# SQL

- SQL is a standard language for storing, manipulating and retrieving data in a database.
- Structured Query Language.
- Not case sensitive.
- **SEQUEL :** Structured English Query Language. [Another name of SQL]

- **What can SQL do ? :**
  **[CRUD** operations **:** Create**,** Read**,** Update**,** Delete**]**
  - Execute queries against the database.
  - Retrieve data from the database.
  - Can **insert, update, delete** records in a database.
  - Can create a new database.
  - Can create stored procedures in a database.
  - Can create views in a database.
  - Can set permission on tables, procedures, and views.

- **No Relational DBMS :** Data which are **not stored in the form of tables.**
  **Eg :** MongoDB

- **RDBMS : Relational Database Management System.**
  - It is the basis of SQL and all of the modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, & Microsoft Access.
  - Data in RDBMS is stored in database objects called **tables.**
  - A **table** is a collection of related entries and it consists of **rows and columns.**
  - Every **table** is broken into smaller entities called **fields.**
  - A **field** is a **column** in a **table** that is designed to maintain specific information about every **record** in the table.
    **Field** is a **Vertical entity.**
    **Columns** give the **structure/schema/design** of the database.
  - A **record,** also called **row**, is each individual entry that exists in a table.
    **Record** is a **Horizontal entity.**

★ **Semicolon** is used at the end of the query. Which will allow more than one SQL command to be executed in the same call to the server.

Types of SQL Commands:

1. **DQL** *(Data Query Language)* : Used to retrieve data from databases. (SELECT)

2. **DDL** *(Data Definition Language)* : Used to create, alter, and delete database objects like tables, indexes, etc. (CREATE, DROP, ALTER, RENAME, TRUNCATE)

3. **DML** *(Data Manipulation Language)*: Used to modify the database. (INSERT, UPDATE, DELETE)

4. **DCL** *(Data Control Language)*: Used to grant & revoke permissions. (GRANT, REVOKE)

5. **TCL** *(Transaction Control Language)*: Used to manage transactions. (COMMIT, ROLLBACK, START TRANSACTIONS, SAVEPOINT)

**SQL STATEMENTS :**

1. **CREATE DATABASE <**db_name**> :** Used to create a new database
2. **DROP DATABASE <**db_name**> :** Used to drop the database which is present in the server
3. **ALTER DATABASE :** Modifies a database
4. **USE <**db_name**> :** Used to change the current working database.
5. **SELECT :** extract data from the database
6. **WHERE :** used to filter the records
7. **UPDATE :** updates the data from the database
8. **DELETE :** deletes the data from the database
9. **INSERT INTO :** insert a new data into the database
10. **CREATE TABLE <**table_name**> :** Creates a new table inside a database
11. **ALTER TABLE <**table_name**> :** Modifies the table
12. **DROP TABLE <**table_name**> :** drops the table from the database
13. **CREATE INDEX :** creates an **index key [**search key**]**
14. **DROP INDEX :** deletes an index
15. **ORDER BY :** used to sort the result-set values in ascending or descending order.

**SELECT :**

- SELECT statement is used to select the data from the database.
- Syntax :

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

  **column1**, **column2**, …… are the **fields** from the table **table_name**

- If you want to select all the columns from the table without typing individual column name, use **asterisk (*)** along with SELECT
  **SELECT * FROM** table_name**;**
- **DISTINCT** is used to along with the SELECT to get the distinct values from the columns
  - No repetition is considered.
  - Syntax :

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

- **COUNT** is used to get the total number of records

**WHERE : [**Conditions on **ROWS]**

- This is used to filter the records in the database.
- Used to extract only those records which fulfill the specific condition.
- Syntax :

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- While specifying the conditions,
  **Text** values should be enclosed with the **quotes.**
  **Numerical** values should not be enclosed with the **quotes.**

- **Operators :**

| Operator | Description |
|---|---|
| = | Equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != |
| BETWEEN | Between a certain range |
| LIKE | Search for a pattern |
| IN | To specify multiple possible values for a column |

**ORDER BY :**

- This is used to sort the result set in ascending or descending order.
- Syntax :

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

**ASC -> Ascending order {Default}**
**DESC -> Descending order**

- For **string** values, it orders **alphabetically as default**

- ORDER BY several column :

Example

```
SELECT * FROM Customers
ORDER BY Country, CustomerName;
```

  - This means it orders by **Country**, but if some records have the same country name then it orders by **CustomerName**
- Using both ASC and DESC :
  - Following SQL statement selects all records from the **Customers** table, sorted ascending by **Country** and descending by **CustomerName** column.

