

Docs

Query Project

Derafu Query

Anonymous · 22/09/2026 · #php

Table of Contents

- Query Project 3
 - Introduction* 4
 - Architecture* 8
 - Expression Syntax* 21
 - Path Syntax* 26
 - Operators Reference* 32
 - Query Builder* 45
 - Framework Bridges* 55
 - Security Guide* 63
 - Query Config* 68

Docs » Data Handling Category » Query Project

Query Project

Derafu Query

Anonymous · 22/09/2026 · #php

Derafu Query

Last updated on 22/09/2026

Introduction

Expressive Path-Based Query Builder for PHP

Anonymous · 22/09/2026

Expressive Path-Based Query Builder for PHP

github

derafu/query

last commit

may

CI

passing

code size

301 KiB

open issues

0

downloads

90

downloads

4 this month

derafu/query is a PHP library for building and filtering SQL queries using a compact, URL-safe string syntax. A single expression like `customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000` carries the full join path, join conditions, and filter in one string — no separate join calls required.

Why Derafu\Query?

Traditional query builders require explicit join definitions, verbose relationship navigation, and different filter syntaxes across engines. `derafu/query` replaces all of that with:

- A **single expression format** that encodes path, joins, and filter in one string.
- **Automatic join generation** from path segments — no manual `join()` calls when using path syntax.
- **50+ operators** covering comparisons, patterns, lists, ranges, dates, NULL, regex, bitwise, and subqueries.
- **Database-specific SQL** generated from the same expression — PostgreSQL, MySQL, and SQLite without code changes.
- **Framework bridges** for Doctrine DBAL, Doctrine ORM, Laravel's Illuminate query builder, and API Platform.

Feature	Derafu\Query	Traditional Query Builders
Path-Based Relationships	☑	☐
Automatic Join Resolution	☑	☐
Configurable Operators	☑	☐
Framework Agnostic Core	☑	⚠
Unified Filter Syntax	☑	⚠
Multi-DB SQL Generation	☑	⚠

Installation

```
composer require derafu/query
```

Quick Start

With SqlQueryBuilder (standalone)

```
use Derafu\Query\Builder\SqlQueryBuilder;
use Derafu\Query\Engine\PdoEngine;
use Derafu\Query\Filter\ExpressionParser;
use Derafu\Query\Filter\PathParser;
use Derafu\Query\Filter\FilterParser;
use Derafu\Query\Filter\CompositeExpressionParser;
use Derafu\Query\Operator\OperatorLoader;
use Derafu\Query\Operator\OperatorManager;

$pdo      = new PDO('sqlite:/path/to/db.sqlite');
$engine    = new PdoEngine($pdo);

$loader    = new OperatorLoader();
$manager   = new
OperatorManager($loader->loadFromFile('vendor/derafu/query/resources/operators.yaml'));
;
$parser    = new CompositeExpressionParser(
    new ExpressionParser(new PathParser(), new FilterParser($manager))
);

$qqb = new SqlQueryBuilder($engine, $parser);

// Simple filter.
$rows = $qqb->table('products')->where('price?>1000')->execute();

// Multi-table path (auto-generates the JOIN).
$rows = $qqb
    ->select('c.name AS customer_name, i.number, i.total')
    ->where('customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000')
    ->execute();
```

With a Framework Bridge

```
// Doctrine DBAL.
```

```
use Derafu\Query\Bridge\DoctrineDBALQueryBuilderConditionApplier;
use Derafu\Query\Filter\CompositeExpressionParser;

$applier = new DoctrineDBALQueryBuilderConditionApplier();
$condition = $parser->parse('status?=active&&total?>1000');
$applier->apply($dbalQueryBuilder, $condition);
```

The Expression Format

Every filter is a string of the form:

```
path?filter
```

The `?` separates the **path** (which column or relationship to target) from the **filter** (which operator and value to apply).

price?>1000	← column "price", operator ">", value "1000"
status?in:paid,issued	← column "status", operator "in:", value "paid,issued"
created_at?date:20240301	← column "created_at", operator "date:", value "20240301"
deleted_at?is:null	← column "deleted_at", operator "is:null", no value

Multi-segment paths navigate relationships and generate joins automatically:

```
customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000
```

Subquery paths starting with `__` generate correlated EXISTS / aggregate subqueries:

__payments[on:id=invoice_id]?is:empty	← NOT EXISTS (payments)
__payments[on:id=invoice_id]__status?=pending	← EXISTS with column filter
__payments[on:id=invoice_id]__SUM(amount)?>=500	← aggregate scalar subquery

Multiple conditions can be combined with `&&` (AND), `||` (OR), and `()` (grouping):

```
status?=active&&total?>1000
category?=electronics||(category?=software&&price?<500)
```

Operator Overview

derafu/query ships with over 50 operators across 10 types:

Type	Examples	SQL produced
Standard	<code>=, !=, >, <, >=, <=</code>	<code>col = :p, col > :p</code>
AutoLike	<code>^, \$, ~~, ~~*, !~~</code>	<code>col LIKE :p</code> with auto %
Like	<code>like:, ilike:, notlike:</code>	<code>col LIKE :p, col ILIKE :p</code>
List	<code>in:, notin:</code>	<code>col IN (:p1, :p2, ...)</code>
Range	<code>between:, notbetween:</code>	<code>col BETWEEN :p1 AND :p2</code>
Date	<code>date:, month:, year:, period:</code>	<code>DATE(col) = :p, etc.</code>
NULL	<code>is:null, isnot:null, <=></code>	<code>col IS NULL</code>
Subquery	<code>is:empty, isnot:empty</code>	<code>NOT EXISTS (...), EXISTS (...)</code>
RegExp	<code>~, ~*, !~, similarto:</code>	<code>col ~ :p, col SIMILAR TO :p</code>
Binary	<code>b&, b , b^, b<<, b>></code>	<code>col & :p, etc.</code>

What This Documentation Covers

Page	Content
Architecture	Layer structure, class responsibilities, data flow
Expression Syntax	<code>path?filter</code> format, composite <code>&& /()</code>
Path Syntax	Segments, options, join paths, subquery paths
Operators Reference	All operators with SQL templates, validation, casting
Query Builder	Fluent API and declarative QueryConfig
Framework Bridges	Doctrine DBAL/ORM, illuminate, API Platform
Security Guide	SQL sanitization and safe usage patterns

Last updated on 22/09/2026

Docs » Data Handling Category » Query Project » Architecture

Architecture

Architecture

Anonymous · 22/09/2026

Architecture

derafu/query is organized into six cooperating layers. Understanding them helps you choose which classes to instantiate, which bridges to use, and where to add custom behavior.

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

Layers at a Glance

Namespace	Responsibility
Derafu\Query\Filter	Parse string expressions into structured condition objects
Derafu\Query\Operator	Load, validate, and manage operator definitions from YAML
Derafu\Query\Builder	Build SQL strings and named-parameter arrays from conditions
Derafu\Query\Engine	Execute SQL against a real database connection
Derafu\Query\Bridge	Apply conditions to third-party query builders (Doctrine, Illuminate, etc.)
Derafu\Query\Config	Load declarative query definitions from arrays, YAML, or JSON

Filter Layer

The filter layer turns a raw string expression like

`customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000` into an object tree.

Key Classes

Class	Interface	Role
CompositeExpressionParser	CompositeExpressionParserInterface	Entry point. Parses composite expressions with <code>&&</code> , <code> </code> , <code>()</code> into a tree of conditions.
ExpressionParser	ExpressionParserInterface	Parses a single <code>path?filter</code> string into a Condition.
PathParser	PathParserInterface	Splits the path part (<code>table__column</code>) into Segment objects.
FilterParser	FilterParserInterface	Matches the filter part (<code>>1000</code>) against known operators.
Path	PathInterface	Immutable value object holding an ordered list of Segment objects.
Segment	SegmentInterface	Immutable value object for one path segment: name + options.
Filter	FilterInterface	Immutable value object: operator + raw value string.

