



ANNAPOORANA ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai)

Accredited by NAAC with 'A' Grade & Accredited by NBA (Civil, CSE &EEE)

2(f) & 12(B) Status as per UGC Act 1956

ISO 9001:2015 Certified Institution

NH-47, Sankari Main Road, Periyaseeragapadi, Salem-636308, Tamil Nadu.

www.aecsalem.edu.in



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

Subject Code: 22CS406

Subject Name: Database Management Systems Lab

Program Name: B. E., CSE

Year & Sem: II & IV

Regulation: 2022

Prepared By: R. Uma Mageswari/AP-CSE

22CS406 - DBMS Lab Manual (1 to 9)

COURSE OBJECTIVES:

- To learn and implement important commands in SQL.
- To learn the usage of nested and joint queries.
- To understand functions, procedures and procedural extensions of databases.
- To understand design and implementation of typical database applications.
- To be familiar with the use of a front end tool for GUI based application development.

LIST OF EXPERIMENTS:

1. Create a database table, add constraints (primary key, unique, check, Not null), insert rows, Update and delete rows using SQL DDL and DML commands.
 2. Create a set of tables, add foreign key constraints and incorporate referential integrity.
 3. Query the database tables using different 'where' clause conditions and also implement Aggregate functions.
 4. Query the database tables and explore sub queries and simple join operations.
 5. Query the database tables and explore natural, equi and outer joins.
 6. Write user defined functions and stored procedures in SQL.
 7. Execute complex transactions and realize DCL and TCL commands.
 8. Write SQL Triggers for insert, delete, and update operations in a database table.
 9. Create View and index for database tables with a large number of records.
 10. Create an XML database and validate it using XML schema.
 11. Create Document, column and graph based data using NOSQL database tools.
 12. Develop a simple GUI based database application and incorporate all the above mentioned Features
 13. Case Study using any of the real life database applications from the following list
 - a) Inventory Management for a EMart Grocery Shop
 - b) Society Financial Management
 - c) Cop Friendly App – Eseva
 - d) Property Management – eMall
 - e) Star Small and Medium Banking and Finance
- Build Entity Model diagram. The diagram should align with the business and functional

Goals stated in the application.

Apply Normalization rules in designing the tables in scope.

- Prepared applicable views, triggers (for auditing purposes), functions for enabling Enterprise grade features.

- Build PL SQL / Stored Procedures for Complex Functionalities,

Ex EOD Batch Processing for calculating the EMI for Gold Loan for each eligible Customer.

- Ability to showcase ACID Properties with sample queries with appropriate settings

List of Equipments :(30 Students per Batch)

MYSQL / SQL: 30 Users

TOTAL: 45 PERIODS

COURSE OUTCOMES:

At the end of this course, the students will be able to:

CO1: Create databases with different types of key constraints.

CO2: Construct simple and complex SQL queries using DML and DCL commands.

CO3: Use advanced features such as stored procedures and triggers and incorporate in GUI based application development.

CO4: Create an XML database and validate with meta-data (XML schema).

CO5: Create and manipulate data using NOSQL database.

Ex No: 1	Create a database table, add constraints (primary key, unique, check, Not null), Insert rows, Update and delete rows using SQL DDL and DML commands.
-----------------	---

Aim:

To create table and Execute Data Definition Commands, Data Manipulation Commands for Inserting, Deleting, Updating and Retrieving Tables with constraints.

SQL: create command

Create is a DDL SQL command used to create a table or a database in relational database management system.

Creating a Database

To create a database in RDBMS, create command is used. Following is the syntax,

CREATE DATABASE <DB_NAME>;

Example for creating Database

CREATE DATABASE Test;

The above command will create a database named Test, which will be an empty schema without any table.

To create tables in this newly created database, we can again use the create command.

Creating a Table

Create command can also be used to create tables. Now when we create a table, we have to specify the details of the columns of the tables too. We can specify the names and data types of various columns in the create command itself.

Following is the syntax,

CREATE TABLE <TABLE_NAME>

(

column_name1 datatype1,

column_name2 datatype2,

column_name3 datatype3,

column_name4 datatype4

);

Create table command will tell the database system to create a new table with the given table name and column information.

Most commonly used data types for Table columns

Here we have listed some of the most commonly used data types used for columns in tables.

Structured Query Language (SQL) as we all know is the database language by the use of which we can perform certain operations on the existing database and also we can use this language to create a database. SQL uses certain commands like Create, Drop, Insert, etc. to carry out the required tasks.

MySQL String Data Types

Data Type	Description	Size
CHAR(size)	A FIXED length string containing letters, numbers, and special characters. The size parameter determines the length of the column in characters. The default value is 1.	0 - 255 bytes
VARCHAR(size)	A VARIABLE length string containing letters, numbers, and special characters. The size option sets the maximum character length for a column.	0 - 65535 bytes

INT(size)	An integer value. The signed range is -2147483648 to -2147483647. The unsigned range is 0 to 4294967295. The size parameter determines the maximum width of the display (which is 255).	4 bytes
INTEGER(size)	As same as to INT(size).	4 bytes
BIGINT(size)	A large integer. The signed range is -9223372036854775808 to 9223372036854775807. Unsigned numbers range from 0 to 18446744073709551615. The size parameter determines the maximum width of the display (which is 255).	8 bytes

MySQL Date and Time Data Types

Data Type	Description	Format
DATE	It's a date. The permitted value range is '1000-01-01' to '9999-12-31'.	YYYY-MM-DD
DATETIME(fsp)	A combination of date and time. The range supported is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. By including DEFAULT and ON UPDATE in the column definition, automatic initialization and updating to the current date and time is achieved.	YYYY-MM-DD hh:mm
TIMESTAMP(fsp)	The number of seconds is represented in TIMESTAMP values. The acceptable time range is '1970-01-01 00:00:01 UTC' to '2038-01-09 03:14:07 UTC'. Automatic initialization and updating to the current date and time can be defined in the	YYYY-MM-DD hh:mm

These SQL commands are mainly categorized into five categories as:

- 1.DDL – Data Definition Language
- 2.DQL – Data Query Language
- 3.DML – Data Manipulation Language
- 4.DCL – Data Control Language

5. TCL – Transaction Control Language

DDL (Data Definition Language):

DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema.

It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

DDL is a set of SQL commands used to create, modify, and delete database structures but not data.

These commands are normally not used by a general user, who should be accessing the database via an application.

List of DDL commands:

- **CREATE:** This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).
- **DROP:** This command is used to delete objects from the database.
- **ALTER:** This is used to alter the structure of the database.
- **TRUNCATE:** This is used to remove all records from a table, including all spaces allocated for the records are removed.
- **COMMENT:** This is used to add comments to the data dictionary.
- **RENAME:** This is used to rename an object existing in the database.

DQL (Data Query Language):

DQL statements are used for performing queries on the data within schema objects. The purpose of the DQL Command is to get some schema relation based on the query passed to it.

It includes the **SELECT** statement. This command allows getting the data out of the database to perform operations with it.

List of DQL:

- **SELECT:** It is used to retrieve data from the database.

DML (Data Manipulation Language):

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

List of DML commands:

- **INSERT :** It is used to insert data into a table.
- **UPDATE:** It is used to update existing data within a table.
- **DELETE :** It is used to delete records from a database table.
- **LOCK:** Table control concurrency.
- **CALL:** Call a PL/SQL or JAVA subprogram.
- **EXPLAIN PLAN:** It describes the access path to data.

Data Definition Language

In SQL DDL commands are used to create and modify the structure of a database and database objects. These commands are **CREATE**, **DROP**, **ALTER**, **TRUNCATE**, and **RENAME**. Let us discuss these commands one at a time.

CREATE

The **syntax** for create command is:

For creating a database:

```
CREATE DATABASE database_name;
```

Creates a database named 'database_name'. Let us create our own database and table which we will use as a reference throughout this article:

```
CREATE DATABASE testdb;
```

For creating a table:

```
CREATE TABLE table_name(  
    column1 datatype,  
    column2 datatype,  
    ....  
    columnN datatype  
);
```

This creates a new table with the name 'table_name' in our database. It will have N columns for each of the same datatypes as mentioned adjacent to it in the create syntax.

For instance:

```
CREATE TABLE my_table(  
    first_name varchar(20),  
    age INT  
);
```

This creates an empty table. For clarity suppose we add an entry (row) in the column (as in the image) the result will be:

first_name	age
myFirstName	20

DROP

To drop a table:

```
DROP TABLE table_name;
```

Deletes the table named 'table_name' if present.

Suppose if we drop our my_table, the result is as:

```
drop table my_table;
```

Output:

```
Query OK, 0 rows affected (0.01 sec)
```

To drop a database:

```
DROP DATABASE database_name;
```

This removes the database named 'database_name' if present. Suppose we try this with our database, the result looks like this:

```
drop database testdb;
```

Output:

```
Query OK, 0 rows affected (0.01 sec)
```

ALTER

Following is the syntax to alter the contents of a table:

```
ALTER TABLE table_name  
ADD column_name column_definition;
```

Let's try performing this operation on our table my_table:

```
alter table my_table  
add column employed boolean;
```

Output:

```
| first_name | age |  
|:-----:|:---:|  
| myFirstName | 20 |
```

In the above example, we are adding a column to our table. Apart from that, we can perform other operations such as dropping a column, modifying it, etc.

To change the properties of a column such as its type, type capacity, add constraints, etc:

```
ALTER TABLE table_name  
ALTER COLUMN column_name column_type;
```

Lets try using it in our database:

```
alter my_table modify column  
first_name varchar(25);
```

Output:

```
Query OK, 0 rows affected (0.01 sec)
```

We changed the column first_name from data type varchar(20) to varchar(25).

Similarly, we can drop a column using alter with the following **syntax**:

```
ALTER TABLE table_name  
DROP COLUMN column_name;
```

We can remove our column 'employed' as:

```
alter table my_table  
drop column employed;
```

Output:

```
Query OK, 0 rows affected (0.01 sec)
```

TRUNCATE

This command is similar to the drop table command. The only difference is that while the drop command removes the table as well as its contents, the truncate command only erases the contents of the table's contents and not the table itself. Let us take an example to be clearer:

```
truncate table my_table;  
select * from my_table;
```

Output:

```
Query OK, 0 rows affected (0.01 sec)  
Empty set (0.00 sec)
```

As shown in the image above, truncate has removed the contents of the table. The table, though now empty, still exists. Think of truncating a table as emptying it rather than deleting it.

RENAME

This command is used to change the name of an existing table. The syntax is

```
RENAME TABLE table_name TO table_name_new;
```

The following image shows how to rename our table from 'my_table' to 'some_table':

```
rename table my_table to some_table;
```

Output:

```
Query OK, 0 rows affected (0.01 sec)
```

Data Manipulation Language

DML is used for inserting, deleting, and updating data in a database. It is used to retrieve and manipulate data in a relational database. It includes INSERT, UPDATE, and DELETE. Let's discuss these commands one at a time.

INSERT

Insert statement is used to insert data in a SQL table. Using the Insert query, we can add one or more rows to the table. Following is the syntax of the MySQL INSERT Statement.

```
INSERT INTO table_name  
(attribute1, attribute2, ...)  
VALUES(val1, val2, ...)
```

Let's take a quick instance for understanding this better. Suppose that we need to insert 2 rows {"myName3", 2, true} and {"myName4", 28, false}. This is how we would accomplish that:

```
insert into my_table  
values("myName3", 5, false),  
("myName4", 51, true);
```

Output:

```
Query OK, 2 rows affected (0.01 sec)
```

We don't necessarily have to include all column values. We can for example omit the age column as shown below:

```
insert into my_table  
(first_name, employed)  
values("myName4", true);
```

Output:

```
Query OK, 1 rows affected (0.01 sec)
```

This is also a fine example of how we insert just one row.

UPDATE

This command is used to make changes to the data in the table. Its syntax is:

```
UPDATE table_name
SET column1 = val1,
column2 = val2,
...
WHERE CLAUSE;
```

How about we update the employment status of myName of age 12 which has it as NULL? This is how we achieve it:

```
update my_table
set employed=true
where age=12;
```

Output:

```
Query OK, 1 rows affected (0.01 sec)
```

Let's batch-update all our rows without filtering them on any condition. Suppose we want to increment the value of every row in the age column. The following query is what we need:

```
update my_table
set age = age + 1;
```

Output:

```
Query OK, 5 rows affected (0.01 sec)
```

DELETE

This command is used to remove a row from a table. the syntax for delete is

```
DELETE FROM table_name
WHERE CLAUSE;
```

The where clause is optional. To delete the row with first_name "myName5 run this query:

```
delete from my_table
where first_name = "myName5";
```

Output:

```
Query OK, 0 rows affected (0.01 sec)
```

It is to be noted that omitting the use where clause results in the emptying of the table just as truncating does.

Data Query Language

DQL commands are used for fetching data from a relational database. They perform read-only queries of data. The only command, '**SELECT**' is equivalent to the projection operation in relational algebra. It command selects the attribute based on the condition described by the **WHERE** clause and returns them.

Select is one of the most important SQL commands. It is used to retrieve data from a database.

