Operators Reference

Operators Reference

Anonymous · 09/09/2026

Operators Reference

Every operator has a **symbol** — the prefix that appears immediately after `?` in an expression. The filter parser matches operators longest-first, so `!~~*` is matched before `!~~` and `!~*` before `!~`.

This page documents all built-in operators grouped by type.

Value Format Conventions

Placeholder	Meaning
`VALUE`	Any string value
`PATTERN`	A LIKE pattern (may contain `%` and `_`)
`DATE`	`YYYYMMDD` or `YYMMDD`
`MONTH`	`MM` or `M` (1–12)
`YEAR`	`YYYY` or `YY`
`PERIOD`	`YYYYMM` or `YYMM`
`INT`	Non-negative integer (no decimals)
`V1,V2`	Two comma-separated values
`V1,V2,…`	One or more comma-separated values
(empty)	No value; the symbol alone is the full filter

Standard Comparison Operators

Map directly to SQL comparison operators. Accept any string or numeric value.

Symbol	Name	Value	SQL Template
`=`	Equals	`VALUE`	`{{column}} = {{value}}`
`!=`	Not Equal	`VALUE`	`{{column}} != {{value}}`
`!`	Not Equal (shorthand)	`VALUE`	alias of `!=`
`<>`	Not Equal (SQL standard)	`VALUE`	alias of `!=`
`>=`	Greater Than or Equal	`VALUE`	`{{column}} >= {{value}}`
`<=`	Less Than or Equal	`VALUE`	`{{column}} <= {{value}}`
`>`	Greater Than	`VALUE`	`{{column}} > {{value}}`
`<`	Less Than	`VALUE`	`{{column}} < {{value}}`

Examples

```
price?=1000            →  price = :p            (:p = '1000')
status?!=archived      →  status != :p          (:p = 'archived')
total?>=500            →  total >= :p           (:p = '500')
created_at?<2025-01-01 →  created_at < :p       (:p = '2025-01-01')
```

Tip: Standard operators work with any data type — numbers, strings, ISO dates. The database engine applies its own type coercion.

AutoLike Operators (Automatic Pattern Generation)

These operators accept a plain text value and wrap it in `%` wildcards automatically before binding. They delegate to the `like:` / `ilike:` family for the actual SQL.

Starts With

Symbol	Case	Value	Bound value	SQL
`^`	Sensitive	`VALUE`	`VALUE%`	`col LIKE :p` (pgsql/sqlite), `col LIKE BINARY :p` (mysql)
`^*`	Insensitive	`VALUE`	`VALUE%`	`col ILIKE :p` (pgsql), `col LIKE :p` (mysql/sqlite)
`!^`	Sensitive	`VALUE`	`VALUE%`	`col NOT LIKE :p`
`!^*`	Insensitive	`VALUE`	`VALUE%`	`col NOT ILIKE :p` / `col NOT LIKE :p`

Contains

Symbol	Case	Value	Bound value	SQL
`~~`	Sensitive	`VALUE`	`%VALUE%`	`col LIKE :p`
`~~*`	Insensitive	`VALUE`	`%VALUE%`	`col ILIKE :p` / `col LIKE :p`
`!~~`	Sensitive	`VALUE`	`%VALUE%`	`col NOT LIKE :p`
`!~~*`	Insensitive	`VALUE`	`%VALUE%`	`col NOT ILIKE :p` / `col NOT LIKE :p`

Ends With

Symbol	Case	Value	Bound value	SQL
`$`	Sensitive	`VALUE`	`%VALUE`	`col LIKE :p`
`$*`	Insensitive	`VALUE`	`%VALUE`	`col ILIKE :p` / `col LIKE :p`
`!$`	Sensitive	`VALUE`	`%VALUE`	`col NOT LIKE :p`
`!$*`	Insensitive	`VALUE`	`%VALUE`	`col NOT ILIKE :p` / `col NOT LIKE :p`

Examples

```
name?^John      →  name LIKE :p         (:p = 'John%')
name?$*LLC      →  name ILIKE :p        (:p = '%LLC')    (pgsql)
description?~~keyword  →  description LIKE :p  (:p = '%keyword%')
title?!~~*draft →  title NOT ILIKE :p  (:p = '%draft%')  (pgsql)
```

Note on case sensitivity: PostgreSQL natively supports `ILIKE`; MySQL uses `LIKE BINARY` for case-sensitive and plain `LIKE` for case-insensitive; SQLite uses plain `LIKE` for both (SQLite's `LIKE` is case-insensitive by default for ASCII).

Pattern (LIKE) Operators

Explicit LIKE operators where you supply the full pattern including `%` and _ wildcards.

