Ecto
▾ v3.14.3

GUIDES MODULES MIX TASKS

QUERY APIS

Ecto.Query.API

Sections +
Summary
Functions –

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

# Ecto.Query.API

Lists all functions allowed in the query API.

- Comparison operators: `==`, `!=`, `<=`, `>=`, `<`, `>`
- Arithmetic operators: `+`, `-`, `*`, `/`
- Boolean operators: `and`, `or`, `not`
- Inclusion operator: `in/2`
- Subquery operators: `any`, `all` and `exists`
- Search functions: `like/2` and `ilike/2`
- Null check functions: `is_nil/1`
- Aggregates: `count/0`, `count/1`, `avg/1`, `sum/1`, `min/1`, `max/1`
- Date/time intervals: `datetime_add/3`, `date_add/3`, `from_now/2`, `ago/2`
- Inside select: `struct/2`, `map/2`, `merge/2`, `selected_as/2` and literals (map, tuples, lists, etc)
- General: `fragment/1`, `field/2`, `type/2`, `as/1`, `parent_as/1`, `selected_as/1`

Note the functions in this module exist for documentation purposes and one should never need to invoke them directly. Furthermore, it is possible to define your own macros and use them in Ecto queries (see docs for `fragment/1`).

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

## Intervals

Ecto supports following values for `interval` option: `"year"`, `"month"`, `"week"`, `"day"`, `"hour"`, `"minute"`, `"second"`, `"millisecond"`, and `"microsecond"`.

`Date`/`Time` functions like `datetime_add/3`, `date_add/3`, `from_now/2`, `ago/2` take `interval` as an argument.

## Window API

Ecto also supports many of the windows functions found in SQL databases. See `Ecto.Query.WindowAPI` for more information.

## About the arithmetic operators

The Ecto implementation of these operators provide only a thin layer above the adapters. So if your adapter allows you to use them in a certain way (like adding a date and an interval in PostgreSQL), it should work just fine in Ecto queries.

## Summary

### Functions

`left * right`

Binary `*` operation.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

**`left + right`**

Binary `+` operation.

**`left - right`**

Binary `-` operation.

**`left / right`**

Binary `/` operation.

**`left != right`**

Binary `!=` operation.

**`left < right`**

Binary `<` operation.

**`left <= right`**

Binary `<=` operation.

**`left == right`**

Binary `==` operation.

**`left > right`**

Binary `>` operation.

**`left >= right`**

Binary `>=` operation.

**`ago(count, interval)`**

Subtracts the given interval from the current time in UTC.

**`all(subquery)`**

Evaluates whether all values returned from the provided subquery match in a comparison operation.

**`left and right`**

Binary `and` operation.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

`any(subquery)`

Tests whether one or more values returned from the provided subquery match in a comparison operation.

`as(binding)`

Refer to a named atom binding.

`avg(value)`

Calculates the average for the given entry.

`coalesce(value, expr)`

Takes the first value which is not null, or null if they both are.

`constant(value)`

Allows a dynamic string or number to be injected into a fragment

`count()`

Counts the entries in the table.

`count(value)`

Counts the given entry.

`count(value, atom)`

Counts the distinct values in given entry.

`date_add(date, count, interval)`

Adds a given interval to a date.

`datetime_add(datetime, count, interval)`

Adds a given interval to a datetime.

`exists(subquery)`

Evaluates to true if the provided subquery returns 1 or more rows.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

**`field(source, field)`**

Allows a field to be dynamically accessed.

**`filter(value, filter)`**

Applies the given expression as a FILTER clause against an aggregate. This is currently only supported by Postgres.

**`fragment(fragments)`**

Send fragments directly to the database.

**`from_now(count, interval)`**

Adds the given interval to the current time in UTC.

**`identifier(binary)`**

Allows a dynamic identifier to be injected into a fragment

**`ilike(string, search)`**

Searches for `search` in `string` in a case insensitive fashion.

**`left in right`**

Checks if the left-value is included in the right one.

**`is_nil(value)`**

Checks if the given value is nil.

**`json_extract_path(json_field, path)`**

Returns value from the `json_field` pointed to by `path`.

