CHAPTER 2

SQL Inbuild Functions and Joins

Topics:

1. **Operators Arithmetic, Comparison, Logical SQL functions- Single row function.**
    I. **Single row function.**
    II. **Date functions (add-months, months-between, round, truncate).**
    III. **Numeric Functions (abs, power, mod, round, trunc, sqrt)**
    IV. **Character Functions (initcap, lower, upper, ltrim, rtrim, replace, substring, instr).**
    V. **Conversion Functions (to-char, todate, to-number).**
2. **Groupby, Having and Order by clause.**
3. **Joins: Simple, Equi-join, Nonequi, Self-Joins, Outer-joins.**
4. **Subqueries - Multiple, Correlated.**
5. **Implementation of Queries using SQL Set operators: Union, union all, Intersect, Minus.**

1. Operators Arithmetic, Comparison, Logical SQL functions- Single row function.

=> **1. Operators**

Operators are symbols that perform operations on variables and values.

Types of Operators in SQL

Different types of operators in SQL are:

- Arithmetic operator
- Comparison operator
- Logical operator

Arithmetic operator

Operator	Description
+	The addition is used to perform an addition operation on the data values.
–	This operator is used for the subtraction of the data values.
/	This operator works with the 'ALL' keyword and it calculates division operations.
*	This operator is used for multiplying data values.
%	Modulus is used to get the remainder when data is divided by another.

Comparison operator

Operator	Description
=	Equal to.
>	Greater than.
<	Less than.
>=	Greater than equal to.
<=	Less than equal to.
<>	Not equal to.

Logical operator

Operator	Description
AND	Logical AND compares two Booleans as expressions and returns true when both expressions are true.
OR	Logical OR compares two Booleans as expressions and returns true when one of the expressions is true.
NOT	Not takes a single Boolean as an argument and change its value from false to true or from true to false.

i. SQL Functions - Single Row Functions

=> **Single row functions** in SQL are functions that operate on a single row and return one result for each row. They are often used in SELECT statements to modify, format, or analyse data.

Types of Single Row Functions

1. **Character Functions**: Work with strings/text.
    - UPPER(): Converts text to uppercase.
        - **Example**: SELECT UPPER('hello') FROM DUAL; → **HELLO**
    - LOWER(): Converts text to lowercase.
        - **Example**: SELECT LOWER('HELLO') FROM DUAL; → **hello**
    - SUBSTR(): Extracts a substring.
        - **Example**: SELECT SUBSTR('SQL Functions', 5, 4) FROM DUAL; → **Func** (Starts at position 5, takes 4 characters)
    - LENGTH(): Returns the length of a string.
        - **Example**: SELECT LENGTH('hello') FROM DUAL; → **5**
2. **Number Functions**: Perform calculations on numeric values.
    - ROUND(): Rounds a number to a specified number of decimal places.
        - **Example**: SELECT ROUND(123.456, 2) FROM DUAL; → **123.46**
    - TRUNC(): Truncates a number to a specific number of decimal places without rounding.
        - **Example**: SELECT TRUNC(123.456, 2) FROM DUAL; → **123.45**
    - MOD(): Returns the remainder of a division.
        - **Example**: SELECT MOD(10, 3) FROM DUAL; → **1**

3. **Date Functions**: Work with date values.
    - SYSDATE: Returns the current system date and time.
        - **Example**: SELECT SYSDATE FROM DUAL; → Shows current date/time.
    - MONTHS_BETWEEN(): Calculates the number of months between two dates.
        - **Example**: SELECT MONTHS_BETWEEN('2024-12-01', '2024-01-01') FROM DUAL; → **11**
    - ADD_MONTHS(): Adds a specified number of months to a date.
        - **Example**: SELECT ADD_MONTHS('2024-01-01', 6) FROM DUAL; → **2024-07-01**
4. **Conversion Functions**: Convert data types.
    - TO_CHAR(): Converts a number or date to a string.
        - **Example**: SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD') FROM DUAL; → **2024-09-07**
    - TO_NUMBER(): Converts a string to a number.
        - **Example**: SELECT TO_NUMBER('123.45') FROM DUAL; → **123.45**
    - TO_DATE(): Converts a string to a date.
        - **Example**: SELECT TO_DATE('2024-09-07', 'YYYY-MM-DD') FROM DUAL;

ii. Date functions (add-months, months-between, round, truncate).