The syntax for the SELECT statement is:

```
SELECT column1 as c1
column2 as c2
...
from table_name;
```

'as c1' implies that the result table column that holds the data from column1 of the original table, will be identified as 'c1' and its usage is optional.

To get the age of all our rows, we would run the first of the following query:

```
select age as myAge
from my_table;

select * from my_table;
```

Output:

```
| myAge |
|-----|
| 47    |

| first_name | age | employed |
|-----|-----|-----|
| myName     | 47  | 1        |
```

Assume the present state of our table is as shown in the image below.

```
select * from my_table;
```

Output:

```
| first_name | age | employed |
|-----|-----|-----|
| myName     | 47  | 1        |
| maName1    | 10  | 0        |
| myName2    | 20  | 0        |
| myName3    | 32  | 1        |
```

Let's say that we need to fetch all rows where the age is strictly greater than 20. The query that we need is as follows:

```
select * from
my_table where age>20;
```

Output:

```
| first_name | age | employed |
|-----|-----|-----|
| myName    | 47  | 1        |
| myName2   | 32  | 1        |
```

To extract all the rows where the name matches a certain pattern we use the 'like' operator. The following snippet shows its action:

```
select * from my_table
where first_name like "myName_";
```

Output:

```
| first_name | age | employed |
|-----|-----|-----|
| myName0    | 10  | 1        |
| maName1    | 10  | 0        |
| myName2    | 20  | 0        |
```

'like' is an operator used for pattern matching. It uses two special characters embedded in the string to be matched; The percent sign '%' represents zero, one, or any number of characters, while the underscore sign '_' represents a single character.

In the previous example, we used underscore, and therefore all rows except the first were returned because 'myName_' implies a matching string of the form 'myName' + an additional character. The last 3 strings are of this form and hence are matched, unlike the first one which does not have any additional character present at last.

Similarly, we can obtain all the rows in the first_name field which ends with zero ('0').

```
select * from my_table
where first_name like "%0";
```


Output:

```
| first_name | age | employed |
|-----|-----|-----|
| myName0   | 10  | 0        |
```

Syntax of SELECT Statement in SQL

The basic syntax for using SELECT is to specify the column names we want to retrieve, followed by the FROM clause to designate the name of the database table:

```
SELECT
    column1,
    column2,
    .....
FROM table name;
```

In case we want to retrieve all the columns, we can replace writing individual column names with *.

```
SELECT * FROM table name;
```

Examples of SELECT Statement in SQL

1. Basics

- **Exploring the Data:** We can simply use SELECT to look up the data present in the table. Let us see how we can do that in the following example:

```
SELECT * FROM student;
```

Output

S_ID	Name	State	Age	Marks
1	LOKI	DELHI	19	90
2	KISHAN	KERALA	20	78
3	RISHI	ASSAM	21	82
4	SANJOY	KOLKATA	18	87
5	VISHAL	TELENGANA	20	92
6	PRIYA	ASSAM	19	68
7	PURU	BIHAR	17	70
8	RIYA	KARNATAKA	19	72

This selects all of the columns and fields from the student table.

In case we want to select only the Name and State columns, we can do so by using the following query:

```
SELECT
    Name AS 'First Name',
    State
FROM student;
```

Output

First Name	State
LOKI	DELHI
KISHAN	KERALA
RISHI	ASSAM
SANJOY	KOLKATA
VISHAL	TELENGANA
PRIYA	ASSAM
PURU	BIHAR
RIYA	KARNATAKA

The column Name has been renamed temporarily to First Name here. This is done using the AS clause which can be used to add aliases to column names or even table names.

- **WHERE:** While dealing with a large data set, the data usually needs to be filtered for easy exploration and examination. The *WHERE* clause helps us in this regard.

Let us say, we need to find the Name and Age students from ASSAM. We can use the following query for filtering the data:

```
SELECT
    Name,
    Age
FROM student
WHERE State = "Assam";
```

Output

Name	Age
RISHI	21
PRIYA	19

A comparison operator, i.e. equals, was used in the example above. We can also use the other typical comparison operators with WHERE such as greater than, less than, greater than, or equal to, etc.

```
SELECT
    Name,
    Age
FROM student
WHERE Age > 18;
```

Output

Name	Age
LOKI	19
KISHAN	20
RISHI	21
VISHAL	20
PRIYA	19
RIYA	19

WHERE can also be paired with LIKE for finding words in a string or for pattern matching. In case, we want to check a range or filter by passing a list we can use IN. Let us take one example to show these:

```
SELECT
    Name,
    Age
FROM student
WHERE Age IN (18, 19, 21)
AND Name LIKE 'R%';
```

Output

Name	Age
RISHI	21
RIYA	19

This returns the Names and Ages of the student(s) whose names start with R **and** age is either 18, 19, or 20. Multiple filters can be chained together using **AND/OR**.

- **ORDER BY:** The order of functions is essential in SQL. Sorting of the data can be done in both ascending and descending order. ASC and DESC keywords is used for this purpose.

NOTE: If neither of the keywords is used then the default order is ascending.

Let us sort the earlier student table according to their ages in descending order. We can use the query as follows:

```
SELECT
    Name,
    Age
FROM student
ORDER BY Age DESC;
```

Output

Name	Age
RISHI	21
KISHAN	20
VISHAL	20
LOKI	19
PRIYA	19
RIYA	19
SANJOY	18
PURU	17

- **DISTINCT:** It is used to retrieve unique elements from the table. For instance, some of the ages repeat in the table. To find all the unique ages we can use the following command:

```
SELECT DISTINCT
    Age
FROM student;
```

Output

Age
19
20
21
18
17

- **LIMIT:** We can restrict the number of records we want to access by using the LIMIT query.
- Let's say we need the five youngest students. We can run the following command for this:

```
SELECT
    Name,
    Age
FROM student
ORDER BY Age
LIMIT 5;
```

Output

Name	Age
PURU	17
SANJOY	18
LOKI	19
PRIYA	19
RIYA	19

SOL CASE

The **CASE** is a statement that operates if-then-else type of logical queries. This statement returns the value when the specified condition evaluates to True. When no condition evaluates to True, it returns the value of the ELSE part.

In Structured Query Language, CASE statement is used in SELECT, INSERT, and DELETE statements with the following three clauses:

1. WHERE Clause

2. ORDER BY Clause

3. GROUP BY Clause

This statement in SQL is always followed by at least one pair of WHEN and THEN statements and always finished with the END keyword.

The CASE statement is of two types in relational databases:

1. Simple CASE statement
2. Searched CASE statement

Syntax of CASE statement in SQL

CASE <expression>

WHEN condition_1 THEN statement_1

WHEN condition_2 THEN statement_2

WHEN condition_N THEN statement_N

ELSE result

END;

- Here, the CASE statement evaluates each condition one by one.

- If the expression matches the condition of the first WHEN clause, it skips all the further WHEN and THEN conditions and returns the statement_1 in the result.
- If the expression does not match the first WHEN condition, it compares with the seconds WHEN condition. This process of matching will continue until the expression is matched with any WHEN condition.
- If no condition is matched with the expression, the control automatically goes to the ELSE part and returns its result. In the CASE syntax, the ELSE part is optional.
- In Syntax, CASE and END are the most important keywords which show the beginning and closing of the CASE statement.

Examples of CASE statement in SQL

Let's take the Student_Details table, which contains roll_no, name, marks, subject, and city of students.

Roll_No	Stu_Name	Stu_Subject	Stu_Marks	Stu_City
2001	Akshay	Science	92	Noida
2002	Ram	Math	49	Jaipur
2004	Shyam	English	52	Gurgaon
2005	Yatin	Hindi	45	Lucknow
2006	Manoj	Computer	70	Ghaziabad
2007	Sheetal	Math	82	Noida
2008	Parul	Science	62	Gurgaon

Example 1: The following SQL statement uses single WHEN and THEN condition to the CASE statement:

SELECT Roll_No, Stu_Name, Stu_Subject, Stu_marks,

CASE

WHEN Stu_Marks >= 50 THEN 'Student_Passed'

ELSE 'Student_Failed'

END AS Student_Result

FROM Student_Details;

Explanation of above query:

Here, the CASE statement checks that if the **Stu_Marks** is greater than and equals 50, it returns **Student_Passed** otherwise moves to the **ELSE** part and returns **Student_Failed** in the **Student_Result** column.

Output:

Roll_No	Stu_Name	Stu_Subject	Stu_Marks	Student_Result
2001	Akshay	Science	92	Student_Passed
2002	Ram	Math	49	Student_Failed
2004	Shyam	English	52	Student_Passed
2005	Yatin	Hindi	45	Student_Failed
2006	Manoj	Computer	70	Student_Passed
2007	Sheetal	Math	82	Student_Passed
2008	Parul	Science	62	Student_Passed

Example 2: The following SQL statement adds more than one WHEN and THEN condition to the CASE statement:

```
SELECT Roll_No, Stu_Name, Stu_Subject, Stu_marks,
```

```
CASE
```

```
WHEN Stu_Marks >= 90 THEN 'Outstanding'
```

```
WHEN Stu_Marks >= 80 AND Stu_Marks < 90 THEN 'Excellent'
```

```
WHEN Stu_Marks >= 70 AND Stu_Marks < 80 THEN 'Good'
```

```
WHEN Stu_Marks >= 60 AND Stu_Marks < 70 THEN 'Average'
```

```
WHEN Stu_Marks >= 50 AND Stu_Marks < 60 THEN 'Bad'
```

```
WHEN Stu_Marks < 50 THEN 'Failed'
```

```
END AS Stu_Remarks FROM Student_Details;
```

Aim:

To create a set of tables, add foreign key constraints and incorporate referential integrity

Key Constraints in DBMS:

- Constraints or nothing but the rules that are to be followed while entering data into columns of the database table
- Constraints ensure that data entered by the user into columns must be within the criteria specified by the condition
- For example, if you want to maintain only unique IDs in the employee table or if you want to enter only age under 18 in the student table etc
- **We have 5 types of key constraints in DBMS**
 - **NOT NULL:** ensures that the specified *column doesn't contain a NULL value.*
 - **UNIQUE:** *provides a unique/distinct values to specified columns.*
 - **DEFAULT:** *provides a default value to a column if none is specified.*
 - **CHECK:** *checks for the predefined conditions before inserting the data inside the table.*
 - **PRIMARY KEY:** *it uniquely identifies a row in a table.*
 - **FOREIGN KEY:** ensures *referential integrity* of the relationship

Primary Key:

```
CREATE TABLE users(  
    user_id INT AUTO_INCREMENT PRIMARY KEY,  
    username VARCHAR(40),  
    password VARCHAR(255),  
    email VARCHAR(255)  
);
```

```
CREATE TABLE roles(  
    role_id INT AUTO_INCREMENT,  
    role_name VARCHAR(50),  
    PRIMARY KEY(role_id)  
);
```

Using Alter Table to Define a Primary MySQL Key

Step 1: If due to some reason, a table does not have a Primary Key, you can leverage the ALTER TABLE statement to supply a Primary Key to the table as mentioned in the code snippet below:

```
ALTER TABLE table_name  
ADD PRIMARY KEY(column_list);
```

Step 2: The example mentioned below supplies the id column to the Primary Key. But first, you need to create the pkdemos table without a Primary Key:

```
CREATE TABLE pkdemos(  
    id INT,  
    title VARCHAR(255) NOT NULL  
);
```

Step 3: Next, you need to add the Primary Key to the table by leveraging the ALTER TABLE statement as follows:

```
ALTER TABLE pkdemos  
ADD PRIMARY KEY(id);
```

Step 4: If you want to add a Primary Key to a table that already has data, then, the data in the column(s) which will be encapsulated within the PRIMARY KEY must be unique and NOT NULL

What is a MySQL Unique Key?

A group of one or more table fields/columns that uniquely identify a record in a database table is known as a unique key in MySQL Keys.

It's similar to a primary key in that it can only accept one null value and cannot have duplicate values.

Both the unique key and the primary key ensure that a column or set of columns is unique.

Using Create Table to Define a Unique MySQL Key

Step 1: You can use the following syntax if you want to define the Unique Key for a solitary column:

```
CREATE TABLE <table_name>
(
Column_name1 datatype() UNIQUE,
Column_name2 datatype(),
);
```

Step 2: If you wish to define more than one Unique Key within a table you can use the following syntax:

```
CREATE TABLE <table_name>
(
Column_name1 datatype(),
Column_name2 datatype(),...
Column_namen datatype(),
UNIQUE (column_name1, column_name2)
);
```

Step 3: You will be checking this with the help of an example now. Create a table as follows:

```
CREATE TABLE VATSA(
    ID INT AUTO_INCREMENT PRIMARY KEY,
    Company_name varchar(100) UNIQUE,
    Address varchar(250) UNIQUE
);
```

Step 4: Next, you need to insert some rows in this table as follows:

```
INSERT INTO VATSA VALUES (1, 'Vatsa_Empire', '132, ABC, Near Royal Club, BSR');
INSERT INTO VATSA VALUES (2, 'Vatsa_Hotel', '138, ABC, Near Royal Club, BSR');
```

Step 5: Now, if you try to fetch the values from this table, here's the output you can expect:

Step 6: Now, if you try to insert a new record with existing data, MySQL will throw an error for the following code snippet:

```
INSERT INTO VATSA VALUES (10, 'Vatsa_Empire', '204, ABC, Near Royal Club, BSR');
```

MySQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a column it will allow only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

```
CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CHECK (Age>=18)  
);
```

MySQL DEFAULT Constraint

The DEFAULT constraint is used to set a default value for a column.