Example

```
SELECT * FROM Customers
ORDER BY Country ASC, CustomerName DESC;
```

**AND or OR operator:**

- Used when there are two or more conditions to pass.
- Syntax :

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

The `AND` operator displays a record if *all* the conditions are TRUE.

The `OR` operator displays a record if *any* of the conditions are TRUE.

**NOT operator :**

- It is used to get the **opposite result** of the condition specified.
- Also called as **negative result**.
- Used along with other operators and keywords.
- Syntax :

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

**INSERT INTO :**

- Is used to insert new records into the table.
- Two ways of using INSERT INTO statement :
    1. Specify both the column names and values to be inserted

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

    2. Maintaining the order of the columns while specifying the values which need to be inserted into the table.

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

- Inserting multiple values :

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES
('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway'),
('Greasy Burger', 'Per Olsen', 'Gateveien 15', 'Sandnes', '4306', 'Norway'),
('Tasty Tee', 'Finn Egan', 'Streetroad 19B', 'Liverpool', 'L1 0AA', 'UK');
```

Each record should be separated with a **comma(,).**

**NULL Value :**

- A field with **NULL** value is a field with **NO VALUE.**

**Note:** A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

- **IS NULL** or **IS NOT NULL** :

These keywords is used to test for the NULL values

IS NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

IS NOT NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

- IS NULL : use it for finding the empty values
- IS NOT NULL : use it for finding the non-empty values

**UPDATE : [**To update the records**]**

- Used to modify the existing records in a table.
- Syntax :

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

- WHERE clause determines how many records will be updated.
- Be careful while updating. If you omit the WHERE clause then the entire table gets updated.

**DELETE :**

- Used to delete existing records in the table.
- Syntax :

```
DELETE FROM table_name WHERE condition;
```

- WHERE clause determines which record to be deleted.
- If you omit the WHERE clause, then all the records will be deleted .
- If you delete all the records from the table, this **doesn't delete the table**, instead it keeps the table structure, attributes, and indexes.

```
DELETE FROM table_name;
```

- To delete the complete table, use **DROP TABLE <**table_name**>;**

**TOP / LIMIT / FETCH FIRST n ROWS ONLY :**

- used to specify the number of records from the top to return.
- Syntax :

**SQL Server / MS Access Syntax:**

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE condition;
```

**MySQL Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

**Oracle 12 Syntax:**

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s)
FETCH FIRST number ROWS ONLY;
```

- **PERCENT** is used to specify the percentage of records to return.

- **OFFSET** is used to set the starting number to count the LIMIT.

**AGGREGATE FUNCTIONS :**

1. **MIN() :** function returns the smallest value of the selected column
2. **MAX() :** function returns the largest value of the selected column

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

* When you use **MIN()** or **MAX()**, the returned column will be named **MIN(*field*)** or **MAX(*field*)** by default. To give the column a new name, use the **AS** keyword:

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

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

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

* WHERE clause is used to specify the condition.
* To ignore the duplicates use **DISTINCT** keyword
* To **ignore the NULL** values, specify the column names
* To **consider the NULL** values in the count then use **asterisk (*)**
* Give the counted column a name by using the **AS** keyword.

4. **SUM() :** function returns the **total sum** of the **numeric column**

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

* WHERE clause is used to specify the condition.
* Give the summarized column a name by using the **AS** keyword.
* Parameter inside the SUM() function can also be an expression.

5. **AVG() :** function returns the **average** value of the **numeric column.**

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

* NULL values are ignored
* WHERE clause to specify the conditions
* Give the average column a name by using the **AS** keyword

**LIKE operator :**

- LIKE operator is used in WHERE clause to search for a specified pattern in a column.
- Two **wildcards** often used in conjunction with LIKE operator :
    1. Percent sign **%** represents **zero, one or multiple** characters.
    2. Underscore **_** represents a **single** character.

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

- Without wildcards then it has to be an **exact match** to return a result.

**WILDCARDS :**

- Wildcard character is used to substitute one or more characters in a string.
- Wildcard characters are used with LIKE operator. LIKE operator is used in a WHERE clause to search for  a specified pattern.

| Symbol | Description |
|---|---|
| % | Represents zero or more characters |
| _ | Represents a single character |
| [] | Represents any single character within the brackets * |
| ^ | Represents any character not in the brackets * |
| - | Represents any single character within the specified range * |
| {} | Represents any escaped character ** |

* Not supported in PostgreSQL and MySQL databases.

** Supported only in Oracle databases.

**IN operator :**

- This operator allows specifying multiple values in a WHERE clause.
- This is shorthand for OR conditions.
- Syntax :

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

- By using **NOT** in front of **IN**, it returns all the records that are NOT any of the values in the list.

**BETWEEN operator :**

- BETWEEN operator selects the value within a given range.
- The values might be **numbers, texts or dates**.
- Syntax :

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

- To return the records outside the specified range, use NOT in front of BETWEEN.
- When the range is given in string type, then the record will be selected alphabetically.

**AS :**

- This keyword is used to create **alias.**
- Used to give temporary names for the column, table.
- It's optional.
- An alias exists only for the duration of that query.
- Syntax :

When alias is used on column:

```
SELECT column_name AS alias_name
FROM table_name;
```

When alias is used on table:

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

- If the alias is a string with spaces, then use quotes around them.
- Aliases can be useful when :
- There are more than one table involved in the query.
- Functions are used in the query.
- Column names are big or not readable.
- Two or more columns are combined together.

**JOINS :**

- JOIN clause is used to combine the rows from two or more tables, based on the related column between them.
- Different types of JOINS :

1. **INNER JOIN :** returns records that have matching values in both tables.

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

* **JOIN** or **INNER JOIN** is default

2. **LEFT (OUTER) JOIN :** returns all records from the left table, and the matched records from the right table

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

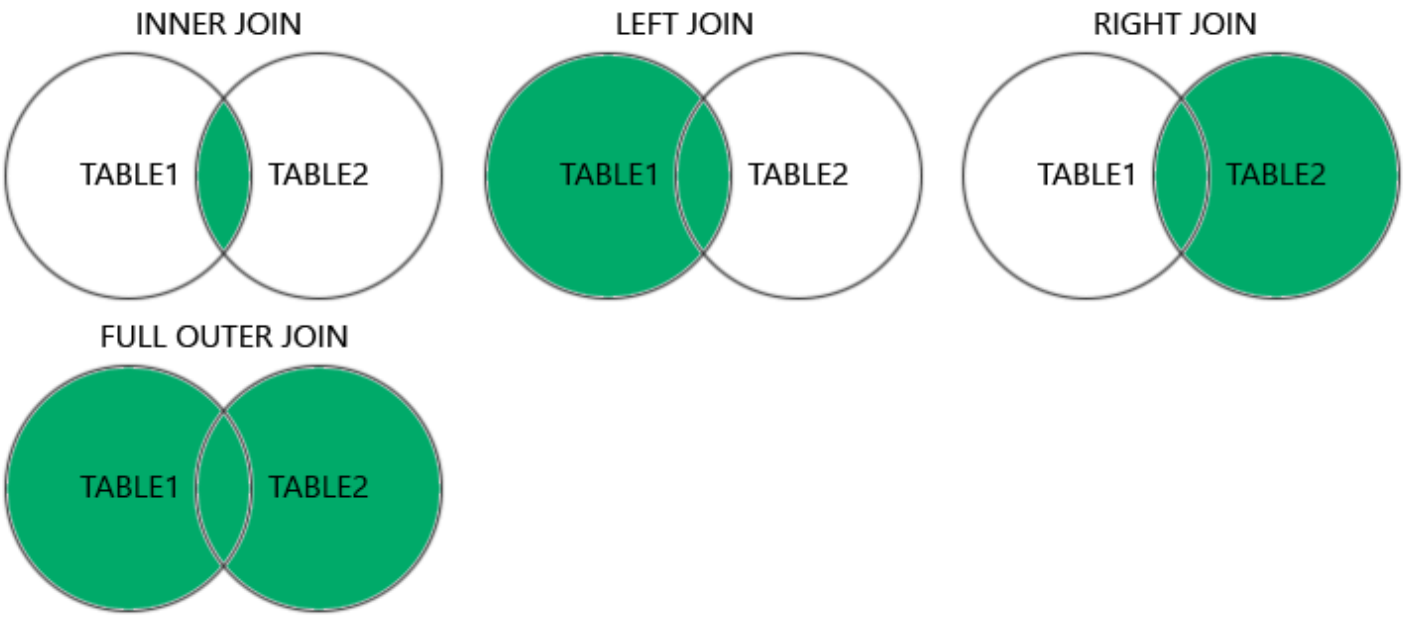
3. **RIGHT (OUTER) JOIN :** returns all records from the right table, and the matched records from the left table.

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

4. **FULL (OUTER) JOIN :** returns all the records when there is match in either left or right table

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```

* **FULL JOIN** or **FULL OUTER JOIN** or **CROSS JOIN** both are same
* If there are some records which have **no match** with the other table, then the values for the column would be displayed as **NULL.**



**UNION operator :**

- UNION operator is used to combine the result sets of two or more SELECT statements.
- **Conditions for UNION :**
- Every SELECT statement within UNION must have the **same number of columns**.
- Columns must also have **similar data types**.
- Columns in every SELECT statement must also be in the **same order**.
- Syntax :

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

- UNION operator selects only distinct values by default.
- In order to allow duplicate values, use **ALL** after the **UNION** keyword.