=> SQL provides a variety of date functions to work with date and time data. Below are some key date functions, including ADD_MONTHS(), MONTHS_BETWEEN(), ROUND(), and TRUNC(), with examples to help you understand how to use them effectively.

1. ADD_MONTHS()

- **Purpose**: Adds a specific number of months to a date.

Syntax:

ADD_MONTHS(date, number_of_months)

Example:

SELECT ADD_MONTHS('2024-01-15', 6) AS future_date FROM DUAL;

- **Result**: 2024-07-15
- Adds 6 months to January 15, 2024, resulting in July 15, 2024.

2. MONTHS_BETWEEN()

- **Purpose**: Calculates the number of months between two dates.

Syntax:

MONTHS_BETWEEN(date1, date2)

Example:

SELECT MONTHS_BETWEEN('2024-12-01', '2024-01-01') AS months_diff FROM DUAL;

- **Result**: 11
- Calculates the number of months between December 1, 2024, and January 1, 2024.

3. ROUND() (for dates)

- **Purpose: Rounds a date to the nearest day, month, or year.**

Syntax:

ROUND(date, format)

Example:

SELECT ROUND(TO_DATE('2024-09-16', 'YYYY-MM-DD'), 'MM') AS rounded_date FROM DUAL;

- **Result**: 2024-10-01
- Rounds the date 2024-09-16 to the nearest month (October 1st, 2024).

4. TRUNC() (for dates)

- **Purpose**: Truncates (removes) the date to a specific level like day, month, or year, without rounding.

Syntax:

TRUNC(date, format)

Example:

SELECT TRUNC(TO_DATE('2024-09-16', 'YYYY-MM-DD'), 'MM') AS truncated_date FROM DUAL; **Result**: 2024-09-01

iii. Numeric Functions (abs, power, mod, round, trunc, sqrt)

=> **SQL Numeric Functions**

Numeric functions in SQL are used to perform mathematical operations on numeric data. Here's a breakdown of some important numeric functions, explained simply, with examples.

1. ABS() - Absolute Value

- **Purpose**: Returns the absolute (positive) value of a number, ignoring its sign.

  Syntax:

  ABS(number)

  Example:

  SELECT ABS(-10) AS result FROM DUAL;

- **Result**: 10
- Turns -10 into 10, removing the negative sign.

2. POWER() - Exponentiation

- **Purpose**: Raises a number to the power of another number (like x^y).

  Syntax:
  POWER(base, exponent)
  Example:
  SELECT POWER(2, 3) AS result FROM DUAL;
- **Result**: 8
- 2 raised to the power of 3 equals 8 (because 2 × 2 × 2 = 8).

3. MOD() - Modulo (Remainder)

- **Purpose**: Returns the remainder of a division.

  Syntax:
  MOD(dividend, divisor)

  Example:

  SELECT MOD(10, 3) AS result FROM DUAL;

- **Result**: 1
- Dividing 10 by 3 leaves a remainder of 1 (because 10 ÷ 3 = 3 with a remainder of 1).

4. ROUND() - Rounding a Number

- **Purpose**: Rounds a number to a specified number of decimal places.
  Syntax:
  ROUND(number, decimal_places)

  Example:

  SELECT ROUND(123.456, 2) AS result FROM DUAL;

- **Result**: 123.46
- Rounds 123.456 to two decimal places, giving 123.46.

5. TRUNC() - Truncate a Number

- **Purpose:** Truncates (chops off) a number to a specified number of decimal places, without rounding.
  Syntax:
  TRUNC(number, decimal_places)

  Example:

  SELECT TRUNC(123.456, 2) AS result FROM DUAL;

- **Result**: 123.45
- Truncates 123.456 to two decimal places without rounding, leaving 123.45.

6. SQRT() - Square Root

Purpose: Returns the square root of a number.
Syntax:
SQRT(number)

Example:

SELECT SQRT(16) AS result FROM DUAL;

- **Result**: 4
- The square root of 16 is 4 (because 4 × 4 = 16).

iv. Character Functions (initcap, lower, upper, ltrim, rtrim, replace, substring, instr).