**`like(string, search)`**

Searches for `search` in `string`.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

**`map(source, fields)`**

Used in `select` to specify which fields should be returned as a map.

**`max(value)`**

Calculates the maximum for the given entry.

**`merge(left_map, right_map)`**

Merges the map on the right over the map on the left.

**`min(value)`**

Calculates the minimum for the given entry.

**`not value`**

Unary `not` operation.

**`left or right`**

Binary `or` operation.

**`parent_as(binding)`**

Refer to a named atom binding in the parent query.

**`selected_as(name)`**

Refer to an alias of a selected value.

**`selected_as(selected_value, name)`**

Creates an alias for the given selected value.

**`splice(list)`**

Allows a list argument to be spliced into a fragment.

**`struct(source, fields)`**

Used in `select` to specify which struct fields should be returned.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

`sum(value)`

Calculates the sum for the given entry.

`type(interpolated_value, type)`

Casts the given value to the given type at the database level.

`values(values, types)`

Creates a values list/constant table.

## Functions

```
left * right
```

Binary `*` operation.

```
left + right
```

Binary `+` operation.

```
left - right
```

Binary `-` operation.

```
left / right
```

Binary `/` operation.

```
left != right
```

Binary `!=` operation.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
left < right
```

Binary `<` operation.

```
left <= right
```

Binary `<=` operation.

```
left == right
```

Binary `==` operation.

```
left > right
```

Binary `>` operation.

```
left >= right
```

Binary `>=` operation.

```
ago(count, interval)
```

Subtracts the given interval from the current time in UTC.

The current time in UTC is retrieved from Elixir and not from the database.

See Intervals for supported `interval` values.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

## Examples

```
from p in Post, where:
p.published_at > ago(3, "month")
```

**all(subquery)** </>

Evaluates whether all values returned from the provided subquery match in a comparison operation.

```
from p in Post, where: p.visits
>= all(
  from(p in Post, select:
avg(p.visits), group_by:
[p.category_id])
)
```

For a post to match in the above example it must be visited at least as much as the average post in all categories.

```
from p in Post, where: p.visits
== all(
  from(p in Post, select:
max(p.visits))
)
```

The above example matches all the posts which are tied for being the most visited.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

Both `any` and `all` must be given a subquery as an argument, and they must be used on the right hand side of a comparison. Both can be used with every comparison operator: `==`, `!=`, `>`, `>=`, `<`, `<=`.

### `left and right` </>

Binary `and` operation.

### `any(subquery)` </>

Tests whether one or more values returned from the provided subquery match in a comparison operation.

```
from p in Product, where: p.id
== any(
  from(li in LineItem, select:
[li.product_id], where:
li.created_at > ^since and
li.qty >= 10)
)
```

A product matches in the above example if a line item was created since the provided date where the customer purchased at least 10 units.

Both `any` and `all` must be given a subquery as an argument, and they must be used on the right hand side

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

of a comparison. Both can be used with every comparison operator: `==`, `!=`, `>`, `>=`, `<`, `<=`.

### `as(binding)` </>

Refer to a named atom binding.

See Named Bindings for more information.

### `avg(value)` </>

Calculates the average for the given entry.

```
from p in Payment, select:
avg(p.value)
```

### `coalesce(value, expr)` </>

Takes the first value which is not null, or null if they both are.

In SQL, COALESCE takes any number of arguments, but in ecto it only takes two, so it must be chained to achieve the same effect.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from p in Payment, select:
p.value |>
coalesce(p.backup_value) |>
coalesce(0)
```

### `constant(value)` </>

Allows a dynamic string or number to be injected into a fragment:

```
limit = 10
"posts" |> select([p], p.title)
|> limit(fragment("?",
constant(^limit)))
```

The example above will inject the value of `limit` directly into the query instead of treating it as a query parameter. It will generate a query such as `SELECT p0.title FROM "posts" AS p0 LIMIT 1` as opposed to `SELECT p0.title FROM "posts" AS p0 LIMIT $1`.

Note that each different value of `limit` will emit a different query. For this reason, queries with interpolated constants are not cached.