Class	Interface	Role
Condition	ConditionInterface	Ties a Path to a Filter. Carries a literal flag.
CompositeCondition	CompositeConditionInterface	AND or OR container of Condition/CompositeCondition objects.

Parsing Pipeline

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

```
"status?=active&&total?>1000"
  |
  ▼ CompositeExpressionParser
CompositeCondition (AND)
├── Condition
│   ├── Path [Segment("status")]
│   └── Filter [Operator("="), value="active"]
└── Condition
    ├── Path [Segment("total")]
    └── Filter [Operator(">"), value="1000"]
```

The `literal` flag on `Condition` distinguishes normal filter values from expression references (used with the `?E` marker for self-referencing conditions like `id?E!=other_alias.id`).

Operator Layer

Operators are defined in `resources/operators.yaml` and loaded at startup. Each operator carries:

- A **symbol** (e.g. `>=`, `like:`, `b&`)
- A **type** classifying its behavior
- Engine-specific **SQL templates** with `{{column}}` / `{{value}}` / `{{values}}` / `{{value_1}}` / `{{value_2}}` placeholders
- An optional **validation pattern** (regex) for the value
- Optional **casting rules** that transform the value before binding
- An optional **alias** pointing to another operator's SQL template

Key Classes

Class	Interface	Role
<code>Operator</code>	<code>OperatorInterface</code>	Immutable value object for one operator definition
<code>OperatorLoader</code>	<code>OperatorLoaderInterface</code>	Reads and validates <code>operators.yaml</code>
<code>OperatorManager</code>	<code>OperatorManagerInterface</code>	Registry; provides operators sorted by symbol length for greedy matching
<code>OperatorManagerFactory</code>	<code>OperatorManagerFactoryInterface</code>	Convenience factory: loader + manager in one call

Builder Layer

The builder layer converts condition objects into SQL strings with named parameters.

Key Classes

Class	Interface	Role
<code>SqlBuilderWhere</code>	<code>QueryBuilderWhereInterface</code>	Converts a single <code>Condition</code> or <code>CompositeCondition</code> into a WHERE fragment + parameters
<code>SqlQueryBuilder</code>	<code>QueryBuilderInterface</code>	Full query builder: SELECT, FROM, JOIN, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET
<code>SqlQuery</code>	<code>QueryInterface</code>	Immutable value object holding SQL string and parameters. Implements <code>ArrayAccess</code> for <code>\$q['sql']</code> and <code>\$q['parameters']</code> .
<code>SqlSanitizerTrait</code>	—	Sanitizes identifiers and expressions; supports a custom quoting callback

SqlBuilderWhere Internals

`SqlBuilderWhere` is constructed with a database driver name (`pgsql`, `mysql`, `sqlite`). For each condition it:

1. Resolves the effective SQL template (operator's own `sql` key, or its base operator's).
2. Validates the raw value against the operator's pattern (if defined).
3. Normalizes the value: booleans → 0/1, arrays → delimiter-joined string.
4. Applies casting rules (e.g. `like_start` appends %, date normalizes `YYYYMMDD` → `YYYY-MM-DD`).
5. Creates named parameters (`param_{column}_{uniqid}`).
6. Replaces `{{column}}`, `{{value}}`, `{{values}}`, `{{value_1}}`, `{{value_2}}` in the template.

For composite conditions it recursively builds each child and joins them with `AND` / `OR` inside parentheses.

For EXISTS paths (starting with `___`) it generates correlated EXISTS (`SELECT 1 FROM ...`) or NOT EXISTS (...) subqueries. Aggregate column segments like `SUM(amount)` produce scalar subqueries: (`SELECT SUM(p.amount) FROM payments p WHERE ...`) `>= :param`.

Column Resolution from Paths

When a path has more than one segment, `SqlBuilderWhere` qualifies the column with the previous segment's alias (or name):

```
authors__books__title → books.title
```

```
authors[alias:a]__books[alias:b]__title → b.title
```

For `SqlQueryBuilder`, intermediate segments automatically become JOIN clauses (INNER by default; override with `join:left` option).

Engine Layer

The engine layer executes the SQL produced by the builder against a real connection.

Class	Interface	Role
<code>PdoEngine</code>	<code>SqlEngineInterface</code>	Wraps a PDO instance; prepares, executes, and fetches rows
<code>DoctrineEngine</code>	<code>SqlEngineInterface</code>	Wraps a Doctrine DBAL Connection
<code>AbstractSqlEngine</code>	—	Shared <code>getDriver()</code> detection logic

Bridge Layer

Bridges apply parsed conditions to existing third-party query builder instances, without requiring the full `SqlQueryBuilder`.

Class	Target QB	Notes
<code>DoctrineDBALQueryBuilderConditionApplier</code>	Doctrine DBAL QueryBuilder	Resolves driver via reflection; handles FROM inference and JOIN deduplication
<code>DoctrineORMQueryBuilderConditionApplier</code>	Doctrine ORM QueryBuilder	Generates DQL; rewrites single-segment paths to qualify with root alias; throws <code>UnsupportedOperatorException</code> for DQL-incompatible operators
<code>IlluminateQueryBuilderConditionApplier</code>	Illuminate Query\Builder / Eloquent\Builder	Accepts both; resolves Eloquent to base via <code>toBase()</code>
<code>SmartFilter</code> (API Platform)	Doctrine ORM QueryBuilder	Wraps <code>DoctrineORMQueryBuilderConditionApplier</code> for use as an API Platform <code>#[QueryParameter]</code> filter

All bridge appliers share `ConditionApplierTrait`, which provides:

- `buildConditionSql()` — compiles to SQL via `SqlBuilderWhere`
- `extractPaths()` — collects all paths from a condition tree
- `inferFromTable()` — detects the base table from multi-segment paths

- `buildJoinSpecsFromPaths()` — builds JOIN specs for SQL-level bridges
 - `buildOrmJoinSpecsFromPaths()` — builds JOIN specs for Doctrine ORM DQL
-

Config Layer

The config layer provides a declarative alternative to the fluent builder API.

Class	Role
<code>QueryConfig</code>	Wraps an array; calls builder methods when <code>applyTo(\$builder)</code> is invoked
<code>YamlConfigLoader</code>	Loads YAML files or strings into arrays
<code>JsonConfigLoader</code>	Loads JSON files or strings into arrays

`QueryConfig::fromFile()` auto-detects YAML vs JSON by file extension.

Data Flow Summary

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

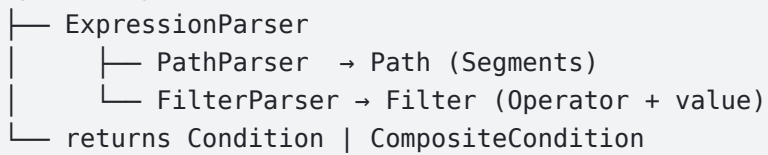
[REDACTED]

[REDACTED]

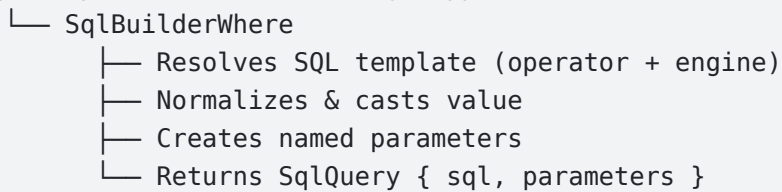
User provides string expression



CompositeExpressionParser



SqlQueryBuilder (or a Bridge applier)



Engine::execute(sql, parameters)



array of result rows

Last updated on 22/09/2026

Docs » Data Handling Category » Query Project » Expression Syntax

Expression Syntax

Expression Syntax

Anonymous · 22/09/2026

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
<code>&&</code>	AND	Higher
<code> </code>	OR	Lower
<code>()</code>	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:null',
  '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 22/09/2026

Path Syntax

Path Syntax

Anonymous · 22/09/2026

Path Syntax

The **path** is the left-hand side of a `path?filter` expression. It identifies which column to apply the filter to, and optionally encodes the JOIN chain required to reach that column.