=> Character functions in SQL help you manipulate and format string data.

1. INITCAP() - Capitalize First Letter

- **Purpose**: Converts the first letter of each word in a string to uppercase, and all other letters to lowercase.

  Syntax:

  INITCAP(string)

  Example:

  SELECT INITCAP('hello world') AS result FROM DUAL;

  - **Result**: Hello World
  - Capitalizes the first letter of each word.

2. LOWER() - Convert to Lowercase

- **Purpose: Converts all characters in a string to lowercase**

  Syntax:

  LOWER(string)

  Example:

  SELECT LOWER('HELLO WORLD') AS result FROM DUAL;

  - **Result**: hello world
  - Converts all letters to lowercase.

3. UPPER() - Convert to Uppercase

- **Purpose: Converts all characters in a string to uppercase.**

  Syntax:

  UPPER(string)

  Example:

  SELECT UPPER('hello world') AS result FROM DUAL;

  - **Result**: HELLO WORLD
  - Converts all letters to uppercase.

4. LTRIM() - Remove Leading Characters

- **Purpose**: Removes specified characters from the start (left side) of a string.

  Syntax:

  LTRIM(string, trim_characters)

  Example:

  SELECT LTRIM('---hello', '-') AS result FROM DUAL;

  - **Result**: hello
  - Removes leading dashes from the string.

5. RTRIM() - Remove Trailing Characters

- **Purpose: Removes specified characters from the end (right side) of a string.**

  Syntax:

  RTRIM(string, trim_characters)

  Example:

  SELECT RTRIM('hello---', '-') AS result FROM DUAL;

  - **Result**: hello
  - Removes trailing dashes from the string.

6. REPLACE() - Replace Substring

- **Purpose: Replaces occurrences of a substring with another substring.**

  Syntax:

  REPLACE(string, old_substring, new_substring)

  Example:

  SELECT REPLACE('hello world', 'world', 'everyone') AS result FROM DUAL;

  - **Result**: hello everyone
  - Replaces 'world' with 'everyone'.

7. SUBSTRING() - Extract Part of a String

- **Purpose: Extracts a portion of a string.**

  Syntax:

  SUBSTRING(string, start_position, length)

  Example:

  SELECT SUBSTRING('hello world', 1, 5) AS result FROM DUAL;

  - **Result**: hello
  - Extracts the first 5 characters starting from position 1.

8. INSTR() - Find Position of Substring

- **Purpose: Returns the position of the first occurrence of a substring within a string.**

  Syntax:

  INSTR(string, substring)

  Example:

  SELECT INSTR('hello world', 'world') AS result FROM DUAL;

  - **Result**: 7
  - Finds the position where 'world' starts in the string hello world (starting at position 7).

vi. Conversion Functions (to-char, todate, to-number).

=> Conversion functions in SQL are used to change the data type of a value. These functions are especially useful when you need to format or change data types for calculations, comparisons, or formatting.

1. TO_CHAR() - Convert to Character/String

- **Purpose**: Converts a number or date into a string (character) format. This is often used to format numbers or dates in a specific way.

  Syntax:

  SELECT TO_CHAR(12345, '99999') AS result FROM DUAL;

  Example:

  SELECT TO_CHAR(12345, '99999') AS result FROM DUAL;

- **Result**: '12345'
- Converts the number 12345 into a string with the specified format.

2. TO_DATE() - Convert to Date

- **Purpose:** Converts a string into a date format. Useful for interpreting string-based dates as actual dates for calculations and comparisons.

  Syntax:

  TO_DATE(string, format)

  Example:

  SELECT TO_DATE('2024-09-07', 'YYYY-MM-DD') AS result FROM DUAL;

- **Result:** 07-SEP-2024
- Converts the string '2024-09-07' into an actual date type.

3. TO_NUMBER() - Convert to Number

- **Purpose**: Converts a string (or a formatted number in string format) into a numeric value. This is useful for performing mathematical operations on data stored as text.

