# SQL Learning Notes

**Database:** Database is a structured collection of data that is stored, managed, and accessed electronically. It is designed to efficiently store, retrieve, update, and manage data in a way that ensures integrity, consistency, and security**.**

Database management system

Your APP

(Amazon)

Database

**Database Management System:** A **Database Management System (DBMS)** is **software that interacts with databases** to perform tasks such as **storing, retrieving, updating, and managing data** efficiently while ensuring **data integrity, security, and concurrency control**.

**MySQL, PostgreSQL, SQLite, etc., are not databases themselves, but rather Database Management Systems (DBMS)** that provide the tools and functionalities to interact with, manage, and manipulate data stored in databases.

## Creating Databases and Tables

* SHOW DATABASES;
* It will show the databases which is available in the system.
* CREATE DATABASE yethishwar\_rao;
* It will create a database.
* DROP DATABASE yethishwar\_rao;
* It will delete the database permanently.

NOTE: using snake\_case is a good practice naming databases, tables, column\_names etc…

* USE database\_name;
* Switches the active database to the specified one.
* Just use the above query to switch to the working database and work on it.
* SELECT DATABASE();
* Displays the currently selected database.

Inside a Database there will be Tables this is the area where we actually focus on and manipulate the things. A table consists of rows and columns, almost all SQL flavoured languages deals with tables, other than MangoDB and some more.

### Data Types

We have many numerical, character and date time data types, among all some are frequently used like VARCHAR(n), CHAR(n), INT, FLOAT, BIGINT etc…

To create tables use the below query

* CREATE TABLE methodist\_collage

( students VARCHAR(100),

roll\_no INT,

grade CHAR(10)

);

#To see the structure of the table

* DESCRIBE collages\_in\_hyderabad;

-- OR --

* SHOW COLUMNS FROM gurunanak;

-- OR --

* SHOW TABLES;

-- To delete a table permanamtly

* DROP TABLE gurunanak;

-- Insert values into the Table

* INSERT INTO malla\_reddy(students, roll\_no, grade)
* VALUES('Siddhartha\_rao', 90, 'S++');
* INSERT INTO malla\_reddy(students, roll\_no, grade)
* VALUES('Sai Kiran Rao',89,'S++');

We can also interchange the values like this👇… the order does not matter but make sure to tally the column names and values.

* INSERT INTO malla\_reddy(roll\_no, grade, students)
* VALUES(89,'S++','Sai Kiran Rao');

-- Insert multiple rows at a time with single insert command

* INSERT INTO malla\_reddy(students,roll\_no)
* VALUES('Kiara',33),
* ('preethi',24),
* ('Sushil',12),
* ('Rushi',22)
* ;

-- To see the values in a table

* SELECT \* FROM malla\_reddy;

-- By specifying NOT NULL while creating a table... The table dooesn't support any null values

* CREATE TABLE population(

state VARCHAR(20) NOT NULL,

population INT NOT NULL);

If we try to insert values like this, 👇 it will throw error

* INSERT INTO population(state)
* VALUES('TELANGANA')

ERROR: ERROR 1364 (HY000): Field 'age' doesn't have a default value

So to overcome this error we need to specify a default value which is suitable for that data type like below… DEFAULT value **only applies when a column is entirely omitted** from the INSERT statement, **not when NULL is explicitly inserted**. 👇

* create table people2(
* -> name varchar(20) DEFAULT 'No value here',
* -> age int default 0);

If we insert null values also it just show the default value instead of ‘NULL’ when we select the table.

If you want MySQL to **always** replace NULL with the default value, you **must** use NOT NULL in the table👇

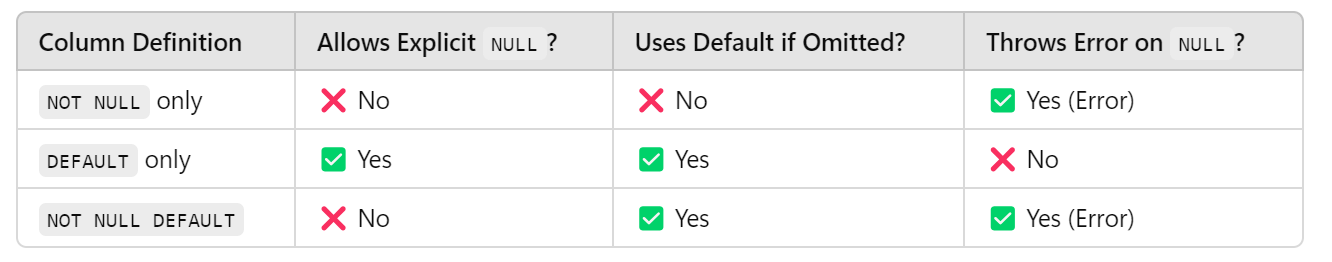
* create table people2(
* -> name varchar(20) NOT NULL DEFAULT 'No value here',
* -> age int NOT NULL default 0);

If you explicitly inserts a null value like this👇 it will take NULL value unless you mention NO NULL wile creating

* insert into people2(name, age)
* -> values(NULL,NULL);

-- If we want double codes in the middle of the name just mention back slash like thisit will ignore the next symbol and proceeds with no ERROR.👇

* INSERT INTO people2(name)
* VALUES('Yethishwar\'s Family');



### #Primary Key

A **Primary Key** is a unique identifier for each record in a database table. It ensures that no two rows have the same value in the primary key column and that the value cannot be **NULL**. This helps maintain data integrity and allows efficient data retrieval.

This query ensures no null values allowed and two different rows will contain unique values.

* create table unique\_people(
* aadhar bigint not null primary key,
* name varchar(100),
* age int);

You can also specify it at the end like this

* PRIMARY KEY (aadhar)

If you try to insert identical values like this it will throw an error like below

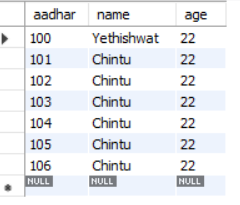
* insert into unique\_people(aadhar,name,age)
* values(1,'Yethishwar',21),
* (1,'Yethishwar',21);

ERROR 1062 (23000): Duplicate entry '1' for key 'unique\_people.PRIMARY'

### #AUTO\_INCREMENT

he **AUTO\_INCREMENT** constraint is used in SQL (specifically in MySQL and some other databases) to automatically generate a unique, incrementing value for a **Primary Key** column.

- If we you want the primary key values to be auto incremented please specify auto\_increment liek this

* CREATE TABLE people4(
* aadhar BIGINT auto\_increment,
* name VARCHAR(100),
* age INT,
* PRIMARY KEY(aadhar)
* );
* INSERT INTO people4(aadhar,name,age)
* VALUES(100,'Yethishwat',22);
* select \* from people4;
* INSERT INTO people4(name,age)
* VALUES('Chintu', 22),
* ('Chintu', 22),
* ('Chintu', 22),
* ('Chintu', 22),
* ('Chintu', 22),
* ('Chintu', 22);

MySQL picks the next available number after the highest existing value in the column, not the lowest missing number.

### FINAL ASSIGNMENT

* create table employee(
* id int primary key auto\_increment,
* last\_name varchar(100) not null,
* first\_name varchar(100) not null,
* middle\_name varchar(100),
* age int not null,
* current\_status varchar(100) not null default 'Employed');

## CRUD(Create, Read, Update, Delete) BASICS

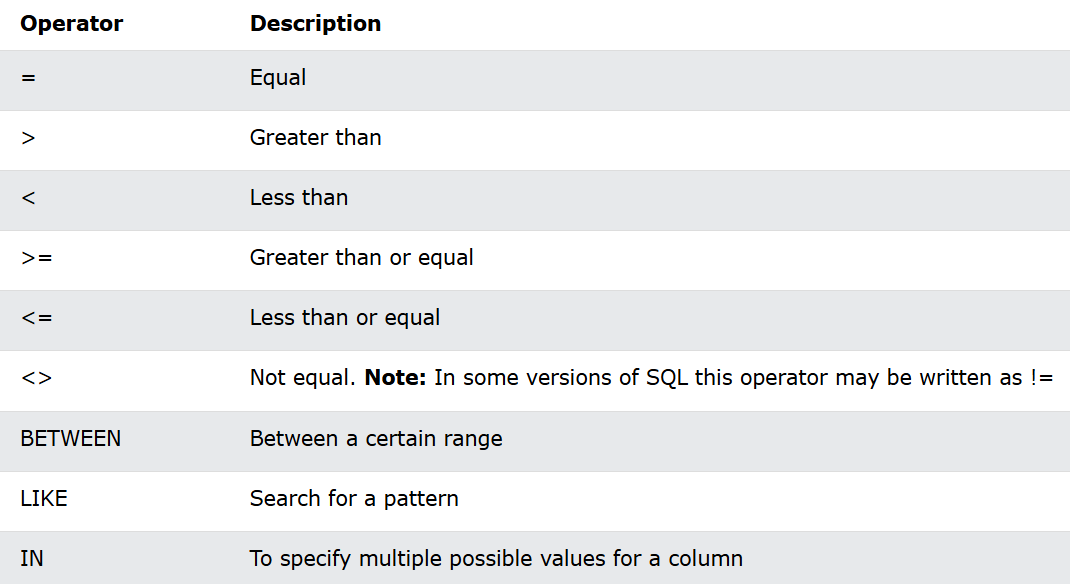
### READ

In **MySQL**, a **read operation** refers to any query that retrieves data from the database without modifying it.

