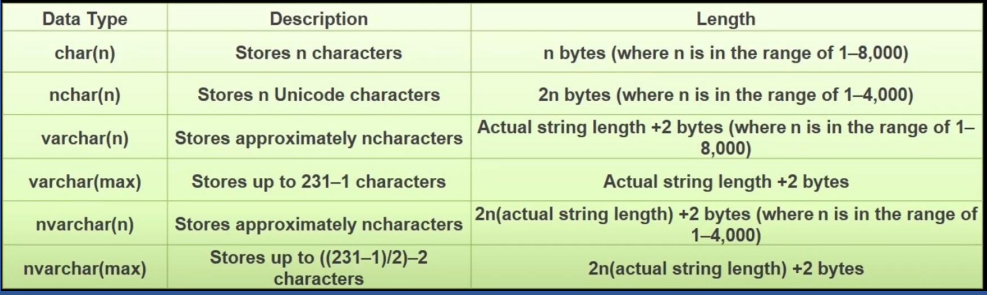
# Class -1:

1. **MySQL Data Type**

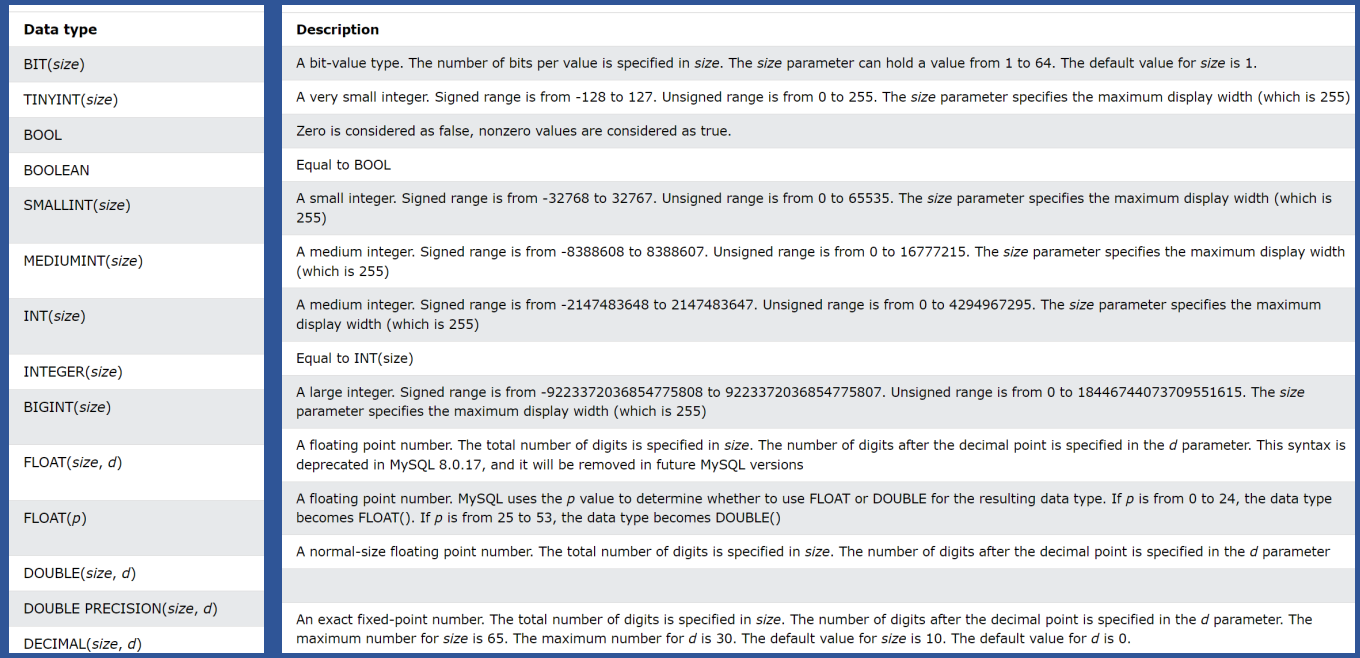
The data type of a column defines what value the column can hold: integer, character, money, date and time, binary, and so on.

In MySQL there are three main data types: string, numeric, and date and time.

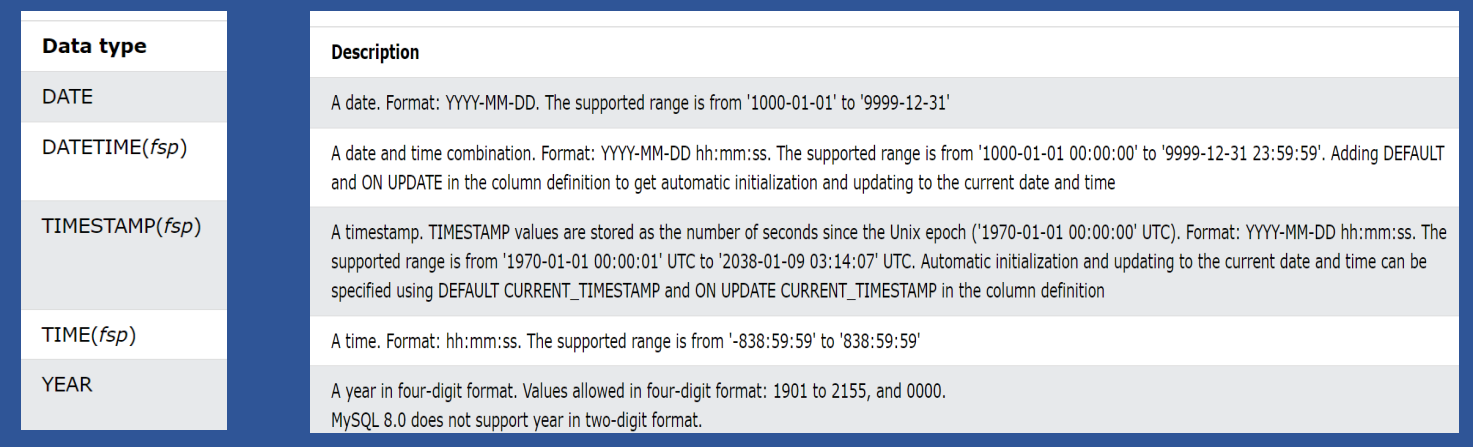
**String data types:**



**Numeric datatypes:**



**Date & time data types:**



1. **SQL Constraints**

SQL constraints / integrity 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.

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.

**Practical:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Primary and foreign key:**   |  |  |  |  | | --- | --- | --- | --- | | -- Primary key  **CREATE** **TABLE** Employee (  EmpID int **PRIMARY** **KEY**,  Firts\_name varchar(**50**),  last\_name varchar(**50**),  age int  );  -- Forgien key  **CREATE** **TABLE** orders(  OrderId int **PRIMARY** **KEY**,  CustomerID int,  OrderDate DATE,  **FOREIGN** **KEY** (CustomerID) **REFERENCES** Employee(EmpID)  );  **Try to insert null & duplicate values in primary key constraint column:**   |  | | --- | | **insert** **into** Employee(Firts\_name, last\_name, age)  **Values** ('Firts\_name', 'last\_name', **20**);  -- Error Code: 1364. Field 'EmpID' doesn't have a default value  **insert** **into** Employee(EmpID,Firts\_name, last\_name, age)  **Values** (**1**,'Firts\_name', 'last\_name', **20**);  -- Row inserted  **insert** **into** Employee(EmpID,Firts\_name, last\_name, age)  **Values** (**1**,'Firts\_name', 'last\_name', **20**);  -- Error Code: 1062. Duplicate entry '1' for key 'employee.PRIMARY' |   **Try to insert value in foreign key constraint column that not available in primary key constraint column:**   |  | | --- | | **select \* from Employee;**    **INSERT** **INTO** orders(OrderId, OrderDate)  **VALUES**(**001**,STR\_TO\_DATE('01/17/2025', '%m/%d/%Y'));  -- Inserted new record  **INSERT** **INTO** orders(OrderId, CustomerID, OrderDate)  **VALUES**(**002**,**3**,STR\_TO\_DATE('01/17/2025', '%m/%d/%Y'));  -- Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`class2nd`.`orders`, CONSTRAINT `orders\_ibfk\_1` FOREIGN KEY (`CustomerID`) REFERENCES `employee` (`EmpID`))  **INSERT** **INTO** orders(OrderId, CustomerID, OrderDate)  **VALUES**(**002**,**1**,STR\_TO\_DATE('01/17/2025', '%m/%d/%Y'));  -- Inserted new record |   **Not null , Unique , Check and default:**   |  | | --- | | **CREATE** **TABLE** Employee2(  Id int **NOT** **NULL** **UNIQUE**, -- Primary key  First\_name varchar(**50**) **NOT** **NULL**,  Middle\_name varchar(**50**),  Last\_name varchar(**50**) **NOT** **NULL**,  age INT **NOT** **NULL** **CHECK** (age>**18**),  City varchar(**100**) **DEFAULT** 'Hyderabad'  );  --  **INSERT** **INTO** Employee2(Id, First\_name, Last\_name, age, City)  **VALUES**(**1**,'First\_name1', 'Last\_name1', **19**, 'Mumbai');  -- Inserted new record  **INSERT** **INTO** Employee2(Id, First\_name, Last\_name, age)  **VALUES**(**2**,'First\_name2', 'Last\_name2', **33**);  -- Inserted new record with city as hyderbad  **INSERT** **INTO** Employee2(Id, First\_name, Last\_name, age)  **VALUES**(**3**,'First\_name3', 'Last\_name3', **13**);  -- Error Code: 3819. Check constraint 'employee2\_chk\_1' is violated.  **INSERT** **INTO** Employee2(Id, Last\_name, age)  **VALUES**(**3**, 'Last\_name3', **33**);  -- Error Code: 1364. Field 'First\_name' doesn't have a default value  ---- Check ------  **CREATE TABLE Student(**  **Id int Primary key,**  **First\_name varchar(50) NOT NULL,**  **Last\_name varchar(50) NOT NULL,**  **age INT ,**  **GRADE Char(1) CHECK (GRADE in ('O','A','B','C','D'))**  **);**  **INSERT INTO Student(Id, First\_name, Last\_name, age, GRADE)**  **VALUES (1,'First\_name', 'Last\_name', 12, 'O');**  -- Inserted new record  **INSERT INTO Student(Id, First\_name, Last\_name, age, GRADE)**  **VALUES (1,'First\_name', 'Last\_name', 12, 'M');**  -- Error Code: 3819. Check constraint 'student\_chk\_1' is violated. | | |

**Working on constraints using alter:**

|  |
| --- |
|  |

# Class -2:

1. **Primary, Foreign key & Referential Integrity**

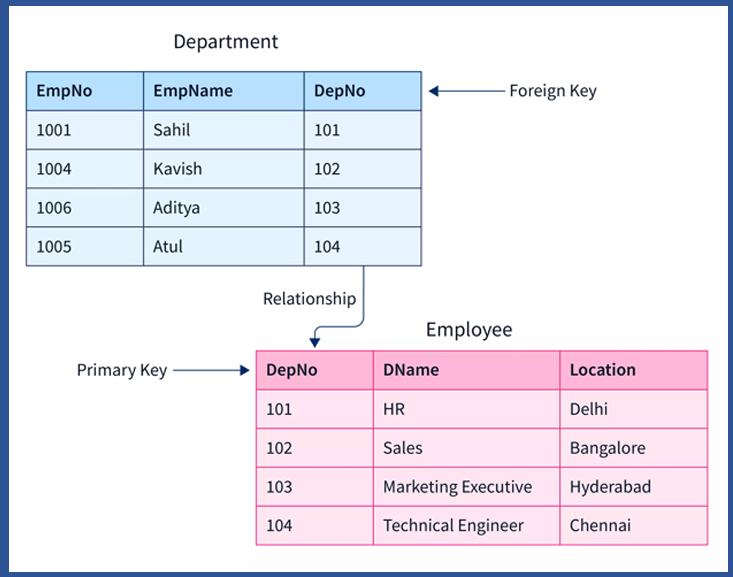
**What is Primary key?**

A Primary Key is a type of key that can uniquely identify each record (rows) of a table.

* A primary key may consist of a single column or multiple columns according to the data set.
* A database table can have only one primary key.
* A primary key cannot have null values. We can say that primary Keys are **NOT NULL** by nature.
* A primary key is **UNIQUE**. The primary key attribute cannot contain duplicate values (there cannot have two records having the same value of the primary key field). A primary key ensures that there are no duplicate records.
* A **composite key** is a combination of two or more columns in a database table that uniquely identifies each record.

**What is foreign key?**

A foreign key is a column or a group of columns in a table whose values are referenced from a primary key in another table.



**Note :** For example , we don’t have EmpNo in above table then we can use First name and last name to make primary key then this called **composite primary key**.

**Primary Key Vs Foreign Key:**



**Practical:**

|  |  |  |
| --- | --- | --- |
| **create** **database** class2 ;  use class2;  **Referential Integrity:**  **Definition**: Referential integrity in SQL is a feature that ensures the accuracy and consistency of data between related tables by enforcing valid relationships based on specified columns.  It requires that all foreign key values in a table must reference valid primary key values in another table. This is typically achieved through foreign key constraints, which prevent the insertion of records with invalid foreign key values and ensure that referenced records are not deleted if they are still in use.  **In General words :** Referential integrity , which make sure that relation between 2 tables remains consistence. It achieves this by following below 2 rules  **Rule -1:** If we try to insert something in child table that does not exist in parent table then this should not allow.  **Rule -2:** If we delete something in parent table then referenced child data with this should also deleted automatically. (To achieve this, we need to use ON DELETE CASCADE / ON UPDATE CASCADE while creating table or alter table later if table created already)  Simply, it makes sure that both parent and child have same data.  To understand this Referential integrity, then check practical session below.  **Rule-1:** Student and Courses Relationship  students Table : Each student has a unique student\_id (Primary Key).  enrollments Table: Tracks which courses each student has enrolled in. It uses student\_id as a Foreign Key to reference the student’s table.   |  | | --- | | -- Students Table  **CREATE** **TABLE** students (  student\_id INT,  student\_name VARCHAR(**50**),  age INT,  **PRIMARY** **KEY** (student\_id)  );  -- Enrollments Table  **CREATE** **TABLE** enrollments (  enrollment\_id INT,  student\_id INT,  course\_name VARCHAR(**50**),  **PRIMARY** **KEY** (enrollment\_id),  **FOREIGN** **KEY** (student\_id) **REFERENCES** students(student\_id)  );  **INSERT** **INTO** students (student\_id,student\_name, age)  **VALUES**(**1**,'Alice', **22**),(**2**,'Bob', **24**),  (**3**,'Charlie', **23**),(**4**,'Diana', **21**);  **INSERT** **INTO** enrollments (enrollment\_id,student\_id, course\_name)  **VALUES**(**1**,**1**, 'Mathematics'),(**2**,**1**, 'Physics'),  (**3**,**2**, 'Chemistry'),(**4**,**3**, 'Biology'),  (**5**,**3**, 'Mathematics'),(**6**,**4**, 'Physics');  **select** \* **from** students;  **select** \* **from** enrollments;  -- Testing Referential Integrity  -- Valid Insert:  **INSERT** **INTO** enrollments (enrollment\_id,student\_id, course\_name)  **VALUES** (**7**,**2**, 'English');    -- Invalid Insert: **Rule -1**  **INSERT** **INTO** enrollments (enrollment\_id,student\_id, course\_name)  **VALUES** (**10**, 'History'); |   **Rule -2 :**   |  | | --- | | **Try to delete parent without ON DELETE CASCADE:**  **DELETE** **FROM** students **where** student\_id=**1**;    **Let’s add ON DELETE CASCADE:**  -- Students Table  **CREATE** **TABLE** students (  student\_id INT,  student\_name VARCHAR(**50**),  age INT,  **PRIMARY** **KEY** (student\_id)  );  -- Enrollments Table  **CREATE** **TABLE** enrollments (  enrollment\_id INT,  student\_id INT,  course\_name VARCHAR(**50**),  **PRIMARY** **KEY** (enrollment\_id),  **FOREIGN** **KEY** (student\_id) **REFERENCES** students(student\_id) **ON** **DELETE** **CASCADE**  );  **INSERT** **INTO** students (student\_id,student\_name, age)  **VALUES**(**1**,'Alice', **22**),(**2**,'Bob', **24**),  (**3**,'Charlie', **23**),(**4**,'Diana', **21**);  **INSERT** **INTO** enrollments (enrollment\_id,student\_id, course\_name)  **VALUES**(**1**,**1**, 'Mathematics'),(**2**,**1**, 'Physics'),  (**3**,**2**, 'Chemistry'),(**4**,**3**, 'Biology'),  (**5**,**3**, 'Mathematics'),(**6**,**4**, 'Physics');  **select** \* **from** students;  **select** \* **from** enrollments;    **Records after delete:**  **DELETE** **FROM** students **where** student\_id=**1**;    Note : Deleted child records automatically. Same for update in we have to use **ON** **UPDATE** **CASCADE**. | |

1. **AUTO INCREMENT:**

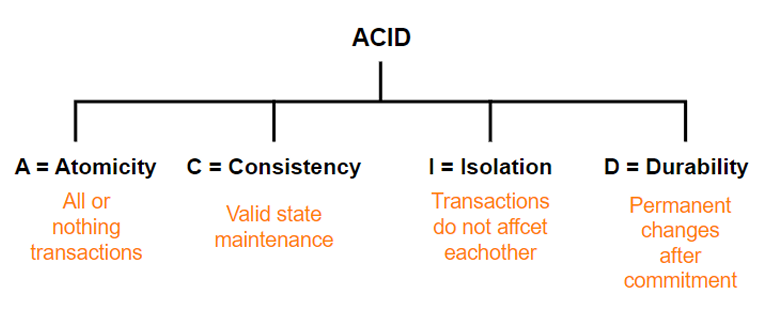
* The AUTO\_INCREMENT attribute is used to generate a unique value automatically for a column whenever a new record is inserted into the table.
* Typically used for primary key columns.
* MySQL starts counting from 1 by default but allows specifying a custom starting value.