```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

**GROUP BY :**

- This statement groups rows that have same values into summary rows, like “find number of customers in each country”
- It is often used with aggregate functions to group the result set by one - more columns.
- Syntax :

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

**HAVING : [**Conditions on **COLUMNS]**

- HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
- Syntax :

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

**EXISTS operator :**

- Used to test for the existence of any record in a sub-query.
- EXISTS operator returns TRUE, if the subquery returns one or more records.
- Syntax :

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

**ANY and ALL operator :**

- These two operators allow you to perform a comparison between a single column value and a range of other values.
- **ANY :** condition will be true if the operation is true for any of the values in the range.
- Returns a boolean value as a result.
- Returns TRUE if ANY of the subquery meets the condition.

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

- **ALL :** condition will be true only if the operation is true for all values in the range.
- Returns a boolean value as a result.
- Returns TRUE if ALL of the subquery values meet the condition.
- Used with **SELECT, WHERE, HAVING**

ALL Syntax With SELECT

```
SELECT ALL column_name(s)
FROM table_name
WHERE condition;
```

ALL Syntax With WHERE or HAVING

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

**SELECT INTO:**

Copy all columns into a new table:

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

Copy only some columns into a new table:

```
SELECT column1, column2, column3, ...
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

- The new table will be created with the column names and types defined in the older table.
- You can create new column names using the **AS** clause.
- **[IN <**db_name**>]** specifies to create the new table in another database.
- **SELECT INTO** can also be used to create a new, empty table using the **schema of another**. Just add a **WHERE** clause that causes the query to return no data:

```
SELECT * INTO newtable
FROM oldtable
WHERE 1 = 0;
```

**INSERT INTO SELECT :**

- This copies data from one table and inserts into the other table.
- This statement requires that the data types in source and target tables match.
- Existing records in the target table won't be affected.
- Syntax :

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;
```

**CASE expression :**

- CASE expression goes through conditions and returns the values when the first condition is MET.
- When the condition is MET, then it stops reading and returns the result.
- If no condition is MET, then it returns the value of ELSE clause.
- If there is no ELSE clause and no conditions MET, then it returns NULL value.
- Syntax :

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END;
```

**NULL FUNCTIONS :**

1. **IFNULL() : MySQL** has this function to return an alternate value if the specified column has a NULL value in it.

```
SELECT ProductName, UnitPrice * (UnitsInStock + IFNULL(UnitsOnOrder, 0))
FROM Products;
```

2. **ISNULL() : SQL Server and MS Access** has this function to the same job

The SQL Server `ISNULL()` function lets you return an alternative value when an expression is NULL:

```
SELECT ProductName, UnitPrice * (UnitsInStock + ISNULL(UnitsOnOrder, 0))
FROM Products;
```

3. **COALESCE() :**

or we can use the `COALESCE()` function, like this:

```
SELECT ProductName, UnitPrice * (UnitsInStock + COALESCE(UnitsOnOrder, 0))
FROM Products;
```

4. **NVL() :**

The Oracle `NVL()` function achieves the same result:

```
SELECT ProductName, UnitPrice * (UnitsInStock + NVL(UnitsOnOrder, 0))
FROM Products;
```

**STORED Procedures :**

- Stored procedure is a **prepared SQL code** that you can save, so the code can be **reused** over and over again.
- You can also pass parameters to a stored procedure, so that the procedure can act based on the parameter value(s) that is passed.
- Syntax :

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

- To execute a stored procedure :

```
EXEC procedure_name;
```

- To have a parameter, use the user defined name for the parameter and **nvarchar(<**value**>)** for passing it to the parameter.
  **<value>** means it can take upto that value number of characters.

Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
GO;
```

**Comments :**

- **Single** line comments start with **–** till the end of the line.
- **Multi** line comments starts with **/*** and ends with ***/.**

**OPERATORS :**

## SQL Arithmetic Operators

| Operator | Description |
|---|---|
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |
| % | Modulo |

## SQL Bitwise Operators

| Operator | Description |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise exclusive OR |

## SQL Comparison Operators

| Operator | Description |
|---|---|
| = | Equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| <> | Not equal to |

