9.7. Pattern Matching

There are three separate approaches to pattern matching provided by PostgreSQL: the traditional SQL `LIKE` operator, the more recent `SIMILAR TO` operator (added in SQL:1999), and POSIX-style regular expressions. Aside from the basic "does this string match this pattern?" operators, functions are available to extract or replace matching substrings and to split a string at matching locations.

> **Tip**
>
> If you have pattern matching needs that go beyond this, consider writing a user-defined function in Perl or Tcl.

> **Caution**
>
> While most regular-expression searches can be executed very quickly, regular expressions can be contrived that take arbitrary amounts of time and memory to process. Be wary of accepting regular-expression search patterns from hostile sources. If you must do so, it is advisable to impose a statement timeout.
>
> Searches using `SIMILAR TO` patterns have the same security hazards, since `SIMILAR TO` provides many of the same capabilities as POSIX-style regular expressions.
>
> `LIKE` searches, being much simpler than the other two options, are safer to use with possibly-hostile pattern sources.

The pattern matching operators of all three kinds do not support nondeterministic collations. If required, apply a different collation to the expression to work around this limitation.

9.7.1. LIKE

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

The `LIKE` expression returns true if the ***string*** matches the supplied ***pattern***. (As expected, the `NOT LIKE` expression returns false if `LIKE` returns true, and vice versa. An equivalent expression is `NOT (`***string*** `LIKE` ***pattern***`)`.)

If ***pattern*** does not contain percent signs or underscores, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in ***pattern*** stands for (matches) any single character; a percent sign (%) matches any sequence of zero or more characters.

Some examples:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

`LIKE` pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in ***pattern*** must be preceded by the escape character. The default escape character is the backslash but a different one can be selected by using the `ESCAPE` clause. To match the escape character itself, write two escape characters.

> **Note**
>
> If you have **standard_conforming_strings** turned off, any backslashes you write in literal string constants will need to be doubled. See **Section 4.1.2.1** for more information.

It's also possible to select no escape character by writing `ESCAPE ''`. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

According to the SQL standard, omitting `ESCAPE` means there is no escape character (rather than defaulting to a backslash), and a zero-length `ESCAPE` value is disallowed. PostgreSQL's behavior in this regard is therefore slightly nonstandard.

The key word `ILIKE` can be used instead of `LIKE` to make the match case-insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator `~~` is equivalent to `LIKE`, and `~~*` corresponds to `ILIKE`. There are also `!~~` and `!~~*` operators that represent `NOT LIKE` and `NOT ILIKE`, respectively. All of these operators are PostgreSQL-specific. You may see these operator names in `EXPLAIN` output and similar places, since the parser actually translates `LIKE` et al. to these operators.

The phrases `LIKE`, `ILIKE`, `NOT LIKE`, and `NOT ILIKE` are generally treated as operators in PostgreSQL syntax; for example they can be used in ***expression operator*** ANY (***subquery***) constructs, although an `ESCAPE` clause cannot be included there. In some obscure cases it may be necessary to use the underlying operator names instead.

Also see the starts-with operator `^@` and the corresponding `starts_with()` function, which are useful in cases where simply matching the beginning of a string is needed.

9.7.2. `SIMILAR TO` Regular Expressions

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

The `SIMILAR TO` operator returns true or false depending on whether its pattern matches the given string. It is similar to `LIKE`, except that it interprets the pattern using the SQL standard's definition of a regular expression. SQL regular expressions are a curious cross between `LIKE` notation and common (POSIX) regular expression notation.

Like `LIKE`, the `SIMILAR TO` operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string. Also like `LIKE`, `SIMILAR TO` uses `_` and `%` as wildcard characters denoting any single character and any string, respectively (these are comparable to `.` and `.*` in POSIX regular expressions).

In addition to these facilities borrowed from `LIKE`, `SIMILAR TO` supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

- `|` denotes alternation (either of two alternatives).
- `*` denotes repetition of the previous item zero or more times.
- `+` denotes repetition of the previous item one or more times.
- `?` denotes repetition of the previous item zero or one time.
- `{`***m***`}` denotes repetition of the previous item exactly ***m*** times.
- `{`***m***`,}` denotes repetition of the previous item ***m*** or more times.
- `{`***m***`,`***n***`}` denotes repetition of the previous item at least ***m*** and not more than ***n*** times.
- Parentheses `()` can be used to group items into a single logical item.
- A bracket expression `[...]` specifies a character class, just as in POSIX regular expressions.

Notice that the period (`.`) is not a metacharacter for `SIMILAR TO`.

As with `LIKE`, a backslash disables the special meaning of any of these metacharacters. A different escape character can be specified with `ESCAPE`, or the escape capability can be disabled by writing `ESCAPE ''`.

According to the SQL standard, omitting `ESCAPE` means there is no escape character (rather than defaulting to a backslash), and a zero-length `ESCAPE` value is disallowed. PostgreSQL's behavior in this regard is therefore slightly nonstandard.

Another nonstandard extension is that following the escape character with a letter or digit provides access to the escape sequences defined for POSIX regular expressions; see **Table 9.20**, **Table 9.21**, and **Table 9.22** below.

