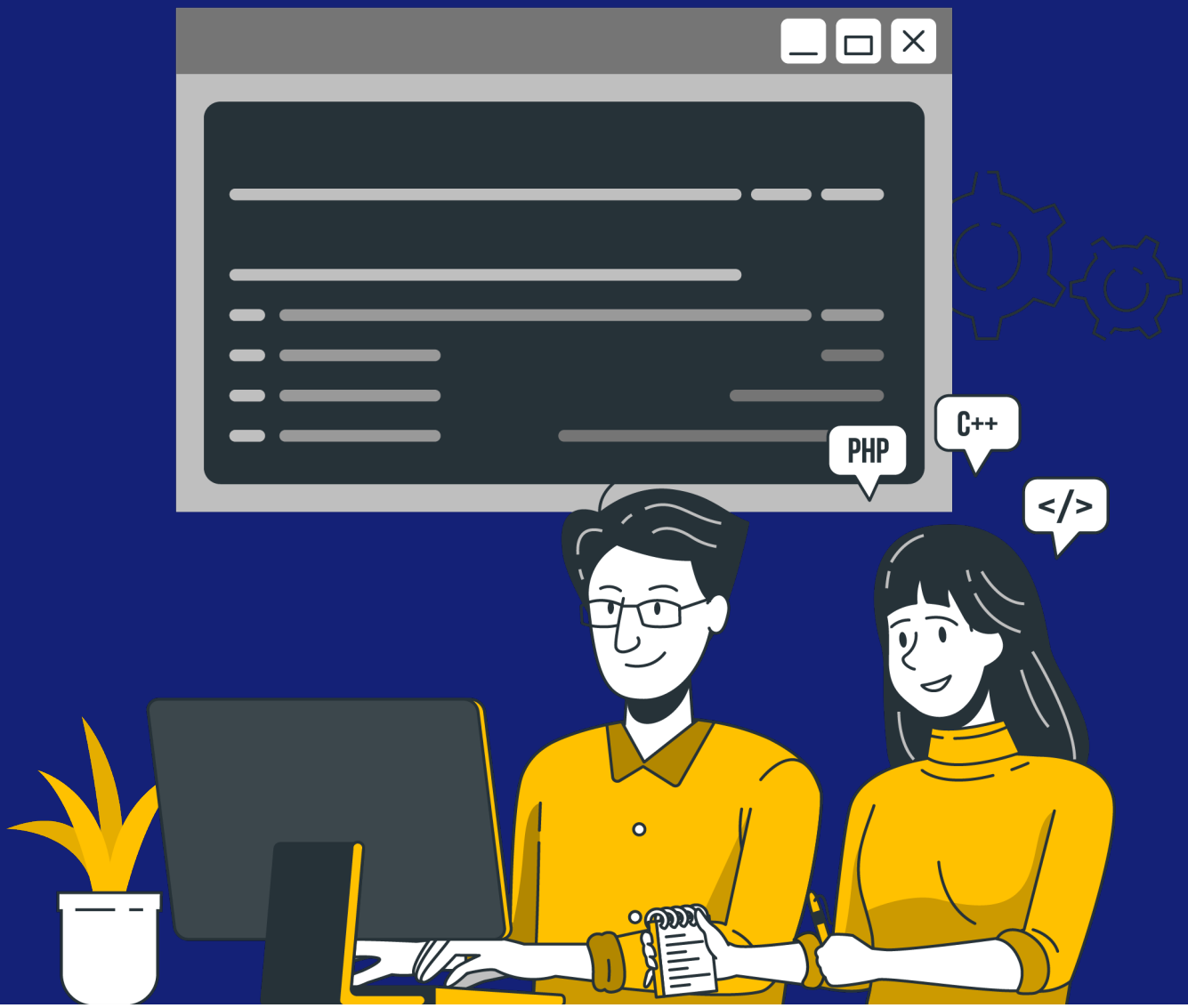
PW SKILLS

# Lesson Plan

# SQL Commands



# DDL

Structured Query Language (SQL), as we are all aware, is a database language that allows us to conduct certain operations on an already-existing database as well as utilise it to construct new databases. For the necessary actions, SQL employs specific commands like CREATE, DROP, INSERT, etc.

A table receives instructions from SQL statements. It is used to perform various operations on the database. Additionally, it is utilised to carry out particular duties, activities, and data inquiries. Table creation, data addition, table deletion, table modification, and user permission setting are just a few of the activities that SQL is capable of.

