**Basics of SQL (Structured Query Language)**

SQL is a standard programming language used to manage and manipulate relational databases. It allows you to perform various tasks such as retrieving data, updating records, and creating database structures.

**Key Categories of SQL Commands**

**1. DDL (Data Definition Language)**

DDL commands are used to define and manage the structure of a database. These commands deal with the schema (structure) of the database.

* **CREATE**: Used to create a new table, view, or other database objects.
  + Example:
  + CREATE TABLE Students (
  + StudentID INT,
  + Name VARCHAR(100),
  + Age INT
  + );
* **ALTER**: Used to modify the structure of an existing database object, such as adding a column.
  + Example:
  + ALTER TABLE Students ADD COLUMN Address VARCHAR(255);
* **DROP**: Used to delete an existing database object.
  + Example:
  + DROP TABLE Students;
* **TRUNCATE**: Removes all rows from a table but keeps the structure intact.
  + Example:
  + TRUNCATE TABLE Students;

**2. DML (Data Manipulation Language)**

DML commands are used to manipulate data stored in the database. These commands deal with the actual data (records) rather than the structure.

* **SELECT**: Retrieves data from one or more tables.
  + Example:
  + SELECT \* FROM Students WHERE Age > 18;
* **INSERT**: Adds new records into a table.
  + Example:
  + INSERT INTO Students (StudentID, Name, Age) VALUES (1, 'John Doe', 19);
* **UPDATE**: Modifies existing records in a table.
  + Example:
  + UPDATE Students SET Age = 20 WHERE StudentID = 1;
* **DELETE**: Removes existing records from a table.
  + Example:
  + DELETE FROM Students WHERE StudentID = 1;

**3. DCL (Data Control Language)**

DCL commands are used to control access to data in a database. These are used to grant or revoke permissions.

* **GRANT**: Gives a user or role permission to perform a specific action on a database object.
  + Example:
  + GRANT SELECT ON Students TO User1;
* **REVOKE**: Removes a user’s permission to perform an action on a database object.
  + Example:
  + REVOKE SELECT ON Students FROM User1;

**Defining Constraints in SQL**

Constraints are rules applied to columns in SQL tables to enforce data integrity and ensure that data entered into the table meets certain conditions. Constraints can be defined when creating or altering a table, and they are crucial for maintaining the accuracy and consistency of data.

Here’s an overview of common constraints in SQL:

**1. PRIMARY KEY**

A **Primary Key** constraint uniquely identifies each record in a table. A primary key must contain unique values, and it cannot contain NULL values. A table can have only one primary key, but the primary key can consist of multiple columns (composite key).

* **Usage:** Ensures that no two rows can have the same value in the primary key column(s).
* **Example:**
* CREATE TABLE Employees (
* EmployeeID INT PRIMARY KEY,
* Name VARCHAR(100)
* );

In this example, EmployeeID must be unique for each employee, and no employee can have a NULL EmployeeID.

**2. FOREIGN KEY**

A **Foreign Key** constraint is used to link two tables together. It ensures that the value in one table matches a value in another table’s primary key or unique column. It helps maintain referential integrity between the two tables.

* **Usage:** Enforces that each value in the foreign key column corresponds to a valid value in the referenced table.
* **Example:**
* CREATE TABLE Orders (
* OrderID INT PRIMARY KEY,
* CustomerID INT,
* FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
* );

In this example, the CustomerID in the Orders table must match a CustomerID in the Customers table. This maintains integrity by ensuring that an order cannot exist without a valid customer.

**3. UNIQUE**

A **Unique** constraint ensures that all values in a column (or a combination of columns) are distinct. Unlike the primary key, a table can have multiple unique constraints, and a column with a unique constraint can contain NULL values.

* **Usage:** Ensures that each value in the column is unique.
* **Example:**
* CREATE TABLE Employees (
* EmployeeID INT PRIMARY KEY,
* Email VARCHAR(100) UNIQUE
* );

In this example, the Email column must have unique values, meaning no two employees can share the same email.

**4. NOT NULL**

A **Not Null** constraint ensures that a column cannot have a NULL value. This is useful when you want to ensure that every record contains a valid value for that column.

* **Usage:** Prevents NULL values from being inserted into the column.
* **Example:**
* CREATE TABLE Employees (
* EmployeeID INT PRIMARY KEY,
* Name VARCHAR(100) NOT NULL
* );

In this example, the Name column cannot contain NULL values, meaning every employee must have a name.

**5. CHECK**

A **Check** constraint ensures that all values in a column meet a specified condition or rule. It is used to enforce domain integrity by limiting the range of values in a column.

* **Usage:** Restricts the values that can be inserted into the column based on the condition defined in the CHECK constraint.
* **Example:**
* CREATE TABLE Employees (
* EmployeeID INT PRIMARY KEY,
* Age INT CHECK (Age >= 18)
* );

In this example, the Age column must have values that are greater than or equal to 18. This prevents the insertion of employees under 18 years old.

**6. IN Operator**

The **IN** operator is not a constraint itself, but it is used within CHECK or WHERE clauses to specify multiple values in a condition. It allows you to check whether a column value matches one of a set of values.

* **Usage:** Checks if the value in a column matches one of the values in a predefined list.
* **Example:**
* CREATE TABLE Employees (
* EmployeeID INT PRIMARY KEY,
* Department VARCHAR(50) CHECK (Department IN ('HR', 'Finance', 'IT', 'Sales'))
* );

In this example, the Department column can only contain one of the following values: 'HR', 'Finance', 'IT', or 'Sales'. Any other value will be rejected.