Lets take a cats table and work on it

* SELECT \* FROM cats;
* SELECT name FROM cats;
* SELECT name, breed FROM cats;
* SELECT breed, name, age, age FROM cats;

### WHERE



* SELECT name FROM cats
* WHERE breed = 'Tabby';
* SELECT name,age FROM cats
* WHERE age <= 10;
* SELECT \* FROM cats
* WHERE breed != 'Tabby';
* SELECT \* FROM cats
* WHERE age BETWEEN 5 and 10;
* SELECT name FROM cats
* WHERE breed LIKE 'T%';
* SELECT \* FROM cats
* WHERE breed IN ('Tabby','Sphynx');

### ALIASES

* Aliases are used to give a table, or a column in a table, a temporary name.
* Aliases are often used to make column names more readable.
* An alias only exists for the duration of that query.
* An alias is created with the AS keyword.

If the query is big and a column name is used multiple times, using an **alias** can make the query **shorter, more readable, and easier to maintain**.

* SELECT cat\_id AS id, breed AS b,name AS n FROM cats;

### UPDATE

The UPDATE statement is used to modify the existing records in a table.

Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

* UPDATE cats
* SET name = 'Preeti'
* WHERE age >13;
* UPDATE cats
* SET name = ‘MISTY’, breed = ‘DHUNNA’
* WHERE age = 13;

**A Good Rule of Thumb** **👉 Always SELECT the data before performing an UPDATE or DELETE.because once you done update you cannot undo it.**

### DELETE

The DELETE statement is used to delete existing records in a table by applying a where condition.

* DELETE FROM cats
* WHERE age > 10;
* DELETE FROM cats; #Total values will be deleted but remains structure.

## String Operations

NOTE:- source filename.sql -> executes the commands present in that file.

### CONCAT(parameters)

Used to combine text and give custom column name

* SELECT CONCAT('Hi', 'Yethishwar') as Lovely;
* SELECT CONCAT(author\_fname, ' ', author\_lname) FROM books;
* SELECT CONCAT(author\_fname, ' ', author\_lname) AS author\_fullname FROM books;

### CONCAT\_WS(‘separator’, parameters…)

It just works like concat but it combines with a given separator

* SELECT CONCAT\_WS(' ~ ',author\_fname, author\_lname, pages, released\_year) as loverboy FROM books;



### SUBSTRING() OR SUBSTR()

* SELECT SUBSTRING(author\_fname, 2) FROM books;
* Extract a substring from a string (start from the position 2 onwards till at the last)
* SELECT SUBSTRING(author\_fname, 1, 3) FROM books;
* Extract a substring from a string (start at position 1, extract 3 characters)
* SELECT SUBSTRING(author\_fname, -1),author\_fname FROM books;
* Extract the last letter of the string
* SELECT SUBSTRING('Yethishwar', -4,2);
* Extract a substring from a string (start from the end, at position -4, extract 2 characters):

### COMBINING CONCAT(SUBSTRING(), ‘ ’)

The query starts executing from nested function and moving to outer function and filtering the stuff.

* SELECT
* CONCAT(SUBSTRING(author\_fname, 1, 5),
* ' ',
* SUBSTRING(author\_lname, 1, 3),
* ' ',
* '!!!!!') AS combined\_string
* FROM
* books;
* SELECT
* SUBSTRING(author\_fname, 1, 1), SUBSTRING(author\_lname, 1, 1)
* FROM
* books;
* SELECT
* CONCAT(SUBSTRING(author\_fname, 1, 1),
* ' . ',
* SUBSTRING(author\_lname, 1, 1))
* FROM
* books;

### REPLACE()

The REPLACE() function replaces all occurrences of a substring within a string, with a new substring.

* SELECT REPLACE('Ye&\*th&\*is&\*hw&\*ar', '&\*', '');
* SELECT REPLACE(title, ' ', '-') from books;
* REPLACE(string, substring, new\_string)
* SELECT
* CONCAT(SUBSTRING(author\_fname, 1, 5),
* ' ',
* SUBSTRING(author\_lname, 1, 3),
* ' ',
* '!!!!!', ' ',
* REPLACE(title, ' ', '-')
* ) AS combined\_string
* FROM
* books;

### REVERSE()

The REVERSE() function reverses a string and returns the result.

* SELECT REVERSE(‘yethishwar’);
* SELECT REVERSE(author\_fname) from books;

### CHAR\_LENGHTH()

It prints the number of charaters of specified string

* SELECT CHAR\_LENGTH('Yethishwar'); -> 10
* SELECT CHAR\_LENGTH('Yethishwar') AS length , title FROM books;

### LENGTH()

It prints the number of bites of specfied string

* SELECT LENGTH('字字典'); -> 9

### UPPER() & LOWER() or UCASE() & LCASE()

Converts into upper case and lower case

* SELECT UPPER(title) FROM books;
* SELECT LOWER(title) FROM books;
* SELECT CONCAT('I LOVE ', UPPER(title), '!!!') FROM books;

### INSERT()

The INSERT() function inserts a string within a string at the specified position and for a certain number of characters.

* SELECT INSERT('HELLO WORLD',6,5, ' YETHISHWAR ');

### LEFT() & RIGHT()

Return the left and right most charaters with the specified length

* SELECT LEFT('Yethishwar',5);
* SELECT RIGHT('Yethishwar rao', 3);

### REPEAT()

Repeats the string upto specified count times

* SELECT REPEAT('Yethishwar Rao ' , 5);
* SELECT REPEAT(author\_fname, 3) FROM books;

### TRIM()

Trims specified charaters at the beginning or ending.

* SELECT TRIM(' yETHISHWAR ');

**yETHISHWAR**

* SELECT TRIM(LEADING '\*' FROM '\*\*\*\*\*\*\*\*\*\*yETHISHWAR\*\*\*\*\*\*\*\*');

**yETHISHWAR\*\*\*\*\*\*\*\***

* SELECT TRIM(BOTH '\*' FROM '\*\*\*\*\*\*\*\*\*\*yETHISHWAR\*\*\*\*\*\*\*\*');

**yETHISHWAR**

## Refining Selections

### DISTINCT

The SELECT DISTINCT statement is used to return only distinct (different) values.

* SELECT DISTINCT title FROM books;
* SELECT DISTINCT author\_fname FROM books;
* SELECT DISTINCT(CONCAT(author\_fname, ' ', author\_lname)) FROM books;
* SELECT DISTINCT author\_fname, author\_lname FROM books;

### ORDER BY

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

* SELECT \* FROM books
* ORDER BY pages DESC;

For string values the ORDER BY keyword will order alphabetically

* SELECT \* FROM books
* ORDER BY title ASC, pages DESC;

we can also use number to select column by specified its location starting from 1.