The five primary categories into which these SQL instructions fall are as follows:

- Data Definition Language (DDL)
- Data Query Language (DQL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- Transaction Control Language (TCL)
- We will now examine each of these in more depth.

# Data Definition Language (DDL).

The SQL statements that may be used to specify the database structure make up DDL, or Data Definition Language. It is used to construct and alter the structure of database objects in the database and only works with descriptions of the database schema. Although data cannot be created, modified, or deleted with DDL, database structures can. In most cases, a typical user shouldn't use these commands; instead, they should use an application to access the database.

# Commands in DDL

We will discuss the following DDL instructions in this section.

- Create
- Alter
- Truncate
- Drop
- Rename

Let's talk about each one in turn.

### CREATE

In SQL, a new table may be created with this command. Information like the table name, column names, and datatypes must be provided by the user.

**Syntax –**

```
CREATE TABLE table_name
(
column_1 datatype,
column_2 datatype,
column_3 datatype,
....
);
```

**Example:**
To store student data from a certain college, we must construct a table. Create syntax would look like this.

```
create database PW
USE PW
CREATE TABLE Student_info
(
College_Id int (5),
College_name varchar(30),
Branch varchar(10)
);
```

## ALTER

The existing table's columns can be changed, deleted, or added with this command. The user may quickly add, delete, or change tasks; however, they require knowledge of the name of the current table.

## SYNTAX

```
ALTER TABLE table_name
ADD column_name datatype;
```

**Example:**
We want to add a new column for Percentage to our Student_info database. The syntax would be as shown in the list below.

## TRUNCATE

The table's structure remains even after all of the rows are removed with this command.

**Syntax**

```
The syntax for deleting an existing table.
TRUNCATE TABLE table name;
```

**Example:**
The College Authority wishes to preserve the table structure but delete all student information for fresh batches. They may issue the following command.

```
TRUNCATE TABLE Student_info;
```

### DROP

This command is used to delete an existing table from the database, complete with its structure.

### SYNTAX

```
A table can be dropped using syntax.
DROP TABLE table_name;
```

**Example:**
As an illustration, suppose the College Authority decides to modify its database by eliminating the Student_info Table.

```
DROP TABLE Student_info;
```

### RENAME

Using the simple RENAME command, a table's name may be changed with or without data present.
Any table object may be renamed at any moment.

**Syntax**

```
RENAME TABLE <Table Name> To <New_Table_Name>;
```

**Example:**
We may use the rename command as follows to alter the table's name from Student_info To ST_INFO;

```
RENAME TABLE Student_info To ST_INFO;
```

# DML(Data Manipulation Language)

The majority of SQL statements are part of the DML, or Data Manipulation Language, which is used to manipulate data that is available in databases. It is the part of the SQL statement in charge of managing database and data access. Essentially, DML statements and DCL statements belong together.

# DML commands list

**INSERT**: Data is inserted into a table using the INSERT command.

**UPDATE**: A table's existing data is updated using UPDATE.

**DELETE**: Delete records from a database table using the DELETE command.

**LOCK**: Concurrency under table management.

# INSERT

To add a new row or record to a table, use the SQL INSERT INTO command. The SQL INSERT INTO command can insert records in one of two ways.

**SQL INSERT Query**

**Only Values**

In the first approach, the column names are omitted and just the value of the data to be entered is specified.

**INSERT INTO Syntax:**

```
INSERT INTO table_name VALUES (value1, value2, value3);
table_name: name of the table. value1, value2
```

**Column Names and Values Both**

In the second approach, we will specify the columns we wish to fill as well as the values that go with each one, as seen below:

**Insert Data in Specified Columns – Syntax:**

```
INSERT INTO table_name (column1, column2, column3)
VALUES ( value1, value2, value3);
```

Here, table_name: name of the table.
column1: name of first column, second column .
value1, value2, value3  value of first column, second column,... for the new record
Suppose there is a Student database and we want to add values.

| ROLL_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | RAMAN | DEHRADUN | xxxxxxxxxxx | 19 |
| 2 | RAHA | GORAKHPUR | xxxxxxxxxxx | 19 |
| 3 | SUJATA | ROHINI | xxxxxxxxxxx | 21 |
| 4 | SURBHI | ROHINI | xxxxxxxxxxx | 19 |
| 5 | SUJOY | ROHINI | xxxxxxxxxxx | 21 |
| 6 | RAM | GORAKHPUR | xxxxxxxxxxx | 22 |