Simple Paths (Single Segment)

A simple path is just a column name:

```
price
status
created_at
deleted_at
```

When used with `SqlQueryBuilder.where()`, the column is unqualified — it refers to a column in the driving table set via `table()` or `from()`.

When used with the Doctrine ORM bridge, single-segment paths are automatically qualified with the root entity alias (e.g. `status` → `c.status`).

Multi-Segment Paths (Join Chains)

Segments are separated by double underscores `__`. The first segment is the **base table**, intermediate segments are **join targets**, and the last segment is the **column**:

```
customers__name
invoices__customers__name
products__invoice_details__invoices__customers__type
```

```
customers    __    name
^— table      ^— column
```

```
invoices __ customers __ name
^-- table      ^-- join target  ^-- column
```

When `SqlQueryBuilder` processes a multi-segment path, it automatically generates INNER JOIN clauses for all intermediate segments that carry on: options. The base table is inferred from the first segment when no explicit `table()` call has been made.

Column Qualification

`SqlBuilderWhere` qualifies the column with the alias (or name) of the segment immediately before it:

```
invoices__customers__name          → customers.name
invoices[alias:i]__customers[alias:c]__name → c.name
```

Segment Options

Each segment can carry metadata inside square brackets `[key:value, key2:value2]`:

```
customers[alias:c]
invoices[on:id=customer_id,alias:i,join:left]
```

alias:value

Sets an alias for the table in the generated SQL. The alias is used for column qualification, JOIN declarations, and correlation conditions.

```
customers[alias:c]__invoices[alias:i]__total
→ FROM customers AS c INNER JOIN invoices AS i ... → i.total
```

on:left_col=right_col

Defines the JOIN condition between the previous segment and this segment. `left_col` belongs to the previous table; `right_col` belongs to this table.

```
invoices__customers[on:customer_id=id]__name
→ ... INNER JOIN customers ON invoices.customer_id = customers.id
```

Multiple `on:` pairs in the same brackets add multiple conditions (AND):

```
orders__items[on:order_id=id,on:branch_id=branch_id]__product
→ ... INNER JOIN items ON orders.order_id = items.id AND orders.branch_id =
```

```
items.branch_id
```

join:type

Overrides the join type for this segment. Valid values: `inner` (default), `left`, `right`, `cross`.

```
customers__invoices[on:id=customer_id,join:left]__total
→ ... LEFT JOIN invoices ON customers.id = invoices.customer_id
```

Combining Options

All options can be combined in any order, separated by commas:

```
customers[alias:c]__invoices[on:id=customer_id,join:left,alias:i]__total
```

The segment above sets `alias=i`, `on.id=customer_id`, and `join=left` simultaneously.

Exists/Subquery Paths (___)

A path that starts with **triple underscore** `___` generates a correlated subquery instead of a JOIN. This is the mechanism for filtering by the existence or properties of child records.

Basic Existence Check

```
___payments[on:id=invoice_id]
```

No column segment → pure existence check. Combine with `is:empty` or `isnot:empty`:

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

Column Filter Inside the Subquery

Adding a column segment after `___` filters on that column inside the EXISTS:

```
___payments[on:id=invoice_id]__status?=pending
→ EXISTS (SELECT 1 FROM payments WHERE invoices.id = payments.invoice_id AND
payments.status = :p)
```

Any operator can be used for the column filter inside the subquery:

```
__payments[on:id=invoice_id]__amount?>500
__payments[on:id=invoice_id]__method?in:card,transfer
__payments[on:id=invoice_id]__created_at?date:20240101
```

Aggregate Scalar Subqueries

When the column segment is an aggregate function call, a scalar subquery is generated:

```
__payments[on:id=invoice_id]__SUM(amount)?>=1200
→ (SELECT SUM(p.amount) FROM payments p WHERE p.invoice_id = invoices.id) >= :p
```

Supported aggregate functions: `SUM`, `AVG`, `COUNT`, `MIN`, `MAX`.

```
__payments[on:id=invoice_id]__COUNT(*)?>1
__invoices[on:id=customer_id]__AVG(total)?<1000
__invoices[on:id=customer_id]__MIN(total)?>=100
```

COUNT(*) optimizations: when the comparison is a pure existence check, it is automatically rewritten to `EXISTS` / `NOT EXISTS`:

Expression	Rewritten as
<code>COUNT(*)?=0</code>	<code>NOT EXISTS (...)</code>
<code>COUNT(*)?>0</code>	<code>EXISTS (...)</code>
<code>COUNT(*)?!=0</code>	<code>EXISTS (...)</code>
<code>COUNT(*)?>=1</code>	<code>EXISTS (...)</code>
<code>COUNT(*)?<1</code>	<code>NOT EXISTS (...)</code>
<code>COUNT(*)?<=0</code>	<code>NOT EXISTS (...)</code>
<code>COUNT(*)?>1</code>	scalar subquery (kept as-is)

Aliases on Exists Segments

You can add `alias:` to a subquery path segment:

```
__payments[alias:p,on:id=invoice_id]__status?=pending
→ EXISTS (SELECT 1 FROM payments AS p WHERE invoices.id = p.invoice_id AND p.status = :p)
```

Nested Exists (Multi-Level)

Chain multiple `___` separators to traverse deeper relationships:

```
___payments[on:id=invoice_id]___items[on:id=payment_id]___amount?>100
```

This generates an EXISTS subquery with an inner JOIN to `items`, filtering on `items.amount`.

Function Notation in Paths

The **last segment** may be an aggregate or SQL function call. This is used for HAVING conditions and similar:

```
AVG(price)?>500      ← in a having() call
SUM(i.total)?>1000   ← qualified with a prior alias
```

When a parent segment exists, `SqlBuilderWhere` qualifies the function arguments automatically:

```
products[alias:p]___AVG(price)?>500   →  AVG(p.price) > :p
```

Path Validation Rules

`PathParser` enforces these rules:

- A path cannot be empty.
- Segment names cannot be empty — `author___` (trailing `___`) and `___books` (leading `___`) are invalid.
- Segment names must start and end with `[a-zA-Z0-9_]` (closing `)` also allowed at the end, for function calls).
- Option keys and values must both be non-empty.
- An `on:` option value must contain `=` with non-empty parts on both sides.
- Exists paths (`___`) must have a non-empty body after the triple underscore.

