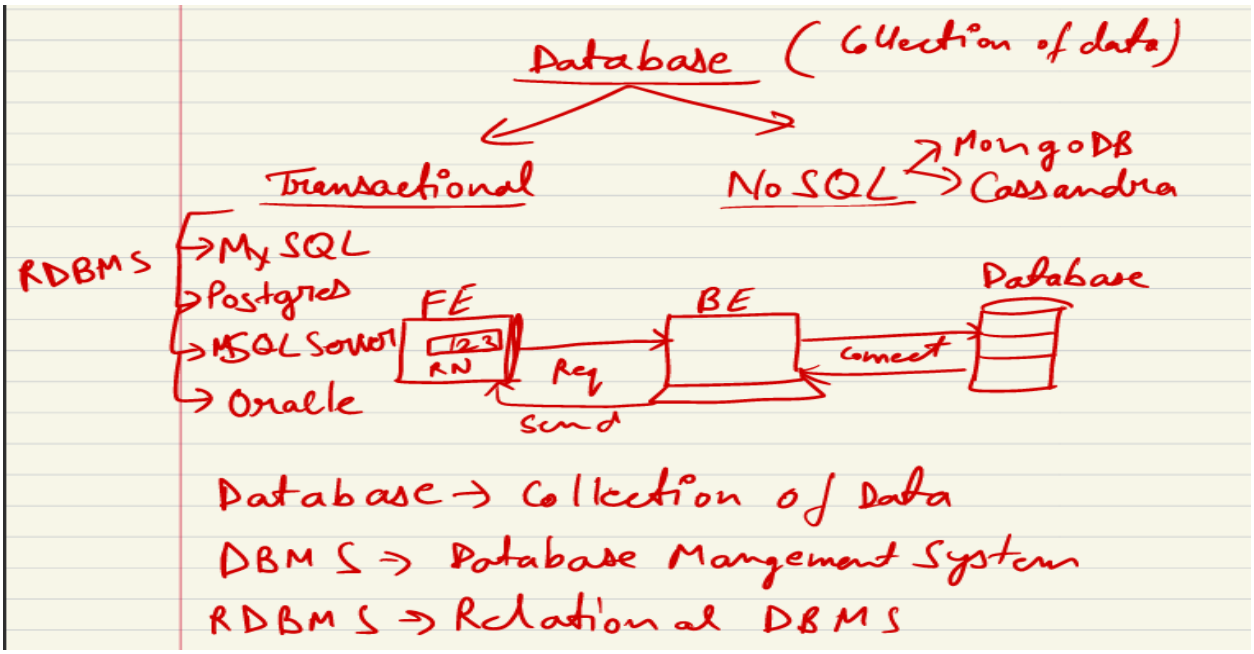
MYSQL

A Database Management System (DBMS) is basically a software where you can store, retrieve, define, and manage your data in a database.



Advantage of DBMS

- DBMS has lots of techniques to store, manipulate, and retrieve data
- DBMS considered as an most efficient handler to balance the data
- A DBMS uses lots of powerful functions to store, manipulate and retrieve data efficiently.
- Data Integrity and Security is one of the most strong part of DBMS
- The DBMS use data integrity to protect data and maintains the privacy
- Helps to reduced Application Development Time

RDBMS

A Relational database management system (RDBMS) is used for the database management system (DBMS). The concept is based on the relational model as introduced by E. F. Codd.

- The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries, and it consists of columns and rows.
- A record, also called a row, is each individual entry that exists in a table.
- A column is a vertical entity in a table that contains all information associated with a specific field in a table.

What is SQL ?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

MYSQL DataType

String Data types in MySQL

Data Type	Description	Length
char(n)	Stores n characters	n bytes (where n is in the range of 1–8,000)
nchar(n)	Stores n Unicode characters	2n bytes (where n is in the range of 1–4,000)
varchar(n)	Stores approximately ncharacters	Actual string length +2 bytes (where n is in the range of 1–8,000)
varchar(max)	Stores up to 231–1 characters	Actual string length +2 bytes
nvarchar(n)	Stores approximately ncharacters	2n(actual string length) +2 bytes (where n is in the range of 1–4,000)
nvarchar(max)	Stores up to ((231–1)/2)–2 characters	2n(actual string length) +2 bytes

Date Data types in MySQL

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(*fsp*)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(*fsp*)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.

Numeric Data types in MySQL

Data type	Description
BIT(*size*)	A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.
TINYINT(*size*)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(*size*)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)
MEDIUMINT(*size*)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)
INT(*size*)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255)
INTEGER(*size*)	Equal to INT(size)
BIGINT(*size*)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)
FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter
DOUBLE PRECISION(*size*, *d*)	
DECIMAL(*size*, *d*)	An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.

Type of command in SQL

It consists of SQL commands that can be used to define the database structures but not data.

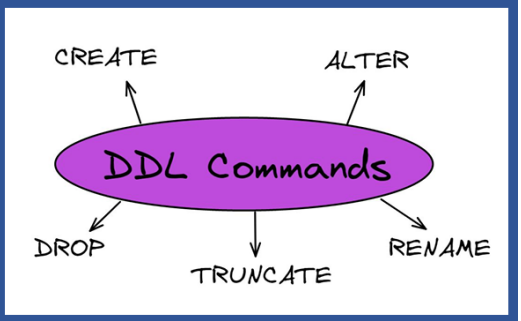
CREATE: This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).

DROP: This command is used to delete objects from the database.

ALTER: This is used to alter the structure of the database.

TRUNCATE: This is used to remove all records from a table, including all spaces allocated for the records are removed.

RENAME: This is used to rename an object existing in the database.



CREATE DATABASE

DROP DATABASE

CREATE TABLE

DROP TABLE

TRUNCATE TABLE

ALTER TABLE – ADD Column

ALTER TABLE – DROP COLUMN

ALTER TABLE – RENAME Column

ALTER TABLE – ALTER/MODIFY DATATYPE

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level.

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified

The SQL commands that deal with the manipulation of data present in the database belong to DML and this includes most of the SQL statements. It is the component of the SQL statement that controls access to data and to the database. Basically, DCL statements are grouped with DML statements.

- **INSERT**: It is used to insert data into a table.
- **UPDATE:** It is used to update existing data within a table.
- **DELETE**: It is used to delete records from a database table.

- It is a SQL statement that allows getting data from the database and imposing order upon it.
- It includes the **SELECT** statement.
- This command allows getting the data out of the database to perform operations with it.
- When a SELECT is fired against a table or tables the result is compiled into a further temporary table, which is displayed or perhaps received by the program i.e. a front-end.

- It includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.
- **GRANT**: This command gives users access privileges to the database.
- **REVOKE**: This command withdraws the user's access privileges given by using the GRANT command.

- SQL keywords are NOT case sensitive:

`select` is the same as `SELECT`

- Some database systems require a semicolon at the end of each SQL statement.
- Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.
- In this tutorial, we will use semicolon at the end of each SQL statement.

Command to see the list of databases

```
Show databases;
```

Command to Create database

```
create database practice_db;
```

Command to delete database

```
drop database class2_db;
```

Command to get inside a database (Highlight that DB)

```
use practice_db;
```

Command to create a table

```
CREATE TABLE if not EXISTS employee
(
    id INT,
    emp_name VARCHAR(20)
);
```

Dirctly use table for database (Practice_db.CREATE Table.... above command i.e it directly add table in Practice_db)

Command to see the list of tables

```
show tables;
```

Command to see the table definition

```
show create table employee;
```

Table:	employee

Create Table:	CREATE TABLE `employee` (
	  `id` int DEFAULT NULL,
	  `emp_name` varchar(20) DEFAULT NULL
	) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

Create employee table with few more columns

```
CREATE TABLE if not EXISTS employee_v1
(
    id INT,
    name VARCHAR(50),
    salary DOUBLE,
    hiring_date DATE
);
```

insert data into a table

```
--- Syntax 1 to insert data into a table
insert into employee_v1 values(1,'Shashank',1000,'2021-09-15');

--- Syntax 2 to insert data into a table
insert into employee_v1(salary,name,id)
values(2000,'Rahul',2);


--- This statement will fail
insert into employee_v1 values(1,'Shashank','2021-09-15');

Error Code: 1136. Column count doesn't match value count at row 1
```

Command to insert multiple records into a table

```
insert into employee_v1 values(3,'Amit',5000,'2021-10-28'),
(4,'Nitin',3500,'2021-09-16'),
(5,'Kajal',4000,'2021-09-20');
```

Command to see the table definition

show create table employee;

Table:	employee
Create Table:	CREATE TABLE `employee` (`id` int DEFAULT NULL, `emp_name` varchar(20) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

How to query or fetch the data from a table ?

```
select * from employee_v1;
```

Result Grid | Filter Rows:

id	name	salary	hiring_date
1	Shashank	1000	2021-09-15
2	Rahul	2000	NULL
3	Amit	5000	2021-10-28
4	Nitin	3500	2021-09-16
5	Kajal	4000	2021-09-20

```
select name,salary from employee_v1;
```

Result Grid | Filter Rows: | Export:

name	salary
Shashank	1000
Rahul	2000
Amit	5000
Nitin	3500
Kajal	4000

Example table for integrity constraints

```
CREATE TABLE if not EXISTS employee_with_constraints
(
    id INT,
    name VARCHAR(50) NOT NULL,
    salary DOUBLE,
    hiring_date DATE DEFAULT '2021-01-01',
    UNIQUE (id),
    CHECK (salary > 1000)
);
```

```
--- Example 1 for IC failure

--- Exception - Column 'name' cannot be null
insert into employee_with_constraints values(1,null,3000,'2021-11-20');


--- Correct record
insert into employee_with_constraints values(1,'Shashank',3000,'2021-11-20');
```

```
--- Example 2 for IC failure

--- Exception - Duplicate entry '1' for key 'employee_with_constraints.id'
insert into employee_with_constraints values(1,'Rahul',5000,'2021-10-23');


--- Another correct record because Unique can accept NULL as well (i.e null has on unique shape)
insert into employee_with_constraints values(null,'Rahul',5000,'2021-10-23');
```

The SQL standard allows multiple null values in a unique column, but some database systems, like MS SQL, only allow one.

```
--- Example 3 for IC failure

--- Exception - Duplicate entry NULL for key 'employee_with_constraints.id'
insert into employee_with_constraints values(null,'Rajat',2000,'2020-09-20');


--- Example 4 for IC failure

--- Exception - Check constraint 'employee_with_constraints_chk_1' is violated
insert into employee_with_constraints values(5,'Amit',500,'2023-10-24');
```

```
1  Create database class2_db;
2
3  use class2_db;
4
5  create table if not exists employee(
6      id int,
7      name VARCHAR(50),
8      address VARCHAR(50),
9      city VARCHAR(50)
10 );
11
12 insert into employee values(1, 'Shashank', 'RJPM', 'Lucknow');
13
14 select * from employee;
15
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	address	city
1	Shashank	RJPM	Lucknow

alter command used to change structure and add new column named DOB in the TABLE

```
alter table employee add DOB date;

select * from employee;
```

Result Grid | Filter Rows: | Export

id	name	address	city	DOB
1	Shashank	RJPM	Lucknow	NULL

modify existing column in a TABLE or change datatype of name column or increase length of name column

```
alter table employee modify column name varchar(100);
--- to cross check changes happen or not
show create table employee;
```

Form Editor | Navigate: 1 / 1

Table: employee

Create Table:

```
CREATE TABLE `employee` (
  `id` int DEFAULT NULL,
  `name` varchar(100) DEFAULT NULL,
  `address` varchar(50) DEFAULT NULL,
  `city` varchar(50) DEFAULT NULL,
  `DOB` date DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

delete existing column from given TABLE or remove city column from employee table

```
alter table employee drop column city;

select * from employee;
```

Result Grid | Filter Rows: | Export:

id	name	address	DOB
1	Shashank	RJPM	NULL

rename the column name to full_name

```
alter table employee rename column name to full_name;

select * from employee;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	full_name	address	DOB
1	Shashank	RJPM	NULL

To delete structure of entire table

```
drop table employee;

show tables;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Conte

Tables_in_class2_db

```
create table if not exists employee_v1(
    id int,
    name VARCHAR(50),
    age int,
    hiring_date date,
    salary int,
    city varchar(50)
);
insert into employee_v1 values(2,'Rahul', 25, '2021-08-10', 20000, 'Khajuraho');
insert into employee_v1 values(3,'Sunny', 22, '2021-08-11', 11000, 'Banaglore');
insert into employee_v1 values(5,'Amit', 25, '2021-08-11', 12000, 'Noida');
insert into employee_v1 values(6,'Puneet', 26, '2021-08-12', 50000, 'Gurgaon');
select * from employee_v1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	age	hiring_date	salary	city
2	Rahul	25	2021-08-10	20000	Khajuraho
3	Sunny	22	2021-08-11	11000	Banaglore
5	Amit	25	2021-08-11	12000	Noida
6	Puneet	26	2021-08-12	50000	Gurgaon

add unique integrity constraint on id COLUMN