The default value will be added to all new records, if no other value is specified.

```
CREATE TABLE cart_items  
(  
    item_id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(255) NOT NULL,  
    quantity INT NOT NULL,  
    price DEC(5,2) NOT NULL,  
    sales_tax DEC(5,2) NOT NULL DEFAULT 0.1,  
    CHECK(quantity > 0),  
    CHECK(sales_tax >= 0)  
);
```

DESC cart_items;

Code language: SQL (Structured Query Language) (sql)

Output:

Field	Type	Null	Key	Default	Extra
item_id	int	NO	PRI	NULL	auto_increment
name	varchar(255)	NO		NULL	
quantity	int	NO		NULL	
price	decimal(5,2)	NO		NULL	
sales_tax	decimal(5,2)	NO		0.10	

INSERT INTO cart_items(name, quantity, price)

VALUES('Keyboard', 1, 50);

SELECT * FROM cart_items;

Output:

item_id	name	quantity	price	sales_tax
1	Keyboard	1	50.00	0.10

INSERT INTO cart_items(name, quantity, price, sales_tax)

VALUES('Battery',4, 0.25 , DEFAULT);

```
SELECT * FROM cart_items;
```

Output:

```
+-----+-----+-----+-----+-----+
| item_id | name    | quantity | price | sales_tax |
+-----+-----+-----+-----+-----+
| 1 | Keyboard | 1 | 50.00 | 0.10 |
| 2 | Battery  | 4 | 0.25  | 0.10 |
+-----+-----+-----+-----+-----+
```

Adding a DEFAULT constraint to a column

To add a default constraint to a column of an existing table, you use the ALTER TABLE statement:

```
ALTER TABLE table_name
```

```
ALTER COLUMN column_name SET DEFAULT default_value;
```

Code language: SQL (Structured Query Language) (sql)

The following example adds a DEFAULT constraint to the quantity column of the cart_items table:

```
ALTER TABLE cart_items
```

```
ALTER COLUMN quantity SET DEFAULT 1;
```

Code language: SQL (Structured Query Language) (sql)

If you describe the cart_items table, you'll see the changes:

```
DESC cart_items;
```

Code language: SQL (Structured Query Language) (sql)

Output:

```
+-----+-----+-----+-----+-----+
| Field  | Type    | Null | Key | Default | Extra    |
+-----+-----+-----+-----+-----+
| item_id | int     | NO   | PRI | NULL    | auto_increment |
```

name	varchar(255)	NO		NULL	
quantity	int	NO		1	
price	decimal(5,2)	NO		NULL	
sales_tax	decimal(5,2)	NO		0.10	
+-----+-----+-----+-----+-----+					

The following statement inserts a new row into the cart_items table without specifying a value for the quantity column:

```
INSERT INTO cart_items(name, price, sales_tax)
VALUES('Maintenance services',25.99, 0)
```

Code language: SQL (Structured Query Language) (sql)

The value of the quantity column will default to 1:

```
SELECT * FROM cart_items;
```

Code language: SQL (Structured Query Language) (sql)

Output:

+-----+-----+-----+-----+-----+					
item_id	name		quantity	price	sales_tax
+-----+-----+-----+-----+-----+					
1	Keyboard		1	50.00	0.10
2	Battery		4	0.25	0.10
3	Maintenance services		1	25.99	0.00
+-----+-----+-----+-----+-----+					

Removing a DEFAULT constraint from a column

To remove a DEFAULT constraint from a column, you use the ALTER TABLE statement:

```
ALTER TABLE table_name
```

```
ALTER column_name DROP DEFAULT;
```

Code language: SQL (Structured Query Language) (sql)

The following example removes the DEFAULT constraint from the quantity column of the cart_items table:

```
ALTER TABLE cart_items
```

```
ALTER COLUMN quantity DROP DEFAULT;
```

Code language: SQL (Structured Query Language) (sql)

And here's the new cart_items structure:

```
DESC cart_items;
```

Code language: SQL (Structured Query Language) (sql)

Output:

Field	Type	Null	Key	Default	Extra
item_id	int	NO	PRI	NULL	auto_increment
name	varchar(255)	NO		NULL	
quantity	int	NO		NULL	
price	decimal(5,2)	NO		NULL	
sales_tax	decimal(5,2)	NO		0.10	

We will discuss the overview of foreign keys and will discuss how to add a foreign key using ALTER in MySQL step by step. Let's discuss it one by one.

Foreign key:

If an attribute is a primary key in one table but was not used as a primary key in another table then the attribute which is not a primary key in the other table is called a foreign key. If the changes made or any data is manipulated in any of the tables the changes get reflected in both the tables with the help of foreign key constraint.

Steps to add a foreign key using ALTER in MySQL :

Here let us see how to add an attribute of student which is the primary key in the student table as a foreign key in another table exam as follows.

Step-1: Creating a database university :

Here, you will see how to create a database in MySQL as follows.

```
CREATE DATABASE university;
```

```
USE university;
```

```
CREATE TABLE student
```

```
(  
    student_id INT PRIMARY KEY,  
    student_name varchar,  
    student_branch varchar  
);
```

Here, you will see how to verify the table as follows.

```
DESCRIBE student;
```

Output :

Here, as you can see in the description the key column of student_id is PRI which means it is the primary key in that table student.

Field	Type	Null	Key	Default	Extra
student_id	int	NO	PRI	NULL	
student_name	varchar(20)	YES		NULL	
student_branch	varchar(20)	YES		NULL	

Step-5: Creating another table exam:

In this step, you will see one more table for reference.

```
CREATE TABLE exam
```

```
(
  exam_id INT PRIMARY KEY,
  exam_name varchar(20)
);
```

Step-6: Adding another column student_id into the exam table :

Here, you will see how to add another column student_id into the exam table as follows.

```
ALTER TABLE exam
```

```
ADD COLUMN student_id INT;
```

Step-7: Making a foreign key :

Here, you will see how to make the student_id attribute foreign key in the exam table which is the primary key in the student table as follows.

Syntax –

```
ALTER TABLE table_name
```

```
ADD FOREIGN KEY (column_name)
```

```
REFERENCE table_name(Referencing column_name in table_name);
```

Query –

```
ALTER TABLE exam
```

```
ADD FOREIGN KEY(student_id)
```

```
REFERENCES student(student_id);
```

Step-9: Verifying the exam table :

Here, you will see the description of the exam table as follows.

DESCRIBE exam;

Output :

Now as you can see in the description of the table exam one more column student_id is added and in the Key column of description, the student_id has MUL which means it is a foreign key.

Field	Type	Null	Key	Default	Extra
exam_id	int	NO	PRI	NULL	
exam_name	varchar(20)	YES		NULL	
student_id	int	YES	MUL	NULL	

Ex No: 3

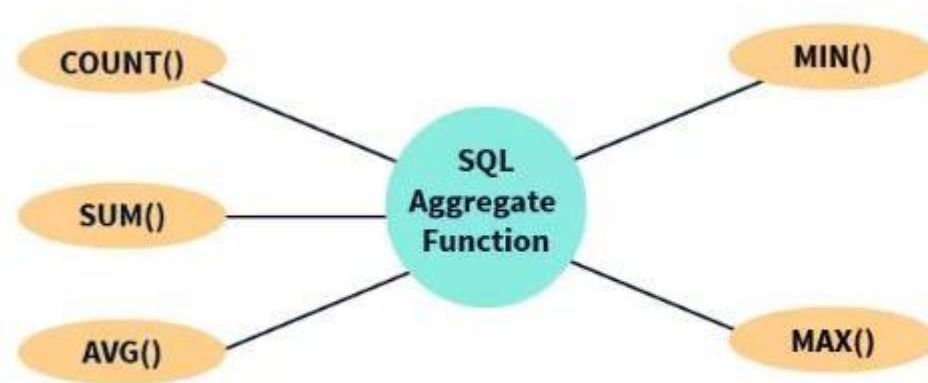
Query the database tables using different 'where' clause conditions and also implement Aggregate functions

Aim:

To Query the database tables using different 'where' clause conditions and also implement Aggregate functions.

Types of Aggregate Functions in SQL

Aggregate functions in SQL can be of the following types as shown in the figure. We will be understanding the working of these functions one by one.



Employee_ID	Name	Department	Salary
1	Ram	Marketing	80000
2	Henry	Production	76000
3	Disha	R&D	76000
4	Helen	R&D	84000
5	Meera	Marketing	80000
6	Ashish	Production	64000
7	Bob	Production	60000
8	Hari	R&D	60000
9	Preeti	Marketing	NULL
10	Mark	Production	66000

1. COUNT() Function

The COUNT() aggregate function returns the total number of rows from a database table that matches the defined criteria in the SQL query.

Syntax:

```
COUNT (*) OR COUNT (COLUMN NAME)
```

COUNT(*) returns the total number of rows in a given table. COUNT(COULUMN_NAME) returns the total number of non-null values present in the column which is passed as an argument in the function.

Example:

Suppose you want to know the total number of employees working in the organization. You can do so by the below-given query.

```
SELECT COUNT (*) FROM EMP DATA;
```

As COUNT(*) returns the total number of rows and the table named EMP_DATA provided above consists of 10 rows, so the COUNT(*) function returns 10. The output is printed as shown below.

Output:

```
COUNT (*)  
10
```

Note: Except for COUNT(*), all other SQL aggregate functions ignore NULL values.

Suppose you need to count the number of people who are getting a salary. The query given below can help you achieve this.

```
SELECT COUNT (Salary) FROM EMP DATA;
```

Here, the Salary column is passed as a parameter to the COUNT() function, and hence, this query returns the number of non NULL values from the column Salary, i.e. 9.

Output:

```
COUNT (Salary)  
9
```

Suppose you need to count the number of distinct departments present in the organization. The following query can help you achieve this.

```
SELECT COUNT (DISTINCT Department) FROM EMP DATA;
```

The above query returns the total number of distinct non NULL values over the column Department i.e. 3 (Marketing, Production, R&D). The **DISTINCT** keyword makes sure that only non-repetitive values are counted.

Output:

```
COUNT(DISTINCT Department)
3
```

What if you want to calculate the number of people whose salaries are more than a given amount(say 70,000)? Check out the example below.

```
SELECT COUNT(Salary) WHERE Salary >= 70000 FROM EMP_DATA;
```

The query returns the number of rows where the salary of the employee is greater than or equal to 70,000 i.e 5.

Output:

```
COUNT(Salary)
5
```

2. SUM() Function

The SUM() function takes the name of the column as an argument and returns the sum of all the non NULL values in that column. It works only on numeric fields(i.e the columns contain only numeric values). When applied to columns containing both non-numeric(ex - strings) and numeric values, only numeric values are considered. If no numeric values are present, the function returns 0.

Syntax:

The function name is SUM() and the name of the column to be considered is passed as an argument to the function.

```
SUM(COLUMN_NAME)
```

```
SELECT SUM(Salary) FROM EMP_DATA;
```

The above-mentioned query returns the sum of all non-NULL values over the column Salary i.e $80000 + 76000 + 76000 + 84000 + 80000 + 64000 + 60000 + 60000 + 66000 = 646000$

Output:

```
SUM(Salary)
646000
```

What if you need to consider only distinct salaries? The following query will help you achieve that.

```
SELECT SUM(DISTINCT Salary) FROM EMP_DATA;
```

The DISTINCT keyword makes sure that only non-repetitive values are considered. The query returns the sum of all distinct non NULL values over the column Salary i.e. $80000 + 76000 + 84000 + 64000 + 60000 + 66000 = 430000$

Output:

```
SUM(DISTINCT Salary)
430000
```

Suppose you need to know the collective salaries for each department(say Marketing). The query given below can help you achieve this.

```
SELECT SUM(SALARY) FROM EMP_DATA WHERE Department = "Marketing";
```

The query returns the sum of salaries of employees who are working in the Marketing Department i.e 80000 + 80000 = 160000

Output:

```
SUM(Salary)
160000
```

Note: There are 3 rows consisting of Marketing as Department value but the third value is a NULL value. Thus, the sum is returned considering only the first two entries having Marketing as Department.

3.AVG() Function

The AVG() aggregate function uses the name of the column as an argument and returns the average of all the non NULL values in that column. It works only on numeric fields(i.e the columns contain only numeric values).

Syntax:

The function name is AVG() and the name of the column to be considered is passed as an argument to the function.

```
AVG(COLUMN NAME)
```

Example:

To obtain the average salary of an employee of an organization, the following query can be used.

```
SELECT AVG(Salary) FROM EMP_DATA;
```

Here, the column name Salary is passed as an argument and thus the values present in column Salary are considered. The above query returns the average of all non NULL values present in the Salary column of the table.