Some examples:

```
'abc' SIMILAR TO 'abc'          true
'abc' SIMILAR TO 'a'            false
'abc' SIMILAR TO '%(b|d)%'      true
'abc' SIMILAR TO '(b|c)%'       false
'-abc-' SIMILAR TO '%\mabc\M%'  true
'xabcy' SIMILAR TO '%\mabc\M%'  false
```

The `substring` function with three parameters provides extraction of a substring that matches an SQL regular expression pattern. The function can be written according to standard SQL syntax:

```
substring(string similar pattern escape escape-character)
```

or using the now obsolete SQL:1999 syntax:

```
substring(string from pattern for escape-character)
```

or as a plain three-argument function:

```
substring(string, pattern, escape-character)
```

As with `SIMILAR TO`, the specified pattern must match the entire data string, or else the function fails and returns null. To indicate the part of the pattern for which the matching data sub-string is of interest, the pattern should contain two occurrences of the escape character followed by a double quote ("). The text matching the portion of the pattern between these separators is returned when the match is successful.

The escape-double-quote separators actually divide `substring`'s pattern into three independent regular expressions; for example, a vertical bar (|) in any of the three sections affects only that section. Also, the first and third of these regular expressions are defined to match the smallest possible amount of text, not the largest, when there is any ambiguity about how much of the data string matches which pattern. (In POSIX parlance, the first and third regular expressions are forced to be non-greedy.)

As an extension to the SQL standard, PostgreSQL allows there to be just one escape-double-quote separator, in which case the third regular expression is taken as empty; or no separators, in which case the first and third regular expressions are taken as empty.

Some examples, with #" delimiting the return string:

```
substring('foobar' similar '%#"o_b#"%' escape '#')   oob
substring('foobar' similar '#"o_b#"%' escape '#')    NULL
```

9.7.3. POSIX Regular Expressions

Table 9.16 lists the available operators for pattern matching using POSIX regular expressions.

Table 9.16. Regular Expression Match Operators

Operator Description Example(s)

`text ~ text → boolean`
String matches regular expression, case sensitively
`'thomas' ~ 't.*ma' → t`

`text ~* text → boolean`
String matches regular expression, case insensitively
`'thomas' ~* 'T.*ma' → t`

`text !~ text → boolean`
String does not match regular expression, case sensitively
`'thomas' !~ 't.*max' → t`

`text !~* text → boolean`
String does not match regular expression, case insensitively
`'thomas' !~* 'T.*ma' → f`

POSIX regular expressions provide a more powerful means for pattern matching than the `LIKE` and `SIMILAR TO` operators. Many Unix tools such as `egrep`, `sed`, or `awk` use a pattern matching language that is similar to the one described here.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a *regular set*). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with `LIKE`, pattern characters match string characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than `LIKE` does. Unlike `LIKE` patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Some examples:

```
'abcd' ~ 'bc'     true
'abcd' ~ 'a.c'    true — dot matches any character
'abcd' ~ 'a.*d'   true — * repeats the preceding pattern item
'abcd' ~ '(b|x)'  true — | means OR, parentheses group
'abcd' ~ '^a'     true — ^ anchors to start of string
'abcd' ~ '^(b|c)' false — would match except for anchoring
```

The POSIX pattern language is described in much greater detail below.

The `substring` function with two parameters, `substring(`***`string`*** `from` ***`pattern`***`)`, provides extraction of a substring that matches a POSIX regular expression pattern. It returns null if there is no match, otherwise the first portion of the text that matched the pattern. But if the pattern contains any parentheses, the portion of the text that matched the first parenthesized subexpression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception. If you need parentheses in the pattern before the subexpression you want to extract, see the non-capturing parentheses described below.

Some examples:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

The `regexp_count` function counts the number of places where a POSIX regular expression pattern matches a string. It has the syntax `regexp_count(`***`string`***`,` ***`pattern`*** `[,` ***`start`*** `[,` ***`flags`*** `]])`. ***`pattern`*** is searched for in ***`string`***, normally from the beginning of the string, but if the ***`start`*** parameter is provided then beginning from that character index. The ***`flags`*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. For example, including `i` in ***`flags`*** specifies case-insensitive matching. Supported flags are described in Table 9.24.

Some examples:

```
regexp_count('ABCABCAXYaxy', 'A.')          3
regexp_count('ABCABCAXYaxy', 'A.', 1, 'i')  4
```

The `regexp_instr` function returns the starting or ending position of the ***`N`***th match of a POSIX regular expression pattern to a string, or zero if there is no such match. It has the syntax `regexp_instr(`***`string`***`,` ***`pattern`*** `[,` ***`start`*** `[,` ***`N`*** `[,` ***`endoption`*** `[,` ***`flags`*** `[,` ***`subexpr`*** `]]]])`. ***`pattern`*** is searched for in ***`string`***, normally from the beginning of the string, but if the ***`start`*** parameter is provided then beginning from that character index. If ***`N`*** is specified then the ***`N`***th match of the pattern is located, otherwise the first match is located. If the ***`endoption`*** parameter is omitted or specified as zero, the function returns the position of the first character of the match. Otherwise, ***`endoption`*** must be one, and the function returns the position of the character following the match. The ***`flags`*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24. For a pattern containing parenthesized subexpressions, ***`subexpr`*** is an integer indicating which subexpression is of interest: the result identifies the position of the substring matching that subexpression. Subexpressions are numbered in the order of their leading parentheses. When ***`subexpr`*** is omitted or zero, the result identifies the position of the whole match regardless of parenthesized subexpressions.

