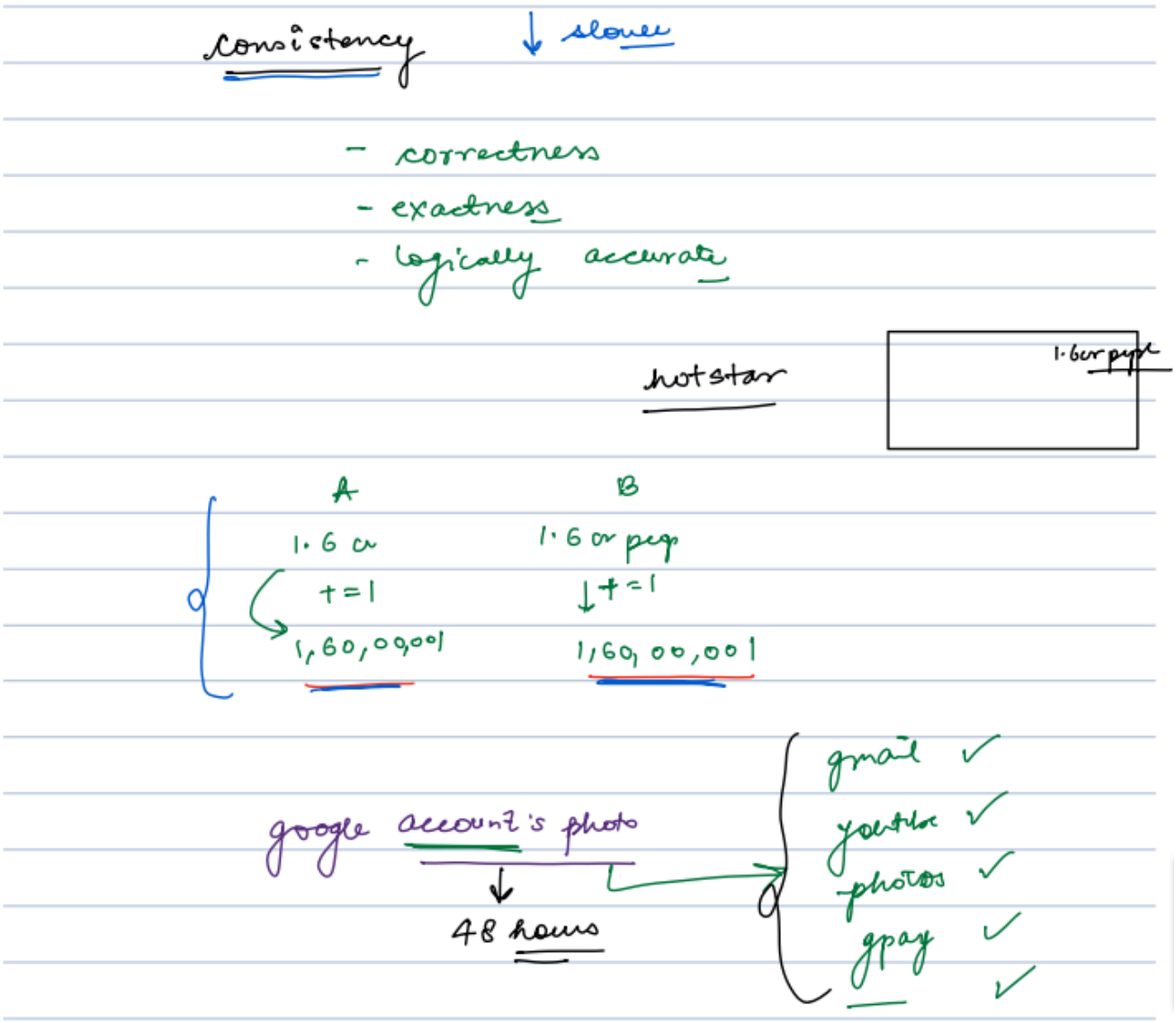
ACID Properties
Transactions has certain proprties ->



Here are the key characteristics and principles of transactions:

1. **Atomicity**: A transaction is atomic, meaning it is treated as a single, indivisible unit. All the operations within a transaction are either fully completed (committed) or fully undone (rolled back). If any part of the transaction fails, the entire transaction is rolled back to its previous state to maintain data consistency.

2. **Consistency**: Transactions are designed to preserve data consistency. They ensure that the database moves from one consistent state to another, adhering to defined rules and constraints. **Correctness, Exactness, Logically accurate**
   Some time this cossitency is not required like audience count for google photo update consistency is subjective.

