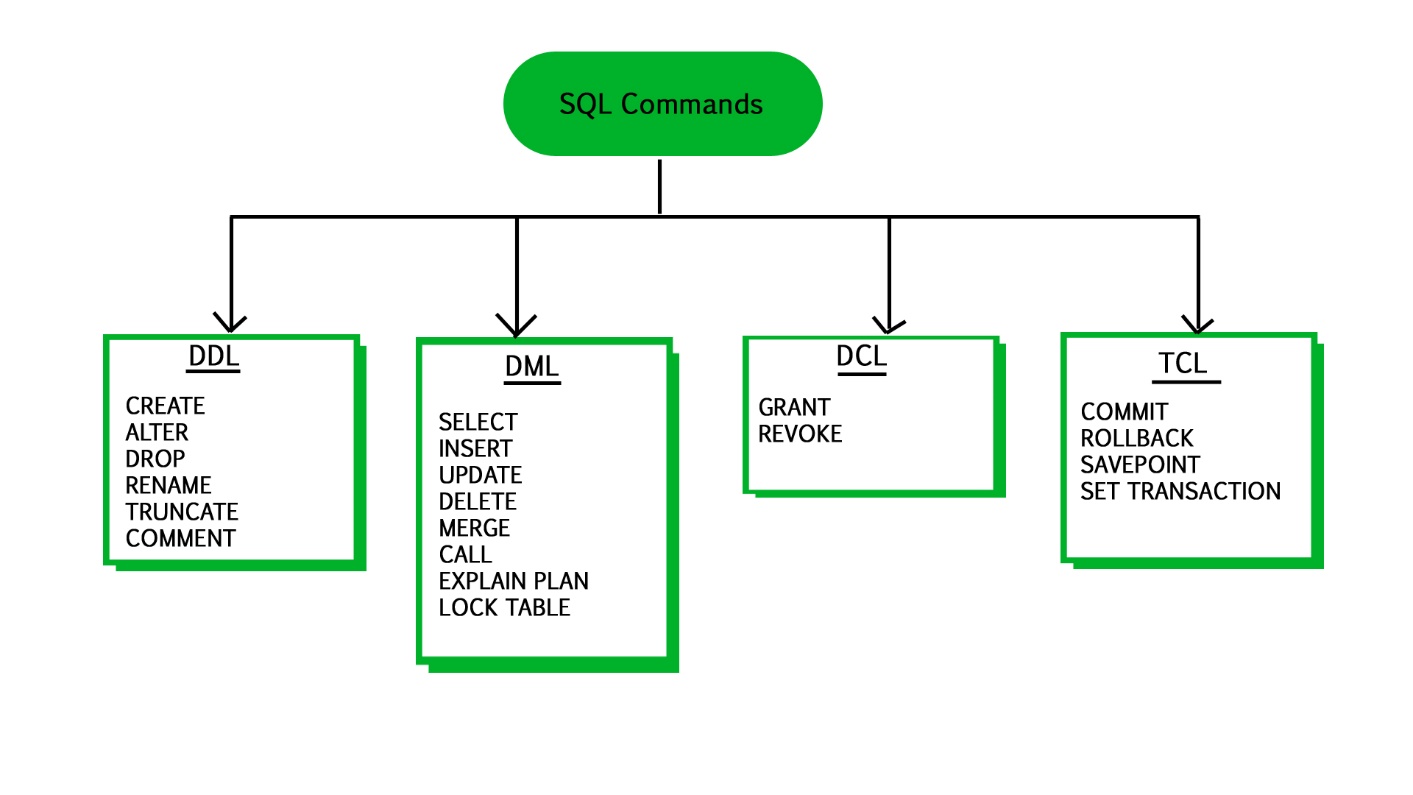
**SQL Structured query language**



The **CREATE DATABASE** statement is used to create a new SQL database.

CREATE DATABASE databasename;

The **DROP DATABASE** statement is used to drop an existing SQL database.

DROP DATABASE databasename;

The **CREATE TABLE** statement is used to create a new table in a database.

SYNTAX

CREATE TABLE table\_name (  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
   ....  
 );

EXAMPLE

CREATE TABLE Persons (  
    PersonID int, gender char(1)  
    LastName varchar(255),  
    DOB date,  
    Salary float(255),  
   );

The **DROP TABLE** statement is used to drop an existing table in a database.

