PostgreSQL Regular Expressions

Basics of Regular Expressions:

Regular expressions (regex) are patterns used to match text. PostgreSQL supports regex operations to perform complex pattern matching.

Regex Syntax:

- `**.**`: Matches any single character except a newline.
- `**[FGz]**`: Matches any single character within the brackets (F, G, or z).
- `**[a-z]**`: Matches any single lowercase letter from a to z.
- `**[^a-z]**`: Matches any character that is not a lowercase letter.
- `**\w**`: Matches any word character (alphanumeric and underscore), equivalent to `[A-Za-z0-9_]`.
- `**\d**`: Matches any digit.
- `**\t**`: Matches a tab character.
- `**\s**`: Matches any whitespace character.
- `**\n**`: Matches a newline character.
- `**\r**`: Matches a carriage return character.
- `**^**`: Matches the start of a string.
- `**$**`: Matches the end of a string.
- `**?**`: Matches the preceding element zero or one time.
- `***`: Matches the preceding element zero or more times.
- `**+**`: Matches the preceding element one or more times.
- `**{m}**`: Matches the preceding element exactly m times.
- `**{m,n}**`: Matches the preceding element between m and n times.
- `**a|b**`: Matches either 'a' or 'b'.
- `**( )**`: Groups patterns and captures the match.
- `**(?: )**`: Groups patterns without capturing.

SIMILAR TO Operator:

The `**SIMILAR TO**` operator allows pattern matching similar to regex but using SQL standard syntax:

Example Queries:

SELECT 'same' SIMILAR TO 'same'; -> true

SELECT 'same' SIMILAR TO 'Same'; -> false

SELECT 'same' SIMILAR TO 's'; -> false

SELECT 'same' SIMILAR TO 's%'; -> true

SELECT 'same' SIMILAR TO 'sa%'; -> true

SELECT 'same' SIMILAR TO '%(s|a)%'; -> true

SELECT 'same' SIMILAR TO '(m|e)%'; -> false

POSIX Regular Expressions:

PostgreSQL supports **POSIX** regular expressions, which offer powerful pattern matching capabilities.

POSIX Operators:

- `~`: Matches the regular expression (case sensitive).
- `~*`: Matches the regular expression (case insensitive).
- `!~`: Does not match the regular expression (case sensitive).
- `!~*`: Does not match the regular expression (case insensitive).

Example Queries:

SELECT 'same' ~ 'same'; -> true

SELECT 'same' ~ 'Same'; -> false

SELECT 'same' ~* 'Same'; -> true

SELECT 'same' !~ 'Same'; -> true

SELECT 'same' !~* 'Same'; -> false

Using `substring` with POSIX Regex:

The `**substring**` function can extract parts of a string based on regex patterns.

Example Queries:

- Single character (matches the first character)

  SELECT substring('The event will start at 8 p.m on Dec 19, 2023.' FROM '.'); -- T

- All characters (matches the entire string)

SELECT substring('The event will start at 8 p.m on Dec 19, 2023.' FROM '.*'); -- The event will start at 8 p.m on Dec 19, 2023.

- Any character after 'event'

SELECT substring('The event will start at 8 p.m on Dec 19, 2023.' FROM 'event.+'); -- event will start at 8 p.m on Dec 19, 2023.

- One or more characters from the start

  SELECT substring('The event will start at 8 p.m on Dec 19, 2023.' FROM '\w+'); -- The

- One or more characters from the end

SELECT substring('The event will start at 8 p.m on Dec 19, 2023.' FROM '\w+$'); -- 2023

- Two digits (e.g., day or year)

SELECT substring('The event will start at 8 p.m on Dec 19, 2023.' FROM '\d{2}'); -- 19

REGEXP_MATCHES Function:

The `**REGEXP_MATCHES**` function in PostgreSQL is used to find matches of a regular expression pattern within a source string. It can return matched substrings based on the specified flags.

Syntax:

REGEXP_MATCHES(source_string, pattern [, flags])

- **source_string**: The string to be searched.
- **pattern**: The regular expression pattern to match against the source string.
- **flags (optional):** Modifiers that control the matching behavior.
  Available flags include:

  g: Performs a global search, returning all matches in the source string.

  i: Makes the search case-insensitive.

Examples Queries:

Global Search: Finds all occurrences of the pattern in the source string.

SELECT REGEXP_MATCHES('Amazing #PostgreSQL #SQL', '#[A-Za-z0-9_]+', 'g'); -- Returns: {#PostgreSQL, #SQL}

Case Insensitivity: Makes the pattern matching case-insensitive.

SELECT REGEXP_MATCHES('Amazing #PostgreSQL', '#[A-Za-z0-9_]+', 'i'); -- Returns: {#PostgreSQL}

SELECT REGEXP_MATCHES('XYZ', '^(X)(..)$', 'g'); -- Result: {X, YZ}

SELECT REGEXP_MATCHES('My cat always jumps around', 'cat|dog', 'g'); -- Result: {cat}

You can combine these flags as needed to customize your search.

REGEXP_REPLACE Function:

The `**REGEXP_REPLACE**` function in PostgreSQL is used to replace parts of a string that match a regular expression pattern with a specified replacement string. This function allows for complex string transformations based on pattern matching.

