# SQL language

SQL (Structured Query Language) is a standard programming language used to manage and manipulate data in a relational database management system (RDBMS). It provides a set of commands for storing, retrieving, updating, and deleting data, as well as for defining and controlling the structure of databases. SQL is both declarative and user-friendly, allowing users to focus on what data they need rather than how to get it.

Creating SQL Table:

## *1. CREATE TABLE*

SQL CREATE TABLE statement is used to create table in a database. If you want to create a table, you should name the table and define its column and each column's data type. Let's see the simple syntax to create the table.

**NULL**: The term NULL in SQL is used to specify that a data value does not exist in the database. It is not the same as an empty string or a value of zero, and it signifies the absence of a value or the unknown value of a data field. Some common reasons why a value may be NULL −

The value may not be provided during the data entry.

The value is not yet known.

**NOT NULL**: Here, NOT NULL signifies that column should always accept an explicit value of the given data type.

**AUTO INCREMENT:** Auto-increment is a concept in SQL that automatically generates the unique number in the field when the new row is entered into the table.

This feature is generally used for the Primary Key field, where we need to create a unique value for every record.

**PRIMARY KEY**

```
create table "tablename" (
"column1" "data type",
"column2" "data type",
"column3" "data type",
    ...
"columnN" "data type“
);
```

Example:

```
CREATE TABLE STUDENTS (
ID INT AUTO_INCREMENT PRIMARY KEY,
NAME VARCHAR (20) ,
AGE INT,
ADDRESS CHAR (25)
);
```

**Composite primary key example**:

```
CREATE TABLE students  (
S_Id int NOT NULL,
LastName varchar (255) NOT NULL,
FirstName varchar (255),
Address varchar (255),
City varchar (255),
PRIMARY KEY (S_Id, LastName)
) ;
```

**Foreign key declaration in table:**

Person:

| S_Id | FirstName | LastName | CITY |
|---|---|---|---|
| 1 | Molla | Omar | Kandahar |
| 2 | Osama | Muhammd | Kabul |
| 3 | Abdullah | Azzam | Khaibar |

Order:

| O_Id | OrderNo | S_Id |
|---|---|---|
| 1 | 99586465 | 2 |
| 2 | 78466588 | 2 |

```
CREATE TABLE Orders (
O_Id int NOT NULL,
Order_No  int NOT NULL,
S_Id int,
PRIMAY KEY (O_Id),
FOREIGN KEY (S_Id) REFERENCES Persons
 (S_Id)   );
```

## *2. INSERT INTO TABLE*

The INSERT INTO statement is used to add new records (rows) into a table in a database. It allows you to insert data into all columns or only specific columns of a table

- The number of values must match the number of specified columns.
- Data types of inserted values must match the column definitions.
- You can insert multiple rows at once using commas between value sets.

If the inserted data does not satisfy any of the attributes, the INSERT INTO statement displays an error. There are two ways to insert data in a table:

**i.By specifying column names:**

**Syntax:** INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);

Example: INSERT INTO Students (StudentID, Name) VALUES (102, 'Saad');

**ii. Without specifying column names**

Syntax: INSERT INTO table_name VALUES (value1, value2, value3, ...);

Example: INSERT INTO Students VALUES (101, 'Shalauddin', 'CSE', 22);

Example (multiple rows):

INSERT INTO Students (StudentID, Name, Department, Age) VALUES (103, 'Sinwar', 'EEE', 21), (104, 'Abu Ubaida', 'BBA', 23);

***3. DROP TABLE*****:** The DROP TABLE statement is used to permanently delete an entire table from the database, including all of its data, structure, and associated constraints (like primary keys, foreign keys, and indexes). Once a table is dropped, the data cannot be recovered unless a backup exists.

Syntax: DROP TABLE table_name;

Example: DROP TABLE Students;

### *4. TRUNCATE TABLE*

The TRUNCATE TABLE statement is used to delete all rows from a table without removing the table structure itself. It is a Data Definition Language (DDL) command that quickly removes all data while keeping the table ready for future use.

Syntax: TRUNCATE TABLE table_name;

Example: TRUNCATE TABLE Students;

### *5. DELETE TABLE*

The DELETE statement is used to remove specific rows from a table based on a given condition. Unlike DROP TABLE or TRUNCATE TABLE, it does not remove the table structure — only the selected data. The DELETE statement is used to delete rows from a table. If you want to remove a specific row from a table you should use WHERE condition.