```
alter table employee_v1 add constraint id_unique UNIQUE(id);

insert into employee_v1 values(2,'XYZ', 25, '2021-08-10', 50000, 'Gurgaon');
--- Error Code: 1062. Duplicate entry '2' for key 'employee_v1.id_unique'
```

drop unique integrity constraint on id COLUMN

```sql
alter table employee_v1 drop constraint id_unique;
insert into employee_v1 values(2,'XYZ', 25, '2021-08-10', 50000, 'Gurgaon');
select * from employee_v1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	age	hiring_date	salary	city
2	Rahul	25	2021-08-10	20000	Khajuraho
3	Sunny	22	2021-08-11	11000	Banaglore
5	Amit	25	2021-08-11	12000	Noida
6	Puneet	26	2021-08-12	50000	Gurgaon
1	XYZ	25	2021-08-10	50000	Gurgaon
1	XYZ	25	2021-08-10	50000	Gurgaon
2	XYZ	25	2021-08-10	50000	Gurgaon

create table with Primary_Key

```sql
Create table persons
(
    id int,
    name varchar(50),
    age int,
    Primary Key (id)

);
-- constraint pk Primary Key (id)
insert into persons values(1,'Shashank',29);

-- Try to insert duplicate value for primary key COLUMN
insert into persons values(1,'Rahul',28);
-- Error Code: 1062. Duplicate entry '1' for key 'persons.PRIMARY'


--- Try to insert null value for primary key COLUMN
insert into persons values(null,'Rahul',28);
-- Error Code: 1048. Column 'id' cannot be null
```

To check difference between Primary Key and Unique

The SQL standard allows multiple null values in a unique column, but some database systems, like MS SQL, only allow one.

```sql
Create table persons_3
(
    id int,
    name varchar(50),
    age int,
    Primary Key (id)

);
insert into persons_3 values(2,'Rahul',28);
insert into persons_3 values(3,'Amit',27);
alter table persons_3 add constraint age_unq UNIQUE(age);
insert into persons_3 values(4,'Amit',null);
select * from persons_3;
```

Result Grid | Filter Rows: | Edit: | Export/Import

id	name	age
2	Rahul	28
3	Amit	27
4	Amit	NULL
NULL	NULL	NULL

create tables for Foreign Key demo

```sql
create table customer
(
    cust_id int,
    name VARCHAR(50),
    age int,
    constraint pk Primary Key (cust_id)
);
create table orders
(
    order_id int,
    order_num int,
    customer_id int,
    constraint pk Primary Key (order_id),
    constraint fk Foreign Key (customer_id) REFERENCES customer(cust_id)
);
insert into customer values(1,"Shashank",29);
insert into customer values(2,"Rahul",30);
select * from customer;

insert into orders values(1001, 20, 1);
insert into orders values(1002, 30, 2);
select * from orders;
```

It will not allow to insert because referential integrity will violate FK contain same value more than one value present in PK

```
insert into orders values(1004, 35, 5);
-- Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails
-- (`class2_db`.`orders`, CONSTRAINT `fk` FOREIGN KEY (`customer_id`) REFERENCES `customer` (`cust_id`))
```

Difference between truncate and drop

```
--- data only deleted do not structure
truncate table persons;

--- Both data and structure deleted
drop table persons;
```

```
155  create table if not exists employee(
156      id int,
157      name VARCHAR(50),
158      age int,
159      hiring_date date,
160      salary int,
161      city varchar(50)
162  );
163
164  insert into employee values(1,'Shashank', 24, '2021-08-10', 10000, 'Lucknow');
165  insert into employee values(2,'Rahul', 25, '2021-08-10', 20000, 'Khajuraho');
166  insert into employee values(3,'Sunny', 22, '2021-08-11', 11000, 'Bangalore');
167  insert into employee values(5,'Amit', 25, '2021-08-11', 12000, 'Noida');
168  insert into employee values(1,'Puneet', 26, '2021-08-12', 50000, 'Gurgaon');
169  select * from employee;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	age	hiring_date	salary	city
1	Shashank	24	2021-08-10	10000	Lucknow
2	Rahul	25	2021-08-10	20000	Khajuraho
3	Sunny	22	2021-08-11	11000	Bangalore
5	Amit	25	2021-08-11	12000	Noida
1	Puneet	26	2021-08-12	50000	Gurgaon

how to count total records

(in this * will be appiled to every insert command so it count * return count, insted of * we can put anything i.e 1 or 1000)

```
select count(*) from employee;
```

Result Grid | Filter Rows: | Export:

count(*)
5

alias declaration (new name) as

```
select count(*) as total_row_count from employee;
```

Result Grid | Filter Rows: | Export: | Wrap Cell

total_row_count
5

aliases for multiple columns

```
select name as employee_name, salary as employee_salary from employee;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

employee_name	employee_salary
Shashank	10000
Rahul	20000
Sunny	11000
Amit	12000
Puneet	50000

print unique hiring_dates from the employee table when employees joined it

```
select Distinct(hiring_date) as distinct_hiring_dates from employee;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

distinct_hiring_dates
2021-08-10
2021-08-11
2021-08-12

```sql
--- How many unique age values in the table??
select  count(distinct(age)) as total_unique_ages from employee;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

total_unique_ages
4

Increment salary of each employee by 20% and display final result with new salary

```sql
SELECT  id,
        name,
        salary as old_salary,
        (salary + salary * 0.2) as new_salary
FROM employee;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	old_salary	new_salary
1	Shashank	10000	12000.0
2	Rahul	20000	24000.0
3	Sunny	11000	13200.0
5	Amit	12000	14400.0
1	Puneet	50000	60000.0

Updates will be made for all rows

If however you do want to update every row in the table, you can disable safe mode. Run the query below before running your update command. It changes the variable `SQL_SAFE_UPDATES`, which applies per session, so you need to run it every time you log in.

```sql
SET SQL_SAFE_UPDATES = 0;
```

To turn it back on again, run:

```sql
SET SQL_SAFE_UPDATES = 1;
```

```
UPDATE employee SET age = 20;
select * from employee;
```

Result Grid | Filter Rows: | Export:

id	name	age	hiring_date	salary	city
1	Shashank	20	2021-08-10	10000	Lucknow
2	Rahul	20	2021-08-10	20000	Khajuraho
3	Sunny	20	2021-08-11	11000	Bangalore
5	Amit	20	2021-08-11	12000	Noida
1	Puneet	20	2021-08-12	50000	Gurgaon

How to filter data using WHERE Clauses

```
select * from employee where hiring_date = '2021-08-10';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	age	hiring_date	salary	city
1	Shashank	20	2021-08-10	10000	Lucknow
2	Rahul	20	2021-08-10	20000	Khajuraho

how to delete specific records from table using delete command delete records of those employee who joined company on 2021-08-10

```
delete from employee where hiring_date = '2021-08-10';
select * from employee;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content

id	name	age	hiring_date	salary	city
3	Sunny	20	2021-08-11	11000	Bangalore
5	Amit	20	2021-08-11	12000	Noida
1	Puneet	20	2021-08-12	50000	Gurgaon

commit = changes done permanently we cannot go to previous state

rollback = changes done easily to previous state

How to apply auto increment

```
199 •  create table auto_inc_exmp
200    (
201      id int auto_increment,
202      name varchar(20),
203      primary key (id)
204    );
205
206    --- id automatically generated got only unique id
207 •  insert into auto_inc_exmp(name) values('Shashank');
208 •  insert into auto_inc_exmp(name) values('Rahul');
209 •  select * from auto_inc_exmp;
```

Result Grid | Filter Rows: | Edit:

id	name
1	Shashank
2	Rahul
NULL	NULL

Use of limit (how many record we want as result)

```
select * from employee limit 2;
```

Result Grid | Filter Rows: | Export:

id	name	age	hiring_date	salary	city
3	Sunny	20	2021-08-11	11000	Bangalore
5	Amit	20	2021-08-11	12000	Noida

Sorting of data

```
# arrage data in ascending order by default
select * from employee order by name;

# arrage data in descending order
select * from employee order by name desc;
```

Conditional Operator = < , > , <= , >=

Logical Operator = AND, OR, NOT

```
# list all employees who are getting salary more than or equal to 20000
select * from employee where salary>=20000;

# list all employees who are getting less than 20000
select * from employee where salary<20000;

# list all employees who are getting salary less than or equal to 20000
select * from employee where salary<=20000;


# filter the record where age of employees is equal to 20
select * from employee where age=20;

# filter the record where age of employees is not equal to 20
# we can use != or we can use <>
select * from employee where age != 20;
select * from employee where age <> 20;

# find those employees who joined the company on 2021-08-11 and their salary is less than 11500
select * from employee where hiring_date = '2021-08-11' and salary<11500;

# find those employees who joined the company after 2021-08-11 or  their salary is less than 20000
select * from employee where hiring_date > '2021-08-11' or salary<20000;
```

how to use Between operation in where clause

```
# get all employees data who joined the company between hiring_date 2021-08-05 to 2021-08-11
select * from employee where hiring_date between '2021-08-05' and '2021-08-11';

# get all employees data who are getting salary in the range of 10000 to 28000
select * from employee where salary between 10000 and 28000;
```

how to use LIKE operation in where clause

1] % -> Zero, one or more than one characters

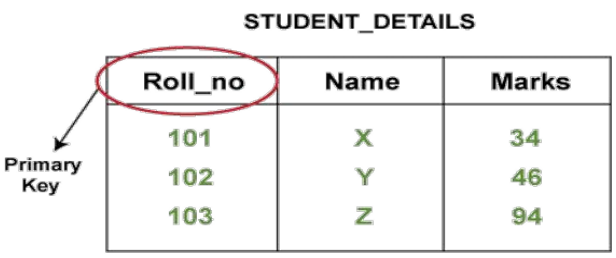
2] _ -> only one character

```sql
# get all those employees whose name starts with 'S'
select * from employee where name like 'S%';

# get all those employees whose name starts with 'Sh'
select * from employee where name like 'Sh%';

# get all those employees whose name ends with 'l'
select * from employee where name like '%l';

# get all those employees whose name starts with 'S' and ends with 'k'
select * from employee where name like 'S%k';

# Get all those employees whose name will have exact 5 (i.e it has 5 placeholder) characters
select * from employee where name like '_____';

# Return all those employees whose name contains atleast 5 characters
select * from employee where name like '%_____';
select * from employee where name like '_____%';
select * from employee where name like '%_____%';
```

Class 3 and 4

Primary Key

- **Uniqueness**: Each primary key value must be unique per table row.
- **Immutable**: Primary keys should not change once set.
- **Simplicity**: Ideal to keep primary keys as simple as possible.
- **Non-Intelligent**: They shouldn't contain meaningful information.
- **Indexed**: Primary keys are automatically indexed for faster data retrieval.
- **Referential Integrity**: They serve as the basis for foreign keys in other tables.
- **Data Type:** Common types are integer or string.



Foreign Key

- **Referential Integrity**: Foreign keys link records between tables, maintaining data consistency.
- **Nullable**: Foreign keys can contain null values unless specifically restricted.
- **Match Primary Keys**: Each foreign key value must match a primary key value in the parent table, or be null.
- **Ensure Relationships**: They define the relationship between tables in a database.
- **No Uniqueness**: Foreign keys don't need to be unique.

Delete command

The DELETE command in SQL is used to remove existing records from a table. Here's a basic syntax:

```
DELETE FROM table_name WHERE condition;
```

For example, to delete a record from a Students table where ID equals 5:

```
DELETE FROM Students WHERE ID = 5;
```

⚠️ **Be careful:** if you run the DELETE command without a WHERE clause, it will delete all records from the table.

	DROP	TRUNCATE	DELETE
Purpose	Completely removes the entire table structure from the database	Removes all rows from a table, but the table structure remains	Removes specific rows based on a condition or all rows from a table, but the table structure remains
Transaction Control	Cannot be rolled back	Cannot be rolled back	Can be rolled back
Space Reclaiming	Releases the object and its space	Frees the space containing the table	Doesn't free up space, but leaves empty space for future use
Speed	Fastest as it removes all data and structure	Faster than DELETE as it doesn't log individual row deletions	Slowest as it logs individual row deletions
Referential Integrity	Not checked	Checked	Checked
Where Clause	Not applicable	Not applicable	Applicable
Command Type	DDL (Data Definition Language)	DDL (Data Definition Language)	DML (Data Manipulation Language)

How to use IS NULL or IS NOT NULL in the where clause

