## SQL Data Types:

SQL provides a variety of data types to store and manipulate data effectively. They are broadly categorized into **numeric**, **date and time**, and **string** types.

## 1. Numeric Data Types:

These store numerical values for mathematical operations.

***Integer Types:***

- **TINYINT**: 1-byte storage.
- **SMALLINT**: 2-byte storage.
- **MEDIUMINT**: 3-byte storage.
- **INT (INTEGER)**: 4-byte storage.
- **BIGINT**: 8-byte storage.

***Floating-Point Types:***

- **FLOAT**: Approximate numeric with 4-byte storage.
- **DOUBLE (DOUBLE PRECISION)**: Approximate numeric with 8-byte storage.

***Fixed-Point Type:***

- **DECIMAL (NUMERIC)**: Exact numeric with specified precision (e.g., DECIMAL(10,2)).

## 2. Date and Time Data Types:

These store date and time values.

- **DATE**: Stores date values (e.g., `YYYY-MM-DD`).
- **DATETIME**: Stores date and time values (e.g., `YYYY-MM-DD HH:MM:SS`)
- **TIME**: Stores time values (e.g., `HH:MM:SS`)
- **YEAR**: Stores year values in `YYYY` format.

## 3. String Data Types:

These store alphanumeric values.

***Fixed-Length Types:***

- **CHAR:** Fixed-length string (e.g., CHAR(10)).

***Variable-Length Types:***

- **VARCHAR:** Variable-length string (e.g., VARCHAR(255)).

## Data Definition Language (DDL):

**Data Definition Language** is a subset of SQL commands used to define, modify, and manage the structure of database objects like tables, indexes, views etc.

DDL statements directly affect the database schema and are automatically committed, meaning changes are permanent.

## Common DDL Commands

***1. CREATE***

Used to create new database objects (e.g., tables, databases, indexes).

**Syntax:**

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

**Example:**

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    hire_date DATE
);
```

***2. ALTER***

Used to modify the structure of existing database objects (e.g., adding, modifying, or deleting columns).

**Syntax:**

```
ALTER TABLE table_name ADD/DROP COLUMN column_name datatype;
```

**Example:**

- Add a new column:

```
ALTER TABLE employees ADD salary DECIMAL(10,2);
```

- Drop a column:

```
ALTER TABLE employees DROP COLUMN hire_date;
```

***3. DROP***

Used to delete database objects permanently (e.g., tables, databases, views).

**Syntax:**

```
DROP TABLE table_name;
DROP DATABASE database_name;
```

**Example:**

```
DROP TABLE employees;
DROP DATABASE company_db;
```

***4. TRUNCATE***

Used to remove all records from a table but keeps the table structure intact. This operation cannot be rolled back.

**Syntax:**

```
TRUNCATE TABLE table_name;
```

**Example:**

```
TRUNCATE TABLE employees;
```

***5. RENAME***

Used to rename database objects such as tables.

**Syntax:**

```
RENAME TABLE old_table_name TO new_table_name;
```

**Example:**

```
RENAME TABLE employees TO staff;
```

## Data Manipulation Language (DML)

**Data Manipulation Language** commands in SQL are used to manage and manipulate data stored in database tables. Unlike Data Definition Language (DDL), DML commands focus on the data itself rather than the schema or structure.

**Common DML Commands**

***1. INSERT***

Used to add new records (rows) into a table.

**Syntax:**

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

**Example:**

```
INSERT INTO employees (id, name, hire_date, salary)
VALUES (1, 'John Doe', '2023-05-01', 55000);
```

***2. UPDATE***

Used to modify existing records in a table.

**Syntax:**

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

**Example:**

```
UPDATE employees
SET salary = 60000
WHERE id = 1;
```

***3. DELETE***

Used to remove records from a table. Unlike **TRUNCATE** (DDL), it allows conditional deletion.

**Syntax:**

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

**Example:**

```
DELETE FROM employees WHERE id = 2;
```

## Data Query Language (DQL):

**Data Query Language** in SQL is primarily used for querying data from a database. The most common and fundamental DQL command is **SELECT**, which retrieves data based on specified conditions.

***SELECT***

The **SELECT** statement retrieves data from one or more tables or views in the database. You can customize the query to specify which columns to fetch, filter results, sort data, group data, and perform aggregate functions.

**Syntax:**

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
GROUP BY column
HAVING condition
ORDER BY column [ASC|DESC];
```

## Key Clauses in SELECT:

***1. WHERE Clause***

Filter rows based on a condition.