Path Examples Reference

Path	Result
<code>status</code>	<code>status</code> (no qualification)
<code>invoices___status</code>	<code>invoices.status</code>

Path	Result
<code>invoices[alias:i]__status</code>	<code>i.status</code>
<code>customers[alias:c]__invoices[on:id=customer_id,alias:i]__total</code>	<code>i.total</code> with JOIN
<code>products[alias:p]__invoice_details[on:id=product_id,alias:id]__invoices[on:invoice_id=id,alias:i]__customers[on:customer_id=id,alias:c]__name</code>	3-level deep join
<code>__payments[on:id=invoice_id]</code>	Subquery path, no column
<code>__payments[on:id=invoice_id]__status</code>	Subquery path with column
<code>__payments[on:id=invoice_id]__SUM(amount)</code>	Aggregate subquery
<code>__payments[on:id=invoice_id]__items[on:id=payment_id]__amount</code>	Nested subquery

Paths in Doctrine ORM

The Doctrine ORM bridge handles paths differently from SQL bridges:

- **Single-segment paths** are automatically qualified with the root entity alias (e.g. `status` → `c.status`).
- **Multi-segment paths** use the association name from Doctrine entity mappings, not SQL column names. The `on:` option is **ignored** — Doctrine resolves join conditions from the mapping.
- **EXISTS paths** use DQL `SIZE(alias.assoc) = 0` for simple emptiness checks, and correlated `EXISTS(SELECT sub.id FROM EntityClass sub WHERE ...)` for column filters.

Last updated on 22/09/2026

Operators Reference

Operators Reference

Anonymous · 22/09/2026

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 <code>%</code> and <code>_</code>)
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
<code>=</code>	Equals	VALUE	<code>{{column}} = {{value}}</code>

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

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
<code>^</code>	Sensitive	VALUE	VALUE%	<code>col LIKE :p</code> (pgsql/sqlite), <code>col LIKE BINARY :p</code> (mysql)
<code>^*</code>	Insensitive	VALUE	VALUE%	<code>col ILIKE :p</code> (pgsql), <code>col LIKE :p</code> (mysql/sqlite)
<code>!^</code>	Sensitive	VALUE	VALUE%	<code>col NOT LIKE :p</code>
<code>!^*</code>	Insensitive	VALUE	VALUE%	<code>col NOT ILIKE :p</code> / <code>col NOT LIKE :p</code>

Contains

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

Ends With

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

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
<code>like:</code>	Sensitive	PATTERN	pgsql: <code>col LIKE :p</code> · mysql: <code>col LIKE BINARY :p</code> · sqlite: <code>col LIKE :p</code>
<code>notlike:</code>	Sensitive	PATTERN	pgsql: <code>col NOT LIKE :p</code> · mysql: <code>col NOT LIKE BINARY :p</code>
<code>ilike:</code>	Insensitive	PATTERN	pgsql: <code>col ILIKE :p</code> · mysql: <code>col LIKE :p</code> · sqlite: <code>col LIKE :p</code>
<code>notilike:</code>	Insensitive	PATTERN	pgsql: <code>col NOT ILIKE :p</code> · mysql/sqlite: <code>col NOT LIKE :p</code>

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 `notlike:` 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
<code>in:</code>	<code>V1,V2,...</code>	<code>{{column}} IN (:p1, :p2, ...)</code>
<code>notin:</code>	<code>V1,V2,...</code>	<code>{{column}} NOT IN (:p1, :p2, ...)</code>

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:YMMM
```

Matches rows in a specific year+month.

Engine	SQL
pgsql	TO_CHAR({{column}}, "YYYYMM") = {{value}}
mysql	DATE_FORMAT({{column}}, "%Y%m") = {{value}}

Engine	SQL
sqlite	<code>strftime("%Y%m", {{column}}) = {{value}}</code>

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

```
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
<code>is:null</code>	—	<code>{{column}} IS NULL</code>
<code><=></code>	<code>is:null</code>	<code>{{column}} IS NULL</code>
<code>isnot:null</code>	—	<code>{{column}} IS NOT NULL</code>

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)
<code>~</code>	Sensitive	<code>{{column}} ~ {{value}}</code>	<code>{{column}} REGEXP BINARY {{value}}</code>
<code>~*</code>	Insensitive	<code>{{column}} ~* {{value}}</code>	<code>{{column}} REGEXP {{value}}</code>
<code>!~</code>	Sensitive	<code>{{column}} !~ {{value}}</code>	<code>{{column}} NOT REGEXP BINARY {{value}}</code>
<code>!~*</code>	Insensitive	<code>{{column}} !~* {{value}}</code>	<code>{{column}} NOT REGEXP {{value}}</code>

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
<code>regexp:</code>	<code>~</code>
<code>notregexp:</code>	<code>!~</code>
<code>rlike:</code>	<code>~</code>
<code>notrlike:</code>	<code>!~</code>

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
<code>~~*</code>	<code>ilike:</code>	auto-casts with <code>like</code>
<code>!~~</code>	<code>notlike:</code>	auto-casts with <code>like</code>
<code>!~~*</code>	<code>notilike:</code>	auto-casts with <code>like</code>
<code>\$</code>	<code>like:</code>	auto-casts with <code>like_end</code>
<code>\$*</code>	<code>ilike:</code>	auto-casts with <code>like_end</code>
<code>!\$</code>	<code>notlike:</code>	auto-casts with <code>like_end</code>
<code>!\$*</code>	<code>notilike:</code>	auto-casts with <code>like_end</code>
<code>regexp:</code>	<code>~</code>	MySQL familiarity
<code>notregexp:</code>	<code>!~</code>	MySQL familiarity
<code>rlike:</code>	<code>~</code>	MySQL familiarity
<code>notrlike:</code>	<code>!~</code>	MySQL familiarity

Casting Rules

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

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

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 22/09/2026

Docs » Data Handling Category » Query Project » Query Builder

Query Builder

Query Builder

Anonymous · 22/09/2026

Query Builder

derafu/query provides two complementary ways to build queries:

1. **Fluent API** (`SqlQueryBuilder`) — chainable method calls, useful in application code.
2. **Declarative Config** (`QueryConfig`) — array, YAML, or JSON definitions, useful for reusable templates and API-driven queries.

Both produce the same output: a `SqlQuery` with an SQL string and named-parameter array.

Setting Up SqlQueryBuilder

`SqlQueryBuilder` requires an engine and an expression parser:

```
use Derafu\Query\Builder\SqlQueryBuilder;
use Derafu\Query\Engine\PdoEngine;
use Derafu\Query\Filter\CompositeExpressionParser;
use Derafu\Query\Filter\ExpressionParser;
use Derafu\Query\Filter\FilterParser;
use Derafu\Query\Filter\PathParser;
use Derafu\Query\Operator\OperatorLoader;
use Derafu\Query\Operator\OperatorManager;

// 1. Set up the operator registry.
$loader = new OperatorLoader();
$manager = new OperatorManager(
    $loader->loadFromFile('vendor/derafu/query/resources/operators.yaml')
);

// 2. Build the expression parser.
$parser = new CompositeExpressionParser(
    new ExpressionParser(new PathParser(), new FilterParser($manager))
);

// 3. Create a database engine.
$pdo = new PDO('pgsql:host=localhost;dbname=mydb', $user, $pass);
$engine = new PdoEngine($pdo);
```

```
// 4. Instantiate the builder.
$qb = new SqlQueryBuilder($engine, $parser);
```

For Doctrine DBAL use `DoctrineEngine` instead of `PdoEngine`. For framework-integrated setups, consider the [bridges](#) instead of `SqlQueryBuilder`.

Fluent API Reference

All mutating methods return `$this` (or a clone via `new()`) so they can be chained.

`table(string $table, ?string $alias = null): self`

Sets the FROM clause and resets the builder state via an internal `new()` clone. Call this first when starting a new query from a builder instance that may have prior state.

```
$qb->table('products')->where('price?>1000')->execute();
$qb->table('invoices', 'i')->select('i.id, i.total')->execute();
```

`from(string $table, ?string $alias = null): self`

Sets the FROM clause without resetting. Use `table()` for new queries; use `from()` internally or when you already have a fresh builder.