Symbol	Case	Value	SQL
`like:`	Sensitive	`PATTERN`	pgsql: `col LIKE :p` · mysql: `col LIKE BINARY :p` · sqlite: `col LIKE :p`
`notlike:`	Sensitive	`PATTERN`	pgsql: `col NOT LIKE :p` · mysql: `col NOT LIKE BINARY :p`
`ilike:`	Insensitive	`PATTERN`	pgsql: `col ILIKE :p` · mysql: `col LIKE :p` · sqlite: `col LIKE :p`

Symbol	Case	Value	SQL
`notilike:`	Insensitive	`PATTERN`	pgsql: `col NOT ILIKE :p` · mysql/sqlite: `col NOT LIKE :p`

Examples

```
email?like:%.example.com  →  email LIKE :p   (:p = '%.example.com')
name?ilike:jo_n%          →  name ILIKE :p   (:p = 'jo_n%')   (pgsql)
code?notlike:TEST_%       →  code NOT LIKE BINARY :p  (mysql)
```

Note: `ilike:` and `notilike:` are **not supported** in Doctrine ORM DQL (only usable via SQL/DBAL/Illuminate bridges).

List Operators

Work with comma-separated lists of values. The list is split and each item becomes a separate named parameter.

Symbol	Value	SQL
`in:`	`V1,V2,…`	`{{column}} IN (:p1, :p2, …)`
`notin:`	`V1,V2,…`	`{{column}} NOT IN (:p1, :p2, …)`

Value Format

Values are separated by commas. To include a literal comma in a value, escape it with `\,`:

```
status?in:active,pending,review   →  status IN (:p1, :p2, :p3)
tags?in:a\,b,c\,d                 →  tags IN (:p1, :p2)   (:p1='a,b', :p2='c,d')
```

Validation

The value must contain at least one item. Each item may contain word characters, dots, hyphens, and Unicode letters/marks. Whitespace inside values is not supported.

Examples

```
status?in:paid,issued           →  status IN (:p1, :p2)
category?notin:archived,draft   →  category NOT IN (:p1, :p2)
id?in:1,2,3,4,5                 →  id IN (:p1, :p2, :p3, :p4, :p5)
```

Range Operators

Filter values between two bounds (inclusive) or outside them.

Symbol	Value	SQL
`between:`	`V1,V2`	`{{column}} BETWEEN {{value_1}} AND {{value_2}}`
`notbetween:`	`V1,V2`	`{{column}} NOT BETWEEN {{value_1}} AND {{value_2}}`

Value Format

Exactly two comma-separated values. Each value may contain word characters, dots, and hyphens (no spaces).

Examples

```
price?between:100,500      →  price BETWEEN :p1 AND :p2
price?notbetween:0,10      →  price NOT BETWEEN :p1 AND :p2
created_at?between:2024-01-01,2024-12-31  →  created_at BETWEEN :p1 AND :p2
```

Note: `BETWEEN` is inclusive on both ends. For exclusive ranges, use `>` and `<` separately.

Date Operators

Specialized operators for date and time filtering. They apply SQL date functions and normalize the value.

`date:` — Specific Date

```
date:YYYYMMDD   or   date:YYMMDD
```

Matches rows where the date portion of a datetime column equals a specific day.

Engine	SQL
All	`DATE({{column}}) = {{value}}`

Value normalization:

- `YYYYMMDD` (8 digits) → `YYYY-MM-DD`

- `YYMMDD` (6 digits) → `20YY-MM-DD`

```
created_at?date:20240823   →  DATE(created_at) = :p   (:p = '2024-08-23')
created_at?date:240101     →  DATE(created_at) = :p   (:p = '2024-01-01')
```

`month:` — Month Number

```
month:MM   or   month:M
```

Matches rows in a specific calendar month, regardless of year.

Engine	SQL
All	`MONTH({{column}}) = {{value}}`

Value normalization: zero-padded to 2 digits (`8` → `08`).

```
created_at?month:08    →  MONTH(created_at) = :p   (:p = '08')
created_at?month:3     →  MONTH(created_at) = :p   (:p = '03')
```

`year:` — Year

```
year:YYYY   or   year:YY
```

Matches rows in a specific year.

Engine	SQL
All	`YEAR({{column}}) = {{value}}`

Value normalization: 2-digit years are expanded to `20YY`.

```
created_at?year:2024   →  YEAR(created_at) = :p   (:p = '2024')
created_at?year:24     →  YEAR(created_at) = :p   (:p = '2024')
```

`period:` — Year and Month

```
period:YYYYMM   or   period:YYMM
```

Matches rows in a specific year+month.