**Summary of Constraints**

| **Constraint** | **Purpose** | **Example** |
| --- | --- | --- |
| **PRIMARY KEY** | Uniquely identifies each record in a table. | EmployeeID INT PRIMARY KEY |
| **FOREIGN KEY** | Enforces a link between two tables based on primary/unique keys. | FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) |
| **UNIQUE** | Ensures all values in a column are distinct. | Email VARCHAR(100) UNIQUE |
| **NOT NULL** | Ensures a column cannot have NULL values. | Name VARCHAR(100) NOT NULL |
| **CHECK** | Restricts the values that can be entered in a column. | Age INT CHECK (Age >= 18) |
| **IN** | Specifies multiple allowed values for a column. | Department VARCHAR(50) CHECK (Department IN ('HR', 'Finance')) |

**Aggregate Functions in SQL**

Aggregate functions are used in SQL to perform calculations on multiple rows of a table and return a single value. These functions are typically used with the **GROUP BY** clause to group rows and perform the calculation on each group. Aggregate functions can also be used without **GROUP BY** to perform calculations on all rows of a table.

Here’s an overview of the most commonly used **aggregate functions** in SQL:

**1. COUNT()**

The **COUNT()** function returns the number of rows that match a specified condition. It counts all rows, including those with NULL values, unless specified otherwise.

* **Usage:** To count the number of rows in a table or the number of non-NULL values in a column.
* **Example (Count all rows):**
* SELECT COUNT(\*) FROM Employees;

This will return the total number of rows in the Employees table.

* **Example (Count non-NULL values):**
* SELECT COUNT(Email) FROM Employees;

This will count the number of non-NULL entries in the Email column.

**2. SUM()**

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

* **Usage:** To calculate the sum of values in a numeric column.
* **Example:**
* SELECT SUM(Salary) FROM Employees;

This will return the total sum of the Salary column from the Employees table.

**3. AVG()**

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

* **Usage:** To calculate the average of values in a numeric column.
* **Example:**
* SELECT AVG(Salary) FROM Employees;

This will return the average value of the Salary column from the Employees table.

**4. MIN()**

The **MIN()** function returns the smallest value in a specified column.

* **Usage:** To find the minimum value in a column.
* **Example:**
* SELECT MIN(Salary) FROM Employees;

This will return the smallest salary in the Employees table.

**5. MAX()**

The **MAX()** function returns the largest value in a specified column.

* **Usage:** To find the maximum value in a column.
* **Example:**
* SELECT MAX(Salary) FROM Employees;

This will return the highest salary in the Employees table.

**6. GROUP\_CONCAT() / STRING\_AGG()**

These functions are used to concatenate values from multiple rows into a single string. The function syntax may vary depending on the SQL database (e.g., MySQL vs. PostgreSQL).

* **Usage:** To combine multiple row values into a single string, separated by a delimiter.
* **Example (MySQL):**
* SELECT GROUP\_CONCAT(Name SEPARATOR ', ') FROM Employees;

This will return a comma-separated list of employee names.

* **Example (PostgreSQL):**
* SELECT STRING\_AGG(Name, ', ') FROM Employees;

This will return the employee names as a single, comma-separated string.

**7. VARIANCE() / VAR\_POP()**

The **VARIANCE()** and **VAR\_POP()** functions calculate the variance of a set of numeric values. Variance measures how far values are spread out from the mean.

* **Usage:** To calculate the variance of a numeric column.
* **Example (Variance):**
* SELECT VARIANCE(Salary) FROM Employees;

This will return the variance of the Salary column in the Employees table.

**8. STDDEV() / STDDEV\_POP()**

The **STDDEV()** and **STDDEV\_POP()** functions calculate the standard deviation of a set of numeric values. Standard deviation is the square root of variance and indicates the spread or dispersion of values in the dataset.

* **Usage:** To calculate the standard deviation of a numeric column.
* **Example (Standard Deviation):**
* SELECT STDDEV(Salary) FROM Employees;

This will return the standard deviation of the Salary column in the Employees table.

**9. FIRST() and LAST()**

Some SQL databases, like MySQL, support the **FIRST()** and **LAST()** functions to return the first and last value in a column.

* **Usage:** To retrieve the first or last value in a column (based on sorting).
* **Example (MySQL):**
* SELECT FIRST(Name) FROM Employees ORDER BY JoiningDate;
* SELECT LAST(Name) FROM Employees ORDER BY JoiningDate DESC;

**Using Aggregate Functions with GROUP BY**

Aggregate functions are commonly used with the **GROUP BY** clause to group rows based on specified columns and then perform calculations on each group.

**Example (Group by Department and calculate average salary):**

SELECT Department, AVG(Salary)

FROM Employees

GROUP BY Department;

This will return the average salary for each department.

**Filtering Aggregate Results with HAVING**

The **HAVING** clause is used to filter the results of aggregate functions after the **GROUP BY** operation. It is similar to the **WHERE** clause but is used for aggregated data.

**Example (Filter departments with average salary greater than 50,000):**

SELECT Department, AVG(Salary)

FROM Employees

GROUP BY Department

HAVING AVG(Salary) > 50000;

This will return only those departments where the average salary is greater than 50,000.

**Summary of Aggregate Functions**

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| **COUNT()** | Returns the number of rows that match a condition. | COUNT(\*) |
| **SUM()** | Calculates the total sum of a numeric column. | SUM(Salary) |
| **AVG()** | Calculates the average of a numeric column. | AVG(Salary) |
| **MIN()** | Returns the smallest value in a column. | MIN(Salary) |
| **MAX()** | Returns the largest value in a column. | MAX(Salary) |
| **GROUP\_CONCAT() / STRING\_AGG()** | Concatenates values from multiple rows into a single string. | GROUP\_CONCAT(Name) |
| **VARIANCE()** | Calculates the variance of a numeric column. | VARIANCE(Salary) |
| **STDDEV()** | Calculates the standard deviation of a numeric column. | STDDEV(Salary) |
| **FIRST()** | Returns the first value in a column (depending on sorting). | FIRST(Name) |
| **LAST()** | Returns the last value in a column (depending on sorting). | LAST(Name) |

These aggregate functions are fundamental for analyzing and summarizing large datasets.

**Built-in Functions in SQL**

SQL provides a wide range of built-in functions that you can use to perform operations on numeric, string, and date data types. These functions help simplify and automate tasks like mathematical calculations, string manipulation, and working with dates.