`select(string|array $columns, bool $sanitize = true): self`

Sets the SELECT columns. Accepts a comma-separated string or an array.

```
$qb->select('id, name, price');
$qb->select(['id', 'name', 'price']);
$qb->select('c.name AS customer_name, i.total');
$qb->select('COUNT(*) AS total');
$qb->select('DISTINCT type'); // Use distinct() instead for the DISTINCT keyword
```

Pass `$sanitize = false` to skip identifier sanitization (use only for trusted, pre-validated expressions).

`distinct(bool $distinct = true): self`

Adds DISTINCT to the SELECT clause.

```
$qb->table('customers')->select('type')->distinct()->execute();
// → SELECT DISTINCT type FROM customers
```

where(string|array|Condition|CompositeCondition \$condition): self

Sets the `WHERE` clause. Resets any previous `WHERE` conditions. All elements are AND-combined.

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

andWhere(string|array|Condition|CompositeCondition \$condition): self

Appends conditions to the existing `WHERE` with AND. If no `WHERE` exists yet, behaves like `where()`.

```
$qb->where('status?=active')->andWhere('total?>1000');
```

orWhere(string|array|Condition|CompositeCondition \$condition): self

Wraps the existing `WHERE` and the new condition in an OR composite.

```
// WHERE (status = 'electronics' OR category = 'hardware')
$qb->where('category?=electronics')->orWhere('category?=hardware');

// WHERE (category = 'software' OR (category = 'hardware' AND price > 200))
$qb->where('category?=software')->orWhere(['category?=hardware', 'price?>200']);

// Multiple OR groups:
// WHERE (status = 'cancelled' OR status = 'draft' OR (status = 'issued' AND total > 1000))
$qb->where('status?=cancelled')
    ->orWhere(['status?=draft', ['status?=issued', 'total?>1000']]);
```

andWhereOr(array|Condition|CompositeCondition \$conditions): self

Appends an OR composite to the existing AND `WHERE`. Each element of the array is one OR branch; arrays within the array become AND groups.

```
// WHERE status = 'active' AND (type = 'person' OR tax_id LIKE '78%')
$qb->where('status?=active')
    ->andWhereOr(['type?=person', 'tax_id?^78']);

// WHERE status = 'active' AND ((status = 'paid' AND total > 1000) OR (status = 'issued' AND date >= '2024-03-01'))
$qb->where('customer_id?in:1,2')
    ->andWhereOr([
        ['status?=paid', 'total?>1000'],
        ['status?=issued', 'date?>=2024-03-01'],
    ]);
```

```
join(string $table, string $condition, string $type = 'INNER', ?string $alias = null): self
```

Adds an explicit JOIN clause. Valid types: `INNER`, `LEFT`, `RIGHT`, `CROSS`. Duplicates (same table+alias combination) are ignored.

```
$qb->join('customers', 'i.customer_id = c.id', 'INNER', 'c');
```

Convenience wrappers:

```
$qb->innerJoin('customers', 'i.customer_id = c.id', 'c');
$qb->leftJoin('payments', 'i.id = p.invoice_id', 'p');
$qb->rightJoin('invoices', 'c.id = i.customer_id', 'i');
$qb->crossJoin('currencies');
```

Note: When using path expressions in `where()`, joins are generated automatically from the path segments. Manual `join()` calls are only needed when you want explicit control, or when building the query without path expressions.

```
groupBy(string|array $columns): self
```

Adds GROUP BY columns. Multiple calls accumulate.

```
$qb->groupBy('category');
$qb->groupBy(['c.id', 'c.name']);
```

```
having(string|array|Condition|CompositeCondition $condition): self
```

Sets the HAVING clause. Accepts the same expression formats as `where()`.

```
$qb->groupBy('category')->having('AVG(price)?>500');
$qb->groupBy('status')->having('COUNT(*)?>1');
```

```
orderBy(string|array $columns, string $direction = 'ASC'): self
```

Sets ORDER BY. Accepts a column string, or an associative array of column => direction.

```
$qb->orderBy('price', 'DESC');
$qb->orderBy(['category' => 'ASC', 'price' => 'DESC']);
$qb->orderBy(['created_at', 'id']); // defaults to ASC
```

```
limit(int $limit): self / offset(int $offset): self
```

Set pagination. `OFFSET` is only appended when both `LIMIT` and `OFFSET` are set.

```
$qb->limit(20)->offset(40);  
// → LIMIT 20 OFFSET 40
```

```
getQuery(): QueryInterface
```

Builds and returns the SQL without executing. Returns a `SqlQuery` that implements both `QueryInterface` and `ArrayAccess`:

```
$query = $qb->table('products')->where('price?>1000')->getQuery();  
  
echo $query['sql'];           // SELECT * FROM products WHERE (price > :param_price_...)  
echo $query['parameters'];    // ['param_price_...' => '1000']
```

```
execute(): array
```

Builds and executes the query, returning rows as an array of associative arrays.

```
$rows = $qb->table('products')->where('price?>1000')->execute();
```

Complete Query Examples

Simple Filters

```
// Active customers.  
$qb->table('customers')->where('status?=active')->execute();  
  
// Products in a price range.  
$qb->table('products')->where('price?between:100,1000')->execute();  
  
// Soft-deleted records.  
$qb->table('users')->where('deleted_at?isnot:null')->execute();  
  
// March 2024 invoices (SQLite).  
$qb->table('invoices')->where('date?period:202403')->execute();
```

Composite Conditions

```
// AND: software products over $200.
```

```
$qb->table('products')
->where(['category?=software', 'price?>200'])
->execute();

// OR: electronics or hardware.
$qb->table('products')
->where('category?=electronics')
->orWhere('category?=hardware')
->execute();

// AND + OR: active customers who are persons OR whose tax_id starts with "78".
$qb->table('customers')
->where('status?=active')
->andWhereOr(['type?=person', 'tax_id?^78'])
->execute();
```

Path-Based Joins

```
// Auto-generated JOIN from path.
$qb->select('c.name AS customer_name, i.number, i.total')
->where('customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000')
->execute();
// → SELECT ... FROM customers AS c INNER JOIN invoices AS i ON c.id = i.customer_id
WHERE i.total > :p

// Multi-level join.
$qb->select('p.name, i.number, c.name AS customer')
->where([
    'products[alias:p]__category?=electronics',
    'products[alias:p]__invoice_details[on:id=product_id,alias:id]__invoices[on:invoice_id=id,alias:i]__status?=paid',
    'products[alias:p]__invoice_details[on:id=product_id,alias:id]__invoices[on:invoice_id=id,alias:i]__customers[on:customer_id=id,alias:c]__type?=company',
])
->execute();
```

Subquery / EXISTS

```
// Invoices with no payments.
$qb->table('invoices')
->where('___payments[on:id=invoice_id]?is:empty')
->execute();
// → SELECT * FROM invoices WHERE NOT EXISTS (SELECT 1 FROM payments WHERE invoices.id = payments.invoice_id)

// Invoices with at least one pending payment.
$qb->table('invoices')
->where('___payments[on:id=invoice_id]__status?=pending')
->execute();
```

```
// Invoices where total payments >= 1200.
$qb->table('invoices')
    ->where('__payments[on:id=invoice_id]__SUM(amount)?>=1200')
    ->execute();
```

Grouping, Having, Ordering, Pagination

```
// Category stats.
$qb->table('products')
    ->select('category, AVG(price) AS avg_price')
    ->groupBy('category')
    ->having('AVG(price)?>500')
    ->orderBy('avg_price', 'DESC')
    ->limit(10)
    ->execute();

// Paginated customer list.
$qb->table('customers')
    ->orderBy(['name' => 'ASC'])
    ->limit(25)
    ->offset(50)
    ->execute();
```

Explicit Joins

```
// Manual JOIN for a complex ON clause.
$qb->table('invoices', 'i')
    ->select('i.id, i.number, i.total, c.name AS customer_name')
    ->innerJoin('customers', 'i.customer_id = c.id', 'c')
    ->where('i.status?=paid')
    ->execute();

// LEFT JOIN with GROUP BY.
$qb->table('customers', 'c')
    ->select('c.name, COUNT(i.id) AS invoice_count')
    ->leftJoin('invoices', 'c.id = i.customer_id', 'i')
    ->groupBy(['c.id', 'c.name'])
    ->execute();
```

Declarative Query Configuration

`QueryConfig` provides an array-based way to describe a query. It is particularly useful for storing query templates in files and for API-driven queries.

