Expression Syntax

Expression Syntax

Anonymous · 09/09/2026

Expression Syntax

An expression is the fundamental unit passed to `where()`, `andWhere()`, `orWhere()`, `having()`, and the bridge appliers. It encodes **what** to filter, **how** to filter it, and (optionally) how to combine multiple filters into one string.

Single Expression: `path?filter`

A single expression has the form:

```
path?filter
```

The `?` is the mandatory separator between the **path** (which column or relationship to target) and the **filter** (which operator and value to apply).

```
price?>1000
^_____  ^____
path    filter
```

Path

The path identifies a column. For a simple column in the driving table, it is just the column name:

```
price
status
created_at
```

For a column in a related table, segments are chained with double underscores `__`:

```
invoices__customers__name
```

For more details about the path syntax — including aliases, join conditions, and subquery paths — see

Path Syntax.

Filter

The filter starts with an operator symbol immediately followed by the value (if any):

```
>1000         ← operator ">", value "1000"
=active       ← operator "=", value "active"
in:a,b,c      ← operator "in:", value "a,b,c"
is:null       ← operator "is:null", no value (the value part is empty)
^^start       ← NOT valid — "^^" is not a registered operator
```

The `FilterParser` tries registered operators longest-first to avoid ambiguity between, for example, `!~*` and `!~`.

For the complete list of operators with their symbols, value formats, and SQL output, see Operators Reference.

Composite Expressions

Multiple conditions can be combined in a single string using:

Operator	Meaning	Precedence
`&&`	AND	Higher
`||`	OR	Lower
`( )`	Grouping	Overrides default precedence

The grammar follows standard boolean precedence: `&&` binds tighter than `||`.

```
A && B || C        is parsed as   (A && B) || C
A || B && C        is parsed as   A || (B && C)
(A || B) && C      grouping overrides, result: (A || B) && C
```

Examples

Simple AND:

```
status?=active&&total?>1000
```

Produces: `status = :p1 AND total > :p2`

Simple OR:

```
category?=electronics||category?=hardware
```

Produces: `category = :p1 OR category = :p2`

AND with nested OR:

```
status?=active&&(type?=person||tax_id?^78)
```

Produces: `status = :p1 AND (type = :p2 OR tax_id LIKE :p3)`

OR with nested AND:

```
category?=software||(category?=hardware&&price?>200)
```

Produces: `category = :p1 OR (category = :p2 AND price > :p3)`

Parser Rules

- `&&` and `||` are only split at **parenthesis depth 0**, so function calls like `SUM(amount)` inside paths are not split.
- Whitespace around `&&` and `||` is trimmed.
- A fully parenthesized expression like `(A&&B)` is unwrapped and parsed recursively.

Multiple Expressions as an Array

Instead of combining with `&&`/`||` in a string, you can pass an array of expressions to `where()` or `andWhere()`. Each element is ANDed together:

```
$qb->where(['status?=active', 'total?>1000']);
// Equivalent to: status?=active&&total?>1000
```

This is useful when expressions are generated dynamically:

```
$filters = [];
if ($status) {
    $filters[] = 'status?=' . $status;
}
if ($minTotal) {
    $filters[] = 'total?>=' . $minTotal;
}
```

```
$qb->where($filters);
```

The `?E` Literal-Expression Marker

By default the value part of a filter is treated as a **literal** — it becomes a bound parameter:

```
price?>1000   →   price > :param   with :param = '1000'
```

To reference another column or expression (not a literal value), replace `?` with `?E`:

```
id?E!=other_alias.id
```

This tells the parser that the value is an SQL identifier or expression, not a bound parameter. The value is sanitized as an identifier and inserted directly into the SQL — **no parameter binding**.

Use this sparingly and only for trusted, controlled inputs, since the sanitizer strips characters but does not provide the same guarantees as prepared statement binding.

Example from the test suite — simulating a self-join:

```
$qb->where([
'products[alias:p1]__invoice_details[on:id=product_id,alias:id1]__invoice_id?isnot:nul
l',
'products[alias:p1]__invoice_details[...alias:id2]__products[...alias:p2]__id?E!=p1.id
',
]);
```

The second condition generates `p2.id != p1.id` (column reference, not a literal).

How Expressions Are Parsed

The entry point is `CompositeExpressionParser::parse(string $expression)`.

1. Trim whitespace.
2. Split on `||` at depth 0 → if more than one part, build an OR composite.
3. For each part, split on `&&` at depth 0 → if more than one part, build an AND composite.
4. For each atom, if wrapped in `(…)` unwrap and recurse; otherwise, call `ExpressionParser::parse()`.

`ExpressionParser::parse()`:

1. Look for `?E` marker → set `literal = false`, replace `?E` with `?`.
2. Split on the first `?` → `pathExpression` and `filterExpression`.
3. Call `PathParser::parse(pathExpression)` → `Path`.
4. Call `FilterParser::parse(filterExpression)` → `Filter`.
5. Return `new Condition(path, filter, literal)`.

`FilterParser::parse()`:

1. Retrieve operators sorted longest-first from `OperatorManager`.
2. Try each symbol as a prefix of `filterExpression`.
3. On match: extract value, create `Filter`, call `validate()`, return.
4. No match: throw `InvalidArgumentException`.

Validation at Parse Time

The `Filter::validate()` method checks the value against the operator's `pattern` field (if defined):

- Operators whose symbol ends in `:` (e.g. `like:`, `in:`, `between:`) always require a non-empty value.
- Specific patterns: dates must match `YYYYMMDD` or `YYMMDD`, bitwise values must be integers, list values must match the allowed character set.
- NULL operators (`is:null`, `isnot:null`, `is:empty`, `isnot:empty`) require an **empty** value — the operator symbol itself is the full expression.

Invalid expressions throw `InvalidArgumentException` immediately, before any SQL is generated.

Last updated on 09/09/2026