Here’s a breakdown of **numeric**, **date**, **string** functions, and **set operations** in SQL:

**1. Numeric Functions**

Numeric functions in SQL perform mathematical operations on numeric data types (like INT, FLOAT, DECIMAL).

**Common Numeric Functions:**

* **ABS(x)**: Returns the absolute value of a number.
  + **Example:**
  + SELECT ABS(-100);

Returns 100.

* **CEIL(x) or CEILING(x)**: Returns the smallest integer greater than or equal to the number.
  + **Example:**
  + SELECT CEIL(7.25);

Returns 8.

* **FLOOR(x)**: Returns the largest integer less than or equal to the number.
  + **Example:**
  + SELECT FLOOR(7.25);

Returns 7.

* **ROUND(x, d)**: Rounds a number x to d decimal places.
  + **Example:**
  + SELECT ROUND(123.456, 2);

Returns 123.46.

* **POWER(x, y)**: Returns x raised to the power of y.
  + **Example:**
  + SELECT POWER(2, 3);

Returns 8.

* **MOD(x, y)**: Returns the remainder when x is divided by y.
  + **Example:**
  + SELECT MOD(10, 3);

Returns 1.

* **SQRT(x)**: Returns the square root of x.
  + **Example:**
  + SELECT SQRT(16);

Returns 4.

* **RAND()**: Returns a random floating-point value between 0 and 1.
  + **Example:**
  + SELECT RAND();

**2. Date Functions**

Date functions are used to manipulate or retrieve information about date and time data types.

**Common Date Functions:**

* **CURRENT\_DATE or CURDATE()**: Returns the current date (without time).
  + **Example:**
  + SELECT CURRENT\_DATE;

Returns 2025-04-22 (current date).

* **CURRENT\_TIME or CURTIME()**: Returns the current time (without date).
  + **Example:**
  + SELECT CURRENT\_TIME;
* **CURRENT\_TIMESTAMP or NOW()**: Returns the current date and time.
  + **Example:**
  + SELECT CURRENT\_TIMESTAMP;
* **DATE\_ADD(date, INTERVAL value unit)**: Adds a specified time interval to a date.
  + **Example:**
  + SELECT DATE\_ADD('2025-04-22', INTERVAL 5 DAY);

Returns 2025-04-27.

* **DATE\_SUB(date, INTERVAL value unit)**: Subtracts a specified time interval from a date.
  + **Example:**
  + SELECT DATE\_SUB('2025-04-22', INTERVAL 5 DAY);

Returns 2025-04-17.

* **DATEDIFF(date1, date2)**: Returns the difference in days between two dates.
  + **Example:**
  + SELECT DATEDIFF('2025-04-22', '2025-04-17');

Returns 5.

* **YEAR(date)**: Extracts the year part of a date.
  + **Example:**
  + SELECT YEAR('2025-04-22');

Returns 2025.

* **MONTH(date)**: Extracts the month part of a date.
  + **Example:**
  + SELECT MONTH('2025-04-22');

Returns 4.

* **DAY(date)**: Extracts the day part of a date.
  + **Example:**
  + SELECT DAY('2025-04-22');

Returns 22.

* **DATE\_FORMAT(date, format)**: Formats a date value according to a specified format.
  + **Example:**
  + SELECT DATE\_FORMAT('2025-04-22', '%Y-%m-%d');

Returns 2025-04-22.

**3. String Functions**

String functions in SQL help manipulate and process string (text) data.

**Common String Functions:**

* **CONCAT(string1, string2, ...)**: Concatenates two or more strings into one.
  + **Example:**
  + SELECT CONCAT('Hello', ' ', 'World');

Returns Hello World.

* **LENGTH(string)**: Returns the length of the string in characters.
  + **Example:**
  + SELECT LENGTH('Hello World');

Returns 11.

* **LOWER(string)**: Converts the string to lowercase.
  + **Example:**
  + SELECT LOWER('Hello World');

Returns hello world.

* **UPPER(string)**: Converts the string to uppercase.
  + **Example:**
  + SELECT UPPER('Hello World');

Returns HELLO WORLD.

* **SUBSTRING(string, start, length)**: Extracts a substring from the string, starting from position start and with a specified length.
  + **Example:**
  + SELECT SUBSTRING('Hello World', 1, 5);

Returns Hello.

* **TRIM(string)**: Removes leading and trailing spaces from a string.
  + **Example:**
  + SELECT TRIM(' Hello World ');

Returns Hello World.

* **REPLACE(string, old\_substring, new\_substring)**: Replaces occurrences of a substring in the string.
  + **Example:**
  + SELECT REPLACE('Hello World', 'World', 'SQL');

Returns Hello SQL.

* **INSTR(string, substring)**: Returns the position of the first occurrence of a substring within a string.
  + **Example:**
  + SELECT INSTR('Hello World', 'World');

Returns 7.

**4. Set Operations in SQL**

Set operations in SQL are used to combine the results of two or more queries. The key set operations are **UNION**, **INTERSECT**, and **EXCEPT** (or **MINUS** in some SQL databases).

**Common Set Operations:**

* **UNION**: Combines the results of two or more SELECT queries and removes duplicate rows.
  + **Example:**
  + SELECT Name FROM Employees
  + UNION
  + SELECT Name FROM Managers;

This combines the results of Employees and Managers and removes duplicates.

* **UNION ALL**: Similar to UNION, but does not remove duplicate rows.
  + **Example:**
  + SELECT Name FROM Employees
  + UNION ALL
  + SELECT Name FROM Managers;

This combines the results, including duplicates.

* **INTERSECT**: Returns only the rows that are common to both SELECT queries.
  + **Example:**
  + SELECT Name FROM Employees
  + INTERSECT
  + SELECT Name FROM Managers;

This returns only the names that appear in both Employees and Managers tables.

* **EXCEPT / MINUS**: Returns the rows from the first SELECT query that do not exist in the second SELECT query (in some SQL databases, the operation is called MINUS).
  + **Example:**
  + SELECT Name FROM Employees
  + EXCEPT
  + SELECT Name FROM Managers;

This returns the names of employees who are not managers.