* SELECT title, author\_fname, author\_lname,pages FROM books ORDER BY 4;
* SELECT author\_fname, released\_year, book\_id FROM books
* ORDER BY author\_fname, released\_year, book\_id;
* SELECT CONCAT(author\_fname, author\_lname) AS author FROM books ORDER BY author;

### LIMIT 1

If we want top 10 rows or from specific location to these many rows we can do it using LIMIT

* SELECT book\_id, author\_fname, author\_lname, released\_year FROM books ORDER BY released\_year limit 5;
* SELECT book\_id, author\_fname, author\_lname, released\_year FROM books ORDER BY released\_year limit 2,5;

### LIKE

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

* SELECT book\_id, author\_fname, author\_lname FROM books WHERE author\_fname LIKE '%Da%' AND author\_lname LIKE 'E%';
* SELECT author\_fname FROM books WHERE author\_fname LIKE '\_\_\_\_\_';
* SELECT author\_fname FROM books WHERE author\_fname LIKE '\_a\_';

IF we want to search ‘%, \_’ these items in the data we need to use backslash(\) to tell query to this is to be search

* SELECT title FROM books WHERE title LIKE '%\%%';
* SELECT title FROM books WHERE title LIKE '%\\_%';

## Aggregate Functions

### COUNT()

* SELECT COUNT(\*) FROM books;
* SELECT COUNT(DISTINCT author\_fname) FROM books;
* SELECT COUNT(DISTINCT released\_year) FROM books;
* SELECT COUNT(title) FROM books
* WHERE title LIKE '%THE%';

### GROUP BY

* SELECT \* FROM books;
* SELECT author\_lname,COUNT(\*) AS books\_written FROM books
* GROUP BY author\_lname ORDER BY books\_written;

All non-aggregated columns in SELECT must be in GROUP BY

* SELECT released\_year, COUNT(book\_id) AS total\_books
* FROM books
* GROUP BY released\_year
* ORDER BY total\_books;

### MIN()/MAX()

* SELECT MIN(released\_year) FROM books;
* SELECT MAX(pages) FROM books;
* SELECT MAX(author\_lname), MIN(author\_lname) FROM books;

### SUBQUERY

---- TO find the book which has highest pages ---------

* SELECT title FROM books
* WHERE pages = 634;
* SELECT title, pages FROM books
* WHERE pages = (SELECT MAX(pages) FROM books);
* INSERT INTO books(title, pages)
* VALUES('Drag me to the Hell', 634);
* SELECT title, released\_year FROM books
* WHERE released\_year = (SELECT MIN(released\_year) FROM books);
* SELECT title FROM books
* ORDER BY pages DESC LIMIT 4;

### GROUP BY MULTIPLE COLUMNS

* SELECT \* FROM books;
* SELECT CONCAT(author\_fname, ' ', author\_lname) AS author\_fullname, COUNT(pages) FROM books GROUP BY author\_fullname;
* SELECT author\_fname, author\_lname, COUNT(pages) FROM books GROUP BY author\_lname, author\_fname;

### GROUP BY WITH MIN/MAX

-------- Find the year each author published their first book --------

* SELECT
* CONCAT(author\_fname, ' ', author\_lname) AS author,
* MIN(released\_year) AS earliest\_release,
* MAX(released\_year) AS recent\_released,
* COUNT(\*) AS books\_written
* FROM
* books
* GROUP BY author;

### SUM()

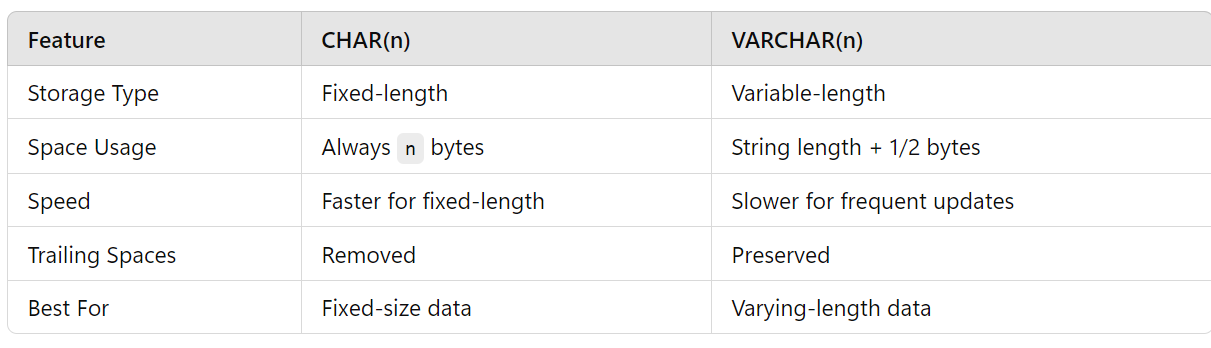
* SELECT
* released\_year, SUM(pages) AS page\_count
* FROM
* books
* GROUP BY released\_year
* ORDER BY page\_count DESC;
* SELECT
* author\_lname, SUM(pages) AS page\_count
* FROM
* books
* GROUP BY author\_lname
* ORDER BY page\_count;
* SELECT SUM(author\_lname) FROM books;

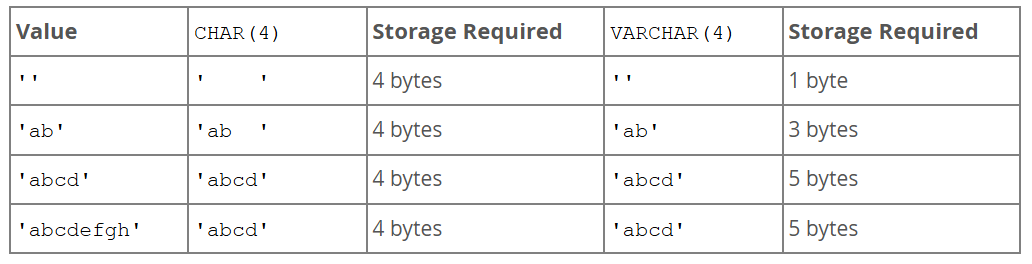
### AVG()

* SELECT AVG(stock\_quantity) AS stock, author\_lname FROM books GROUP BY released\_year ORDER BY stock;

## Data Types

### CHAR VS VARCHAR



* CHAR(6) –> ‘A B \_ \_ \_ \_ ’ ‘A B C \_ \_ \_’ ‘A B C D E \_ ’ -> 6 bytes
* VARCHAR(5) -> ‘abc’ -> 3 Bytes + 1 for length (meta data)
* When char values are retieved , trailing spaces are removed.
* Use CHAR for fixed lengths like pincodes, phone numbers etc. otherwise use VARCHAR

### TINYINT, SMALLINT, MEDIUMINT, INT BIGINT

* CREATE TABLE parents(children INT UNSIGNED);

By mentioning UNSIGNED in the query tells that it should only accepts positive integers where storage valules will be doubled because of we excluded negative integres

Above integer data types only varies in their storage which is very important aspect to optimize it.

### DECIMAL(maximum\_length, length\_after\_point)



MySQL **truncates** or **rounds** it to **23.323 (23.32332323 has more than 3 decimal places.)**

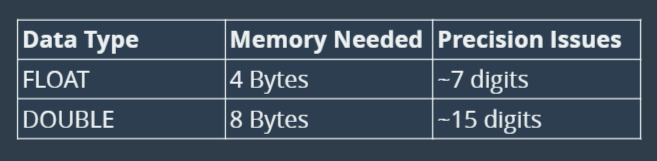
233.3 is **valid** and will be stored as **233.300** (auto-filling three decimal places).

223.0 is **valid** and will be stored as **223.000**.

When you define a column as **DECIMAL(5,3)**, it means:

* **Total 5 digits** (including both whole and decimal parts).
* **Last 3 digits** are reserved for the **decimal fraction**.
* The **first 2 digits** (before the decimal point) are for the **integer part**.

### FlOAT & DOUBLE



* Use FLOAT when you need fast, approximate values with less memory.
* Use DOUBLE when you need more precision at the cost of extra storage.
* Use DECIMAL when exact values are required (like currency calculations).

### DATE TIME DATE-TIME



* CREATE TABLE people(
* name VARCHAR(100),
* birthdate DATE,
* birthtime TIME,
* birthdatetime DATETIME
* );

* DESC people;
* INSERT INTO people(name, birthdate, birthtime, birthdatetime)
* VALUES('Kiran', '1989-05-22', '01:00:12', '1989-05-22 1:00:12'),
* ('Chintu', '1998-05-22', '09:10:02', '1998-05-22 19:10:02'),
* ('Sai', '2017-11-22', '11:10:12', '2017-11-22 11:10:12'),
* ('Siddhu', '1810-05-02', '01:10:12', '1810-05-22 01:10:12');
* SELECT \* FROM people;

### CURRENT TIME, CURRENT DATE, TIMESTAMP

* SELECT CURDATE(), CURTIME(), NOW();
* SELECT CURRENT\_DATE(), CURRENT\_TIME(), CURRENT\_TIMESTAMP();
* Above 👆 2 queries are same but the only difference is function names
* INSERT INTO people(name, birthdate, birthtime, birthdatetime)

VALUES('Yethishwar Rao', CURDATE(), CURTIME(), NOW());

* We can also use these 👆 functions to insert values

### IMPORTANT FUNCTIONS ON DATE, TIME & DATETIME

* SELECT SYSDATE(), NOW();
* SELECT DATE\_ADD(CURDATE(), INTERVAL 10 DAY);
* SELECT ADDDATE('2025-03-08', 4);
* SELECT DATE\_SUB('2025-03-08', INTERVAL 7 YEAR);
* SELECT SUBDATE('2025-03-08', 1);