***Fetch all employees with a salary greater than 50,000:***

```
SELECT * FROM employees WHERE salary > 50000;
```

***2. ORDER BY Clause***

Sort results in ascending (default) or descending order.

```
SELECT * FROM employees ORDER BY salary DESC;
```

***3. GROUP BY Clause***

Group rows that share a value in specified columns.

***Get the count of employees in each department:***

```
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department;
```

***4. HAVING Clause***

Filter grouped data (used with GROUP BY).

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING avg_salary > 60000;
```

## Data Control Language (DCL)

**Data Control Language** in SQL is used to control access and permissions on database objects. It primarily involves granting or revoking privileges to ensure data security.

## Key DCL Commands

***1. GRANT***

Used to assign privileges to users or roles, allowing them to perform specific actions on database objects.

**Syntax:**

```
GRANT privilege ON object TO user [WITH GRANT OPTION];
```

**Privileges:**

- **ALL PRIVILEGES**: Grants all permissions.
- **SELECT**: Allows reading data.
- **INSERT**: Allows inserting data.
- **UPDATE**: Allows modifying data.
- **DELETE**: Allows deleting data.
- **EXECUTE**: Allows running stored procedures.
- **USAGE**: Allows access to an object without privileges for actions.

**Example:**

- Grant SELECT and INSERT permissions on a table:

```
GRANT SELECT, INSERT ON employees TO 'john';
```

- Grant all privileges on a database:

```
GRANT ALL PRIVILEGES ON company_db.* TO 'admin';
```

***2. REVOKE***

Used to remove previously granted privileges from users or roles.

**Syntax:**

```
REVOKE privilege ON object FROM user;
```

**Example:**

- Revoke INSERT permission from a user:

```
REVOKE INSERT ON employees FROM 'john';
```

- Revoke all privileges from a user on a database:

```
REVOKE ALL PRIVILEGES ON company_db.* FROM 'admin';
```

***3. WITH GRANT OPTION***

Allows a user to grant the same privileges to other users.

**Example:**

- Grant SELECT privilege with the ability to grant it to others:

```
GRANT SELECT ON employees TO 'john' WITH GRANT OPTION;
```

## Transaction Control Language (TCL)

**Transaction Control Language** in SQL is used to manage transactions in a database. Transactions ensure the integrity and consistency of data by grouping multiple SQL statements into a single, atomic unit of work.

## Key TCL Commands:

***1. COMMIT***

Used to save all changes made during the current transaction to the database permanently.

**Syntax:**

```
COMMIT;
```

**Example:**

```
INSERT INTO employees (id, name, salary) VALUES (1, 'Alice', 50000);
INSERT INTO employees (id, name, salary) VALUES (2, 'Bob', 55000);
COMMIT; -- Saves these changes permanently
```

***2. ROLLBACK***

Used to undo all changes made during the current transaction.

**Syntax:**

```
ROLLBACK;
```

**Example:**

```
INSERT INTO employees (id, name, salary) VALUES (3, 'Charlie', 60000);
ROLLBACK; -- Removes the inserted row
```

***3. SAVEPOINT***

Used to set a point within a transaction to which you can later roll back.

**Syntax:**

```
SAVEPOINT savepoint_name;
```

**Example:**

```
BEGIN; -- Start a transaction
INSERT INTO employees (id, name, salary) VALUES (4, 'David', 65000);
SAVEPOINT sp1; -- Savepoint set
INSERT INTO employees (id, name, salary) VALUES (5, 'Eve', 70000);
ROLLBACK TO sp1; -- Reverts changes after the savepoint
COMMIT; -- Saves changes up to the savepoint
```

**Transaction Example:**