Some examples:

```
regexp_instr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                 23
regexp_instr('ABCDEFGHI', '(c..)(...)', 1, 1, 0, 'i', 2)
                                6
```

The `regexp_like` function checks whether a match of a POSIX regular expression pattern occurs within a string, returning boolean true or false. It has the syntax `regexp_like(`***`string`***`,` ***`pattern`*** `[,` ***`flags`*** `])`. The ***`flags`*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24. This function has the same results as the `~` operator if no flags are specified. If only the `i` flag is specified, it has the same results as the `~*` operator.

Some examples:

```
regexp_like('Hello World', 'world')       false
regexp_like('Hello World', 'world', 'i')  true
```

The `regexp_match` function returns a text array of matching substring(s) within the first match of a POSIX regular expression pattern to a string. It has the syntax `regexp_match(`***`string`***`,` ***`pattern`*** `[,` ***`flags`*** `])`. If there is no match, the result is `NULL`. If a match is found, and the ***`pattern`*** contains no parenthesized subexpressions, then the result is a single-element text array containing the substring matching the whole pattern. If a match is found, and the ***`pattern`*** contains parenthesized subexpressions, then the result is a text array whose ***`n`***'th element is the substring matching the ***`n`***'th parenthesized subexpression of the ***`pattern`*** (not counting “non-capturing” parentheses; see below for details). The ***`flags`*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24.

Some examples:

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

> **Tip**
>
> In the common case where you just want the whole matching substring or `NULL` for no match, the best solution is to use `regexp_substr()`. However, `regexp_substr()` only exists in PostgreSQL version 15 and up. When working in older versions, you can extract the first element of `regexp_match()`'s result, for example:
>
> ```
> SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
>  regexp_match
> --------------
>  barbeque
> (1 row)
> ```

The `regexp_matches` function returns a set of text arrays of matching substring(s) within matches of a POSIX regular expression pattern to a string. It has the same syntax as `regexp_match`. This function returns no rows if there is no match, one row if there is a match and the `g` flag is not given, or ***N*** rows if there are ***N*** matches and the `g` flag is given. Each returned row is a text array containing the whole matched substring or the substrings matching parenthesized subexpressions of the ***pattern***, just as described above for `regexp_match`. `regexp_matches` accepts all the flags shown in Table 9.24, plus the `g` flag which commands it to return all matches, not just the first one.

Some examples:

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

> **Tip**
>
> In most cases `regexp_matches()` should be used with the `g` flag, since if you only want the first match, it's easier and more efficient to use `regexp_match()`. However, `regexp_match()` only exists in PostgreSQL version 10 and up. When working in older versions, a common trick is to place a `regexp_matches()` call in a sub-select, for example:
>
> ```
> SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
> ```
>
> This produces a text array if there's a match, or `NULL` if not, the same as `regexp_match()` would do. Without the sub-select, this query would produce no output at all for table rows without a match, which is typically not the desired behavior.

The `regexp_replace` function provides substitution of new text for substrings that match POSIX regular expression patterns. It has the syntax `regexp_replace(`***source***, ***pattern***, ***replacement*** [, ***start*** [, ***N***]] [, ***flags***]). (Notice that ***N*** cannot be specified unless ***start*** is, but ***flags*** can be given in any case.) The ***source*** string is returned unchanged if there is no match to the ***pattern***. If there is a match, the ***source*** string is returned with the ***replacement*** string substituted for the matching substring. The ***replacement*** string can contain `\`***n***, where ***n*** is 1 through 9, to indicate that the source substring matching the ***n***'th parenthesized subexpression of the pattern should be inserted, and it can contain `\&` to indicate that the substring matching the entire pattern should be inserted. Write `\\` if you need to put a literal backslash in the replacement text. ***pattern*** is searched for in ***string***, normally from the beginning of the string, but if the ***start*** parameter is provided then beginning from that character index. By default, only the first match of the pattern is replaced. If ***N*** is specified and is greater than zero, then the ***N***th match of the

pattern is replaced. If the g flag is given, or if ***N*** is specified and is zero, then all matches at or after the ***start*** position are replaced. (The g flag is ignored when ***N*** is specified.) The ***flags*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags (though not g) are described in Table 9.24.