**Summary of Functions**

| **Function Type** | **Function** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Numeric** | ABS(), CEIL(), FLOOR(), ROUND(), etc. | Mathematical operations on numbers. | ROUND(12.678, 1) |
| **Date** | CURRENT\_DATE, DATE\_ADD(), DATEDIFF(), etc. | Operations on date and time values. | DATE\_ADD('2025-04-22', INTERVAL 7 DAY) |
| **String** | CONCAT(), LOWER(), LENGTH(), REPLACE(), etc. | Operations on string values. | CONCAT('Hello', ' ', 'World') |
| **Set Operations** | UNION, INTERSECT, EXCEPT, etc. | Combine and compare results from multiple queries. | SELECT \* FROM A UNION SELECT \* FROM B |

These functions make SQL more powerful and allow you to handle a wide range of data processing tasks efficiently.

**Subqueries in SQL**

A **subquery** (also called a nested query or inner query) is a query embedded inside another query. Subqueries are used to retrieve data that will be used in the main query’s condition or SELECT statement. Subqueries can be categorized into two main types: **single-row subqueries** and **multiple-row subqueries**. They can also be classified based on whether they are **correlated** or **non-correlated**.

**1. Subquery Basics**

A **subquery** is a query embedded within a larger query, often within a WHERE, HAVING, or FROM clause. The subquery returns a value or a set of values that the outer query uses to filter or manipulate data.

**General Syntax for Subqueries:**

SELECT column1, column2

FROM table1

WHERE column1 = (SELECT column1 FROM table2 WHERE condition);

**2. Types of Subqueries**

**2.1 Non-Correlated Subquery (Independent Subquery)**

A **non-correlated subquery** is a subquery that can be executed independently of the outer query. It does not depend on the outer query's data and is executed once for the entire outer query.

* **Usage:** Typically used when you want to compare a column value with a result from another table.
* **Example:**
* SELECT EmployeeID, Name
* FROM Employees
* WHERE DepartmentID = (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'HR');

In this example:

* + The subquery finds the DepartmentID of the "HR" department.
  + The outer query then retrieves all employees in the HR department using the DepartmentID from the subquery.
  + The subquery is independent and can be executed on its own.

**2.2 Correlated Subquery**

A **correlated subquery** is a subquery that depends on the outer query. It references one or more columns from the outer query in its conditions. For each row processed by the outer query, the subquery is re-executed, making it "correlated" with the outer query.

* **Usage:** Typically used when you need to compare each row from the outer query with a condition that involves the current row from the outer query.
* **Example:**
* SELECT EmployeeID, Name
* FROM Employees e
* WHERE EXISTS (
* SELECT 1
* FROM Departments d
* WHERE d.DepartmentID = e.DepartmentID AND d.DepartmentName = 'HR'
* );

In this example:

* + The subquery references the outer query's e.DepartmentID value.
  + For each employee, the subquery checks if the department of that employee is "HR".
  + The subquery is re-executed for each employee, making it correlated with the outer query.

**3. Correlated vs. Non-Correlated Subqueries**

| **Type of Subquery** | **Description** | **Example** |
| --- | --- | --- |
| **Non-Correlated Subquery** | Independent of the outer query; it can be executed separately. | SELECT Name FROM Employees WHERE DepartmentID = (SELECT DepartmentID FROM Departments WHERE Name = 'HR') |
| **Correlated Subquery** | Dependent on the outer query; for each row, the subquery is executed again. | SELECT Name FROM Employees e WHERE EXISTS (SELECT 1 FROM Departments d WHERE d.DepartmentID = e.DepartmentID AND d.Name = 'HR') |

**4. Subquery in Different Clauses**

**4.1 Using Subquery in the WHERE Clause**

Subqueries in the WHERE clause are used to filter records based on the results of the subquery.

* **Example (Non-Correlated Subquery in WHERE):**
* SELECT Name
* FROM Employees
* WHERE DepartmentID = (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'IT');
* **Example (Correlated Subquery in WHERE):**
* SELECT Name
* FROM Employees e
* WHERE e.Salary > (SELECT AVG(Salary) FROM Employees WHERE DepartmentID = e.DepartmentID);

In this example, the subquery calculates the average salary within each department and compares each employee’s salary to that average.

**4.2 Using Subquery in the FROM Clause**

Subqueries in the FROM clause are treated as a derived table (also called a subquery in the FROM clause). The outer query can treat the result of the subquery as a table.

* **Example (Non-Correlated Subquery in FROM):**
* SELECT DeptAvg.DepartmentID, DeptAvg.AvgSalary
* FROM (SELECT DepartmentID, AVG(Salary) AS AvgSalary
* FROM Employees
* GROUP BY DepartmentID) AS DeptAvg;

In this example, the subquery in the FROM clause calculates the average salary per department, and the outer query selects the department ID and the average salary.

**4.3 Using Subquery in the SELECT Clause**

Subqueries in the SELECT clause allow you to return additional calculated values for each row.

* **Example (Non-Correlated Subquery in SELECT):**
* SELECT Name, (SELECT MAX(Salary) FROM Employees WHERE DepartmentID = e.DepartmentID) AS MaxSalaryInDept
* FROM Employees e;

In this example, for each employee, the subquery finds the highest salary in the same department and includes that as MaxSalaryInDept in the result.