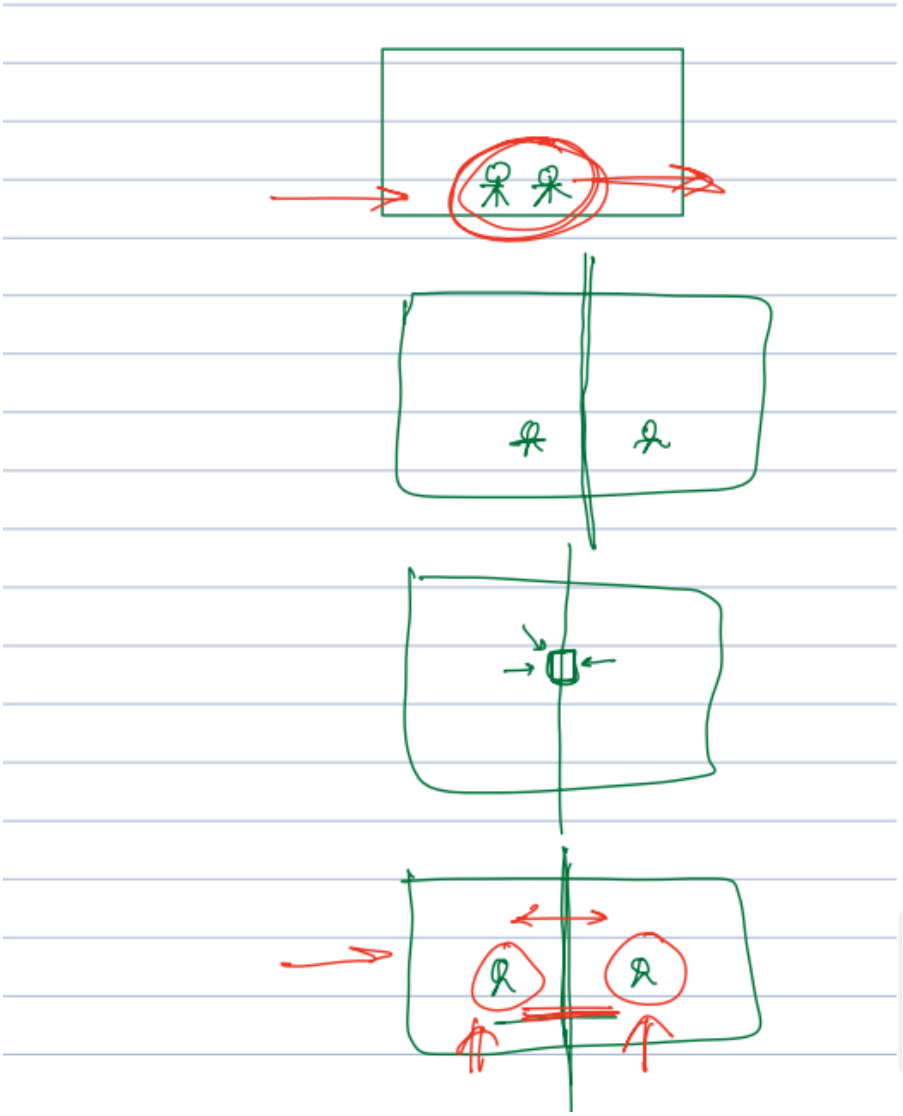


3. **Isolation**: Each transaction is isolated from other concurrent transactions, meaning the changes made by one transaction are not visible to other transactions until the first transaction is completed (committed). This isolation prevents interference and maintains data integrity. **(One transaction sholud not affect the other transaction running at the same time and on the same data)**

   Let's consider an example of two prisoners in jail. In the first image, there are no barriers between them. This lack of separation can be very harmful.

In the next image, there is a glass partition between them, offering some isolation, but they can still communicate with each other.

Moving on to the next image, there's a solid wall with a window. In this case, they can see each other but cannot communicate effectively. This provides more isolation.
In the final image, there's a solid wall without a window. Here, there is complete isolation, and the actions of one prisoner do not affect the actions of the other.



4. **Durability**: Once a transaction is committed, its changes become permanent and are saved in a way that ensures they can survive system failures, such as power outages or crashes. This durability property ensures that the changes made during the transaction are not lost.
   Once a transaction has complted the changes sholud persists

Transactions are commonly used in database management systems (DBMS) to ensure data integrity. For example, if you are transferring money between bank accounts, a transaction ensures that the debit from one account and the credit to another account occur together. If there's an issue during the transfer (e.g., the system crashes), the transaction will be rolled back to maintain the integrity of the accounts.
In addition to databases, the concept of transactions is used in various other contexts, such as in messaging systems, file systems, and distributed systems, to ensure data integrity and reliability in the face of errors and concurrent operations.

Commits and Rollbacks

- Commits: Committing a transaction means saving the data in the database, making it accessible to others. It is necessary to complete a transaction and ensure that the changes take effect.
- Rollbacks: Rolling back a transaction means reverting it to its previous state. This is done when an error occurs during a transaction, and it helps restore the database to its former condition before the transaction began.

In SQL, every query you run is considered a transaction by default, and it automatically commits. It is happen because be default auto commit is on in mysql but in the transaction autocommit is offf by default and we have to manually commit it after start transaction using below command.

start transaction;

If transaction is started then we only can read the changes but other can not see the changes till then commit is not happen.

to save changes

commit;

to reset

rollback;

Transaction Isolation Level

Transaction Isolation Level - Read Uncommitted

Transaction query

use sakila; // select database

to show autocomit is on or off —----------

show variables like **'autocommit'**;
set autocommit = true;

set autocommit = false;

// It will set session to see uncommitted change to others
set session transaction isolation level read uncommitted;

// This command for start transaction all quey will be in temp mode
start transaction;

// update

update film
set title = 'read uncommitted'
where film_id = 3;

// commit changes save to database

commit;

// Rollback change. this will reset to initial stage where we started transactions

rollback;

Transaction Isolation Level

There is 4 Isolation lavels

- **Read uncomitted**
- **Read comitted**
- **Repeatative reads**
- **Serilizable reads**

// This command show isolation label in editor

show variables like 'transaction_isolation';

default it is set to **Repetable read** It is a strongest level

Using below command we can change isolation

set session transaction isolation level read uncommitted;

after set read uncommitted other also can see your change without commit from your temp variable but it is vary dangerous because if transaction gets failed then this read called **dirty readsv** because whatever we are reading it is not available and it is rollback.