```
4  • create table if not exists employee(
5        id int,
6        name VARCHAR(50),
7        age int,
8        hiring_date date,
9        salary int,
10       city varchar(50)
11   );
12   # How to use IS NULL or IS NOT NULL in the where clause
13 • insert into employee values(10,'Kapil', null, '2021-08-10', 10000, 'Assam');
14 • insert into employee values(11,'Nikhil', 30, '2021-08-10', null, 'Assam');
15
16   # get all those employees whos age value is null
17 • select * from employee where age is null;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	age	hiring_date	salary	city
10	Kapil	NULL	2021-08-10	10000	Assam

```
16      # get all those employees whos salary value is not null
17 •    select * from employee where salary is not null;
18
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	age	hiring_date	salary	city
10	Kapil	NULL	2021-08-10	10000	Assam

Table and Data for Group By

```
create table orders_data
(
 cust_id int,
 order_id int,
 country varchar(50),
 state varchar(50)
);

insert into orders_data values(1,100,'USA','Seattle');
insert into orders_data values(2,101,'INDIA','UP');
insert into orders_data values(2,103,'INDIA','Bihar');
insert into orders_data values(4,108,'USA','WDC');
insert into orders_data values(5,109,'UK','London');
insert into orders_data values(4,110,'USA','WDC');
insert into orders_data values(3,120,'INDIA','AP');
insert into orders_data values(2,121,'INDIA','Goa');
insert into orders_data values(1,131,'USA','Seattle');
insert into orders_data values(6,142,'USA','Seattle');
insert into orders_data values(7,150,'USA','Seattle');

select * from orders_data;
```

calculate total order placed country wise

```
44 •    select country, count(*) as order_count_by_each_country from orders_data group by country;
45
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

country	order_count_by_each_country
USA	6
INDIA	4
UK	1

Write a query to find the total salary by each age group

```
select age, sum(salary) as total_salary_by_each_age_group from employee group by age;
```

calculate different aggregated metrices for salary

```
50 •  select age,
51          sum(salary) as total_salary_by_each_age_group,
52          max(salary) as max_salary_by_each_age_group,
53          min(salary) as min_salary_by_each_age_group,
54          avg(salary) as avg_salary_by_each_age_group,
55          count(*) as total_employees_by_each_age_group
56    from employee group by age;
57
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

age	total_salary_by_each_age_group	max_salary_by_each_age_group	min_salary_by_each_age_group	avg_salary_by_each_age_group	total_employees_by_each_age_group
NULL	10000	10000	10000	10000.0000	1
30	NULL	NULL	NULL	NULL	1
40	1000	1000	1000	1000.0000	1
20	5000	5000	5000	5000.0000	1

Group by on multiple columns

```
64 •    select
65          country,
66          state,
67          count(*) as state_wise_order
68      from orders_data
69      group by country, state;
```

Result Grid | Filter Rows:

country	state	state_wise_order
USA	Seattle	4
INDIA	UP	1
INDIA	Bihar	1
USA	WDC	2
UK	London	1
INDIA	AP	1
INDIA	Goa	1

Use of Having Clause

Write a query to find the country where only 1 order was placed

```
71 •    select country from orders_data group by country having count(*)=1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

country
UK

Where Clause and Group By Clause --> What should be the proper sequence??

Answer -> Where Clause and then Group By

IMP : select * from table_1 , table_2; // it give cross product of two table

How to use GROUP_CONCAT

Query - Write a query to print distinct states present in the dataset for each country?

```
select country, GROUP_CONCAT(state) as states_in_country from orders_data group by country;

select country, GROUP_CONCAT(distinct state) as states_in_country from orders_data group by country;

select country, GROUP_CONCAT(distinct state order by state desc) as states_in_country from orders_data group by country;

select country, GROUP_CONCAT(distinct state order by state desc separator '<->') as states_in_country from orders_data group by country;
```

Output:

```
+---------+-----------------------------------------+
| country | states_in_country                       |
+---------+-----------------------------------------+
| INDIA   | UP,Bihar,AP,Goa                         |
| UK      | London                                  |
| USA     | Seattle,WDC,WDC,Seattle,Seattle,Seattle |
+---------+-----------------------------------------+
+---------+-------------------+
| country | states_in_country |
+---------+-------------------+
| INDIA   | AP,Bihar,Goa,UP   |
| UK      | London            |
| USA     | Seattle,WDC       |
+---------+-------------------+
+---------+-------------------+
| country | states_in_country |
+---------+-------------------+
| INDIA   | UP,Goa,Bihar,AP   |
| UK      | London            |
| USA     | WDC,Seattle       |
+---------+-------------------+
+---------+------------------------+
| country | states_in_country      |
+---------+------------------------+
| INDIA   | UP<->Goa<->Bihar<->AP  |
| UK      | London                 |
| USA     | WDC<->Seattle          |
+---------+------------------------+
```

Subqueries in SQL

Write a query to print all those employee records who are getting more salary than 'Rohit'

```
create table employees
(
    id int,
    name varchar(50),
    salary int
);
insert into employees values(1,'Shashank',5000),(2,'Amit',5500),(3,'Rahul',7000),(4,'Rohit',6000),(5,'Nitin',4000),(6,'Sunny',7500);

# Wrong solution -> select * from employees where salary > 6000;

select * from employees where salary > (select salary from employees where name='Rohit');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	salary
3	Rahul	7000
6	Sunny	7500

Use of IN and NOT IN

Write a query to print all orders which were placed in 'Seattle' or 'Goa'

```
SELECT * FROM orders_data WHERE state in ('Seattle', 'Goa');
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

cust_id	order_id	country	state
1	100	USA	Seattle
2	121	INDIA	Goa
1	131	USA	Seattle
6	142	USA	Seattle
7	150	USA	Seattle

```
create table customer_order_data
(
    order_id int,
    cust_id int,
    supplier_id int,
    cust_country varchar(50)
);
insert into customer_order_data values(101,200,300,'USA'),(102,201,301,'INDIA'),(103,202,302,'USA'),(104,203,303,'UK');

create table supplier_data
(
    supplier_id int,
    sup_country varchar(50)
);
insert into supplier_data values(300,'USA'),(303,'UK');
```

write a query to find all customer order data where all customers are from same countries as the suppliers

```
112 •   select * from customer_order_data where cust_country in
113     (select distinct sup_country from supplier_data);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

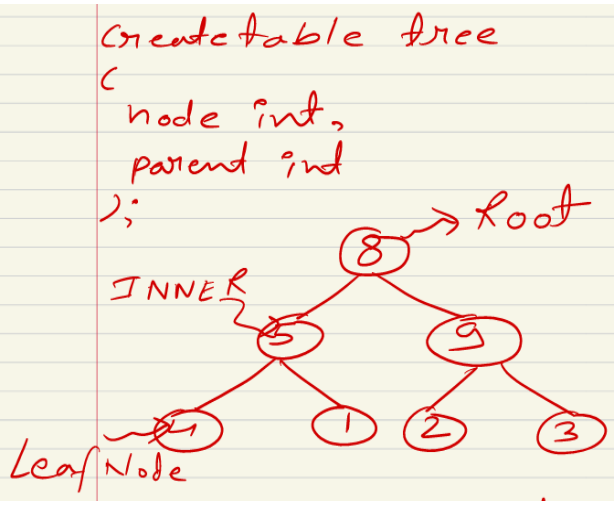
order_id	cust_id	supplier_id	cust_country
101	200	300	USA
103	202	302	USA
104	203	303	UK

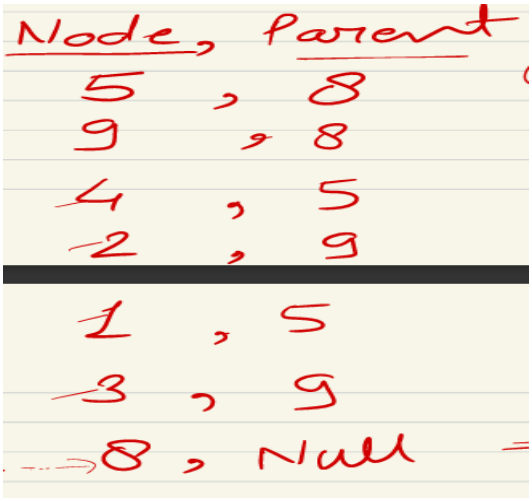
```
115     # Another example of Sub-Query
116 •   select *
117     from (select
118             country,
119             count(*) as country_wise_order
120         from orders_data
121         group by country) result
122     where country_wise_order=1;
```

Result Grid | Filter Rows: | Export:

country	country_wise_order
UK	1

Uber SQL Interview questions





```sql
create table tree
(
    node int,
    parent int
);
insert into tree values (5,8),(9,8),(4,5),(2,9),(1,5),(3,9),(8,null);
select node,
       CASE
            when node not in (select distinct parent from tree where parent is not null) then 'LEAF'
            when parent is null then 'ROOT'
            else 'INNER'
       END as node_type
from tree;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

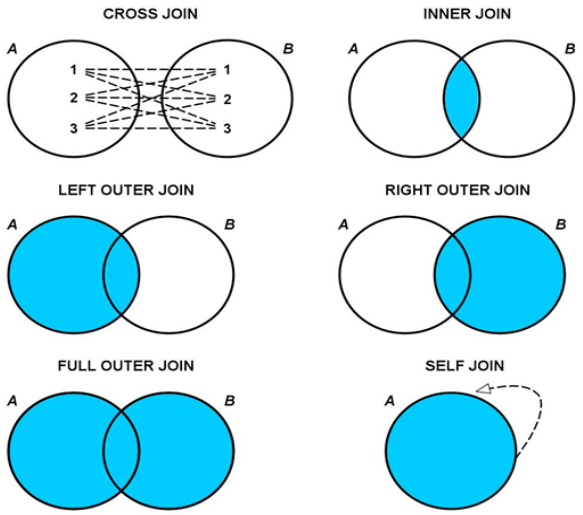
node	node_type
5	INNER
9	INNER
4	LEAF
2	LEAF
1	LEAF
3	LEAF
8	ROOT

1. **LEAF Node**: A node is considered a **LEAF** if it does not appear as a parent for any other node. This means it has no children.
2. **ROOT Node**: A node is considered a **ROOT** if its parent is `**NULL**`. This means it is the starting point of the tree and has no parent.
3. **INNER Node**: A node is considered an **INNER** node if it has a parent and is also a parent to other nodes. It is neither a leaf nor a root.

Examples for join

SQL joins are used to combine rows from two or more tables, based on a related column between them. Here are the main types of SQL joins:

- Inner Join
- Left Join
- Right Join
- Full Join
- Cross Join



```sql
create table orders
(
    order_id int,
    cust_id int,
    order_dat date,
    shipper_id int
);

create table customers
(
    cust_id int,
    cust_name varchar(50),
    country varchar(50)
);

create table shippers
(
    ship_id int,
    shipper_name varchar(50)
);

insert into orders values(10308, 2, '2022-09-15', 3);
insert into orders values(10309, 30, '2022-09-16', 1);
insert into orders values(10310, 41, '2022-09-19', 2);

insert into customers values(1, 'Neel', 'India');
insert into customers values(2, 'Nitin', 'USA');
insert into customers values(3, 'Mukesh', 'UK');

insert into shippers values(3,'abc');
insert into shippers values(1,'xyz');

select * from orders;
select * from customers;
select * from shippers;
```

STDIN

Input for the program (Optional)

Output:

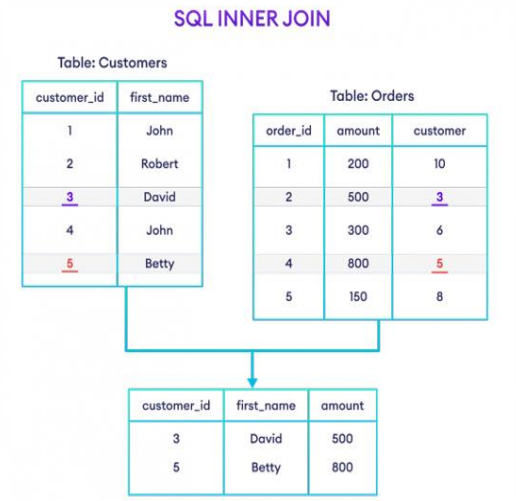
```
+----------+---------+------------+------------+
| order_id | cust_id | order_dat  | shipper_id |
+----------+---------+------------+------------+
|    10308 |       2 | 2022-09-15 |          3 |
|    10309 |      30 | 2022-09-16 |          1 |
|    10310 |      41 | 2022-09-19 |          2 |
+----------+---------+------------+------------+
+---------+-----------+---------+
| cust_id | cust_name | country |
+---------+-----------+---------+
|       1 | Neel      | India   |
|       2 | Nitin     | USA     |
|       3 | Mukesh    | UK      |
+---------+-----------+---------+
+---------+-------------+
| ship_id | shipper_name |
+---------+-------------+
|       3 | abc          |
|       1 | xyz          |
+---------+-------------+
```

Inner Join

Returns records that have matching values in both tables.