```
BEGIN;
UPDATE employees SET salary = salary * 1.1 WHERE department = 'Sales';
SAVEPOINT sp1;
DELETE FROM employees WHERE id = 10;
ROLLBACK TO sp1; -- Undo deletion
COMMIT; -- Save salary updates
```

## SQL JOIN:

In SQL, a **JOIN** is used to combine rows from two or more tables based on a related column. Joins allow you to retrieve data from multiple tables.

## Types of SQL Joins

***1. INNER JOIN***

Returns only the rows where there is a match in both tables.

**Syntax:**

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```

**Example:**

```
SELECT employees.name, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.id;
```

***2. LEFT JOIN (LEFT OUTER JOIN)***

Returns all rows from the left table and matching rows from the right table. If no match exists, NULL values are returned for columns from the right table.

**Syntax:**

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

**Example:**

```
SELECT employees.name, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.id;
```

***3. RIGHT JOIN (RIGHT OUTER JOIN)***

Returns all rows from the right table and matching rows from the left table. If no match exists, NULL values are returned for columns from the left table.

**Syntax:**

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.common_column = table2.common_column;
```

**Example:**

```
SELECT employees.name, departments.department_name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.id;
```

***4. FULL JOIN (FULL OUTER JOIN)***

Returns all rows when there is a match in either the left or the right table. Rows without matches in either table will have NULL in columns from the other table.

**Syntax:**

```
SELECT columns
FROM table1
FULL JOIN table2
ON table1.common_column = table2.common_column;
```

**Example:**

```
SELECT employees.name, departments.department_name
FROM employees
FULL JOIN departments
ON employees.department_id = departments.id;
```

*5.* ***CROSS JOIN***

Returns the Cartesian product of the two tables (all possible combinations of rows).

**Syntax:**

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

**Example:**

```
SELECT employees.name, departments.department_name
FROM employees
CROSS JOIN departments;
```

***6. SELF JOIN***

A join where a table is joined with itself.

**Syntax:**

```
SELECT a.column1, b.column2
FROM table_name a, table_name b
WHERE a.common_column = b.common_column;
```

**Example:**

```
SELECT e1.name AS Employee, e2.name AS Manager
FROM employees e1
INNER JOIN employees e2
ON e1.manager_id = e2.id;
```

## Nested Query:

A **nested query** in SQL, also known as a **sub-query**, is a query within another SQL query. The inner query provides data that the outer query uses for its execution. Nested queries are often used to perform complex operations in a more modular and readable way.

**Operators Commonly Used with Subqueries**

**IN**: Matches any value in a list.

```
SELECT name FROM employees WHERE department_id IN (SELECT id FROM
departments);
```

**ANY**: Matches any value returned by the subquery.

```
SELECT name FROM employees WHERE salary > ANY (SELECT salary FROM employees
WHERE department_id = 10);
```

**ALL**: Matches all values returned by the subquery.

```
SELECT name FROM employees WHERE salary > ALL (SELECT salary FROM employees
WHERE department_id = 10);
```

**EXISTS**: Checks for the existence of rows returned by the subquery.

```
SELECT name FROM employees WHERE EXISTS (SELECT 1 FROM departments WHERE
employees.departme
```

## SQL Operators:

SQL operators are special symbols or keywords used to specify conditions in SQL statements and manipulate data. Here's a breakdown of the key types of operators in SQL:

**1. Arithmetic Operators**

Used to perform mathematical operations.

| Operator | Description |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modulus (remainder) |

**Example**:

```
SELECT 10 + 5 AS AdditionResult;
```

**2. Comparison Operators**

Used to compare two values.