## SQL Compound Operators

| Operator | Description |
|---|---|
| += | Add equals |
| -= | Subtract equals |
| *= | Multiply equals |
| /= | Divide equals |
| %= | Modulo equals |
| &= | Bitwise AND equals |
| ^-= | Bitwise exclusive equals |
| \|*= | Bitwise OR equals |

## SQL Logical Operators

| Operator | Description |
|---|---|
| ALL | TRUE if all of the subquery values meet the condition |
| AND | TRUE if all the conditions separated by AND is TRUE |
| ANY | TRUE if any of the subquery values meet the condition |
| BETWEEN | TRUE if the operand is within the range of comparisons |
| EXISTS | TRUE if the subquery returns one or more records |
| IN | TRUE if the operand is equal to one of a list of expressions |
| LIKE | TRUE if the operand matches a pattern |
| NOT | Displays a record if the condition(s) is NOT TRUE |
| OR | TRUE if any of the conditions separated by OR is TRUE |
| SOME | TRUE if any of the subquery values meet the condition |

**SQL DATABASE :**

**CREATE DATABASE :**

- Used to create new database
- Syntax :

```
CREATE DATABASE databasename;
```

- **SHOW DATABASES** is used to display all the databases present in the server.

**DROP DATABASE :**

- Used to drop an existing database.
- Syntax :

```
DROP DATABASE databasename;
```

**BACKUP DATABASE :**

- Used to create a full backup of an existing database.
- Syntax :

```
BACKUP DATABASE databasename
TO DISK = 'filepath';
```

- A **differential backup** only backs up the **parts of the database that have changed** since the last full database backup.

```
BACKUP DATABASE databasename
TO DISK = 'filepath'
WITH DIFFERENTIAL;
```

**CREATE TABLE :**

- Used to create a new table in the database.
- Syntax :

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

- **column** parameter specifies the **name of the column** of the table.
  **datatype** parameter specifies the **type of data** the column can hold.(eg., **varchar, integer, date…….**)

- Creating new table using the old existing table:

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE ....;
```

* New table gets the same column definitions. All columns or specific columns can be selected.
* If you create a new table using an existing table, then the new table will be filled with the records from the old table.

**DROP TABLE :**

- Used to drop an existing table from the database.
- Syntax :

```
DROP TABLE table_name;
```

- **TRUNCATE TABLE :** used to delete the data inside the table , but not the table itself

```
TRUNCATE TABLE table_name;
```

**ALTER TABLE : [**To ALTER the schema of the table**]**

- Fields, Data type, Constraint.
- Used to add, delete or modify the columns in an existing table.
- Used to add or drop various constraints on an existing table.
- To add column :

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

- To drop column :

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

- To rename column :

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

- To **change the datatype :**

**SQL Server / MS Access:**

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype;
```

**My SQL / Oracle (prior version 10G):**

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

**SQL Constraints :**

- Used to specify the **rules for data** in a table.
- It can be specified while the table is created using **CREATE TABLE** or after the table is created with **ALTER TABLE** statement.
- Syntax :

```
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 the table. This ensures the accuracy and reliability of the data in the table. If there is any **violation** between data action and the constraints, the **action is aborted**.
- Constraints can be applied on **Column level** or **Table level**.
  Column level constraints apply to the **column**, and
  Table level constraints apply to the **whole table**.

- `NOT NULL` - Ensures that a column cannot have a NULL value
- `UNIQUE` - Ensures that all values in a column are different
- `PRIMARY KEY` - A combination of a `NOT NULL` and `UNIQUE`. Uniquely identifies each row in a table
- `FOREIGN KEY` - Prevents actions that would destroy links between tables
- `CHECK` - Ensures that the values in a column satisfies a specific condition
- `DEFAULT` - Sets a default value for a column if no value is specified
- `CREATE INDEX` - Used to create and retrieve data from the database very quickly

1. **NOT NULL :**
   - By default, a column can hold NULL values.
   - **NOT NULL** constraint enforces the column to not to accept the NULL values.

2. **UNIQUE :**
   - This constraint ensures that all the values in the column are different.
   - **UNIQUE** and **PRIMARY KEY** both constraints provide the uniqueness for a column or a set of columns.
   - **PRIMARY KEY** automatically has a **UNIQUE** constraint.
   - Can have many UNIQUE constraints for the table.

3. **PRIMARY KEY :**
   - This constraint uniquely identifies each record in a table.
   - It is a combination of **UNIQUE** and **NOT NULL.**
   - A table can have only **one** PRIMARY KEY.

```
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).