Syntax:

```
SELECT Customers.customer_id,
       Customers.first_name,
       Orders.amount
FROM Customers
INNER JOIN Orders
ON Orders.customer = Customers.customer_id;
```



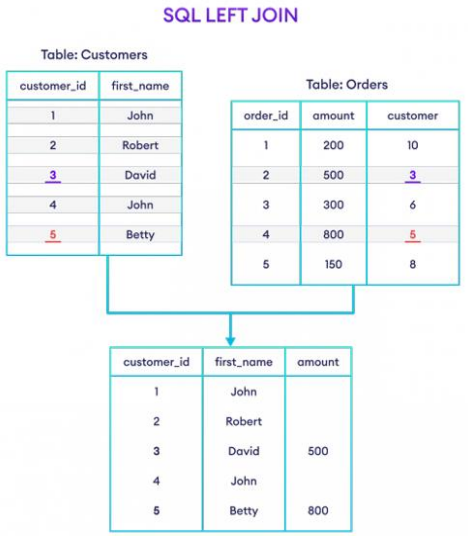

```sql
# get the customer informations for each order order, if value of customer is present in orders TABLE
select
o.*, c.*
from orders o
inner join customers c on o.cust_id = c.cust_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	cust_id	order_dat	shipper_id	cust_id	cust_name	country
10308	2	2022-09-15	3	2	Nitin	USA

Left Join

Returns all records from the left table (table1), and the matched records from the right table (table2). If no match, the result is NULL on the right side.



Syntax:

```
SELECT Customers.customer_id,
       Customers.first_name,
       Orders.amount
FROM Customers
LEFT JOIN Orders
ON Orders.customer = Customers.customer_id;
```

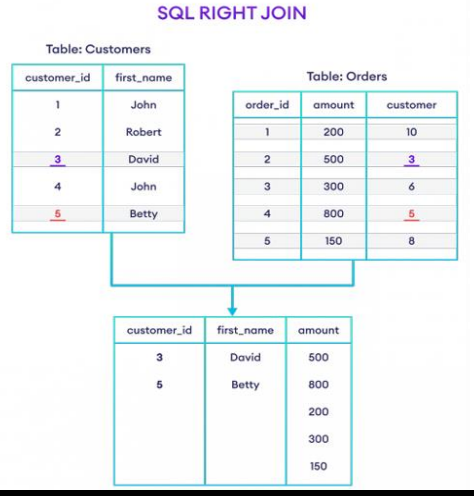
```
177 •   select
178     o.*, c.*
179     from orders o
180     left join customers c on o.cust_id = c.cust_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	cust_id	order_dat	shipper_id	cust_id	cust_name	country
10308	2	2022-09-15	3	2	Nitin	USA
10309	30	2022-09-16	1	NULL	NULL	NULL
10310	41	2022-09-19	2	NULL	NULL	NULL

Right Join

Returns all records from the right table (table2), and the matched records from the left table (table1). If no match, the result is NULL on the left side.



Syntax:

```
SELECT Customers.customer_id,
       Customers.first_name,
       Orders.amount
FROM Customers
RIGHT JOIN Orders
ON Orders.customer = Customers.customer_id;
```

```
182 •   select
183     o.*, c.*
184     from orders o
185     right join customers c on o.cust_id = c.cust_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

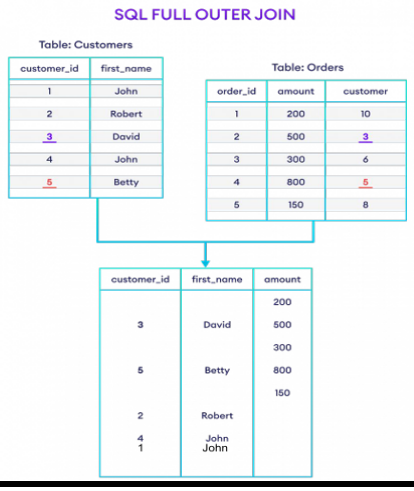
order_id	cust_id	order_dat	shipper_id	cust_id	cust_name	country
NULL	NULL	NULL	NULL	1	Neel	India
10308	2	2022-09-15	3	2	Nitin	USA
NULL	NULL	NULL	NULL	3	Mukesh	UK

Full Join

Returns all records when there is a match in either left (table1) or right (table2) table records.

Syntax:

```
SELECT Customers.customer_id,
       Customers.first_name,
       Orders.amount
FROM Customers
FULL OUTER JOIN Orders
ON Orders.customer = Customers.customer_id;
```

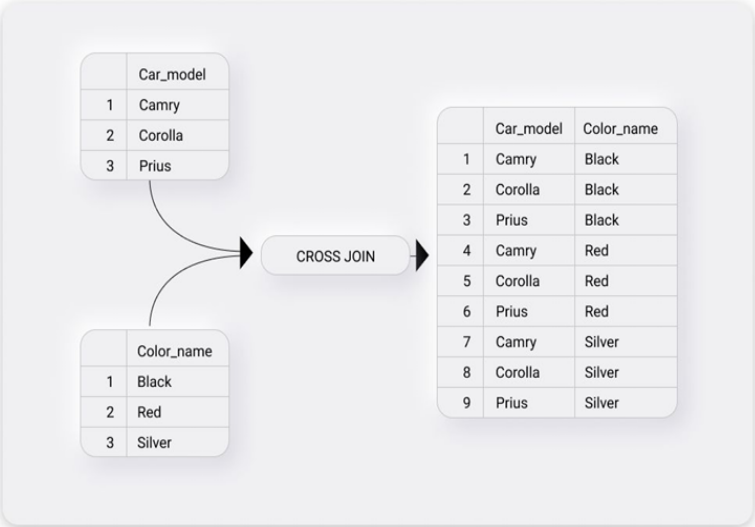


Cross Join

Returns the Cartesian product of the sets of records from the two or more joined tables when no WHERE clause is used with CROSS JOIN.

Syntax:

```
SELECT Model.car_model,
       Color.color_name
FROM Model
Cross JOIN Color;
```



Self Join

A regular join, but the table is joined with itself.

Let's take a look at an example. Consider the table `Employees`:

Id	FullName	Salary	ManagerId
1	John Smith	10000	3
2	Jane Anderson	12000	3
3	Tom Lanon	15000	4
4	Anne Connor	20000	
5	Jeremy York	9000	1

Now, to show the name of the manager for each employee in the same row, we can run the following query:

```
SELECT
   employee.Id,
       employee.FullName,
       employee.ManagerId,
       manager.FullName as ManagerName
FROM Employees employee
JOIN Employees manager
ON employee.ManagerId = manager.Id
```

Id	FullName	ManagerId	ManagerName
1	John Smith	3	Tom Lanon
2	Jane Anderson	3	Tom Lanon
3	Tom Lanon	4	Anne Connor
5	Jeremy York	1	John Smith

```
187     # How to join more than 2 datasets?
188     # perform inner JOIN
189     # get the customer informations for each order order, if value of customer is present in orders TABLE
190     # also get the information of shipper name
191     select
192     o.*, c.*, s.*
193     from orders o
194     inner join customers c on o.cust_id = c.cust_id
195     inner join shippers s on o.shipper_id = s.ship_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	cust_id	order_dat	shipper_id	cust_id	cust_name	country	ship_id	shipper_name
10308	2	2022-09-15	3	2	Nitin	USA	3	abc

```sql
create table employees_full_data
(
    emp_id int,
    name varchar(50),
    mgr_id int
);
insert into employees_full_data values(1, 'Shashank', 3);
insert into employees_full_data values(2, 'Amit', 3);
insert into employees_full_data values(3, 'Rajesh', 4);
insert into employees_full_data values(4, 'Ankit', 6);
insert into employees_full_data values(6, 'Nikhil', null);

# Write a query to print the distinct names of managers??

select
emp.name as manager
from employees_full_data emp
inner join (select distinct mgr_id as mgr_id from employees_full_data) mgr on mgr.mgr_id = emp.emp_id;
```

STDIN

Input for the

Output:

```
+---------+
| manager |
+---------+
| Rajesh  |
| Ankit   |
| Nikhil  |
+---------+
```

Logic : (entire table) **INNER JOIN** (distinct manager name table)

Group By WITH ROLLUP

The GROUP BY WITH ROLLUP clause in MySQL is an extension of the GROUP BY clause that allows you to perform subtotals and grand totals in a single query. This is particularly useful when you need to generate reports that include aggregated data at different levels of detail.

Basic Usage of GROUP BY

Before diving into `ROLLUP`, let's start with a basic example of `GROUP BY`:

```sql
SELECT
    department,
    SUM(salary) AS total_salary
FROM
    employees
GROUP BY
    department;
```

Output:

Department	Total Salary
Sales	100,000
HR	80,000
IT	120,000

In this query, we are calculating the total salary for each department. The `GROUP BY` clause groups the results by department, and the `SUM()` function calculates the total salary for each group.

Using GROUP BY WITH ROLLUP

Now, let's add `ROLLUP` to the mix:

```sql
SELECT
    department,
    SUM(salary) AS total_salary
FROM
    employees
GROUP BY
    department WITH ROLLUP;
```

Output:

Department	Total Salary
Sales	100,000
HR	80,000
IT	120,000
NULL	300,000

In this output, the last row (`NULL`) represents the grand total for all departments combined. The `WITH ROLLUP` modifier calculates not only the total salary for each department but also a grand total across all departments.

How ROLLUP Works

`ROLLUP` adds an extra row to your result set that represents the aggregate of all the other rows. Here's how it works in more detail:

1. **Single-Level ROLLUP**:
   - When used with a single column, `ROLLUP` generates a summary row that shows the total for all groups combined.
2. **Multi-Level ROLLUP**:
   - When used with multiple columns, `ROLLUP` creates subtotals for each level of aggregation in addition to the grand total.

Example with Multiple Columns

Consider a more complex example where we want to calculate subtotals by department and job title:

```sql
SELECT
    department,
    job_title,
    SUM(salary) AS total_salary
FROM
    employees
GROUP BY
    department,
    job_title WITH ROLLUP;
```

Output:

Department	Job Title	Total Salary
Sales	Manager	60,000
Sales	Executive	40,000
Sales	NULL	100,000
HR	Manager	50,000
HR	Executive	30,000
HR	NULL	80,000
IT	Developer	90,000
IT	Analyst	30,000

Department	Job Title	Total Salary
IT	NULL	120,000
NULL	NULL	300,000

In this output:

- Rows where `job_title` is `NULL` represent subtotals for each department.
- The final row with both `department` and `job_title` as `NULL` represents the grand total for all departments and job titles.

Key Points to Remember

- **NULLs in ROLLUP**: `NULL` values in the result set indicate subtotals or grand totals. They do not represent missing data but rather an aggregated level of detail.
- **Order of Columns**: The order of columns in the `GROUP BY` clause affects the roll-up hierarchy. Make sure to list columns in the desired order of aggregation.
- **Performance Considerations**: Using `ROLLUP` can be resource-intensive, especially with large datasets or complex queries. Ensure your database is optimized for performance.

VIEW

Views

A view in SQL is a virtual table based on the result-set of an SQL statement. It contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

Here are some key points about views:

- You can add SQL functions, WHERE, and JOIN statements to a view and display the data as if the data were coming from one single table.
- A view always shows up-to-date data. The database engine recreates the data every time a user queries a view.
- Views can be used to encapsulate complex queries, presenting users with a simpler interface to the data.
- They can be used to restrict access to sensitive data in the underlying tables, presenting only non-sensitive data to users.

Syntax to create Views

```
CREATE VIEW View_Products AS
SELECT ProductName, Price
FROM Products
WHERE Price > 30;
```

Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
CREATE VIEW emp_view AS
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID;
```

Create View of grouped records on Employee table

emp_view

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

Droping view : drop view department_wise_salary;

Any Operation

The ANY operator in MySQL is used in SQL queries to compare a value against a set of values returned by a subquery. It's a logical operator that evaluates to TRUE if the comparison is TRUE for **any** of the values in the list. This operator is particularly useful when you need to check if a value matches at least one value in a subquery.