DROP TABLE table\_name;

DROP TABLE employee;

The **TRUNCATE TABLE** statement is used to delete the data inside a table, but not the table itself.

(This means the table structure, attributes, and indexes will be intact)

TRUNCATE TABLE table\_name;

TRUNCATE TABLE employee;

The **UPDATE** statement is used to modify the existing records in a table.(rows)

SYNTAX

UPDATE table\_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;

EXAMPLE

UPDATE Customers  
SET ContactName = 'prathamesh', City= 'kolhapur'  
WHERE CustomerID = 1;

The**DELETE** statement is used to delete existing records in a table. It will delete entire row

SYNTAX

DELETE FROM table\_name

WHERE condition;

EXAMPLE

DELETE FROM Customers

WHERE CustomerName='rohan';

Idelete all rows in a table without deleting the table. Like truncate

DELETE FROM table\_name;

DELETE FROM Customers;

The **ALTER TABLE** statement is used to add, delete, or modify columns in an existing table.

**To add a column in a table, use the following syntax:**

SYNTAX

ALTER TABLE table\_name  
ADD column\_name datatype;

EXAMPLE

ALTER TABLE Customers  
ADD Email varchar(255);

**To delete a column in a table, use the following syntax**

SYNTAX

ALTER TABLE table\_name  
DROP COLUMN column\_name;

EXAMPLE

ALTER TABLE Customers  
DROP COLUMN Email;

**To change the data type of a column in a table**

SYNTAX

ALTER TABLE table\_name  
MODIFY COLUMN column\_name datatype;

EXAMPLE

ALTER TABLE person   
MODIFY COLUMN age float;

**To add constraints on the column**

SYNTAX

ALTER TABLE table\_name  
MODIFY COLUMN column\_name datatype constraint;

EXAMPLE

ALTER TABLE t  
MODIFY COLUMN city varchar(20) NOT NULL;

**we can also use following syntax to add constraints**

ALTER TABLE table\_name  
ADD constraint (coloumn\_name);

EXAMPLE

ALTER TABLE Persons | ALTER TABLE Persons   
ADD UNIQUE (ID); | ADD NOT NULL (First\_Name);

**To rename table**

SYNTAX

ALTER TABLE old\_table | ALTER TABLE old\_table  
RENAME TO new\_table; | RENAME new\_table;

EXAMPLE

ALTER TABLE person   
RENAME TO people;

**To rename column**

SYNTAX

ALTER TABLE old\_table   
CHANGE old\_colunm TO new\_colunm datatype;

EXAMPLE

ALTER TABLE person  
CHANGE name TO firstname varchar(20);

The **SELECT** statement is used to select data from a database.

SELECT \* FROM table\_name; selects all data

SELECT column1, column2, ... selects **specific columns**FROM table\_name;

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

SELECT Country FROM Customers; selects all including the **duplicates** values

SELECT DISTINCT Country FROM Customers; selects only the **DISTINCT** values

The **SELECT TOP** clause is used to specify the number of records to return.

In **mysql** we use **LIMIT**

SYNTAX

SELECT column\_name  
FROM table\_nameWHERE condition  
LIMIT number;

EXAMPLE

SELECT \* FROM Customers  
WHERE Country='Germany'  
LIMIT 3;

SQL **ALIASES** are used to give a table, or a column in a table, a temporary name.

An alias only exists for the duration of that query.

An alias is created with the **AS** keyword.

SYNTAX

SELECT column\_name AS alias\_name  
FROM table\_name;

EXAMPLE

SELECT CustomerID AS ID, CustomerName AS Customer  
FROM Customers;

The **INSERT INTO** statement is used to insert new records in a table.

It is possible to write the INSERT INTO statement in two ways:

**1. Specify both the column names and the values to be inserted:**

SYNTAX

INSERT INTO table\_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);

EXAMPLE

INSERT INTO employee (id, name, age)  
VALUES (1, “prathamesh”, 22), (2, “rohan”, 23);

**2. Specify only values which are to be inserted:**

SYNTAX

INSERT INTO table\_name  
VALUES (value1, value2, value3, ...);

EXAMPLE

INSERT INTO employee   
VALUES (1, “prathamesh”, 22,“kolhapur”),(2, “rohan”, 23, “goa”);

The **INSERT INTO SELECT** statement copies data from one table and inserts it into another table.