Syntax: DELETE FROM table_name WHERE condition;

Example: DELETE FROM Students WHERE StudentID = 101;

But if you do not specify the WHERE condition it will remove all the rows from the table.

Example: DELETE FROM Students;

Differentiate DELETE and TRUNCATE and DROP statements

### *6. ALTER TABLE*

The ALTER TABLE statement is used to modify the structure of an existing table without deleting it. It allows you to add, delete, or modify columns, as well as add or drop constraints like primary keys or foreign keys.

Here we use the ALTER TABLE statement for,

**i. RENAME Table**

Syntax: ALTER TABLE old_table_name RENAME TO new_table_name;

Example: ALTER TABLE students RENAME TO person;

**ii. RENAME column**

Syntax: ALTER TABLE table_name RENAME COLUMN old_name TO new_name;

Example: ALTER TABLE Students RENAME COLUMN Age TO StudentAge;

**iii. ADD column**

Syntax: ALTER TABLE table_name ADD column_name data_type;

Example: ALTER TABLE Students ADD Email VARCHAR(100);

**iv. MODIFY Column**

Syntax: ALTER TABLE table_name MODIFY column_name new_data_type;

Example: ALTER TABLE Students MODIFY Name VARCHAR(150);

**v. DROP Column**

Syntax: ALTER TABLE table_name DROP COLUMN column_name;

Example: ALTER TABLE Students DROP COLUMN Email;

### *7. UPDATE TABLE*

The UPDATE statement is used to **modify or change existing records** in a table. It allows you to update one or more columns for one or multiple rows based on a specified condition.

Syntax: UPDATE table_name SET column1 = value1, column2 = value2, … WHERE condition;

Example:UPDATE Students SET Age = 23, Department = 'CSE' WHERE StudentID = 102;

## *8. SQL SELECT*

The SELECT statement is used to retrieve data from one or more tables in a database. The SELECT statement allows you to choose specific columns, filter rows, sort results, and even perform calculations.

Syntax: SELECT * FROM Table_name [to access all records of all fields]

Syntax: SELECT Column_Name_1, Column_Name_2, ….., Column_Name_ FROM Table_name

[to access all records of one or more fields]

Example: SELECT Name, Department FROM Students;

The SQL **DISTINCT** command is used with SELECT key word to retrieve only distinct or unique data

Syntax: SELECT DISTINCT column_name ,column_name FROM table_name;

Example: SELECT DISTINCT Department FROM Students;

The SQL **FIRST()** function is used to return the first value of the selected column.

Syntax: SELECT FIRST(column_name) FROM table_name;

The **LAST()** function in Structured Query Language shows the last value from the specified column of the table.

Syntax: SELECT LAST (column_name) FROM Table_Name ;

### SQL SELECT WHERE Clause

The WHERE clause in SQL is used to filter records in a table based on specific conditions. It is applied with commands like SELECT, UPDATE, and DELETE to control which rows are affected. Only the rows that satisfy the condition in the WHERE clause are returned or modified.

Syntax: SELECT *column1*, *column2, ...* FROM *table_name* WHERE *condition*;

### Common Operators Used with WHERE:

**Comparison Operators**: =, < >(!=), >, <, >=, <=

**Logical Operators**: AND, OR, NOT

**Range**: BETWEEN

**List**: IN

**Pattern Matching**: LIKE

Null Check: IS NULL, IS NOT NULL

**SQL:** SELECT * FROM Customers WHERE Country='kabul';

SELECT * FROM Customers WHERE CustomerID=1;

SELECT * FROM Customers WHERE CustomerID > 80;

SELECT * FROM Products WHERE Price < > 18;

SELECT * FROM Products WHERE Price BETWEEN 50 AND 60;

SELECT * FROM Students WHERE Age BETWEEN 18 AND 22;

SELECT * FROM Customers WHERE City IN ('Helmand', 'Herat');

SELECT * FROM Students WHERE Department IN ('CSE', 'EEE');

The SQL **LIKE Operator**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

- The percent sign % represents zero, one, or multiple characters
- The underscore sign _ represents one, single character

**Starts With:** To return records that starts with a specific letter or phrase, add the % at the end of the letter or phrase.