**Time Functions**

* SELECT HOUR('12:30:45');
* SELECT MINUTE('12:30:45');
* SELECT SECOND('12:30:45');
* SELECT TIME\_TO\_SEC('01:30:00'); -- Output: 5400 seconds
* SELECT SEC\_TO\_TIME(5400); -- Output: 01:30:00

**TimeStamp Functions**

* SELECT TIMESTAMPADD(YEAR, 5, '2025-03-08 10:30:00');
* SELECT TIMESTAMPADD(MONTH, 5, '2025-03-08 10:30:00');
* SELECT TIMESTAMPADD(DAY, 5, '2025-03-08 10:30:00');
* SELECT TIMESTAMPADD(HOUR, 5, '2025-03-08 10:30:00');
* SELECT TIMESTAMPADD(MINUTE, 5, '2025-03-08 10:30:00');
* SELECT TIMESTAMPADD(SECOND, 5, '2025-03-08 10:30:00');

**Date extracting and formatting**

* SELECT EXTRACT(YEAR FROM '2025-03-08');
* SELECT EXTRACT(DAY FROM '2025-03-08');
* SELECT DAYOFMONTH('2025-03-08');
* SELECT DAYOFWEEK('2025-03-08');
* SELECT DAYOFYEAR('2025-12-08');
* SELECT MONTH('2025-03-08');
* SELECT MONTHNAME('2025-03-08');
* SELECT YEAR('2025-03-08');

**Difference**

* SELECT TIMEDIFF('12:30:00', '10:00:00'); -- Output: 02:30:00
* SELECT DATEDIFF('2025-03-08', '2025-03-01');
* SELECT TIMESTAMPDIFF(DAY, '2029-11-01', '2025-03-08');
* SELECT
* birthdatetime,
* MONTH(birthdatetime),
* DAY(birthdatetime),
* HOUR(birthdatetime),
* MINUTE(birthdatetime),
* YEAR(birthdatetime),
* SECOND(birthdatetime)
* FROM people;
* SELECT
* birthtime,
* HOUR(birthtime),
* MINUTE(birthtime)
* FROM people;

**Date Format**

* SELECT DATE\_FORMAT('2004-05-22', '%W, %M %D, %Y');
* SELECT birthdate, DATE\_FORMAT(birthdate, '%a %b %D') FROM people;

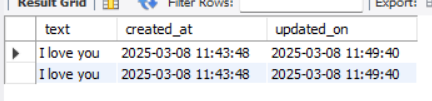
* SELECT birthdatetime, DATE\_FORMAT(birthdatetime, '%H:%i') FROM people;
* SELECT birthdatetime, DATE\_FORMAT(birthdatetime, 'BORN ON: %r') FROM people;

**Date Formatcodes**

| **Code** | **Description** | **Example Output** |
| --- | --- | --- |
| %Y | Year (4 digits) | 2025 |
| %y | Year (2 digits) | 25 |
| %M | Full month name | March |
| %m | Month (2 digits) | 03 |
| %D | Day with suffix | 8th |
| %d | Day (2 digits) | 08 |
| %W | Full weekday name | Saturday |
| %w | Weekday index (0=Sunday, 6=Saturday) | 6 |

### DEFAULT & ON UPDATE

* CREATE TABLE captions(
* text VARCHAR(100),
* created\_at TIMESTAMP default CURRENT\_TIMESTAMP
* );
* INSERT INTO captions(text)
* VALUES( 'yETHISHWATR'),
* ('Madavaram Chintu');
* SELECT \* FROM captions;
* CREATE TABLE captions2(
* text VARCHAR(100),
* created\_at TIMESTAMP default CURRENT\_TIMESTAMP,
* updated\_on TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP
* );



After updating text column the update \_on column wil be altered and replaced with the current updated date and time in case of timestamp.

* INSERT INTO captions2(text)
* VALUES( 'hI THERE I AM CHINTU'),
* (Preethi CHINRU Chintu');
* SELECT \* FROM captions2;
* UPDATE captions2 SET text='I love you';

## Comparision & Logical Operators

### NOT EQUAL(!=) – COMPARISION

* It is used for excluding single value like selecting all rows excluding year 2017
  + SELECT title FROM books WHERE released\_year != 2017;

### NOT LIKE – COMPARISION

The NOT LIKE operator is used in SQL when you want to **exclude** rows based on a **pattern match** using wildcards (% and \_). It is different from != and NOT IN, which check for exact values instead of patterns.

* + SELECT \* FROM books WHERE released\_year NOT LIKE '2%';
  + SELECT author\_lname, author\_fname FROM books WHERE author\_fname LIKE 'D\_\_';
  + SELECT title FROM books WHERE title NOT LIKE '% %';

### GREATER THAN(>) – COMPARISION

It is used to compare one vaue with another value and filter out the data

* + SELECT author\_fname, author\_lname, released\_year FROM books WHERE released\_year > 2015;
  + 1 > NULL -> NULL

### LESS THAN(<) – COMPARISION

* + SELECT \* FROM books WHERE released\_year < 2000;
  + SELECT author\_fname, released\_year FROM books WHERE released\_year < 2000 ORDER BY released\_year;
  + SELECT author\_fname, released\_year FROM books WHERE released\_year <= 1989;

### AND(&&) – LOGICAL

The **AND** operator in SQL is a **logical operator** used to combine multiple conditions in a WHERE clause. It ensures that **all conditions must be true** for a row to be included in the result.

* The number of copies sold must be greater than 50 but less than 500.
* SELECT COUNT(\*) FROM books WHERE stock\_quantity > 50 AND stock\_quantity < 500;
* SELECT 1 > -4 AND 1 = 1; -> 1
* SELECT title FROM books WHERE CHAR\_LENGTH(title) > 20 AND pages > 300;

### OR(||) - LOGICAL

The **OR** operator in SQL is a logical operator used to combine multiple conditions in a WHERE clause. It ensures that at **least one** of the conditions must be true for a row to be included in the result.

* SELECT title, released\_year FROM books
* WHERE released\_year > 2010 OR char\_length(title) > 20;
* SELECT 2 > 10 OR 1 < 0;

### BETWEEN

We can also use <= some value AND >= some value bit it is little bit complicated so we use BETWEEN

* SELECT title FROM books WHERE released\_year BETWEEN 2004 AND 2015;
* SELECT title FROM books WHERE released\_year NOT BETWEEN 2004 AND 2015;

### CAST – To convert datatype

The CAST function in SQL is primarily used to convert one data type into another.

* CAST(expression AS target\_data\_type)
* SELECT birthtime FROM people WHERE CAST(birthtime AS TIME) > '-1:00:00';
* SELECT birthdate FROM people WHERE birthdate < '1990-11-01';
* ------------ OR ---------------
* SELECT birthdate FROM people WHERE YEAR(birthdate) < '2004';
* SELECT birthtime FROM people WHERE birthtime < '1:12:12';
* SELECT birthtime FROM people WHERE HOUR(birthtime) > '5';

When we comparing dates and time in sql generally we comvert it into the required datatype using cast or else if you want to compare it with specific parts, we use YEAR(), HOUR(), MONTH() like this to compare in more effective way

The **IN operator** in MySQL is used to check if a value **matches** any value in a specified list. It is commonly used in **WHERE** clauses to filter records efficiently.

* SELECT author\_lname FROM books WHERE author\_lname IN ('Gaiman', 'Harris', 'Eggers');
* SELECT author\_lname FROM books WHERE author\_lname NOT IN ('Gaiman', 'Harris', 'Eggers');
* ----- find all books not published in even released\_years ----
* SELECT \* FROM books WHERE released\_year NOT IN ('2000', '2002', '2004', '2006','2008', '2010', '2012');
* SELECT \* FROM books WHERE released\_year % 2 != 0;

### CASE

* CASE column\_name
* WHEN value1 THEN result1
* WHEN value2 THEN result2
* ELSE default\_result
* END

It acts like a if-else statemrnt in programming

* SELECT released\_year,
* CASE
* WHEN released\_year >= 2000 THEN 'It\'s 21st century'
* WHEN released\_year <= 2000 THEN '1900\'s Stuff'
* ELSE 'NO STUFF HERE!!!'
* END AS 'CATEGORIZATION'
* FROM books;

It will create a table with first column released\_year and second column CATEGORIZATION and based on the conditions the values will be categorized

* SELECT stock\_quantity,
* CASE
* WHEN stock\_quantity BETWEEN 0 AND 50 THEN '\*'
* WHEN stock\_quantity BETWEEN 50 AND 100 THEN '\*\*'
* ELSE '\*\*\*'
* END AS 'STOCK'
* FROM books;

As compared to above code below one is more effective way because the mySQL checks the conditions line by line

* SELECT stock\_quantity,
* CASE
* WHEN stock\_quantity <= 50 THEN '\*'
* WHEN stock\_quantity <=100 THEN '\*\*'
* ELSE '\*\*\*'
* END AS 'STOCK'
* FROM books;

### IS NULL

In SQL, **NULL is not equal to anything**, even another NULL.