**5. Subquery with EXISTS and IN**

* **EXISTS**: The EXISTS operator is used to check if a subquery returns any rows. It is generally used with correlated subqueries to check if a certain condition holds true for any row.
  + **Example (Correlated Subquery with EXISTS):**
  + SELECT Name
  + FROM Employees e
  + WHERE EXISTS (
  + SELECT 1
  + FROM Departments d
  + WHERE d.DepartmentID = e.DepartmentID AND d.DepartmentName = 'HR'
  + );
* **IN**: The IN operator is used with a subquery to check if a value matches any value in a set returned by the subquery. It can be used with both correlated and non-correlated subqueries.
  + **Example (Non-Correlated Subquery with IN):**
  + SELECT Name
  + FROM Employees
  + WHERE DepartmentID IN (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'HR');
  + **Example (Correlated Subquery with IN):**
  + SELECT Name
  + FROM Employees e
  + WHERE e.DepartmentID IN (SELECT DepartmentID FROM Departments d WHERE d.DepartmentName = 'HR');

**6. Performance Considerations**

* **Non-Correlated Subqueries** are generally faster because they can be executed once and the result used by the outer query.
* **Correlated Subqueries** can be less efficient, as the subquery is executed repeatedly for each row in the outer query, which can be slow for large datasets.

**Summary of Subqueries**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Non-Correlated Subquery** | Independent query; can be executed alone. | WHERE DepartmentID = (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'IT') |
| **Correlated Subquery** | Dependent on outer query; executed for each row. | WHERE Salary > (SELECT AVG(Salary) FROM Employees WHERE DepartmentID = e.DepartmentID) |
| **EXISTS** | Returns TRUE if subquery returns at least one row. | WHERE EXISTS (SELECT 1 FROM Departments WHERE DepartmentID = e.DepartmentID) |
| **IN** | Compares if a value is within a set returned by a subquery. | WHERE DepartmentID IN (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'HR') |

Subqueries are a powerful tool for querying databases, enabling more complex queries and calculations.

**JOINS in DBMS**

A **JOIN** in DBMS is used to combine rows from two or more tables based on a related column between them. Joins allow you to retrieve data from multiple tables in a single query, and they are crucial for working with normalized relational databases.

There are several types of joins, each serving a different purpose. Let’s explore the different types of joins in SQL:

**1. INNER JOIN**

The **INNER JOIN** returns records that have matching values in both tables. If there is no match between the tables, the row will not be included in the result set.

**Syntax:**

SELECT columns

FROM table1

INNER JOIN table2 ON table1.column = table2.column;

**Example:**

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

INNER JOIN Departments

ON Employees.DepartmentID = Departments.DepartmentID;

In this example:

* The query retrieves the names of employees and the department they belong to.
* It only returns employees who belong to a department (i.e., there’s a matching DepartmentID in both the Employees and Departments tables).

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

The **LEFT JOIN** (or **LEFT OUTER JOIN**) returns all records from the left table (table1), and the matched records from the right table (table2). If there is no match, the result is NULL on the side of the right table.

**Syntax:**

SELECT columns

FROM table1

LEFT JOIN table2 ON table1.column = table2.column;

**Example:**

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

LEFT JOIN Departments

ON Employees.DepartmentID = Departments.DepartmentID;

In this example:

* The query retrieves all employees and their department names.
* If an employee is not assigned to any department (i.e., no matching DepartmentID), the DepartmentName will be NULL.

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

The **RIGHT JOIN** (or **RIGHT OUTER JOIN**) returns all records from the right table (table2), and the matched records from the left table (table1). If there is no match, the result is NULL on the side of the left table.

**Syntax:**

SELECT columns

FROM table1

RIGHT JOIN table2 ON table1.column = table2.column;

**Example:**

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

RIGHT JOIN Departments

ON Employees.DepartmentID = Departments.DepartmentID;

In this example:

* The query retrieves all departments and their employee names.
* If a department does not have any employees (i.e., no matching DepartmentID), the EmployeeName will be NULL.

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

The **FULL JOIN** (or **FULL OUTER JOIN**) returns all records when there is a match in either the left (table1) or right (table2) table. It returns NULL for non-matching rows from both tables.

**Syntax:**

SELECT columns

FROM table1

FULL JOIN table2 ON table1.column = table2.column;

**Example:**

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

FULL JOIN Departments

ON Employees.DepartmentID = Departments.DepartmentID;

In this example:

* The query retrieves all employees and all departments.
* If an employee is not assigned to any department, the DepartmentName will be NULL.
* If a department has no employees, the EmployeeName will be NULL.

**5. CROSS JOIN**

The **CROSS JOIN** returns the Cartesian product of the two tables, i.e., it returns all possible combinations of rows from both tables. It does not require a condition to match rows between the tables.

**Syntax:**

SELECT columns

FROM table1

CROSS JOIN table2;

**Example:**

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

CROSS JOIN Departments;

In this example:

* The query retrieves every possible combination of employees and departments.
* If there are 3 employees and 2 departments, the result will contain 6 rows (3 × 2).

**Note:** A **CROSS JOIN** can result in a large number of rows, so it should be used cautiously, especially with large tables.

**6. SELF JOIN**

A **SELF JOIN** is a join where a table is joined with itself. It’s useful when you want to compare rows within the same table.

**Syntax:**

SELECT a.columns, b.columns

FROM table a, table b

WHERE a.column = b.column;

**Example:**

SELECT e1.Name AS EmployeeName, e2.Name AS ManagerName

FROM Employees e1

JOIN Employees e2

ON e1.ManagerID = e2.EmployeeID;

In this example:

* The query retrieves the names of employees along with their manager's name by joining the Employees table with itself.
* e1 represents employees, and e2 represents managers.

**7. JOIN with Multiple Tables**

You can join more than two tables by chaining multiple join conditions together.

**Example:**

SELECT e.Name, d.DepartmentName, p.ProjectName

FROM Employees e

JOIN Departments d ON e.DepartmentID = d.DepartmentID

JOIN Projects p ON e.EmployeeID = p.EmployeeID;

In this example:

* The query retrieves the employee name, department name, and project name by joining the Employees, Departments, and Projects tables.

**8. Using JOIN with WHERE Clause**

You can combine JOIN operations with a WHERE clause to filter the results based on specific conditions.

**Example:**

SELECT e.Name, d.DepartmentName

FROM Employees e

INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID

WHERE d.DepartmentName = 'HR';

In this example:

* The query retrieves all employees from the "HR" department by joining the Employees and Departments tables and filtering on the DepartmentName.

