#PostgreSQL#

Section 15. Conditional Expressions & Operators
CASE – show you how to form conditional queries with CASE expression.
COALESCE – return the first non-null argument. You can use it to substitute NULL by a default value.
NULLIF – return NULL if the first argument equals the second one.
CAST – convert from one data type into another e.g., from a string into an integer, from a string into a date.

Can You Teach Me above all the topics with example and show output also(create a simple table and insert some values, Also Explain clear cut explanation and Explain why this needed)

Certainly! Let's go through each of these PostgreSQL conditional expressions and operators with examples, explanations, and outputs. We'll also create a simple table, insert some values, and show how these expressions work.

1. CASE Expression

The `CASE` expression is used for conditional queries, allowing you to handle different situations depending on the values of columns or variables. It allows you to replace values with other values based on specified conditions.

Example:

```sql


-- Create a simple table
CREATE TABLE employee (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    salary INTEGER,
    department VARCHAR(50)
);

-- Insert some values into the table
INSERT INTO employee (name, salary, department) VALUES
('Alice', 50000, 'IT'),
('Bob', 30000, 'HR'),
('Charlie', 60000, 'Finance'),
('David', NULL, 'IT');

-- Use CASE expression to categorize salary
SELECT name,
       CASE
           WHEN salary IS NULL THEN 'No Salary Data'
           WHEN salary < 40000 THEN 'Low Salary'
           WHEN salary BETWEEN 40000 AND 60000 THEN 'Average Salary'
           ELSE 'High Salary'
       END AS salary_category
FROM employee;
```

Output:

```
lua


  |   name    | salary_category |
  |-----------|-----------------|
  | Alice     | Average Salary   |
  | Bob       | Low Salary       |
  | Charlie   | High Salary      |
  | David     | No Salary Data   |
```

Explanation:

- The `CASE` expression here categorizes employees based on their salary:
  - If `salary` is `NULL`, it returns 'No Salary Data'.
  - If `salary` is less than 40,000, it returns 'Low Salary'.
  - If `salary` is between 40,000 and 60,000, it returns 'Average Salary'.
  - For salaries above 60,000, it returns 'High Salary'.

2. COALESCE

The `COALESCE` function returns the first non-null value from its arguments. It can be used to substitute `NULL` values with a default value.

Example:

```
sql


  -- Create a simple table
  CREATE TABLE product (
      id SERIAL PRIMARY KEY,
      name VARCHAR(50),
      price DECIMAL(10, 2),
      discount DECIMAL(5, 2)
  );

  -- Insert some values into the table
  INSERT INTO product (name, price, discount) VALUES
  ('Product A', 100.00, NULL),
  ('Product B', NULL, 15.00),
  ('Product C', 250.00, 20.00);

  -- Use COALESCE to substitute NULL values
  SELECT name,
         COALESCE(price, 0) AS price_with_default,
         COALESCE(discount, 10.00) AS discount_with_default
  FROM product;
```

Output:

```
mathematica


  |   name    | price_with_default | discount_with_default |
  |-----------|--------------------|-----------------------|
  | Product A | 100.00             | 10.00                  |
  | Product B | 0.00               | 15.00                  |
  | Product C | 250.00             | 20.00                  |
```

Explanation:

- The `COALESCE` function replaces `NULL` values with a specified default:
  - For `price`, if it's `NULL`, it returns `0`.
  - For `discount`, if it's `NULL`, it defaults to `10.00`.

3. NULLIF

The `NULLIF` function returns `NULL` if the two given arguments are equal; otherwise, it returns the first argument. This can be useful when you want to treat certain values as `NULL` when they match a specific value.

Example:

```
sql


  -- Create a simple table
  CREATE TABLE student (
      id SERIAL PRIMARY KEY,
      name VARCHAR(50),
      grade INTEGER
  );

  -- Insert some values into the table
  INSERT INTO student (name, grade) VALUES
  ('Emily', 80),
  ('Michael', NULL),
  ('Sarah', 100),
  ('John', 80);

  -- Use NULLIF to treat matching values as NULL
  SELECT name,
         NULLIF(grade, 80) AS grade_without_match
  FROM student;
```

Output:

```
sql


  |    name   | grade_without_match |
  |-----------|---------------------|
  |  Emily    | NULL                |
  | Michael   | NULL                |
  |  Sarah    | 100                 |
  |   John    | NULL                |
```

Explanation:

- The `NULLIF` function checks if the `grade` is equal to `80`:
  - For `Emily` and `John`, who have a `grade` of `80`, it returns `NULL`.
  - For `Michael`, who has no `grade` (`NULL`), it returns `NULL` as well.
  - For `Sarah`, who has a `grade` of `100`, it returns `100`.

4. CAST

The `CAST` function converts a value from one data type to another. This is useful when you need to manipulate data that has a different data type than required by the query.

Example:

```sql
-- Create a simple table
CREATE TABLE sales (
    id SERIAL PRIMARY KEY,
    sale_date VARCHAR(50),
    amount VARCHAR(50)
);

-- Insert some values into the table
INSERT INTO sales (sale_date, amount) VALUES
('2024-12-12', '1500'),
('2024-12-13', '2500.75'),
('2024-12-14', NULL),
('2024-12-15', '3200');

-- Use CAST to convert string to date and amount to numeric
SELECT sale_date,
       CAST(sale_date AS DATE) AS date,
       CAST(amount AS DECIMAL(10, 2)) AS numeric_amount
FROM sales;
```

Output:

```yaml
| sale_date | date       | numeric_amount |
|-----------|------------|----------------|
| 2024-12-12 | 2024-12-12 | 1500.00        |
| 2024-12-13 | 2024-12-13 | 2500.75        |
| 2024-12-14 | NULL       | NULL           |
| 2024-12-15 | 2024-12-15 | 3200.00        |
```

Explanation:

- The `**CAST**` function is used to convert `**sale_date**` from a `**VARCHAR**` to a `**DATE**`.
- It also converts `**amount**` from a `**VARCHAR**` to a `**DECIMAL(10, 2)**`, allowing for precise numeric manipulation.

These expressions are needed to handle specific scenarios in queries, ensuring data consistency and clarity when dealing with potentially missing or inconsistent values. They make it easier to manage data and perform operations that consider specific conditions or defaults.