4. **FOREIGN KEY :**
   - This constraint is used to **prevent actions that would destroy the links between tables**
   - **FOREIGN KEY** is a field (or collection of fields) in one table, that refers to the **PRIMARY KEY** in another table.
   - Table with FOREIGN KEY is called a **child table**, and the table with PRIMARY KEY is called a **referenced / parent table.**
   - The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

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

   - **CASCADE :**
     - **ON UPDATE CASCADE:** When we create a FOREIGN KEY using this option, the **referencing rows are updated in the child table** when the **referenced row is updated in the parent table** which has a primary key.
     - **ON DELETE CASCADE:** When we create a FOREIGN KEY using this option, it deletes the referencing rows in the child table when the referenced row is deleted in the parent table which has a primary key.

```
CREATE TABLE student (
  id INT PRIMARY KEY,
  courseID INT,
  FOREIGN KEY(courseID) REFERENCES course(id)
  ON DELETE CASCADE
  ON UPDATE CASCADE
);
```

5. **CHECK :**
   - This constraint is used to **limit the value range** that can be placed in a column, i.e., it only allows **only certain values** for the 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)
);
```

6. **DEFAULT :**
   - This constraint is used to set a default value for a column.
   - The specified 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 **GETDATA()**

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

7. **INDEX :**
   - **CREATE INDEX** is used to create indexes in tables.
   - Indexes are used to retrieve the data from the database more quickly than otherwise.
   - Syntax for allowing Duplicate values:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

   - Syntax for not allowing Duplicate values[UNIQUE] :

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

8. **AUTO INCREMENT :**
    - This allows a unique number to be generated automatically when a new record is inserted into a table.
    - Often this is a PRIMARY KEY field that we would like to be created automatically every time a new record is inserted into the table.

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

    - By default the AUTO_INCREMENT starting value is 1.
    - To let the AUTO_INCREMENT start from different value :

```
ALTER TABLE Persons AUTO_INCREMENT=100;
```

9. **DATES :**

- `DATE` - format YYYY-MM-DD
- `DATETIME` - format: YYYY-MM-DD HH:MI:SS
- `TIMESTAMP` - format: YYYY-MM-DD HH:MI:SS
- `YEAR` - format YYYY or YY

**SQL VIEWS :**

- View is a **Virtual Table** based on the result set of an SQL statement.
- Views contain rows and columns just like a real table. The fields in a view are fields from one or more real tables in the database.
- Syntax :

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- A view always shows up–to-date data. The database engine recreates the view, every time a user queries it.
- Updating the VIEW using **CREATE OR REPLACE VIEW** :

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- **DROP VIEW <**view_name**> :** used to **drop the view**

**SQL INJECTION :**

- Is a **code injection technique** which might destroy your database.
- One of the most common **web hacking** techniques.
- SQL Injection is the **placement of malicious code in SQL statements**, via web page input.
- So in order to protect the database from being hacked, use **SQL parameters** for protection.
- SQL parameters are values that are added to an **SQL query at execution time,** in a controlled manner.

**SQL Subqueries :**

- Sub-query / Inner query / Nested query
- A query within another query
- Involves two statements.
- Can be written on **WHERE, FROM, SELECT**

**STRING FUNCTIONS :**

**CONCAT :** Combine strings or columns

```
SELECT CONCAT('My', 'S', 'QL');
 -> 'MySQL'
SELECT CONCAT('My', NULL, 'QL');
 -> NULL
SELECT CONCAT(14.3);
 -> '14.3'
```

**CONCAT_WS :** Combine strings or columns **with separators.**

Separator specified inside the function will occur in between each of the columns or strings passed inside the function.

```
mysql> SELECT CONCAT_WS('!','hi','bye','lol');
+-----------------------------------+
| CONCAT_WS('!','hi','bye','lol') |
+-----------------------------------+
| hi!bye!lol                        |
+-----------------------------------+
```

**SUBSTRING / SUBSTR :** To extract a substring from the string or column of strings .

```
SUBSTRING(string, start, length)
```

- If the start and length are negative numbers then it'll start from the back

**OR:**

```
SUBSTRING(string FROM start FOR length)
```

**REPLACE :** replaces all occurrences of a substring within a string, with a new string.

```
REPLACE(string, from_string, new_string)
```

**String**……..original string / column of strings
**from_string**……..substring to be replaced
**new_string**……..string to replace with

- case sensitive

**REVERSE :** returns the reversed version of the string which is been passed

```
REVERSE(string)
```

**CHAR_LENGTH / CHARACTER_LENGTH :** returns the length of a string or **number of characters** in the string.

```
CHAR_LENGTH(string)
```

**LENGTH:** returns the length of the string **in bytes**.