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

**Example**:

```
SELECT * FROM Students WHERE Age >= 18;
```

**3. Logical Operators**

Used to combine multiple conditions.

| Operator | Description |
|---|---|
| AND | Returns true if both conditions are true |
| OR | Returns true if at least one condition is true |
| NOT | Reverses the logical state of the condition |

**Example**:

```
SELECT * FROM Students WHERE Age > 18 AND Grade = 'A';
```

**4. Set Operators**

Used to combine results from multiple SELECT statements.

| Operator | Description |
|---|---|
| UNION | Combines results, removes duplicates |
| UNION ALL | Combines results, includes duplicates |
| INTERSECT | Returns common rows |
| EXCEPT | Returns rows in the first query not in the second |

**Example**:

```
SELECT Name FROM Students UNION SELECT Name FROM Teachers;
```

## BETWEEN Operator in SQL

The BETWEEN operator is used to filter records within a specific range. The range can be numeric, date, or even text. It includes the boundary values.

***Example: Numeric Range***

```
SELECT *
FROM Employees
WHERE Salary BETWEEN 30000 AND 50000;
```

*This retrieves all employees whose salaries are between 30,000 and 50,000 (inclusive).*

***Example: Date Range***

```
SELECT *
FROM Orders
WHERE OrderDate BETWEEN '2023-01-01' AND '2023-12-31';
```

*This retrieves all orders placed within the year 2023.*

***Example: NOT BETWEEN***

```
SELECT *
FROM Products
WHERE Price NOT BETWEEN 100 AND 500;
```

*This retrieves products priced outside the range of 100 to 500.*

## LIKE Operator in SQL

The LIKE operator is used for pattern matching in a string column. It is commonly used with wildcards % which matches zero or more characters.

```
SELECT *
FROM Customers
WHERE Name LIKE 'A%';
```

*Finds all names starting with "A".*

***Using NOT LIKE***

```
SELECT *
FROM Customers
WHERE Name NOT LIKE '%Smith';
```

*This retrieves all customers whose names do not end with "Smith".*

## SQL Functions:

SQL provides several predefined functions that make it easier to perform operations on data. These functions are:

**1. Aggregate Functions**

Operate on a set of rows and return a single result.

| Function | Description |
|---|---|
| AVG() | Returns the average of a numeric column |
| COUNT() | Returns the count of rows |
| MAX() | Returns the maximum value in a column |
| MIN() | Returns the minimum value in a column |
| SUM() | Returns the sum of a numeric column |

**Example**:

```
SELECT AVG(Salary) AS AverageSalary FROM Employees;
```

**2. String Functions**

Used to perform operations on strings.

| Function | Description |
|---|---|
| UPPER() | Converts a string to uppercase |
| LOWER() | Converts a string to lowercase |
| LTRIM() | Removes leading spaces |
| RTRIM() | Removes trailing spaces |
| LEN() or LENGTH() | Returns the length of a string |
| CONCAT() | Concatenates two or more strings |
| SUBSTRING() | Extracts a substring from a string |
| CHARINDEX() or INSTR() | Finds the position of a substring |
| REPLACE() | Replaces occurrences of a substring |
| REVERSE() | Reverses a string |

**Example**:

```
SELECT UPPER('hello world') AS UppercaseText;
```

**3. Numeric Functions**

Perform operations on numeric data.

| Function | Description |
|---|---|
| ABS() | Returns the absolute value of a number |
| CEILING() | Rounds up to the nearest integer |
| FLOOR() | Rounds down to the nearest integer |
| ROUND() | Rounds a number to a specified number of digits |
| POWER() | Returns a number raised to a power |
| SQRT() | Returns the square root of a number |
| RAND() | Generates a random number |

**Example**:

```
SELECT ROUND(123.456, 2) AS RoundedNumber;
```

**4. Date/Time Functions**

Used to manipulate and retrieve date and time values.

| Function | Description |
|---|---|
| GETDATE() | Returns the current date and time |
| DATEADD() | Adds a specified time interval to a date |
| DATENAME() | Returns the name of a part of a date |
| FORMAT() | Formats a date or number as a string |
| NOW() | Returns the current timestamp |

**Example**:

```
SELECT GETDATE() AS CurrentDateTime;
```

## Views in SQL

A **view** in SQL is a virtual table based on the result of a query. It allows you to simplify complex queries, enhance security by restricting access to certain columns or rows, and present data in a more readable format. A view does not store data itself but retrieves data from the underlying tables each time it is queried.