**Practical:**

|  |
| --- |
| **CREATE** **TABLE** student(  student\_id int auto\_increment,  name varchar(**50**),  age int,  **PRIMARY** **KEY**(student\_id)  );  **Insert** **into** student (name,age) **Values**('name1',**20**);  **Insert** **into** student (name,age) **Values**('name2',**24**);  **SELECT** \* **FROM** student;    **alter** **table** student auto\_increment=**1000**;  **Insert** **into** student (name,age) **Values**('name3',**34**);  **SELECT** \* **FROM** student; |

1. **ACID Properties:**

Any DBMS follow below ACID properties:



**Atomicity :**

**Definition** : By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all.

**Example:** When we do any money transfer either money will send to other person or it won’t debit from sender account.

**Consistency :**

**Definition** : Consistency ensures that a database remains in a valid state before and after a transaction. It guarantees that any transaction will take the database from one consistent state to another, maintaining the rules and constraints defined for the data.

**Example:** If transaction successful then money should debit from sender and money should credited to receiver. If transaction fails no debit and credit should happen. Means it went to previous consistent state.

**Isolation :**

**Definition** : This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

**Example:** Like when we are doing transaction like sending money to other bank and checking balance. Updated balance only visible in balance check if that transaction is committed.

**Durability :**

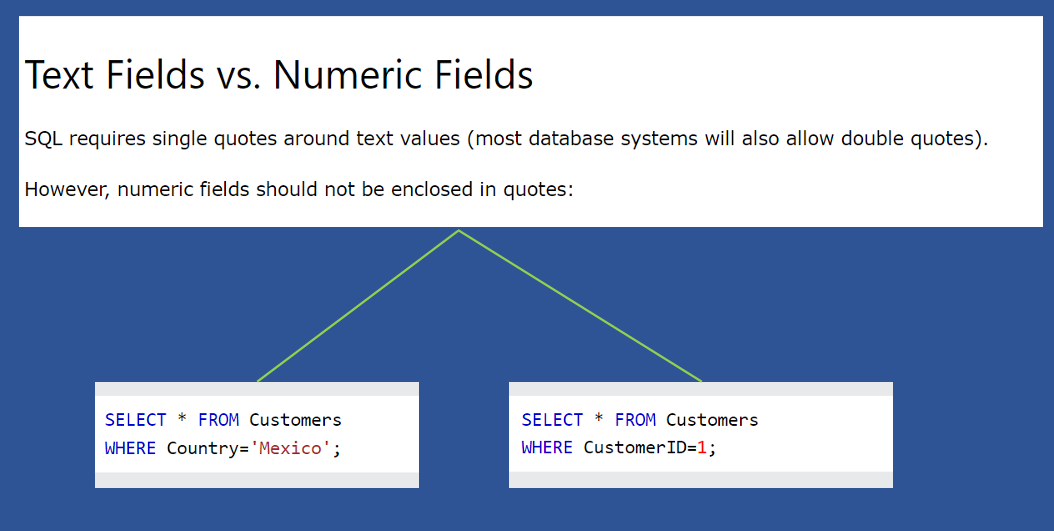
**Definition** : This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory.

**Example :** Like once transaction completed , updated balance in both sender and receiver are permeant.

1. **Where, Limit, Offset, Order by, alias and IN** **clauses :**

The **WHERE** clause is used to filter records. It is used to extract only those records/rows that fulfill a specified condition.

**Note :** Where is not only used in select clause. It can use delete and update also.

****

**Where:** Used to filter the rows.

**Limit** : Used to restrict the number of rows returned in a query result.

**Off set** : Number of rows to skip before returning the result set. Default is 0.

**Order By** : Used to order by ascending or descending.

**Alias** : A temporary name given to table or column.

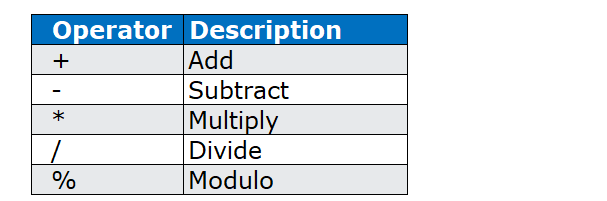
**IN** : To provide multiple options.

**Practical on Where, Limit, Off-set and order by:**

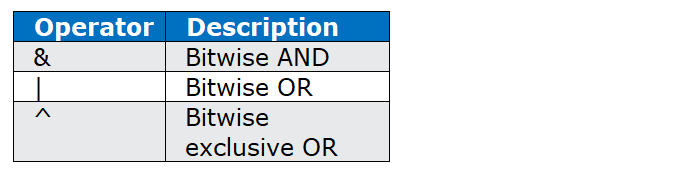
|  |
| --- |
| **drop** **database** if **exists** class2;  **create** **database** class2;  use class2;  **CREATE** **TABLE** employees (  employee\_id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  employee\_name VARCHAR(**100**) **NOT** **NULL**,  department VARCHAR(**50**) **NOT** **NULL**,  salary DECIMAL(**10**, **2**) **DEFAULT** **0**.**00**,  hire\_date DATE  );  **INSERT** **INTO** employees (employee\_name, department, salary, hire\_date)  **VALUES** ('Alice', 'HR', **50000**.**00**, '2020-05-15'),  ('Bob', 'IT', **60000**.**00**, '2019-08-20'),  ('Charlie', 'Sales', **55000**.**00**, '2021-03-10'),  ('David', 'IT', **62000**.**00**, '2018-11-25'),  ('Eve', 'HR', **48000**.**00**, '2022-01-05'),  ('Frank', 'Sales', **51000**.**00**, '2023-07-12');  **select** \* **from** employees;    -- LIMIT and OFFSET  -- fetch 1st 3 rows(Limit)  **select** \* **from** employees  **LIMIT** **3**;  -- -- Skip 3 rows and fetch the next 3 rows  **select** \* **from** employees  **LIMIT** **3** **OFFSET** **0**;  **select** \* **from** employees  **LIMIT** **3**,**3**;  -- Order by  **select** \* **from** employees **order** **by** salary;  **select** \* **from** employees **order** **by** salary **ASC**;  **select** \* **from** employees **order** **by** salary **DESC**;  -- Alias  **select** employee\_name **as** emp\_name  **from** employees  **order** **by** salary;  -- Where  **select** \* **from** employees **where** employee\_id=**4**;  **select** \* **from** employees **where** department='IT';  **select** \* **from** employees **where** salary>**50000**;  -- IN  **update** employees  **set** department = 'Marketing'  **where** employee\_id **in** (**1**,**3**);    -- Not in  **update** employees  **set** department = 'Tech'  **where** employee\_id **not** **in** (**1**,**3**,**5**,**6**);  -- Update record using where  **update** employees  **set** employee\_name = 'Eve\_updated'  **where** employee\_id=**5**;  -- Update 2 values at a time  **update** employees  **set** employee\_name = 'Eve',salary=**1000000**  **where** employee\_id=**5**;  -- Delete record using where  **delete** **from** employees  **where** employee\_id=**5**; |

1. **Operators in MySQL**

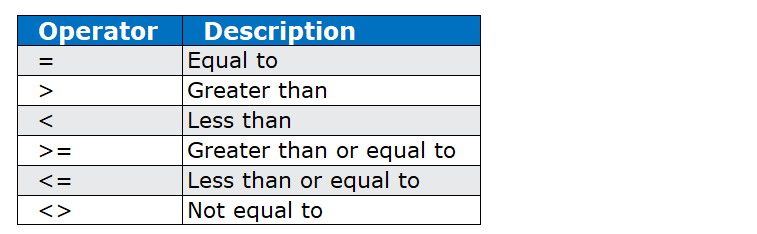
**Arithmetic Operators:**



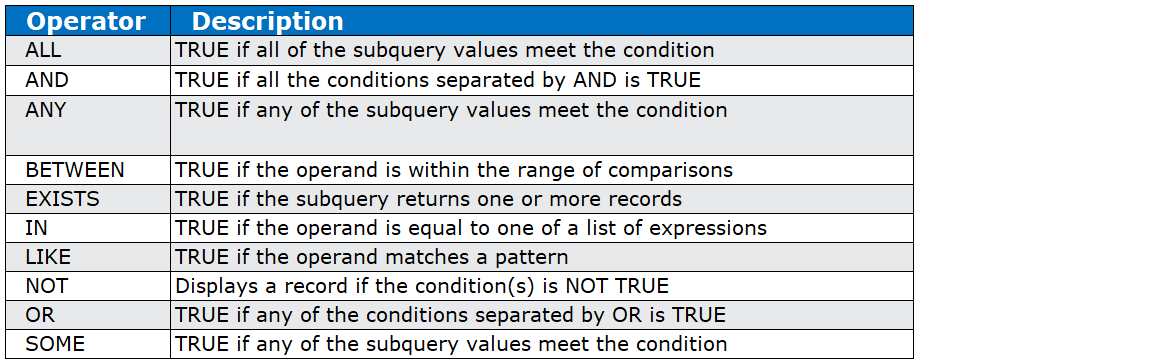
**Bitwise Operators:**



**Comparison Operators**



**Logical Operators:**



**Note**: Any, All, Exists & subquery we will discuss one we understand sub-query.

**Practical:**

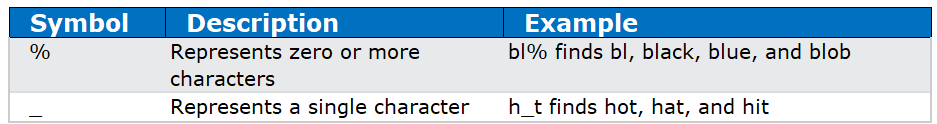
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Arithmetic Operators:**   |  | | --- | | **SELECT** **10** + **5**;  **SELECT** **20** - **8**;  **SELECT** **4** \* **3**;  **SELECT** **15** / **3**;  **SELECT** **17** % **5**; |   **Bitwise Operators :**   |  | | --- | | **SELECT** **10** & **5**; -- Result: 0 (binary: 1010 & 0101 = 0000)  **SELECT** **10** | **5**; -- Result: 15 (binary: 1010 | 0101 = 1111)  **SELECT** **10** ^ **5**; -- Result: 15 (binary: 1010 ^ 0101 = 1111)  -- XOR returns 1 if the two input bits are different,0 if the input bits are the same. |   Create table:   |  | | --- | | **create** **database** mysql\_demo;  use mysql\_demo;  **CREATE** **TABLE** employees\_new (  id INT **PRIMARY** **KEY**,  first\_name VARCHAR(**50**),  last\_name VARCHAR(**50**),  department VARCHAR(**50**),  salary DECIMAL(**10**, **2**),  age INT  );  -- Insert data into the 'employees' table  **INSERT** **INTO** employees\_new (id, first\_name, last\_name, department, salary, age)  **VALUES**  (**1**, 'John', 'Doe', 'Sales', **60000**.**00**, **28**),  (**2**, 'Jane', 'Smith', 'HR', **55000**.**00**, **32**),  (**3**, 'Michael', 'Johnson', 'IT', **75000**.**00**, **40**),  (**4**, 'Emily', 'Brown', 'Finance', **62000**.**00**, **25**),  (**5**, 'David', 'Williams', 'Admin', **58000**.**00**, **30**);    **select** \* **from** employees\_new; |   **Comparison Operators :**   |  | | --- | | -- Equal to (=):Retrieve employees with an age of 32.  **SELECT** \* **FROM** employees\_new **WHERE** age = **32**;  --Not equal to(<> or !=):Retrieve employees who are not in the Admin department.  **SELECT** \* **FROM** employees\_new **WHERE** department <> 'Admin';  **SELECT** \* **FROM** employees\_new **WHERE** **NOT** department = 'Admin';  -- Less than (<):Retrieve employees with a salary less than $60000  **SELECT** \* **FROM** employees\_new **WHERE** salary < **60000**;  -- Greater than (>):Retrieve employees with a age greater than 28.  **SELECT** \* **FROM** employees\_new **WHERE** age > **28**;  -- Less than or equal to (<=) and Greater than or equal to (>=):  -- Retrieve employees with a salary greater than or equal to $58000.  **SELECT** \* **FROM** employees\_new **WHERE** salary >= **58000**; |   **Logical Operators :**   |  | | --- | | -- AND  **SELECT** \* **FROM** employees\_new  **WHERE** department = 'Sales' **AND** salary > **60000**;  -- OR  **SELECT** \* **FROM** employees\_new  **WHERE** department = 'HR' **OR** age < **30**;  -- NOT  **SELECT** \* **FROM** employees\_new  **WHERE** **NOT** department = 'IT';  -- IN  **SELECT** \* **FROM** employees\_new  **WHERE** department **IN** ('Sales', 'IT');  -- BETWEEN  **SELECT** \* **FROM** employees\_new  **WHERE** age **BETWEEN** **25** **AND** **35**; | |

1. **Pattern matching in SQL:**

**Wildcard Characters:**

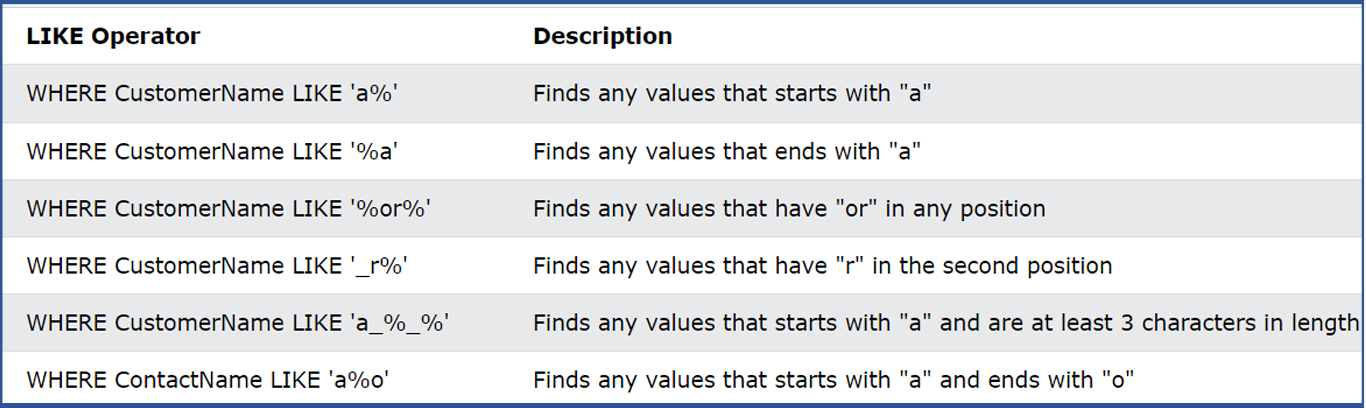
A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the **LIKE** operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.