Syntax:

REGEXP_REPLACE(source_string, pattern, replacement_string [, flags])

- **source_string**: The string to be searched and modified.
- **pattern**: The regular expression pattern that identifies the substring to be replaced.
- **replacement_string**: The string that will replace the matched substrings.
- **flags (optional):** Modifiers that alter the behavior of the replacement.
  Common flags include:

  g: Global search; replaces all occurrences of the pattern.

  i: Case-insensitive search.

Examples Queries:

1. Reverse Word Order:

SELECT REGEXP_REPLACE('Najeeb Mohd', '(.*) (.*)', '\2 \1'); --Result: Mohd Najeeb

Explanation: This query reverses the order of two words in the string. The pattern `(.*) (.*)` captures two groups of characters separated by a space. The replacement string `\2 \1` swaps these groups.

2. Remove All Digits:

SELECT REGEXP_REPLACE('ABCD12345xyz', '[[:digit:]]', '', 'g'); --Result: `ABCDxyz`

Explanation: This query removes all digits from the string. The pattern `[[:digit:]]` matches any digit, and the replacement string is an empty string. The `g` flag ensures all occurrences are replaced.

3. Remove All Letters:

SELECT REGEXP_REPLACE('ABCD12345xyz', '[[:alpha:]]', '', 'g'); --Result: `12345`

Explanation: This query removes all alphabetic characters from the string. The pattern `[[:alpha:]]` matches any letter, and the replacement string is an empty string. The `g` flag ensures all occurrences are replaced.

4. Update Date Year:

SELECT REGEXP_REPLACE('2023-12-19', '\d{4}', '2024'); --Result: `2024-12-19`

Explanation: This query changes the year in a date string from `2023` to `2024`. The pattern `\d{4}` matches a four-digit year, and the replacement string is `2024`.

In summary, `**REGEXP_REPLACE**` is a powerful function for performing search-and-replace operations using regular expressions, allowing for flexible and complex text manipulations.

REGEXP_SPLIT_TO_TABLE Function:

The `**REGEXP_SPLIT_TO_TABLE**` function in PostgreSQL is used to split a string into multiple rows based on a regular expression pattern. This function is particularly useful when you need to break down a delimited string into individual components and process each component separately.

Syntax:

REGEXP_SPLIT_TO_TABLE(source_string, pattern)

source_string: The string to be split.

pattern: The regular expression pattern used to determine the splitting points.

Examples Queries:

1. Split a Comma-Separated List into Rows:
   SELECT REGEXP_SPLIT_TO_TABLE('1,2,3,4', ',');
   Result:
   1
   2
   3
   4
   Explanation: This query splits the string `'1,2,3,4'` into separate rows using a comma as the delimiter.

2. Split a CSV String into Rows:
   SELECT REGEXP_SPLIT_TO_TABLE('Najeeb,Uddin,Mohd', ',');
   Result:
   Najeeb
   Uddin
   Mohd
   Explanation: This query splits the string `'Najeeb,Uddin,Mohd'` into separate rows using a comma as the delimiter.

3. Split a String by Spaces:
   SELECT REGEXP_SPLIT_TO_TABLE('Najeeb Uddin Mohd', ' ');
   Result:
   Najeeb
   Uddin
   Mohd
   Explanation: This query splits the string `'Najeeb Uddin Mohd'` into separate rows using a space as the delimiter.

Key Points:

Output: Each part of the split string is returned as a separate row.

Delimiter Pattern: The pattern can be any regular expression that defines how the string should be split. For example, you can use multiple delimiters or whitespace characters.

Use Case: Ideal for converting delimited strings into rows for further processing or analysis.

In summary, `**REGEXP_SPLIT_TO_TABLE**` is a powerful function for breaking down strings based on complex patterns, providing a straightforward way to handle and analyze delimited data in PostgreSQL.

REGEXP_SPLIT_TO_ARRAY Function:

The `**REGEXP_SPLIT_TO_ARRAY**` function in PostgreSQL splits a string into an array of substrings based on a regular expression pattern. This function is useful for converting a delimited string into an array format, which can then be processed or manipulated as needed.

Syntax:

REGEXP_SPLIT_TO_ARRAY(source_string, pattern)

source_string: The string to be split.

pattern: The regular expression pattern used to determine the splitting points.

Examples Queries:

1. Split a Comma-Separated List into an Array:
   SELECT REGEXP_SPLIT_TO_ARRAY('1,2,3,4', ',');
   Result:
   {1,2,3,4}
   Explanation: This query splits the string `'1,2,3,4'` into an array using a comma as the delimiter.

2. Split a CSV String into an Array:
   SELECT REGEXP_SPLIT_TO_ARRAY('Najeeb,Uddin,Mohd', ',');
   Result:
   {Najeeb,Uddin,Mohd}
   Explanation: This query splits the string `'Najeeb,Uddin,Mohd'` into an array using a comma as the delimiter.
3. Split a String by Spaces:
   SELECT REGEXP_SPLIT_TO_ARRAY('Najeeb Uddin Mohd', ' ');
   Result:
   {Najeeb,Uddin,Mohd}
   Explanation: This query splits the string `'Najeeb Uddin Mohd'` into an array using a space as the delimiter.

Key Points:

Output: The function returns an array of text values, where each element is a substring from the original string.

Delimiter Pattern: The pattern can be any regular expression that defines how the string should be split. This includes single or multiple delimiters.

Use Case: Useful for converting delimited strings into an array format for further analysis or manipulation.

In summary, `**REGEXP_SPLIT_TO_ARRAY**` is a versatile function that allows you to transform delimited text into an array, making it easier to handle and process each substring separately in PostgreSQL.