Commands

-- Person 2

```
use sakila;
set session transaction isolation level read uncommitted;
select * from film where film_id = 3;
```

-- transaction starts

```
start transaction;

update film
set title = 'read uncommitted'
where film_id = 3;

select * from film where film_id = 3;

delete from film
where film_iaddressd = 3;

delete from language
where language_id = 6;

select * from language;

rollback;

commit;
```

-- Transaction Ends—--

to show autocomit is on or off —----------

show variables like **'autocommit'**;
set autocommit = true;

-- autocommit : ON
-- transaction starts
-- changes will be finalised - written into disks only when you commit
-- until that point these will be temp changes

```
show variables like 'transaction_isolation';
```

-- Person 1

```
use sakila;

set session transaction isolation level read uncommitted;

select * from film where film_id = 3;

select * from language;

update film

set title = 'Mohit sharma phir ek baar phirse'

where film_id = 3;

show variables like 'transaction_isolation';
```

SQL: Transactions-2 - 26th Oct 2023

- **Read Committed**
- **Repeatable Reads**
- **Serializable**
- **Deadlock**

Note -> Various database systems can exhibit varying behaviors with respect to isolation levels.

1. Read Committed - In a transaction with this isolation label we can only see the committed changes. Read Committed" isolation level, transactions are isolated from each other in a way that prevents "dirty reads.

Phantom Reads (Ghost read): "Read Committed" does not prevent "phantom reads." A phantom read occurs when a transaction reads a set of rows that satisfy a search condition and, before it finishes, another transaction inserts or deletes rows that also satisfy the search condition. This can result in the first transaction seeing a different set of rows upon subsequent reads.

2. Repeatable Reads -> In this isolabel after start the transaction read will be the same till we will not commit the transaction

Consistency: In the "Repeatable Reads" isolation level, a transaction ensures that it sees a consistent snapshot of the database at the start of the transaction. This means that any data

read during the transaction will remain the same throughout the transaction, even if other transactions modify the data.

3. Repeatable Reads: The name "Repeatable Reads" implies that if a transaction reads a particular set of rows, it can re-read the same rows multiple times within the same transaction, and the data will not change. This is in contrast to lower isolation levels like "Read Committed," where data can change between reads in the same transaction.

Locking: To achieve this level of isolation, the database system may use various locking mechanisms. When a transaction reads data, it may acquire read locks to prevent other transactions from modifying the data until the reading transaction is complete.

Blocking: While "Repeatable Reads" provides strong data consistency, it can lead to increased contention and potential for blocking between transactions. If multiple transactions are trying to read and modify the same data simultaneously, they may need to wait for locks to be released, which can slow down the system.

4. Serializable -> "Serializable" is the highest level of transaction isolation in a relational database. In this isolation level, transactions are executed in a manner that provides the strictest guarantees for data consistency and integrity.

The key characteristics of the serializable isolation level are:

1. **Serializability**: Transactions at this level are executed as if they were running in isolation from other transactions, even when multiple transactions are being processed concurrently. This ensures that the final state of the database is equivalent to some serial (non-overlapping) execution of the transactions.
2. **Strongest Data Consistency**: Serializable transactions provide the strongest guarantees for data consistency. They prevent all types of anomalies(**unexpected and often undesirable behaviors or results**), including **dirty reads**, **non-repeatable reads**, and **phantom reads**. No two transactions can modify the same data simultaneously.
3. **Locking**: Achieving serializability often involves the use of various locking mechanisms. Transactions acquire locks on the data they read or modify to prevent other transactions from accessing the same data until the first transaction completes.

4. **Blocking and Potential Deadlocks**: Due to stringent locking requirements, serializable transactions can lead to blocking, where transactions wait for locks to be released. This may also increase the potential for deadlocks, which require special handling to ensure system stability.
5. **Performance Impact**: Serializable isolation comes at a cost of potentially reduced performance, as it minimizes concurrency and may lead to more waiting for locks to be released.
6. **Strictest Data Integrity**: Serializable isolation is ideal when maintaining the highest data integrity and consistency is crucial, but it may not be necessary for all types of applications. Many applications can work effectively with lower isolation levels, such as "Read Committed" or "Repeatable Reads," which provide better performance by allowing more concurrent access to the data.

Note- > When ever we will set Serializable we have to set both the session Serializable

Deadlock -> Deadlock occurs when two or more transactions are blocked, each waiting for a resource (e.g., a table, row, or page) that is currently locked by another transaction. This situation creates a circular dependency where each transaction is waiting for a resource that is held by another transaction in the cycle. As a result, no transaction can proceed, leading to a state of inactivity and effectively "locking" the affected transactions.

Deadlock Situation like
Imagine there are two separate sessions, and both are set to use the "serializable" isolation level. In the first session, a transaction is initiated and reads record 5. Similarly, in the second session, a transaction is started, and it also tries to read record 5. Now, both sessions attempt to update record 5 simultaneously.