Basic Usage

```
use Derafu\Query\Config\QueryConfig;

$config = new QueryConfig([
    'table'    => 'products',
    'select'   => 'id, name, price',
    'where'    => 'category?=electronics',
    'orderBy'  => ['price' => 'DESC'],
    'limit'    => 10,
]);

$result = $config->applyTo($qb)->execute();
```

`applyTo()` calls the corresponding builder methods in order and returns the modified builder, so you can chain additional calls after:

```
$builder = $config->applyTo($qb);
if ($extraFilter) {
    $builder->andWhere('status?=active');
}
$rows = $builder->execute();
```

Loading from Files

```
// YAML file.
$config = QueryConfig::fromYamlFile('/path/to/queries/product_report.yaml');

// JSON file.
$config = QueryConfig::fromJsonFile('/path/to/queries/sales.json');

// Auto-detect by extension (.yaml, .yml, .json).
$config = QueryConfig::fromFile('/path/to/query.yaml');

// From strings.
$config = QueryConfig::fromYamlString($yamlString);
$config = QueryConfig::fromJsonString($jsonString);
```

YAML Template Example

```
# recent_products.yaml
table: products
select: id, name, price, created_at
where: deleted_at?is:null
orderBy:
    created_at: DESC
limit: 20
```

```

$config = QueryConfig::fromYamlFile('queries/recent_products.yaml');
$builder = $config->applyTo($qb);

// Add extra filter at runtime.
if ($category) {
    $builder->andWhere('category?=' . $category);
}

$rows = $builder->execute();

```

Complete Configuration Reference

All QueryConfig keys and their equivalent builder calls:

Key	Builder Method	Value Type
table	table()	string
alias	table(\$t, \$alias)	string
select	select()	string or array
distinct	distinct(true)	boolean
where	where()	string, array, or nested
andWhere	andWhere()	string or array
orWhere	orWhere()	string or array
andWhereOr	andWhereOr()	array of arrays
innerJoin	innerJoin()	{table, condition, alias?}
leftJoin	leftJoin()	{table, condition, alias?}
rightJoin	rightJoin()	{table, condition, alias?}
crossJoin	crossJoin()	{table, alias?}
groupBy	groupBy()	string or array
having	having()	string or array
orderBy	orderBy()	{column: direction}
limit	limit()	integer
offset	offset()	integer

```

$config = new QueryConfig([
    'table'    => 'invoices',
    'alias'    => 'i',
    'select'   => 'i.id, i.number, c.name AS customer_name',
]);

```

```

    'distinct' => true,

    'where'      => 'i.status?=paid',
    'andWhere'   => 'i.total?>1000',
    'orWhere'    => 'i.date?period:202403',
    'andWhereOr' => [
        ['i.category?=service', 'i.total?>500'],
        ['i.category?=product', 'i.total?>1000'],
    ],

    'innerJoin' => ['table' => 'customers', 'alias' => 'c', 'condition' =>
        'i.customer_id = c.id'],

    'groupBy' => ['i.status'],
    'having'   => 'COUNT(*)?>1',
    'orderBy' => ['i.created_at' => 'DESC'],
    'limit'    => 20,
    'offset'   => 40,
]);

```

API-Driven Queries

QueryConfig is well-suited for accepting query parameters from an API:

```

$requestData = $request->getJsonBody();

$config = new QueryConfig([
    'table'    => 'products',
    'where'    => $requestData['filters'] ?? [],
    'orderBy'  => $requestData['sort'] ?? ['id' => 'ASC'],
    'limit'    => min($requestData['limit'] ?? 20, 100),
    'offset'   => max($requestData['offset'] ?? 0, 0),
]);

$rows = $config->applyTo($qb)->execute();

```

Ensure the `where` filters are either pre-validated strings that only allow known column names, or use the full path syntax so that only correctly-structured expressions reach the parser.

Last updated on 22/09/2026

Docs » Data Handling Category » Query Project » Framework Bridges

Framework Bridges

Framework Bridges

Anonymous · 22/09/2026

Framework Bridges

Bridges allow you to apply derafu/query expressions to existing third-party query builder instances — without replacing them. You keep your existing query builder and add Derafu filters on top.

All bridges accept a parsed `ConditionInterface` | `CompositeConditionInterface` and apply it to the query builder. You parse expressions with `CompositeExpressionParser` and pass the result to the bridge.

Shared Setup: The Expression Parser

All bridges share the same expression parser setup:

```
use Derafu\Query\Filter\CompositeExpressionParser;
use Derafu\Query\Filter\ExpressionParser;
use Derafu\Query\Filter\FilterParser;
use Derafu\Query\Filter\PathParser;
use Derafu\Query\Operator\OperatorLoader;
use Derafu\Query\Operator\OperatorManager;

$loader = new OperatorLoader();
$manager = new OperatorManager(
    $loader->loadFromFile('vendor/derafu/query/resources/operators.yaml')
);
$parser = new CompositeExpressionParser(
    new ExpressionParser(new PathParser(), new FilterParser($manager))
);
```

Doctrine DBAL Bridge

Class: `Derafu\Query\Bridge\DoctrineDBALQueryBuilderConditionApplier`

Applies conditions to a Doctrine DBAL `QueryBuilder`. Supports all operators including date, regex, and

bitwise operators.

Methods

```
public function apply(object $queryBuilder,
ConditionInterface|CompositeConditionInterface $condition): void
public function applyHaving(object $queryBuilder,
ConditionInterface|CompositeConditionInterface $condition): void
```

Usage

```
use Derafu\Query\Bridge\DoctrineDBALQueryBuilderConditionApplier;

$applier = new DoctrineDBALQueryBuilderConditionApplier();

// Parse the expression.
$condition = $parser->parse('status?=active&&total?>1000');

// Apply to an existing DBAL QueryBuilder.
$applier->apply($dbalQb, $condition);

// The QB now has: WHERE (status = :param_status_... AND total > :param_total_...)
$rows = $dbalQb->executeQuery()->fetchAllAssociative();
```

FROM and JOIN Inference

When a condition uses multi-segment paths, the bridge:

1. **Infers FROM** from the first segment of the first multi-segment path, if no **FROM** has been set yet.
2. **Generates JOINS** from intermediate segments, with deduplication (same alias → skip).

```
// No from() call needed – it's inferred from the path.
$condition = $parser->parse(
    'customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000'
);
$applier->apply($dbalQb, $condition);
// Sets FROM customers AS c, adds INNER JOIN invoices AS i ON c.id = i.customer_id
// WHERE i.total > :p
```

Driver Detection

The bridge reads the database platform from the DBAL connection (via reflection on the internal connection property) and maps it to the driver name used by `QueryBuilderWhere`:

Platform	Driver
PostgreSQLPlatform	pgsql
MySQLPlatform	mysql
SQLitePlatform	sqlite
SQLServerPlatform	sqlsrv
OraclePlatform	oci
Other	pgsql (fallback)

HAVING

```
$dbalQb->groupBy('status');
$condition = $parser->parse('COUNT(*)?>1');
$applier->applyHaving($dbalQb, $condition);
```

Doctrine ORM Bridge

Class: Derafu\Query\Bridge\DoctrineORMQueryBuilderConditionApplier

Applies conditions to a Doctrine ORM `QueryBuilder` as DQL. This bridge generates DQL-compatible fragments — it does not produce raw SQL.