The wildcards can also be used in combinations!

**Examples:**

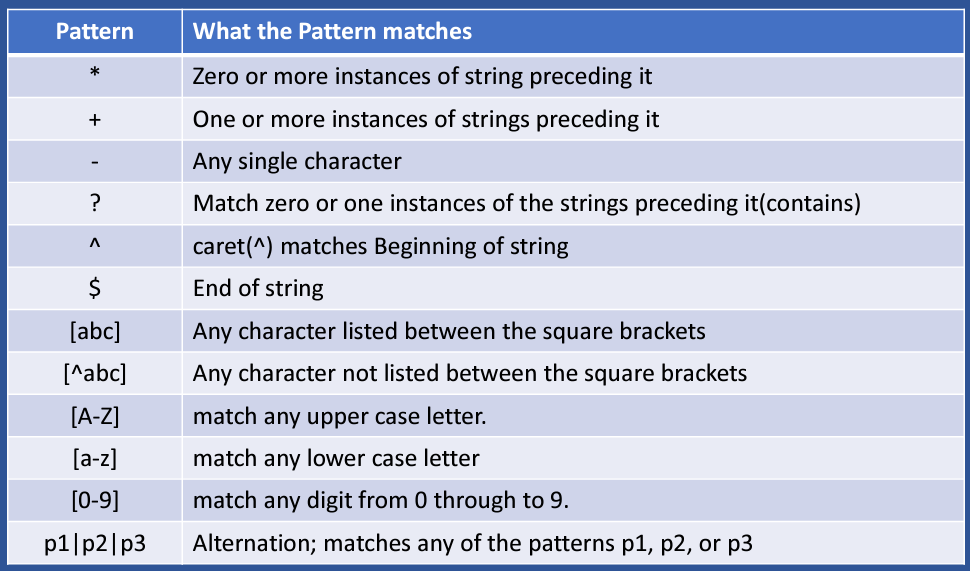


**Practical**:

|  |
| --- |
| **select** \* **from** employees\_new;    -- Select all rows where the 'name' column starts with 'Joh'  **SELECT** \* **FROM** employees\_new **where** first\_name **LIKE** 'Joh%';    -- Select all rows where the second letter in the 'first\_name' column is 'a'  **SELECT** \* **FROM** employees\_new **WHERE** first\_name **LIKE** '\_a%'; |

Note : Like is used to match basic patten matchings. To work on patten matching we have to use **Regexp**.

**Regular Expressions(Regexp):**



**Practical:**

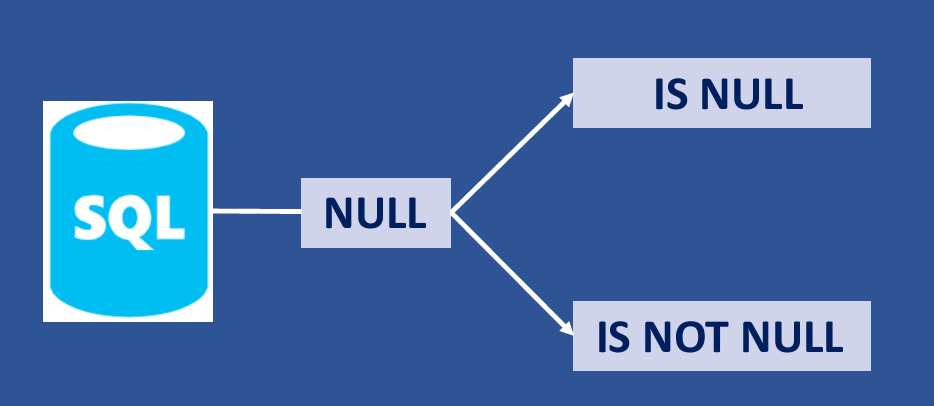
|  |
| --- |
| use mysql\_demo;  -- Create the table  **CREATE** **TABLE** product (  id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  product\_name VARCHAR(**100**),  description TEXT  );  -- Insert sample data  **INSERT** **INTO** product (product\_name, description) **VALUES**  ('apple', 'Fresh and juicy apple.'),  ('apples', 'Variety of apples available.'),  ('banana', 'Yellow and delicious banana.'),  ('orange', 'Sweet and tangy orange.'),  ('pear', 'Juicy and ripe pear.'),  ('grape', 'Bunch of grapes.'),  ('applesauce', 'Homemade applesauce.'),  ('kiwi', 'Tropical kiwi fruit.'),  ('123', 'Number 123 for testing.'),  ('Abc', 'Uppercase letters Abc.');  **SELECT** \* **from** product;    -- Find products with names containing atleat 1 time 'a' character in it.  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP 'a+'; -- one or more  -- Find products with names containing zero or more 1 time 'a' character in it(Means all retun).  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP 'a\*'; -- zero or more  -- Find products with names containing exact one character between p and l  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP 'ap.les'; -- Single character  -- Find products with names containing 'grape' or 'pear'  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP 'grape|pear'; -- Options  -- Find products with names containing 'app' or 'kiw'  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP 'app|kiw';  -- Find products with names containing 'ban' followed by exactly one character using '.'  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP 'ban.$'; -- End character  -- Find products with names starting with 'a', 'b', or 'c' using character classes  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP '^[abc]'; -- Start character  -- Find products with names starting with a letter  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP '^[A-Z]';  -- Find products with names containing a digit  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP '^[0-9]';  -- Find products with names NOT starting with 'a', 'b', or 'c' using negated character class  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP '^[^abc]';  -- Find products with names containing 'ora' followed by zero or one 'n' character using '?'  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP 'ora?'; -- Contains  -- Find products with names containing 'apple' followed by zero or more 's' characters using '\*'  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP 'apples\*';  -- Find products with names starting with 'ki' and ending with 'i'  **SELECT** \* **FROM** product **WHERE** product\_name REGEXP '^ki.\*i$'; |

# Class -3:

**AGENDA:**

1. Handling null values
2. Functions in SQL
3. Group by, having, group concat, group rollup
4. Case statement
5. SQL joins
6. Sub queries in SQL
7. **Handling null values**

Null values mean missing data in SQL. We can use IS NULL / IS NOT NULL to check is values are empty or not...?



**Handling null values:**

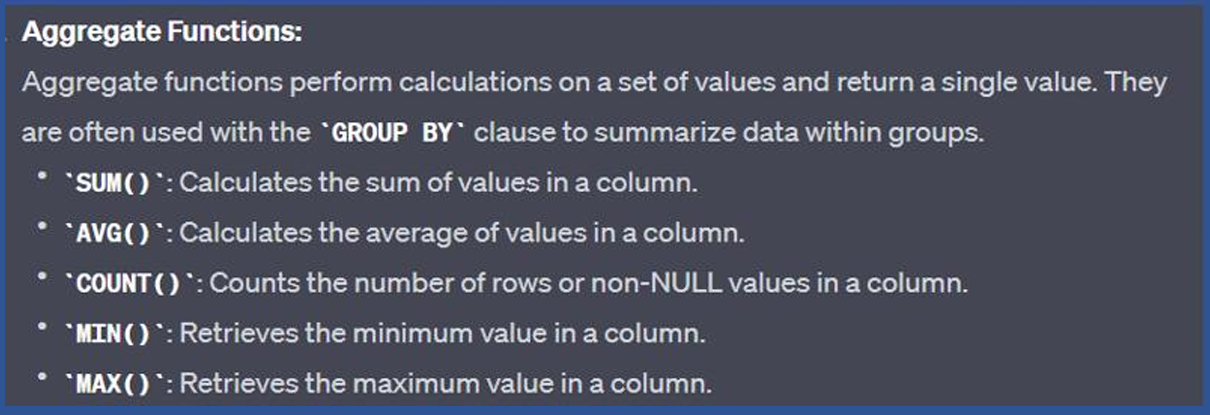
1. By adding constraints, we can restrict null values.
2. Using COALESCE or IFNULL functions (These only temporary purpose)
3. Update the null values with default value.

**Practical:**

|  |
| --- |
| **drop** **database** if **exists** mysql\_demo;  **create** **database** mysql\_demo;  use mysql\_demo;  -- Create the table with NULL and NOT NULL constraints  **CREATE** **TABLE** cust (  id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  first\_name VARCHAR(**50**) **NOT** **NULL**,  last\_name VARCHAR(**50**),  email VARCHAR(**100**) **UNIQUE**,  phone VARCHAR(**15**) **UNIQUE**  );  -- Insert data with NULL and NOT NULL values  **INSERT** **INTO** cust (first\_name, last\_name, email, phone) **VALUES**  ('John', 'Doe', 'john@example.com', '123-456-7890'),  ('Jane', 'Smith', **NULL**, '987-654-3210'),  ('Michael', 'Johnson', 'michael@example.com', **NULL**),  ('Emily', **NULL**, 'emily@example.com', '555-123-4567'),  ('Robert', 'Williams', 'robert@example.com', '111-222-3333'),  ('Shailja', **NULL**, 'shailja@example.com', '110-202-3003');  **Select** \* **from** cust;  -- Checking null values  -- Find customers with no last name  **SELECT** \* **FROM** cust **WHERE** last\_name **IS** **NULL**;  -- Find customers with email addresses  **SELECT** \* **FROM** cust **WHERE** email **IS** **NOT** **NULL**;  -- Find customers with phone numbers  **SELECT** \* **FROM** cust **WHERE** phone **IS** **NOT** **NULL**;  -- Using COALESCE:  **SELECT** COALESCE(last\_name, 'default\_value') **FROM** cust;  -- Using IFNULL (MySQL) or ISNULL (SQL Server):  **SELECT** IFNULL(last\_name, 'default\_value') **FROM** cust;  -- UPDATE to Replace Nulls:  **UPDATE** cust  **SET** last\_name = 'default\_value'  **WHERE** last\_name **IS** **NULL**; |

1. **Functions in SQL**

**Aggregate functions:**



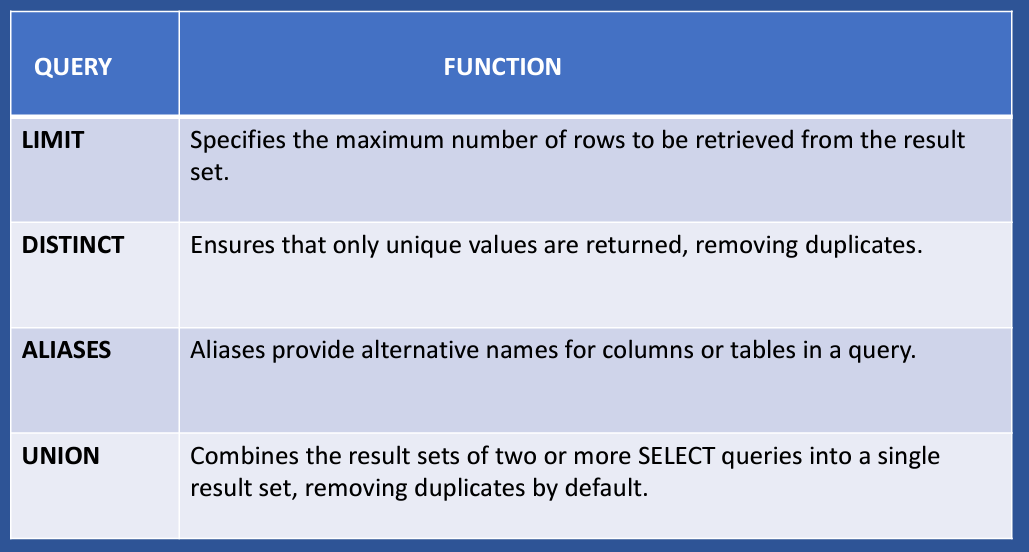
**Mathematical functions:**



**Date and time functions:**

|  |
| --- |
|  |

Other functions:



**Note**: Check difference between UNION and UNION ALL.

**Practical:**

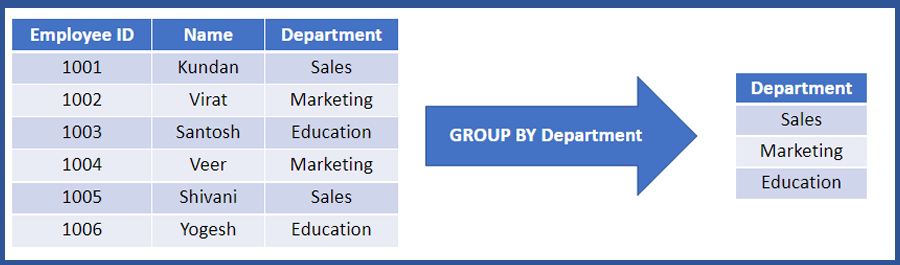
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Aggregate Functions:**   |  | | --- | | **drop** **database** if **exists** mysql\_demo;  **create** **database** mysql\_demo;  use mysql\_demo;  -- Create the table  **CREATE** **TABLE** sales (  product\_id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  quantity INT,  sale\_date DATE  );  -- Insert sample data  **INSERT** **INTO** sales (quantity, sale\_date) **VALUES**  (**5**, '2023-08-01'),  (**3**, '2023-08-02'),  (**7**, '2023-08-01'),  (**4**, '2023-08-03'),  (**10**, '2023-08-02'),  (**NULL**,'2024-02-01');  **Select** \* **from** sales;    -- SUM():Returns the sum of a numeric column.  **SELECT** **SUM**(quantity) **AS** total\_quantity **FROM** sales;  **SELECT** **SUM**(sale\_date) **AS** total\_quantity **FROM** sales;  -- If you pass a non-numeric column to SUM(),MySQL will try to convert the string values to numbers. This can result in unexpected behavior.  -- AVG():Returns the average value of a numeric column.  **SELECT** **AVG**(quantity) **AS** average\_quantity **FROM** sales;  -- COUNT(): Returns the number of non-null values in a set.  **SELECT** **count**(quantity) **AS** TOTAL\_COUNT **FROM** sales;  **SELECT** **count**(\*) **AS** TOTAL\_COUNT **FROM** sales;  -- MIN(): Returns the minimum value in a set.  **SELECT** **MIN**(quantity) **AS** min\_quantity **FROM** sales;  **SELECT** **MIN**(sale\_date) **AS** min\_quantity **FROM** sales;  -- MIN(): Returns the maximum value in a set.  **SELECT** **MAX**(quantity) **AS** max\_quantity **FROM** sales;  **SELECT** **MAX**(sale\_date) **AS** min\_quantity **FROM** sales;  -- When MAX() or MIN() is used on a VARCHAR column, MySQL performs a lexicographical (dictionary) comparison of the strings.  -- When MAX() or MIN() is used on a DATE column, MySQL compares the dates chronologically. |   **Mathematical Functions:**   |  | | --- | | -- Create the data table  **CREATE** **TABLE** **data** (  id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  price DECIMAL(**10**, **4**),  quantity INT,  column1 INT,  column2 INT  );  -- Insert sample data  **INSERT** **INTO** **data** (price, quantity, column1, column2) **VALUES**  (**10**.**0000**, **5**, **10**, **15**),  (**20**.**5034**, **3**, **20**, **8**),  (**15**.**7522**, **8**, **5**, **12**),  (**5**.**252**, **2**, **8**, **20**),  (**25**.**9999**, **10**, **25**, **18**);  **select** \* **from** **data**;    -- Round the price to two decimal places  **SELECT** price, ROUND(price, **2**) **AS** rounded\_price  **FROM** **data**;  -- Find the absolute difference between column1 and column2  **SELECT** price, column2, **ABS**(price - column2) **AS** absolute\_difference  **FROM** **data**;  -- Calculate the square root of the quantity  **SELECT** quantity, SQRT(quantity) **AS** sqrt\_quantity  **FROM** **data**;  -- Calculate the power of column1 raised to column2  **SELECT** column1, column2, POWER(column1, column2) **AS** power\_result  **FROM** **data**;  -- Find the remainder when dividing column2 by 3  **SELECT** column2, **MOD**(column2, **3**) **AS** remainder  **FROM** **data**;  -- The CEIL function returns the smallest integer greater than or equal to a given number.  -- The FLOOR function returns the largest integer less than or equal to a given number.  -- Example using CEIL function  **SELECT** price, CEIL(price) **AS** ceil\_price  **FROM** **data**;  -- Example using FLOOR function  **SELECT** price, FLOOR(price) **AS** floor\_price  **FROM** **data**; |   **Date and time functions:**   |  | | --- | | **drop** **table** if **exists** datetime\_data;  **create** **database** datetime\_data;  use datetime\_data;  -- Create the datetime\_data table  **CREATE** **TABLE** datetime\_data (  id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  event\_name VARCHAR(**50**),  event\_datetime DATETIME  );  -- Insert sample data  **INSERT** **INTO** datetime\_data (event\_name, event\_datetime) **VALUES**  ('Event A', '2023-08-15 09:00:00'),  ('Event B', '2023-08-17 13:30:00'),  ('Event C', '2023-08-16 10:15:00'),  ('Event D', '2023-08-15 14:00:00'),  ('Event E', '2024-07-18 11:45:00');  **select** \* **from** datetime\_data;    -- Example 1: Extract year, month,day from event\_date  **SELECT** event\_name, event\_datetime, DATE(event\_datetime) **AS** event\_date  **FROM** datetime\_data;  **SELECT** event\_name, event\_datetime, **YEAR**(event\_datetime) **AS** event\_year  **FROM** datetime\_data;  **SELECT** event\_name, event\_datetime, **MONTH**(event\_datetime) **AS** event\_month  **FROM** datetime\_data;  **SELECT** event\_name, event\_datetime, **DAY**(event\_datetime) **AS** event\_day  **FROM** datetime\_data;  **SELECT** event\_name, event\_datetime, DAYNAME(event\_datetime) **AS** event\_day  **FROM** datetime\_data;  -- Example 2: Calculate difference in days between two event dates  **SELECT** event\_name, event\_datetime,  DATEDIFF('2024-07-27', event\_datetime) **AS** days\_until\_aug\_20  **FROM** datetime\_data;  -- Example 3: Add to event\_date  **SELECT** event\_name, event\_datetime,  DATE\_ADD(event\_datetime, INTERVAL **3** HOUR) **AS** updated\_event\_date  **FROM** datetime\_data;  **SELECT** event\_name, event\_datetime,  DATE\_ADD(event\_datetime, INTERVAL **10** **DAY**) **AS** updated\_event\_date  **FROM** datetime\_data;  -- Example 4: Format event\_date in a different way  **SELECT** event\_name, event\_datetime,  DATE\_FORMAT(event\_datetime, '%Y-%m-%d %h:%i %p') **AS** formatted\_event\_date  **FROM** datetime\_data;  /\*Format String Breakdown  %Y: Four-digit year (e.g., 2023)  %m: Two-digit month (e.g., 07 for July)  %d: Two-digit day of the month (e.g., 24)  %h: Hour in 12-hour format (01 to 12)  %i: Minutes (00 to 59)  %p: AM or PM\*/  **SELECT** event\_name, event\_datetime,  DATE\_FORMAT(event\_datetime, '%W, %M %d, %Y %r') **AS** formatted\_event\_date  **FROM** datetime\_data;  /\*Format String Breakdown  %W: Full weekday name (e.g., Monday)  %M: Full month name (e.g., July)  %d: Day of the month, numeric (00 to 31)  %Y: Year as a four-digit number (e.g., 2023)  %r: Time in 12-hour format followed by AM or PM (e.g., 02:30:00 PM)\*/  -- Example 5: Find the earliest and latest event dates  **SELECT** **MIN**(event\_datetime) **AS** earliest\_event\_date,  **MAX**(event\_datetime) **AS** latest\_event\_date  **FROM** datetime\_data;  -- Example 6: Extract day of the week from event\_date (1 = Sunday, 2 = Monday, ...)  **SELECT** event\_name, event\_datetime, DAYOFWEEK(event\_datetime) **AS** day\_of\_week  **FROM** datetime\_data;  -- Example 7: Calculate the number of months between two dates  **SELECT** event\_name, event\_datetime,  TIMESTAMPDIFF(**MONTH**, event\_datetime, '2023-09-01') **AS** months\_until\_sep\_1  **FROM** datetime\_data;  -- Example 8: Get the current date and time  **SELECT** CURDATE() **AS** todays\_date;  **SELECT** **CURRENT\_TIMESTAMP**();  **SELECT** NOW(); |   **Other functions:**   |  | | --- | | -- Create the employees table  **CREATE** **TABLE** emp (  id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  first\_name VARCHAR(**50**),  last\_name VARCHAR(**50**),  department VARCHAR(**50**)  );  -- Insert sample data  **INSERT** **INTO** emp (first\_name, last\_name, department) **VALUES**  ('John', 'Doe', 'HR'),  ('Jane', 'Smith', 'IT'),  ('Michael', 'Johnson', 'IT'),  ('Emily', 'Williams', 'Finance'),  ('Robert', 'Brown', 'Finance');  **select** \* **from** emp;  -- Example using DISTINCT to retrieve unique department names  **SELECT** **DISTINCT** department  **FROM** emp;  -- Example using UNION to combine results from two queries  **SELECT** first\_name, last\_name  **FROM** emp  **WHERE** department = 'HR'  **UNION**  **SELECT** first\_name, last\_name  **FROM** emp  **WHERE** department = 'IT'; | |

1. **GROUP BY, HAVING, Group Concat, Group RollUP**

The GROUP BY statement groups rows that have the same values into summary rows It is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result set by one or more columns.

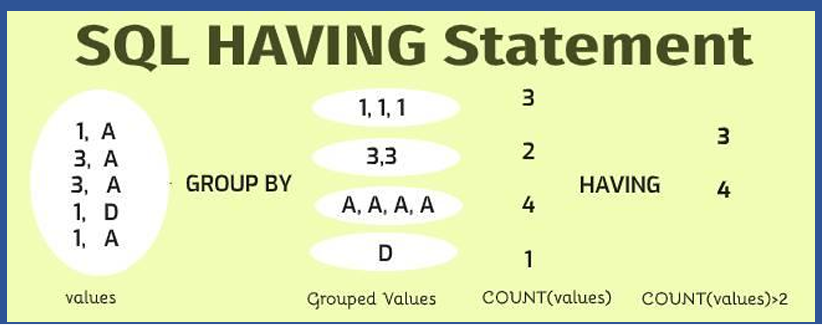
**Grouping Rows :** GROUP BY organizes rows into groups based on the values of one or more columns.

**Aggregate Functions:** After grouping, aggregate functions can be applied to each group to get summary information.



**Having:**

Having clause is added to SQL because the WHERE keyword cannot be used with aggregate functions



**Where** is used to filter total query set. Whereas **having** used to filter only grouped data.

**Practical:**

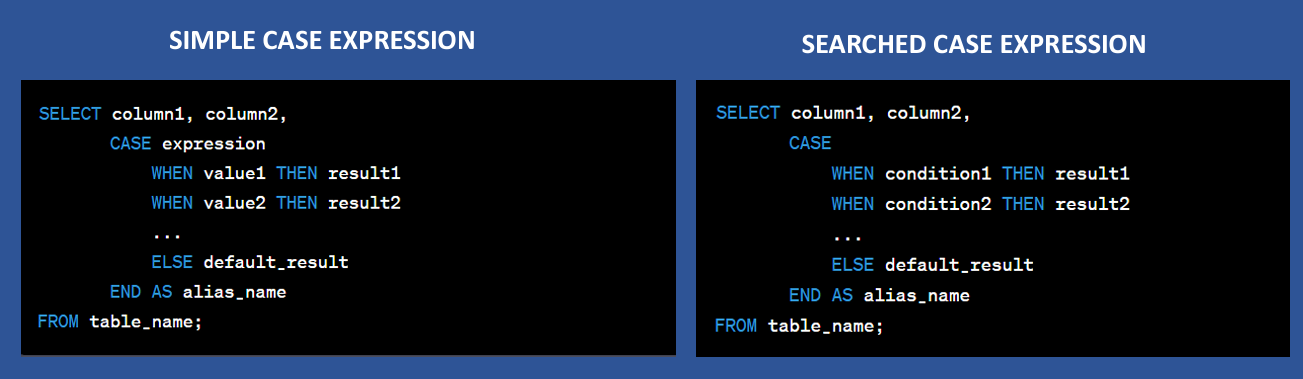
|  |
| --- |
| **drop** **database** if **exists** mysql\_class3;  **create** **database** mysql\_class3;  use mysql\_class3;  **CREATE** **TABLE** sales\_data (  sale\_id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  region VARCHAR(**50**),  product\_id INT,  product\_name VARCHAR(**100**),  customer\_name VARCHAR(**100**),  sale\_date DATE,  sales\_amount DECIMAL(**10**, **2**)  );  **INSERT** **INTO** sales\_data (region, product\_id, product\_name, customer\_name, sale\_date, sales\_amount) **VALUES**  ('North', **101**, 'Laptop', 'John', '2025-01-01', **1000**.**00**),  ('North', **102**, 'Tablet', 'Alice', '2025-01-02', **500**.**00**),  ('North', **101**, 'Laptop', 'Bob', '2025-01-03', **1200**.**00**),  ('South', **103**, 'Smartphone', 'Carol', '2025-01-04', **800**.**00**),  ('South', **101**, 'Laptop', 'David', '2025-01-05', **1500**.**00**),  ('East', **104**, 'Headphones', 'Eve', '2025-01-06', **200**.**00**),  ('East', **104**, 'Headphones', 'Frank', '2025-01-07', **250**.**00**),  ('West', **102**, 'Tablet', 'Grace', '2025-01-08', **400**.**00**),  ('West', **101**, 'Laptop', 'Hannah', '2025-01-09', **900**.**00**);  **select** \* **from** sales\_data;    -- Total sales by each region  **select** region,**sum**(sales\_amount) **AS** total\_sales  **from** sales\_data  **group** **by** region;  -- 2 rules of group by:  -- 1st is always use group by keyword after from.  -- 2nd is always selected columsn which are part of group by or add aggregate functions  -- Total no of sales by each region  **SELECT** region, **count**(\*) **AS** count\_region  **FROM** sales\_data  **GROUP** **BY** region;  -- Group by with order by  **SELECT** region, **count**(\*) **AS** count\_region  **FROM** sales\_data  **GROUP** **BY** region  **order** **by** region;  -- Total sales by each region product wise  **SELECT** region, product\_name, **SUM**(sales\_amount) **AS** total\_sales  **FROM** sales\_data  **GROUP** **BY** region, product\_name;  -- having  **SELECT** region, **SUM**(sales\_amount) **AS** total\_sales  **FROM** sales\_data  **GROUP** **BY** region  **HAVING** **SUM**(sales\_amount) > **2000**;  **SELECT** region, product\_name, **SUM**(sales\_amount) **AS** total\_sales  **FROM** sales\_data  **GROUP** **BY** region, product\_name  **HAVING** **SUM**(sales\_amount) > **1000**;  -- group\_concat  **SELECT** region, GROUP\_CONCAT(customer\_name SEPARATOR ', ') **AS** customers  **FROM** sales\_data  **GROUP** **BY** region;  **SELECT** region, GROUP\_CONCAT(product\_name **ORDER** **BY** product\_name **ASC** SEPARATOR ', ') **AS** products  **FROM** sales\_data  **GROUP** **BY** region;  -- rollup  **SELECT** region, **SUM**(sales\_amount) **AS** total\_sales  **FROM** sales\_data  **GROUP** **BY** region **WITH** **ROLLUP**;  **SELECT** region, product\_name, **SUM**(sales\_amount) **AS** total\_sales  **FROM** sales\_data  **GROUP** **BY** region, product\_name **WITH** **ROLLUP**; |

**SQL quires:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| -- Step 1: Create the table  **CREATE** **TABLE** employes (  employee\_id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  department VARCHAR(**50**),  salary DECIMAL(**10**, **2**),  years\_of\_experience INT  );  -- Step 2: Insert sample data  **INSERT** **INTO** employes (department, salary, years\_of\_experience) **VALUES**  ('HR', **60000**.**00**, **3**),  ('IT', **75000**.**00**, **5**),  ('IT', **80000**.**00**, **8**),  ('Sales', **55000**.**00**, **2**),  ('Sales', **62000**.**00**, **4**);  **Select** \* **from** employes;     1. Find average salary for each department  |  | | --- | | **Select** department,**avg**(salary) **as** avg\_salary  **from** employes  **group** **by** department; |  1. Find average salary above $65000 for each department  |  | | --- | | **Select** department,**avg**(salary) **as** avg\_salary  **from** employes  **group** **by** department  **having** avg\_salary>**65000**; |  1. Find average salary above $40000 for departments with more than one employee  |  | | --- | | **Select** department,**avg**(salary) **as** avg\_salary  **from** employes  **group** **by** department  **having** avg\_salary>**40000** **and** **count**(employee\_id)>**1**; |  1. Find average salary, number of employees for departments with avg\_salary > $40000 and more than one employee.  |  | | --- | | **Select** department,**avg**(salary) **as** avg\_salary,**count**(employee\_id) **as** no\_of\_emp  **from** employes  **group** **by** department  **having** avg\_salary>**40000** **and** no\_of\_emp>**1**; |  1. Find average salary for each department and experience level  |  | | --- | | **SELECT** department, years\_of\_experience, **AVG**(salary) **AS** avg\_salary  **FROM** employes  **GROUP** **BY** department, years\_of\_experience; | |

1. **Case statement:**

The CASE statement is used for conditional logic within a query. It allows you to perform different actions based on different conditions. The CASE statement has two main forms: the simple CASE expression and the searched CASE expression.