The INSERT INTO SELECT statement requires that the data types in source and target tables match. **(table must be created previously)**

Copy all columns from one table to another table:

INSERT INTO *table2*  
SELECT \* FROM *table1*WHERE *condition*;

Copy only some columns from one table into another table:

INSERT INTO *table2*(*column1*, *column2*, *column3*, ...)  
SELECT *column1*, *column2*, *column3*, ...  
FROM *table1*  
WHERE *condition*;

**Clauses / Operators:**

The **WHERE** clause is used to filter records.

It is used to extract only those records that fulfil a specified condition.

SYNTAX

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
WHERE *condition*;

SELECT *name,city*  
FROM *employee*  
WHERE *where age>18*;

The **AND**and**OR** operators are used to filter records based on more than one condition:

1. TheANDoperator displays a record if all the conditions separated by AND are TRUE.

SELECT \* FROM Customers  
 WHERE Country='Germany' AND City='Berlin';

1. The ORoperator displays a record if any of the conditions separated by OR is TRUE.

SELECT \* FROM Customers  
 WHERE City='Berlin' OR City='München';

1. The **NOT** operator displays a record if the condition(s) is NOT TRUE.

SELECT \* FROM Customers  
 WHERE NOT Country='Germany';

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

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

SELECT \* FROM Customers  
ORDER BY Country;

SELECT \* FROM Customers  
ORDER BY Country DESC;

The **MIN()** function returns the smallest value of the selected column.

SELECT MIN(Price) AS SmallestPrice  
FROM Products;

The **MAX()** function returns the largest value of the selected column.

SELECT MAX(Price) AS LargestPrice  
FROM Products;

The **COUNT()** function returns the number of rows that matches a specified criterion.

SELECT COUNT(ProductID)  
FROM Products;

The **AVG()** function returns the average value of a numeric column.

SELECT AVG(Price)  
FROM Products;

The **SUM()** function returns the total sum of a numeric column.

SELECT SUM(Quantity)  
FROM OrderDetails;

The **IN** operator allows you to specify multiple values in a WHERE clause.

The **IN** operator is a shorthand for multiple OR conditions.

SELECT \* FROM Customers  
WHERE Country IN ('Germany', 'France', 'UK');

SELECT \* FROM Customers  
WHERE Country NOT IN ('Germany', 'France', 'UK');

SELECT \* FROM Customers  
WHERE Country IN (SELECT Country FROM Suppliers);

The**BETWEEN** operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

SELECT \* FROM Products  
WHERE Price BETWEEN 10 AND 20;

SELECT \* FROM Products  
WHERE Price NOT BETWEEN 10 AND 20;

The **GROUP BY** statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country;

The **HAVING** clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5

**SQL Constraints**

Constraints can be specified when the table is created with the CREATE TABLE statement,

or after the table is created with the ALTER TABLE statement.

CREATE TABLE table\_name (  
    column1 datatype *constraint*,  
    column2 datatype *constraint*,  
    column3 datatype *constraint*,  
    ....  
);

Constraints are used to limit the type of data that can go into a table

The following constraints are commonly used in SQL:

* [NOT NULL](https://www.w3schools.com/sql/sql_notnull.asp) - Ensures that a column cannot have a NULL value
* [UNIQUE](https://www.w3schools.com/sql/sql_unique.asp) - Ensures that all values in a column are different
* [PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
* [FOREIGN KEY](https://www.w3schools.com/sql/sql_foreignkey.asp) - Prevents actions that would destroy links between tables
* [CHECK](https://www.w3schools.com/sql/sql_check.asp) - Ensures that the values in a column satisfies a specific condition
* [DEFAULT](https://www.w3schools.com/sql/sql_default.asp) - Sets a default value for a column if no value is specified
* [CREATE INDEX](https://www.w3schools.com/sql/sql_create_index.asp) - Used to create and retrieve data from the database very quickly

The **NOT NULL** constraint enforces a column to NOT accept NULL values.

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255) NOT NULL,  
    Age int  
);

The **UNIQUE** constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have **many** UNIQUE constraints per table,

but **only one** PRIMARY KEY constraint per table.