  Syntax:

  TO_NUMBER(string)

  Example:

  SELECT TO_NUMBER('12345') AS result FROM DUAL;

- **Result: 12345**
- **Converts the string '12345' into the number 12345.**

2. Groupby, Having and Order by clause.

=> These three SQL clauses are crucial for organizing and filtering data in queries.

1. GROUP BY - Grouping Data Together

- **Purpose**: The GROUP BY clause is used to group rows that have the same values in specified columns. It's often used with aggregate functions like COUNT(), SUM(), AVG(), etc., to perform calculations on each group of data.

  Syntax:

  SELECT column_name, AGGREGATE_FUNCTION(column_name)

  FROM table_name

  GROUP BY column_name;

  Example:

  SELECT department, COUNT(employee_id)

  FROM employees

  GROUP BY department;

2. HAVING - Filtering Groups (After Grouping)

- **Purpose**: The HAVING clause is used to filter data after it has been grouped by the GROUP BY clause. It's similar to the WHERE clause, but WHERE is applied before grouping, and HAVING is applied after.

  Syntax:

  SELECT column_name, AGGREGATE_FUNCTION(column_name)

  FROM table_name

  GROUP BY column_name

  HAVING AGGREGATE_FUNCTION(column_name) condition;

  Example:

  SELECT department, COUNT(employee_id)

  FROM employees

  GROUP BY department

  HAVING COUNT(employee_id) > 10;

3. ORDER BY - Sorting Data

- **Purpose**: The ORDER BY clause is used to sort the result set by one or more columns. You can sort in ascending order (default) or descending order.

  Syntax:

  SELECT column_name

  FROM table_name

  ORDER BY column_name [ASC|DESC];

  Example:

  SELECT employee_name, salary

  FROM employees

  ORDER BY salary DESC;

3. Joins: Simple, Equi-join, Nonequi, Self-Joins, Outer-joins.

=> SQL joins are used to combine rows from two or more tables based on a related column.

1. Simple Join (INNER JOIN)

- **Purpose**: A simple or INNER JOIN returns only the rows where there is a match between two tables based on a common column.

  Syntax:

  SELECT columns

  FROM table1

  INNER JOIN table2

  ON table1.common_column = table2.common_column;

  Example:

  SELECT employees.name, departments.name

  FROM employees

  INNER JOIN departments

  ON employees.department_id = departments.department_id;

2. Equi-Join

- **Purpose**: An Equi-Join is just a specific type of INNER JOIN where the condition is based on equality (=). It matches rows between two tables where values in specific columns are equal.

  Syntax and Example:

  SELECT employees.name, departments.name

  FROM employees, departments

  WHERE employees.department_id = departments.department_id;

3. Non-Equi Join

- **Purpose**: A Non-Equi Join is used when you want to join tables based on a condition that is not an equality (=), such as <, >, !=, etc.

  Syntax:

  SELECT columns

  FROM table1

  JOIN table2

  ON table1.column > table2.column;

  Example:

  SELECT employees.name, salaries.amount

  FROM employees

  JOIN salaries

  ON employees.salary_id < salaries.salary_id;

4. Self-Join

- **Purpose**: A Self-Join is when a table is joined with itself. This is often used when you want to compare rows within the same table.

  Syntax:

  SELECT A.column, B.column

  FROM table_name A, table_name B

  WHERE A.column = B.column;

Example:

```
SELECT e1.name AS employee, e2.name AS manager
FROM employees e1
JOIN employees e2
ON e1.manager_id = e2.employee_id;
```

5. Outer Joins

There are three types of outer joins, and they return rows even when there is no match between the two tables.

a. LEFT OUTER JOIN (LEFT JOIN)

- **Purpose**: Returns all rows from the left table and the matched rows from the right table. If no match is found, the result is NULL for columns from the right table.

  Syntax:

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

  Example:

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

b. RIGHT OUTER JOIN (RIGHT JOIN)

- **Purpose**: Returns all rows from the right table and the matched rows from the left table. If no match is found, the result is NULL for columns from the left table.

  Syntax:

  SELECT columns

  FROM table1

  RIGHT JOIN table2

  ON table1.column = table2.column;

  Example:

  SELECT employees.name, departments.name

  FROM employees

  RIGHT JOIN departments

  ON employees.department_id = departments.department_id;

c. FULL OUTER JOIN

- **Purpose**: Returns all rows when there is a match in either table. Rows from both tables are included, with NULL in places where no match exists.