Some examples:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                    fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                    fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                    fooXarYXazY
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 0, 'i')
                                    X PXstgrXSQL fXnctXXn
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 3, 'i')
                                    A PostgrXSQL function
```

The `regexp_split_to_table` function splits a string using a POSIX regular expression pattern as a delimiter. It has the syntax `regexp_split_to_table(`***`string`***`, `***`pattern`*** [, ***`flags`***]). If there is no match to the ***pattern***, the function returns the ***string***. If there is at least one match, for each match it returns the text from the end of the last match (or the beginning of the string) to the beginning of the match. When there are no more matches, it returns the text from the end of the last match to the end of the string. The ***flags*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. `regexp_split_to_table` supports the flags described in Table 9.24.

The `regexp_split_to_array` function behaves the same as `regexp_split_to_table`, except that `regexp_split_to_array` returns its result as an array of `text`. It has the syntax `regexp_split_to_array(`***`string`***`, `***`pattern`*** [, ***`flags`***]). The parameters are the same as for `regexp_split_to_table`.

Some examples:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', '\s+') AS foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');
             regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

As the last example demonstrates, the regexp split functions ignore zero-length matches that occur at the start or end of the string or immediately after a previous match. This is contrary to the strict definition of regexp matching that is implemented by the other regexp functions, but is usually the most convenient behavior in practice. Other software systems such as Perl use similar definitions.

The `regexp_substr` function returns the substring that matches a POSIX regular expression pattern, or `NULL` if there is no match. It has the syntax `regexp_substr(`***`string`***`, `***`pattern`***` [, `***`start`***` [, `***`N`***` [, `***`flags`***` [, `***`subexpr`***` ]]]])`. ***`pattern`*** is searched for in ***`string`***, normally from the beginning of the string, but if the ***`start`*** parameter is provided then beginning from that character index. If ***`N`*** is specified then the ***`N`***th match of the pattern is returned, otherwise the first match is returned. The ***`flags`*** parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Supported flags are described in Table 9.24. For a pattern containing parenthesized subexpressions, ***`subexpr`*** is an integer indicating which subexpression is of interest: the result is the substring matching that subexpression. Subexpressions are numbered in the order of their leading parentheses. When ***`subexpr`*** is omitted or zero, the result is the whole match regardless of parenthesized subexpressions.

Some examples:

```
regexp_substr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                    town zip
regexp_substr('ABCDEFGHI', '(c..)(...)', 1, 1, 'i', 2)
                                   FGH
```

9.7.3.1. Regular Expression Details

PostgreSQL's regular expressions are implemented using a software package written by Henry Spencer. Much of the description of regular expressions below is copied verbatim from his manual.

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: *extended* REs or EREs (roughly those of `egrep`), and *basic* REs or BREs (roughly those of `ed`). PostgreSQL supports both forms, and also implements some extensions that are not in the POSIX standard, but have become widely used due to their availability in programming languages such as Perl and Tcl. REs using these non-POSIX extensions are called *advanced* REs or AREs in this documentation. AREs are almost an exact superset of EREs, but BREs have several notational incompatibilities (as well as being much more limited). We first describe the ARE and ERE forms, noting features that apply only to AREs, and then describe how BREs differ.

> **Note**
>
> PostgreSQL always initially presumes that a regular expression follows the ARE rules. However, the more limited ERE or BRE rules can be chosen by prepending an *embedded option* to the RE pattern, as described in Section 9.7.3.4. This can be useful for compatibility with applications that expect exactly the POSIX 1003.2 rules.

A regular expression is defined as one or more *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is zero or more *quantified atoms* or *constraints*, concatenated. It matches a match for the first, followed by a match for the second, etc.; an empty branch matches the empty string.

A quantified atom is an *atom* possibly followed by a single *quantifier*. Without a quantifier, it matches a match for the atom. With a quantifier, it can match some number of matches of the atom. An *atom* can be any of the possibilities shown in Table 9.17. The possible quantifiers and their meanings are shown in Table 9.18.

A *constraint* matches an empty string, but matches only when specific conditions are met. A constraint can be used where an atom could be used, except it cannot be followed by a quantifier. The simple constraints are shown in Table 9.19; some more constraints are described later.

Table 9.17. Regular Expression Atoms

Atom	Description
(***`re`***)	(where ***`re`*** is any regular expression) matches a match for ***`re`***, with the match noted for possible reporting
(?:***`re`***)	as above, but the match is not noted for reporting (a "non-capturing" set of parentheses) (AREs only)
.	matches any single character
[***`chars`***]	a *bracket expression*, matching any one of the ***`chars`*** (see Section 9.7.3.2 for more detail)
`k`	(where ***`k`*** is a non-alphanumeric character) matches that character taken as an ordinary character, e.g., `\\` matches a backslash character
`c`	where ***`c`*** is alphanumeric (possibly followed by other characters) is an *escape*, see Section 9.7.3.3 (AREs only; in EREs and BREs, this matches ***`c`***)
{	when followed by a character other than a digit, matches the left-brace character {; when followed by a digit, it is the beginning of a ***`bound`*** (see below)
`x`	where ***`x`*** is a single character with no other significance, matches that character

An RE cannot end with a backslash (`\`).

> **Note**
>
> If you have standard_conforming_strings turned off, any backslashes you write in literal string constants will need to be doubled. See Section 4.1.2.1 for more information.

Table 9.18. Regular Expression Quantifiers

Quantifier	Matches
*	a sequence of 0 or more matches of the atom
+	a sequence of 1 or more matches of the atom
?	a sequence of 0 or 1 matches of the atom
{***m***}	a sequence of exactly ***m*** matches of the atom
{***m***,}	a sequence of ***m*** or more matches of the atom
{***m***,***n***}	a sequence of ***m*** through ***n*** (inclusive) matches of the atom; ***m*** cannot exceed ***n***
*?	non-greedy version of *
+?	non-greedy version of +
??	non-greedy version of ?
{***m***}?	non-greedy version of {***m***}
{***m***,}?	non-greedy version of {***m***,}
{***m***,***n***}?	non-greedy version of {***m***,***n***}

The forms using **{...}** are known as *bounds*. The numbers ***m*** and ***n*** within a bound are unsigned decimal integers with permissible values from 0 to 255 inclusive.

Non-greedy quantifiers (available in AREs only) match the same possibilities as their corresponding normal (*greedy*) counterparts, but prefer the smallest number rather than the largest number of matches. See Section 9.7.3.5 for more detail.

> **Note**
>
> A quantifier cannot immediately follow another quantifier, e.g., ** is invalid. A quantifier cannot begin an expression or subexpression or follow ^ or |.

Table 9.19. Regular Expression Constraints

Constraint	Description
^	matches at the beginning of the string
$	matches at the end of the string
(?=***re***)	*positive lookahead* matches at any point where a substring matching ***re*** begins (AREs only)
(?!***re***)	*negative lookahead* matches at any point where no substring matching ***re*** begins (AREs only)
(?<=***re***)	*positive lookbehind* matches at any point where a substring matching ***re*** ends (AREs only)
(?<!***re***)	*negative lookbehind* matches at any point where no substring matching ***re*** ends (AREs only)

Lookahead and lookbehind constraints cannot contain *back references* (see Section 9.7.3.3), and all parentheses within them are considered non-capturing.

9.7.3.2. Bracket Expressions

A *bracket expression* is a list of characters enclosed in `[]`. It normally matches any single character from the list (but see below). If the list begins with ^, it matches any single character *not* from the rest of the list. If two characters in the list are separated by `-`, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g., `[0-9]` in ASCII matches any decimal digit. It is illegal for two ranges to share an endpoint, e.g., `a-c-e`. Ranges are very collating-sequence-dependent, so portable programs should avoid relying on them.

To include a literal `]` in the list, make it the first character (after ^, if that is used). To include a literal `-`, make it the first or last character, or the second endpoint of a range. To use a literal `-` as the first endpoint of a range, enclose it in `[.` and `.]` to make it a collating element (see below). With the exception of these characters, some combinations using `[` (see next paragraphs), and escapes (AREs only), all other special characters lose their special significance within a bracket expression. In particular, `\` is not special when following ERE or BRE rules, though it is special (as introducing an escape) in AREs.

Within a bracket expression, a collating element (a character, a multiple-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in `[.` and `.]` stands for the sequence of characters of that collating element. The sequence is treated as a single element of the bracket expression's list. This allows a bracket expression containing a multiple-character collating element to match more than one character, e.g., if the collating sequence includes a `ch` collating element, then the RE `[[.ch.]]*c` matches the first five characters of `chchcc`.

> **Note**
>
> PostgreSQL currently does not support multi-character collating elements. This information describes possible future behavior.

Within a bracket expression, a collating element enclosed in `[=` and `=]` is an *equivalence class*, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were `[.` and `.]`.) For example, if o and ^ are the members of an equivalence class, then `[[=o=]]`, `[[=^=]]`, and `[o^]` are all synonymous. An equivalence class cannot be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in `[:` and `:]` stands for the list of all characters belonging to that class. A character class cannot be used as an endpoint of a range. The POSIX standard defines these character class names: `alnum` (letters and numeric digits), `alpha` (letters), `blank` (space and tab), `cntrl` (control characters), `digit` (numeric digits), `graph` (printable characters except space), `lower` (lower-case letters), `print` (printable characters including space), `punct` (punctuation), `space` (any white space), `upper` (upper-case letters), and `xdigit` (hexadecimal digits). The behavior of these standard character classes is generally consistent across platforms for characters in the 7-bit ASCII set. Whether a given non-ASCII character is considered to belong to one of these classes depends on the *collation* that is used for the regular-expression function or operator (see Section 24.2), or by default on the database's LC_CTYPE locale setting (see Section 24.1). The classification of non-ASCII characters can vary across platforms even in similarly-named locales. (But the C locale never considers any non-ASCII characters to belong to any of these classes.) In addition to these standard character classes, PostgreSQL defines the `word` character class, which is the same as `alnum` plus the underscore (_) character, and the `ascii` character class, which contains exactly the 7-bit ASCII set.

There are two special cases of bracket expressions: the bracket expressions `[[:<:]]` and `[[:>:]]` are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is any character belonging to the `word` character class, that is, any letter, digit, or underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. The constraint escapes described below are usually preferable; they are no more standard, but are easier to type.

9.7.3.3. Regular Expression Escapes

Escapes are special sequences beginning with `\` followed by an alphanumeric character. Escapes come in several varieties: character entry, class shorthands, constraint escapes, and back references. A `\` followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a `\` followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, `\` is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes exist to make it easier to specify non-printing and other inconvenient characters in REs. They are shown in Table 9.20.