```
CREATE TABLE Students (
    StudentID INT,
    StudentName VARCHAR(50)
);
INSERT INTO Students VALUES
(1, 'John'),
(2, 'Alice'),
(3, 'Bob');

CREATE TABLE Courses (
    CourseID INT,
    CourseName VARCHAR(50)
);
INSERT INTO Courses VALUES
(100, 'Math'),
(101, 'English'),
(102, 'Science');

CREATE TABLE Enrollments (
    StudentID INT,
    CourseID INT
);
INSERT INTO Enrollments VALUES
(1, 100),
(1, 101),
(2, 101),
(2, 102),
(3, 100),
(3, 102);
Select* From students;
Select*From Courses;
Select*From Enrollments;
```

Output:

```
+-----------+-------------+
| StudentID | StudentName |
+-----------+-------------+
|         1 | John        |
|         2 | Alice       |
|         3 | Bob         |
+-----------+-------------+
+----------+------------+
| CourseID | CourseName |
+----------+------------+
|      100 | Math       |
|      101 | English    |
|      102 | Science    |
+----------+------------+
+-----------+----------+
| StudentID | CourseID |
+-----------+----------+
|         1 |      100 |
|         1 |      101 |
|         2 |      101 |
|         2 |      102 |
|         3 |      100 |
|         3 |      102 |
+-----------+----------+
```

```
#Example: Lets find the students who are enrolled in any course taken by 'John':

Select Distinct s.StudentName
From Students s
INNER JOIN Enrollments e ON s.StudentID = e.StudentID
WHERE s.StudentName <> 'John' and e.CourseID = ANY (SELECT e2.CourseID
    FROM Enrollments e2
    INNER JOIN Students s2 ON e2.StudentID = s2.StudentID
    WHERE s2.StudentName = 'John');
```

Output:

```
+-------------+
| StudentName |
+-------------+
| Alice       |
| Bob         |
+-------------+
```

All Operation

The ALL operator in MySQL is used to compare a value against a set of values returned by a subquery. Unlike the ANY operator, which evaluates to TRUE if at least one comparison is TRUE, the ALL operator evaluates to TRUE only if **all** comparisons are TRUE. This makes ALL useful for conditions where you want a value to meet a certain criteria relative to every value in a subquery.

```sql
CREATE TABLE Products (
    ProductID INT,
    ProductName VARCHAR(50),
    Price DECIMAL(5,2)
);

INSERT INTO Products VALUES
(1, 'Apple', 1.20),
(2, 'Banana', 0.50),
(3, 'Cherry', 2.00),
(4, 'Date', 1.50),
(5, 'Elderberry', 3.00);


CREATE TABLE Orders (
    OrderID INT,
    ProductID INT,
    Quantity INT
);

INSERT INTO Orders VALUES
(1001, 1, 10),
(1002, 2, 20),
(1003, 3, 30),
(1004, 1, 5),
(1005, 4, 25),
(1006, 5, 15);

select * from products;
select * from Orders;
```

Output:

```
+-----------+-------------+-------+
| ProductID | ProductName | Price |
+-----------+-------------+-------+
|         1 | Apple       |  1.20 |
|         2 | Banana      |  0.50 |
|         3 | Cherry      |  2.00 |
|         4 | Date        |  1.50 |
|         5 | Elderberry  |  3.00 |
+-----------+-------------+-------+
+---------+-----------+----------+
| OrderID | ProductID | Quantity |
+---------+-----------+----------+
|    1001 |         1 |       10 |
|    1002 |         2 |       20 |
|    1003 |         3 |       30 |
|    1004 |         1 |        5 |
|    1005 |         4 |       25 |
|    1006 |         5 |       15 |
+---------+-----------+----------+
```

```sql
#Now, suppose we want to find the products that have a price less than the
#price of all products ordered in order 1001:

SELECT p.ProductName
FROM Products p
WHERE p.Price < ALL (
    SELECT pr.Price
    FROM Products pr
    INNER JOIN Orders o ON pr.ProductID = o.ProductID
    WHERE o.OrderID = 1001
);
```

Output:

```
+-------------+
| ProductName |
+-------------+
| Banana      |
+-------------+
```

EXISTS Operation

The EXISTS operation in MySQL is a subquery used to test for the existence of any records in a subquery. It returns TRUE if the subquery returns one or more records, and FALSE if the subquery returns no records. It's often used in WHERE clauses to filter rows based on whether certain conditions are met in related tables.

```
CREATE TABLE Customers (
    CustomerID INT,
    CustomerName VARCHAR(50)
);

INSERT INTO Customers VALUES
(1, 'John Doe'),
(2, 'Alice Smith'),
(3, 'Bob Johnson'),
(4, 'Charlie Brown'),
(5, 'David Williams');

CREATE TABLE Orders (
    OrderID INT,
    CustomerID INT,
    OrderDate DATE
);

INSERT INTO Orders VALUES
(1001, 1, '2023-01-01'),
(1002, 2, '2023-02-01'),
(1003, 1, '2023-03-01'),
(1004, 3, '2023-04-01'),
(1005, 5, '2023-05-01');

select*from Customers;
select*from Orders;
```

Output:

```
+------------+---------------+
| CustomerID | CustomerName  |
+------------+---------------+
|          1 | John Doe      |
|          2 | Alice Smith   |
|          3 | Bob Johnson   |
|          4 | Charlie Brown |
|          5 | David Williams |
+------------+---------------+
+---------+------------+------------+
| OrderID | CustomerID | OrderDate  |
+---------+------------+------------+
|    1001 |          1 | 2023-01-01 |
|    1002 |          2 | 2023-02-01 |
|    1003 |          1 | 2023-03-01 |
|    1004 |          3 | 2023-04-01 |
|    1005 |          5 | 2023-05-01 |
+---------+------------+------------+
```

```
#Example: Lets find the customers who have placed at least one order.

SELECT c.CustomerName
FROM Customers c
WHERE EXISTS (
    SELECT 1
    FROM Orders o
    WHERE o.CustomerID = c.CustomerID
);
```

```
+----------------+
| CustomerName   |
+----------------+
| John Doe       |
| Alice Smith    |
| Bob Johnson    |
| David Williams |
+----------------+
```

NOT EXISTS Operation

The NOT EXISTS operator in MySQL is used to check for the non-existence of records in a subquery. It is essentially the opposite of EXISTS and returns TRUE when the subquery returns no records and FALSE when the subquery returns one or more records. This is particularly useful for finding records that do not have a corresponding entry in another table.

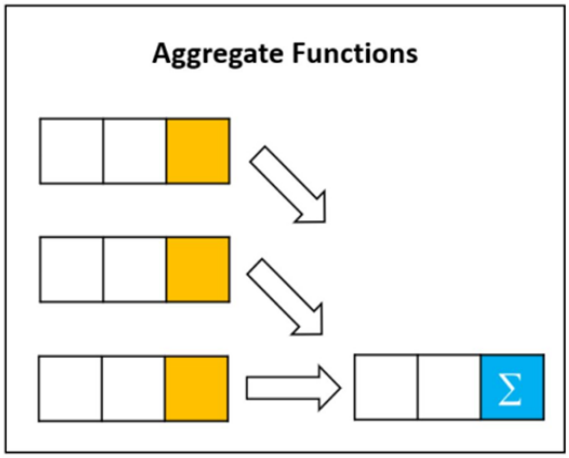
```sql
#Example: Lets find the customers who have not placed any orders.

SELECT c.CustomerName
FROM Customers c
WHERE NOT EXISTS (
    SELECT 1
    FROM Orders o
    WHERE o.CustomerID = c.CustomerID
);
```

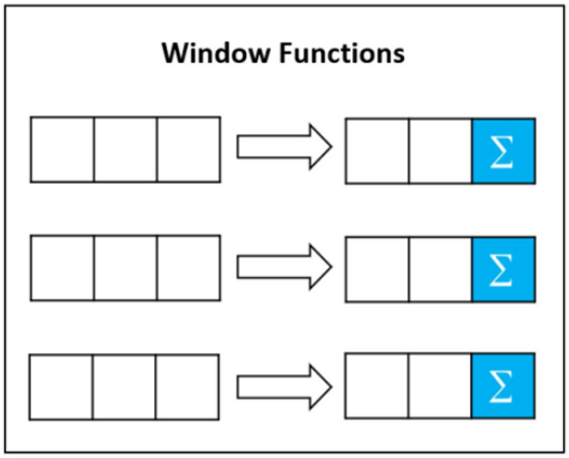
```
+---------------+
| CustomerName  |
+---------------+
| Charlie Brown |
+---------------+
```

Window Functions

Window Functions In SQL

- **Window functions:** These are special SQL functions that perform a calculation across a set of related rows.
- **How it works**: Instead of operating on individual rows, a window function operates on a group or **'window'** of rows that are somehow related to the current row. This allows for complex calculations based on these related rows.
- **Window definition**: The **'window'** in window functions refers to a **set of rows**. The window can be defined using different criteria depending on the requirements of your operation.
- **Partitions**: By using the **PARTITION BY** clause, you can divide your data into smaller sets or **'partitions'**. The window function will then be applied individually to each partition.
- **Order of rows**: You can specify the order of rows in each partition using the ORDER BY clause. This order influences how some window functions calculate their result.
- **Frames**: The **ROWS/RANGE** clause lets you further narrow down the window by defining a **'frame'** or subset of rows within each partition.
- **Comparison with Aggregate Functions:** Unlike aggregate functions that return a **single result per group**, window functions return a **single result for each row** of the table based on the group of rows defined in the window.
- **Advantage**: Window functions allow for **more complex operations** that need to take into account not just the current row, but also its **'neighbours'** in some way.





Window Function Syntax

```
function_name (column) OVER (
   [PARTITION BY column_name_1, ..., column_name_n]
   [ORDER BY column_name_1 [ASC | DESC], ..., column_name_n [ASC | DESC]]
)
```

- **function_name**: This is the window function you want to use. Examples include ROW_NUMBER(), RANK(), DENSE_RANK(), SUM(), AVG(), and many others.
- (**column**): This is the column that the window function will operate on. For some functions like SUM(salary)
- **OVER** (): This is where you define the window. The parentheses after OVER contain the specifications for the window.
- **PARTITION BY column_name_1, ..., column_name_n**: This clause divides the result set into partitions upon which the window function will operate independently. For example, if you have PARTITION BY salesperson_id, the window function will calculate a result for each salesperson independently.
- **ORDER BY column_name_1 [ASC | DESC], ..., column_name_n [ASC | DESC]**: This clause specifies the order of the rows in each partition. The window function operates on these rows in the order specified. For example, ORDER BY sales_date DESC will make the window function operate on rows with more recent dates first.

Common Window Functions

1. `` `ROW_NUMBER()` ``: Assigns a unique sequential integer to rows within a partition of a result set.
2. `` `RANK()` ``: Assigns a rank to each row within a partition of a result set, with gaps for tied values.
3. `` `DENSE_RANK()` ``: Similar to `` `RANK()` ``, but without gaps for tied values.
4. `` `NTILE(n)` ``: Divides the rows in a partition into `` `n` `` buckets and assigns a bucket number to each row.
5. **Aggregate Functions** (`` `SUM()` ``, `` `AVG()` ``, `` `COUNT()` ``, etc.): Performs calculations over a set of rows.