**Creating a View**

You can create a view using the CREATE VIEW statement followed by a SELECT query.

***Syntax***

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

***Example: Creating a Simple View***

```
CREATE VIEW EmployeeView AS
SELECT EmployeeID, FirstName, LastName, Department
FROM Employees
WHERE Status = 'Active';
```

*This creates a view EmployeeView that shows only active employees from the Employees table.*

**Using a View**

Once a view is created, you can query it just like a table:

***Example: Querying a View***

```
SELECT * FROM EmployeeView;
```

*This will return all the active employees (EmployeeID, FirstName, LastName, Department) from the EmployeeView.*

**Modifying a View**

You can modify an existing view using the CREATE OR REPLACE VIEW statement.

***Syntax***

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

***Example: Modifying a View***

```
CREATE OR REPLACE VIEW EmployeeView AS
SELECT EmployeeID, FirstName, LastName, Department, Salary
FROM Employees
WHERE Status = 'Active';
```

*This will modify the EmployeeView to include the Salary column.*

**Dropping a View**

If you no longer need a view, you can remove it using the DROP VIEW statement.

***Syntax***

```
DROP VIEW view_name;
```

***Example: Dropping a View***

```
DROP VIEW EmployeeView;
```

*This will delete the EmployeeView.*

## Benefits of Views

1. **Simplifies Complex Queries**: By encapsulating complex queries, views make it easier to run common queries.
2. **Security**: You can restrict access to sensitive data by exposing only specific columns or rows via a view.
3. **Data Abstraction**: Views can present data in a more meaningful format, abstracting the complexity of the underlying database schema.

**Limitations of Views**

1. **Performance**: Since views are virtual tables, querying them might be slower compared to querying base tables, especially if the view is complex.
2. **Update Restrictions**: Not all views are updatable, particularly those involving complex joins or aggregation.
3. **No Indexes**: Views do not support indexing directly, which may impact performance in large datasets.

## Sequences in SQL:

A **sequence** in SQL is a database object that generates a series of unique numbers. It is commonly used for generating primary key values or for cases where unique identifiers are needed.

**Creating a Sequence**

The CREATE SEQUENCE statement is used to create a sequence. You can specify parameters like the starting value, increment, and other properties.

***Syntax***

CREATE SEQUENCE sequence_name

```
START WITH start_value
INCREMENT BY increment_value
[MINVALUE min_value | NOMINVALUE]
[MAXVALUE max_value | NOMAXVALUE]
[CYCLE | NOCYCLE]
[CACHE cache_size | NOCACHE];
```

***Parameters***

- START WITH: Specifies the first number in the sequence.
- INCREMENT BY: Specifies the interval between numbers (default is 1).
- MINVALUE and MAXVALUE: Optional limits on the sequence.
- CYCLE: Allows the sequence to restart once it reaches the maximum value.
- CACHE: Specifies the number of sequence numbers to be pre-allocated for faster access.

***Example: Creating a Sequence***

```
CREATE SEQUENCE EmployeeSeq
START WITH 1000
INCREMENT BY 1
MINVALUE 1000
MAXVALUE 9999
CYCLE;
```

*This creates a sequence EmployeeSeq starting at 1000, incrementing by 1, and cycling back to 1000 once it reaches 9999.*

**Using a Sequence**

Once a sequence is created, you can use it to generate a new value using the NEXTVAL and CURRVAL functions.

- NEXTVAL: Retrieves the next value in the sequence and increments the sequence.
- CURRVAL: Returns the current value of the sequence (after NEXTVAL has been used at least once in the session).