### `count()` </>

Counts the entries in the table.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from p in Post, select: count()
```

**count(value)** </>

Counts the given entry.

```
from p in Post, select:
count(p.id)
```

**count(value, atom)** </>

Counts the distinct values in given entry.

```
from p in Post, select:
count(p.id, :distinct)
```

**date_add(date, count, interval)** </>

Adds a given interval to a date.

See `datetime_add/3` for more information.

See Intervals for supported `interval` values.

**datetime_add(datetime, count, interval)** </>

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

Adds a given interval to a datetime.

The first argument is a `datetime`, the second one is the count for the interval, which may be either positive or negative and the interval value:

```
# Get all items published since
the last month
from p in Post, where:
p.published_at >

datetime_add(^NaiveDateTime.utc_no
-1, "month")
```

In the example above, we used `datetime_add/3` to subtract one month from the current datetime and compared it with the `p.published_at`. If you want to perform operations on date, `date_add/3` could be used.

See Intervals for supported `interval` values.

**`exists(subquery)`** </>

Evaluates to true if the provided subquery returns 1 or more rows.

Ecto

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from p in Post,
  as: :post,
  where:
    exists(
      from(
        c in Comment,
        where:
parent_as(:post).id ==
c.post_id and c.replies_count >
5,
        select: 1
      )
    )
```

This is best used in conjunction with `parent_as/1` to correlate the subquery with the parent query to test some condition on related rows in a different table. In the above example the query returns posts which have at least one comment that has more than 5 replies.

### `field(source, field)` </>

Allows a field to be dynamically accessed.

The source name can be a binding (`p` in `from p in Post`) or a named binding using `as/1` or `parent_as/1`. The named binding maybe a literal atom or an interpolation.

The field name can be given as either an atom or a string. In a

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

schemaless query, the two types of names behave the same. However, when referencing a field from a schema the behaviours are different.

Using an atom to reference a schema field will inherit all the properties from the schema. For example, the field name will be changed to the value of `:source` before generating the final query and its type behaviour will be dictated by the one specified in the schema.

Using a string to reference a schema field is equivalent to bypassing all of the above and accessing the field directly from the source (i.e. the underlying table). This means the name will not be changed to the value of `:source` and the type behaviour will be dictated by the underlying driver (e.g. Postgrex or MyXQL).

Take the following schema and query:

Ecto

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```elixir
defmodule Car do
  use Ecto.Schema

  schema "cars" do
    field :doors, source:
:num_doors
    field :tires, source:
:num_tires
  end
end

def
at_least_four(doors_or_tires) do
  from c in Car,
    where: field(c,
^doors_or_tires) >= 4
end

def at_least_four(query,
doors_or_tires) do
  from q in query,
    where: field(as(:car),
^doors_or_tires) >= 4
end

def at_least_four(query,
binding, doors_or_tires) do
  from q in query,
    where: field(as(^binding),
^doors_or_tires) >= 4
end
```

In the example above, `at_least_four(:doors)` and `at_least_four("num_doors")` would be valid ways to return the set of cars having at least 4 doors.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

String names can be particularly useful when your application is dynamically generating many schemaless queries at runtime and you want to avoid creating a large number of atoms.

### `filter(value, filter)` </>

Applies the given expression as a FILTER clause against an aggregate. This is currently only supported by Postgres.

```
from p in Payment, select:
filter(avg(p.value), p.value >
0 and p.value < 100)

from p in Payment, select:
avg(p.value) |> filter(p.value
< 0)
```

### `fragment(fragments)` </>

Send fragments directly to the database.

It is not possible to represent all possible database queries using Ecto's query syntax. When such is required, it is possible to use fragments to send any expression to the database:

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
def unpublished_by_title(title)
do
  from p in Post,
    where:
is_nil(p.published_at) and
           fragment("lower(?)",
p.title) == ^title
end
```

Every occurrence of the `?` character will be interpreted as a place for parameters, which must be given as additional arguments to `fragment`. If the literal character `?` is required as part of the fragment, it can be escaped with `\\?` (one escape for strings, another for fragment).

In the example above, we are using the lower procedure in the database to downcase the title column.