You **cannot use = to compare NULL values**. Instead, you must use IS NULL.

## Constraints & Alter Table

Primary keys are not null and must by unique and every table must contain only 1 primary key

UNIQUE can be applied for multiple columns

* CREATE TABLE companies(
* supplier\_id INT AUTO\_INCREMENT,
* name VARCHAR(100) NOT NULL,
* phone VARCHAR(100) NOT NULL UNIQUE,
* address VARCHAR(100) NOT NULL,
* PRIMARY KEY(supplier\_id)
* );
* desc companies;
* INSERT INTO companies(name, phone, address)
* VALUES( 'Yethishwar', '9879798797', 'Miryalaguda');
* INSERT INTO companies(name, phone,address) ->This throws error
* VALUES( 'Yethishwar', '9879798797', 'Miryalaguda');

### CHECK

* CREATE TABLE parties(
* name VARCHAR(100) NOT NULL,
* age INT CHECK (age > 18)
* );
* INSERT INTO parties(name, age)
* VALUES('yethishwar', 20);
* INSERT INTO parties(name, age)
* VALUES('Chintu', 12);

This will get an error because condition not satisfied

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 17 | Error Code: 3819. Check constraint 'parties\_chk\_1' is violated. | 0.000 sec |

**PALINDROME**

* CREATE table palindrome(
* palin\_check VARCHAR(100) NOT NULL CHECK(REVERSE(palin\_check) = palin\_check));
* INSERT INTO palindrome(palin\_check)
* VALUES('Yethishwar');

This above code will get an error 👇

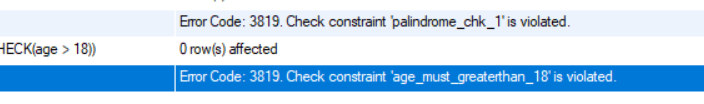


* SELECT \* FROM palindrome;

### NAMED CONSTRAINTS

* CREATE TABLE constraints(
* age INT,
* CONSTRAINT age\_must\_greaterthan\_18 CHECK(age > 18));
* INSERT INTO constraints(age)
* VALUES(12);

The condition is violated and the use defined error will be shown



### MULTIPLE COLUMN CONSTTAINTS

We can combine multiple combinations multiple columns and apply constraints on it by using check or any other constraints like UNIQUE

* CREATE TABLE multiple(
* name VARCHAR(100) NOT NULL,
* address VARCHAR(100) NOT NULL,
* CONSTRAINT name\_address\_combo\_must\_be\_unique UNIQUE(name, address));
* INSERT INTO multiple(name, address)
* VALUES('yethishwar', 'MLG');
* INSERT INTO multiple(name, address)
* VALUES('yethishwar', 'Karimnagar');
* INSERT INTO multiple(name, address)
* VALUES('Chintu', 'Karimnagar');
* INSERT INTO multiple(name, address)
* VALUES('Chintu', 'Karimnagar');

Example:- 2

* CREATE TABLE sales(
* purchase\_price INT NOT NULL,
* sale\_price INT NOT NULL,
* CONSTRAINT sellingP\_is\_greater\_purchseP CHECK (sale\_price > purchase\_price));
* INSERT INTO sales(purchase\_price, sale\_price)
* VALUES(100, 200);
* INSERT INTO sales(purchase\_price, sale\_price)
* VALUES(300, 200);

### ALTER TABLE

* **INSERT** → Adds new rows (data) into a table.
* **UPDATE** → Modifies existing rows (data) inside a table.
* **ALTER** → Modifies the structure (columns, data types, constraints) of the table.

The ALTER TABLE statement in SQL is used to **modify** an existing table structure. This can include:

* Adding new columns
* Modifying existing columns
* Dropping columns
* Adding constraints
* Dropping constraints
* Renaming tables and columns

|  |  |
| --- | --- |
| **Add a Column** | * ALTER TABLE employees ADD age INT; |

|  |  |
| --- | --- |
| **Modify a Column** | * ALTER TABLE employees MODIFY age BIGINT; |

|  |  |
| --- | --- |
| **Rename a Column** | * ALTER TABLE employees RENAME COLUMN age TO employee\_age; |

|  |  |
| --- | --- |
| **Drop a Column** | * ALTER TABLE employees DROP COLUMN age; |

|  |  |
| --- | --- |
| **Add a Constraint** | * ALTER TABLE employees ADD PRIMARY KEY (employee\_id); |

|  |  |
| --- | --- |
| **Drop a Constraint** | * ALTER TABLE employees DROP CONSTRAINT unique\_email; |

|  |  |
| --- | --- |
| **Rename a Table** | * ALTER TABLE employees RENAME TO staff; |

|  |  |
| --- | --- |
| **Set Default Value** | * ALTER TABLE employees ALTER COLUMN status SET DEFAULT 'Active'; |

|  |  |
| --- | --- |
| **Remove Default Value** | * ALTER TABLE employees ALTER COLUMN status DROP DEFAULT; |

|  |  |
| --- | --- |
| **Make Column NOT NULL** | * ALTER TABLE employees MODIFY email VARCHAR(255) NOT NULL; |

----- add column to a table -----

* ALTER TABLE parties
* ADD fname VARCHAR(100);

----- Adding multiple columns to a table -----

* ALTER TABLE parties
* ADD (firstname VARCHAR(100), lastname VARCHAR(100));

----- modify existing column data type -----

* ALTER TABLE parties
* MODIFY age SMALLINT;

----- Chainging column size -----

* ALTER TABLE parties
* MODIFY name VARCHAR(200);

----- renaming column name -----

* ALTER TABLE parties
* RENAME COLUMN name TO parrty\_name;

----- Dropping single column -----

* ALTER TABLE parties
* DROP COLUMN age;

----- Dropping multiple columns -----

* ALTER TABLE parties
* DROP COLUMN age, DROP COLUMN parrty\_name;

----- Adding a primary key constraint -----

* ALTER TABLE employees
* ADD PRIMARY KEY (employee\_id);

----- Adding unique constraint -----

* ALTER TABLE employees
* ADD CONSTRAINT unique\_email UNIQUE (email);

----- Dropping a constraint -----

* ALTER TABLE employees
* DROP CONSTRAINT unique\_email;

----- Renaming a table -----

* ALTER TABLE employees
* RENAME TO staff;

----- Setting a default value for column -----

* ALTER TABLE employees
* ALTER COLUMN status SET DEFAULT 'Active';

----- Removing a default value -----

* ALTER TABLE employees
* ALTER COLUMN status DROP DEFAULT;

----- Changing a column to NOT NULL -----

* ALTER TABLE employees
* MODIFY email VARCHAR(255) NOT NULL;

---- using CHANGE keyword -----

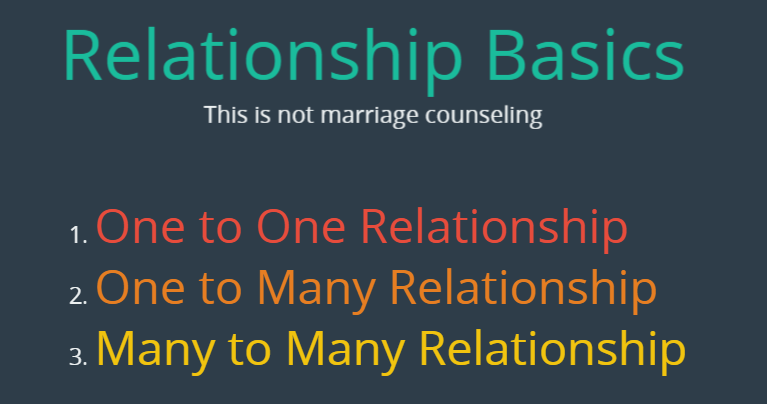
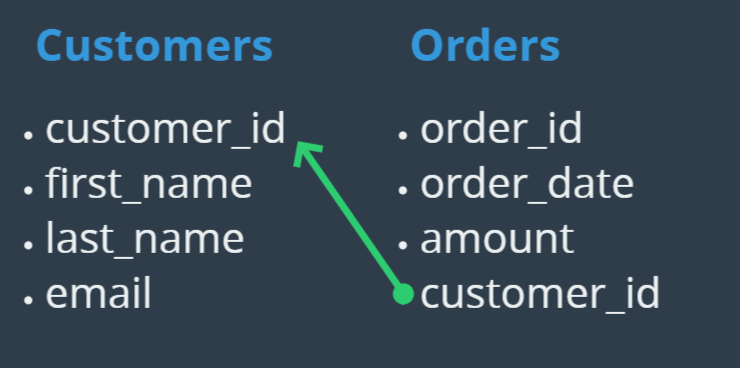
* ALTER TABLE employees
* CHANGE buisiness biz VARCHAR(100)