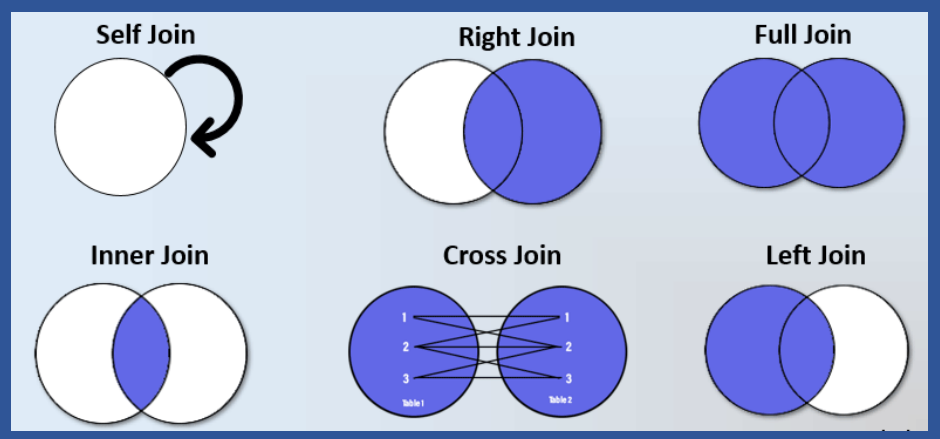
**Practical:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Example – 1:**   |  | | --- | | **drop** **database** if **exists** sql\_class\_3;  **create** **database** sql\_class\_3;  use sql\_class\_3;  -- create table  **CREATE** **TABLE** employee (  id INT auto\_increment **PRIMARY** **KEY**,  name VARCHAR(**50**),  department\_id INT  );  **INSERT** **INTO** employee (name, department\_id) **VALUES**  ("Shailja",**1**),  ("Amie",**2**),  ("Sam",**3**),  ("Sai",**4**);  **Select** \* **from** employee;  -- We want to classify employee based on their department:  **SELECT** id,name,department\_id,  **CASE**  **WHEN** department\_id=**1** **THEN** 'HR'  **WHEN** department\_id=**2** **THEN** 'Marketing'  **WHEN** department\_id=**3** **THEN** 'Sales'  **ELSE** 'UNKNOWN'  **END** **as** department\_name  **FROM** employee; |   **Example – 2:**  Employee Bonus Calculation: Suppose you have an employee’s table with information about employees, including their names, salaries, and years of service. You want to categorize employees into different bonus tiers based on their years of service and calculate the bonus for each employee accordingly.   |  | | --- | | -- Create the employees table  **CREATE** **TABLE** employees (  employee\_id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  employee\_name VARCHAR(**255**),  salary DECIMAL(**10**, **2**),  years\_of\_service INT  );  -- Insert some sample data into the employees table  **INSERT** **INTO** employees (employee\_name, salary, years\_of\_service)  **VALUES**  ('John Doe', **60000**.**00**, **3**),  ('Jane Smith', **75000**.**00**, **7**),  ('Bob Johnson', **85000**.**00**, **12**),  ('Alice Brown', **100000**.**00**, **18**);  **Select** \* **from** employees;  **SELECT** employee\_id,employee\_name,years\_of\_service,  **CASE**  **WHEN** years\_of\_service < **5** **THEN** **0**  **WHEN** years\_of\_service >= **5** **AND** years\_of\_service < **10** **THEN** **500**  **WHEN** years\_of\_service >= **10** **AND** years\_of\_service < **15** **THEN** **1000**  **ELSE** **2000**  **END** **AS** bonus\_amount  **FROM** employees; |   **Example – 3 :**  let's use the CASE statement with values to calculate the shipping cost for orders based on the shipping method chosen by customers:   |  | | --- | | -- Create the orders table  **CREATE** **TABLE** orders (  order\_id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  order\_date DATE,  total\_amount DECIMAL(**10**, **2**),  shipping\_method VARCHAR(**255**)  );  -- Insert some sample data into the orders table  **INSERT** **INTO** orders (order\_date, total\_amount, shipping\_method)  **VALUES**  ('2023-09-01', **150**.**00**, 'Standard'),  ('2023-09-02', **200**.**00**, 'Express'),  ('2023-09-03', **75**.**00**, 'Standard'),  ('2023-09-04', **300**.**00**, 'Express'),  ('2023-09-05', **50**.**00**, 'Economy');  **Select** \* **from** orders;  **SELECT** order\_id,order\_date,total\_amount,shipping\_method,  **CASE** shipping\_method  **WHEN** 'Standard' **THEN** total\_amount \* **0**.**1**  **WHEN** 'Express' **THEN** total\_amount \* **0**.**2**  **ELSE** total\_amount \* **0**.**05**  **END** **AS** shipping\_cost  **FROM** orders;    /\*  In this query:  We use the CASE statement with values to calculate the shipping cost for each order based on the shipping method chosen by customers. For 'Standard' shipping, we charge 10% of the total amount, for 'Express' shipping, we charge 20%, and for any other shipping method (in this case, 'Economy'), we charge 5%.  The result of this query will provide you with a list of orders, their shipping methods, and the calculated shipping cost for each order.  \*/ |   **Example – 4:**  IF() in SQL   |  | | --- | | **SELECT** id,name,department\_id,  IF(department\_id = **1**, 'HR Department',  IF(department\_id = **2**, 'Engineering Department',  IF(department\_id = **3**, 'Sales Department', 'Other Department')  )  ) **AS** department\_name  **FROM** employee; | |

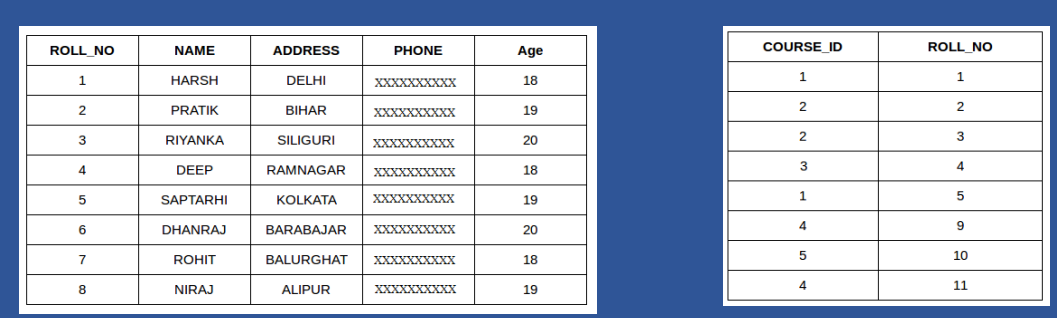
1. **SQL Joins**

A **JOIN** clause is used to combine rows from two or more tables, based on a related column between them.

**Below are types of joins:**



**Example:**



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **INNER JOIN:**   |  | | --- | | The INNER JOIN keyword selects records that have matching values in both tables. |   **LEFT JOIN:**   |  | | --- | | The LEFT JOIN returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match. |   **RIGHT JOIN:**   |  | | --- | | The RIGHT JOIN returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match. |   **OUTER JOIN:**   |  | | --- | | The FULL OUTER JOIN returns all records when there is a match in left (table1) or right (table2) table records.  Note: FULL OUTER JOIN and FULL JOIN are the same. |   **CROSS JOIN:**   |  | | --- | | Cross join return cartesian product of 2 tables. |   **SELF JOIN:**   |  | | --- | | A self-join is a regular join, but the table is joined with itself. | |

**Practical:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **INNER JOIN:**   |  | | --- | | **drop** **database** if **exists** sql\_class\_3;  **create** **database** sql\_class\_3;  use sql\_class\_3;  **CREATE** **TABLE** Employees (  EmployeeID INT,  EmployeeName VARCHAR(**50**),  DepartmentID INT  );  **INSERT** **INTO** Employees (EmployeeID, EmployeeName, DepartmentID) **VALUES**  (**1**, 'Alice', **10**),  (**2**, 'Bob', **20**),  (**3**, 'Charlie', **30**),  (**4**, 'David', **10**),  (**5**, 'Eve', **20**),  (**6**, 'Frank', **30**),  (**7**, 'Grace', **40**),  (**8**, 'Heidi', **40**),  (**9**, 'Ivan', **50**),  (**10**, 'Judy', **NULL**);  **CREATE** **TABLE** Departments (  DepartmentID INT,  DepartmentName VARCHAR(**50**)  );  **INSERT** **INTO** Departments (DepartmentID, DepartmentName) **VALUES**  (**10**, 'HR'),  (**20**, 'Engineering'),  (**30**, 'Finance'),  (**40**, 'Marketing'),  (**50**, 'Sales'),  (**60**, 'Operations');  **select** \* **from** Employees;  **select** \* **from** Departments;    -- inner join: An INNER JOIN returns rows when there is a match in both tables.  **SELECT** e.EmployeeID, e.EmployeeName, d.DepartmentName  **FROM** Employees e  **INNER** **JOIN** Departments d **ON** e.DepartmentID = d.DepartmentID; |   **LEFT JOIN:**   |  | | --- | | -- LEFT JOIN: A LEFT JOIN returns all rows from the left table and the matched rows from the right table. If no match, NULL values are returned.  **SELECT** e.EmployeeID, e.EmployeeName, d.DepartmentName  **FROM** Employees e  **LEFT** **JOIN** Departments d **ON** e.DepartmentID = d.DepartmentID; |   **RIGHT JOIN:**   |  | | --- | | -- RIGHT JOIN: A RIGHT JOIN returns all rows from the right table and the matched rows from the left table. If no match, NULL values are returned.  **SELECT** e.EmployeeID, e.EmployeeName, d.DepartmentName  **FROM** Employees e  **RIGHT** **JOIN** Departments d **ON** e.DepartmentID = d.DepartmentID; |   **OUTER JOIN:**   |  | | --- | | -- FULL OUTER JOIN: A FULL OUTER JOIN returns all rows when there is a match in one of the tables. If no match, NULL values are returned for missing matches on either side.  **SELECT** e.EmployeeID, e.EmployeeName, d.DepartmentName  **FROM** Employees e  **LEFT** **JOIN** Departments d **ON** e.DepartmentID = d.DepartmentID  **UNION**  **SELECT** e.EmployeeID, e.EmployeeName, d.DepartmentName  **FROM** Employees e  **RIGHT** **JOIN** Departments d **ON** e.DepartmentID = d.DepartmentID; |   **CROSS JOIN:**   |  | | --- | | -- SELF JOIN: A SELF JOIN is a regular join, but the table is joined with itself. This is often used to compare rows within the same table.  **CREATE** **TABLE** Employee\_table (  EmployeeID INT,  EmployeeName VARCHAR(**50**),  ManagerID INT  );  **INSERT** **INTO** Employee\_table (EmployeeID, EmployeeName, ManagerID) **VALUES**  (**1**, 'Alice', **NULL**),  (**2**, 'Bob', **1**),  (**3**, 'Charlie', **1**),  (**4**, 'David', **2**),  (**5**, 'Eve', **2**),  (**6**, 'Frank', **3**);  **select** \* **from** Employee\_table;    **SELECT** e1.EmployeeID **AS** EmployeeID, e1.EmployeeName **AS** EmployeeName,  e2.EmployeeID **AS** ManagerID, e2.EmployeeName **AS** ManagerName  **FROM** Employee\_table e1  **LEFT** **JOIN** Employees e2 **ON** e1.ManagerID = e2.EmployeeID; |   **SELF JOIN:**   |  | | --- | | /\* Cross Join:  A cross join (also known as a Cartesian join) combines all rows from one table with all rows from another table, resulting in a Cartesian product of the two tables.  Cross joins are rarely used in practice, but they can be useful in some specific scenarios like You want to create a product catalog that includes all possible combinations of colors and sizes for your products.  \*/  **CREATE** **TABLE** colors (  color\_id INT **PRIMARY** **KEY**,  color\_name VARCHAR(**255**)  );  **CREATE** **TABLE** sizes (  size\_id INT **PRIMARY** **KEY**,  size\_name VARCHAR(**255**)  );  **INSERT** **INTO** colors (color\_id, color\_name)  **VALUES** (**1**, 'Red'), (**2**, 'Blue');  **INSERT** **INTO** sizes (size\_id, size\_name)  **VALUES** (**1**, 'Small'), (**2**, 'Large');  **select** \* **from** colors;    **select** \* **from** sizes;    **SELECT** **c**.color\_name, s.size\_name  **FROM** colors **c**  **CROSS** **JOIN** sizes s; | |

**Extra example:**

|  |
| --- |
| **create** **table** A( id int);  **insert** **into** A (id) **Values**(**1**),(**1**),(**2**),(**NULL**);  **create** **table** B(id int);  **insert** **into** B(id) **Values**(**1**),(**2**),(**2**),(**NULL**);  **select** \* **from** A;  **select** \* **from** B;  -- inner join  **SELECT** A.id, B.id  **FROM** A  **INNER** **JOIN** B **ON** A.id = B.id;  -- left join  **SELECT** A.id, B.id  **FROM** A  **LEFT** **JOIN** B **ON** A.id = B.id;  -- right join  **SELECT** A.id, B.id  **FROM** A  **RIGHT** **JOIN** B **ON** A.id = B.id;  -- full join  **SELECT** A.id, B.id  **FROM** A  **LEFT** **JOIN** B **ON** A.id = B.id  **UNION**  **SELECT** A.id, B.id  **FROM** A  **RIGHT** **JOIN** B **ON** A.id = B.id; |

1. **Sub Queries in SQL**

A subquery, also known as a nested query or inner query, is a query that is embedded within another query. Subqueries are used to retrieve data that will be used in the main query's criteria, calculations, or conditions. They are a powerful tool in SQL for performing complex operations and fetching specific data based on conditions that involve data from multiple tables.



**Types of subqueries include:**

1. **Single-row Subquery**:

A subquery that returns a single value (single row) and is used in comparisons with a single value.

1. **Multi-row Subquery**:

A subquery that returns multiple rows (values) and is used in comparisons involving sets of data.

1. **Correlated Subquery:**

A subquery that references columns from the outer query, allowing for row-by-row comparisons.

**Practical:**

|  |  |
| --- | --- |
| use sql\_class\_4;  **CREATE** **TABLE** employe (  employee\_id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  employee\_name VARCHAR(**255**),  department VARCHAR(**255**),  salary DECIMAL(**10**, **2**)  );  **INSERT** **INTO** employe (employee\_name, department, salary)  **VALUES**  ('John Doe', 'HR', **50000**.**00**),  ('Alice Smith', 'Engineering', **60000**.**00**),  ('Bob Johnson', 'Engineering', **55000**.**00**),  ('Eve Brown', 'Finance', **48000**.**00**),  ('Charlie Brown', 'Finance', **52000**.**00**),  ('Grace Wilson', 'HR', **48000**.**00**);  **select** \* **from** employe;  -- 1.Single-row Subquery  -- Retrieve all employees whose salary is higher than the average salary.  **SELECT** employee\_name, salary  **FROM** employe  **WHERE** salary > (**SELECT** **AVG**(salary) **FROM** employe);  -- 2.Multi-row Subquery :  -- Find all employees who work in departments with more than one employee.  **SELECT** employee\_name, department  **FROM** employe  **WHERE** department **IN**  (**SELECT** department  **FROM** employe  **GROUP** **BY** department  **HAVING** **COUNT**(\*) > **1**);  -- 3.Correlated Subquery:  -- Find employees whose salary is higher than the average salary in their respective departments.  **SELECT** employee\_name, department, salary  **FROM** employe e  **WHERE** salary > (**SELECT** **AVG**(salary) **FROM** employe **WHERE** department = e.department);  -- first outer query is executed then inner query, top-down approach  **Sub-query can used in from and select clause also:**   |  | | --- | | **drop** **database** if **exists** sql\_class\_4;  **create** **database** sql\_class\_4;  use sql\_class\_4;  **CREATE** **TABLE** employees (  employee\_id INT **PRIMARY** **KEY**,  name VARCHAR(**50**),  department\_id INT,  salary DECIMAL(**10**, **2**),  manager\_id INT  );  **INSERT** **INTO** employees (employee\_id, name, department\_id, salary, manager\_id)  **VALUES**  (**1**, 'Alice', **101**, **75000**.**00**, **NULL**),  (**2**, 'Bob', **101**, **55000**.**00**, **1**),  (**3**, 'Charlie', **102**, **60000**.**00**, **1**),  (**4**, 'David', **102**, **80000**.**00**, **3**),  (**5**, 'Eve', **103**, **45000**.**00**, **NULL**);  **select** \* **from** employees;    -- Subquery in FROM Clause  -- Find the total salary for each department and display only those departments with a total salary above 100,000.  **SELECT** department\_id, total\_salary  **FROM** (  **SELECT** department\_id, **SUM**(salary) **AS** total\_salary  **FROM** employees  **GROUP** **BY** department\_id  ) **AS** dept\_salaries  **WHERE** total\_salary > **100000**;  -- Subquery in SELECT Statement  -- Show all employees along with the average salary across all departments.  **SELECT** name, department\_id, salary,  (**SELECT** **AVG**(salary) **FROM** employees) **AS** avg\_salary\_across\_company  **FROM** employees; | |