1) SELECT * FROM Customers WHERE CustomerName LIKE 'La%';

2) SELECT * FROM CustomersWHERE CustomerName LIKE 'a%' OR CustomerName LIKE 'b%';

**Ends With:** To return records that ends with a specific letter or phrase, add the % at the beginning of the letter or phrase.

**SQL:** SELECT * FROM Customers WHERE CustomerName LIKE '%a';

The **AND operator** is used to filter records based on more than one condition, like if you want to return all customers from Spain that start with the letter 'G'.

**Syntax:**SELECT column1, column2,...FROM table_nameWHERE condition1 AND condition2 AN D condition3 ...;

**SQL:**

(1) SELECT * FROM Students WHERE Department = 'CSE' AND Age > 21;

(2) SELECT * FROM Customers WHERE Country= 'Spain' AND CustomerName LIKE 'G%';

(3) SELECT *FROM Customersn WHERE Country= 'Afgan' AND City = 'Panjshir' AND CustomerID > 50;

(4)SELECT * FROM CustomersWHERE Country= 'Andalusia' AND (CustomerName LIKE 'G%' O R CustomerName LIKE 'R%');

The **OR operator** is used to filter records based on more than one condition, like if you want to return all customers from Mali but also those from Somalia:

**Syntax:**SELECT *column1*, *column2,...*FROM *table_name*WHERE *condition1* OR *condition2* OR *co ndition3 ...*;

**SQL:** SELECT * FROM Customers WHERE Country = 'Mali' OR Country = 'Somalia';

The **NOT operator** is used in combination with other operators to give the opposite result, also called the negative result.

**Syntax:** SELECT *column1*, *column2, ...*FROM *table_name*WHERE NOT *condition*;

**SQL:** **1.** SELECT * FROM Customers WHERE NOT Country = 'Andalusia';

(Select customers that do not start with the letter 'A')

**2.** SELECT * FROM Customers WHERE CustomerName NOT LIKE 'A%';

**3.** SELECT * FROM Customers WHERE CustomerID NOT BETWEEN 10 AND 60;

**4.** SELECT * FROM Customers WHERE City NOT IN ('Panjshir', 'Kunduz');

**5.**SELECT * FROM Customers WHERE NOT CustomerId < 50;

**Syntax:** UPDATE *table_name* SET *column1* = *value1*, *column2* = *value2*, ...WHERE *condition*;

**SQL:** UPDATE Customers SET Name= 'Osama', City= 'kabul' WHERE CustomerID = 1;

# Aggregate Function

Aggregate functions in SQL are special functions that perform calculations on a set of values and return a single summarized result. They are commonly used with the SELECT statement, often combined with GROUP BY to summarize data across groups.

**COUNT( )** — Returns the total number of rows in a set.

**SUM( )** — Calculates the sum of values in a numeric column.

**AVG( )** — Computes the average (mean) value of a numeric column.

**MIN( )** — Finds the smallest value in a column.

**MAX( )** — Finds the largest value in a column.

**Example:**

| ID | NAME | DEPT | MARKS |
|---|---|---|---|
| 1 | Oomar | CSE | 85 |
| 2 | Usama | EEE | 78 |
| 3 | Khalid | BBA | 92 |
| 4 | Nusaiba | CSE | 74 |
| 5 | Shalauddin | ENG | 88 |

1) SELECT COUNT(*) FROM Students;     Result: 5

1.1) SELECT COUNT(*) AS Total_Students FROM Students;

| Total_Students |
|---|
| 5 |

1.2) Find the number of students where marks is higher than 80:

SELECT COUNT(ID) FROM students WHERE marks > 80;

1.3) Count unique number of department.

SELECT COUNT(DISTICNT ID) FROM students;

2) SELECT SUM(Marks) FROM Students;     Result: 471

3) SELECT AVG(Marks) FROM Students;     Result: 83.4

4) SELECT MIN(Marks) FROM Students;     Result: 74

5) SELECT MAX(Marks) FROM Students;     Result: 92

**GROUP BY condition**

The GROUP BY clause arrange identical data into groups, allowing aggregate functions to be applied to each group separately. It is typically used when you want to summarize data based on one or more columns. After grouping, each group represents a unique combination of the specified column values. It is especially helpful for generating reports, such as calculating total sales per customer, average marks per subject, or counting employees in each department.