## Method 1 (Inserting only values) – SQL INSERT Query

The following formula is used if we just wish to enter values:

**Query:**

```
INSERT INTO Student VALUES
('15','HARI','KOLKATA,
'XXXXXXXXXX','19');
```

**Output:**
The table **Student** will look like this:

| ROLL_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | RAMAN | DEHRADUN | xxxxxxxxxxx | 19 |
| 2 | RAHA | GORAKHPUR | xxxxxxxxxxx | 19 |
| 3 | SUJATA | ROHINI | xxxxxxxxxxx | 21 |
| 4 | SURBHI | ROHINI | xxxxxxxxxxx | 19 |
| 5 | SUJOY | ROHINI | xxxxxxxxxxx | 21 |
| 6 | RAM | GORAKHPUR | xxxxxxxxxxx | 22 |
| 7 | HARI | KOLKATA | xxxxxxxxxxx | 19 |

## Method 2 (Inserting values in only specified columns) – SQL INSERT INTO Statement

The following query is used if we wish to insert values into the given columns:

**Query:**

```
INSERT INTO Student (ROLL_NO,
NAME, Age) VALUES ('8','PRANAV','19');
```

**Output:**
The table **Student** will look like this:

| ROLL_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | RAMAN | DEHRADUN | xxxxxxxxxxx | 19 |
| 2 | RAHA | GORAKHPUR | xxxxxxxxxxx | 19 |
| 3 | SUJATA | ROHINI | xxxxxxxxxxx | 21 |
| 4 | SURBHI | ROHINI | xxxxxxxxxxx | 19 |
| 5 | SUJOY | ROHINI | xxxxxxxxxxx | 21 |
| 6 | RAM | GORAKHPUR | xxxxxxxxxxx | 22 |
| 7 | PRANAV | null | null | 19 |

Keep in mind that null is used to fill in the columns for which values are missing. Which default values are used for those columns?

## UPDATE

An existing table's data can be updated in a database using the UPDATE command in SQL. Using the UPDATE statement, we may update both single and numerous columns depending on our needs.

In a nutshell, we can state that the SQL commands (UPDATE and DELETE) are used to modify data that has already been stored in the database. A WHERE clause is used in the SQL DELETE statement.

**Syntax**

```
UPDATE table_name SET column1 = value1, column2 = value2,...
WHERE condition;
```

Here, table_name: name of the table

column1: name of first , second, third column....

value1: new value for first, second, third column....

condition: condition to select the rows for which the

values of columns needs to be updated.

**Explanation of the Parameter**

**UPDATE**: A command is used to change a table's column value.

**WHERE**: The condition that we wish to apply to the table is specified by the WHERE clause.

In the above query, the WHERE clause is used to select the rows for which the columns need to be modified, and the SET statement is used to assign new values to the specific column. The columns in all the rows will be changed if the WHERE clause has not been used. Therefore, the WHERE clause is utilised to select the specific rows.

**Query:**

```
CREATE TABLE employee(
    employeeID INT PRIMARY KEY,
    employeeName VARCHAR(50),
     LastName VARCHAR(50),
     Country VARCHAR(50),
     Age int(2),
   Phone int(10)
);

-- Insert some sample data into the employee table
INSERT INTO employee(employeeID, employeeName, LastName, Country, Age, Phone)
VALUES (1, 'Shahil', 'Thakur', 'India',25,  777777778),
       (2, 'Aman ', 'sungh', 'Australia',21, 888888888),
       (3, 'Navi', 'Thakur', 'Sri lanka',25, 777777777),
       (4, 'Adil', 'Arora', 'Austria',21, 333333333),
       (5, 'Nisha', 'Jain', 'Spain',22, 777888653);

       Select * from employee;
```

**OUTPUT:**

| employeeID | employeeName | LastName | Country | Age | Phone |
|---|---|---|---|---|---|
| 1 | Shahil | Thakur | India | 25 | 777777778 |
| 2 | Aman | sungh | Australia | 21 | 888888888 |
| 3 | Navi | Thakur | Sri lanka | 25 | 777777777 |
| 4 | Adil | Arora | Austria | 21 | 333333333 |
| 5 | Nisha | Jain | Spain | 22 | 777888653 |