```
LENGTH(string)
```

**UPPER / UCASE :** converts the string to UPPER case.

```
UPPER(text)
```

**LOWER / LCASE :** converts the string to LOWER case.

```
LOWER(text)
```

**INSERT :** inserts a string within a string at the specified position and for a certain number of characters.

```
INSERT(string, position, number, string2)
```

**string**……string that will be modified
**position**……..position to where the string needs to be added
**number**……number of characters to replace [if mentioned **0,** then nothing will be replaced instead the string will be added]
**string2**…….string to insert into the string

**LEFT :** extracts number of characters from string {from **LEFT**}

```
LEFT(string, number_of_chars)
```

**RIGHT :** extracts number of characters from string {from **RIGHT**}

```
RIGHT(string, number_of_chars)
```

**REPEAT :** repeat the string as many times as specified.

```
REPEAT(string, number)
```

**TRIM :** removes **LEADING** and **TRAILING** spaces from the string

```
TRIM(string)
```

**RTRIM()** removes the TRAILING spaces only and
**LTRIM()** removes the LEADING spaces only

For more **STRING** functions, visit the website.

**TIME & DATE data types :**

- **DATE() :** contains only date, no time.
  Format : **YYYY-MM-DD**
- **TIME() :** contains only time, no date.
  Format : **HH:MM:SS**
- **DATETIME() :** contains both date and time.
  Format : **YYYY-MM-DD HH:MM:SS**
- **CURDATE() / CURRENT_DATE() :** returns the **current date**
- **CURTIME() / CURRENT_TIME() :** returns the **current time**
- **CURRENT_TIMESTAMP() / NOW() :** returns the **current date and time**
- **DAY() :** returns the **day of the month for a date**
- **DAYOFWEEK():** returns the day of the week in the form of **index value**

**Note:** 1=Sunday, 2=Monday, 3=Tuesday, 4=Wednesday, 5=Thursday, 6=Friday, 7=Saturday.

- **DAYOFYEAR() :** returns the day of the year (1 to 365 days)
- **MONTH() :** returns the month number (1 to 12 months)
- **MONTHNAME() :** returns the name of the month supplied.
- **DATEDIFF() :** returns the difference of days between two dates provided

```
DATEDIFF(date1, date2)
```

- **DATE_FORMAT() :** function formats the date by a specified format.

```
DATE_FORMAT(date, format)
```

* format specifier for DATE_FORMAT()

- **TIME_FORMAT() :** function formats the time by a specified format.

```
TIME_FORMAT(time, format)
```

* format specifier for TIME_FORMAT()

- **DATE_ADD(), DATE_SUB() :** does the arithmetic operation of date.

```
DATE_ADD(date, INTERVAL value addunit)

DATE_SUB(date, INTERVAL value interval)
```

| | |
|---|---|
| *interval* | Required. The type of interval to subtract. Can be one of the following values:<br>• MICROSECOND<br>• SECOND<br>• MINUTE<br>• HOUR<br>• DAY<br>• WEEK<br>• MONTH<br>• QUARTER<br>• YEAR<br>• SECOND_MICROSECOND<br>• MINUTE_MICROSECOND<br>• MINUTE_SECOND<br>• HOUR_MICROSECOND<br>• HOUR_SECOND<br>• HOUR_MINUTE<br>• DAY_MICROSECOND<br>• DAY_SECOND<br>• DAY_MINUTE<br>• DAY_HOUR<br>• YEAR_MONTH |

- For more date and time functions, refer to the website.

**WINDOW Functions :**

- Performs **aggregate operations on group of rows**, but they produce result for **each row**
- **OVER() :** over clause is used with window functions to define that window

  OVER clause does two things :

  1. Partition rows into a form set of rows. [ **PARTITION BY** clause is used ]
  2. Orders rows within those partitions into a particular order. [ **ORDER BY** clause is used ]

  **NOTE :** If partitions aren't done, then ORDER BY orders all the rows of the table.

  When the OVER() clause is empty, it applies to all the rows of the table.

| Name | Description |
|---|---|
| CUME_DIST() | Cumulative distribution value |
| DENSE_RANK() | Rank of current row within its partition, without gaps |
| FIRST_VALUE() | Value of argument from first row of window frame |
| LAG() | Value of argument from row lagging current row within partition |
| LAST_VALUE() | Value of argument from last row of window frame |
| LEAD() | Value of argument from row leading current row within partition |
| NTH_VALUE() | Value of argument from N-th row of window frame |
| NTILE() | Bucket number of current row within its partition. |
| PERCENT_RANK() | Percentage rank value |
| RANK() | Rank of current row within its partition, with gaps |
| ROW_NUMBER() | Number of current row within its partition |

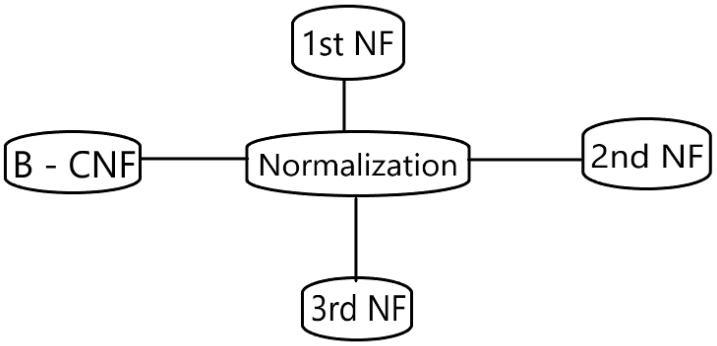
**CTE :**

- Common Table Expressions.
- Simplify various classes of SQL queries for which a derived table was unsuitable.
- Is a temporary named result set that you can reference within SELECT, INSERT, UPDATE or DELETE statements.
- You can also use the CTE in a CREATE view, as a part of the VIEW's SELECT query.
- CTE acts as a virtual table with records and columns that are created during query execution, used by the query, and deleted after the query is executed.