Average = $(80000 + 76000 + 76000 + 84000 + 80000 + 64000 + 60000 + 60000 + 66000) / 9 = 646000 / 9 = 71777.77777$

Output:

```
AVG(Salary)
71777.77777
```

4. MIN() Function

The MIN() function takes the name of the column as an argument and returns the minimum value present in the column. MIN() returns NULL when no row is selected.

Syntax:

The function name is MIN() and the name of the column to be considered is passed as an argument to the function.

```
MIN(COLUMN_NAME)
```

```
SELECT MIN(Salary) FROM EMP_DATA;
```

The query returns the minimum value of all the values present in the mentioned column i.e 60000.

Output:

```
MIN(Salary)
60000
```

Suppose you need to know the minimum salary of an employee belonging to the Production department. The following query will help you achieve that.

```
SELECT MIN(Salary) FROM EMP_DATA WHERE Department = "Production";
```

The query returns the minimum value of all the values present in the mentioned column and has Production as Department value i.e 60000.

Output:

```
MIN(Salary)
60000
```

5. MAX() Function

The MAX() function takes the name of the column as an argument and returns the maximum value present in the column. MAX() returns NULL when no row is selected.

Syntax:

The function name is MAX() and the name of the column to be considered is passed as an argument to the function.

```
MAX(COLUMN_NAME)
```

To get a better idea of how the function works, let's look at some examples.

Example:

Suppose you want to find out what is the maximum salary that is provided by the organization. The MAX() function can be used here with the column name as an argument.

```
SELECT MAX(Salary) FROM EMP_DATA;
```

The query returns the maximum value of all the values present in the mentioned column i.e 84000.

Output:

```
MAX(Salary)
84000
```

Suppose you need to know the maximum salary of an employee belonging to the R&D department. The following query will help you achieve that.

```
SELECT MAX(Salary) FROM EMP_DATA WHERE Department="R&D";
```

The query returns the maximum value of all the values present in the mentioned column and has R&D as Department value i.e 84000.

```
MAX(Salary)
84000
```


Aim:

To Query the database tables and explore sub queries and simple join operations.

Description:

- An **SQL Join statement** is used to combine data or rows from two or more tables based on a common field between them.
- A **sub query** is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another sub query.
- **Joins and sub queries** are both used to combine data from different tables into a single result.

What is a Subquery?

A *subquery* is a nested query (inner query) that's used to filter the results of the outer query. Subqueries can be used as an alternative to joins. A subquery is typically nested inside the WHERE clause.

SQL Sub Query

A Subquery is a query within another SQL query and embedded within the WHERE clause.

Important Rule:

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

1. Subqueries with the Select Statement

SQL Subqueries are most frequently used with the Select statement.

Syntax

SELECT column_name

FROM table_name

WHERE column_name expression operator

(SELECT column_name from table_name WHERE ...);

Example

Consider the EMPLOYEE table have the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
6	Harry	42	China	4500.00
7	Jackson	25	Mizoram	10000.00

The subquery with a SELECT statement will be:

SELECT *

FROM EMPLOYEE

WHERE ID IN (SELECT ID

FROM EMPLOYEE

WHERE SALARY > 4500);

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
7	Jackson	25	Mizoram	10000.00

3. Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

Syntax

UPDATE table

SET column_name = new_value

WHERE VALUE OPERATOR

(SELECT COLUMN_NAME

FROM TABLE_NAME

WHERE condition);

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table.

The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

UPDATE EMPLOYEE

SET SALARY = SALARY * 0.25

WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP

WHERE AGE >= 29);

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	1625.00
5	Kathrin	34	Bangalore	2125.00
6	Harry	42	China	1125.00
7	Jackson	25	Mizoram	10000.00

4. Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

Syntax

DELETE FROM TABLE_NAME

WHERE VALUE OPERATOR

(SELECT COLUMN_NAME

FROM TABLE_NAME

WHERE condition);

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table.

The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

DELETE FROM EMPLOYEE

WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP

WHERE AGE >= 29);

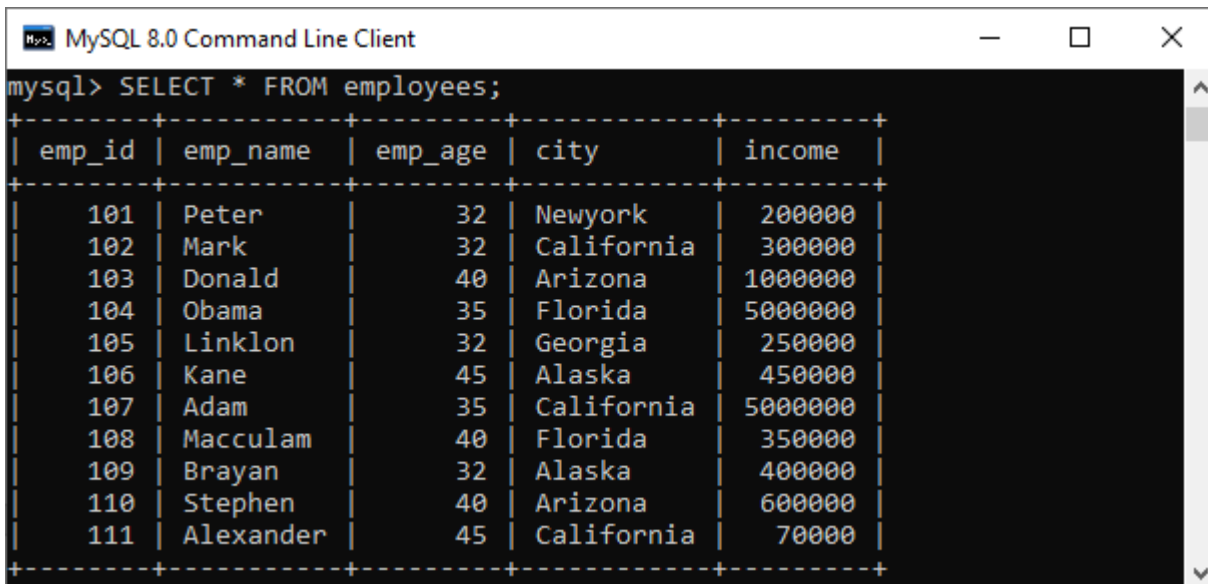
This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
7	Jackson	25	Mizoram	10000.00

MySQL Subquery Example

Let us understand it with the help of an example. Suppose we have a table named "employees" that contains the following data:

Table: employees



```
mysql> SELECT * FROM employees;
```

emp_id	emp_name	emp_age	city	income
101	Peter	32	Newyork	200000
102	Mark	32	California	300000
103	Donald	40	Arizona	1000000
104	Obama	35	Florida	5000000
105	Linklon	32	Georgia	250000
106	Kane	45	Alaska	450000
107	Adam	35	California	5000000
108	Macculam	40	Florida	350000
109	Brayan	32	Alaska	400000
110	Stephen	40	Arizona	600000
111	Alexander	45	California	70000

Following is a simple SQL statement that returns the **employee detail whose id matches in a subquery**:

1. **SELECT** emp_name, city, income **FROM** employees
2. **WHERE** emp_id IN (**SELECT** emp_id **FROM** employees);

This query will return the following output:

```
MySQL 8.0 Command Line Client
mysql> SELECT emp_name, city, income FROM employees
-> WHERE emp_id IN (SELECT emp_id FROM employees);
```

emp_name	city	income
Peter	Newyork	200000
Mark	California	300000
Donald	Arizona	1000000
Obama	Florida	5000000
Linklon	Georgia	250000
Kane	Alaska	450000
Adam	California	5000000
Macculam	Florida	350000
Brayan	Alaska	400000
Stephen	Arizona	600000
Alexander	California	70000

MySQL Subquery with Comparison Operator

A comparison operator is an operator used to compare values and returns the result, either true or false.

The following comparison operators are used in MySQL <, >, =, <>, <=>, etc. We can use the subquery before or after the comparison operators that return a single value. The returned value can be the arithmetic expression or a column function. After that, SQL compares the subquery results with the value on the other side of the comparison operator.

The below example explains it more clearly:

Following is a simple [SQL](#) statement that returns the **employee detail whose income is more than 350000** with the help of subquery:

1. **SELECT * FROM** employees
2. **WHERE** emp_id IN (**SELECT** emp_id **FROM** employees
3. **WHERE** income > 350000);

This query first executes the subquery that returns the **employee id whose income > 350000**. Second, the main query will return the employees all details whose employee id are in the result set returned by the subquery.

After executing the statement, we will get the below output, where we can see the employee detail whose income>350000.

```

MySQL 8.0 Command Line Client
mysql> SELECT * FROM employees
-> WHERE emp_id IN (SELECT emp_id FROM employees
-> WHERE income > 350000);
+-----+-----+-----+-----+-----+
| emp_id | emp_name | emp_age | city      | income |
+-----+-----+-----+-----+-----+
| 103    | Donald  | 40      | Arizona   | 1000000 |
| 104    | Obama   | 35      | Florida   | 5000000 |
| 106    | Kane    | 45      | Alaska    | 450000   |
| 107    | Adam    | 35      | California | 5000000 |
| 109    | Brayan  | 32      | Alaska    | 400000   |
| 110    | Stephen | 40      | Arizona   | 600000   |
+-----+-----+-----+-----+-----+

```

Let us see an example of another comparison operator, such as equality (=) to find employee details with **maximum income** using a subquery.

1. **SELECT** emp_name, city, income **FROM** employees
2. **WHERE** income = (**SELECT MAX**(income) **FROM** employees);

It will give the output where we can see two employees detail who have maximum income.

```

MySQL 8.0 Command Line Client
mysql> SELECT emp_name, city, income FROM employees
-> WHERE income = (SELECT MAX(income) FROM employees);
+-----+-----+-----+
| emp_name | city      | income |
+-----+-----+-----+
| Obama    | Florida   | 5000000 |
| Adam     | California | 5000000 |
+-----+-----+-----+

```

MySQL Subquery with IN or NOT-IN Operator

If the subquery produces more than one value, we need to use the IN or NOT IN operator with the [WHERE clause](#). Suppose we have a table named "Student" and "Student2" that contains the following data:

Table: Student

Stud_ID	Name	Email	City
1	Peter	peter@javatpoint.com	Texas
2	Suzi	suzi@javatpoint.com	California
3	Joseph	joseph@javatpoint.com	Alaska
4	Andrew	andrew@javatpoint.com	Los Angeles
5	Brayan	brayan@javatpoint.com	New York

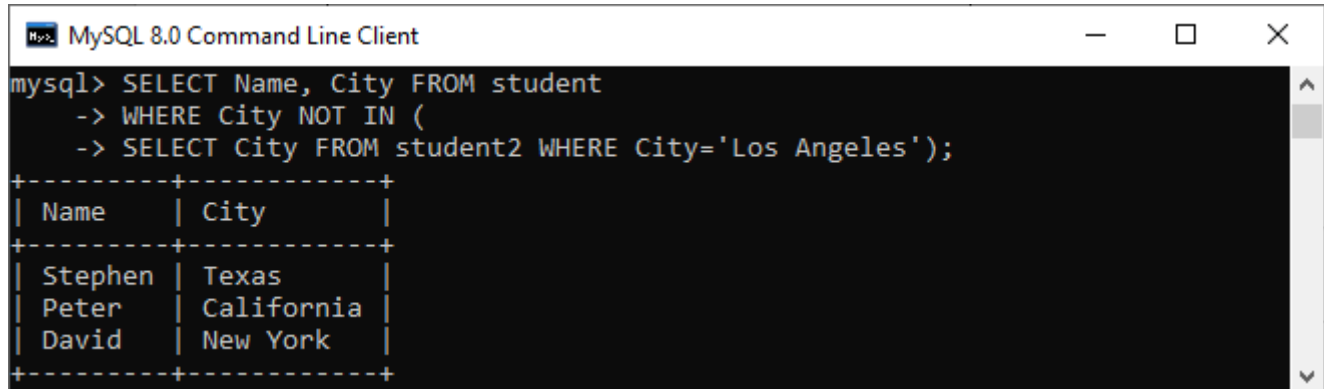
Table: Student2

Stud_ID	Name	Email	City
1	Stephen	stephen@javatpoint.com	Texas
2	Joseph	joseph@javatpoint.com	Los Angeles
3	Peter	peter@javatpoint.com	California
4	David	david@javatpoint.com	New York
5	Maddy	maddy@javatpoint.com	Los Angeles

The following subquery with NOT IN operator returns the **student detail who does not belong to Los Angeles City** from both tables as follows:

1. **SELECT Name, City FROM student**
2. **WHERE City NOT IN (**
3. **SELECT City FROM student2 WHERE City='Los Angeles');**

After execution, we can see that the result contains the student details not belonging to Los Angeles City.



```
mysql> SELECT Name, City FROM student
-> WHERE City NOT IN (
-> SELECT City FROM student2 WHERE City='Los Angeles');
```

Name	City
Stephen	Texas
Peter	California
David	New York

MySQL Subqueries with EXISTS or NOT EXISTS

The [EXISTS operator](#) is a Boolean operator that returns either true or false result.

It is used with a subquery and checks the existence of data in a subquery.

If a subquery returns any record at all, this operator returns true. Otherwise, it will return false.