It is very important to keep in mind that Ecto is unable to do any type casting when fragments are used. Therefore it may be necessary to explicitly cast parameters via `type/2`:

```
fragment("lower(?)", p.title)
== type(^title, :string)
```

## Identifiers and Constants

Sometimes you need to interpolate an identifier or a constant value into

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

a fragment, instead of a query parameter. The latter can happen if your database does not allow parameterizing certain clauses. For example:

```
collation = "es_ES"
fragment("? COLLATE ?", ^name, ^collation)

limit = "10"
"posts" |> select([p], p.title) |> limit(fragment("?", ^limit))
```

The first example above won't work because `collation` needs to be quoted as an identifier. The second example won't work on databases that do not allow passing query parameters as part of `limit`.

You can address this by telling Ecto to treat these values differently than a query parameter:

```
fragment("? COLLATE ?", ^name, identifier(^collation))
"posts" |> select([p], p.title) |> limit(fragment("?", ^constant(limit))
```

Ecto will make these values directly part of the query, handling quoting and escaping where necessary.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

**Query caching**

Because identifiers are made part of the query, each different value will generate a separate query, with its own cache. Queries with interpolated constants are not cached.

## Splicing

Sometimes you may need to interpolate a variable number of arguments into the same fragment. For example, when overriding Ecto's default `where` behaviour for Postgres:

```
from p in Post, where:
fragment("? in (?, ?)", p.id,
val1, val2)
```

The example above will only work if you know the number of arguments upfront. If it can vary, the above will not work.

You can address this by telling Ecto to splice a list argument into the fragment:

Ecto
▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from p in Post, where:
fragment("? in (?)", p.id,
splice(^val_list))
```

This will let Ecto know it should expand the values of the list into separate fragment arguments. For example:

```
from p in Post, where:
fragment("? in (?)", p.id,
splice(^[1, 2, 3]))
```

would be expanded into

```
from p in Post, where:
fragment("? in (?,?,?)", p.id,
^1, ^2, ^3)
```

## Defining custom functions using macros and fragment

You can add a custom Ecto query function using macros. For example to expose SQL's coalesce function you can define this macro:

Ecto
▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
defmodule CustomFunctions do
  defmacro coalesce(left,
right) do
    quote do
      fragment("coalesce(?,
?)", unquote(left),
unquote(right))
    end
  end
end
```

To have coalesce/2 available, just import the module that defines it.

```
import CustomFunctions
```

The only downside is that it will show up as a fragment when inspecting the Elixir query. Other than that, it should be equivalent to a built-in Ecto query function.

## Keyword fragments

In order to support databases that do not have string-based queries, like MongoDB, fragments also allow keywords to be given:

```
from p in Post,
    where: fragment(title:
["$eq": ^some_value])
```

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

**from_now(count, interval)** </>

Adds the given interval to the current time in UTC.

The current time in UTC is retrieved from Elixir and not from the database.

See Intervals for supported `interval` values.

## Examples

```
from a in Account, where:
a.expires_at < from_now(3,
"month")
```

**identifier(binary)** </>

Allows a dynamic identifier to be injected into a fragment:

```
collation = "es_ES"
select("posts", [p],
fragment("? COLLATE ?",
p.title,
identifier(^collation)))
```

The example above will inject the value of `collation` directly into the query instead of treating it as a query parameter. It will generate a query such as `SELECT p0.title`

Ecto

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

`COLLATE "es_ES" FROM "posts" AS p0` as opposed to `SELECT p0.title COLLATE $1 FROM "posts" AS p0`.

Note that each different value of `collation` will emit a different query, which will be independently prepared and cached.

**ilike(string, search)** </>

Searches for `search` in `string` in a case insensitive fashion.

```
from p in Post, where:
ilike(p.body, "Chapter%")
```

Translates to the underlying SQL ILIKE query. This operation is only available on PostgreSQL.