**Need to check:**

|  |
| --- |
| -- Find all employees who work in departments with more than one employee.  **SELECT** employee\_name, department  **FROM** employe  **WHERE** department **IN** (**SELECT** department **FROM** employees **GROUP** **BY** department **HAVING** **COUNT**(\*) > **1**);  -- Find departments that have at least one employee earning more than $55,000.  **SELECT** **DISTINCT** department  **FROM** employe d  **WHERE** **EXISTS** (**SELECT** **1** **FROM** employees **WHERE** department = d.department **AND** salary > **55000**.**00**);  /\*How the Query Works  The outer query iterates over each row in the employe table.  For each row, the subquery checks if there is at least one row in the employees table with the same department & salary> 55000.00.  If the subquery finds such a row, the EXISTS clause evaluates to true,  and the department from the current row of the outer query is included in the result set.  The DISTINCT keyword ensures that only unique department names are included in the final result set.\*/  -- Find the employee(s) with the highest salary.  **SELECT** employee\_name, salary  **FROM** employe  **WHERE** salary = (**SELECT** **MAX**(salary) **FROM** employe);  -- Highest Salary in Each Department:  **SELECT** department,**MAX**(salary) **AS** highest\_salary  **FROM** employe  **GROUP** **BY** department;  -- Second Highest Salary in Each Department:  **SELECT** department,  **MAX**(salary) **AS** second\_highest\_salary  **FROM** employe  **WHERE** (department, salary) **NOT** **IN** (  **SELECT** department, **MAX**(salary) **AS** first\_highest\_salary  **FROM** employe  **GROUP** **BY** department  )  **GROUP** **BY** department;  /\*The subquery (SELECT department, MAX(salary) FROM employe GROUP BY department) computes the highest salary in each department.  The outer query then selects the maximum salary within each department from the employe table, excluding the highest salary identified by the subquery.  This effectively finds the second highest salary for each department.  The WHERE (department, salary) NOT IN clause ensures that the highest salary for each department is excluded from the results of the outer query.  GROUP BY department in the outer query groups the results by department to ensure that the second highest salary is computed for each department.  \*/  -- Extras  **drop** **table** sample\_table;  **CREATE** **TABLE** sample\_table (  id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  column1 VARCHAR(**255**),  column2 INT  );  **INSERT** **INTO** sample\_table (column1, column2)  **VALUES**  ('A', **1**),  ('B', **2**),  ('C', **3**),  ('A', **1**),  ('B', **2**),  ('D', **4**),  ('E', **1**);  **select** \* **from** sample\_table;  -- Find Duplicate Values in 1 Column:  **SELECT** column1, **COUNT**(\*) **AS** frequency  **FROM** sample\_table  **GROUP** **BY** column1  **HAVING** **COUNT**(\*) > **1**;  -- Find Duplicate Values on 2 Columns Combination:  **SELECT** column1, column2, **COUNT**(\*) **AS** frequency  **FROM** sample\_table  **GROUP** **BY** column1, column2  **HAVING** **COUNT**(\*) > **1**;  /\* Find Duplicate Values on all Columns Combination:  SELECT \*  FROM your\_table  GROUP BY column1, column2, ... -- List all columns here  HAVING COUNT(\*) > 1;  \*/  -- How to Delete Duplicate Values:  **SET** SQL\_SAFE\_UPDATES = **0**;  -- Create a temporary table to store the IDs of rows to be deleted  **CREATE** **TEMPORARY** **TABLE** temp\_ids **AS**  **SELECT** **MIN**(id) **AS** min\_id  **FROM** sample\_table  **GROUP** **BY** column1, column2;  **select** \* **from** temp\_ids;    -- Delete rows from the original table based on the temporary table  **DELETE** **FROM** sample\_table  **WHERE** id **NOT** **IN** (**SELECT** min\_id **FROM** temp\_ids);  -- Drop the temporary table  **DROP** **TEMPORARY** **TABLE** temp\_ids;  **select** \* **from** sample\_table; |

# Class -4:

1. **Exists and not exists**

The EXISTS operator is used to test for the existence of any record in a subquery. If the subquery returns at least one record, the EXISTS condition is true; if the subquery returns no records, the EXISTS condition is false.

The NOT EXISTS operator is the exact opposite. It checks if a subquery returns any records. If the subquery returns no records, the NOT EXISTS condition is true; if the subquery returns records, the NOT EXISTS condition is false.

**Syntax :**

|  |
| --- |
| -- EXISTS SELECT column\_name(s) FROM table\_name WHERE EXISTS (subquery);  -- NOT EXISTS SELECT column\_name(Ss) FROM table\_name WHERE NOT EXISTS (subquery); |

**Practical:**

|  |
| --- |
| **drop** **database** if **exists** sql\_class\_4;  **create** **database** sql\_class\_4;  use sql\_class\_4;  **CREATE** **TABLE** employees (  employee\_id INT **PRIMARY** **KEY**,  name VARCHAR(**50**),  department\_id INT,  salary DECIMAL(**10**, **2**),  manager\_id INT  );  **INSERT** **INTO** employees (employee\_id, name, department\_id, salary, manager\_id)  **VALUES**  (**1**, 'Alice', **101**, **75000**.**00**, **NULL**),  (**2**, 'Bob', **101**, **55000**.**00**, **1**),  (**3**, 'Charlie', **102**, **60000**.**00**, **1**),  (**4**, 'David', **102**, **80000**.**00**, **3**),  (**5**, 'Eve', **103**, **45000**.**00**, **NULL**);  **select** \* **from** employees;    -- EXISTS with Subquery  -- Find employees who are managers (i.e., their employee\_id is listed as a manager\_id in the same table).  **SELECT** name  **FROM** employees e1  **WHERE** **EXISTS** (  **SELECT** **1**  **FROM** employees e2  **WHERE** e2.manager\_id = e1.employee\_id  );  -- The EXISTS clause checks if any rows in the subquery satisfy the condition (e2.manager\_id = e1.employee\_id  -- Interview qution  -- Find employees who belong to departments that have at least one employee.  **SELECT** name  **FROM** employees  **WHERE** department\_id **IN** (  **SELECT** **DISTINCT** department\_id  **FROM** employees  );  **SELECT** name  **FROM** employees  **WHERE** department\_id **IN** (  **SELECT** department\_id  **FROM** employees  **GROUP** **BY** department\_id  **HAVING** **COUNT**(\*) > **1**  ); |

1. **COALESCE ():**

The COALESCE () function returns the first non-null value in a list. But it never replaces original values.

**Syntax :**

COALESCE(val1, val2, ...., val\_n)

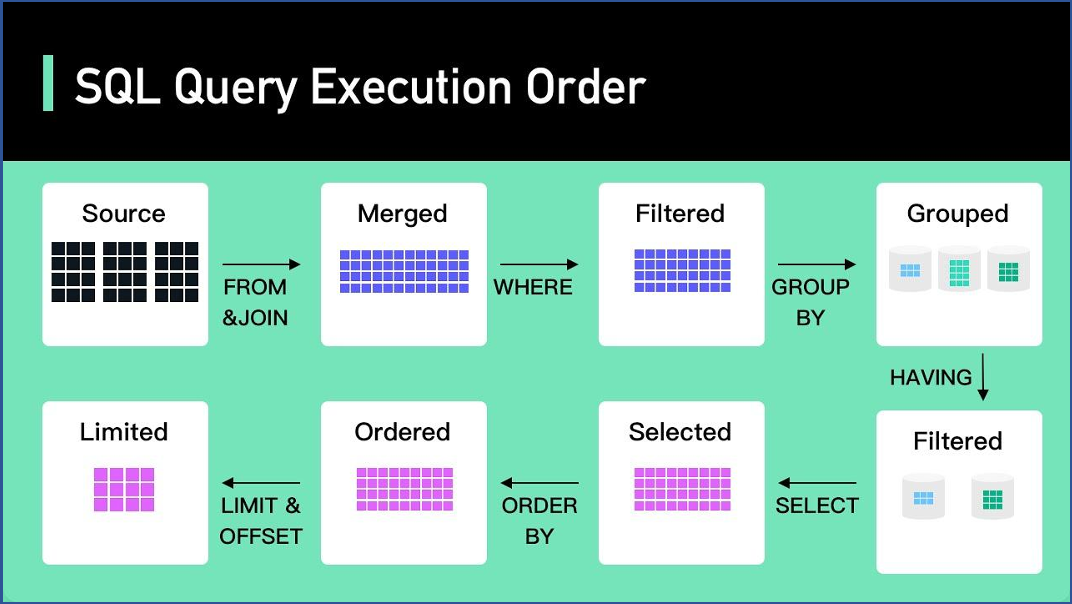
**Example:**

|  |
| --- |
| **SELECT** COALESCE(**NULL**, **NULL**, **NULL**, 'W3Schools.com', **NULL**, 'Example.com') **as** str;    **SELECT** COALESCE(**NULL**, 'NULL', **NULL**, 'W3Schools.com', **NULL**, 'Example.com') **as** str; |

**Practical**:

|  |
| --- |
| **CREATE** **TABLE** employes (  employee\_id INT **PRIMARY** **KEY**,  name VARCHAR(**50**),  salary DECIMAL(**10**, **2**),  bonus DECIMAL(**10**, **2**),  department\_id INT  );  **INSERT** **INTO** employes (employee\_id, name, salary, bonus, department\_id)  **VALUES**  (**1**, 'Alice', **75000**.**00**, **5000**.**00**, **101**),  (**2**, 'Bob', **55000**.**00**, **NULL**, **101**),  (**3**, 'Charlie', **NULL**, **7000**.**00**, **102**),  (**4**, 'David', **80000**.**00**, **NULL**, **NULL**),  (**5**, 'Eve', **NULL**, **NULL**, **103**);  **select** \* **from** employes;    -- Replace NULL with a Default Value  **SELECT** name,COALESCE(bonus, **0**) **AS** bonus\_or\_default  **FROM** employes;  -- Provide a Fallback for Missing Salary  -- If an employee's salary is NULL, use their bonus as a fallback. If both are NULL, display 0.  **SELECT** name, salary , bonus ,COALESCE(salary, bonus, **0**) **AS** effective\_income  **FROM** employes;  -- Use COALESCE in Calculations  -- Calculate the total compensation (salary + bonus) for each employee, replacing NULL values with 0 in the calculation.  **SELECT** name,COALESCE(salary, **0**) + COALESCE(bonus, **0**) **AS** total\_compensation  **FROM** employes;  -- Identify Records with All NULLs  -- Find employees where both salary and bonus are NULL.  **SELECT** name,salary,bonus  **FROM** employes  **WHERE** COALESCE(salary, bonus) **IS** **NULL**; |

1. **SQL query order execution:**



1. **Window functions SQL:**

Window functions allow to retain individual rows while calculating additional insights.

A window function in SQL is a type of function that allows us to perform calculations across a specific set of rows related to the current row. These calculations happen within a defined window of data, and they are particularly useful for aggregates, rankings, and cumulative totals without altering the dataset.

**Practical:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **create** **database** if exits sql\_clas\_4;  use sql\_clas\_4;  **show** tables;  **drop** **table** employee;  **create** **table** employee  ( emp\_ID int  , emp\_NAME varchar(**50**)  , DEPT\_NAME varchar(**50**)  , SALARY int);  **insert** **into** employee **values**(**101**, 'Mohan', 'Admin', **4000**);  **insert** **into** employee **values**(**102**, 'Rajkumar', 'HR', **3000**);  **insert** **into** employee **values**(**103**, 'Akbar', 'IT', **4000**);  **insert** **into** employee **values**(**104**, 'Dorvin', 'Finance', **6500**);  **insert** **into** employee **values**(**105**, 'Rohit', 'HR', **3000**);  **insert** **into** employee **values**(**106**, 'Rajesh', 'Finance', **5000**);  **insert** **into** employee **values**(**107**, 'Preet', 'HR', **7000**);  **insert** **into** employee **values**(**108**, 'Maryam', 'Admin', **4000**);  **insert** **into** employee **values**(**109**, 'Sanjay', 'IT', **6500**);  **insert** **into** employee **values**(**110**, 'Vasudha', 'IT', **7000**);  **insert** **into** employee **values**(**111**, 'Melinda', 'IT', **8000**);  **insert** **into** employee **values**(**112**, 'Komal', 'IT', **10000**);  **insert** **into** employee **values**(**113**, 'Gautham', 'Admin', **2000**);  **insert** **into** employee **values**(**114**, 'Manisha', 'HR', **3000**);  **insert** **into** employee **values**(**115**, 'Chandni', 'IT', **4500**);  **insert** **into** employee **values**(**116**, 'Satya', 'Finance', **6500**);  **insert** **into** employee **values**(**117**, 'Adarsh', 'HR', **3500**);  **insert** **into** employee **values**(**118**, 'Tejaswi', 'Finance', **5500**);  **insert** **into** employee **values**(**119**, 'Cory', 'HR', **8000**);  **insert** **into** employee **values**(**120**, 'Monica', 'Admin', **5000**);  **insert** **into** employee **values**(**121**, 'Rosalin', 'IT', **6000**);  **insert** **into** employee **values**(**122**, 'Ibrahim', 'IT', **8000**);  **insert** **into** employee **values**(**123**, 'Vikram', 'IT', **8000**);  **insert** **into** employee **values**(**124**, 'Dheeraj', 'IT', **11000**);  **COMMIT**;  **select** \* **from** employee;  Using Aggregate function as Window Function   |  | | --- | | -- Without window function, SQL will reduce the no of records.  **select** dept\_name, **max**(salary) **from** employee  **group** **by** dept\_name;  -- By using MAX as an window function, SQL will not reduce records but the result will be shown corresponding to each record.  **select** e.\*,  **max**(salary) over() **as** max\_salary  **from** employee e;  **select** e.\*,  **max**(salary) over(partition **by** dept\_name) **as** max\_salary  **from** employee e;  -- Similarly we can use other aggregate function like min, sum etc as window functions. |  1. **ROW\_NUMBER()** : Assigns a unique sequential integer to rows within a partition of the result set, starting from 1 for the first row in each partition.  |  | | --- | | **select** \*,row\_number() over() **as** rn  **from** employee;  **select** \*,row\_number() over(partition **by** dept\_name) **as** rn  **from** employee;  -- This is giving ranking based in id means there joined data. as old emploee have less employee id  **select** \*,row\_number() over(partition **by** dept\_name **order** **by** emp\_id) **as** rn  **from** employee;  -- Fetch the first 2 employees from each department to join the company.  **select** \* **from** (  **select** \*,  row\_number() over(partition **by** dept\_name **order** **by** emp\_id) **as** rn  **from** employee) x  **where** x.rn < **3**; |  1. **RANK()** : Assigns a rank to each row within a partition of the result set, with the same rank for rows with the same values. The next rank value after a tie is incremented by the number of tied rows, resulting in gaps in the ranking.  |  | | --- | | -- RANK( skips)  **select** \*,  rank() over(partition **by** dept\_name **order** **by** salary **desc**) **as** rnk  **from** employee;  -- Fetch the top 3 employees in each department earning the max salary.  **select** \* **from** (  **select** \*,  rank() over(partition **by** dept\_name **order** **by** salary **desc**) **as** rnk  **from** employee) x  **where** x.rnk < **4**; |  1. **DENSE\_RANK()** : Assigns a rank to each row within a partition of the result set, with the same rank for rows with the same values. The next rank value after a tie is incremented by 1, resulting in no gaps in the ranking.  |  | | --- | | -- DENSE\_RANK( doesnt skip)  **select** \*,  rank() over(partition **by** dept\_name **order** **by** salary **desc**) **as** rnk,  dense\_rank() over(partition **by** dept\_name **order** **by** salary **desc**) **as** dense\_rnk  **from** employee;  -- Checking the different between rank, dense\_rnk and row\_number window functions:  **select** \*,  rank() over(partition **by** dept\_name **order** **by** salary **desc**) **as** rnk,  dense\_rank() over(partition **by** dept\_name **order** **by** salary **desc**) **as** dense\_rnk,  row\_number() over(partition **by** dept\_name **order** **by** salary **desc**) **as** rn  **from** employee; |  1. **lead and lag**   LEAD() and LAG() are SQL window functions used to access data from preceding or following rows in the result set without the need for self-joins.  These functions are useful for comparing values between rows within the same result set.  **LEAD()** : Provides access to a subsequent row (a row after the current row) within the same result set, based on a specified offset.  **LAG()** : Provides access to a preceding row (a row before the current row) within the same result set, based on a specified offset.   |  | | --- | | --- LAG  **select** \*,  lag(salary) over(partition **by** dept\_name **order** **by** emp\_id) **as** prev\_empl\_sal  **from** employee;  **select** \*,  lag(salary,**2**,**0**) over(partition **by** dept\_name **order** **by** emp\_id) **as** prev\_empl\_sal  **from** employee;  /\*LAG(salary, 2, 0): The LAG function is used to retrieve the value of the salary column from two rows before the current row within the partition.  If there are fewer than two preceding rows, it returns 0.\*/  -- Use of lag  -- fetch a query to display if the salary of an employee is higher, lower or equal to the previous employee.  **select** \*,  lag(salary) over(partition **by** dept\_name **order** **by** emp\_id) **as** prev\_empl\_sal,  **case** **when** salary > lag(salary) over(partition **by** dept\_name **order** **by** emp\_id) **then** 'Higher than previous employee'  **when** salary < lag(salary) over(partition **by** dept\_name **order** **by** emp\_id) **then** 'Lower than previous employee'  **when** salary = lag(salary) over(partition **by** dept\_name **order** **by** emp\_id) **then** 'Same than previous employee' **end** **as** sal\_range  **from** employee;  -- Similarly using lead function to see how it is different from lag.  **select** \*,  lag(salary) over(partition **by** dept\_name **order** **by** emp\_id) **as** prev\_empl\_sal,  lead(salary) over(partition **by** dept\_name **order** **by** emp\_id) **as** next\_empl\_sal  **from** employee; |  1. **First value , last value and nth value with frames**  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | -- Script to create the Product table and load data into it.  **DROP** **TABLE** IF **EXISTS** product;  **CREATE** **TABLE** product  (  product\_category varchar(**255**),  brand varchar(**255**),  product\_name varchar(**255**),  price int  );  **INSERT** **INTO** product **VALUES**  ('Phone', 'Apple', 'iPhone 12 Pro Max', **1300**),  ('Phone', 'Apple', 'iPhone 12 Pro', **1100**),  ('Phone', 'Apple', 'iPhone 12', **1000**),  ('Phone', 'Samsung', 'Galaxy Z Fold 3', **1800**),  ('Phone', 'Samsung', 'Galaxy Z Flip 3', **1000**),  ('Phone', 'Samsung', 'Galaxy Note 20', **1200**),  ('Phone', 'Samsung', 'Galaxy S21', **1000**),  ('Phone', 'OnePlus', 'OnePlus Nord', **300**),  ('Phone', 'OnePlus', 'OnePlus 9', **800**),  ('Phone', 'Google', 'Pixel 5', **600**),  ('Laptop', 'Apple', 'MacBook Pro 13', **2000**),  ('Laptop', 'Apple', 'MacBook Air', **1200**),  ('Laptop', 'Microsoft', 'Surface Laptop 4', **2100**),  ('Laptop', 'Dell', 'XPS 13', **2000**),  ('Laptop', 'Dell', 'XPS 15', **2300**),  ('Laptop', 'Dell', 'XPS 17', **2500**),  ('Earphone', 'Apple', 'AirPods Pro', **280**),  ('Earphone', 'Samsung', 'Galaxy Buds Pro', **220**),  ('Earphone', 'Samsung', 'Galaxy Buds Live', **170**),  ('Earphone', 'Sony', 'WF-1000XM4', **250**),  ('Headphone', 'Sony', 'WH-1000XM4', **400**),  ('Headphone', 'Apple', 'AirPods Max', **550**),  ('Headphone', 'Microsoft', 'Surface Headphones 2', **250**),  ('Smartwatch', 'Apple', 'Apple Watch Series 6', **1000**),  ('Smartwatch', 'Apple', 'Apple Watch SE', **400**),  ('Smartwatch', 'Samsung', 'Galaxy Watch 4', **600**),  ('Smartwatch', 'OnePlus', 'OnePlus Watch', **220**);  **COMMIT**;  **select** \* **from** product;  **FIRST\_VALUE**   |  | | --- | | -- Write query to display the most expensive product under each category (corresponding to each record)  -- without window function  **select** \*  **from** product **as** pt  **left** **join** (**select** product\_cty\_price.product\_category,p.product\_name  **from** product **as** p  **JOIN** (  **select** product\_category ,**max**(price) **as** max\_price  **from** product  **group** **by** product\_category) **as** product\_cty\_price  **on** p.product\_category=product\_cty\_price.product\_category **and**  p.price=product\_cty\_price.max\_price) **as** product\_cty\_name  **on** pt.product\_category=product\_cty\_name.product\_category;  -- with window function  **select** \*,  first\_value(product\_name) over(partition **by** product\_category **order** **by** price **desc**) **as** most\_exp\_product  **from** product;  -- Get maximum price product accros for all products  **select** \*,  first\_value(product\_name) over(**order** **by** price **desc**) **as** most\_exp\_product  **from** product; |   **LAST\_VALUE:**   |  | | --- | | -- Write query to display the least expensive product under each category (corresponding to each record)  **select** \*,  last\_value(product\_name) over(partition **by** product\_category **order** **by** price **desc**) **as** least\_exp\_product  **from** product; |   Problem with last value can be solved using **Frames** concept:   |  | | --- | | -- Frame range from start to end row of partition  **select** \*,  first\_value(product\_name) over(partition **by** product\_category **order** **by** price **desc**) **as** most\_exp\_product,  last\_value(product\_name) over(partition **by** product\_category **order** **by** price **desc**  range **between** unbounded preceding **and** unbounded following) **as** least\_exp\_product  **from** product;  -- Frame range from start to current row  **select** \*,  last\_value(product\_name) over(partition **by** product\_category **order** **by** price **desc**  range **between** unbounded preceding **and** **current** **row**) **as** least\_exp\_product  **from** product  **WHERE** product\_category ='Phone'; |   **Nth value:**   |  | | --- | | -- Write query to display the Second most expensive product under each category.  **select** \*,  nth\_value(product\_name, **2**) over w **as** second\_most\_exp\_product  **from** product  window w **as** (partition **by** product\_category **order** **by** price **desc**  range **between** unbounded preceding **and** unbounded following); |   Alternate way to write SQL query using Window functions   |  | | --- | | **select** \*,  first\_value(product\_name) over w **as** most\_exp\_product,  last\_value(product\_name) over w **as** least\_exp\_product  **from** product  **WHERE** product\_category ='Phone'  window w **as** (partition **by** product\_category **order** **by** price **desc**  range **between** unbounded preceding **and** unbounded following); | | |