Methods

```
public function apply(object $queryBuilder,
    ConditionInterface|CompositeConditionInterface $condition): void
public function applyHaving(object $queryBuilder,
    ConditionInterface|CompositeConditionInterface $condition): void
```

Usage

```
use Derafu\Query\Bridge\DoctrineORMQueryBuilderConditionApplier;

$applier = new DoctrineORMQueryBuilderConditionApplier();

$condition = $parser->parse('status?=active&&total?>1000');

$formQb = $em->createQueryBuilder()
    ->select('i')
    ->from(Invoice::class, 'i');

$applier->apply($formQb, $condition);
```

```
$invoices = $ormQb->getQuery()->getResult();
```

Single-Segment Path Qualification

Single-segment paths like `status` are automatically qualified with the root entity alias. The root alias is read from the first FROM clause:

```
status?=active → i.status = :param_status_... (when root alias is 'i')
```

Multi-segment paths are left as-is, allowing explicit alias control.

JOIN Generation

The ORM bridge adds JOINS to the QueryBuilder using Doctrine's association names — the `on:` option in path segments is **ignored**. Doctrine derives join conditions from entity mappings.

```
invoices[alias:i]__payments[alias:p]__status
```

If `Invoice` has a `payments` association, the bridge calls:

```
$ormQb->innerJoin('i.payments', 'p');
```

EXISTS Paths in DQL

Subquery paths (___) are translated to DQL:

Expression	DQL
___payments?is:empty	SIZE(i.payments) = 0
___payments?isnot:empty	SIZE(i.payments) > 0
___payments__status?=pending	EXISTS(SELECT __payments0.id FROM App\Entity\Payment __payments0 WHERE __payments0.invoice = i AND __payments0.status = :p)
___invoices__AVG(total)?<1000	(SELECT AVG(__i0.total) FROM App\Entity\Invoice __i0 WHERE __i0.customer = i) < :p
___invoices__COUNT(*)?=0	SIZE(i.invoices) = 0 (optimized)

DQL-Incompatible Operators

The following operator types throw `UnsupportedOperatorException` when used with this bridge:

- **date type:** `date:`, `month:`, `year:`, `period:` — require SQL functions not available in DQL.
- **binary type:** `b&`, `b|`, `b^`, etc. — bitwise SQL not supported in DQL.

- **regexp type:** `~`, `~*`, `similar to:`, etc. — database-specific regex not in DQL.
- **ilike:** and **not ilike:** — ILIKE is not DQL.

In the API Platform `SmartFilter`, these exceptions are caught and the filter is silently skipped.

Illuminate (Laravel) Bridge

Class: `Derafu\Query\Bridge\IlluminateQueryBuilderConditionApplier`

Applies conditions to Illuminate's `Query\Builder` or Eloquent's `Builder`. Both are accepted — Eloquent builders are resolved to their underlying `Query\Builder` via `toBase()`.

Methods

```
public function apply(object $queryBuilder,
    ConditionInterface|CompositeConditionInterface $condition): void
public function applyHaving(object $queryBuilder,
    ConditionInterface|CompositeConditionInterface $condition): void
```

Usage

```
use Derafu\Query\Bridge\IlluminateQueryBuilderConditionApplier;

$applier = new IlluminateQueryBuilderConditionApplier();

$condition = $parser->parse('status?=active&&total?>1000');

// With a plain Query\Builder.
$qb = DB::table('invoices');
$applier->apply($qb, $condition);
$rows = $qb->get();

// With an Eloquent Builder.
$builder = Invoice::query();
$applier->apply($builder, $condition);
$invoices = $builder->get();
```

FROM and JOIN Inference

Same behavior as the DBAL bridge: infers the `FROM` table from multi-segment paths and adds `join()`, `leftJoin()`, or `rightJoin()` calls as needed. Duplicate joins (same table reference) are skipped.

Driver Detection

The bridge reads the driver name directly from `Connection::getDriverName()`:

Driver name	SQL generated
pgsql	PostgreSQL SQL
mysql	MySQL SQL
sqlite	SQLite SQL

HAVING

```
$condition = $parser->parse('AVG(price)?>500');
$applier->applyHaving($qb, $condition);
// Calls: $qb->havingRaw($sql, $params)
```

API Platform Bridge

Class: `Derafu\Query\Bridge\ApiPlatform\SmartFilter`

Integrates derafu/query as an API Platform filter via the `QueryParam` attribute. When a request comes in, the filter builds a Derafu expression from the parameter's property name and value, parses it, and applies it to the ORM `QueryBuilder`.

Registration

```
use ApiPlatform\Metadata\ApiResource;
use ApiPlatform\Metadata\QueryParam;
use Derafu\Query\Bridge\ApiPlatform\SmartFilter;

#[ApiResource]
#[QueryParam(key: 'price', property: 'price', filter: SmartFilter::class)]
#[QueryParam(key: 'status', property: 'status', filter: SmartFilter::class)]
class Product
{
    // ...
}
```

URL Filter Syntax

The URL parameter value is the filter part (operator + value); the property name provides the path segment:

```
GET /api/products?price=>1000&status=in:paid,issued
```

The `SmartFilter` builds expressions internally:

- `price` property + `>1000` value → expression `price?>1000`
- `status` property + `in:paid,issued` value → expression `status?in:paid,issued`

Generic SmartFilter Property

For a single parameter that accepts a full composite expression, use the special `__derafu_smart_filter` property name:

```
#[QueryParameter(key: 'filter', property: '__derafu_smart_filter', filter: SmartFilter::class)]
```

```
GET /api/products?filter=status?=active&&price?>1000
```

Dependency Injection (Symfony)

Register `CompositeExpressionParser` and `DoctrineORMQueryBuilderConditionApplier` as services and inject them into `SmartFilter`:

```
# services.yaml
services:
    Derafu\Query\Bridge\ApiPlatform\SmartFilter:
        arguments:
            $compositeParser:
                '@Derafu\Query\Filter\Contract\CompositeExpressionParserInterface'
            $applier:
                '@Derafu\Query\Bridge\DoctrineORMQueryBuilderConditionApplier'
```

Error Handling

`SmartFilter` silently catches two exceptions:

- **UnsupportedOperatorException**: DQL-incompatible operators (`date:`, `b&`, `ilike:`, `regex`) — the filter is skipped.
- **Any Throwable**: Malformed expressions — the filter is skipped.

This prevents a bad filter parameter from crashing the entire collection endpoint.

Bridge Comparison

Feature	Doctrine DBAL	Doctrine ORM	Illuminate
SQL operators (all)	☐	☐†	☐
Date operators	☐	☐	☐
Regex operators	☐	☐	☐
Bitwise operators	☐	☐	☐
ilike: / notilike:	☐	☐	☐
FROM inference from paths	☐	☐‡	☐
JOIN from path segments	☐	☐ (ORM assoc)	☐
EXISTS paths	☐	☐ (SIZE/EXISTS DQL)	☐
Aggregate subqueries	☐	☐	☐

† Compatible operators only — DQL-incompatible operators throw `UnsupportedOperatorException`.