The NOT EXISTS operator used for negation that gives true value when the subquery does not return any row. Otherwise, it returns false. Both EXISTS and NOT EXISTS used with correlated subqueries.

The following example illustrates it more clearly. Suppose we have a table **customer and order** that contains the data as follows:


```
MySQL 8.0 Command Line Client
mysql> SELECT * FROM customer;
+-----+-----+-----+-----+
| cust_id | name   | occupation | age |
+-----+-----+-----+-----+
| 101     | Peter  | Engineer   | 32  |
| 102     | Joseph | Developer  | 30  |
| 103     | John   | Leader     | 28  |
| 104     | Stephen| Scientist  | 45  |
| 105     | Suzi   | Carpenter  | 26  |
| 106     | Bob    | Actor      | 25  |
| 107     | NULL   | NULL       | NULL|
+-----+-----+-----+-----+
7 rows in set (0.00 sec)

mysql> SELECT * FROM Orders;
+-----+-----+-----+-----+
| order_id | cust_id | prod_name | order_date |
+-----+-----+-----+-----+
| 1         | 101     | Laptop    | 2020-01-10 |
| 2         | 103     | Desktop   | 2020-02-12 |
| 3         | 106     | Iphone    | 2020-02-15 |
| 4         | 104     | Mobile    | 2020-03-05 |
| 5         | 102     | TV        | 2020-03-20 |
+-----+-----+-----+-----+
```

The below SQL statements uses EXISTS operator to find the name, occupation, and age of the customer who has placed at least one order.

1. **SELECT** name, occupation, age **FROM** customer C
2. **WHERE** EXISTS (SELECT * **FROM** Orders O
3. **WHERE** C.cust_id = O.cust_id);

This statement uses NOT EXISTS operator that returns the customer details who have not placed an order.

1. **SELECT** name, occupation, age **FROM** customer C
2. **WHERE** NOT EXISTS (SELECT * **FROM** Orders O
3. **WHERE** C.cust_id = O.cust_id);

We can see the below output to understand the above queries result.

```
mysql> SELECT name, occupation, age FROM customer C
-> WHERE EXISTS (SELECT * FROM Orders O
-> WHERE C.cust_id = O.cust_id);
```

name	occupation	age
Peter	Engineer	32
Joseph	Developer	30
John	Leader	28
Stephen	Scientist	45
Bob	Actor	25

5 rows in set (0.10 sec)

```
mysql> SELECT name, occupation, age FROM customer C
-> WHERE NOT EXISTS (SELECT * FROM Orders O
-> WHERE C.cust_id = O.cust_id);
```

name	occupation	age
Suzi	Carpenter	26
NULL	NULL	NULL

Aim:

To Query the database tables and explore natural, equi and outer joins.

Different Types of SQL JOINS

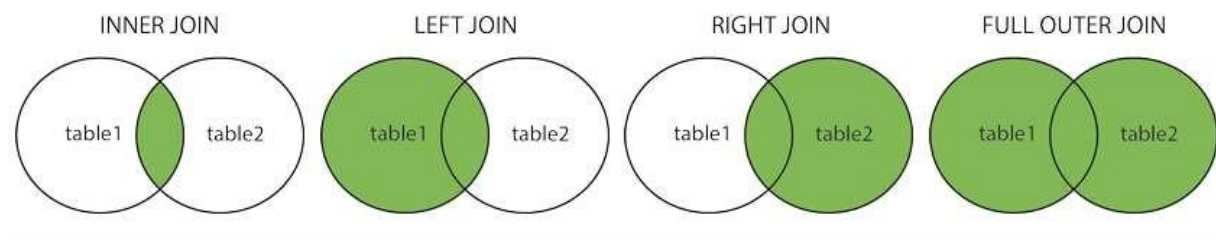
Here are the different types of the JOINS in SQL:

INNER JOIN: Returns records that have matching values in both tables

LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table

RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table

FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table



Sample Table

EMPLOYEE

EMP_ID	EMP_NAME	CITY	SALARY	AGE
1	Angelina	Chicago	200000	30
2	Robert	Austin	300000	26
3	Christian	Denver	100000	42
4	Kristen	Washington	500000	29
5	Russell	Los angels	200000	36
6	Marry	Canada	600000	48

PROJECT

PROJECT_NO	EMP_ID	DEPARTMENT
101	1	Testing
102	2	Development
103	3	Designing
104	4	Development

1. INNER JOIN

Syntax

SELECT table1.column1, table1.column2, table2.column1,

FROM table1

INNER JOIN table2

ON table1.matching_column = table2.matching_column;

Query

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT  
  
FROM EMPLOYEE  
  
INNER JOIN PROJECT  
  
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development

2. LEFT JOIN

Syntax

```
SELECT table1.column1, table1.column2, table2.column1,  
  
FROM table1  
  
LEFT JOIN table2  
  
ON table1.matching_column = table2.matching_column;
```

Query

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT  
  
FROM EMPLOYEE  
  
LEFT JOIN PROJECT  
  
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development
Russell	NULL
Marry	NULL

3. RIGHT JOIN

Syntax

```

SELECT table1.column1, table1.column2, table2.column1,
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;

```

Query

```

SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
RIGHT JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;

```

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development

4 FULL JOIN

Syntax

```
SELECT table1.column1, table1.column2, table2.column1,  
FROM table1  
FULL JOIN table2  
ON table1.matching_column = table2.matching_column;
```

Query

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT  
FROM EMPLOYEE  
FULL JOIN PROJECT  
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

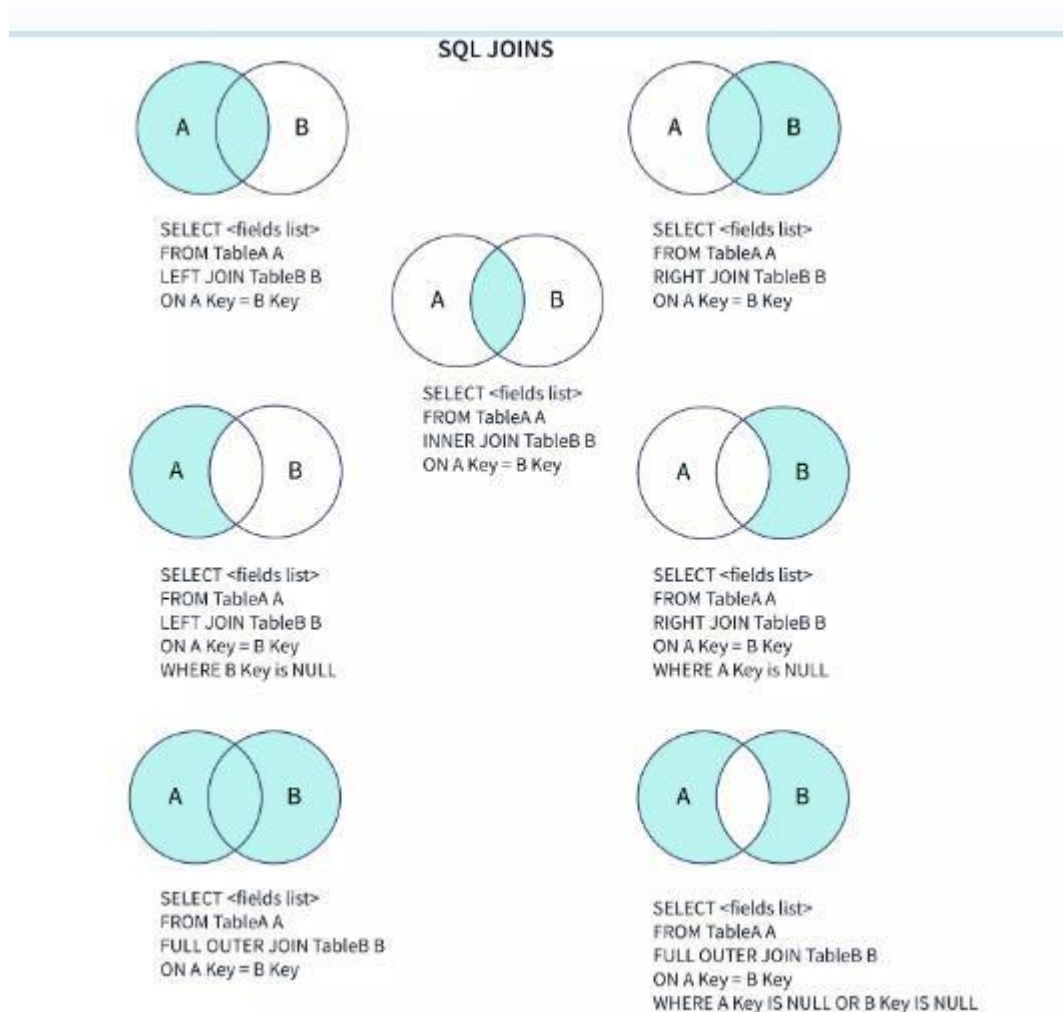
Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development
Russell	NULL
Marry	NULL

What is Natural Join in SQL?

Suppose we have two tables – one which stores the personal information of employees along with their department ID and the other stores the departments' details. How do we bring the data from two different tables into a single table? The answer is Joins.

Joins in SQL are used to combine data from more than one table based on a join condition and view them together as a single table. We have various types of joins in SQL and this article will focus on Natural Join.



Natural Join in SQL combines records from two or more tables based on the common column between them. The common column must have the same name and data type in both the tables. SQL joins the tables based on this common column and hence, we do not need to explicitly specify the join condition.

- There is no need to use the ON clause for join condition in natural join.
- There will always be unique columns in the output of a natural join.

Syntax

```
SELECT * FROM  
tableA NATURAL JOIN tableB
```


Example of Natural Join in SQL

Let us create two tables' employee and department and insert some data into it. We will then implement natural join on these two tables.

Create Employee Table

```
CREATE TABLE employee (  
    EmployeeID varchar(10),  
    FirstName varchar(50),  
    LastName varchar(50),  
    DeptID varchar(10)  
);
```

Insert Records into Employee Table

```
INSERT INTO employee  
VALUES ("E62549", "John", "Doe", "D1001"),  
      ("E82743", "Priya", "Sharma", "D3002"),  
      ("E58461", "Raj", "Kumar", "D1002"),  
      ("E95462", "Ravi", "", "D1001"),  
      ("E25947", "Shreya", "P", "D3000"),  
      ("E42650", "Jane", "Scott", "D3001")
```

We can view the employee table using SELECT query.

```
SELECT * FROM employee
```

Output

EmployeeID	FirstName	LastName	DeptID
E62549	John	Doe	D1001
E82743	Priya	Sharma	D3002
E58461	Raj	Kumar	D1002
E95462	Ravi		D1001
E25947	Shreya	P	D3000
E42650	Jane	Scott	D3001

Create department table

```
CREATE TABLE department (  
    DeptID varchar(10),  
    DeptName varchar(40),  
    Location varchar(40)  
);
```

Insert records into department table

```
INSERT INTO department
VALUES ("D1001", "Technology", "Bangalore"),
      ("D1002", "Technology", "Hyderabad"),
      ("D3001", "Sales", "Gurugram"),
      ("D3002", "Operations", "Hyderabad"),
      ("D4002", "Finance", "Mumbai")
```

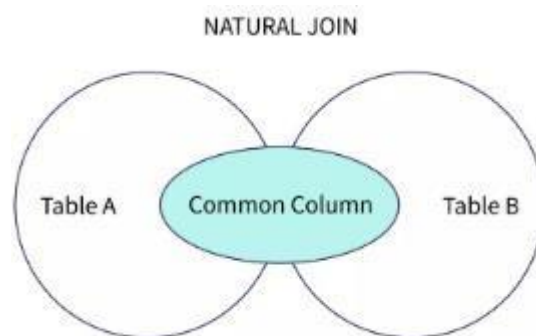
We can view the department table using SELECT query.

```
SELECT * FROM department
```

Output

DeptID	DeptName	Location
D1001	Technology	Bangalore
D1002	Technology	Hyderabad
D3001	Sales	Gurugram
D3002	Operations	Hyderabad
D4002	Finance	Mumbai

Natural Join on Employee and Department Tables



Code

```
SELECT * FROM employee NATURAL JOIN department
```

Output

EmployeeID	FirstName	LastName	DeptID	DeptName	Location
E62549	John	Doe	D1001	Technology	Bangalore
E82743	Priya	Sharma	D3002	Operations	Hyderabad
E58461	Raj	Kumar	D1002	Technology	Hyderabad
E95462	Ravi		D1001	Technology	Bangalore
E42650	Jane	Scott	D3001	Sales	Gurugram

Explanation

We look for common column(s) in the *employee* and the *department* tables. There is one column in both the tables which has the same name DeptID and the same data type varchar. Hence we use DeptID for joining our tables. SQL fetches the department ID of each employee from the *employee* table, maps the department ID to the corresponding record in the *department* table and displays the output as a new table with records merged from both the input tables.

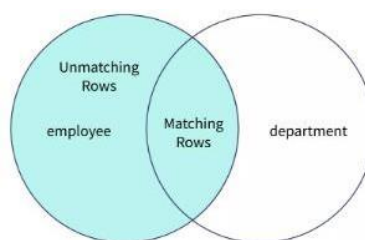
- The output table consists of all columns in the input tables but the common column occurs only once.
- Natural join only displays records for those *DeptID* (common column) that are present in all the tables being joined.
- Natural join is an intersection of tables based on a common column. It is also known as natural inner join.