This scenario leads to a deadlock situation because both sessions are waiting for the other to release the record. As a result, neither can proceed until one of them times out or is forcibly closed.

SQL: SQL Window function

https://dev.mysql.com/blog-archive/mysql-8-0-2-introducing-window-functions/

[In the CHEATSHEET as well]: All query elements are processed in a very strict order:

- **FROM** - the database gets the data from tables in FROM clause and if necessary, performs the JOINs,
- **WHERE** - the data are filtered with conditions specified in the WHERE clause,
- **GROUP BY** - the data are grouped by conditions specified in the WHERE clause,
- **Aggregate functions** - the aggregate functions are applied to the groups created in the GROUP BY phase,
- **HAVING** - the groups are filtered with the given condition,
- **Window functions,**
- **SELECT** - the database selects the given columns,
- **DISTINCT** - repeated values are removed,
- **UNION/INTERSECT/EXCEPT** - the database applies set operations,
- **ORDER BY** - the results are sorted,
- **OFFSET** - the first rows are skipped,
- **LIMIT/FETCH/TOP** - only the first rows are selected

OVER -> The OVER() function serves as a window function that is employed within the SELECT statement. It allows us to apply aggregate functions to row labels. Usually, when we use the MAX function, it provides us with the maximum value within a column and displays all rows associated with that maximum value. However, if we simply use "OVER()," it signifies that we are applying the function to the entire table. This operation is executed just before the final SELECT statement.

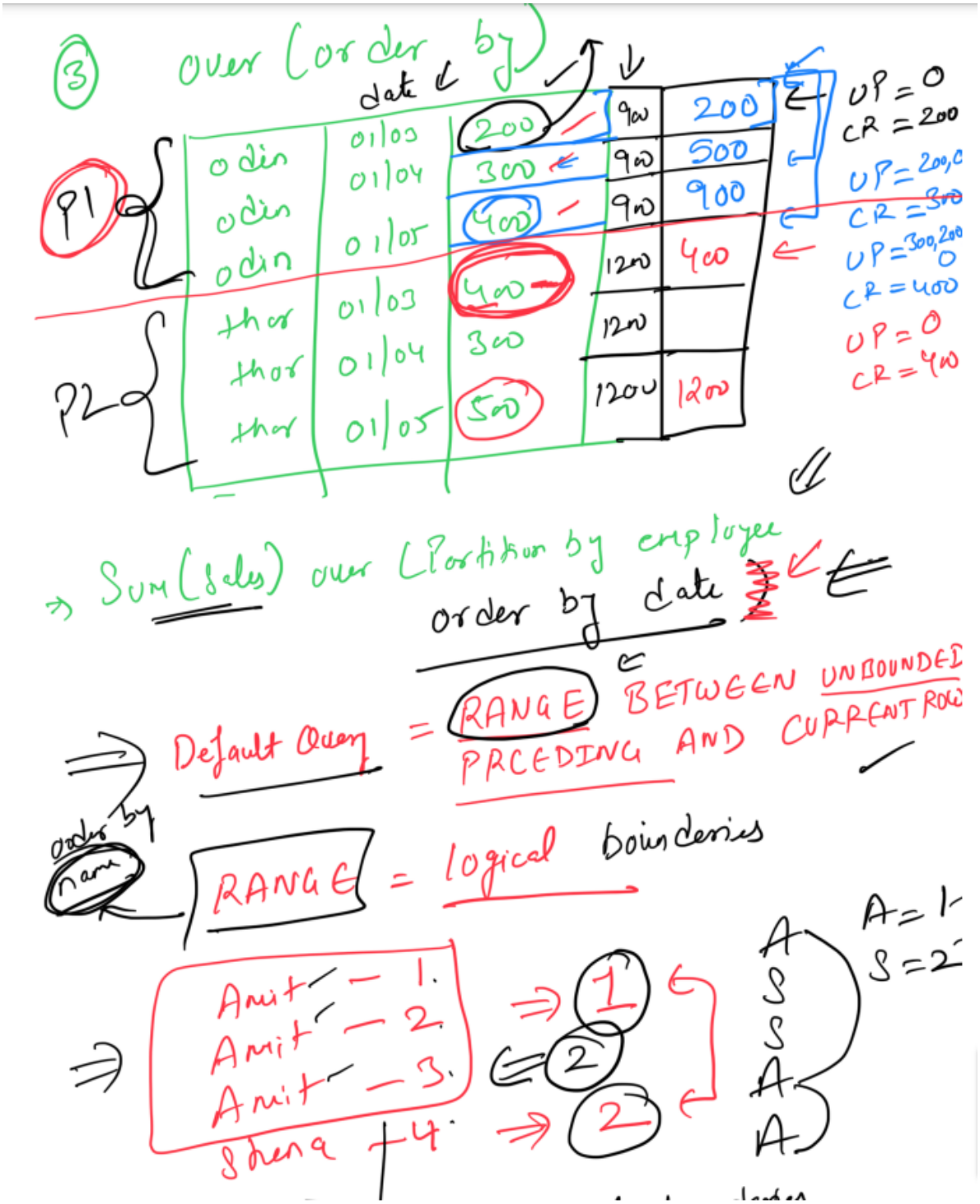
PARTITION BY -> It is used with over() function and it is optional it is behave same like Group By but It will works with row on row lavel.
SELECT over(partition by columnName)