**9. USING Clause in JOIN**

The **USING** clause simplifies writing joins when the columns being joined have the same name in both tables.

**Syntax:**

SELECT columns

FROM table1

JOIN table2 USING (column);

**Example:**

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

INNER JOIN Departments USING (DepartmentID);

In this example:

* The USING clause automatically matches the DepartmentID column from both the Employees and Departments tables.

**10. JOIN vs. UNION**

* **JOIN** is used to combine data from multiple tables based on a relationship between columns.
* **UNION** is used to combine the results of two or more SELECT queries that have the same number of columns and compatible data types.

**Summary of SQL JOIN Types**

| **Join Type** | **Description** | **Result** |
| --- | --- | --- |
| **INNER JOIN** | Returns rows with matching values in both tables. | Only matching rows from both tables. |
| **LEFT JOIN** (OUTER) | Returns all rows from the left table and matching rows from the right table. | All rows from the left table, and matched rows from the right. |
| **RIGHT JOIN** (OUTER) | Returns all rows from the right table and matching rows from the left table. | All rows from the right table, and matched rows from the left. |
| **FULL JOIN** (OUTER) | Returns all rows when there is a match in either table. | All rows from both tables, matched where possible, NULL where not. |
| **CROSS JOIN** | Returns the Cartesian product of both tables. | Every combination of rows from both tables. |
| **SELF JOIN** | Joins a table with itself. | Comparing rows within the same table. |

Joins are a powerful way to query related data from multiple tables in SQL. Each type of join serves a specific use case, and knowing when to use which join is key to writing efficient SQL queries.

**EXIST, ANY, and ALL in SQL**

These are logical operators that help in making conditional queries with subqueries. They are commonly used in combination with subqueries to filter records based on the results of the inner query.

**1. EXISTS**

The **EXISTS** operator is used to check whether a subquery returns any rows. It returns TRUE if the subquery returns at least one row, and FALSE if the subquery returns no rows.

**Syntax:**

SELECT columns

FROM table

WHERE EXISTS (subquery);

**Example:**

SELECT EmployeeID, Name

FROM Employees

WHERE EXISTS (

SELECT 1

FROM Departments

WHERE Departments.DepartmentID = Employees.DepartmentID

);

In this example:

* The query checks if an employee is assigned to a department. If the subquery finds any matching department for an employee, the EXISTS condition will return TRUE, and the employee’s details will be included in the result.
* The 1 inside the subquery is just a placeholder; EXISTS only cares if any rows are returned, not what data is returned.

**2. ANY**

The **ANY** operator is used to compare a value with a set of values returned by a subquery. It returns TRUE if the comparison is true for **any** of the values in the set.

**Syntax:**

SELECT columns

FROM table

WHERE column operator ANY (subquery);

**Example:**

SELECT Name, Salary

FROM Employees

WHERE Salary > ANY (SELECT Salary FROM Employees WHERE DepartmentID = 1);

In this example:

* The query retrieves employees whose salary is greater than **any** of the salaries in the same department (DepartmentID = 1).
* The ANY operator compares the salary of each employee to the set of salaries from the subquery and returns those that satisfy the condition.

**Operators with ANY:**

* You can use operators like =, >, <, >=, <=, != with ANY. It compares the left-hand side value with each value returned by the subquery.

**3. ALL**

The **ALL** operator is used to compare a value to all values returned by a subquery. It returns TRUE if the comparison is true for **all** values in the result set of the subquery.

**Syntax:**

SELECT columns

FROM table

WHERE column operator ALL (subquery);

**Example:**

SELECT Name, Salary

FROM Employees

WHERE Salary > ALL (SELECT Salary FROM Employees WHERE DepartmentID = 1);

In this example:

* The query retrieves employees whose salary is greater than **all** the salaries in the department with DepartmentID = 1.
* The ALL operator ensures that the employee’s salary is greater than each salary returned by the subquery.

**Operators with ALL:**

* Similar to ANY, you can use operators like =, >, <, >=, <=, != with ALL. It compares the left-hand side value with all the values returned by the subquery.

**Key Differences Between EXISTS, ANY, and ALL**

| **Operator** | **Description** | **Returns True When** | **Usage with Subquery** |
| --- | --- | --- | --- |
| **EXISTS** | Checks if the subquery returns any rows. | Returns TRUE if the subquery returns at least one row. | Used with subqueries to test existence of rows. |
| **ANY** | Compares a value with **any** value in the result of the subquery. | Returns TRUE if the condition is met for **any** of the values in the subquery’s result. | Typically used with =, >, <, >=, <=, !=. |
| **ALL** | Compares a value with **all** values in the result of the subquery. | Returns TRUE if the condition is met for **all** values in the subquery’s result. | Typically used with =, >, <, >=, <=, !=. |

**Examples to Illustrate the Differences**

**EXISTS Example:**

SELECT EmployeeID, Name

FROM Employees

WHERE EXISTS (

SELECT 1

FROM Departments

WHERE Employees.DepartmentID = Departments.DepartmentID

);

* This query retrieves employees who are assigned to a department. If the Departments subquery finds a match, the employee will be included in the result.

**ANY Example:**

SELECT Name, Salary

FROM Employees

WHERE Salary > ANY (SELECT Salary FROM Employees WHERE DepartmentID = 1);

* This query retrieves employees whose salary is greater than any salary in DepartmentID = 1.

**ALL Example:**

SELECT Name, Salary

FROM Employees

WHERE Salary > ALL (SELECT Salary FROM Employees WHERE DepartmentID = 1);

* This query retrieves employees whose salary is greater than **all** salaries in DepartmentID = 1.

**Conclusion**

* **EXISTS** is used when you just need to know whether a subquery returns any result.
* **ANY** allows you to compare a value to any one of the values returned by a subquery.
* **ALL** allows you to compare a value to all the values returned by a subquery.

Each of these operators is useful in different situations, depending on the specific query requirements.