Types of Natural Join in SQL

Natural inner join only displays records with a common department ID. What if I want to include the departments which do not have an employee yet? Let us look at some examples which combine natural join with left, right, and full outer joins in SQL.

Natural Left Join in SQL

Natural left join displays every single record from the left table, irrespective of whether the common column value (*here, DeptID*) occurs in the right table or not.



Syntax

```
SELECT * FROM  
tableA NATURAL LEFT JOIN tableB
```

Code

```
SELECT * FROM employee NATURAL LEFT JOIN department
```

Output

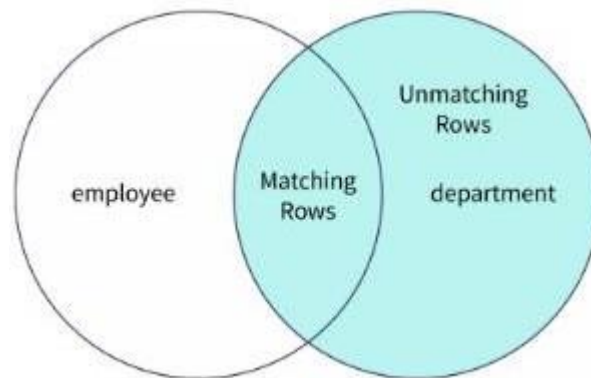
EmployeeID	FirstName	LastName	DeptID	DeptName	Location
E62549	John	Doe	D1001	Technology	Bangalore
E82743	Priya	Sharma	D3002	Operations	Hyderabad
E58461	Raj	Kumar	D1002	Technology	Hyderabad
E95462	Ravi		D1001	Technology	Bangalore
E25947	Shreya	P	D3000	null	null
E42650	Jane	Scott	D3001	Sales	Gurugram

Explanation

The department ID D3000 is not present in *department* table. Hence the details of *Shreya* whose department ID is *D3000* were not available in the natural join output. In the above example though, *Shreya's* records are available since natural left join displays all the records from the left (*employee*) table.

Natural Right Join in SQL

Natural right join is similar to natural left join but here, every record from the right table will be present in the output. Suppose there is a new Finance department but it does not have any employees yet. We can still see its details in the joined table using natural right join!



Syntax

```
SELECT * FROM  
tableA NATURAL RIGHT JOIN tableB
```

Code

```
SELECT * FROM employee NATURAL RIGHT JOIN department
```

Output

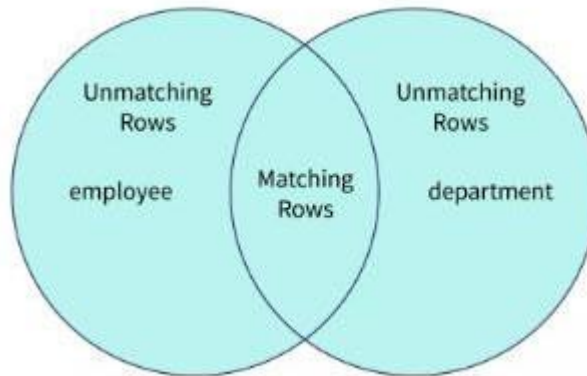
EmployeeID	FirstName	LastName	DeptID	DeptName	Location
E62549	John	Doe	D1001	Technology	Bangalore
E82743	Priya	Sharma	D3002	Operations	Hyderabad
E58461	Raj	Kumar	D1002	Technology	Hyderabad
E95462	Ravi		D1001	Technology	Bangalore
E42650	Jane	Scott	D3001	Sales	Gurugram
null	null	null	D4002	Finance	Mumbai

Explanation

We have an extra row for the Finance department with ID D4002, when compared to the natural join example. Since no employee has joined this department yet, the employee details are all null.

Natural Full Join in SQL

Natural full join is like a union of both input tables. If a given *DeptID* is not available in the other table, the missing data will be filled with null values in the output.



Syntax

```
SELECT * FROM
tableA NATURAL FULL JOIN tableB
```

Code:

```
SELECT * FROM employee NATURAL FULL JOIN department
```

Output:

EmployeeID	FirstName	LastName	DeptID	DeptName	Location
E62549	John	Doe	D1001	Technology	Bangalore
E82743	Priya	Sharma	D3002	Operations	Hyderabad
E58461	Raj	Kumar	D1002	Technology	Hyderabad
E95462	Ravi		D1001	Technology	Bangalore
E25947	Shreya	P	D3000	null	null
E42650	Jane	Scott	D3001	Sales	Gurugram
null	null	null	D4002	Finance	Mumbai

Natural Join with WHERE Clause

Syntax

```
SELECT * FROM
tableA NATURAL JOIN tableB
WHERE filterCondition
```

Code

```
SELECT * FROM employee NATURAL JOIN department
WHERE DeptName = "Technology"
```

Output

EmployeeID	FirstName	LastName	DeptID	DeptName	Location
E62549	John	Doe	D1001	Technology	Bangalore
E95462	Ravi		D1001	Technology	Bangalore
E58461	Raj	Kumar	D1002	Technology	Hyderabad

Explanation We first create a natural join of the *employee* and *department* tables. Then we filter out those records from the resultant table where the department name is Technology.

Natural Join using Three Tables

We can join more than two tables using natural join in SQL. We already have two tables *employee* and *department*. Let us create another table *address* which will store the location city and state of each department.

Create address table

```
CREATE TABLE address (
    Location varchar(40),
    State varchar(40)
);
```

Insert records into the address table

```
INSERT INTO address
VALUES ("Bangalore", "Karnataka"),
       ("Hyderabad", "Telangana"),
       ("Gurugram", "Haryana"),
       ("Mumbai", "Maharashtra")
```

We can view the address table using the SELECT query.

```
SELECT * FROM address
```

Output:

Location	State
Bangalore	Karnataka
Hyderabad	Telangana
Gurugram	Haryana
Mumbai	Maharashtra

EmployeeID	FirstName	LastName	DeptID
E62549	John	Doe	D1001
E82743	Priya	Sharma	D3002
E58461	Raj	Kumar	D1002
E95462	Ravi		D1001
E525947	Shreya	P	D3000
E42650	Jane	Scott	D3001

Same column
name "DeptID"

Rows with
common DeptID
are retained

Others rows
are dropped

DeptID	DeptName	Location
D1001	Technology	Bangalore
D1002	Technology	Hyderabad
D3001	Sales	Gurugram
D3002	Operations	Hyderabad
D4002	Fianance	Mumbai

Same column
name "Location"

Same column
name "Location"

Rows with
common Location
are retained

Others rows
are dropped

Location	State
Bangalore	Karnataka
Hyderabad	Telangana
Gurugram	Haryana
Mumbai	Maharashtra

Syntax

```
SELECT * FROM
tableA NATURAL JOIN tableB NATURAL JOIN tableC
```

Code

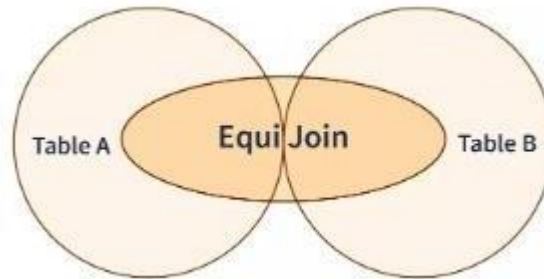
```
SELECT * FROM
employee NATURAL JOIN department NATURAL JOIN address
```

Output

EmployeeID	FirstName	LastName	DeptID	DeptName	Location	State
E62549	John	Doe	D1001	Technology	Bangalore	Karnataka
E82743	Priya	Sharma	D3002	Operations	Hyderabad	Telangana
E58461	Raj	Kumar	D1002	Technology	Hyderabad	Telangana
E95462	Ravi		D1001	Technology	Bangalore	Karnataka
E42650	Jane	Scott	D3001	Sales	Gurugram	Haryana

Equi join in SQL is a type of Inner Join in which two or more tables are joined by equating the common column values of the tables. The table participating in Equi join should have at least one common column on which equality operation can be performed.

What is Equi Join in SQL?



Syntax

The Equi join in SQL can be written with the WHERE clause and also the ON clause for equating the columns of the participating tables.

Example

- with WHERE clause

```
> SELECT *  
FROM Table1, Table2  
WHERE TableName1.ColumnName = TableName2.ColumnName;
```

- with ON clause

```
> SELECT *  
FROM Table1  
JOIN Table2  
ON Table1.ColumnName = Table2.ColumnName;
```

Why Use Equi Join in SQL?

Equi join in SQL is used mostly to join two or more tables on the basis of equality of column values of the tables.

While working on web applications the data are stored in different tables in the database. So, if one needs to combine the data of two tables to get better data insights then the need for joining the table arises. Equi join is one of the [SQL Joins](#) which can be used to join the tables for better data insights.

For example - Let's understand Equi join with the help of two tables.

1. Products table

id	product_name
----	--------------

1	Iwatch
---	--------

2	IPhone
---	--------

3	IPod
---	------

4	macbook
---	---------

2. Discount table

id	dicount_percentage
----	--------------------

2	20%
---	-----

4	17%
---	-----

Suppose we have a need to know the discounts available on the products then can we only know that by the Product table or the discount table? No, we have to combine both the table to know what are the discounts available on the products.

With the help of inner join, we can join both the products table and the discount table to know on which products the discount exists. The column on which we can perform Equi join is the id column in both tables.

The resultant table after performing Equi join on products and the discount table will be –

```
SELECT *  
FROM Products, Discount  
WHERE Products.id = Discount.id;
```

d	product_name	id	discount_percentage
2	Iphone	2	20%
4	macbook	4	17%

How is Equi Join Different from Natural Join?

The difference between Equi join and natural join lies in column names which are equated for performing equi join or natural join.

In Natural join, the tables should have the same column names to perform equality operations on them. In Equi join, the common column name can be the same or different. Also in the resultant table of Equi join the common column of both the tables are present. But in the natural join, the common column is present only once in the resultant table.

If we would have performed natural join on the above products and discount table then the resultant table would be –

```
SELECT *  
FROM Products  
NATURAL JOIN Discount;
```

id	product_name	discount_percentage
2	Iphone	20%
4	macbook	17%

Like programming languages SQL Server also provides User Defined Functions (UDFs). From SQL Server 2000 the UDF feature was added. UDF is a programming construct that accepts parameters, does actions and returns the result of that action. The result either is a scalar value or result set. UDFs can be used in scripts, Stored Procedures, triggers and other UDFs within a database.

Benefits of UDF

1. UDFs support modular programming. Once you create a UDF and store it in a database then you can call it any number of times. You can modify the UDF independent of the source code.
2. UDFs reduce the compilation cost of T-SQL code by caching plans and reusing them for repeated execution.
3. They can reduce network traffic. If you want to filter data based on some complex constraints then that can be expressed as a UDF. Then you can use this UDF in a WHERE clause to filter data.

Types of UDF

1. Scalar Functions
2. Table Valued Functions

Consider the following Student and Subject tables for examples.

Rno	Marks	Name
1	60	Ram
2	50	Ganesh
3	55	Sham
4	70	Raj

Rno	Subject1	Subject2	Subject3
1	10	30	20
2	10	10	30
3	20	25	10
4	15	20	35

1. Scalar Functions

A Scalar UDF accepts zero or more parameters and return a single value. The return type of a scalar function is any data type except text, ntext, image, cursor and timestamp. Scalar functions can be use in a WHERE clause of the SQL Query.

Crating Scalar Function

To create a scalar function the following syntax is used.

1. **CREATE FUNCTION** **function-name** (Parameters)
2. **RETURNS** **return-type**
3. **AS**
4. **BEGIN**

```
5. Statement 1
6. Statement 2
7. .
8. .
9. Statement n
10. RETURN return-value
11. END
```

Example

Create a function as follows.

```
1. CREATE FUNCTION GetStudent(@Rno INT)
2. RETURNS VARCHAR(50)
3. AS
4. BEGIN
5. RETURN (SELECT Name FROM Student WHERE Rno=@Rno)
6. END
```

To execute this function use the following command.

```
1. PRINT dbo.GetStudent(1)
```

Output: Ram

1. Table Valued Functions

A Table Valued UDF accepts zero or more parameters and return a table variable. This type of function is special because it returns a table that you can query the results of a join with other tables. A Table Valued function is further categorized into an “Inline Table Valued Function” and a “Multi-Statement Table Valued Function”.

A. Inline Table Valued Function

An Inline Table Valued Function contains a single statement that must be a SELECT statement. The result of the query becomes the return value of the function. There is no need for a BEGIN-END block in an Inline function.

Crating Inline Table Valued Function

To create a scalar function the following syntax is used.

```
1. CREATE FUNCTION function-name (Parameters)
2. RETURNS return-type
3. AS
4. RETURN
```

Query

Example

1. **CREATE FUNCTION** GetAllStudents(@Mark **INT**)
2. **RETURNS TABLE**
3. **AS**
4. **RETURN**
5. **SELECT *FROM** Student **WHERE** Marks>=@Mark

To execute this function use the following command.