ORDER BY -> Order by is used in over() function and it will works on row label whenever we will add orderby with in window function it will add default query

Range Between Unbounded Preceding and Current Row

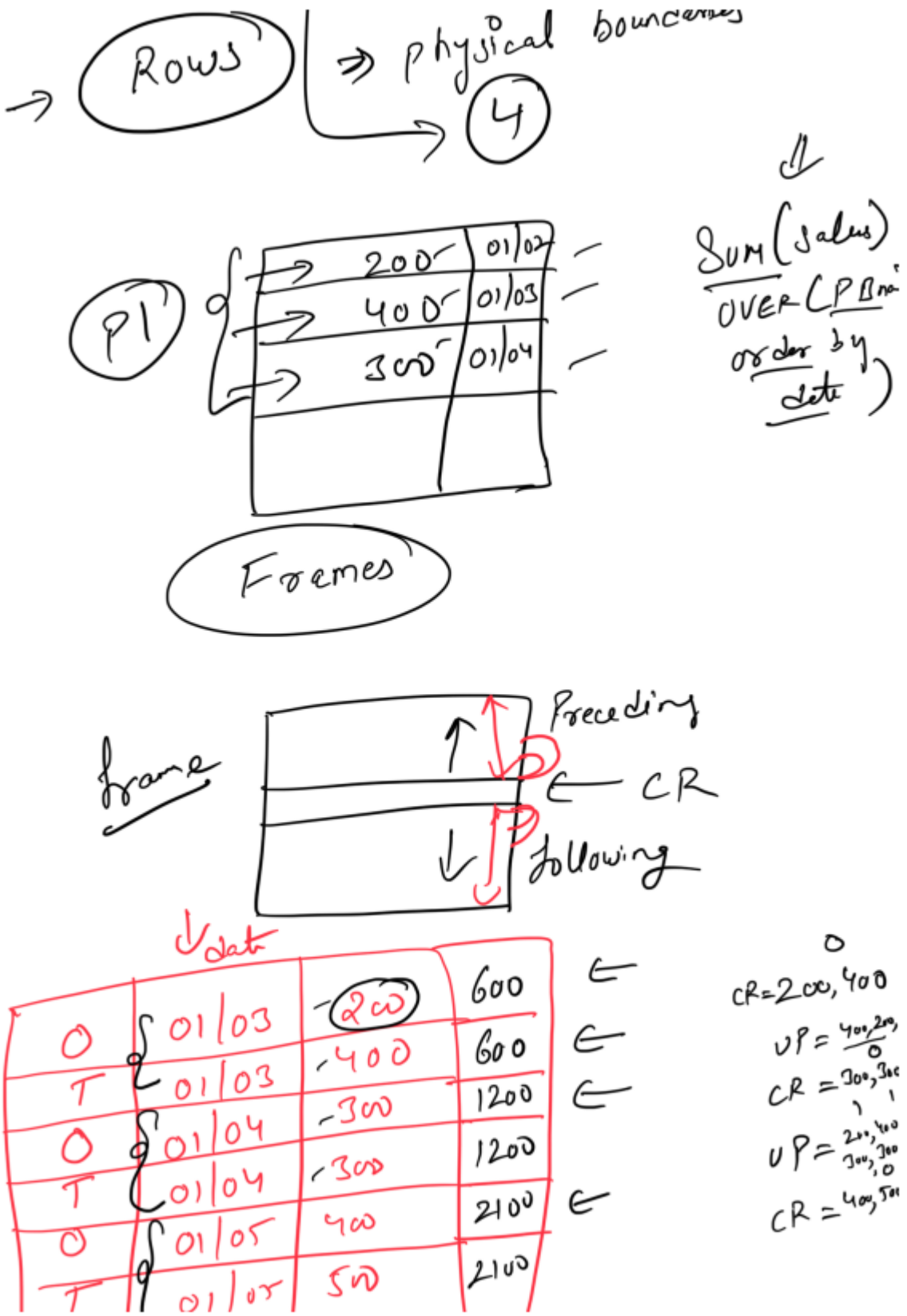
This means that it will add a frame to the calculation,

a frame comprises three parts. The first part is "unbounded preceding," which includes the previous rows, the current row, and the following "n" number of rows. Whatever aggregate function we apply will behave as if it is adding each row to the previous result until all rows within the frame have been processed. If we apply "PARTITION BY," it will work on every partition, and when the first partition ends, the counting of results will restart.



Note -> If values are duplicate of column of ORDER BY then because of RANGE it will we merge result of that rows and it will display combine result for that duplicates row like Partition

By display without order by because RANGE always see duplicates values as 1 row.