**UPDATE SINGLE COLUMN**

In rows where the Age is 22, update the NAME column and set the value to "Shahil."

```
UPDATE employee SET employeeName
= 'Shahil' WHERE Age = 22;
```

**OUTPUT:**

| | employeeID | employeeName | LastName | Country | Age | Phone |
|---|---|---|---|---|---|---|
| ▶ | 1 | Shahil | Thakur | India | 25 | 777777778 |
| | 2 | Aman | sungh | Australia | 21 | 888888888 |
| | 3 | Navi | Thakur | Sri lanka | 25 | 777777777 |
| | 4 | Adil | Arora | Austria | 21 | 333333333 |
| | 5 | Shahil | Jain | Spain | 22 | 777888653 |
| * | NULL | NULL | NULL | NULL | NULL | NULL |

**Multiple Columns Updating**

Where employeeID is 1, change the columns NAME to "Navi" and Country to "USA."

```
UPDATE employee SET employeeName = 'Satyam',
Country = 'USA' WHERE employeeID = 1;
```

**OUTPUT:**

| employeeID | employeeName | LastName | Country | Age | Phone |
|---|---|---|---|---|---|
| 1 | Satyam | Thakur | USA | 25 | 777777778 |
| 2 | Aman | sungh | Australia | 21 | 888888888 |
| 3 | Navi | Thakur | Sri lanka | 25 | 777777777 |
| 4 | Adil | Arora | Austria | 21 | 333333333 |
| 5 | Shahil | Jain | Spain | 22 | 777888653 |

**Removal of WHERE Clause**

All of the rows will be updated if the WHERE clause is left out of the update query.

```
UPDATE employee SET employeeName = 'Shubham';
```

**OUTPUT:**

| employeeID | employeeName | LastName | Country | Age | Phone |
|---|---|---|---|---|---|
| 1 | Shubham | Thakur | USA | 25 | 777777778 |
| 2 | Shubham | sungh | Australia | 21 | 888888888 |
| 3 | Shubham | Thakur | Sri lanka | 25 | 777777777 |
| 4 | Shubham | Arora | Austria | 21 | 333333333 |
| 5 | Shubham | Jain | Spain | 22 | 777888653 |

## DELETE

The SQL DELETE Statement can be used to remove existing records from a table. Depending on the criteria we set in the WHERE clause, we can remove either a single record or several entries.

**Syntax**

```
DELETE FROM table_name WHERE some_condition;
```

**Note**: Depending on the criteria we use in the WHERE clause, we can remove a single record or several entries. The table will be empty if the WHERE clause is left out since all of the records will be erased.

Let's take a sample table to understand:

```
CREATE TABLE employee_info
(
employee_Id int (5),
Name varchar(30),
EMP_EMAIL_ID varchar(30),
Dept varchar(30)
);
select * from employee_info
INSERT INTO employee_info (employee_Id, Name, EMP_EMAIL_ID,Dept)
VALUES (1, 'Shahil', 'shah@yahoo.com', 'sales'),
       (2, 'Aman ', 'Aman@yahoo.com', 'development'),
       (3, 'Navi', 'Nav@yahoo.com', 'HR'),
       (4, 'Adil', 'Ad@yahoo.com','IT'),
       (5, 'Nisha', 'Nis@yahoo.com', 'Sales');
```

| employee_Id | Name | EMP_EMAIL_ID | Dept |
|---|---|---|---|
| 1 | Shahil | shah@yahoo.com | sales |
| 2 | Aman | Aman@yahoo.com | development |
| 3 | Navi | Nav@yahoo.com | HR |
| 4 | Adil | Ad@yahoo.com | IT |
| 5 | Nisha | Nis@yahoo.com | Sales |

**Deleting One Record**

Remove the rows with NAME equal to "Navi." The third row will be the only one deleted.

```
57 •    DELETE FROM employee_info WHERE NAME = 'Navi';
58 •    select * from employee_info
```

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

| employee_Id | Name | EMP_EMAIL_ID | Dept |
|---|---|---|---|
| 1 | Shahil | shah@yahoo.com | sales |
| 2 | Aman | Aman@yahoo.com | development |
| 4 | Adil | Ad@yahoo.com | IT |
| 5 | Nisha | Nis@yahoo.com | Sales |

**Multi-Record Erasure**

Employee_info table records with the department "Development" should be deleted. The first row and the seventh row will be removed as a result of this.