**left in right** </>

Checks if the left-value is included in the right one.

```
from p in Post, where: p.id in
[1, 2, 3]
```

The right side may either be a literal list, an interpolated list, any struct that implements the `Enumerable`

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

protocol, or even a column in the database with array type:

```
from p in Post, where: "elixir"
in p.tags
```

Additionally, the right side may also be a subquery, which should return a single column:

```
from c in Comment, where:
c.post_id in subquery(
  from(p in Post, where:
p.created_at > ^since, select:
p.id)
)
```

**is_nil(value)** </>

Checks if the given value is nil.

```
from p in Post, where:
is_nil(p.published_at)
```

To check if a given value is not nil use:

```
from p in Post, where: not
is_nil(p.published_at)
```

**json_extract_path(json_field, path)**

Ecto

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
</>
```

Returns value from the `json_field` pointed to by `path`.

```
from(post in Post, select:
json_extract_path(post.meta,
["author", "name"]))
```

The path can be dynamic:

```
path = ["author", "name"]
from(post in Post, select:
json_extract_path(post.meta,
^path))
```

And the field can also be dynamic in combination with it:

```
path = ["author", "name"]
from(post in Post, select:
json_extract_path(field(post,
:meta), ^path))
```

The query can be also rewritten as:

```
from(post in Post, select:
post.meta["author"]["name"])
```

Path elements can be integers to access values in JSON arrays:

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from(post in Post, select:
post.meta["tags"][0]["name"])
```

Some adapters allow path elements to be references to query source fields

```
from(post in Post, select:
post.meta[p.title])
from(p in Post, join: u in
User, on: p.user_id == u.id,
select: p.meta[u.name])
```

Any element of the path can be dynamic:

```
field = "name"
from(post in Post, select:
post.meta["author"][^field])

source_field = :source_column
from(post in Post, select:
post.meta["author"][field(p,
^source_field)])
```

Interpolated path elements are inlined into the query, so each path emits a different query. For this reason, queries with interpolated paths are not cached.

## Warning: indexes on PostgreSQL

PostgreSQL supports indexing on jsonb columns via GIN indexes.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

Whenever comparing the value of a jsonb field against a string or integer, Ecto will use the containment operator @> which is optimized. You can even use the more efficient `jsonb_path_ops` GIN index variant. For more information, consult PostgreSQL's docs on JSON indexing.

### Warning: return types

The underlying data in the JSON column is returned without any additional decoding. This means "null" JSON values are not the same as SQL's "null". For example, the `Repo.all` operation below returns an empty list because `p.meta["author"]` returns JSON's null and therefore `is_nil` does not succeed:

```
Repo.insert!(%Post{meta: %
{author: nil}})
Repo.all(from(post in Post,
where:
is_nil(p.meta["author"])))
```

Similarly, other types, such as datetimes, are returned as strings. This means conditions like `post.meta["published_at"] > from_now(-1, "day")` may return incorrect results or fail as the underlying database tries to compare incompatible types. You can,

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

however, use `type/2` to force the types on the database level.

### `like(string, search)` </>

Searches for `search` in `string`.

```
from p in Post, where:
like(p.body, "Chapter%")
```

Translates to the underlying SQL LIKE query, therefore its behaviour is dependent on the database. In particular, PostgreSQL will do a case-sensitive operation, while the majority of other databases will be case-insensitive. For performing a case-insensitive `like` in PostgreSQL, see `ilike/2`.

You should be very careful when allowing user sent data to be used as part of LIKE query, since they allow to perform LIKE-injections.

### `map(source, fields)` </>

Used in `select` to specify which fields should be returned as a map.

For example, if you don't need all fields to be returned or neither need a struct, you can use `map/2` to achieve both:

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from p in Post,
  select: map(p, [:title,
:body])
```

`map/2` can also be used to dynamically select fields:

```
fields = [:title, :body]
from p in Post, select: map(p,
^fields)
```

If the same source is selected multiple times with a `map`, the fields are merged in order to avoid fetching multiple copies from the database. In other words, the expression below:

```
from(city in City, preload:
:country,
     select: {map(city,
[:country_id]), map(city,
[:name])})
```

is expanded to:

```
from(city in City, preload:
:country,
     select: {map(city,
[:country_id, :name]),
map(city, [:country_id,
:name])})
```

For preloads, the selected fields may be specified from the parent:

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from(city in City, preload:
:country,
     select: map(city,
[:country_id, :name, country:
[:id, :population]]))
```

It's also possible to select a struct from one source but only a subset of fields from one of its associations:

```
from(city in City, preload:
:country,
     select: %{city | country:
map(country: [:id,
:population])})
```

**IMPORTANT**: When filtering fields for associations, you MUST include the foreign keys used in the relationship, otherwise Ecto will be unable to find associated records.