1. **SELECT *FROM** GetAllStudents(60)

Output

Rno	Marks	Name
1	60	Ram
4	70	Raj

B. Multi-Statement Table Valued Function

A Multi-Statement contains multiple SQL statements enclosed in BEGIN-END blocks. In the function body you can read data from databases and do some operations. In a Multi-Statement Table valued function the return value is declared as a table variable and includes the full structure of the table to be returned. The RETURN statement is without a value and the declared table variable is returned.

Crating Multi-Statement Table Valued Function

To create a scalar function the following syntax is used.

1. **CREATE FUNCTION** function-name (Parameters)
2. **RETURNS** @TableName **TABLE**
3. (Column_1 datatype,
4. .
5. .
6. Column_n datatype
7.)
8. **AS**
9. **BEGIN**
10. Statement 1
11. Statement 2
12. .
13. .
14. Statement n
15. **RETURN**
16. **END**

Example

Create a function as follows.

```
1. CREATE FUNCTION GetAvg(@Name varchar(50))
2. RETURNS @Marks TABLE
3. (Name VARCHAR(50),
4. Subject1 INT,
5. Subject2 INT,
6. Subject3 INT,
7. Average DECIMAL(4,2)
8. )
9. AS
10. BEGIN
11.     DECLARE @Avg DECIMAL(4,2)
12.     DECLARE @Rno INT
13.     INSERT INTO @Marks (Name)VALUES(@Name)
14.     SELECT @Rno=Rno FROM Student WHERE Name=@Name
15. SELECT @Avg=(Subject1+Subject2+Subject3)/3 FROM Subjects WHERE Rno=@Rno
16.
17.     UPDATE @Marks SET
18. Subject1=(SELECT Subject1 FROM Subjects WHERE Rno=@Rno),
19. Subject2=(SELECT Subject2 FROM Subjects WHERE Rno=@Rno),
20. Subject3=(SELECT Subject3 FROM Subjects WHERE Rno=@Rno),
21. Average=@Avg
22. WHERE Name=@Name
23. RETURN
24. END
```

To execute this function use the following command.

```
1. SELECT * FROM GetAvg('Ram')
```

Output

Name	Subject1	Subject2	Subject3	Average
Ram	10	30	20	20.00

Ex No: 7

Execute complex transactions and realize DCL and TCL commands.

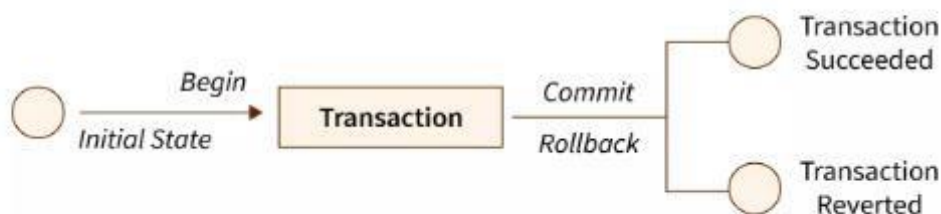
TCL stands for **Transaction Control Language** in SQL. Transaction Control Language (TCL) is a set of special commands that **deal with the transactions** within the database. Basically, they are used to manage transactions within the database. TCL commands are also used for **maintaining the consistency of the database**.

Introduction

A **transaction** is a unit of work that is **performed against a database in SQL**. In other words, a transaction is a single, indivisible database action. If the transaction contains multiple statements, it is called a multi-statement transaction (MST). By default, all transactions are multi-statement transactions.

For example, suppose we are creating a new record or updating or deleting any record from a table (in general, performing any changes on the table). In that case, we are performing a transaction on the table.

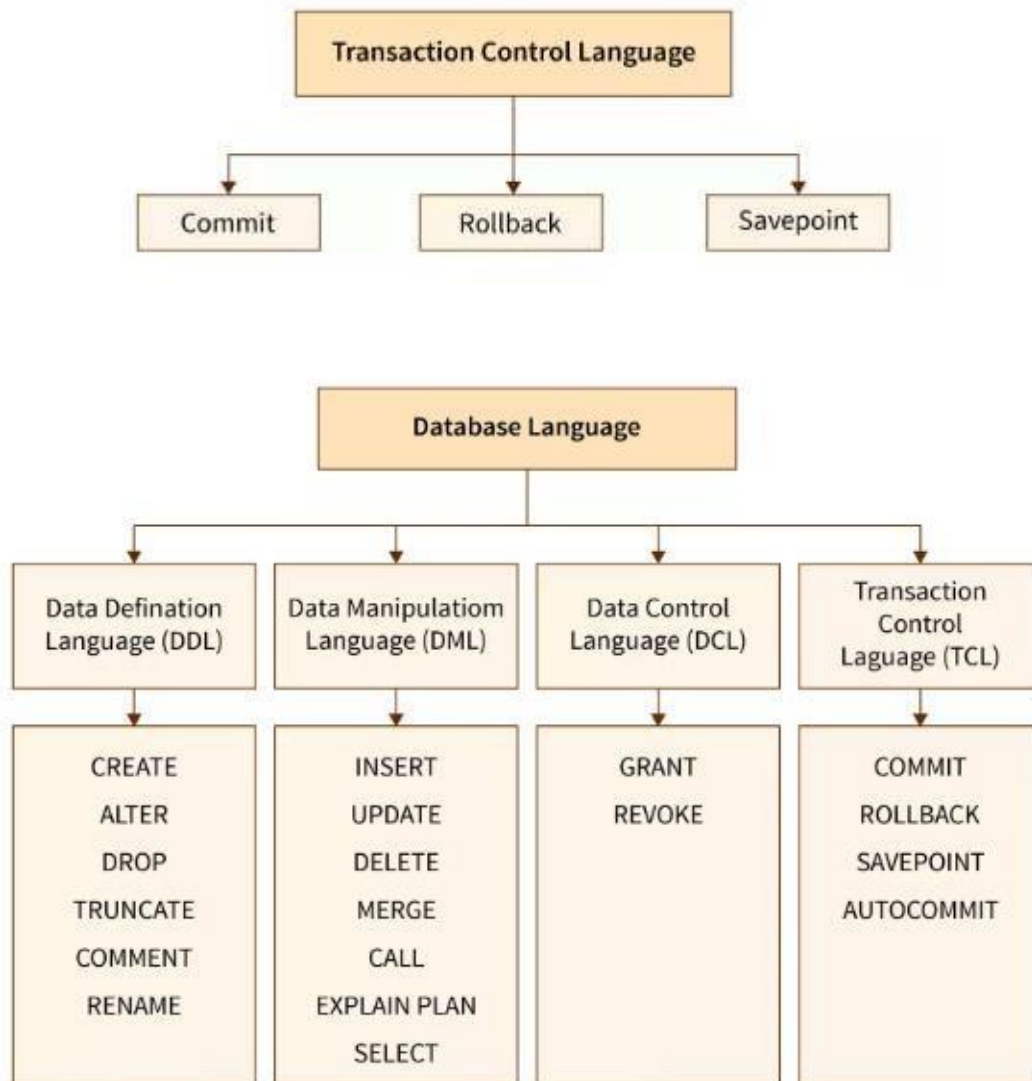
In SQL, each transaction begins with a particular set of task and ends only when all the tasks in the set is completed successfully. However, if any (or a single) task fails, the transaction is said to fail.



What is meant by Maintaining the Consistency of a Database?

As a part of database systems, consistency refers to ensuring that any particular database transaction makes changes only in ways that are permitted or allowed. There are a set of rules that must be met to ensure that the database is consistent. The rules might include different constraints, cascades, triggers, or anything else in general.

Types of TCL Commands with Examples



Transactional control commands (TCL commands) are used mostly in the DML Commands in SQL, such as INSERT, UPDATE, and DELETE.

In general, the **TCL commands** consist of the below commands:

1. Commit
2. RollBack
3. SavePoint

Types of SQL Commands

In this article on SQL basics, we study about following types of commands:

- **DDL(Data Definition Language):** To make/perform changes to the physical structure of any table residing inside a database, DDL is used. These commands when executed are auto-commit in nature and all the changes in the table are reflected and saved immediately.
- **DML(Data Manipulation Language):** Once the tables are created and the database is generated using DDL commands, manipulation inside those tables and databases is done using DML commands. The advantage of using DML commands is, that if in case any wrong changes or values are made, they can be changed and rolled back easily.
- **DQL(Data Query Language):** Data query language consists of only one command upon which data selection in SQL relies. The **SELECT** command in combination with other SQL clauses is used to retrieve and fetch data from databases/tables based on certain conditions applied by the user.
- **DCL(Data Control Language):** DCL commands as the name suggests manage the matters and issues related to the data controller in any database. DCL includes commands such as **GRANT** and **REVOKE** which mainly deal with the rights, permissions, and other controls of the database system.
- **TCL(Transaction Control Language):** Transaction Control Language as the name suggests manages the issues and matters related to the transactions in any database. They are used to roll back or commit the changes in the database.

Data Control Language

DCL is used to access the stored data. It is used to revoke and grant the user the required access to a database. In the database, this language does not have the feature of rollback. It is a part of the **structured query language (SQL)**.

It helps in controlling access to information stored in a database. It complements the data manipulation language and the data definition language. It is the simplest of three commands.

It provides the administrators, to remove and set database permissions to desired users as needed. These commands are employed to grant, remove and deny permissions to users for retrieving and manipulating a database. There are two relevant commands under this category: grant and revoke.

GRANT

GRANT is a command used to provide access or privileges on the database objects to the users.

SYNTAX

```
GRANT PRIVILEGES  
ON OBJECT  
TO USER;
```

Privilege	Description
SELECT	<i>select statement on tables</i>
INSERT	<i>insert statement on the table</i>
DELETE	<i>delete statement on the table</i>
INDEX	<i>Create an index on an existing table</i>
CREATE	<i>Create table statements</i>
ALTER	<i>Ability to perform ALTER TABLE to change the table definition</i>
DROP	<i>Drop table statements</i>
ALL	<i>Grant all permissions except GRANT OPTION</i>
UPDATE	<i>Update statements on the table</i>
GRANT	<i>Allow to grant and manage privileges</i>

1. SELECT:

To grant Select Privilege to a table named "tableName", the user name is "userName", and the following GRANT statement should be executed.

```
GRANT SELECT
ON tableName
TO 'userName'@'localhost';
```

2. Granting multiple privileges to a user:

To grant multiple Privileges to a user named "username" in table "tableName", the following GRANT statement should be executed:

```
GRANT SELECT, INSERT, DELETE, UPDATE
ON tableName
TO 'userName'@'localhost';
```

3. Granting all the privileges to a user:

To Grant all the privileges to a user named “userName” in a table “tableName”, the following Grant statement should be executed.

```
GRANT ALL
ON tableName
TO 'userName'@'localhost';
```

4. Granting a privilege to all users:

To Grant a specific privilege to all the users in a table “tableName” this Grant statement should be executed.

```
GRANT SELECT
ON tableName
TO '*'@'localhost';
```

REVOKE

Once you have granted privileges, you may need to revoke some or all of these privileges. To do this, you can run a revoke command. You can revoke any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, or ALL.

```
REVOKE privileges
ON object
FROM user;
```

Object is the name of the database object that you are revoking privileges for. In the case of revoking privileges on a table, this would be the table name. Username of the user that will have these privileges revoked.

Suppose we need to revoke delete permission for the 'tableName' table' from a user named 'userName', the following would be the query.

```
REVOKE DELETE
ON tableName
FROM userName;
```

To remove every permission use 'ALL'.

```
REVOKE ALL
ON tableName
FROM userName;
```

Transaction Control Language

TCL includes statements that are used to manage the changes that are made from DML statements. It enhances the transactional nature of SQL. The TCL commands in SQL are:

- **COMMIT:** It's a SQL command used in the transaction tables or database to make the current transaction or database statement permanent. It shows the successful completion of a transaction. If we have successfully executed the transaction statement or a simple database query, we want to make the changes permanent. We need to perform the commit command to save the changes, and these changes become permanent for all users. Furthermore, once the commit command is executed in the database, we cannot regain its previous states in which it was earlier before the execution of the first statement.

SYNTAX

```
commit;
```

- **ROLLBACK:** Undoes any changes made to the database. ROLLBACK is the SQL command that is used for reverting changes performed by a transaction. When a ROLLBACK command is issued it reverts all the changes since the last COMMIT or ROLLBACK.

SYNTAX

```
ROLLBACK;
```

- **SAVEPOINT:** This command creates a point in your transaction to which you can roll back. It is a command in SQL that is used with the rollback command. It is a command in Transaction Control Language that is used to mark the transaction in a table.