  Syntax:

  SELECT columns

  FROM table1

  FULL JOIN table2

  ON table1.column = table2.column;

  Example:

  SELECT employees.name, departments.name

  FROM employees

  FULL JOIN departments

  ON employees.department_id = departments.department_id;

4. Subqueries - Multiple, Correlated.

=> Subqueries are queries within other queries, often used to break down complex logic into simpler steps. There are two main types of subqueries: **Multiple Subqueries** and **Correlated Subqueries**.

1. Multiple Subqueries (Nested Subqueries)

- **Purpose**: A multiple subquery, also called a **nested subquery**, is a query inside another query. The inner query runs first, and its result is used by the outer query.

  Syntax:

  SELECT column_name

  FROM table_name

  WHERE column_name = (SELECT column_name FROM another_table
  WHERE condition);

  Example:

  SELECT name

  FROM employees

  WHERE salary > (SELECT AVG(salary) FROM employees);

- **Single-row subquery**: Returns one value (e.g., using =, <, >, etc.).
- **Multi-row subquery**: Returns multiple rows (e.g., using IN, ANY, ALL).

Example of Multi-row Subquery:

SELECT name

FROM employees

WHERE department_id IN (SELECT department_id FROM departments WHERE location = 'New York');

2. Correlated Subqueries

- **Purpose:** A correlated subquery is a subquery that depends on the outer query for its values. It is executed for each row processed by the outer query, meaning the inner query runs multiple times (once for each row of the outer query).
  Syntax:
  SELECT column_name
  FROM table_name outer
  WHERE column_name operator (SELECT column_name FROM table_name inner WHERE outer.column_name = inner.column_name);
  Example:
  SELECT e1.name, e1.salary
  FROM employees e1
  WHERE e1.salary > (SELECT AVG(e2.salary) FROM employees e2 WHERE e2.department_id = e1.department_id);

5. Implementation of Queries using SQL Set operators: Union, union all, Intersect, Minus.

=> SQL Set operators allow you to combine the results of two or more SELECT queries into a single result set. Each operator has different behaviours regarding duplicates and the way it handles the combination of results. Here's a simple and easy-to-remember guide to the four key SQL set operators: **UNION**, **UNION ALL**, **INTERSECT**, and **MINUS**.

1. UNION

- **Purpose**: Combines the results of two or more SELECT queries and **removes duplicate rows**.

  Syntax:

  SELECT column_name FROM table1

  UNION

  SELECT column_name FROM table2;

  Example:

  SELECT name FROM employees

  UNION

  SELECT name FROM customers;

2. UNION ALL

- **Purpose**: Combines the results of two or more SELECT queries and **keeps all rows**, including duplicates.

  Syntax:

  SELECT column_name FROM table1

  UNION ALL

  SELECT column_name FROM table2;

  Example:

  SELECT name FROM employees

  UNION ALL

  SELECT name FROM customers;

3. INTERSECT

- **Purpose**: Returns only the rows that are **common** between two or more SELECT queries. In other words, it returns the intersection of the result sets.

  Syntax:

  SELECT column_name FROM table1

  INTERSECT

  SELECT column_name FROM table2;

  Example:

  SELECT name FROM employees

  INTERSECT

  SELECT name FROM customers;

4. MINUS (or EXCEPT in some SQL systems)

- **Purpose**: Returns the rows from the first SELECT query that are **not present** in the second SELECT query.

  Syntax:

  SELECT column_name FROM table1

  MINUS

  SELECT column_name FROM table2;

  Example:

  SELECT name FROM employees

  MINUS

  SELECT name FROM customers;

Key Differences:

Operator	Purpose	Duplicates	Common Use
UNION	Combines results, removes duplicates	Duplicates removed	Get a unique combined result set
UNION ALL	Combines results, keeps all rows (including duplicates)	Duplicates kept	Get all results, including duplicates
INTERSECT	Returns only rows common to both queries	No duplicates in common rows	Find common rows between two result sets
MINUS	Returns rows from the first query not in the second	No duplicates in the result set	Get the difference between two result sets