**HAVING condition**

The HAVING clause in SQL is used to filter groups created by the GROUP BY clause based on a specified condition. While the WHERE clause filters individual rows before grouping, the HAVING clause filters the groups after aggregate functions have been applied.

**Example-1:**

| ID | PRODUCT | CATEGORY | QUANTITY | PRICE |
|---|---|---|---|---|
| 1 | Pen | Stationery | 10 | 5 |
| 2 | Pen | Stationery | 20 | 5 |
| 3 | Book | Books | 5 | 100 |
| 4 | Book | Books | 8 | 100 |
| 5 | Bag | Accessories | 3 | 500 |

**Question:** Find total quantity sold for each product.

**SQL:** SELECT Product, SUM(Quantity) AS TotalQty FROM SALES GROUP BY Product;

Result outcome:

| PRODUCT | TotalQty |
|---|---|
| Pen | 30 |
| Book | 13 |
| Bag | 3 |

**Question:** Find average price of each category (AVG).

SQL: SELECT Category, AVG(Price) AS AvgPrice FROM SALES GROUP BY Category;

Result outcome:

| CATEGORY | AvgPrice |
|---|---|
| Stationery | 5 |
| Books | 100 |
| Accessories | 500 |

**Question:** Find maximum quantity sold for each product (MAX).

SQL: SELECT Product, MAX(Quantity) AS MaxQty FROM SALES GROUP BY Product;

Result outcome:

| PRODUCT | MaxQty |
|---|---|
| Pen | 20 |
| Book | 8 |
| Bag | 3 |

**Question:** Count number of entries for each category (COUNT).

SQL: SELECT Category, COUNT(*) AS TotalSales FROM SALES GROUP BY Category;

Result outcome:

| CATEGORY | TotalSales |
|---|---|
| Stationery | 2 |
| Books | 2 |
| Accessories | 1 |

**Question:** Show products whose total quantity sold is more than 10.

SQL: SELECT Product, SUM(Quantity) AS TotalQty FROM SALES GROUP BY Product HAVING SUM(Quantity) > 10;

Result outcome:

| PRODUCT | TotalQty |
|---|---|
| Pen | 30 |
| Book | 13 |

**Question:** Find total quantity of products from 'Books' category only.

SQL: SELECT Product, SUM(Quantity) AS TotalQty FROM SALES WHERE Category = 'Books' GROUP BY Product;

Result outcome:

| PRODUCT | TotalQty |
|---|---|
| Book | 13 |

**Question:** Show categories whose total sales quantity is more than 10.

SQL: SELECT Category, SUM(Quantity) AS TotalQty FROM SALES GROUP BY Category HAVING SUM(Quantity) > 10;

Result outcome:

| Category | TotalQty |
|---|---|
| Stationery | 30 |
| Book | 13 |

**Question:** Show products whose MAX quantity sold is greater than 10.

SQL: SELECT Product, MAX(Quantity) AS MaxQty

FROM SALES GROUP BY Product HAVING MAX(Quantity) > 10;

Result outcome: Pen

**Question:** Show products priced below 100 whose total quantity sold is greater than 10.

SQL: SELECT Product, SUM(Quantity) AS TotalQty FROM SALES WHERE Price < 100

GROUP BY Product HAVING SUM(Quantity) > 10;

Result outcome:

| Category | TotalQty |
|---|---|
| Pen | 30 |

**Example-2**

| ID | Product | Category | Quantity | Price |
|---|---|---|---|---|
| 1 | Laptop | Electronics | 2 | 55000 |
| 2 | Mouse | Electronics | 5 | 800 |
| 3 | Chair | Furniture | 3 | 2500 |
| 4 | Table | Furniture | 1 | 7000 |
| 5 | Keyboard | Electronics | 4 | 1500 |

**Questions:**

1) Count the number of items in each category where quantity is greater than 1, and show only categories having at least 2 such items.

2) Calculate total price of items priced above 1000, grouped by category, but show only those categories where the sum exceeds 2000.

3) Find the average quantity for categories where quantity is greater than 1, but show only categories whose average is at least 3.

4) Find the minimum price in each category for items costing more than 500, but show only categories whose min price is below 2000.

5) Find the maximum price per category for all items with quantity ≥ 1, but show only categories where max price exceeds 5000.