Class-shorthand escapes provide shorthands for certain commonly-used character classes. They are shown in Table 9.21.

A *constraint escape* is a constraint, matching the empty string if specific conditions are met, written as an escape. They are shown in Table 9.22.

A *back reference* (`\`***n***) matches the same string matched by the previous parenthesized subexpression specified by the number ***n*** (see Table 9.23). For example, `([bc])\1` matches `bb` or `cc` but not `bc` or `cb`. The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions. The back reference considers only the string characters matched by the referenced subexpression, not any constraints contained in it. For example, `(^\d)\1` will match 22.

Table 9.20. Regular Expression Character-Entry Escapes

Escape	Description
`\a`	alert (bell) character, as in C
`\b`	backspace, as in C
`\B`	synonym for backslash (`\`) to help reduce the need for backslash doubling
`\c`***X***	(where ***X*** is any character) the character whose low-order 5 bits are the same as those of ***X***, and whose other bits are all zero
`\e`	the character whose collating-sequence name is ESC, or failing that, the character with octal value `033`
`\f`	form feed, as in C
`\n`	newline, as in C
`\r`	carriage return, as in C
`\t`	horizontal tab, as in C
`\u`***wxyz***	(where ***wxyz*** is exactly four hexadecimal digits) the character whose hexadecimal value is `0x`***wxyz***
`\U`***stuvwxyz***	(where ***stuvwxyz*** is exactly eight hexadecimal digits) the character whose hexadecimal value is `0x`***stuvwxyz***
`\v`	vertical tab, as in C
`\x`***hhh***	(where ***hhh*** is any sequence of hexadecimal digits) the character whose hexadecimal value is `0x`***hhh*** (a single character no matter how many hexadecimal digits are used)
`\0`	the character whose value is `0` (the null byte)
`\`***xy***	(where ***xy*** is exactly two octal digits, and is not a *back reference*) the character whose octal value is `0`***xy***
`\`***xyz***	(where ***xyz*** is exactly three octal digits, and is not a *back reference*) the character whose octal value is `0`***xyz***

Hexadecimal digits are `0`-9, `a`-`f`, and `A`-`F`. Octal digits are `0`-7.

Numeric character-entry escapes specifying values outside the ASCII range (0–127) have meanings dependent on the database encoding. When the encoding is UTF-8, escape values are equivalent to Unicode code points, for example `\u1234` means the character U+1234. For other multibyte encodings, character-entry escapes usually just specify the concatenation of the byte values for the character. If the escape value does not correspond to any legal character in the database encoding, no error will be raised, but it will never match any data.

The character-entry escapes are always taken as ordinary characters. For example, `\135` is `]` in ASCII, but `\135` does not terminate a bracket expression.

Table 9.21. Regular Expression Class-Shorthand Escapes

Escape	Description
`\d`	matches any digit, like `[[:digit:]]`
`\s`	matches any whitespace character, like `[[:space:]]`
`\w`	matches any word character, like `[[:word:]]`
`\D`	matches any non-digit, like `[^[:digit:]]`
`\S`	matches any non-whitespace character, like `[^[:space:]]`

Escape	Description
`\W`	matches any non-word character, like `[^[:word:]]`

The class-shorthand escapes also work within bracket expressions, although the definitions shown above are not quite syntactically valid in that context. For example, `[a-c\d]` is equivalent to `[a-c[:digit:]]`.

Table 9.22. Regular Expression Constraint Escapes

Escape	Description
`\A`	matches only at the beginning of the string (see Section 9.7.3.5 for how this differs from ^)
`\m`	matches only at the beginning of a word
`\M`	matches only at the end of a word
`\y`	matches only at the beginning or end of a word
`\Y`	matches only at a point that is not the beginning or end of a word
`\Z`	matches only at the end of the string (see Section 9.7.3.5 for how this differs from $)

A word is defined as in the specification of `[[:<:]]` and `[[:>:]]` above. Constraint escapes are illegal within bracket expressions.

Table 9.23. Regular Expression Back References

Escape	Description
`\`***m***	(where ***m*** is a nonzero digit) a back reference to the ***m***'th subexpression
`\`***mnn***	(where ***m*** is a nonzero digit, and ***nn*** is some more digits, and the decimal value ***mnn*** is not greater than the number of closing capturing parentheses seen so far) a back reference to the ***mnn***'th subexpression

> **Note**
>
> There is an inherent ambiguity between octal character-entry escapes and back references, which is resolved by the following heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multi-digit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e., the number is in the legal range for a back reference), and otherwise is taken as octal.

9.7.3.4. Regular Expression Metasyntax

In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

An RE can begin with one of two special *director* prefixes. If an RE begins with `***:`, the rest of the RE is taken as an ARE. (This normally has no effect in PostgreSQL, since REs are assumed to be AREs; but it does have an effect if ERE or BRE mode had been specified by the ***flags*** parameter to a regex function.) If an RE begins with `***=`, the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.

An ARE can begin with *embedded options*: a sequence `(?`***xyz***`)` (where ***xyz*** is one or more alphabetic characters) specifies options affecting the rest of the RE. These options override any previously determined options — in particular, they can override the case-sensitivity behavior implied by a regex operator, or the ***flags*** parameter to a regex function. The available option letters are shown in Table 9.24. Note that these same option letters are used in the ***flags*** parameters of regex functions.

Table 9.24. ARE Embedded-Option Letters

Option	Description
`b`	rest of RE is a BRE
`c`	case-sensitive matching (overrides operator type)
`e`	rest of RE is an ERE
`i`	case-insensitive matching (see Section 9.7.3.5) (overrides operator type)
`m`	historical synonym for `n`
`n`	newline-sensitive matching (see Section 9.7.3.5)
`p`	partial newline-sensitive matching (see Section 9.7.3.5)
`q`	rest of RE is a literal (“quoted”) string, all ordinary characters
`s`	non-newline-sensitive matching (default)
`t`	tight syntax (default; see below)
`w`	inverse partial newline-sensitive (“weird”) matching (see Section 9.7.3.5)
`x`	expanded syntax (see below)

Embedded options take effect at the `)` terminating the sequence. They can appear only at the start of an ARE (after the `***:` director if any).

In addition to the usual (*tight*) RE syntax, in which all characters are significant, there is an *expanded* syntax, available by specifying the embedded `x` option. In the expanded syntax, white-space characters in the RE are ignored, as are all characters between a # and the following newline (or the end of the RE). This permits paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

- a white-space character or # preceded by `\` is retained
- white space or # within a bracket expression is retained
- white space and comments cannot appear within multi-character symbols, such as `(?:`

For this purpose, white-space characters are blank, tab, newline, and any character that belongs to the ***space*** character class.

Finally, in an ARE, outside bracket expressions, the sequence `(?#`***`ttt`***`)` (where ***`ttt`*** is any text not containing a `)`) is a comment, completely ignored. Again, this is not allowed between the characters of multi-character symbols, like `(?:`. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if an initial `***=` director has specified that the user's input be treated as a literal string rather than as an RE.

9.7.3.5. Regular Expression Matching Rules

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, either the longest possible match or the shortest possible match will be taken, depending on whether the RE is *greedy* or *non-greedy*.

Whether an RE is greedy or not is determined by the following rules:

- Most atoms, and all constraints, have no greediness attribute (because they cannot match variable amounts of text anyway).
- Adding parentheses around an RE does not change its greediness.
- A quantified atom with a fixed-repetition quantifier (`{`***`m`***`}` or `{`***`m`***`}?`) has the same greediness (possibly none) as the atom itself.
- A quantified atom with other normal quantifiers (including `{`***`m`***`,`***`n`***`}` with ***`m`*** equal to ***`n`***) is greedy (prefers longest match).
- A quantified atom with a non-greedy quantifier (including `{`***`m`***`,`***`n`***`}?` with ***`m`*** equal to ***`n`***) is non-greedy (prefers shortest match).
- A branch — that is, an RE that has no top-level `|` operator — has the same greediness as the first quantified atom in it that has a greediness attribute.
- An RE consisting of two or more branches connected by the `|` operator is always greedy.

The above rules associate greediness attributes not only with individual quantified atoms, but with branches and entire REs that contain quantified atoms. What that means is that the matching is done in such a way that the branch, or whole RE, matches the longest or shortest possible substring *as a whole*. Once the length of the entire match is determined, the part of it that matches any particular subexpression is determined on the basis of the greediness attribute of that subexpression, with subexpressions starting earlier in the RE taking priority over ones starting later.

An example of what this means:

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Result: 1
```