1. **CTE (Common Table Expression) Or WITH clause:**

**Uses of CTE:**

* This is to simplify the complex queries.
* It improves the readability of a query.
* Improves performance.

**Practical:**

|  |
| --- |
| **drop** **database** if **exists** SQL\_CLASS\_5;  **create** **database** SQL\_CLASS\_5;  use SQL\_CLASS\_5;  **CREATE** **TABLE** employees (  employee\_id INT **PRIMARY** **KEY**,  first\_name VARCHAR(**50**),  last\_name VARCHAR(**50**),  department\_id INT,  salary DECIMAL(**10**, **2**),  manager\_id INT **NULL**  );  **INSERT** **INTO** employees (employee\_id, first\_name, last\_name, department\_id, salary, manager\_id) **VALUES**  (**1**, 'John', 'Doe', **1**, **60000**.**00**, **NULL**),  (**2**, 'Jane', 'Smith', **1**, **75000**.**00**, **1**),  (**3**, 'Emily', 'Davis', **2**, **50000**.**00**, **1**),  (**4**, 'Michael', 'Brown', **2**, **55000**.**00**, **3**),  (**5**, 'Jessica', 'Williams', **3**, **80000**.**00**, **3**),  (**6**, 'Daniel', 'Jones', **3**, **65000**.**00**, **5**),  (**7**, 'Laura', 'Garcia', **4**, **90000**.**00**, **NULL**);  **select** \* **from** employees;    -- Example1: Select the first and last names of all employees.  **WITH** EmployeeNames **AS** (  **SELECT** first\_name, last\_name  **FROM** employees  )  **SELECT** \* **FROM** EmployeeNames;  -- Example2: Calculate the average salary by department.  **WITH** AvgSalaryByDept **AS** (  **SELECT** department\_id, **AVG**(salary) **AS** avg\_salary  **FROM** employees  **GROUP** **BY** department\_id  )  **SELECT** department\_id, avg\_salary  **FROM** AvgSalaryByDept;  -- Example3: Find departments with an average salary greater than $60,000.  **WITH** AvgSalaryByDept **AS** (  **SELECT** department\_id, **AVG**(salary) **AS** avg\_salary  **FROM** employees  **GROUP** **BY** department\_id  )  **SELECT** department\_id, avg\_salary  **FROM** AvgSalaryByDept  **WHERE** avg\_salary > **60000**;  -- Example4: Combine multiple CTEs to find the top paid employee in each department.  **WITH** DepartmentSalaries **AS** (  **SELECT** department\_id, employee\_id, salary  **FROM** employees  ),  MaxSalaries **AS** (  **SELECT** department\_id, **MAX**(salary) **AS** max\_salary  **FROM** DepartmentSalaries  **GROUP** **BY** department\_id  )  **SELECT** e.department\_id, e.employee\_id, e.salary  **FROM** employees e  **JOIN** MaxSalaries ms **ON** e.department\_id = ms.department\_id **AND** e.salary = ms.max\_salary; |

**Subquery vs CTE:**

If your query is very complex and using query again and again then go for CTE.

|  |  |  |
| --- | --- | --- |
| **-- QUERY 1 :**   |  | | --- | | **create** **table** emp  ( emp\_ID int  , emp\_NAME varchar(**50**)  , SALARY int);  **insert** **into** emp **values**(**101**, 'Mohan', **40000**);  **insert** **into** emp **values**(**102**, 'James', **50000**);  **insert** **into** emp **values**(**103**, 'Robin', **60000**);  **insert** **into** emp **values**(**104**, 'Carol', **70000**);  **insert** **into** emp **values**(**105**, 'Alice', **80000**);  **insert** **into** emp **values**(**106**, 'Jimmy', **90000**);  **select** \* **from** emp;  -- Find employee who earns more than average salaray of other employees  -- with subquey  **select** \*  **from** emp  **where** salary > ( **select** **avg**(salary) **from** emp);  -- with CTE  **with** avg\_salary **as**(  **select** **avg**(salary) **as** avg\_sal  **from** emp  )  **select** \* **from** emp  **inner** **join** avg\_salary  **on** salary> avg\_sal;  **Note :** Here better go for Sub-query rather than CTE. As CTE increasing complexity. |   **-- QUERY 2 :**   |  | | --- | | -- Find stores whose sales where better than the average total sales across all stores  TO solve above query, we need to know below things:  1.Total sales per each store -- Total\_Sales  2.Average sales with respect to all stores. - Average\_Sales  Once find above filter( where Total\_Sales > Average\_Sales of all stores)  -- Find total sales per each store- Total\_Sales  **select** store\_id, **sum**(cost) **as** total\_sales\_per\_store  **from** sales  **group** **by** store\_id;  -- Find average sales with respect to all stores-Average\_Sales  **select** **avg**(total\_sales\_per\_store) **as** avg\_sale\_for\_all\_store  **from** (**select** store\_id, **sum**(cost) **as** total\_sales\_per\_store  **from** sales s  **group** **by** store\_id) x;  **Using Subquery:**  **select** \*  **from** (**select** store\_id, **sum**(cost) **as** total\_sales\_per\_store  **from** sales  **group** **by** store\_id) total\_sales  **join** (**select** **avg**(total\_sales\_per\_store) **as** avg\_sale\_for\_all\_store  **from** (**select** store\_id, **sum**(cost) **as** total\_sales\_per\_store  **from** sales  **group** **by** store\_id) x  ) avg\_sales  **on** total\_sales.total\_sales\_per\_store > avg\_sales.avg\_sale\_for\_all\_store;  **Using CTE:**  **WITH** total\_sales **as**  (**select** store\_id, **sum**(cost) **as** total\_sales\_per\_store  **from** sales  **group** **by** store\_id),  avg\_sales **as**  (**select** **avg**(total\_sales\_per\_store) **as** avg\_sale\_for\_all\_store  **from** total\_sales)  **select** \*  **from** total\_sales  **join** avg\_sales  **on** total\_sales.total\_sales\_per\_store > avg\_sales.avg\_sale\_for\_all\_store;  **Note :** Here better go for CTE rather than Sub-query. As it increasing readability and decrease complexity. | |

**Recursive CTE:**

Call same query again and again.

**Syntax:**

|  |
| --- |
| **WITH** [**RECURSIVE**] CTE\_name **AS**  (  **SELECT** query (Non **Recursive** query **or** the Base query)  **UNION** [**ALL**]  **SELECT** query (**Recursive** query **using** CTE\_name [**with** a termination condition])  )  **SELECT** \* **FROM** CTE\_name; |

**Note :**

* Union eradicates duplicates and gives only distinct result from both the tables
* Union all gives all records from both tables including duplicates

**Practical:**

|  |
| --- |
| -- Q: Display number from 1 to 10 without using any in built functions.  **with** **recursive** num **as**  (  **select** **1** **as** n  **union**  **select** n+**1** **as** n **from** num **where** n < **10**  )  **select** \* **from** num;  -- Example: Find the hierarchy of employees.  **WITH** **RECURSIVE** EmployeeHierarchy **AS** (  **SELECT** employee\_id,first\_name,last\_name,manager\_id,**1** **AS** **level**  **FROM** employees  **WHERE** manager\_id **IS** **NULL**  **UNION** **ALL**    **SELECT** e.employee\_id, e.first\_name, e.last\_name,e.manager\_id,eh.**level** + **1**  **FROM** employees e **INNER** **JOIN** EmployeeHierarchy eh  **ON** e.manager\_id = eh.employee\_id  )  **SELECT** \* **FROM** EmployeeHierarchy;  /\* This recursive CTE builds an employee hierarchy.  The base case selects employees without a manager (manager\_id IS NULL) and sets the level to 1.  The recursive part joins the employees table with the EmployeeHierarchy CTE to  find employees managed by the employees in the previous level, incrementing the level by 1 each time.  \*/ |

**Iterative CTE** means normal CTE’s where you have multiple queries make temp table and use in mail query.

1. **Views:**

* View is a virtual table based on the result-set of an SQL statement.
* A view 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.
* You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.
* A view is created with the CREATE VIEW statement.

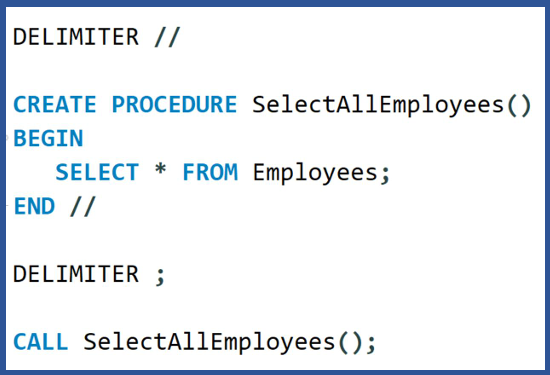
**Practical:**

|  |
| --- |
| USE SQL\_CLASS\_5;  **drop** **table** Employees;  **CREATE** **TABLE** Employees (  EmployeeID INT **PRIMARY** **KEY**,  FirstName VARCHAR(**50**),  LastName VARCHAR(**50**),  Department VARCHAR(**50**),  Salary DECIMAL(**10**, **2**)  );  **INSERT** **INTO** Employees (EmployeeID, FirstName, LastName, Department, Salary)  **VALUES**  (**1**, 'John', 'Doe', 'HR', **50000**.**00**),  (**2**, 'Jane', 'Smith', 'IT', **60000**.**00**),  (**3**, 'Bob', 'Johnson', 'Finance', **55000**.**00**),  (**4**, 'Alice', 'Williams', 'Marketing', **52000**.**00**),  (**5**, 'Eve', 'Anderson', 'IT', **62000**.**00**);  **Select** \* **from** Employees;  -- CREATE A VIEW  **CREATE** **VIEW** vw\_EmployeeNames **AS**  **SELECT** EmployeeID, FirstName, LastName  **FROM** Employees;  -- Call view  **SELECT** \* **FROM** vw\_EmployeeNames;  -- CREATE OR REPLACE A VIEW –> update view  **CREATE** **OR** **REPLACE** **VIEW** EmployeeNames **AS**  **SELECT** EmployeeID, FirstName, LastName, Department  **FROM** Employees;  -- Call view  **SELECT** \* **FROM** vw\_EmployeeNames;  -- DROP A VIEW  **DROP** **VIEW** vw\_EmployeeNames; |

1. **Stored procedures in SQL**

* A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.
* So, if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.
* You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