CREATE TABLE Persons ( | CREATE TABLE Persons (  
    ID int NOT NULL UNIQUE, | ID int UNIQUE,   
    LastName varchar(255) NOT NULL, | LastName varchar(255) NOT NULL,  
    FirstName varchar(255), | FirstName varchar(255),   
    Age int | Age int   
); | );

**UNIQUE constraint on multiple columns**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CONSTRAINT UC\_Person UNIQUE (ID,LastName)  
);

The **PRIMARY KEY** constraint uniquely identifies each record in a table.

Primary keys must contain **UNIQUE** values, and **cannot contain NULL** values.(basically primary key is combination of unique and not null)

A table can have **only ONE** primary key; and in the table, this primary key can consist of single or multiple columns (fields).

CREATE TABLE Persons (  
    ID int NOT NULL PRIMARY KEY,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
);

**PRIMARY KEY constraint on multiple columns**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName)  
);

**Note:** In the example above there is only ONE PRIMARY KEY (PK\_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

**Auto-increment** allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

CREATE TABLE Persons (  
    Personid int PRIMARY KEY AUTO\_INCREMENT,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int  
    );

By default, the starting value for AUTO\_INCREMENT is 1, and it will increment by 1 for each new record.

To let the AUTO\_INCREMENT sequence start with another value, use the following SQL statement:

CREATE TABLE Persons (  
    Personid int PRIMARY KEY ,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255)  
    ) AUTO\_INCREMENT = 100;

The **FOREIGN KEY** constraint is used to prevent actions that would destroy links between tables.

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the [PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp) in another table.

The table with the foreign key is called the **child table**, and the table with the primary key is called the referenced or **parent table**.

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    FOREIGN KEY (PersonID) REFERENCES Persons(ID)  
);

Instead of using above syntax use following syntax it is helpful while deleting the foreign key using its name

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    CONSTRAINT FK\_PersonOrder FOREIGN KEY (PersonID)  
    REFERENCES Persons(PersonID)  
);

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.

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

**CHECK constraint on multiple columns**

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    City varchar(255),  
    CONSTRAINT CHK\_Person CHECK (Age>=18 AND City='kolhapur')  
);

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 Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    City varchar(255) DEFAULT 'Sandnes'  
);

The DEFAULT constraint can also be used to insert system values, by using functions like [GETDATE()](https://www.w3schools.com/sql/func_sqlserver_getdate.asp):

CREATE TABLE Orders (  
    ID int NOT NULL,  
    OrderNumber int NOT NULL,  
    OrderDate date DEFAULT GETDATE()  
);

## **CONSTRAINTS on ALTER TABLE**

**NOT NULL**

ALTER TABLE Persons  
MODIFY Age int NOT NULL;

ALTER TABLE Persons (To drop not null constraints use this syntax)  
MODIFY Age int;

**UNIQUE**

ALTER TABLE Persons  
MODIFY ID int UNIQUE;

ALTER TABLE Persons  
DROP INDEX ID ;

ALTER TABLE Persons  
ADD CONSTRAINT UC\_Person UNIQUE (ID,LastName);

ALTER TABLE Persons  
DROP INDEX UC\_Person;

**PRIMARY KEY**

ALTER TABLE Persons  
ADD PRIMARY KEY (ID);

ALTER TABLE Persons  
DROP PRIMARY KEY;

ALTER TABLE Persons  
ADD CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName);

ALTER TABLE Persons  
DROP CONSTRAINT PK\_Person;

**FOREIGN KEY**

ALTER TABLE Orders  
ADD FOREIGN KEY (PersonID) REFERENCES Persons(ID); **OR**

ALTER TABLE Orders  
ADD CONSTRAINT FK\_PersonOrder  
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

ALTER TABLE Orders  
DROP CONSTRAINT FOREIGN KEY FK\_PersonOrder;

**CHECK**

ALTER TABLE Persons  
ADD CHECK (Age>=18); **OR**

ALTER TABLE Persons   
ADD CONSTRAINT CHK\_PersonAge CHECK (Age>=18);

ALTER TABLE Persons  
DROP CONSTRAINT CHK\_PersonAge;

why we need to give name to constraint?

beacuse its helpful to delete constraints

**DEFAULT**

ALTER TABLE Persons  
ALTER City SET DEFAULT 'Sandnes';

ALTER TABLE Persons  
ALTER City DROP DEFAULT;

**SQL JOINS**

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

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



The INNER JOIN keyword selects records that have matching values in both tables.