**Query:**

```
DELETE FROM employee_info
WHERE Dept = 'Development';
```

**OUTPUT:**

| employee_Id | Name | EMP_EMAIL_ID | Dept |
|---|---|---|---|
| 1 | Shahil | shah@yahoo.com | sales |
| 4 | Adil | Ad@yahoo.com | IT |
| 5 | Nisha | Nis@yahoo.com | Sales |

**Delete each and every record.**

As seen below, there are two queries to achieve this.

**Query:**

```
DELETE FROM employee_info;
```

Or

```
DELETE * FROM employee_info;
```

There will be no more records to display since the table will be cleared of all data. The employee_info table will be left empty!

**OUTPUT:**

There will be no more records to display since the table will be cleared of all data. The employee_info table will be left empty!

| | employee_Id | Name | EMP_EMAIL_ID | Dept |
|---|---|---|---|---|

# DQL

DQL statements are employed to conduct queries on the information contained in schema objects. The DQL Command's objective is to obtain a schema relation based on the query that is supplied to it. DQL may be explained as follows: It's a part of a SQL statement that enables retrieving data from a database and applying order to it. The SELECT statement is present. With the help of this command, you may retrieve data from the database and manipulate it. A second temporary table, known as a front-end, is created when a SELECT is executed on a table or tables. This table is presented or may even be used by the programme.

## SELECT

Data from a database may be retrieved or fetched using the SELECT Statement in SQL.
Either the full table, or a subset of it, can be retrieved. A result table contains the data that was returned. The result set also refers to this result table. The columns we wish to be displayed in the query result and, optionally, whatever column headings we like to appear above the result table, are specified in the SELECT clause of a SELECT command statement.

The database server examines the select clause, which is the first and one of the last clauses of the select statement, last. This is because we need to be aware of every potential column that may be included in the final result before we can decide what to include in the final result set.

Let’s create a table first and insert some values:

```
CREATE TABLE Customers(
        CustomerID INT PRIMARY KEY,
    CustomerName VARCHAR(50),
    LastName VARCHAR(50),
    Country VARCHAR(50),
    Age int(2),
  Phone int(10)
);

INSERT INTO Customers (CustomerID, CustomerName, LastName, Country, Age, Phone)
VALUES (1, 'Shiv', 'Thanvi', 'India','29',1234516665),
       (2, 'Amandeep ', 'Singh', 'Austria','22',1234516661),
       (3, 'Navneet', 'Kumar', 'Canada','22',1234516668),
       (4, 'Aditya', 'Arpan', 'Austria','27',1234516669),
       (5, 'Nidhi', 'Jain', 'Spain','22',1234516666);
```

**The output will be:**

| CustomerID | CustomerName | LastName | Country | Age | Phone |
|---|---|---|---|---|---|
| 1 | Shiv | Thanvi | India | 29 | 1234516665 |
| 2 | Amandeep | Singh | Austria | 22 | 1234516661 |
| 3 | Navneet | Kumar | Canada | 22 | 1234516668 |
| 4 | Aditya | Arpan | Austria | 27 | 1234516669 |
| 5 | Nidhi | Jain | Spain | 22 | 1234516666 |
| NULL | NULL | NULL | NULL | NULL | NULL |

## To retrieve any table column.

**Syntax:**

```
SELECT column1,column2 FROM table_name;
```

column1 , column2: names of the fields of the table
table_name: from where we want to apply query
All rows in the table containing the fields column1 and column2 will be returned by this query.

## SELECT Statement

- To retrieve the whole table or all of its fields:

**Syntax:**

```
SELECT * FROM table_name;
```

asterisks represent all attributes of the table

| CustomerID | CustomerName | LastName | Country | Age | Phone |
|---|---|---|---|---|---|
| 1 | Shiv | Thanvi | India | 29 | 1234516665 |
| 2 | Amandeep | Singh | Austria | 22 | 1234516661 |
| 3 | Navneet | Kumar | Canada | 22 | 1234516668 |
| 4 | Aditya | Arpan | Austria | 27 | 1234516669 |
| 5 | Nidhi | Jain | Spain | 22 | 1234516666 |
| NULL | NULL | NULL | NULL | NULL | NULL |

- To fetch the fields CustomerName, Age from the table Customer:

**Syntax:**

SELECT CustomerName, LastName FROM Customer;

**OUTPUT:**

| CustomerName | LastName |
|---|---|
| Shiv | Thanvi |
| Amandeep | Singh |
| Navneet | Kumar |
| Aditya | Arpan |
| Nidhi | Jain |

## SELECT Statement with WHERE Clause

If we want to view table values that meet certain criteria, we would use the SELECT statement in conjunction with the WHERE clause.