Engine	SQL
pgsql	`TO_CHAR({{column}}, "YYYYMM") = {{value}}`

Engine	SQL
mysql	`DATE_FORMAT({{column}}, "%Y%m") = {{value}}`
sqlite	`strftime("%Y%m", {{column}}) = {{value}}`

Value normalization: 4-digit `YYMM` is expanded to `20YYMM`.

```
created_at?period:202408   →  strftime("%Y%m", created_at) = :p   (:p = '202408')
(sqlite)
created_at?period:2408     →  …                                   (:p = '202408')
```

Note: Date operators use SQL functions (`DATE()`, `MONTH()`, etc.) and are **not compatible with Doctrine ORM DQL**. Use the Doctrine DBAL or Illuminate bridges for date filtering.

NULL Operators

Symbol	Alias of	SQL
`is:null`	—	`{{column}} IS NULL`
`<=>`	`is:null`	`{{column}} IS NULL`
`isnot:null`	—	`{{column}} IS NOT NULL`

Value Format

The symbol is the complete filter — no value is expected after the symbol. `is:null` means the expression is literally `column?is:null`.

Examples

```
deleted_at?is:null      →  deleted_at IS NULL
deleted_at?<=>          →  deleted_at IS NULL    (alternative syntax)
email?isnot:null        →  email IS NOT NULL
```

Note: NULL operators produce no bound parameters.

Subquery Operators

Used exclusively with **exists paths** (`___`). They determine whether the correlated subquery checks for existence or absence.

Symbol	SQL
`is:empty`	`NOT EXISTS (SELECT 1 FROM … WHERE …)`
`isnot:empty`	`EXISTS (SELECT 1 FROM … WHERE …)`

```
___payments[on:id=invoice_id]?is:empty
→  NOT EXISTS (SELECT 1 FROM payments WHERE invoices.id = payments.invoice_id)

___payments[on:id=invoice_id]?isnot:empty
→  EXISTS (SELECT 1 FROM payments WHERE invoices.id = payments.invoice_id)
```

In Doctrine ORM DQL, these generate `SIZE(alias.assoc) = 0` and `SIZE(alias.assoc) > 0` respectively.

Regular Expression Operators

These operators match POSIX extended regular expressions. Support varies by engine.

Core Operators

Symbol	Case	SQL (pgsql)	SQL (mysql)
`~`	Sensitive	`{{column}} ~ {{value}}`	`{{column}} REGEXP BINARY {{value}}`
`~*`	Insensitive	`{{column}} ~* {{value}}`	`{{column}} REGEXP {{value}}`
`!~`	Sensitive	`{{column}} !~ {{value}}`	`{{column}} NOT REGEXP BINARY {{value}}`
`!~*`	Insensitive	`{{column}} !~* {{value}}`	`{{column}} NOT REGEXP {{value}}`

SQLite does not have built-in regex support — these operators have no SQLite template and will fail if used against a SQLite connection.

MySQL Aliases

Symbol	Alias of
`regexp:`	`~`
`notregexp:`	`!~`
`rlike:`	`~`
`notrlike:`	`!~`

These are provided for familiarity with MySQL's SQL syntax:

```
name?regexp:^John   →  name REGEXP BINARY :p   (mysql)
name?rlike:John.*   →  name REGEXP BINARY :p   (mysql)
```

PostgreSQL SIMILAR TO

Symbol	SQL
`similarto:`	`{{column}} SIMILAR TO {{value}}`
`notsimilarto:`	`{{column}} NOT SIMILAR TO {{value}}`

`SIMILAR TO` uses SQL-standard regex patterns (a hybrid of LIKE wildcards and regex). It is PostgreSQL-specific.

Pattern rules: may contain `%`, `_`, `[a-z]`, `(a\|b)`, `?`, `*`, `+`. Unbalanced `(`, `)`, `[`, `]` are rejected at parse time.

```
name?similarto:John%           →  name SIMILAR TO :p
code?similarto:[0-9]+          →  code SIMILAR TO :p
status?notsimilarto:(a|b)%     →  status NOT SIMILAR TO :p
```

Examples

```
email?~@example\.com    →  email ~ :p   (pgsql)
name?~*john             →  name ~* :p   (pgsql) / name REGEXP :p (mysql)
phone?!~^\+             →  phone !~ :p  (pgsql)
```

Validation: Regex patterns must contain at least one non-metacharacter character to prevent trivially-matching expressions.

Note: Regex operators are **not compatible with Doctrine ORM DQL**.

Bitwise Operators

Operate on integer values at the bit level. All values must be non-negative integers (no decimals).