### Syntax

SELECT column\_name(s)  
FROM table1  
INNER JOIN table2**ON** table1.column\_name = table2.column\_name;

SELECT Orders.OrderID, Customers.CustomerName  
FROM Orders  
INNER JOIN Customers **ON** Orders.CustomerID = Customers.CustomerID;

**LEFT JOIN**: Returns all records from the left table, and the matched records from the right table



The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

Syntax

SELECT *column\_name(s)*  
FROM *table1*  
LEFT JOIN *table2*ON *table1.column\_name*=*table2.column\_name*;

**RIGHT JOIN**: Returns all records from the right table, and the matched records from the left table



The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2ON table1.column\_name = table2.column\_name;

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



SELECT column\_name(s)  
FROM table1  
FULL OUTER JOIN table2ON table1.column\_name = table2.column\_name;

**UNION** operator is used to combine the result-set of two or more SELECT statements.

The SQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use this UNION clause, each SELECT statement must have

* The same number of columns **selected**

**( not needs of same no of columns )**

* The same number of column expressions
* The same data type and
* Have them in the same order

**SYNTAX**

SELECT column\_name(s) FROM table1  
UNION  
SELECT column\_name(s) FROM table2;

The UNION operator selects only distinct values by default. To allow duplicate values, use ***UNION ALL***:

SELECT *column\_name(s)* FROM *table1*  
UNION ALL  
SELECT *column\_name(s)* FROM *table2*;

**Subquery :**  or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow −

* Subqueries must be enclosed within parentheses.
* A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
* An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
* Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.

**Syntax:** There is not any general syntax for Subqueries. However, Subqueries are seen to be used most frequently with SELECT

SELECT *column\_name(s)*  
FROM *table\_name*  
WHERE *column\_name expression operetor*(SELECT *column\_name*FROM *table1* WHERE CONDITON);

SELECT first\_name, last\_name

FROM employees

WHERE salary > (SELECT avg(salary) FROM employees);

# **SQL Join vs Subquery**

The advantage of a join includes that it executes faster.

Disadvantage of using joins includes that they are not as easy to read as subqueries.

More joins in a query means the database server has to do more work, which means that it is more time consuming process to retrieve data

Subqueries divide the complex query into isolated parts so that a complex query can be broken down into a series of logical steps.

It is easy to understand and code maintenance is also at ease.

**Conclusion :**   
A subquery is easier to write, but a joint might be better optimized by the server. For example a Left Outer join typically works faster because servers optimize it.

**VIEWS**

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the CREATE VIEW statement.

There are two types of creating views

**Creating View from a single table:**

SYNTAX

### CREATE VIEW view\_name AS SELECT column1, column2, ... FROM table\_name WHERE condition;

EXAMPLE

CREATE VIEW Brazil\_Customers AS  
SELECT CustomerName, Age  
FROM Customers  
WHERE Country = 'Brazil';

**Creating View from multiple tables:**

SYNTAX

### CREATE VIEW view\_name AS SELECT table\_name1.column1, table\_name2.column2, ... FROM table\_name1,table\_name2 WHERE condition;

EXAMPLE

CREATE VIEW Brazil\_Customers AS  
SELECT Customers.CustomerName, Customers.Age,Contact.phoneno  
FROM Customers,Contact  
WHERE Customers.Country = Contact.Country;

DROP VIEW view\_name; (A view is deleted with the DROP VIEW statement)

DELETE FROM view\_name WHERE condition; (Delete records form views )

Note:- we cannot delete records from **join view**(view created from multiple tables)

### UPDATE view\_name  SET column\_name= value  WHERE condition;

### There are certain conditions needed to be satisfied to update a view

1. TheSELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.

UPDATE OR REPLACE VIEW statement OR ALTER VIEW statement

UPDATE OR REPLACE VIEW view\_name AS

SELECT column1,coulmn2,..

FROM table\_name

WHERE condition;

OR

ALTER VIEW view\_name AS

SELECT column1,coulmn2,..