In the first case, the RE as a whole is greedy because `Y*` is greedy. It can match beginning at the `Y`, and it matches the longest possible string starting there, i.e., `Y123`. The output is the parenthesized part of that, or `123`. In the second case, the RE as a whole is non-greedy because `Y*?` is non-greedy. It can match beginning at the `Y`, and it matches the shortest possible string starting there, i.e., `Y1`. The subexpression `[0-9]{1,3}` is greedy but it cannot change the decision as to the overall match length; so it is forced to match just `1`.

In short, when an RE contains both greedy and non-greedy subexpressions, the total match length is either as long as possible or as short as possible, according to the attribute assigned to the whole RE. The attributes assigned to the subexpressions only affect how much of that match they are allowed to “eat” relative to each other.

The quantifiers `{1,1}` and `{1,1}?` can be used to force greediness or non-greediness, respectively, on a subexpression or a whole RE. This is useful when you need the whole RE to have a greediness attribute different from what's deduced from its elements. As an example, suppose that we are trying to separate a string containing some digits into the digits and the parts before and after them. We might try to do that like this:

```
SELECT regexp_match('abc01234xyz', '(.*)(\d+)(.*)');
Result: {abc0123,4,xyz}
```

That didn't work: the first `.*` is greedy so it “eats” as much as it can, leaving the `\d+` to match at the last possible place, the last digit. We might try to fix that by making it non-greedy:

```
SELECT regexp_match('abc01234xyz', '(.*?)(\d+)(.*)');
Result: {abc,0,""}
```

That didn't work either, because now the RE as a whole is non-greedy and so it ends the overall match as soon as possible. We can get what we want by forcing the RE as a whole to be greedy:

```
SELECT regexp_match('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Result: {abc,01234,xyz}
```

Controlling the RE's overall greediness separately from its components' greediness allows great flexibility in handling variable-length patterns.

When deciding what is a longer or shorter match, match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example: `bb*` matches the three middle characters of `abbbc`; `(week|wee)(night|knights)` matches all ten characters of `weeknights`; when `(.*).*` is matched against `abc` the parenthesized subexpression matches all three characters; and when `(a*)*` is matched against `bc` both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g., `x` becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, e.g., `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

If newline-sensitive matching is specified, `.` and bracket expressions using ^ will never match the newline character (so that matches will not cross lines unless the RE explicitly includes a newline) and ^ and `$` will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. But the ARE escapes `\A` and `\Z` continue to match beginning or end of string *only*. Also, the character class shorthands `\D` and `\W` will match a newline regardless of this mode. (Before PostgreSQL 14, they did not match newlines when in newline-sensitive mode. Write `[^[:digit:]]` or `[^[:word:]]` to get the old behavior.)