```sql
create table shop_sales_data
(
sales_date date,
shop_id varchar(5),
sales_amount int
);

insert into shop_sales_data values('2022-02-14','S1',200);
insert into shop_sales_data values('2022-02-15','S1',300);
insert into shop_sales_data values('2022-02-14','S2',600);
insert into shop_sales_data values('2022-02-15','S3',500);
insert into shop_sales_data values('2022-02-18','S1',400);
insert into shop_sales_data values('2022-02-17','S2',250);
insert into shop_sales_data values('2022-02-20','S3',300);

select*from shop_sales_data;

# Total count of sales for each shop using window function
# Working functions - SUM(), MIN(), MAX(), COUNT(), AVG()

# If we only use Order by In Over Clause
select *,
       sum(sales_amount) over(order by sales_amount desc) as running_sum_of_sales
from shop_sales_data;

# If we only use Partition By
select *,
       sum(sales_amount) over(partition by shop_id) as total_sum_of_sales
from shop_sales_data;

# If we only use Partition By & Order By together
select *,
       sum(sales_amount) over(partition by shop_id order by sales_amount desc) as running_sum_of_sales
from shop_sales_data;

select *,
       sum(sales_amount) over(partition by shop_id order by sales_date desc)
       as running_sum_of_sales,
       avg(sales_amount) over(partition by shop_id order by sales_date desc)
       as running_avg_of_sales,
       max(sales_amount) over(partition by shop_id order by sales_date desc)
       as running_max_of_sales,
       min(sales_amount) over(partition by shop_id order by sales_date desc)
       as running_min_of_sales
from shop_sales_data;
```

Output:

sales_date	shop_id	sales_amount
2022-02-14	S1	200
2022-02-15	S1	300
2022-02-14	S2	600
2022-02-15	S3	500
2022-02-18	S1	400
2022-02-17	S2	250
2022-02-20	S3	300

sales_date	shop_id	sales_amount	running_sum_of_sales
2022-02-14	S2	600	600
2022-02-15	S3	500	1100
2022-02-18	S1	400	1500
2022-02-15	S1	300	2100
2022-02-20	S3	300	2100
2022-02-17	S2	250	2350
2022-02-14	S1	200	2550

sales_date	shop_id	sales_amount	total_sum_of_sales
2022-02-14	S1	200	900
2022-02-15	S1	300	900
2022-02-18	S1	400	900
2022-02-14	S2	600	850
2022-02-17	S2	250	850
2022-02-15	S3	500	800
2022-02-20	S3	300	800

sales_date	shop_id	sales_amount	running_sum_of_sales
2022-02-18	S1	400	400
2022-02-15	S1	300	700
2022-02-14	S1	200	900
2022-02-14	S2	600	600
2022-02-17	S2	250	850
2022-02-15	S3	500	500
2022-02-20	S3	300	800

sales_date	shop_id	sales_amount	running_sum_of_sales	running_avg_of_sales	running_max_of_sales	running_min_of_sales
2022-02-18	S1	400	400	400.0000	400	400
2022-02-15	S1	300	700	350.0000	400	300
2022-02-14	S1	200	900	300.0000	400	200
2022-02-17	S2	250	250	250.0000	250	250
2022-02-14	S2	600	850	425.0000	600	250
2022-02-20	S3	300	300	300.0000	300	300
2022-02-15	S3	500	800	400.0000	500	300

Running Average (`running_avg_of_sales`)

Explanation:

- **Formula:** `AVG()` calculates the average of `sales_amount` for each shop, considering the rows up to and including the current row as defined by the `ORDER BY` clause.
- **Window:** Defined by `PARTITION BY shop_id ORDER BY sales_date DESC`.

Calculation Steps:

1. For each row within the same `shop_id`, it calculates the average of `sales_amount` up to the current row when ordered by `sales_date` descending.
2. The `ORDER BY sales_date DESC` ensures that later dates are considered first, thus creating a cumulative average for rows with the same shop id.

Example for Shop S1:

- **2022-02-18 (400):** Only this row considered. Average = `400.00`.
- **2022-02-15 (300):** Average of `[400, 300]` = `(400 + 300) / 2 = 350.00`.
- **2022-02-14 (200):** Average of `[400, 300, 200]` = `(400 + 300 + 200) / 3 = 300.00`.

Same for : running_max_of_sales, and running_min_of_sales

```sql
create table amazon_sales_data
(
    sales_date date,
    sales_amount int
);

insert into amazon_sales_data values('2022-08-21',500);
insert into amazon_sales_data values('2022-08-22',600);
insert into amazon_sales_data values('2022-08-19',300);

insert into amazon_sales_data values('2022-08-18',200);

insert into amazon_sales_data values('2022-08-25',800);


# Query - Calculate the date wise rolling average of amazon sales
select * from amazon_sales_data;

select *,
       avg(sales_amount) over(order by sales_date) as rolling_avg
from amazon_sales_data;

select *,
       avg(sales_amount) over(order by sales_date) as rolling_avg,
       sum(sales_amount) over(order by sales_date) as rolling_sum
from amazon_sales_data;
```

Output:

```
+------------+--------------+
| sales_date | sales_amount |
+------------+--------------+
| 2022-08-21 |          500 |
| 2022-08-22 |          600 |
| 2022-08-19 |          300 |
| 2022-08-18 |          200 |
| 2022-08-25 |          800 |
+------------+--------------+
+------------+--------------+-------------+
| sales_date | sales_amount | rolling_avg |
+------------+--------------+-------------+
| 2022-08-18 |          200 |    200.0000 |
| 2022-08-19 |          300 |    250.0000 |
| 2022-08-21 |          500 |    333.3333 |
| 2022-08-22 |          600 |    400.0000 |
| 2022-08-25 |          800 |    480.0000 |
+------------+--------------+-------------+
+------------+--------------+-------------+-------------+
| sales_date | sales_amount | rolling_avg | rolling_sum |
+------------+--------------+-------------+-------------+
| 2022-08-18 |          200 |    200.0000 |         200 |
| 2022-08-19 |          300 |    250.0000 |         500 |
| 2022-08-21 |          500 |    333.3333 |        1000 |
| 2022-08-22 |          600 |    400.0000 |        1600 |
| 2022-08-25 |          800 |    480.0000 |        2400 |
+------------+--------------+-------------+-------------+
```

Rank(), Row_Number(), Dense_Rank() window functions

There are three main categories of window functions in SQL: **Ranking functions**, **Value functions**, and **Aggregate functions**. Here's a brief description and example for each:

Ranking Functions:

- **ROW_NUMBER()**: Assigns a unique row number to each row, ranking start from 1 and keep increasing till the end of last row

```sql
SELECT Studentname,
       Subject,
       Marks,
       ROW_NUMBER() OVER(ORDER BY Marks desc)
RowNumber
FROM ExamResult;
```

	Studentname	Subject	Marks	RowNumber
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	5
6	Lily	Maths	65	6
7	Olivia	Science	60	7
8	Olivia	Maths	55	8
9	Isabella	Maths	50	9

- **RANK()**: Assigns a rank to each row. Rows with equal values receive the same rank, with the next row receiving a rank which skips the duplicate rankings.

```sql
SELECT Studentname,
       Subject,
       Marks,
       RANK() OVER(ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Rank;
```

Results | Messages

	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	4
6	Lily	Maths	65	6
7	Olivia	Science	60	7
8	Olivia	Maths	55	8
9	Isabella	Maths	50	9

- **DENSE_RANK()**: Similar to RANK(), but does not skip rankings if there are duplicates.

```
SELECT Studentname,
       Subject,
       Marks,
       DENSE_RANK() OVER(ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Rank;
```

	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	4
6	Lily	Maths	65	5
7	Olivia	Science	60	6
8	Olivia	Maths	55	7
9	Isabella	Maths	50	8

Similar Rank

```
create table shop_sales_data
(
sales_date date,
shop_id varchar(5),
sales_amount int
);

insert into shop_sales_data values('2022-02-14','S1',200);
insert into shop_sales_data values('2022-02-15','S1',300);
insert into shop_sales_data values('2022-02-14','S2',600);
insert into shop_sales_data values('2022-02-15','S3',500);
insert into shop_sales_data values('2022-02-18','S1',400);
insert into shop_sales_data values('2022-02-17','S2',250);
insert into shop_sales_data values('2022-02-20','S3',300);
insert into shop_sales_data values('2022-02-19','S1',400);
insert into shop_sales_data values('2022-02-20','S1',400);
insert into shop_sales_data values('2022-02-22','S1',300);
insert into shop_sales_data values('2022-02-25','S1',200);
insert into shop_sales_data values('2022-02-15','S2',600);
insert into shop_sales_data values('2022-02-16','S2',600);
insert into shop_sales_data values('2022-02-16','S3',500);
insert into shop_sales_data values('2022-02-18','S3',500);
insert into shop_sales_data values('2022-02-19','S3',300);

select *,
       row_number() over(partition by shop_id order by sales_amount desc) as row_num,

       rank() over(partition by shop_id order by sales_amount desc) as rank_val,

       dense_rank() over(partition by shop_id order by sales_amount desc) as dense_rank_val
from shop_sales_data;
```

Output:

sales_date	shop_id	sales_amount	row_num	rank_val	dense_rank_val
2022-02-18	S1	400	1	1	1
2022-02-19	S1	400	2	1	1
2022-02-20	S1	400	3	1	1
2022-02-15	S1	300	4	4	2
2022-02-22	S1	300	5	4	2
2022-02-14	S1	200	6	6	3
2022-02-25	S1	200	7	6	3
2022-02-14	S2	600	1	1	1
2022-02-15	S2	600	2	1	1
2022-02-16	S2	600	3	1	1
2022-02-17	S2	250	4	4	2
2022-02-15	S3	500	1	1	1
2022-02-16	S3	500	2	1	1
2022-02-18	S3	500	3	1	1
2022-02-20	S3	300	4	4	2
2022-02-19	S3	300	5	4	2

```sql
create table employees
(
    emp_id int,
    salary int,
    dept_name VARCHAR(30)

);

insert into employees values(1,10000,'Software');
insert into employees values(2,11000,'Software');
insert into employees values(3,11000,'Software');
insert into employees values(4,11000,'Software');
insert into employees values(5,15000,'Finance');
insert into employees values(6,15000,'Finance');
insert into employees values(7,15000,'IT');
insert into employees values(8,12000,'HR');
insert into employees values(9,12000,'HR');
insert into employees values(10,11000,'HR');
```

```
+--------+--------+-----------+
| emp_id | salary | dept_name |
+--------+--------+-----------+
|      1 |  10000 | Software  |
|      2 |  11000 | Software  |
|      3 |  11000 | Software  |
|      4 |  11000 | Software  |
|      5 |  15000 | Finance   |
|      6 |  15000 | Finance   |
|      7 |  15000 | IT        |
|      8 |  12000 | HR        |
|      9 |  12000 | HR        |
|     10 |  11000 | HR        |
+--------+--------+-----------+
```

```sql
#-----------------------------------------------------------------------------------------------------------------------

# Query - get one employee from each department who is getting maximum salary (employee can be random if salary is same)

select
    tmp.*
from (select *,
        row_number() over(partition by dept_name order by salary desc) as row_num
    from employees) tmp
where tmp.row_num = 1;

#-----------------------------------------------------------------------------------------------------------------------

# Query - get one employee from each department who is getting maximum salary (employee can be random if salary is same)

select
    tmp.*
from (select *,
        row_number() over(partition by dept_name order by salary desc) as row_num
    from employees) tmp
where tmp.row_num = 1;

#-----------------------------------------------------------------------------------------------------------------------

# Query - get all employees from each department who are getting maximum salary
select
    tmp.*
from (select *,
        rank() over(partition by dept_name order by salary desc) as rank_num
    from employees) tmp
where tmp.rank_num = 1;

#-----------------------------------------------------------------------------------------------------------------------

# Query - get all top 2 ranked employees from each department who are getting maximum salary
select
    tmp.*
from (select *,
        dense_rank() over(partition by dept_name order by salary desc) as dense_rank_num
    from employees) tmp
where tmp.dense_rank_num <= 2;
```

Output:

```
+--------+--------+-----------+----------+
| emp_id | salary | dept_name | row_num |
+--------+--------+-----------+----------+
|      5 |  15000 | Finance   |        1 |
|      8 |  12000 | HR        |        1 |
|      7 |  15000 | IT        |        1 |
|      2 |  11000 | Software  |        1 |
+--------+--------+-----------+----------+
```

```
+--------+--------+-----------+----------+
| emp_id | salary | dept_name | row_num |
+--------+--------+-----------+----------+
|      5 |  15000 | Finance   |        1 |
|      8 |  12000 | HR        |        1 |
|      7 |  15000 | IT        |        1 |
|      2 |  11000 | Software  |        1 |
+--------+--------+-----------+----------+
```

```
+--------+--------+-----------+----------+
| emp_id | salary | dept_name | rank_num |
+--------+--------+-----------+----------+
|      5 |  15000 | Finance   |        1 |
|      6 |  15000 | Finance   |        1 |
|      8 |  12000 | HR        |        1 |
|      9 |  12000 | HR        |        1 |
|      7 |  15000 | IT        |        1 |
|      2 |  11000 | Software  |        1 |
|      3 |  11000 | Software  |        1 |
|      4 |  11000 | Software  |        1 |
+--------+--------+-----------+----------+
```

```
+--------+--------+-----------+----------------+
| emp_id | salary | dept_name | dense_rank_num |
+--------+--------+-----------+----------------+
|      5 |  15000 | Finance   |              1 |
|      6 |  15000 | Finance   |              1 |
|      8 |  12000 | HR        |              1 |
|      9 |  12000 | HR        |              1 |
|     10 |  11000 | HR        |              2 |
|      7 |  15000 | IT        |              1 |
|      2 |  11000 | Software  |              1 |
|      3 |  11000 | Software  |              1 |
|      4 |  11000 | Software  |              1 |
|      1 |  10000 | Software  |              2 |
+--------+--------+-----------+----------------+
```

Value Functions: These functions perform calculations on the values of the window rows.

- **FIRST_VALUE():** Returns the first value in the window.