**Query:**

```
SELECT CustomerName FROM Customer where Age = '21';
```

```
SELECT CustomerName FROM Customers where Country = 'Spain';
```

| CustomerName |
|---|
| Nidhi |

## SELECT Statement with WHERE Clause

**Query:**

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

**OUTPUT:**

| | Age | COUNT(CustomerID) |
|---|---|---|
| ▶ | 29 | 1 |
| | 22 | 3 |
| | 27 | 1 |

## HAVING Clause in a SELECT Statement

Let's create a table first to show the HAVING Clause:

```
Create table employee_detail
 ( ID INT(20),
 Employee_name varchar(200),
 Employee_salary int(8),
 Employee_dept varchar(200),
 no_of_years int(2));

 select * from employee_detail
INSERT INTO employee_detail (ID, Employee_name, Employee_salary, Employee_dept,no_of_years )
VALUES (1, 'Shiv', 70000, 'Sales', 5),
       (2, 'Amandeep ', 90000, 'HR',10),
       (3, 'Navneet', 60000, 'Finance','6'),
       (4, 'Aditya', 56000, 'Finance','12'),
       (5, 'Nidhi', 80000, 'Sales','11');
```

**Query:**

```
SELECT Employee_dept, sum(Employee_salary) as Salary
FROM employee_detail
GROUP BY Employee_dept
HAVING SUM(Employee_salary) >= 50000;
```

**OUTPUT:**

| | Employee_dept | Salary |
|---|---|---|
| ▶ | Sales | 150000 |
| | HR | 90000 |
| | Finance | 116000 |

## SQL's SELECT statement with the ORDER BY clause

**Query:**

```
SELECT * FROM  employee_detail ORDER BY Employee_salary DESC;
```

**OUTPUT:**

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

| | ID | Employee_name | Employee_salary | Employee_dept | no_of_years |
|---|---|---|---|---|---|
| ▶ | 2 | Amandeep | 90000 | HR | 10 |
| | 5 | Nidhi | 80000 | Sales | 11 |
| | 1 | Shiv | 70000 | Sales | 5 |
| | 3 | Navneet | 60000 | Finance | 6 |
| | 4 | Aditya | 56000 | Finance | 12 |

# TCL

A group of tasks are combined into a single execution unit using transactions. Each transaction starts with a particular job and is completed after every activity in the group has been properly completed. The transaction fails if any of the tasks are unsuccessful. Therefore, there are only two outcomes for a transaction: success or failure.
As a result, the following TCL commands are used to manage how a transaction is carried out:

## COMMIT

A command used to save changes made during a transaction in the database permanently. Sets of one or more SQL statements that are performed concurrently as a single piece of work are known as transactions. Until a commit is published, the changes performed during a transaction are not visible to other users or programmes. If a mistake is made during the transaction, the "rollback" command can be used to reverse any changes performed during that transaction.

Let's use a straightforward illustration to show this:

```
CREATE TABLE Employees_detail (
    ID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Dept VARCHAR(50)
);
```

Let's imagine that we want to execute a transaction that adds two new workers to the "Employees_detail" table:

```
BEGIN;
INSERT INTO Employees_detail (ID, FirstName, LastName, Dept)
VALUES (1, 'John', 'Doe', 'Marketing');
INSERT INTO Employees_detail (ID, FirstName, LastName, Dept)
VALUES (2, 'Jane', 'Smith', 'Finance');
COMMIT;
```

The BEGIN; statement, which denotes the start of a transaction block, is used to initiate the transaction in this example. The "Employees_detail" database is then expanded with two additional employees using two INSERT commands. Other users or applications are not yet able to see these changes.
The modifications are then permanently saved to the database by executing the COMMIT command. The new employees, "John Doe" and "Jane Smith," are now a part of the "Employees_detail" table and are visible to other users and apps running database queries after the commit.

We may use the ROLLBACK; statement rather than COMMIT; if, for any reason, a problem arose once the transaction had begun (for instance, an error occurred) and we needed to cancel the transaction and reverse the modifications.