If partial newline-sensitive matching is specified, this affects `.` and bracket expressions as with newline-sensitive matching, but not ^ and `$`.

If inverse partial newline-sensitive matching is specified, this affects ^ and `$` as with newline-sensitive matching, but not `.` and bracket expressions. This isn't very useful but is provided for symmetry.

9.7.3.6. Limits And Compatibility

No particular limit is imposed on the length of REs in this implementation. However, programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that `\` does not lose its special significance inside bracket expressions. All other ARE features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the *** syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.

Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include `\b`, `\B`, the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-sensitive matching, the restrictions on parentheses and back references in lookahead/lookbehind constraints, and the longest/shortest-match (rather than first-match) matching semantics.

9.7.3.7. Basic Regular Expressions

BREs differ from EREs in several respects. In BREs, `|`, +, and ? are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are `\{` and `\}`, with `{` and `}` by themselves ordinary characters. The parentheses for nested subexpressions are `\(` and `\)`, with `(` and `)` by themselves ordinary characters. ^ is an ordinary character except at the beginning of the RE or the beginning of a parenthesized subexpression, `$` is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and * is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading ^). Finally, single-digit back references are available, and `\<` and `\>` are synonyms for `[[:<:]]` and `[[:>:]]` respectively; no other escapes are available in BREs.

9.7.3.8. Differences From SQL Standard And XQuery

Since SQL:2008, the SQL standard includes regular expression operators and functions that performs pattern matching according to the XQuery regular expression standard:

- `LIKE_REGEX`
- `OCCURRENCES_REGEX`
- `POSITION_REGEX`
- `SUBSTRING_REGEX`
- `TRANSLATE_REGEX`

PostgreSQL does not currently implement these operators and functions. You can get approximately equivalent functionality in each case as shown in Table 9.25. (Various optional clauses on both sides have been omitted in this table.)

Table 9.25. Regular Expression Functions Equivalencies

SQL standard	PostgreSQL
string `LIKE_REGEX` ***pattern***	`regexp_like(`***string***`, `***pattern***`)` or ***string*** `~` ***pattern***
`OCCURRENCES_REGEX(`***pattern*** `IN` ***string***`)`	`regexp_count(`***string***`, `***pattern***`)`
`POSITION_REGEX(`***pattern*** `IN` ***string***`)`	`regexp_instr(`***string***`, `***pattern***`)`
`SUBSTRING_REGEX(`***pattern*** `IN` ***string***`)`	`regexp_substr(`***string***`, `***pattern***`)`
`TRANSLATE_REGEX(`***pattern*** `IN` ***string*** `WITH` ***replacement***`)`	`regexp_replace(`***string***`, `***pattern***`, `***replacement***`)`

Regular expression functions similar to those provided by PostgreSQL are also available in a number of other SQL implementations, whereas the SQL-standard functions are not as widely implemented. Some of the details of the regular expression syntax will likely differ in each implementation.

The SQL-standard operators and functions use XQuery regular expressions, which are quite close to the ARE syntax described above. Notable differences between the existing POSIX-based regular-expression feature and XQuery regular expressions include:

- XQuery character class subtraction is not supported. An example of this feature is using the following to match only English consonants: `[a-z-[aeiou]]`.
- XQuery character class shorthands `\c`, `\C`, `\i`, and `\I` are not supported.
- XQuery character class elements using `\p{UnicodeProperty}` or the inverse `\P{UnicodeProperty}` are not supported.
- POSIX interprets character classes such as `\w` (see Table 9.21) according to the prevailing locale (which you can control by attaching a `COLLATE` clause to the operator or function). XQuery specifies these classes by reference to Unicode character properties, so equivalent behavior is obtained only with a locale that follows the Unicode rules.
- The SQL standard (not XQuery itself) attempts to cater for more variants of "newline" than POSIX does. The newline-sensitive matching options described above consider only ASCII NL (`\n`) to be a newline, but SQL would have us treat CR (`\r`), CRLF (`\r\n`) (a Windows-style newline), and some Unicode-only characters like LINE SEPARATOR (U+2028) as newlines as well. Notably, `.` and `\s` should count `\r\n` as one character not two according to SQL.
- Of the character-entry escapes described in Table 9.20, XQuery supports only `\n`, `\r`, and `\t`.
- XQuery does not support the `[:`*`name`*`:]` syntax for character classes within bracket expressions.
- XQuery does not have lookahead or lookbehind constraints, nor any of the constraint escapes described in Table 9.22.
- The metasyntax forms described in Section 9.7.3.4 do not exist in XQuery.
- The regular expression flag letters defined by XQuery are related to but not the same as the option letters for POSIX (Table 9.24). While the `i` and `q` options behave the same, others do not:
  - XQuery's `s` (allow dot to match newline) and `m` (allow ^ and $ to match at newlines) flags provide access to the same behaviors as POSIX's `n`, `p` and `w` flags, but they do *not* match the behavior of POSIX's `s` and `m` flags. Note in particular that dot-matches-newline is the default behavior in POSIX but not XQuery.
  - XQuery's `x` (ignore whitespace in pattern) flag is noticeably different from POSIX's expanded-mode flag. POSIX's `x` flag also allows # to begin a comment in the pattern, and POSIX will not ignore a whitespace character after a backslash.

Submit correction

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use this form to report a documentation issue.