### `max(value)` </>

Calculates the maximum for the given entry.

```
from p in Payment, select:
max(p.value)
```

### `merge(left_map, right_map)` </>

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

Merges the map on the right over the map on the left.

If the map on the left side is a struct, Ecto will check all of the field on the right previously exist on the left before merging.

```
from(city in City, select:
merge(city, %{virtual_field:
"some_value"}))
```

This function is primarily used by `Ecto.Query.select_merge/3` to merge different select clauses.

### `min(value)` </>

Calculates the minimum for the given entry.

```
from p in Payment, select:
min(p.value)
```

### `not value` </>

Unary `not` operation.

It is used to negate values in `:where`. It is also used to match the assert the opposite of `in/2`, `is_nil/1`, and `exists/1`. For example:

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from p in Post, where: p.id not
in [1, 2, 3]

from p in Post, where: not
is_nil(p.title)

# Retrieve all the posts that
doesn't have comments.
from p in Post,
  as: :post,
  where:
    not exists(
      from(
        c in Comment,
        where:
parent_as(:post).id == c.post_id
      )
    )
```

### `left or right` </>

Binary `or` operation.

### `parent_as(binding)` </>

Refer to a named atom binding in the parent query.

This is available only inside subqueries.

See Named Bindings for more information.

Ecto

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
selected_as(name)
```

Refer to an alias of a selected value.

This can be used to refer to aliases created using `selected_as/2`. If the alias hasn't been created using `selected_as/2`, an error will be raised.

Each database has its own rules governing which clauses can reference these aliases. If an error is raised mentioning an unknown column, most likely the alias is being referenced somewhere that is not allowed. Consult the documentation for the database to ensure the alias is being referenced correctly.

```
selected_as(selected_value, name)
```

Creates an alias for the given selected value.

When working with calculated values, an alias can be used to simplify the query. Otherwise, the entire expression would need to be copied when referencing it outside of select statements.

This comes in handy when, for instance, you would like to use the calculated value in

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

`Ecto.Query.group_by/3` or `Ecto.Query.order_by/3`:

```
from p in Post,
  select: %{
    posted:
selected_as(p.posted, :date),
    sum_visits: p.visits |>
coalesce(0) |> sum() |>
selected_as(:sum_visits)
  },
  group_by: selected_as(:date),
  order_by:
selected_as(:sum_visits)
```

The name of the alias must be an atom and it can only be used in the outer most select expression, otherwise an error is raised. Please note that the alias name does not have to match the key when `select` returns a map, struct or keyword list.

Using this in conjunction with `selected_as/1` is recommended to ensure only defined aliases are referenced.

## Subqueries and CTEs

Subqueries and CTEs automatically alias the selected fields, for example, one can write:

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
# Subquery
s = from p in Post, select: %
{visits: coalesce(p.visits, 0)}
from(s in subquery(s), select:
s.visits)

# CTE
cte_query = from p in Post,
select: %{visits:
coalesce(p.visits, 0)}
Post |> with_cte("cte", as:
^cte_query) |> join(:inner,
[p], c in "cte") |> select([p,
c], c.visits)
```

However, one can also use `selected_as` to override the default naming:

```
# Subquery
s = from p in Post, select: %
{visits: coalesce(p.visits, 0)
|> selected_as(:num_visits)}
from(s in subquery(s), select:
s.num_visits)

# CTE
cte_query = from p in Post,
select: %{visits:
coalesce(p.visits, 0) |>
selected_as(:num_visits)}
Post |> with_cte("cte", as:
^cte_query) |> join(:inner,
[p], c in "cte") |> select([p,
c], c.num_visits)
```

Ecto

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

The name given to `selected_as/2` can also be referenced in `selected_as/1`, as in regular queries.

### `splice(list)` </>

Allows a list argument to be spliced into a fragment.

Dynamic lists can be spliced into a query using interpolation