```
SELECT
   employee_name,
   department,
   hours,
   FIRST_VALUE(employee_name) OVER (
      PARTITION BY department
      ORDER BY hours
   ) least_over_time
FROM
   overtime;
```

employee_name	department	hours	least_over_time
Diane Murphy	Accounting	37	Diane Murphy
Jeff Firrelli	Accounting	40	Diane Murphy
Mary Patterson	Accounting	74	Diane Murphy
Gerard Bondur	Finance	47	Gerard Bondur
William Patterson	Finance	58	Gerard Bondur
Anthony Bow	Finance	66	Gerard Bondur
Leslie Thompson	IT	88	Leslie Thompson
Leslie Jennings	IT	90	Leslie Thompson
Loui Bondur	Marketing	49	Loui Bondur
Gerard Hernandez	Marketing	66	Loui Bondur
George Vanauf	Marketing	89	Loui Bondur
Steve Patterson	Sales	29	Steve Patterson
Foon Yue Tseng	Sales	65	Steve Patterson
Julie Firrelli	Sales	81	Steve Patterson
Barry Jones	SCM	65	Barry Jones
Pamela Castillo	SCM	96	Barry Jones
Larry Bott	SCM	100	Barry Jones

- **LAST_VALUE():** Returns the last value in the window.

```
SELECT employee_name, department,salary,
LAST_VALUE(employee_name)
 OVER (
    PARTITION BY department ORDER BY
salary
    ) as max_salary
FROM Employee;
```

employee_name	department	salary	max_salary
Vishal	Accounting	40000	Ravi
Ravi	Accounting	60000	Ravi
Nilesh	Finance	55000	Abdul
Sushant	Finance	65000	Abdul
Abdul	Finance	68000	Abdul
Jai	IT	45000	Mohit
Aman	IT	60000	Mohit
Mohit	IT	70000	Mohit

Example for lag and lead function

- **LAG():** Returns the value of the previous row.

```
SELECT
      Year,
      Quarter,
      Sales,
      LAG(Sales, 1, 0) OVER(
        PARTITION BY Year
        ORDER BY Year,Quarter ASC)
      AS NextQuarterSales
FROM ProductSales;
```

	Year	Quarter	Sales	NextQuarterSales
1	2017	1	55000.00	0.00
2	2017	2	78000.00	55000.00
3	2017	3	49000.00	78000.00
4	2017	4	32000.00	49000.00
5	2018	1	41000.00	0.00
6	2018	2	8965.00	41000.00
7	2018	3	69874.00	8965.00
8	2018	4	32562.00	69874.00
9	2019	1	87456.00	0.00
10	2019	2	75000.00	87456.00
11	2019	3	96500.00	75000.00
12	2019	4	85236.00	96500.00

- **LEAD():** Returns the value of the next row.

```
SELECT Year,
      Quarter,
      Sales,
      LEAD(Sales, 1, 0) OVER(
      PARTITION BY Year
      ORDER BY Year,Quarter ASC)
      AS NextQuarterSales
FROM ProductSales;
```

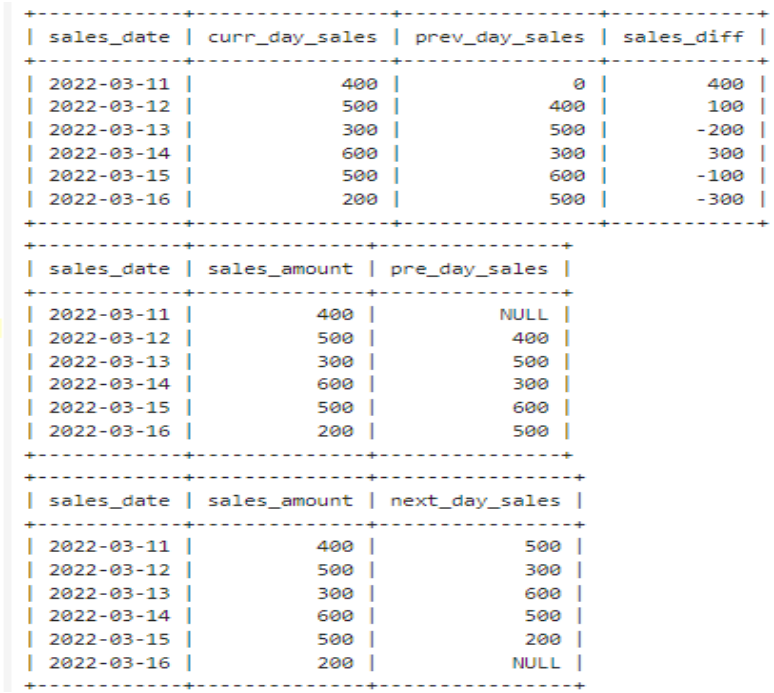
	Year	Quarter	Sales	NextQuarterSales
1	2017	1	55000.00	78000.00
2	2017	2	78000.00	49000.00
3	2017	3	49000.00	32000.00
4	2017	4	32000.00	0.00
5	2018	1	41000.00	8965.00
6	2018	2	8965.00	69874.00
7	2018	3	69874.00	32562.00
8	2018	4	32562.00	0.00
9	2019	1	87456.00	75000.00
10	2019	2	75000.00	96500.00
11	2019	3	96500.00	85236.00
12	2019	4	85236.00	0.00

Lead function on PARTITION for Year column

```sql
create table daily_sales
(
sales_date date,
sales_amount int
);


insert into daily_sales values('2022-03-11',400);
insert into daily_sales values('2022-03-12',500);
insert into daily_sales values('2022-03-13',300);
insert into daily_sales values('2022-03-14',600);
insert into daily_sales values('2022-03-15',500);
insert into daily_sales values('2022-03-16',200);

select * from daily_sales;

select *,
      lag(sales_amount, 1) over(order by sales_date) as pre_day_sales
from daily_sales;

# we can use this to replace null with defualt value like 0
select *,
  coalesce(lag(sales_amount,1) over(order by sales_date), 0) as prev_sales
from daily_sales;

# Query - Calculate the differnce of sales with previous day sales
# Here null will be derived
select sales_date,
       sales_amount as curr_day_sales,
       lag(sales_amount, 1) over(order by sales_date) as prev_day_sales,
       sales_amount - lag(sales_amount, 1) over(order by sales_date) as sales_diff
from daily_sales;

# Here we can replace null with 0
select sales_date,
       sales_amount as curr_day_sales,
       lag(sales_amount, 1, 0) over(order by sales_date) as prev_day_sales,
       sales_amount - lag(sales_amount, 1, 0) over(order by sales_date) as sales_diff
from daily_sales;

# Diff between lead and lag
select *,
      lag(sales_amount, 1) over(order by sales_date) as pre_day_sales
from daily_sales;

select *,
      lead(sales_amount, 1) over(order by sales_date) as next_day_sales
from daily_sales;
```

Output:

```
+------------+--------------+
| sales_date | sales_amount |
+------------+--------------+
| 2022-03-11 |          400 |
| 2022-03-12 |          500 |
| 2022-03-13 |          300 |
| 2022-03-14 |          600 |
| 2022-03-15 |          500 |
| 2022-03-16 |          200 |
+------------+--------------+
+------------+--------------+---------------+
| sales_date | sales_amount | pre_day_sales |
+------------+--------------+---------------+
| 2022-03-11 |          400 |          NULL |
| 2022-03-12 |          500 |           400 |
| 2022-03-13 |          300 |           500 |
| 2022-03-14 |          600 |           300 |
| 2022-03-15 |          500 |           600 |
| 2022-03-16 |          200 |           500 |
+------------+--------------+---------------+
+------------+--------------+------------+
| sales_date | sales_amount | prev_sales |
+------------+--------------+------------+
| 2022-03-11 |          400 |          0 |
| 2022-03-12 |          500 |        400 |
| 2022-03-13 |          300 |        500 |
| 2022-03-14 |          600 |        300 |
| 2022-03-15 |          500 |        600 |
| 2022-03-16 |          200 |        500 |
+------------+--------------+------------+
+------------+----------------+----------------+------------+
| sales_date | curr_day_sales | prev_day_sales | sales_diff |
+------------+----------------+----------------+------------+
| 2022-03-11 |            400 |           NULL |       NULL |
| 2022-03-12 |            500 |            400 |        100 |
| 2022-03-13 |            300 |            500 |       -200 |
| 2022-03-14 |            600 |            300 |        300 |
| 2022-03-15 |            500 |            600 |       -100 |
| 2022-03-16 |            200 |            500 |       -300 |
+------------+----------------+----------------+------------+
```



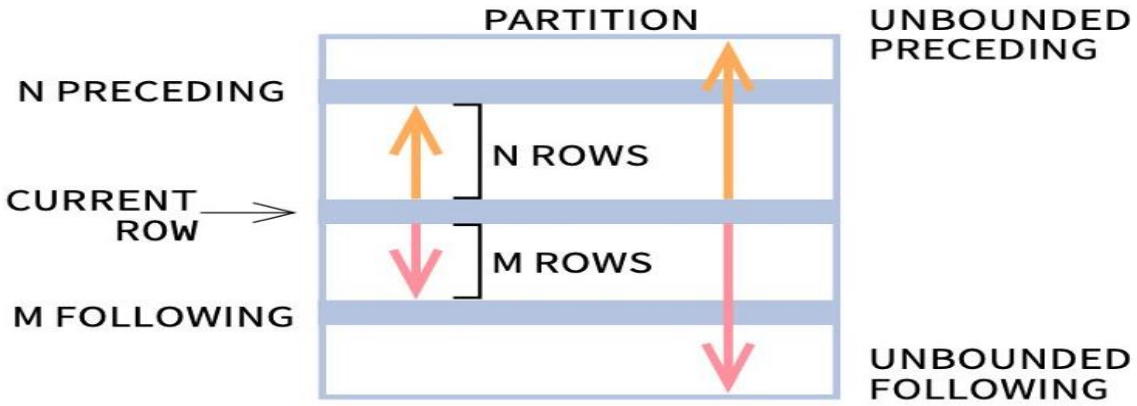
```
+------------+----------------+----------------+------------+
| sales_date | curr_day_sales | prev_day_sales | sales_diff |
+------------+----------------+----------------+------------+
| 2022-03-11 |            400 |              0 |        400 |
| 2022-03-12 |            500 |            400 |        100 |
| 2022-03-13 |            300 |            500 |       -200 |
| 2022-03-14 |            600 |            300 |        300 |
| 2022-03-15 |            500 |            600 |       -100 |
| 2022-03-16 |            200 |            500 |       -300 |
+------------+----------------+----------------+------------+
+------------+--------------+---------------+
| sales_date | sales_amount | pre_day_sales |
+------------+--------------+---------------+
| 2022-03-11 |          400 |          NULL |
| 2022-03-12 |          500 |           400 |
| 2022-03-13 |          300 |           500 |
| 2022-03-14 |          600 |           300 |
| 2022-03-15 |          500 |           600 |
| 2022-03-16 |          200 |           500 |
+------------+--------------+---------------+
+------------+--------------+----------------+
| sales_date | sales_amount | next_day_sales |
+------------+--------------+----------------+
| 2022-03-11 |          400 |            500 |
| 2022-03-12 |          500 |            300 |
| 2022-03-13 |          300 |            600 |
| 2022-03-14 |          600 |            500 |
| 2022-03-15 |          500 |            200 |
| 2022-03-16 |          200 |           NULL |
+------------+--------------+----------------+
```

Frame Clause in Window Functions

- The frame clause in window functions defines the **subset of rows ('frame')** used for calculating the result of the function for the current row.
- It's specified within the OVER() clause after PARTITION BY and ORDER BY.
- The frame is defined by two parts: a **start** and an **end**, each relative to the **current row**.
- Generic syntax for a window function with a frame clause:

```
function_name (expression) OVER (
  [PARTITION BY column_name_1, ..., column_name_n]
  [ORDER BY column_name_1 [ASC | DESC], ..., column_name_n [ASC | DESC]]
  [ROWS|RANGE frame_start TO frame_end]
)
```

- The frame start can be:
  - UNBOUNDED PRECEDING (starts at the first row of the partition)
  - N PRECEDING (starts N rows before the current row)
  - CURRENT ROW (starts at the current row)
- The frame end can be:
  - UNBOUNDED FOLLOWING (ends at the last row of the partition)
  - N FOLLOWING (ends N rows after the current row)
  - CURRENT ROW (ends at the current row)
- For **ROWS**, the frame consists of N rows coming before or after the current row.
- For **RANGE**, the frame consists of rows within a certain value range relative to the value in the current row.



ROWS BETWEEN Example

```
SELECT date, revenue,
   SUM(revenue) OVER (
     ORDER BY date
     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) running_total
FROM sales
ORDER BY date;
```

Input Table

sales		
record_id	date	revenue
1	2021-09-01	1515.45
2	2021-09-02	2345.35
3	2021-09-03	903.99
4	2021-09-04	2158.55
5	2021-09-05	1819.80

Output Table

date	revenue	running_total
2021-09-01	1515.45	1515.45
2021-09-02	2345.35	3860.80
2021-09-03	903.99	4764.79
2021-09-04	2158.55	6923.34
2021-09-05	1819.80	8743.14

RANGE BETWEEN Example