**Example:**



**Practical:**

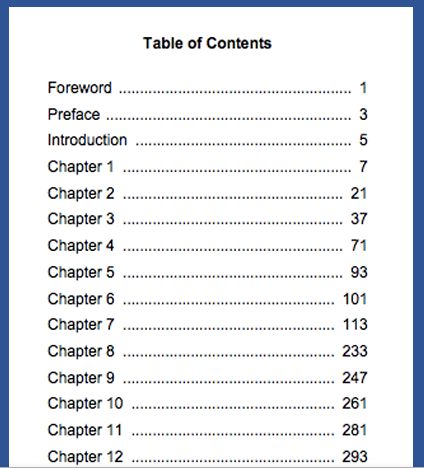
|  |
| --- |
| /\*  Suppose you have a database with a table named Employees  You want to create a stored procedure that retrieves all employees from a specific department.  Here's how you can create the stored procedure:  \*/  **DROP** **PROCEDURE** IF **EXISTS** GetEmployeesByDepartment;  -- Create procedure  **DELIMITER** //  **CREATE** **PROCEDURE** GetEmployeesByDepartment(**IN** DepartmentName VARCHAR(**50**))  **BEGIN**  **SELECT** EmployeeID, FirstName, LastName  **FROM** Employees  **WHERE** Department = DepartmentName;  **END** //  **DELIMITER** ;  -- Call procedure  **CALL** GetEmployeesByDepartment('HR'); |

1. **Indexes in SQL**

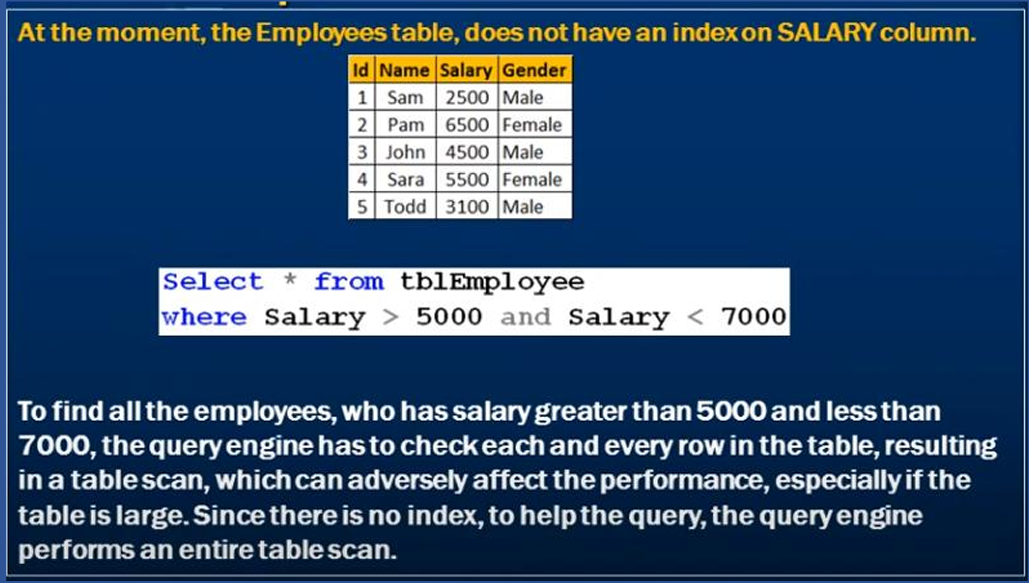
* Indexes are references to data which are used by queries to find data from tables quickly.
* It cannot be viewed by the users and just used to speed up the database access.
* Without index, the query engine checks every row in the table from beginning till the end- **TABLE SCAN**

**Example :**

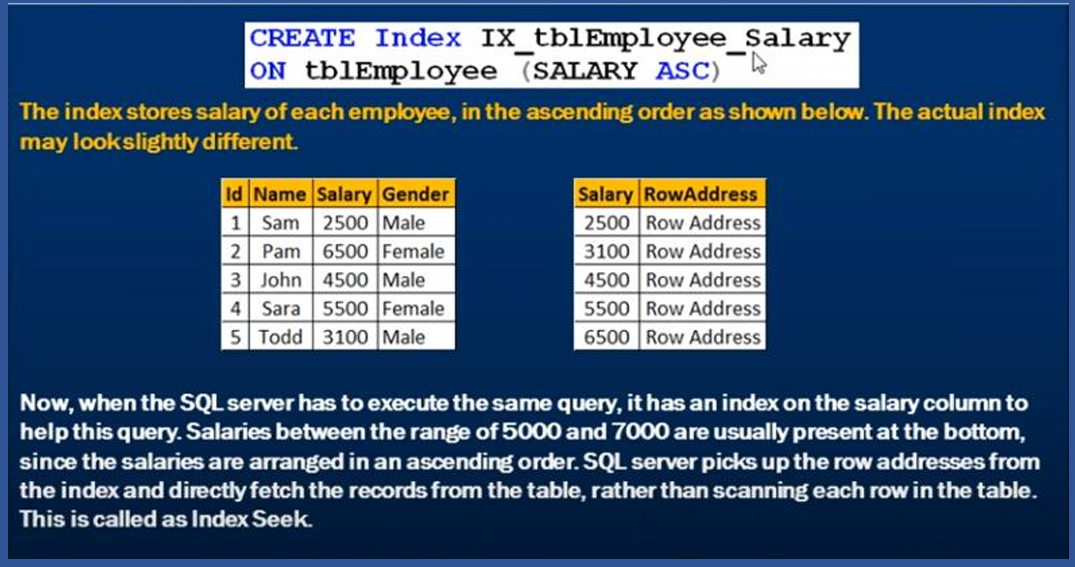
To reference all pages in a book that address a particular subject, you go to the index first, which lists all the topics alphabetically, and then you go to one or more specific page numbers.



**How query scan a table without index :**



**How query scan a table with index**



**Types of indexes:**

1. **Single-Column Index** :

Create a single-column index on customer\_id

CREATE INDEX idx\_customer\_id ON customer\_orders(customer\_id);

1. **Composite Index** :

Create a composite index on customer\_id and order\_date

CREATE INDEX idx\_customer\_order\_date ON customer\_orders(customer\_id, order\_date);

1. **Unique Index** :

Create a unique index on order\_id

CREATE UNIQUE INDEX idx\_unique\_order\_id ON customer\_orders(order\_id)

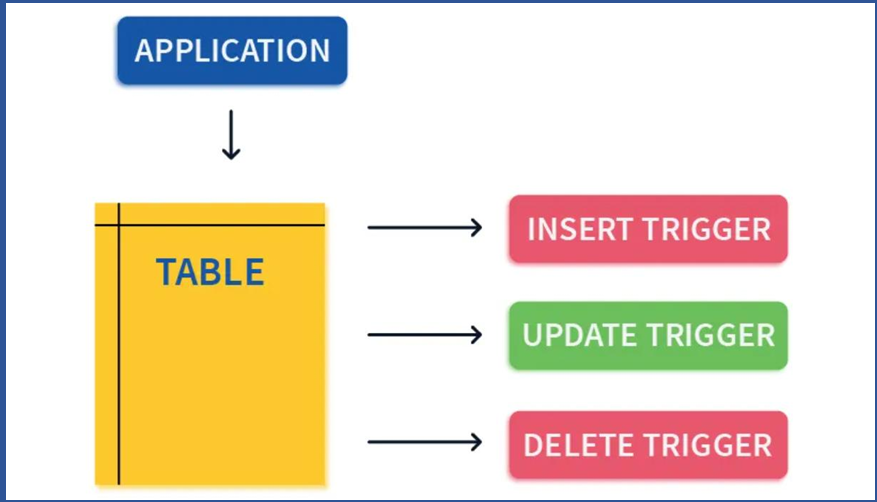
**Practical:**

|  |
| --- |
| **drop** **table** customer\_orders;  -- Create the customer\_orders table  **CREATE** **TABLE** customer\_orders (  order\_id INT **PRIMARY** **KEY** AUTO\_INCREMENT,  order\_date DATE,  customer\_id INT,  order\_total DECIMAL(**10**, **2**)  );  -- Create an index on the customer\_id column  **CREATE** **INDEX** idx\_customer\_id **ON** customer\_orders(customer\_id);  -- Insert sample data  **INSERT** **INTO** customer\_orders (order\_date, customer\_id, order\_total)  **VALUES**  ('2023-09-01', **101**, **150**.**50**),  ('2023-09-02', **102**, **200**.**25**),  ('2023-09-03', **101**, **75**.**80**),  ('2023-09-04', **103**, **120**.**60**),  ('2023-09-05', **102**, **180**.**90**);  **SHOW** **INDEX** **FROM** customer\_orders;  **SELECT** \* **FROM** customer\_orders **WHERE** customer\_id = **102**;  -- Without an index, the database would need to perform a full table scan, examining each row in the customer\_orders table to find all orders with customer\_id equal to 102. This operation can be slow, especially if there are millions of rows in the table.  /\*With the index on the customer\_id column, the database can use the index to quickly locate all rows with customer\_id equal to 102. It doesn't need to scan the entire table, resulting in much faster query performance.  In a real-world scenario, as the size of the customer\_orders table grows, the difference in query performance becomes even more pronounced. The index allows the database to perform the query efficiently, making it ideal for scenarios where you frequently need to retrieve data based on the indexed column's values, such as retrieving all orders for a specific customer in an e-commerce platform, as mentioned in the previous example.  Indexes are a fundamental optimization technique in database design, significantly improving query performance for commonly accessed data.  \*/  **DROP** **INDEX** idx\_customer\_id **ON** customer\_orders; |

1. **Triggers in SQL**

A trigger is a stored procedure that automatically invokes whenever a special event in the database occurs.

For example, a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated in simple words a trigger is a collection of SQL statements with particular names that are stored in system memory. It belongs to a specific class of stored procedures that are automatically invoked in response to database server events. Every trigger has a table attached to it.



**Syntax:**

|  |
| --- |
| **CREATE** **TRIGGER** **trigger\_name**  {**BEFORE** | **AFTER**} {**INSERT** | **UPDATE** | **DELETE**} **ON** **table\_name**  **FOR** **EACH** **ROW**  **BEGIN**  -- SQL statements to be executed when the trigger fires  **END**; |

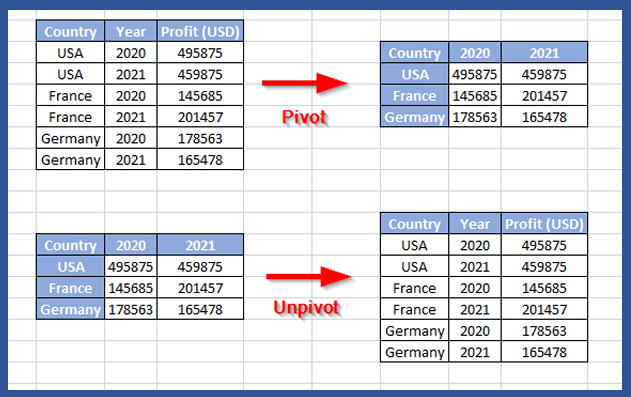
**Practical:**

|  |
| --- |
| **drop** **table** employees;  -- Create the employees table  **CREATE** **TABLE** employees (  employee\_id INT **PRIMARY** **KEY**,  employee\_name VARCHAR(**50**),  salary DECIMAL(**10**, **2**)  );  -- Create the audit\_log table to store salary updates  **CREATE** **TABLE** audit\_log (  log\_id INT AUTO\_INCREMENT **PRIMARY** **KEY**,  employee\_id INT,  old\_salary DECIMAL(**10**, **2**),  new\_salary DECIMAL(**10**, **2**),  update\_timestamp **TIMESTAMP**  );  -- Create a trigger for auditing salary updates  **DELIMITER** $$  **CREATE** **TRIGGER** salary\_update\_audit  **AFTER** **UPDATE** **ON** employees  **FOR** **EACH** **ROW**  **BEGIN**  **INSERT** **INTO** audit\_log (employee\_id, old\_salary, new\_salary, update\_timestamp)  **VALUES** (**OLD**.employee\_id, **OLD**.salary, **NEW**.salary, NOW());  **END**;  $$  **DELIMITER** ;  -- See data before trigger  **select** \* **from** audit\_log;    -- Insert sample data into the employees table  **INSERT** **INTO** employees (employee\_id, employee\_name, salary)  **VALUES**  (**1**, 'John Doe', **50000**.**00**),  (**2**, 'Jane Smith', **55000**.**00**),  (**3**, 'Bob Johnson', **60000**.**00**);  **select** \* **from** employees;    -- Now, let's update an employee's salary and see the trigger in action  **UPDATE** employees  **SET** salary = **55000**  **WHERE** employee\_id = **1**;  **select** \* **from** employees;    **select** \* **from** audit\_log; |

1. **Pivot and unpivot**

Pivoting and unpivoting are techniques used to transform data from a row-wise format to a column-wise format (pivot) or vice versa (unpivot). These operations are often used when you want to reshape your data for reporting or analysis.

* **Pivoting :** Pivoting is the process of converting row-level data into column-level data. This is useful when you have data in a "long" format, and you want to summarize it or make it more readable.
* **Unpivoting :** Unpivoting is the process of converting column-level data into row-level data. This is useful when you have data in a "wide" format, and you want to normalize it or make it suitable for further analysis.



**Practical:**

|  |  |  |
| --- | --- | --- |
| **Pivoting:**   |  | | --- | | **DROP** **TABLE** sales;  **CREATE** **TABLE** sales (  product\_id INT,  **month** VARCHAR(**10**),  sales\_amount DECIMAL(**10**, **2**)  );  **INSERT** **INTO** sales (product\_id, **month**, sales\_amount)  **VALUES**  (**1**, 'Jan', **1000**),  (**1**, 'Feb', **1200**),  (**2**, 'Jan', **800**),  (**2**, 'Feb', **900**);  **select** \* **from** sales;    -- To pivot this data to see total sales by product for each month:  **SELECT**  **month**,  **SUM**(**CASE** **WHEN** product\_id = **1** **THEN** sales\_amount **ELSE** **0** **END**) **AS** product\_1\_sales,  **SUM**(**CASE** **WHEN** product\_id = **2** **THEN** sales\_amount **ELSE** **0** **END**) **AS** product\_2\_sales  **FROM** sales  **GROUP** **BY** **month**;  -- Here, we used conditional aggregation with the SUM function to pivot the data and calculate the total sales for each product by month. |   **Unpivoting:**   |  | | --- | | **CREATE** **TABLE** student\_scores (  student\_id INT,  math\_score INT,  science\_score INT,  history\_score INT  );  **INSERT** **INTO** student\_scores (student\_id, math\_score, science\_score, history\_score)  **VALUES**  (**1**, **95**, **88**, **75**),  (**2**, **88**, **92**, **80**),  (**3**, **90**, **85**, **78**);  **select** \* **from** student\_scores;    -- To unpivot this data and create a structure with rows for each student-subject combination:  **SELECT** student\_id, 'math' **AS** subject, math\_score **AS** score **FROM** student\_scores  **UNION** **ALL**  **SELECT** student\_id, 'science' **AS** subject, science\_score **AS** score **FROM** student\_scores  **UNION** **ALL**  **SELECT** student\_id, 'history' **AS** subject, history\_score **AS** score **FROM** student\_scores;  -- In this example, we used the UNION ALL operator to combine multiple SELECT statements that extract data for each subject column, effectively unpivoting the data. | |

1. **SQL Performance Tuning**

Performance tuning in SQL involves optimizing the performance of your SQL queries and database operations to ensure they run efficiently.

Poorly performing SQL queries can lead to slow response times, which can negatively impact application responsiveness and user experience.

**Methods:**

* INDEX OPTIMIZATION
* QUERY OPTIMIZATION
* DATABASE DESIGN IMPROVEMENTS
* HARDWARE UPGRADES

**Best Practices for SQL Query Optimization:**

Use Where Clause instead of having

Avoid Queries inside a Loop

Use Select instead of Select \*

Add Explain to the Beginning of Queries

Keep Wild cards at the End of Phrases

Use Exist() instead of Count()

7. Create queries with INNER JOIN (not WHERE or cross join)

8. Use LIMIT to sample query results

9. Consider denormalizing data for frequently used reports to reduce the need for complex joins

10. Implement caching mechanisms (e.g., in-memory caching or database query caching) to store and reuse query results for a certain period.

11 Try to run your query during off-peak hours.