**View in SQL**

A **view** in SQL is a virtual table that provides a way to simplify complex queries, enhance security, and manage data more efficiently. A view is essentially a stored SQL query that can be treated like a table, but it does not store data itself. Instead, it provides a way to access data from one or more tables, as defined by the SQL query that creates it.

**Why Use Views?**

* **Simplification**: Views can simplify complex SQL queries by encapsulating complex joins and subqueries.
* **Security**: Views can restrict access to sensitive data by only exposing specific columns or rows.
* **Reusability**: Views allow you to reuse the same SQL query without having to write it every time.
* **Data abstraction**: Views provide an abstraction layer over the base tables, hiding the complexity of data retrieval from the user.

**Types of Views in SQL**

There are mainly two types of views:

1. **Simple View**
2. **Complex View**

Additionally, views can also be classified based on their **updatability**:

* **Updatable View**
* **Non-Updatable View**

Let’s look at each type in detail:

**1. Simple View**

A **simple view** is a view that is based on a single table and does not contain any complex queries like joins, subqueries, or aggregate functions. Simple views are typically used to present data from a single table in a way that simplifies access to frequently used columns.

**Example:**

CREATE VIEW EmployeeNames AS

SELECT Name, DepartmentID

FROM Employees;

* In this example, the EmployeeNames view only selects Name and DepartmentID from the Employees table.
* This is a simple view because it only refers to a single table and doesn’t involve any complex operations.

**2. Complex View**

A **complex view** is a view that is created using multiple tables, joins, subqueries, or aggregate functions. Complex views are used when you need to present data from multiple sources or perform some kind of transformation on the data.

**Example:**

CREATE VIEW EmployeeDepartmentDetails AS

SELECT e.Name, e.Salary, d.DepartmentName

FROM Employees e

JOIN Departments d ON e.DepartmentID = d.DepartmentID;

* This view uses a JOIN between two tables (Employees and Departments), making it a complex view.
* It presents employee names, salaries, and their associated department names.

**3. Updatable View**

An **updatable view** is a view that allows you to perform INSERT, UPDATE, and DELETE operations on the base tables through the view. For a view to be updatable, it must meet certain criteria, such as:

* It must be based on a single table.
* It should not contain aggregate functions or GROUP BY clauses.
* It should not involve complex joins or subqueries that make it impossible to directly map the update to the base table.

**Example:**

CREATE VIEW EmployeeInfo AS

SELECT EmployeeID, Name, DepartmentID

FROM Employees;

* This view is updatable because it is based on a single table (Employees) and does not contain any complex logic.
* You can perform INSERT, UPDATE, and DELETE operations on the EmployeeInfo view, and the changes will be reflected in the Employees table.

**4. Non-Updatable View**

A **non-updatable view** is a view that does not allow direct data modifications (i.e., INSERT, UPDATE, or DELETE) through the view. This can happen for several reasons, including the use of aggregate functions, joins, or subqueries that make it difficult to map changes back to the base tables.

**Example:**

CREATE VIEW EmployeeSalarySummary AS

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY DepartmentID;

* This view is non-updatable because it contains an aggregate function (AVG(Salary)) and a GROUP BY clause.
* Since it aggregates data from multiple rows, it’s not possible to directly update the AvgSalary or the underlying data through the view.

**5. Materialized View**

A **materialized view** (also known as a **snapshot**) is a type of view where the results of the query are stored physically in the database. This contrasts with regular views, which are virtual and don’t store data. Materialized views are typically used to improve query performance when dealing with complex, resource-intensive queries. They need to be refreshed periodically to stay up-to-date.

* **When to use:** Materialized views are useful for storing intermediate results that don’t change frequently, like aggregations or complex joins, so that you can avoid recalculating them each time a query is run.

**Example:**

CREATE MATERIALIZED VIEW EmployeeSalarySummary AS

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY DepartmentID;

* This materialized view stores the result of the query that calculates the average salary per department.
* You can refresh the materialized view periodically to keep it up-to-date.

**6. Indexed View**

An **indexed view** is a type of view where an index is created on the view to improve query performance. Indexed views are particularly useful for improving the speed of data retrieval from complex views, as the index allows for quicker access to the results.

* **When to use:** Indexed views are used when you have a frequently accessed view, and you want to speed up queries that use that view.

**Example:**

CREATE VIEW EmployeeSalaryDetails

WITH SCHEMABINDING AS

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY DepartmentID;

CREATE UNIQUE CLUSTERED INDEX idx\_SalaryDetails ON EmployeeSalaryDetails (DepartmentID);

* In this example:
  + The WITH SCHEMABINDING option ensures that the underlying table cannot be modified in ways that would break the view (e.g., dropping columns).
  + The CREATE INDEX statement creates an index on the DepartmentID column to speed up access to the EmployeeSalaryDetails view.

**Advantages of Using Views**

1. **Simplify Complex Queries**: Views allow users to simplify complex joins and subqueries, making it easier to query the database.
2. **Security**: Views can be used to restrict access to sensitive data by exposing only specific columns or rows.
3. **Consistency**: Views can provide a consistent way of accessing data, even if the underlying schema changes.
4. **Reusability**: You can reuse a view multiple times in different queries, avoiding the need to rewrite complex SQL logic.

**Disadvantages of Using Views**

1. **Performance**: Views, especially complex ones, can sometimes cause performance issues because the underlying query is executed each time the view is referenced.
2. **Limitations**: Some views are non-updatable, meaning you cannot use them to modify data directly.
3. **Storage**: While regular views don’t store data, materialized views and indexed views do take up space in the database.