SYNTAX

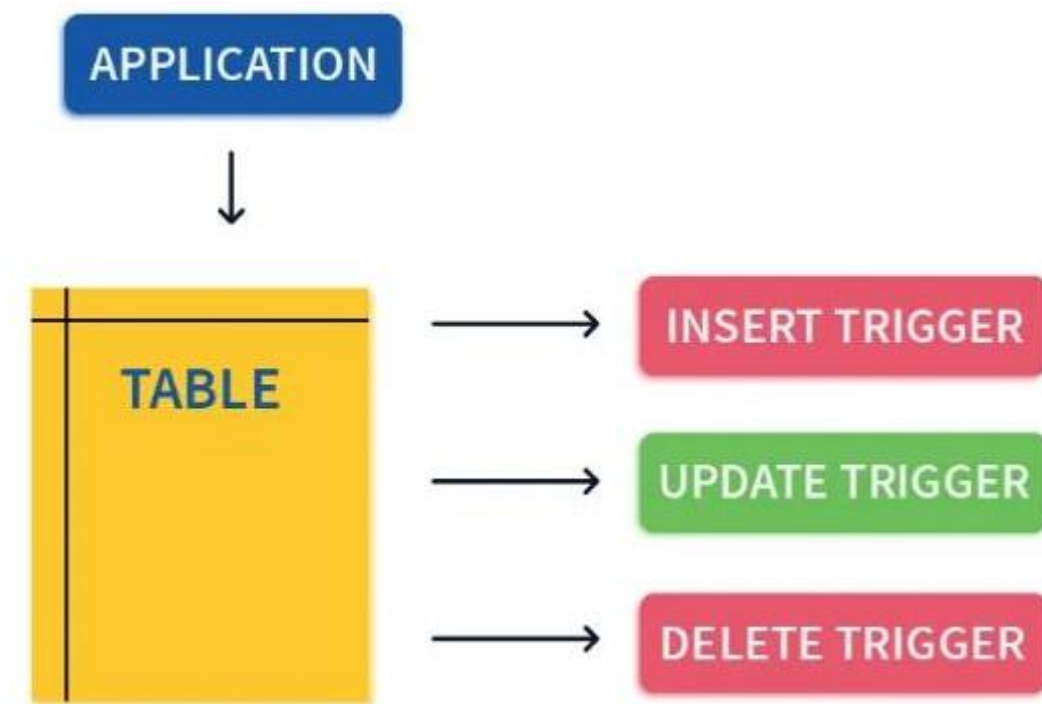
```
SAVEPOINT some_name;
```

A SQL trigger is a database object which fires when an event occurs in a database. For example, a trigger can be set on a record insert in a database table.

Scope

- In this article, we will discuss how to create triggers with SQL syntax and then discuss different parts of the syntax. After that, we will go through how to display and drop a trigger using SQL.

What are the Triggers in SQL?



Trigger Points of a SQL Trigger

- When any DDL operation is done. E.g., CREATE, ALTER, DROP
- For a DML operation. e.g., INSERT, UPDATE, DELETE.
- For a database operation like LOGON, LOGOFF, STARTUP, SHUTDOWN or SERVERERROR

Creating Triggers in SQL

```
CREATE [OR REPLACE ] TRIGGER trigger_name
```

```
{ BEFORE | AFTER | INSTEAD OF }
```

{INSERT [OR] | UPDATE [OR] | DELETE}

ON table_name

[FOR EACH ROW]

WHEN (condition)

[trigger_body]

Let's discuss the different parts of the syntax:

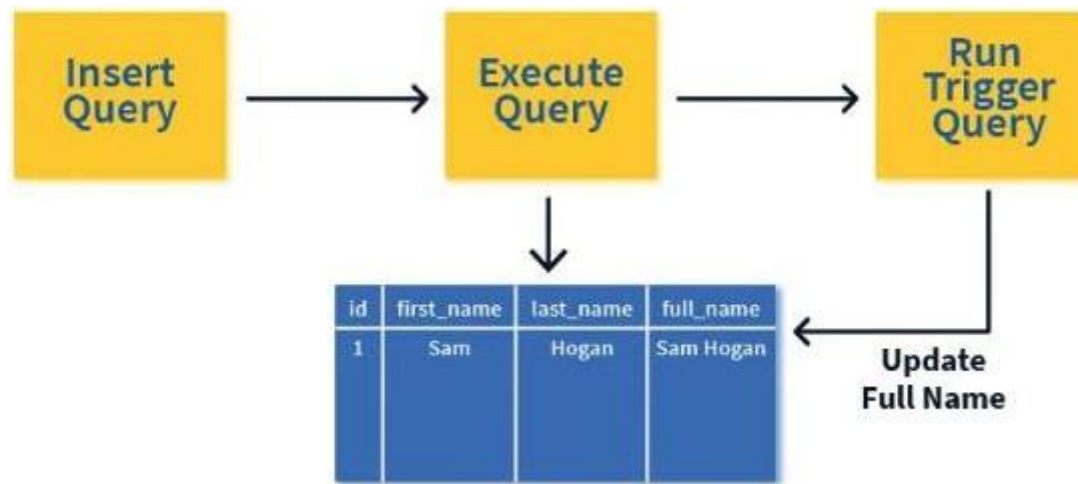
- **CREATE [OR REPLACE] TRIGGER trigger_name:** In the first line, we give the name of the trigger we are creating/updating. Here [trigger_name] is to be replaced by the name you want to give to your trigger.
- **{BEFORE / AFTER / INSTEAD OF }:** Here, we define when the trigger would run, i.e., before or after the DML operation. For example, if we want a trigger to be executed after insertion to a table, we will write after here.
- **{INSERT [OR] / UPDATE [OR] / DELETE}:** This is the operation or event we have to define, which will trigger a procedure to be executed. If we want a procedure to run after a deletion happens on the table, then we will consider writing delete here.
- **on [table_name]:** Here, we have to specify the name of the table on which we are attaching the trigger. [SQL](#) will listen to changes on this table.
- **[for each row]:** This line specifies that the procedure will be executed for each one of the rows present. Here we can set a condition for which rows to be impacted. This part is optional, though; in case we don't use this, the trigger shall convert to a "statement-level" trigger rather than being a "row-level" one, i.e., instead of firing the trigger procedure for each row, it will only execute once for each applicable statement.
- **WHEN (condition):** Here, we mention some condition basis on which the trigger will run. In the absence of a when condition, you can expect it to run for all the eligible rows. This is very important as this will control which rows the trigger must run.
- **[trigger_body]:** This is the main logic of what to perform once the trigger is fired. In the previous statements, all we defined is when this trigger will be fired, but here we have to define what to do after the trigger is fired. This is the main execution code.

Let's take an example. Let's assume a student table with column id, first_name, last_name, and full_name.

Query 1:

```
CREATE TABLE student(Id integer PRIMARY KEY, first_name varchar(50), last_name varchar(50), full_name varchar(50));
```

Here we will create a trigger to fill in the full name by concatenating the first and last names. So while inserting the values, we will only feed the first name and last name, and we will expect the trigger to automatically update each row with an additional column attribute bearing the full name.



Query 2:

```
create trigger student_name
after INSERT
on
student
for each row
BEGIN
  UPDATE student set full_name = first_name || ' ' || last_name;
END;
```

Here we can understand from the trigger query we have set a trigger after an insert is made to the table student. Once the insert is done, this procedure will be fired, which will run an update command to update the students' full names.

Let's insert the students.

Query 3:

```
INSERT INTO student(id, first_name, last_name) VALUES(1,'Alvaro', 'Morte');
INSERT INTO student(id, first_name, last_name) VALUES(2,'Ursula', 'Corbero');
INSERT INTO student(id, first_name, last_name) VALUES(3,'Itziar', 'Ituno');
INSERT INTO student(id, first_name, last_name) VALUES(4,'Pedro', 'Alonso');
INSERT INTO student(id, first_name, last_name) VALUES(5,'Alba', 'Flores');
```

Here we have inserted five students' data, and since we have a trigger created in our system to update the full_name, we are expecting the full name to be non-empty if we run a select query on this table.

Query 4:

```
/* Display all the records from the table */
SELECT * FROM student;
```


Output:

emp_id	first_name	last_name	full_name
1	Alvaro	Morte	Alvaro Morte
2	Ursula	Corbero	Ursula Corbero
3	Itziar	Ituno	Itziar Ituno
4	Pedro	Alonso	Pedro Alonso
5	Alba	Flores	Alba Flores

Display Triggers in SQL

If someone creates a trigger but forgets the trigger's name, then you can find the trigger by running a simple command.

Query 6:

```
SHOW TRIGGERS LIKE 'stu%\G;
```

This command will show you the list of all available triggers matching with the string **'stu'**. Note that the **\G** tag ends the statement just like a semicolon (;). However, since the output table is wide, the **\G** rotates the table visually to vertical mode.

```
mysql> SHOW TRIGGERS LIKE 'stu%\G
***** 1. row *****
Trigger: student_name
Event: INSERT
Table: student
Statement: SET full_name = first_name || " || last_name
Timing: AFTER
Created: NULL
sql_mode: NO_ENGINE_SUBSTITUTION
Definer: me@localhost
character_set_client: utf8
collation_connection: utf8_general_ci
Database Collation: latini_swedish_ci
```

Drop Triggers in SQL

The deletion of a trigger is quite straightforward. You need to run a simple query to delete a trigger from the database.

So in our case, as per the example above, the query will be

Query 5:

```
DROP TRIGGER student_name;
```

Overview

A View in SQL is simply a virtual table created based on a result set of another SQL statement. Views were introduced to reduce the complexity of multiple tables and deliver data in a simple manner. Views help us maintain [data integrity](#) and provide security to the data, thus acting as a security mechanism.

Creating a View in SQL

The Views in SQL are created with the **CREATE VIEW** syntax. Following is the basic syntax to create a VIEW in SQL:

```
CREATE VIEW view_name AS  
SELECT column1, column2...column N  
FROM table1, table2...table N  
WHERE condition;
```

To see the data in the view, we can query the view using the following SELECT statement:

```
SELECT * FROM [view_name];
```

- “CREATE VIEW” is used at the start of the code to initiate the view.
- “SELECT” is used to decide which columns to pick from the tables.
- With the help of the “FROM” term, we can select the tables from which columns(data) have to be picked.
- “Table1..table N” denotes the names of the tables. (Here, for example, we have “Scaler Courses” & “Author Details” as tables.)
- “WHERE” is used to define the condition pertaining to selecting rows.

We will learn more about the different types of operations in the views with some simple examples. Let’s take the below scenario to understand views thoroughly.

At Scaler, some people work in different departments on a particular project to publish information about the company's courses. To adhere to the particular data which needs to be shown at some time to the users, the technical team takes the help of the views in SQL to query the results they want. We will try understanding the whole article through the examples of these tables, namely **SCALER COURSES & AUTHOR DETAILS**, which are already in the database of the company.

Table 1 (ScalerCourses)

SNo	Name	Duration	CourseLanguage	Cost(Rs)
1	Python Foundation	3-4 months	English	1500
2	Django	5 months	English	1000
3	C++	4-5 months	Hindi	500
4	Interview Preparation	6 months	English	1800
5	Node Js	6 months	Hindi	2500

Table 2 (AuthorDetails)

SNo	Name	Rating
1	Anshuman	5
2	Ravi	4
3	Raman	4.5
4	Yash	5
5	Jatin	4.5

Now let's start by creating a view. Here the tech team will create a view named "CourseView" from the table "ScalerCourses" (Table 1) for querying some specific course details for the students, which are below the cost of Rs 2000, to display on the website.

```
CREATE VIEW CourseView AS
SELECT Name, Duration
FROM ScalerCourses
WHERE Cost < 2000;
```

We can see the following data by querying the view as follows:

```
SELECT * FROM CourseView;
```

The output of the above query:

Name	Duration
Python Foundation	3-4 months
Django	5 months
C++	4-5months
Interview Preparation	6 months

Updating a View

A view can be easily updated with the CREATE OR REPLACE VIEW statement, but certain conditions must be considered while updating. For example, the Tech Team wants to update the **CourseView** and add the **CourseLanguage** as a new field to this View.

```
CREATE OR REPLACE VIEW CourseView AS
SELECT Name, Duration, CourseLanguage
FROM ScalerCourses
WHERE Cost < 2000;
```

Now, if we want to look at the data in CourseView, we can query as follows.

```
SELECT * FROM CourseView;
```

Output for the above statements is as follows:

Name	Duration	Course Language
Python Foundation	3-4 months	English
Django	5 months	English
C++	4-5months	Hindi
Interview Preparation	6 months	English

Inserting Rows in a SQL View

The Scaler HR team did hire some teachers in the past month. Now looking into the demand for some courses by the students, they thought of adding a new course to the virtual table. For updating the view with a new row, they can update the view by inserting new rows into the view, by using the INSERT INTO statement.

```
INSERT INTO CourseView(Name, Duration)
VALUES("Java", "4 months");
```

The following will be the data stored in our view after executing the above query.

Name	Duration
Python Foundation	3-4 months
Django	5 months
C++	4-5months
Interview Preparation	6 months
Java	4 months

Deleting Rows Into a View

```
DELETE FROM CourseView  
WHERE Name = "Python Foundation";
```

The data after executing the above query looks as follows.

Name	Duration
Django	5 months
C++	4-5months
Interview Preparation	6 months
Java	4 months

Dropping Views(Deleting a View)

After some months, the team at Scaler thought of scrapping the virtual tables they created for displaying on the website. SQL Views allows them to delete a view using the DROP statement.

SYNTAX:

```
DROP VIEW view name;
```

For Example:

```
DROP VIEW CourseView;
```