```
from p in Post, where:
fragment("? in (?)", p.id,
splice(^[1, 2, 3]))
```

Note that each element of the list will be treated as a separate query parameter. The example above will be transformed at runtime into the following

```
from p in Post, where:
fragment("? in (?,?,?)", p.id,
^1, ^2, ^3)
```

Since lists of different lengths emit different queries, queries with interpolated splices are not cached.

You may also splice compile-time lists. This allows you to combine query parameters with literals and constructs like query bindings

Ecto

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
sep = " "
from p in Post, select:
fragment("concat(?)",
splice([p.count, ^sep,
"count"]))
```

The above example will be transformed into

```
sep = " "
from p in Post, select:
fragment("concat(?,?,?)",
p.count, ^sep, "count")
```

This is especially useful if you would like to create re-usable macros to inject variadic database functions into queries. For example, you may create a macro for the Postgres function `concat_ws` like below

```
defmacro concat_ws(sep, args) do
  quote do
    fragment("concat_ws(?,?)",
unquote(sep),
splice(unquote(args)))
  end
end
```

Then you may call it from your application with argument lists of any size

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from p in Post, select:
concat_ws(":", [p.author,
^year, p.title])
from s in Sequences, select:
concat_ws(".", ["public",
s.relname])
```

Or you may want to create re-usable macros for [Postgres's row constructor comparison syntax] to support multi-column cursor-based pagination:

```
defmacro row_gt(columns,
values) do
  quote do
    fragment("(?) >= (?)",
splice(unquote(columns)),
splice(unquote(values)))
  end
end

from(p in Post, where:
row_gt([p.visits, p.id],
[^lower_visits, ^lower_id]))
```

You may nest others splices and fragment modifiers such as `identifier/1` and `constant/1` inside of compile-time splices

```
from p in Post, where:
fragment("? in (?)", p.id,
splice([constant(^1),
splice(^[2, 3])]))
```

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

This would be transformed into

```
from p in Post, where:
fragment(? in (?,?,?), p.id,
constant(^1), ^2, ^3)
```

**`struct(source, fields)`** </>

Used in `select` to specify which struct fields should be returned.

For example, if you don't need all fields to be returned as part of a struct, you can filter it to include only certain fields by using `struct/2`:

```
from p in Post,
  select: struct(p, [:title,
:body])
```

`struct/2` can also be used to dynamically select fields:

```
fields = [:title, :body]
from p in Post, select:
struct(p, ^fields)
```

As a convenience, `select` allows developers to take fields without an explicit call to `struct/2`:

```
from p in Post, select:
[:title, :body]
```

Ecto

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

Or even dynamically:

```
fields = [:title, :body]
from p in Post, select: ^fields
```

For preloads, the selected fields may be specified from the parent:

```
from(city in City, preload:
:country,
     select: struct(city,
[:country_id, :name, country:
[:id, :population]]))
```

If the same source is selected multiple times with a `struct`, the fields are merged in order to avoid fetching multiple copies from the database. In other words, the expression below:

```
from(city in City, preload:
:country,
     select: {struct(city,
[:country_id]), struct(city,
[:name])})
```

is expanded to:

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
from(city in City, preload:
:country,
     select: {struct(city,
[:country_id, :name]),
struct(city, [:country_id,
:name])})
```

**IMPORTANT**: When filtering fields for associations, you MUST include the foreign keys used in the relationship, otherwise Ecto will be unable to find associated records.

### `sum(value)` </>

Calculates the sum for the given entry.

```
from p in Payment, select:
sum(p.value)
```

### `type(interpolated_value, type)` </>

Casts the given value to the given type at the database level.