**Summary of Views Types**

| **View Type** | **Description** | **Example Use Case** |
| --- | --- | --- |
| **Simple View** | Based on a single table, no complex queries. | Displaying data from a single table. |
| **Complex View** | Involves multiple tables, joins, subqueries, or aggregate functions. | Aggregating data or joining multiple tables. |
| **Updatable View** | Allows data modification via the view. | Managing and modifying data directly. |
| **Non-Updatable View** | Does not allow data modification. | Aggregated or computed data that can’t be modified. |
| **Materialized View** | Stores results physically to improve performance. | Speeding up frequent, complex queries. |
| **Indexed View** | View with an index to improve query performance. | Fast retrieval of data from complex views. |

**Transaction Control Commands in SQL**

In SQL, **transaction control** commands are used to manage and control the behavior of transactions within a database. A **transaction** is a logical unit of work that contains one or more SQL statements executed together. A transaction must be atomic, consistent, isolated, and durable (ACID properties) to ensure data integrity and consistency.

Transaction control commands help you manage the start, commit, and rollback of transactions, ensuring that changes to the database are either completely saved or fully undone in case of an error.

**1. COMMIT**

The **COMMIT** command is used to save all the changes made in the current transaction to the database permanently. Once the changes are committed, they cannot be undone.

**Syntax:**

COMMIT;

**Example:**

BEGIN TRANSACTION;

UPDATE Employees

SET Salary = Salary + 5000

WHERE DepartmentID = 3;

COMMIT;

* In this example, the UPDATE statement will increase the salary of employees in department 3. The COMMIT ensures that this change is permanently saved to the database.

**2. ROLLBACK**

The **ROLLBACK** command is used to undo all the changes made in the current transaction. It reverts the database to its state before the transaction started. If a transaction encounters an error or if you need to discard changes, you can use ROLLBACK.

**Syntax:**

ROLLBACK;

**Example:**

BEGIN TRANSACTION;

UPDATE Employees

SET Salary = Salary + 5000

WHERE DepartmentID = 3;

-- Oops, there's a mistake in the update condition.

ROLLBACK;

* In this example, the UPDATE statement is rolled back (undone) because of an error or change of decision.
* After the ROLLBACK, no changes will be saved to the database.

**3. SAVEPOINT**

The **SAVEPOINT** command is used to set a point within a transaction to which you can later roll back. It allows for partial rollback of a transaction, rather than rolling back the entire transaction. You can have multiple savepoints within a single transaction.

**Syntax:**

SAVEPOINT savepoint\_name;

**Example:**

BEGIN TRANSACTION;

UPDATE Employees

SET Salary = Salary + 5000

WHERE DepartmentID = 3;

SAVEPOINT AfterFirstUpdate;

UPDATE Employees

SET Salary = Salary + 2000

WHERE DepartmentID = 4;

-- If we want to undo the second update:

ROLLBACK TO SAVEPOINT AfterFirstUpdate;

COMMIT;

* In this example:
  + A SAVEPOINT is created after the first UPDATE.
  + If there is a mistake or issue with the second update, you can roll back to the savepoint and retain the first UPDATE.

**4. SET TRANSACTION**

The **SET TRANSACTION** command is used to set certain characteristics of a transaction, such as the isolation level. The isolation level controls how the changes made by one transaction are visible to other transactions.

**Syntax:**

SET TRANSACTION [ISOLATION LEVEL isolation\_level];

**Example:**

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

UPDATE Employees

SET Salary = Salary + 5000

WHERE DepartmentID = 3;

COMMIT;

* In this example, the isolation level for the transaction is set to SERIALIZABLE, which ensures the highest level of isolation and prevents other transactions from accessing the data being modified until the transaction is complete.

**5. BEGIN TRANSACTION**

The **BEGIN TRANSACTION** command is used to start a new transaction. In some SQL dialects, this is implicitly done when the first SQL statement in a session is executed, but in others, you explicitly begin a transaction with this command.

**Syntax:**

BEGIN TRANSACTION;

**Example:**

BEGIN TRANSACTION;

UPDATE Employees

SET Salary = Salary + 5000

WHERE DepartmentID = 3;

COMMIT;

* The transaction starts with BEGIN TRANSACTION, and all changes made within the transaction are either committed or rolled back.

**Transaction Control Flow Example**

Let’s combine all the transaction control commands into one example:

BEGIN TRANSACTION;

-- First update operation

UPDATE Employees

SET Salary = Salary + 5000

WHERE DepartmentID = 3;

-- Set a savepoint after the first update

SAVEPOINT AfterFirstUpdate;

-- Second update operation

UPDATE Employees

SET Salary = Salary + 2000

WHERE DepartmentID = 4;

-- Something went wrong, so we rollback to the first savepoint

ROLLBACK TO SAVEPOINT AfterFirstUpdate;

-- Commit the transaction after rolling back the second update

COMMIT;

In this example:

* The transaction starts with BEGIN TRANSACTION.
* The first UPDATE is executed, and a savepoint is created.
* The second UPDATE is executed, but due to some issue, a ROLLBACK TO SAVEPOINT command undoes the second update.
* Finally, the transaction is committed with COMMIT, ensuring that only the first update is saved to the database.

**Summary of Transaction Control Commands**

| **Command** | **Description** | **Usage** |
| --- | --- | --- |
| **COMMIT** | Saves all changes made during the transaction permanently. | Used to finalize changes in a transaction. |
| **ROLLBACK** | Reverts all changes made during the transaction and undoes any modifications. | Used when you want to undo the entire transaction or part of it. |
| **SAVEPOINT** | Creates a point in a transaction to which you can later roll back. | Allows partial rollback within a transaction. |
| **SET TRANSACTION** | Sets the properties of a transaction, such as its isolation level. | Used to set transaction isolation levels (e.g., READ COMMITTED, SERIALIZABLE). |
| **BEGIN TRANSACTION** | Starts a new transaction. | Used to begin a transaction (if not automatically managed). |

**ACID Properties of a Transaction**

1. **Atomicity**: A transaction is atomic, meaning it either completes entirely or has no effect at all.
2. **Consistency**: A transaction brings the database from one valid state to another.
3. **Isolation**: Transactions are isolated from one another. Changes made by one transaction are not visible to others until committed.
4. **Durability**: Once a transaction is committed, its changes are permanent and survive system failures.

These transaction control commands ensure that the database remains in a consistent state even when errors or unexpected events occur.

Let me know if you need further clarification or examples!