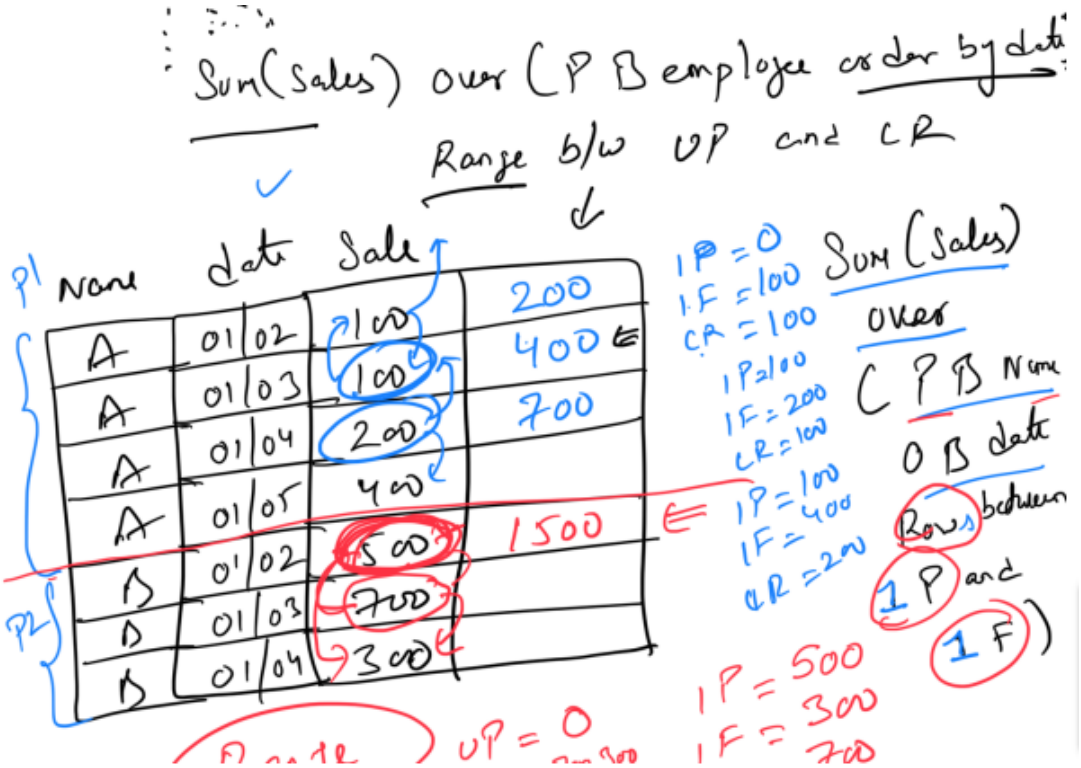


Customize default query -> RANGE BETWEEN unbounded PRECEDING AND CURRENT ROW
Keywords as follow
- **Unbounded Preceding**
- **Unbounded Following**
- **Current Row**

We have the flexibility to customize the default query according to our specific requirements. For example, if we need to calculate a cumulative sum and wish to avoid any duplicate behavior caused by the RANGE clause, we can modify the query. By changing "RANGE" to "ROWS" in the query after the "ORDER BY" clause, we alter the behavior regarding duplicate ordered records, resulting in a distinct calculation for each query.

SELECT
*,SUM(sale) OVER(order by date ROWS BETWEEN unbounded PRECEDING AND CURRENT ROW) as sales_done
FROM
sales;

Here in below example there is catch and we changed the query
from **ROWS BETWEEN** unbounded **PRECEDING AND CURRENT ROW**
to **ROWS BETWEEN 1P(PRECEDING) AND 1F(FOLLOWING)**
Now behaviour changed it will not take cumulative sum for preceding and following till end of FRAME instead it will calculate 1 preceding and 1 following row with current



Window Analytics function

ROW_NUMBER() -> assigns a unique integer value to each row within a result set. The "ROW_NUMBER" function typically operates within the context of an ordered set of rows, with the order specified by an "ORDER BY" clause. It can be a useful tool for various tasks, including ranking results, implementing pagination, and filtering data based on row position.

Order by is required to print row number
SELECT row_number() over (order by salary desc) from employee;

RANK() -> The "RANK" function is commonly used for ranking or ordering rows in a result set, typically in ascending order, and it assigns the same rank to rows with the same values. When multiple rows have the same values, they receive the same rank, and the next row is assigned a rank that skips the number of tied rows.

For example, if two rows have the same value and are assigned a rank of 2, the next row will be assigned a rank of 4, not 3. This is in contrast to the "DENSE_RANK" function, which assigns consecutive ranks to tied rows without gaps.

"RANK" is a valuable tool for tasks such as ranking products by sales, determining the top N results, or identifying ties in a dataset. It operates within the context of an ordered set of rows, with the order specified by an "ORDER BY" clause.

DANSE RANK -> assigns a unique rank or position to each row within a result set, based on the values in one or more specified columns. Similar to the "RANK" function, the "DENSE_RANK" function is commonly used for ranking or ordering rows in a result set. However, unlike "RANK," the "DENSE_RANK" function assigns consecutive ranks to rows with the same values, without any gaps.

When multiple rows have the same values, they receive the same rank, and the next row is assigned the next consecutive rank, without skipping any ranks for tied rows.

"DENSE_RANK" is often used in scenarios where you want to rank items without gaps in the ranking order. For example, if two rows have the same value and are assigned a rank of 2, the next row will also be assigned a rank of 3, ensuring that there are no skipped ranks.

Like "RANK," the "DENSE_RANK" function operates within the context of an ordered set of rows, with the order specified by an "ORDER BY" clause. It is a valuable tool for tasks such as ranking products by sales, determining the top N results, or identifying ties in a dataset.