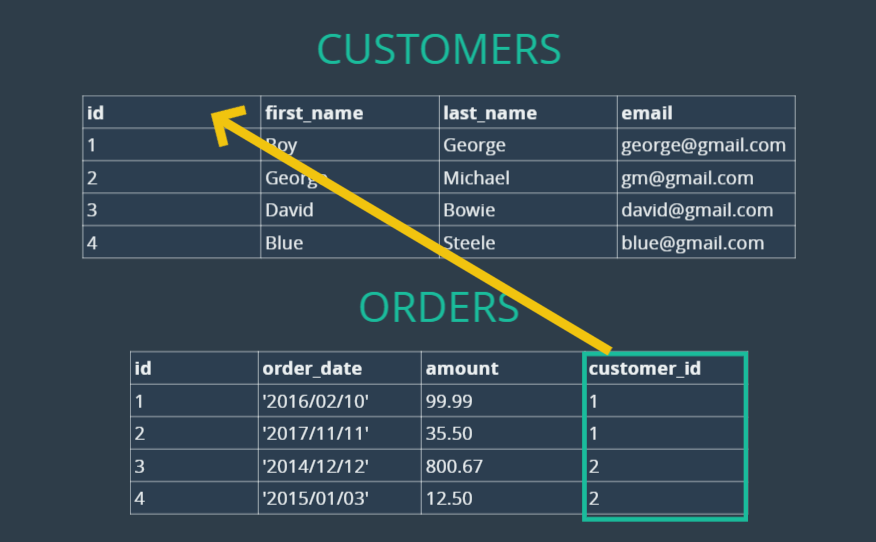
This is used to chnage the datatype and rename at a time

## One to Many & Joins

A **one-to-many (1:M) relationship** is a type of database relationship where **one record in a table (Parent Table) is associated with multiple records in another table (Child Table).** However, each record in the child table relates to **only one record** in the parent table.

* A **customer** can place **multiple orders**.
* But **each order belongs to only one customer**.



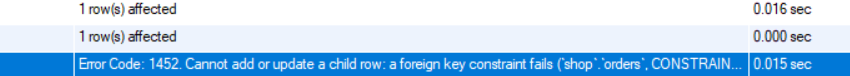


### FOREIGN KEY

A **foreign key** is a **constraint** used in a relational database to establish a relationship between two tables. It ensures **referential integrity**, meaning the values in the foreign key column of one table must correspond to valid values in the primary key column of another table.

* ALTER TABLE orders
* ADD CONSTRAINT FOREIGN KEY(customer\_id) REFERENCES customers(id);
* INSERT INTO orders(order\_date, amount, customer\_id)
* VALUES('2025-12-22', 143.23, 221);

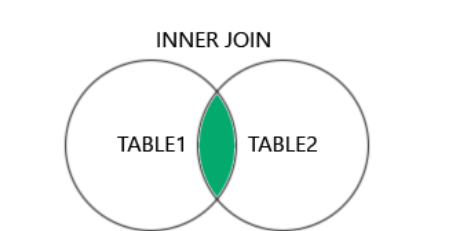
This will throw an error because the customer\_id column is a foreign key which links to the customers table primary key, so it allows the id’s which is present in the primary key in customers table



* INSERT INTO orders(order\_date, amount, customer\_id)
* VALUES('2025-12-22', 143.23, 1);

Cross Join conbines all possible rows from each table it is like a cross product which is useless in many cases.

### INNER JOIN/ JOIN



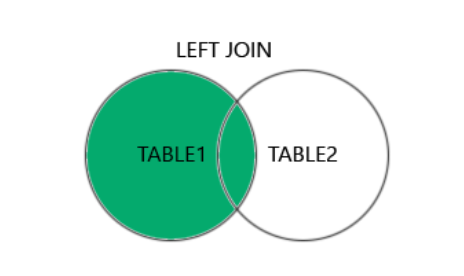
* **JOIN without specifying INNER** defaults to an **INNER JOIN** in SQL.
* **INNER JOIN is very useful in a one-to-many (1:M) relationship**, like the **Customers** and **Orders** scenario.
* **INNER JOIN retrieves only matching rows**: If a customer has no orders, they are **excluded** from the result.
* The INNER JOIN looks for **matching values** in the common column and returns only those rows.
* If there is **no match**, the row is excluded from the result
* SELECT \* FROM customers
* JOIN orders
* ON customers.id = orders.customer\_id;

OR

* SELECT \* FROM customers
* INNER JOIN orders
* ON customers.id = orders.customer\_id;

### LEFT JOIN/ LEFT OUTER JOIN

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records (if any) from the right table (table2) The LEFT JOIN keyword returns all records from the left table (table1), and the matching records (if any) from the right table (table2)



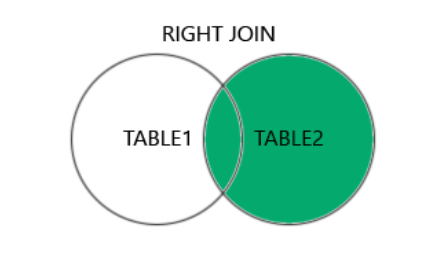
The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

* SELECT first\_name, last\_name, amount, order\_date FROM customers
* LEFT OUTER JOIN orders
* ON customers.id = orders.customer\_id;

Left Join with Group BY and using IFNULL() Func

* SELECT first\_name, last\_name, IFNULL(SUM(amount), 0) FROM customers
* LEFT JOIN orders ON customers.id = orders.customer\_id
* GROUP BY first\_name, last\_name;

### RIGHT JOIN/ RIGHT OUTER JOIN



The RIGHT JOIN keyword returns all records from the left table (table2), and the matching records (if any) from the right table (table1)

In our case where we are joining records which is a foreign key so there is no NULL values exist, in another case if It is not a foreign key then the unmatched recored will be filled with NULL values.

* SELECT \* FROM customers
* RIGHT OUTER JOIN orders
* ON customers.id = orders.customer\_id;

### ON DELETE CASCADE

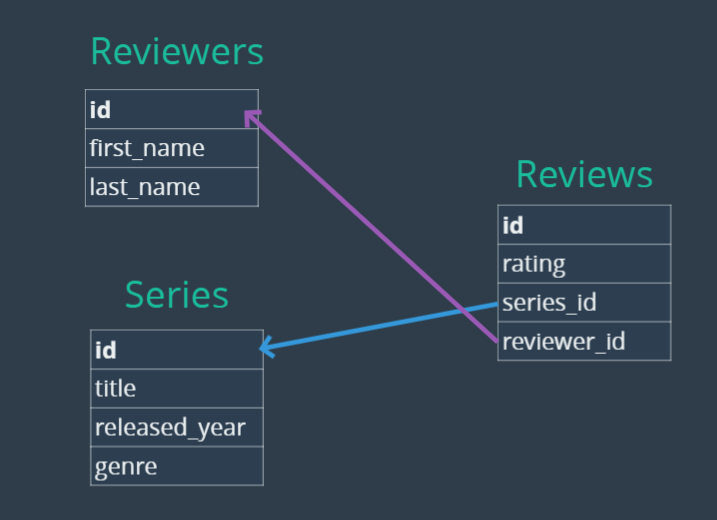
* FOREIGN KEY(customer\_id) REFERENCES customerS(id) ON DELETE CASCADE);

The ON DELETE CASCADE constraint in a foreign key ensures that when a referenced record in the parent table (customers) is deleted, all related records in the child table (which references customer\_id) will also be automatically deleted.

### Many to Many

A **many-to-many (M:N) relationship** occurs when multiple rows in one table can be related to multiple rows in another table. Since relational databases **do not support direct many-to-many relationships**, we use a **junction table (bridge table)** to break it into two **one-to-many (1:M) relationships**.

Example: A **student** can enroll in **multiple courses**, and a **course** can have multiple students.