FROM table\_name

WHERE condition;

**This both statements replace original view and create new view with old name**

**Changes in views**

Any changes made in **views** will affect the **original table**

Changes like – inserting new records, deleting records, updating records

**Drop view view\_name** **won’t affect the original table**

**Changes in original table**

Any changes made in original table will affect views

Changes like – inserting new records,deleting rows, updating records , **dropping original table also affect the view unlike views**

Q-can we insert records in view? – YES

Q-can we create view from another view? – YES

Q-suppose we have created two views from one table and made changes in either of one view does it will affect the table and other view-YES

**Stored procedure**

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

**SYNTAX**

**DELIMITER &&** don’t forget to use spacebarCREATE PROCEDURE procedure\_name() don’t forget to add parenthesis

BEGIN

SELECT column\_name1,column\_name2

FROM table\_name

WHERE condition;

**EDN &&**

**EXAMPLE**

**DELIMITER &&** don’t forget to use spacebar CREATE PROCEDURE emp\_55() don’t forget to add parenthesis

BEGIN

SELECT name,age,city,salary

FROM employee

WHERE age>=55;

**EDN &&**

**Call emp\_55;**

**Why we use delimiter**

Delimiters are used when we need to define the stored procedures as well as to create triggers. Default delimiter is semicolon.

When we are writing SQL statements, we use the semicolon to separate two different statements like in the example below:

SELECT \* FROM employees;

SELECT \* FROM players;

MySQL workbench uses the delimiter semicolon (;) to separate two statements and then execute each statement separately.

But, a stored procedure contains multiple statements segregated by a semicolon (;).

If we are using the MySQL client program to define a stored procedure that consists of semicolon characters, the program will not treat the whole stored procedure as a single statement, but as many statements.

The command which we can use to redefine the delimiter is:

DELIMITER delimiter\_character don’t forget to use spacebar

EXAMPLE

DELIMITER &&

DELIMITER //

DELIMITER $$

The delimiter\_character consists of a single character or multiple characters like for example: // ,$$ ,&& etc.

**DELIMITER &&** don’t forget to use spacebar

CREATE PROCEDURE emp\_55() don’t forget to add parenthesis

BEGIN

SELECT name,age,city,salary

FROM employee

WHERE age>=55;

**EDN &&**

**Code Explanation:**

* Change the default delimiter to &&.
* Now, use a semicolon (;) in the body and && after the END keyword to end the procedure.
* So that default delimiter back to a semicolon (;).

Q-In place of semicolon being used as the default delimiter, we would like to use //. Is it possible in SQL?

Yes, it is possible. The Syntax for this is given below:

**Query:**

Delimeter //

select\*from world//

# **SQL Trigger**

**Trigger:** A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs

SYNTAX

**delimiter &&**

create trigger trigger\_name

before/after

insert/update/delete

**on** table\_name don’t forget to add **on** keyword

for each row don’t forget to add **for each row**

begin

trigger body ; don’t forget to add semicolon at the end of body

**end&&**

EXAMPLE

**delimiter &&**

create trigger ps1

before insert

**on** student don’t forget to add **on** keyword

for each row

begin

if new.age<0 then set new.age=18;

**end if &&**

A **trigger is called a special procedure** because it cannot be called directly like a stored procedure. The key distinction between the trigger and procedure is that a trigger is called **automatically** when a data modification event occurs against a table. A stored procedure, on the other hand, must be invoked directly.

Before triggers can be used to update or validate record values before they are saved to the database.  
  
After triggers can be used to access field values that are set by the database

# **MySQL Functions**

SELECT OrderID, Quantity,  
CASE  
    WHEN Quantity > 30 THEN "The quantity is greater than 30"  
    WHEN Quantity = 30 THEN "The quantity is 30"  
    ELSE "The quantity is under 30"  
END  
FROM OrderDetails;

SELECT OrderID, Quantity, IF(Quantity>10, "MORE", "LESS")  
FROM OrderDetails;

SELECT CURRENT\_DATE(); "YYYY-MM-DD"

SELECT CURRENT\_TIME(); "HH-MM-SS"

SELECT CURRENT\_TIMESTAMP(); "YYYY-MM-DD HH-MM-SS"