Note -> if there are no duplicates then row number, rank and dense rank lead to similar result

Require this output where we have to remove if employee has 1 unique salary because we have to create a team of 2 people who has same salary

employee_id	name	salary	team_id
1	Andrew	5000	1
2	Erin	5000	1
4	Jim	8000	2
5	Oscar	8000	2

```
SELECT
        employee_id,
        name,
        salary,
        dense_rank() OVER(ORDER BY salary) as team_id
FROM employees
where salary NOT IN (
select salary from employees
group by salary
having count(*) = 1;
)
ORDER BY team_id, employee_id;
```

IFNULL() -> It will return first value if it is not null elase return given value like below

```
SELECT std_name, IFNULL(batch_id, ‘NA’)
FROM student s
LEFT JOIN batches b
ON s.batch_id = b.id
```

IF(condition, value-of-true, value-of-false)

Multiple IFS

```
IF(performance =1, 'fired',
        IF(performance = 2, 'need improvement',
                IF(performance = 3, 'Good work',
                        IF(performance = 4, ‘super’,‘Somthing has gone wrong’
                        )
                )
        )
) as performance_score
```

CASE -> If we have multiple if elase then it looks ugly and vary hard to read to solve this we can use CASE it is like Switch Case in programming.

```
SELECT name,
        CASE
                WHEN condition THEN output
                WHEN condition THEN output
                WHEN condition THEN output
                ELSE -> (like default in switch case)
        END as alias_name
FROM employees
```

```
SELECT name,
        CASE
                WHEN performance = 1 THEN 'fired'
                WHEN performance = 2 THEN need improvement'
                WHEN performance = 3 THEN 'Good work'
                WHEN performance = 4 THEN 'super'
                ELSE  'Somthing has gone wrong'   -> (like default in switch case)
        END as alias_name
FROM employees
```

COALESCE -> It is a alternative of case for multiple IFNULL checks it isis vary help full.
Print the batch name for each student. If a batch name is not available, print the father's name. If neither the batch name nor the father's name is available, print "NA."

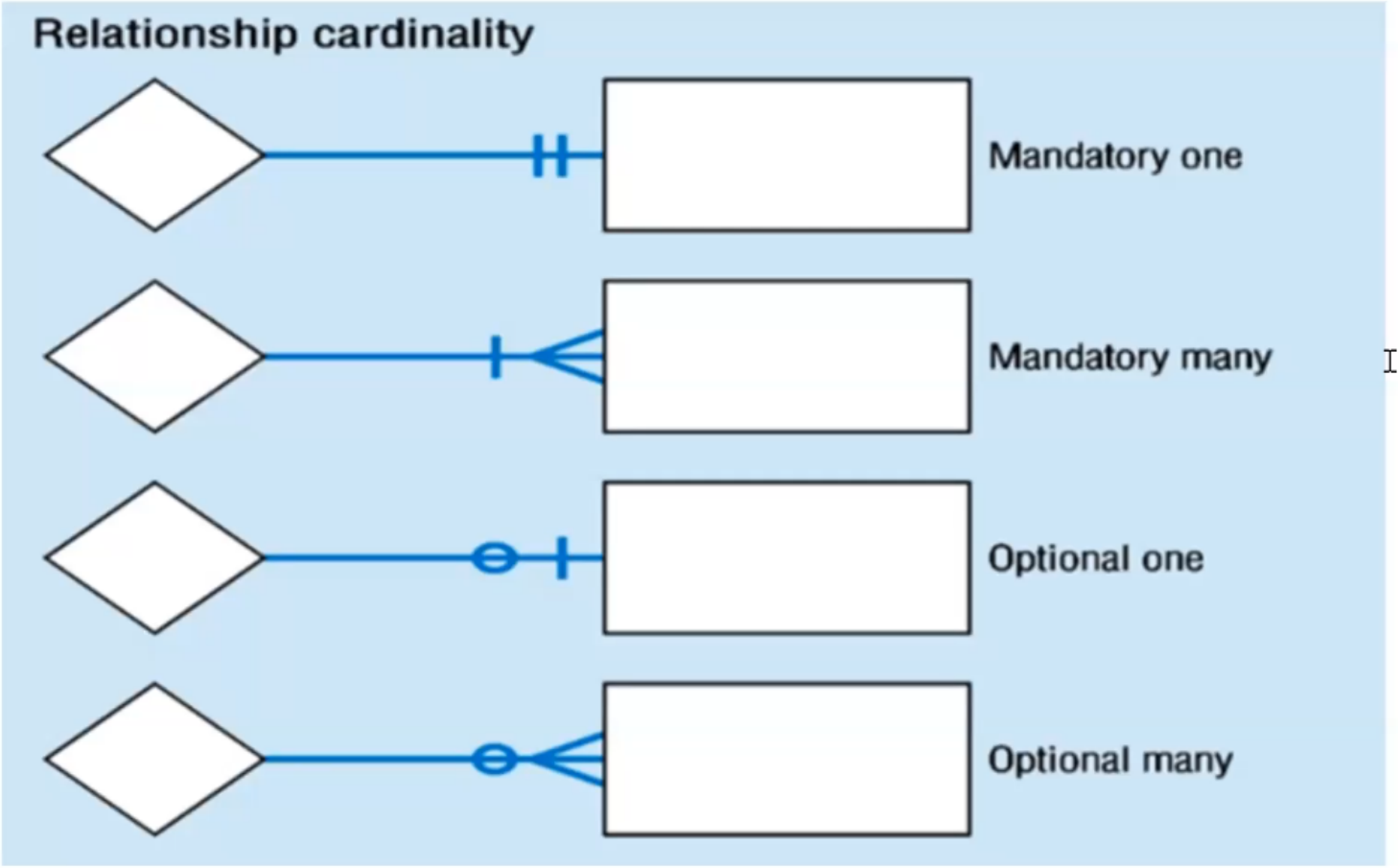
COALESCE(val,val2,val3,val4)
It will print first non null value as per the order in the function like above it will check first value if it null then it will check val2 and so on.