Try out TV review challenges to test your understanding about the topic

NOTE: If we have a case where only one condition is checked means if the condition is true do this or else this, we have a more flexible way to do this instead of using CASE WHEN THEN ELSE…

* IF(COUNT(rating) > 0, 'ACTIVE', 'INACTIVE') AS STATUS

## Views, Mode and More…

### VIEWS

Views in MySQL are **virtual tables** that store the result of a **SELECT statement**. They do not contain actual data but help in:

✅ **Simplifying complex queries** – Instead of writing long SQL queries repeatedly, you can create a view and use it like a table.  
✅ **Restricting access** – You can allow users to see only specific columns or rows without exposing the entire table.  
✅ **Improving maintainability** – If the underlying tables change, you only need to update the view instead of modifying multiple queries.

* CREATE VIEW full\_reviews AS
* SELECT first\_name, last\_name, title, genre, released\_year, rating FROM series
* JOIN reviews ON reviews.series\_id = series.id
* JOIN reviewers ON reviewers.id = reviews.reviewer\_id;
* SELECT genre, ROUND(AVG(rating), 2) AS Avg\_rating FROM full\_reviews
* GROUP BY genre;

Instead of writing query again and again we can just use the view for querying

**✅ When Can We Update a View?**

You can update a view **if it is based on a single table and follows these rules:**  
✔ The view should not contain **aggregations** (SUM(), AVG(), etc.).  
✔ The view should not use **DISTINCT**.  
✔ The view should not include **JOINs** (in most cases).  
✔ The view should not have **GROUP BY** or **HAVING**.  
✔ The view should include **all primary key columns** of the original table (to ensure uniqueness).

**❌ When Can’t We Update a View?**

A view **CANNOT** be updated if it contains:  
❌ **Aggregations** (SUM(), COUNT(), AVG())  
❌ **JOINs** (in most cases)  
❌ **DISTINCT**  
❌ **GROUP BY / HAVING**  
❌ **Subqueries in SELECT**

NOTE: We can insert into a view which is a simple view that doesnot contains above things

* CREATE OR REPLACE VIEW full\_reviews AS
* SELECT \* FROM series ORDER BY released\_year;
* CREATE OR REPLACE VIEW full\_reviews AS
* SELECT \* FROM series ORDER BY released\_year DESC;
* SELECT \* FROM full\_reviews;
* ALTER VIEW full\_reviews AS

✅ CREATE OR REPLACE VIEW → **Drops and recreates the view** (if it exists) or **creates a new one** (if it doesn’t).  
✅ ALTER VIEW → **Modifies an existing view without dropping it** (keeps permissions & dependencies intact).

### HAVING

The **HAVING** clause is used to filter records **after aggregation** (i.e., after GROUP BY). It is similar to WHERE, but the key difference is:

✅ **WHERE** filters **individual rows** **before** aggregation.  
✅ **HAVING** filters **grouped records** **after** aggregation.

* SELECT first\_name, last\_name, COUNT(rating) FROM full\_reviews GROUP BY first\_name, last\_name ;
* SELECT first\_name, last\_name, COUNT(rating) AS count FROM full\_reviews GROUP BY first\_name, last\_name HAVING count > 5

### WITH ROLLUP

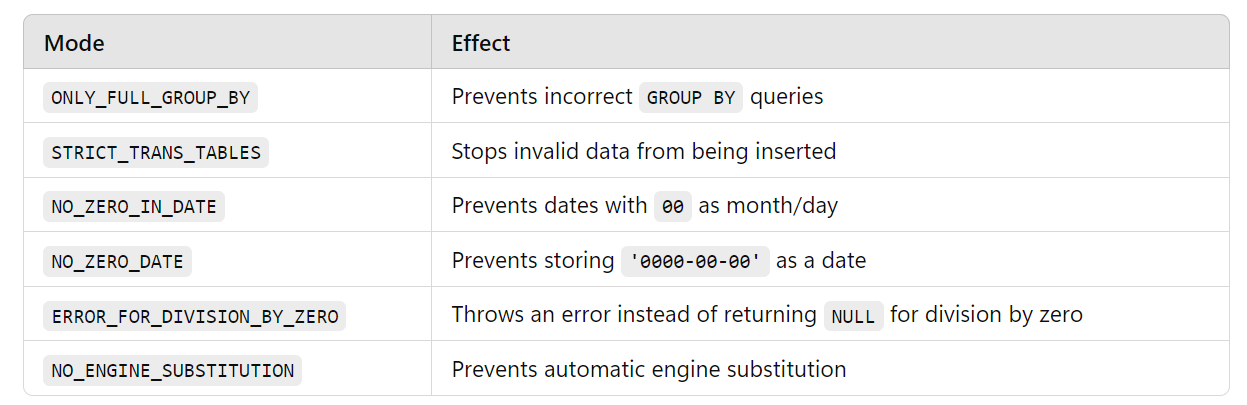
ROLLUP is an **extension of the GROUP BY clause** in MySQL that automatically **calculates subtotals and grand totals** for grouped data. It helps in generating hierarchical summary reports without writing multiple queries.

* SELECT title, MIN(rating) FROM full\_reviews GROUP BY title WITH ROLLUP;
* SELECT title, COUNT(rating) FROM full\_reviews GROUP BY title WITH ROLLUP;
* SELECT title, MAX(rating) FROM full\_reviews GROUP BY title WITH ROLLUP;
* SELECT title, SUM(rating) FROM full\_reviews GROUP BY title WITH ROLLUP;

### SQL MODES

sql\_mode in MySQL is a **system setting** that controls how MySQL behaves when handling queries. It affects how MySQL processes **syntax, data validation, and error handling**.

Think of sql\_mode as a **set of rules** that tell MySQL how strict or flexible it should be when executing SQL queries.



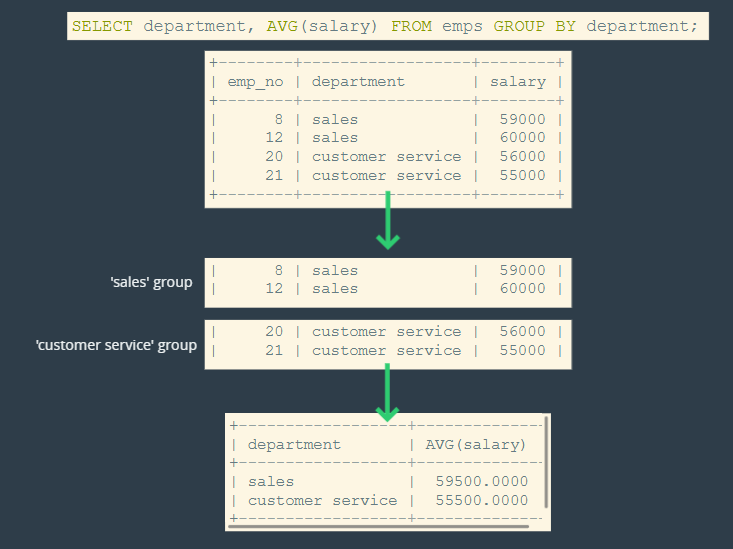
When you **remove STRICT\_TRANS\_TABLES from sql\_mode**, MySQL **stops enforcing strict type validation**. That’s why inserting a string into an INT column results in **MySQL converting it to 1 instead of throwing an error.**

When ONLY\_FULL\_GROUP\_BY is **enabled**, MySQL requires that **every column** in the SELECT list **must either**:

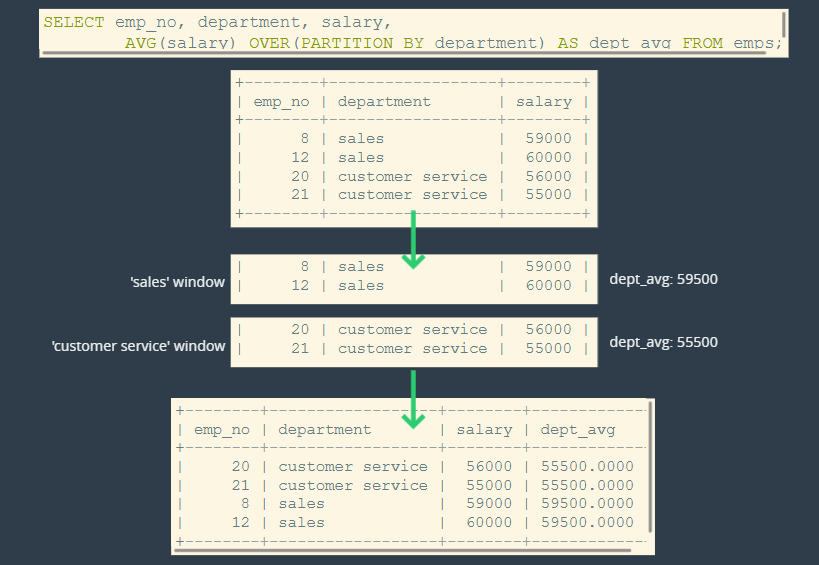
1. **Be part of the GROUP BY clause**, or
2. **Be used inside an aggregate function** (like SUM(), AVG(), COUNT(), etc.)