**NOTE**:
Remember that not all SQL databases enable transactions, and that various database management systems (DBMS) may have somewhat different COMMIT command behaviour and syntax. However, across all SQL implementations with transaction support, the fundamental idea of committing changes to make them permanent and visible to others remains the same.Instead of using the COMMIT; statement, we might use the ROLLBACK; one if a transaction had already begun (for instance, an error had happened) and we needed to cancel it and reverse the modifications.

## ROLLBACK

The ROLLBACK command is used to reverse the effects of a transaction and return the database to its initial state. It is often used when a transaction goes wrong or when we wish to disregard the modifications that have been made for any other reason.
Let's say we want to conduct a transaction to update an employees' dept:

```
BEGIN;
UPDATE Employees_detail
SET Dept = 'Sales'
WHERE ID = 2;
ROLLBACK;
```

**OUTPUT:**

```
170 •  Select * from Employees_detail
171
172
173
```

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

| ID | FirstName | LastName | Dept |
|---|---|---|---|
| 1 | John | Doe | Marketing |
| 2 | Jane | Smith | Sales |
| NULL | NULL | NULL | NULL |

The BEGIN; phrase in this instance signals the start of the transaction. The UPDATE statement is then used to try to update a customer with CustomerID 2's Dept. However, if a mistake is made (for example, as a result of a typo or a constraint violation), we must reject the transaction's modifications. When we use the ROLLBACK; command to reverse an update, the database returns to its initial state.

## SAVEPOINT

The ability to construct points inside a transaction that you may subsequently roll back to if necessary is known as a savepoint. A set of one or more SQL statements run sequentially is referred to as a transaction. Savepoints are in handy when you wish to create interim checkpoints while a transaction is being executed so you can subsequently go back to that particular moment without rolling the transaction back in its entirety.

The primary use is to provide you more precise control over transactions, particularly when you need to handle mistakes or exceptions in a precise manner.

Let's walk through a SQL savepoint usage example. Let's say we have a straightforward database that has the tables "Customers" and "Orders." Each customer's orders are included in the "Orders" database along with information about them in the "Customers" section.

Let's build the tables and add some test data to them:

```
CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(50),
    Email VARCHAR(100)
);
select * from Customer
```

```
INSERT INTO Customer (CustomerID, Name, Email) VALUES
  (1, 'John Doe', 'john@example.com'),
  (2, 'Jane Smith', 'jane@example.com');
```

```
CREATE TABLE Order_detail(
  OrderID INT PRIMARY KEY,
  CustomerID INT,
  Product VARCHAR(50),
  Quantity INT,
  TotalAmount DECIMAL(10, 2)
);
```

```
INSERT INTO Order_detail (OrderID, CustomerID, Product, Quantity, TotalAmount) VALUES
  (101, 1, 'Product A', 2, 100.00),
  (102, 1, 'Product B', 1, 50.00),
  (103, 2, 'Product C', 3, 150.00);
```

Let's now take a look at a situation where we need to update two connected things: the email address of a customer in the "Customers" record and the total cost of one of their orders in the "Orders" table. Both updates must be successful in order for any one to be implemented. We can utilise a transaction with savepoints to do this.

The SQL code would seem as follows:

```
BEGIN;
  UPDATE Customer
  SET Email = 'john.doe@example.com'
  WHERE CustomerID = 1;
  SAVEPOINT email_savepoint;

  select * from Customer
```

In the above query we will start the transaction and updating the customer's email. We will create a savepoint after updating the email.

```
UPDATE Order_detail
SET TotalAmount = 120.00
WHERE OrderID = 101;


select * from Order_detail
```

Again we will try to update the totalAmount for one order

```
UPDATE Order_detail
SET TotalAmount = 120.00
WHERE OrderID = 101;


select * from Order_detail
```

Now, lets say we have incurred something wrong and want to roll back to the email_savepoint;

```
ROLLBACK TO SAVEPOINT email_savepoint;
```

The customer's email will return to its original value since any modifications made after the email_savepoint will be undone.

## COMMIT ; --

If all goes OK, commit the transaction.

In this case, we may set a checkpoint after changing the customer's email thanks to the savepoint email_savepoint. We can go back to that savepoint and undo the email change if something goes wrong later in the transaction. If the transaction is properly committed, the modifications that were made before to the savepoint (in this example, changing the customer's email address) will be kept. Otherwise, all modifications will be undone in order to preserve data consistency.