‡ ORM bridge requires an explicit `FROM / from()` call before applying conditions.

Last updated on 22/09/2026

Security Guide

Security Guide

Anonymous · 22/09/2026

Security Guide

derafu/query uses prepared statements for all filter values. This page explains the security model, what is and is not protected, and how to use the library safely.

How Values Are Protected

All filter values — the part after the operator symbol in `column?operatorVALUE` — become **named parameters** in the generated SQL:

```
price?>1000
→ price > :param_price_5f3a...
   with parameter binding: :param_price_5f3a... = '1000'
```

The SQL string itself never contains the user-supplied value. It goes through PDO/Doctrine/Illuminate's prepared statement mechanism, which prevents SQL injection in values.

This holds for every operator type: standard comparisons, LIKE patterns, lists, ranges, dates, regex values, and bitwise operands are all bound as parameters.

What SqlSanitizerTrait Protects

SqlSanitizerTrait is used for **SQL identifiers** — column names, table names, aliases, and aggregate function names. It strips all characters except `[a-zA-Z0-9_]` from simple identifiers.

Safe uses (identifiers derived from path segments or select expressions):

```
$qb->select('column_name');           // simple identifier
$qb->select('table.column');           // qualified identifier
$qb->select('column AS alias');        // with alias
$qb->select('COUNT(*) AS total');     // aggregate function
$qb->select('SUM(price) AS total');    // aggregate with column
```

Cases requiring attention:

```
// Complex expressions – arithmetic operators are detected and left as-is.
$qb->select('price * quantity AS total');

// Multi-argument functions – outer function is sanitized; inner args are handled
separately.
$qb->select('COALESCE(column1, column2, 0) AS result');
```

The sanitizer is a defense-in-depth measure for identifier injection, not a substitute for input validation. It strips unexpected characters but cannot reason about business logic (e.g. it does not know which columns a user is allowed to access).

What Is NOT Automatically Protected

Explicit JOIN Conditions

The `join()` / `innerJoin()` / `leftJoin()` / `rightJoin()` methods accept a raw `$condition` string that is inserted verbatim into the SQL:

```
// This condition is NOT sanitized.
$qb->innerJoin('customers', 'i.customer_id = c.id', 'c');
```

Never pass user-supplied input directly as a join condition. Use path-based syntax in `where()` instead — those are parsed and sanitized:

```
// Safe: join condition comes from the path parser.
$qb->where('invoices[alias:i]__customers[on:customer_id=id,alias:c]__name?isnot:null')
;
```

The `?E` Expression Marker

When a filter expression uses `?E` (instead of `?`), the value is treated as a column reference and sanitized as an identifier — **not** bound as a parameter:

```
id?E!=other_alias.id
→ id != other_alias.id (identifier, no parameter binding)
```

Only use `?E` for trusted, controlled values (e.g. comparing two known column aliases). Never pass user input as a `?E` value.

`select()` with `$sanitize = false`

```
$qb->select($trustedExpression, sanitize: false);
```

When `$sanitize` is `false`, the expression is inserted without processing. Only use this for pre-validated, application-controlled strings.

Parameters vs. Identifiers

The key distinction in SQL security:

Type	Example	How it's handled
Parameter (value)	price?>1000 — the 1000	Bound via prepared statement
Identifier (column/table)	Path segment names, <code>select()</code> columns	Sanitized by <code>SqlSanitizerTrait</code>

Never try to pass a value as an identifier or vice versa.

Operator Value Validation

Before SQL generation, `FilterParser` validates the raw value against each operator's `pattern` (if defined). Invalid values throw `InvalidArgumentException` immediately:

```
$parser->parse('price?date:not-a-date'); // throws: pattern mismatch
$parser->parse('flags?b&1.5');           // throws: decimal not allowed for binary
op
$parser->parse('status?in:');             // throws: empty list
```

This prevents malformed inputs from reaching the SQL layer, even though the actual injection risk is eliminated by parameter binding.

Input Validation Best Practices

1. **Whitelist columns:** If you accept filter column names from user input (e.g. in an API), validate them against a known-safe list before passing to the expression parser.

2. **Validate operator intent:** Consider whether a user should be allowed to use all operators. For example, regex operators can cause high-CPU queries on large tables; restrict them if necessary.
3. **Limit list sizes:** The `in:` and `notin:` operators accept arbitrarily long lists. Consider validating or capping list length before parsing.
4. **Principle of least privilege:** The database user used by your application should only have `SELECT` (and only `INSERT/UPDATE/DELETE` where needed). A read-only connection cannot be abused into DDL statements even if SQL injection were somehow possible.
5. **Audit HAVING and JOIN conditions:** `having()` accepts the same expression format as `where()` and is equally safe. Explicit `join()` conditions are not sanitized — see above.
6. **Log and monitor:** Log queries that raise `InvalidArgumentException` — they may indicate probing attempts.

Composite Expression Safety

Composite expressions (`&&`, `||`, `()`) are parsed structurally, not evaluated as SQL. The parser splits on `&&` and `||` at parenthesis depth 0 and recurses — it does not execute or interpolate anything. There is no risk of injection through the composite syntax itself.

```
A?=1&&B?=2    → two separate bound parameters
(A?=1||B?=2)  → same, with OR grouping
```

The value inside each leaf expression is always bound as a parameter (unless `?E` is used — see above).

Summary

Area	Protection
Filter values	□ Prepared statement parameters
Column/table identifiers	□ <code>SqlSanitizerTrait</code> stripping
Operator validation patterns	□ Regex validation at parse time

Area	Protection
Composite expression parsing	□ Structural parsing, no SQL eval
Explicit <code>join()</code> conditions	⚠ Not sanitized — keep application-controlled
?E expression references	⚠ Sanitized as identifier — keep application-controlled
<code>select(\$expr, sanitize: false)</code>	⚠ Raw insert — keep application-controlled
Column name whitelisting	□ Application responsibility
Operator allowlisting	□ Application responsibility

Last updated on 22/09/2026

Query Config

Declarative Query Configuration

Anonymous · 22/09/2026

Declarative Query Configuration

The QueryConfig class provides a configuration-based alternative to the fluent builder API. It is covered in detail in the [Query Builder](#) page, under the **Declarative Query Configuration** section.

Quick Reference

```
use Derafu\Query\Config\QueryConfig;

// From an array.
$config = new QueryConfig([
    'table'    => 'products',
    'select'   => 'id, name, price',
    'where'    => 'category?=electronics',
    'orderBy'  => ['price' => 'DESC'],
    'limit'    => 10,
]);
$result = $config->applyTo($queryBuilder)->execute();

// From a YAML file.
$config = QueryConfig::fromYamlFile('queries/product_report.yaml');

// From a JSON file.
$config = QueryConfig::fromJsonFile('queries/sales.json');

// Auto-detect format by extension.
$config = QueryConfig::fromFile('queries/report.yaml');
```

Supported Configuration Keys

Key	Description
table	Table name (FROM clause)
alias	Table alias
select	Columns to select
distinct	true to add DISTINCT

Key	Description
where	WHERE condition(s)
andWhere	Additional AND condition(s)
orWhere	OR condition(s)
andWhereOr	AND with nested OR groups
innerJoin	{table, condition, alias?}
leftJoin	{table, condition, alias?}
rightJoin	{table, condition, alias?}
crossJoin	{table, alias?}
groupBy	GROUP BY column(s)
having	HAVING condition(s)
orderBy	{column: direction} pairs
limit	Max rows to return
offset	Rows to skip

For full documentation, examples, and API-driven query patterns, see [Query Builder → Declarative Query Configuration](#).

Last updated on 22/09/2026