* SELECT @@global.sql\_mode;
* SELECT @@session.sql\_mode;
* SET SESSION sql\_mode = 'ONLY\_FULL\_GROUP\_BY,STRICT\_TRANS\_TABLES,NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,ERROR\_FOR\_DIVISION\_BY\_ZERO,NO\_ENGINE\_SUBSTITUTION';

## Window Functions



Window functions perform aggregate operations o groups of rows, but they produce a result for each row



Window functions work **like GROUP BY**, but instead of collapsing rows, they **assign** the aggregate value to each row within a defined "window" (group).

### OVER()

By specifying empty OVER() which means that the aggregate result will be attached with each and every row in the result dataset.

* SELECT department, SUM(salary) OVER(),
* AVG(salary) OVER(),
* MIN(salary) OVER(),
* COUNT(salary) OVER() FROM employees;

### OVER(PARTITION BY some\_column)

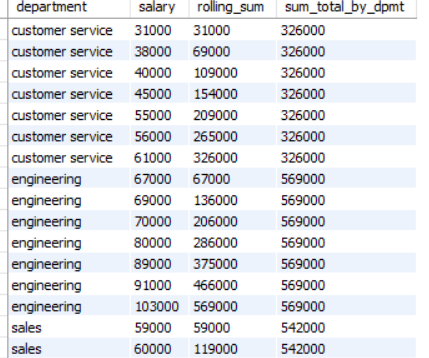
It means to form rows into groups of row and calculate the aggregate result from each group and then attach the result to each and every row in the dataset.

* SELECT department, salary,
* AVG(salary) OVER(PARTITION BY department) AS avg\_salary\_dpmt,
* SUM(salary) OVER(PARTITION BY department) AS total\_payroll,
* COUNT(salary) OVER(PARTITION BY department) AS count\_of\_emps\_in\_each\_depmt FROM employees;

### OVER(ORDER BY some\_column)

It re orders the values in each group and attaches to the each row in the data.

* SELECT department, salary,
* SUM(salary) OVER(PARTITION BY department ORDER BY salary ) AS rolling\_sum,
* SUM(salary) OVER(PARTITION BY department) AS sum\_total\_by\_dpmt
* FROM employees;

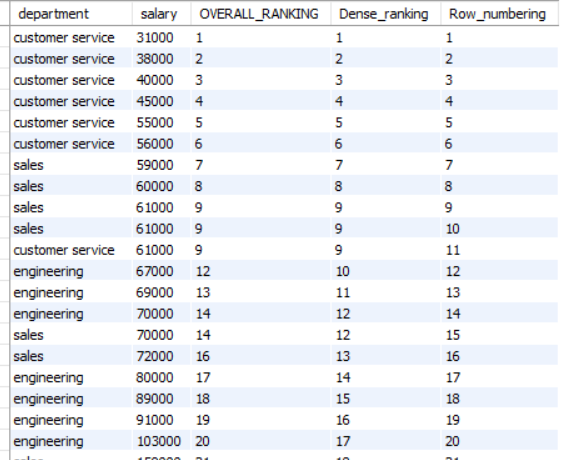


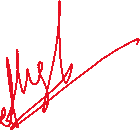


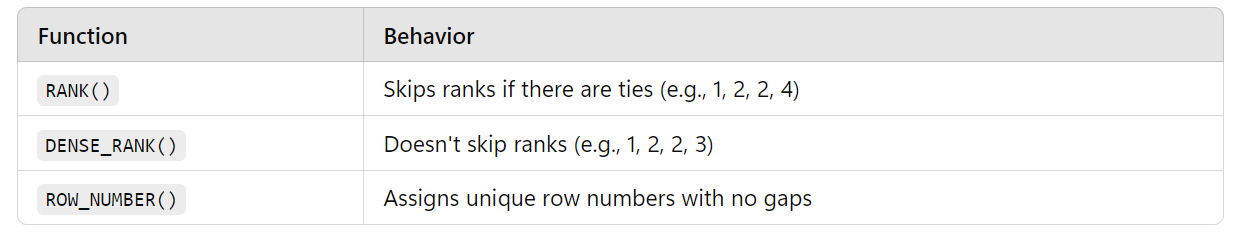
### RANK() , DENSE\_RANK(), ROW\_NUMBER()

The RANK() window function in MySQL is used to assign a unique rank to each row within a partition of a result set. It is commonly used for ranking rows based on a specific column's value.

* SELECT department, salary,
* RANK() OVER(ORDER BY salary) AS OVERALL\_RANKING,
* DENSE\_RANK() OVER(ORDER BY salary) AS Dense\_ranking,
* ROW\_NUMBER() OVER(ORDER BY salary) AS Row\_numbering
* FROM employees ;



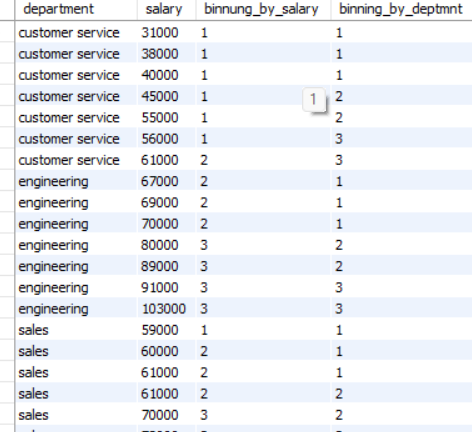




### NTILE(bucket\_number)

The NTILE() window function in MySQL is used to **divide rows into a specified number of groups** (or "tiles") and assign a **bucket number** to each row. It is commonly used for percentile-based ranking.

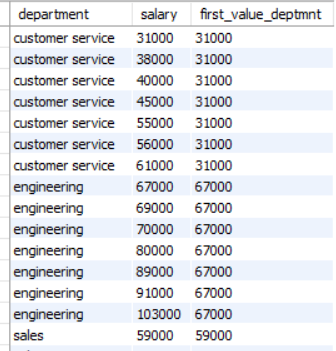
* SELECT department, salary,
* NTILE(3) OVER( ORDER BY salary) AS binnung\_by\_salary,
* NTILE(3) OVER(PARTITION BY department ORDER BY salary) AS binning\_by\_deptmnt
* FROM employees;



### FIRST\_VALUE()

The FIRST\_VALUE() function in MySQL is a **window function** that returns the **first value** of a column **within a specific window (partition)**. It is useful when you need to retrieve the **earliest or first occurrence** of a value in a dataset based on a specific order

* SELECT department, salary,
* FIRST\_VALUE(salary) OVER(PARTITION BY department ORDER BY salary) AS first\_value\_deptmnt
* FROM employees;





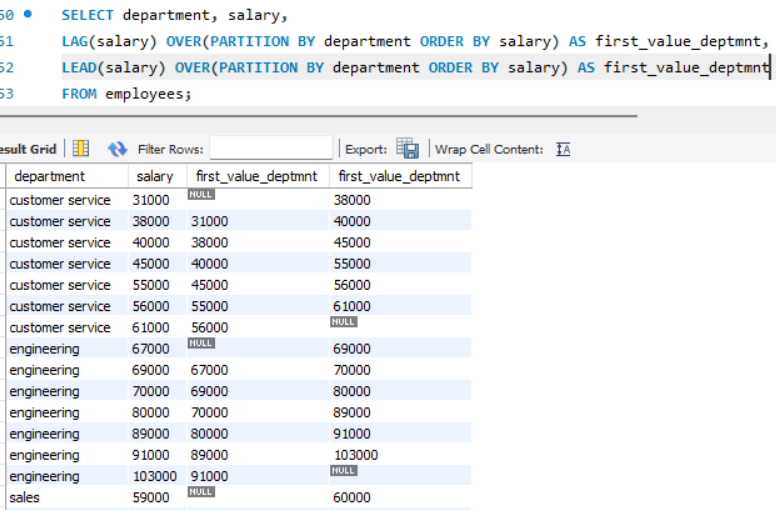
### LEAD & LAG

**LAG**

**Purpose:** Retrieves the **previous row's value** based on the ordering of rows.

**LEAD**

**Purpose:** Retrieves the **next row’s value** based on the ordering of rows.



## INSTAGRAM DATABASE CLONES