```
SELECT
  shop,
  date,
  revenue_amount,
  MAX(revenue_amount) OVER (
    ORDER BY DATE
    RANGE BETWEEN INTERVAL '3' DAY PRECEDING
    AND INTERVAL '1' DAY FOLLOWING
  ) AS max_revenue
FROM revenue_per_shop;
```

shop	date	revenue_amount	max_revenue
Shop 1	2021-05-01	12,573.25	18,847.54
Shop 2	2021-05-01	11,348.22	18,847.54
Shop 1	2021-05-02	14,388.14	18,847.54
Shop 2	2021-05-02	18,847.54	18,847.54
Shop 1	2021-05-03	9,845.29	18,847.54
Shop 2	2021-05-03	14,574.56	18,847.54
Shop 1	2021-05-04	11,500.63	18,847.54
Shop 2	2021-05-04	16,897.21	18,847.54
Shop 1	2021-05-05	9,634.56	21,489.22
Shop 2	2021-05-05	14,255.87	21,489.22
Shop 1	2021-05-06	11,248.33	21,489.22
Shop 2	2021-05-06	21,489.22	21,489.22
Shop 2	2021-05-07	15,517.22	21,489.22
Shop 1	2021-05-07	14,448.65	21,489.22

Output Table

```sql
create table daily_sales
(
sales_date date,
sales_amount int
);


insert into daily_sales values('2022-03-11',400);
insert into daily_sales values('2022-03-12',500);
insert into daily_sales values('2022-03-13',300);
insert into daily_sales values('2022-03-14',600);
insert into daily_sales values('2022-03-15',500);
insert into daily_sales values('2022-03-16',200);

select * from daily_sales;

select *,
	sum(sales_amount) over(order by sales_date rows between 1 preceding and 1 following) as prev_plus_next_sales_sum
from daily_sales;

select *,
	sum(sales_amount) over(order by sales_date rows between 1 preceding and current row) as prev_plus_next_sales_sum
from daily_sales;

select *,
	sum(sales_amount) over(order by sales_date rows between current row and 1 following) as prev_plus_next_sales_sum
from daily_sales;

select *,
	sum(sales_amount) over(order by sales_date rows between 2 preceding and 1 following) as prev_plus_next_sales_sum
from daily_sales;

select *,
	sum(sales_amount) over(order by sales_date rows between unbounded preceding and current row) as prev_plus_next_sales_sum
from daily_sales;

select *,
	sum(sales_amount) over(order by sales_date rows between current row and unbounded following) as prev_plus_next_sales_sum
from daily_sales;

select *,
	sum(sales_amount) over(order by sales_date rows between unbounded preceding and unbounded following) as prev_plus_next_sales_sum
from daily_sales;
```

Output:

sales_date	sales_amount
2022-03-11	400
2022-03-12	500
2022-03-13	300
2022-03-14	600
2022-03-15	500
2022-03-16	200

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-11	400	900
2022-03-12	500	1200
2022-03-13	300	1400
2022-03-14	600	1400
2022-03-15	500	1300
2022-03-16	200	700

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-11	400	400
2022-03-12	500	900
2022-03-13	300	800
2022-03-14	600	900
2022-03-15	500	1100
2022-03-16	200	700

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-11	400	900
2022-03-12	500	800
2022-03-13	300	900
2022-03-14	600	1100
2022-03-15	500	700
2022-03-16	200	200

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-11	400	900
2022-03-12	500	1200
2022-03-13	300	1800
2022-03-14	600	1900
2022-03-15	500	1600
2022-03-16	200	1300

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-11	400	400
2022-03-12	500	900
2022-03-13	300	1200
2022-03-14	600	1800
2022-03-15	500	2300
2022-03-16	200	2500

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-11	400	2500
2022-03-12	500	2100
2022-03-13	300	1600
2022-03-14	600	1300
2022-03-15	500	700
2022-03-16	200	200

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-11	400	2500
2022-03-12	500	2500
2022-03-13	300	2500
2022-03-14	600	2500
2022-03-15	500	2500
2022-03-16	200	2500

```
# Alternate way to esclude computation of current row
select *,
    sum(sales_amount) over(order by sales_date rows between unbounded preceding and unbounded following) - sales_amount as prev_plus_next_sales_sum
from daily_sales;

# How to work with Range Between ( here 100 and 200 are maximum difference range in which we take row only )

select *,
    sum(sales_amount) over(order by sales_amount range between 100 preceding and 200 following) as prev_plus_next_sales_sum
from daily_sales;

# Calculate the running sum for a week
# Calculate the running sum for a month
insert into daily_sales values('2022-03-20',900);
insert into daily_sales values('2022-03-23',200);
insert into daily_sales values('2022-03-25',300);
insert into daily_sales values('2022-03-29',250);


select * from daily_sales;

select *,
    sum(sales_amount) over(order by sales_date range between interval '6' day preceding and current row) as running_weekly_sum
from daily_sales;
```

Output:

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-11	400	2100
2022-03-12	500	2000
2022-03-13	300	2200
2022-03-14	600	1900
2022-03-15	500	2000
2022-03-16	200	2300

sales_date	sales_amount	prev_plus_next_sales_sum
2022-03-16	200	900
2022-03-13	300	1900
2022-03-11	400	2300
2022-03-12	500	2000
2022-03-15	500	2000
2022-03-14	600	1600

sales_date	sales_amount
2022-03-11	400
2022-03-12	500
2022-03-13	300
2022-03-14	600
2022-03-15	500
2022-03-16	200
2022-03-20	900
2022-03-23	200
2022-03-25	300
2022-03-29	250

sales_date	sales_amount	running_weekly_sum
2022-03-11	400	400
2022-03-12	500	900
2022-03-13	300	1200
2022-03-14	600	1800
2022-03-15	500	2300
2022-03-16	200	2500
2022-03-20	900	2200
2022-03-23	200	1100
2022-03-25	300	1400
2022-03-29	250	750

Common Table Expression

A Common Table Expression (CTE) in SQL is a named temporary result set that exists only within the execution scope of a single SQL statement. Here are some important points to note about CTEs:

- CTEs can be thought of as alternatives to derived tables, inline views, or subqueries.
- They can be used in SELECT, INSERT, UPDATE, or DELETE statements.
- CTEs help to simplify complex queries, particularly those involving multiple subqueries or recursive queries.
- They make your query more readable and easier to maintain.
- A CTE is defined using the **WITH keyword,** followed by the CTE name and a query. The CTE can then be referred to by its name elsewhere in the query.

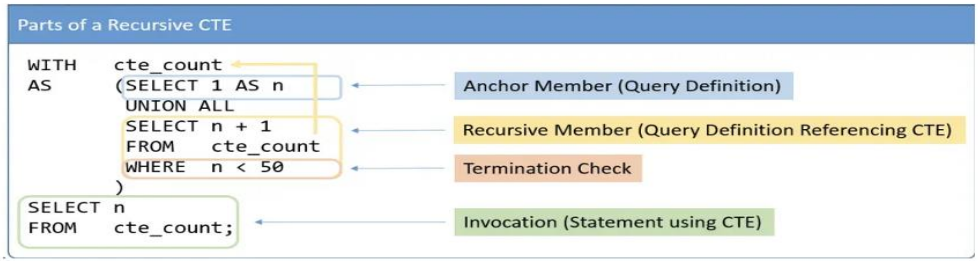
Here's a basic example of a CTE:

```
WITH sales_cte AS (
   SELECT sales_person, SUM(sales_amount) as total_sales
   FROM sales_table
   GROUP BY sales_person
)
SELECT sales_person, total_sales
FROM sales_cte
WHERE total_sales > 1000;
```

- **Recursive CTE:** This is a CTE that references itself. In other words, the CTE query definition refers back to the CTE name, creating a loop that ends when a certain condition is met. Recursive CTEs are useful for working with hierarchical or tree-structured data.

Example:

```
WITH RECURSIVE number_sequence AS (
    SELECT 1 AS number
    UNION ALL
    SELECT number + 1
    FROM number_sequence
    WHERE number < 10
)
SELECT * FROM number_sequence;
```



Common Table Expression (CTE) in MySQL

Common Table Expressions (CTEs) in MySQL allow you to define temporary result sets that can be referenced within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. CTEs are similar to subqueries but are more readable and reusable. They are particularly useful for breaking down complex queries into simpler parts or when recursive queries are required.

Basic Structure of a CTE

A CTE is defined using the `WITH` clause, followed by a query that defines the result set. The general syntax is:

```sql
WITH cte_name AS (
    -- Your query goes here
)
SELECT * FROM cte_name;
```

Example 1: Department-wise Total Salary

Scenario: You have two tables, `amazon_employees` and `department`, and you want to calculate the total salary paid in each department.

Tables:

- `amazon_employees`: Contains employee details including their department ID and salary.
- `department`: Contains department details like department ID and name.

Normal Query:

```sql
SELECT d.dept_name, tmp.total_salary
FROM (SELECT dept_id, SUM(salary) AS total_salary
      FROM amazon_employees
      GROUP BY dept_id) tmp
INNER JOIN department d ON tmp.dept_id = d.dept_id;
```

Using CTE:

```sql
WITH dept_wise_salary AS (
    SELECT dept_id, SUM(salary) AS total_salary
    FROM amazon_employees
    GROUP BY dept_id
)
SELECT d.dept_name, tmp.total_salary
FROM dept_wise_salary tmp
INNER JOIN department d ON tmp.dept_id = d.dept_id;
```

Explanation:

1. The CTE named `dept_wise_salary` computes the total salary for each department.
2. The main query joins this CTE with the `department` table to get the department names along with the total salary.

Example 2: Generating Numbers from 1 to 10 (Recursive CTE)

Scenario: You want to generate numbers from 1 to 10 in SQL.

Using Recursive CTE:

```sql
WITH RECURSIVE generate_numbers AS (
    SELECT 1 AS n
    UNION
    SELECT n + 1 FROM generate_numbers WHERE n < 10
)
SELECT * FROM generate_numbers;
```

Explanation:

1. The CTE `generate_numbers` starts with the number 1.
2. It then recursively adds 1 to the previous number until it reaches 10.
3. The final query selects all generated numbers.

Example 3: Employee Hierarchy for CTO

Scenario: Given an employee hierarchy, you want to present the organizational chart for the CTO (e.g., 'Asha').

Using Recursive CTE:

```sql
WITH RECURSIVE emp_hir AS (
    SELECT id, name, manager_id, designation
    FROM emp_mgr
    WHERE name = 'Asha'
    UNION
    SELECT em.id, em.name, em.manager_id, em.designation
    FROM emp_hir eh
    INNER JOIN emp_mgr em ON eh.id = em.manager_id
)
SELECT * FROM emp_hir;
```

Explanation:

1. The CTE `emp_hir` starts by selecting the CTO's information.
2. It then recursively finds all employees managed by the CTO, building the organizational chart.
3. The final query displays the hierarchy.

Example 4: Employee Hierarchy with Levels

Scenario: You want to include the level of employees in the hierarchy.

Using Recursive CTE with Levels:

```sql
WITH RECURSIVE emp_hir AS (
    SELECT id, name, manager_id, designation, 1 AS lvl
    FROM emp_mgr
    WHERE name = 'Asha'
    UNION
    SELECT em.id, em.name, em.manager_id, em.designation, eh.lvl + 1 AS lvl
    FROM emp_hir eh
    INNER JOIN emp_mgr em ON eh.id = em.manager_id
)
SELECT * FROM emp_hir;
```

Explanation:

1. The CTE `emp_hir` now includes a level (`lvl`) that starts at 1 for the CTO.
2. Each time a new level of employees is added, the level is incremented by 1.
3. The final query displays the hierarchy along with the level of each employee.

Subqueries in SQL

- **IN:** The IN operator allows you to specify multiple values in a WHERE clause. It returns true if a value matches any value in a list.

  SELECT * FROM Orders WHERE ProductName IN ('Apple', 'Banana');

- **NOT IN:** The NOT IN operator excludes the values in the list. It returns true if a value does not match any value in the list.

  SELECT * FROM Orders WHERE ProductName NOT IN ('Apple', 'Banana');

- **ANY:** The ANY operator returns true if any subquery value meets the condition.
- **ALL:** The ALL operator returns true if all subquery value meets the condition.
- **EXISTS:** The EXISTS operator returns true if the subquery returns one or more records.
- **NOT EXISTS:** The NOT EXISTS operator returns true if the subquery returns no records.