Symbol	Name	Value	SQL
`b&`	Bitwise AND	`INT`	pgsql/sqlite: `col & :p` · mysql: `BIT_AND(col, :p)`
`b|`	Bitwise OR	`INT`	all: `col | :p`
`b^`	Bitwise XOR	`INT`	pgsql: `col # :p` · mysql: `BIT_XOR(col, :p)` · sqlite: `(col | :p) & ~(col & :p)`
`b<<`	Left Shift	`INT`	pgsql/sqlite: `col << :p` · mysql: `<< col, :p`

Symbol	Name	Value	SQL
`b>>`	Right Shift	`INT`	pgsql/sqlite: `col >> :p` · mysql: `>> col, :p`
`b&~`	AND NOT	`INT`	all: `col & ~( :p )`

Common Use Case: Feature Flags

Bitwise AND is commonly used to test whether a flag bit is set in a bitmask column:

```
flags?b&1    →  flags & :p   (:p = '1')   — tests bit 0 (value 1)
flags?b&2    →  flags & :p   (:p = '2')   — tests bit 1 (value 2)
flags?b&4    →  flags & :p   (:p = '4')   — tests bit 2 (value 4)
```

Examples

```
permissions?b&4          →  permissions & :p   (:p = '4')   (pgsql)
permissions?b|3          →  permissions | :p   (:p = '3')
version?b^255            →  (version | :p) & ~(version & :p)   (sqlite)
offset?b<<2              →  offset << :p   (:p = '2')
mask?b&~8                →  mask & ~( :p )   (:p = '8')
```

Validation: Non-integer values (letters, decimals like `1.5`) are rejected at parse time.

Note: Bitwise operators are **not compatible with Doctrine ORM DQL**.

Operator Aliases

Several operators are aliases that reuse another operator's SQL template:

Symbol	Alias of	Reason
`!`	`!=`	Shorthand not-equal
`<>`	`!=`	SQL-standard not-equal
`<=>`	`is:null`	MySQL NULL-safe equality syntax
`^`	`like:`	auto-casts with `like_start`
`^*`	`ilike:`	auto-casts with `like_start`
`!^`	`notlike:`	auto-casts with `like_start`
`!^*`	`notilike:`	auto-casts with `like_start`
`~~`	`like:`	auto-casts with `like`

Symbol	Alias of	Reason
`~~*`	`ilike:`	auto-casts with `like`
`!~~`	`notlike:`	auto-casts with `like`
`!~~*`	`notilike:`	auto-casts with `like`
`$`	`like:`	auto-casts with `like_end`
`$*`	`ilike:`	auto-casts with `like_end`
`!$`	`notlike:`	auto-casts with `like_end`
`!$*`	`notilike:`	auto-casts with `like_end`
`regexp:`	`~`	MySQL familiarity
`notregexp:`	`!~`	MySQL familiarity
`rlike:`	`~`	MySQL familiarity
`notrlike:`	`!~`	MySQL familiarity

Casting Rules

Casting rules transform the raw value string before it is bound as a parameter:

Rule	Input	Output
`like_start`	`John`	`John%`
`like`	`John`	`%John%`
`like_end`	`Smith`	`%Smith`
`list`	`a,b,c`	`['a', 'b', 'c']` (split)
`date`	`20240823`	`2024-08-23`
`date`	`240823`	`2024-08-23`
`month`	`8`	`08`
`year`	`24`	`2024`
`period`	`202408`	`202408` (unchanged)
`period`	`2408`	`202408`

DQL Compatibility Matrix

The Doctrine ORM bridge generates DQL, which does not support all SQL features:

Operator type	DQL compatible?
Standard (=, !=, >, etc.)	✅
AutoLike (^, ~~, $, etc.)	✅
like:, notlike:	✅
ilike:, notilike:	✅
in:, notin:	✅
between:, notbetween:	✅
Date (date:, month:, year:, period:)	✅
NULL (is:null, isnot:null)	✅
Subquery (is:empty, isnot:empty)	✅ (→ SIZE())
RegExp (~, ~*, similarto:, etc.)	✅
Binary (b&, b|, etc.)	✅

For incompatible operators, use the Doctrine DBAL bridge or `IlluminateQueryBuilderConditionApplier` instead. In API Platform's `SmartFilter`, incompatible operators are silently skipped.

Custom Operators

Operators are defined in `resources/operators.yaml`. You can load a custom YAML file to add or override operators:

```
# my-operators.yaml
version: '1.0.0'
description: 'Custom operators'

types:
    standard:
        name: 'Standard Operators'
        description: 'SQL comparison operators'

operators:
    '=':
        type: standard
```

```
        name: Equals
        description: Exact match
        sql: '{{column}} = {{value}}'

    'startswith:':
        type: like
        name: Starts With (verbose)
        description: Explicit starts-with LIKE operator
        sql: '{{column}} LIKE {{value}}'
        cast: ['like_start']
```

```
use Derafu\Query\Operator\OperatorLoader;
use Derafu\Query\Operator\OperatorManagerFactory;

$factory = new OperatorManagerFactory(new OperatorLoader());
$manager = $factory->create('/path/to/my-operators.yaml');
```

Required fields per operator: `type`, `name`, `description`. The `type` key must reference an entry in the `types` section.

Last updated on 09/09/2026