Note ->

- If last is also null it will print null value
- Idealy it is recomonded to have the last value i coleasce as a constant value like any string like 'NA'

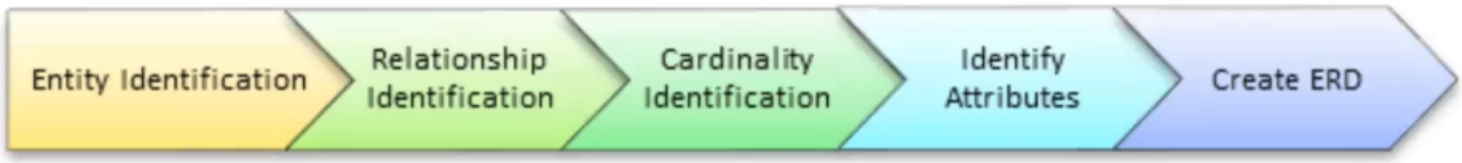
```
SELECT std_name, COALESCE(b.batch_name, s.father_name, 'NA')
FROM student s
LEFT JOIN batches b
ON s.batch_id = b.id
```

SQL: ER digram



ER- Diagram is a visual representation of data that describe how data is related to each other.

- **Rectangles:** This symbol represent entity types
- **Ellipses :** Symbol represent attributes
- **Diamonds:** This symbol represents relationship types
- **Lines:** It links attributes to entity types and entity types with other relationship types
- **Primary key:** attributes are underlined
- **Double Ellipses:** Represent multi-valued attributes



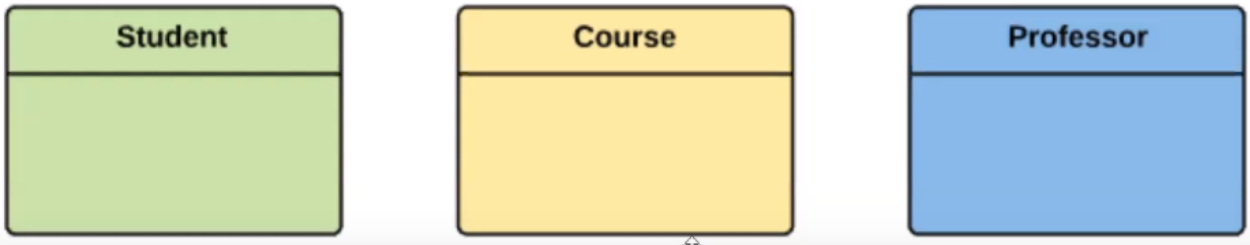
In a university, a Student enrolls in Courses. A student must be assigned to ar least one or more courses. Each course is taught by a single professor.

To maintain instruction quality, a professor can deliver only one course

Step 1) Entity Identification

We have three entities

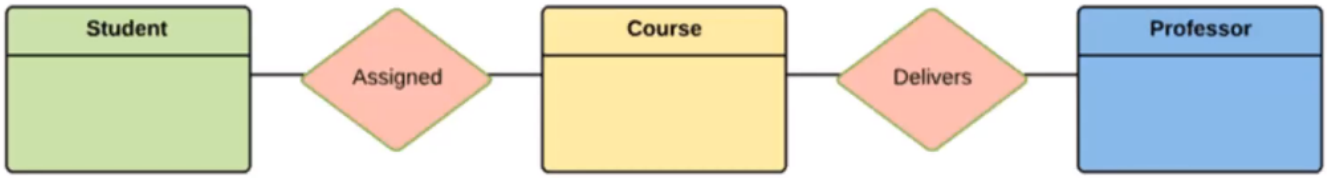
- Student
- Course
- Professor



Step 2) Relationship Identification

We have the following two relationships

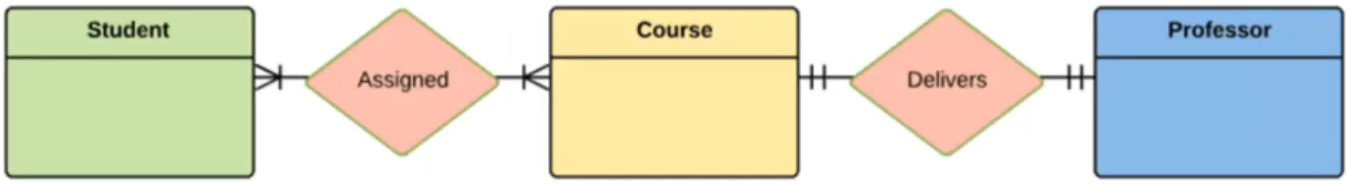
- The student is **assigned** a course
- Professor **delivers** a course



Cardinality Identification

For them problem statement we know that,

- A student can be assigned **multiple** courses
- A Professor can deliver only **one** course



- Create table for each entity.
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key.

Mapping Relationship

A relationship is an association among entities.