Most of the times, Ecto is able to proper cast interpolated values due to its type checking mechanism. In some situations though, you may want to tell Ecto that a parameter has some particular type:

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
type(^title, :string)
```

It is also possible to say the type must match the same of a column:

```
type(^title, p.title)
```

Or a parameterized type, which must be previously initialized with `Ecto.ParameterizedType.init/2`:

```
@my_enum
Ecto.ParameterizedType.init(Ecto.E
values: [:foo, :bar, :baz])
type(^title, ^@my_enum)
```

Ecto will ensure `^title` is cast to the given type and enforce such type at the database level. If the value is returned in a `select`, Ecto will also enforce the proper type throughout.

When performing arithmetic operations, `type/2` can be used to cast all the parameters in the operation to the same type:

```
from p in Post,
  select: type(p.visits +
^a_float + ^a_integer, :decimal)
```

Inside `select`, `type/2` can also be used to cast fragments:

Ecto
▾

QUERY APIS
Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
type(fragment("NOW"),
:naive_datetime)
```

Or to type fields from schemaless queries:

```
from p in "posts", select:
type(p.cost, :decimal)
```

Or to type aggregation results:

```
from p in Post, select:
type(avg(p.cost), :integer)
from p in Post, select:
type(filter(avg(p.cost), p.cost
> 0), :integer)
```

Or to type comparison expression results:

```
from p in Post, select:
type(coalesce(p.cost, 0),
:integer)
```

Or to type fields from a parent query using `parent_as/1`:

Ecto

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
child = from c in Comment,
where:
type(parent_as(:posts).id,
:string) == c.text
from Post, as: :posts,
inner_lateral_join: c in
subquery(child), select: c.text
```

### `type` vs `fragment`

`type/2` is all about Ecto types. Therefore, you can perform `type(expr, :string)` but not `type(expr, :text)`, because `:text` is not an actual Ecto type. If you want to perform casting exclusively at the database level, you can use fragment. For example, in PostgreSQL, you might do `fragment("?::text", p.column)`.

**`values(values, types)`** </>

Creates a values list/constant table.

A values list can be used as a source in a query, both in `Ecto.Query.from/2` and `Ecto.Query.join/5`.

The first argument is a list of maps representing the values of the constant table. An error is raised if the list is empty or if every map does not have exactly the same fields.

Ecto ▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

The second argument is either a map of types or an Ecto schema containing all the fields in the first argument.

Each field must be given a type or an error is raised. Any type that can be specified in a schema may be used.

Queries using a values list are not cacheable by Ecto.

### Select with map types example

```
values = [%{id: 1, text:
"abc"}, %{id: 2, text: "xyz"}]
types = %{id: :integer, text:
:string}

query =
  from v1 in values(values,
types),
    join: v2 in values(values,
types),
    on: v1.id == v2.id

Repo.all(query)
```

### Select with schema types example

Ecto

QUERY APIS

Ecto.Query.API

Sections

Summary

Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
values = [%{id: 1, text:
"abc"}, %{id: 2, text: "xyz"}]
types = ValuesSchema

query =
  from v1 in values(values,
types),
    join: v2 in values(values,
types),
    on: v1.id == v2.id

Repo.all(query)
```

## Delete example

```
values = [%{id: 1, text:
"abc"}, %{id: 2, text: "xyz"}]
types = %{id: :integer, text:
:string}

query =
  from p in Post,
    join: v in values(values,
types),
    on: p.id == v.id,
    where: p.counter == ^0

Repo.delete_all(query)
```

## Update example

Ecto
▾

QUERY APIS

Ecto.Query.API

Sections
Summary
Functions

*/2
+/2
-/2
//2
!=/2
</2
<=/2
==/2
>/2
>=/2
ago/2
all/1
and/2
any/1
as/1
avg/1
coalesce/2
constant/1
count/0
count/1
count/2
date_add/3
datetime_add/3
exists/1
field/2
filter/2
fragment/1
from_now/2
identifier/1
ilike/2
in/2
is_nil/1
json_extract_path/2
like/2
map/2
max/1

```
values = [%{id: 1, text:
"abc"}, %{id: 2, text: "xyz"}]
types = %{id: :integer, text:
:string}

query =
  from p in Post,
    join: v in values(values,
types),
    on: p.id == v.id,
    update: [set: [text:
v.text]]

Repo.update_all(query, [])
```

Built using ExDoc (v0.40.3) for the Elixir programming language