***Example: Retrieving the Next Value***

```
SELECT EmployeeSeq.NEXTVAL AS NewEmployeeID;
```

*This retrieves the next value from the EmployeeSeq sequence.*

***Example: Using Sequence in Insert***

```
INSERT INTO Employees (EmployeeID, FirstName, LastName)
VALUES (EmployeeSeq.NEXTVAL, 'John', 'Doe');
```

*This inserts a new employee with a unique EmployeeID generated by the sequence.*

***Example: Retrieving the Current Value***

```
SELECT EmployeeSeq.CURRVAL AS CurrentEmployeeID;
```

*This returns the current value of EmployeeSeq after a call to NEXTVAL.*

**Modifying a Sequence**

You can modify certain properties of a sequence using the ALTER SEQUENCE statement, though you cannot change the START WITH or INCREMENT BY values directly.

***Example: Altering a Sequence***

```
ALTER SEQUENCE EmployeeSeq
INCREMENT BY 2;
```

*This alters the sequence to increment by 2 instead of 1.*

**Dropping a Sequence**

If you no longer need a sequence, you can remove it using the DROP SEQUENCE statement.

***Syntax***

```
DROP SEQUENCE sequence_name;
```

***Example: Dropping a Sequence***

```
DROP SEQUENCE EmployeeSeq;
```

*This drops the EmployeeSeq sequence from the database.*

## Indexes in SQL:

An **index** in SQL is a database object that improves the speed of data retrieval operations on a table at the cost of additional space and time required for updates. Indexes work similarly to indexes in books, helping the database engine to find data faster without scanning every row in a table.

**Types of Indexes in SQL**

1. **Single-Column Index**: Created on a single column of a table.
2. **Multi-Column Index**: Created on two or more columns of a table.
3. **Unique Index**: Ensures that all values in the indexed column(s) are unique.
4. **Clustered Index**: The table data is physically sorted according to the index.
5. **Non-Clustered Index**: The index is stored separately from the table data and points to the data rows.

**Creating an Index**

You can create an index using the CREATE INDEX statement.

***Syntax***

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

- UNIQUE: Ensures that the index enforces unique values in the indexed columns.
- index_name: Name of the index.
- table_name: Name of the table on which the index is created.
- column1, column2: The columns on which the index is created.

***Example: Creating a Single-Column Index***

```
CREATE INDEX idx_employee_name
ON Employees (LastName);
```

*This creates an index on the LastName column of the Employees table, improving the speed of queries that search by LastName.*

***Example: Creating a Unique Index***

```
CREATE UNIQUE INDEX idx_unique_employee_id
ON Employees (EmployeeID);
```

*This creates a unique index on the EmployeeID column, ensuring that all values in EmployeeID are unique.*

***Example: Creating a Multi-Column (Composite) Index***

```
CREATE INDEX idx_employee_department
ON Employees (Department, Salary);
```

*This creates an index on both Department and Salary columns. This improves performance for queries that filter or sort by both of these columns.*

**Dropping an Index**

You can remove an index using the DROP INDEX statement.

***Syntax***

```
DROP INDEX index_name;
```

***Example: Dropping an Index***

```
DROP INDEX idx_employee_name;
```

*This removes the idx_employee_name index from the Employees table.*

**When to Use Indexes**

- **Speed up SELECT queries**: Indexes are most useful for read-heavy applications where query performance is a concern.
- **Unique constraints**: Ensuring that values in a column (or set of columns) are unique can be efficiently handled using unique indexes.
- **Sorting and Filtering**: Indexes improve performance for queries that involve sorting (ORDER BY) and filtering (WHERE clauses).
- **Join Optimization**: Indexes on the columns used in JOIN conditions can significantly improve performance.

**Disadvantages of Indexes**

1. **Increased Storage**: Indexes consume additional disk space.
2. **Slower Insert, Update, and Delete Operations**: Since indexes must be updated when data is modified, operations like INSERT, UPDATE, and DELETE become slower.
3. **Complexity**: Having too many indexes can lead to database bloat and reduced performance for writes.