```
[WITH  [, ...]]

::=

cte_name [(column_name [, ...])]

AS (cte_query)
```

**Normalization :**

- Normalization is the process of **reducing the redundancy of data** in the table and also improving the **data integrity** .
- It entails organizing the columns and tables of a database to ensure that **their dependencies are properly enforced** by database integrity constraints.
- Without normalization, we may face issues like :
    - **Insertion Anomaly :** It occurs when we cannot insert the data into the table without the presence of another attribute.
    - **Update Anomaly :** It is a data inconsistency that results from the data redundancy and a partial update of data.
    - **Deletion Anomaly :** It occurs when certain attributes are lost because of deletion of other attributes.
- Normalization in SQL will enhance the distribution of data




- **1st NF :**
    - In this Normal Form, we tackle the problem of **atomicity.**
    - **Atomicity** - Values in the table should not be further divided.
    - Means, a single **cell should not hold multiple values and also not repeating groups of columns**.
    - If a **table contains a composite or multivalued attribute, it violates the 1st NF**

- **2nd NF :**
    - The table is in 1st NF and all the columns **depend on the table's primary key**
    - Table should **not contain partial dependency**.
    - **Partial dependency -** proper subset of candidate key [**composite key**] determines a non-prime attribute.

* **Composite Attribute :** attribute which is **composed of many other attributes**
* **Multivalued Attribute :** attributes which can have **more than one value**

- **3rd NF :**
    - The table is in 2nd NF and all of its columns are **not transitively dependent on the table's primary key**.
    - There should be **no transitive dependency for non-prime attributes**.
    - Means, non-prime attributes [which doesn't form a candidate key] should not be dependent on other non-prime attributes in the given table.

* **Transitive dependency :** it is an indirectly formed dependency between two functional dependencies.
X >>> Y **;** but Y doesn't >>>> X **;** Y >>>> Z
Which tells X has indirectly formed dependency over Z

- **Boyce Codd NF [** BCNF **] :**
    - Also known as **3.5 NF**, higher version of **3rd NF** and was developed by **Raymond F. Boyce & Edgar F.**
    - Table has to be in 3rd NF and for any dependency A >> B, **A should be SUPER KEY.**
    - Means, **A cannot be non-prime attribute when B being a prime-attribute.**

- **4th NF :**
    - Table is in 3rd NF and **does not contain two or more independent multi valued facts about an entity**.

**ACID Properties :**
- A transaction is a single logical unit of work which accesses & possibly modifies the contents of a database.
- Transactions access data using read and write operations.
- In order to maintain consistency in a database, before and after transactions, certain properties are followed termed **ACID** properties.

**A…….Atomicity :** Entire transaction takes place at once or doesn't happen at all.
No midway **;** Either it runs to completion or is not executed at all
Involves 2 operations :

1. **Abort :** if a transaction is aborted , then the changes made to the database are not visible.
2. **Commit :** if a transaction commits, changes made to the database are visible.

Atomicity is also known as **All or Nothing rule**

**C…….Consistency :** Database must be consistent before and after transaction.
It refers to the correctness of the database.

**I……Isolation :** Multiple transactions occur independently without interference.
Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

**D……..Durability :** Changes of a successful transaction occurs even if the system failure occurs.