

DATABASES

THE Database IS YOUR
FRIEND

PostgresQL

EMBRACE THE
DATABASE
WITH *Ecto*

WHO AM I?

- ▶ Josh Branchaud
- ▶ Software Developer at *Hashrocket*



AGENDA

- ▶ Data Integrity
- ▶ Schemaless Queries
- ▶ Ecto's Escape Hatch
- ▶ Enhancing Ecto with Custom Functions
- ▶ Bonus: Tricky Query

OUR SAMPLE DATA SOURCE

The screenshot shows a light blue header with the text "TODAY I LEARNED" in large, bold, dark blue capital letters. Below it, in smaller text, is "A HASHROCKET PROJECT" next to a red rocket icon, and "FOLLOW ON TWITTER" next to a blue Twitter icon.

The main content area has a white background. At the top, the title "Compute md5 Hash Of A String" is displayed in a dark font. Below the title, a paragraph of text reads: "To compute the md5 digest of a string, we can use Erlang's top-level `md5` function." Following this is a code block with a dark grey background and a black border, labeled "ELIXIR" in the top right corner. The code contains the following text:

```
> :erlang.md5("#myelixirstatus")
<<145, 148, 139, 99, 194, 176, 105, 18, 242, 246, 37, 69, 142, 69, 226, 199>>
```

Below the code block, a paragraph explains: "This, however, gives us the result in the raw binary representation. We would like it in a base 16 encoding, as md5 digests tend to be." A final note states: "We can wrap (or pipe) this with `Base.encode16` to get the result we are looking for." This is followed by another code block with a dark grey background and a black border, labeled "ELIXIR" in the top right corner. The code contains:

```
> Base.encode16(:erlang.md5("#myelixirstatus"), case: :lower)
"91948b63c2b06912f2f625458e45e2c7"
```

WHAT IS TIL?

TIL is an open-source project by the team at Hashrocket that catalogues the sharing & accumulation of knowledge as it happens day-to-day.

TIL'S DATABASE SCHEMA

- ▶ Posts
- ▶ Developers
- ▶ Channels

POSTS TABLE

id	integer
title	character varying
body	text
likes	integer
developer_id	integer
channel_id	integer

Foreign-key constraints:

```
"fk_rails_447dc2e0a3" FOREIGN KEY (channel_id) REFERENCES channels(id)
"fk_rails_b3ec63b3ac" FOREIGN KEY (developer_id) REFERENCES developers(id)
```

DEVELOPERS TABLE

id	integer
email	character varying
username	character varying

Referenced by:

```
TABLE "posts" CONSTRAINT "fk_rails_b3ec63b3ac"  
FOREIGN KEY (developer_id) REFERENCES developers(id)
```

CHANNELS TABLE

id	integer
name	text

Referenced by:

```
TABLE "posts" CONSTRAINT "fk_rails_447dc2e0a3"  
  FOREIGN KEY (channel_id) REFERENCES channels(id)
```

DATABASE FULL OF DATA

So many answers just waiting to be asked the right question

ASKING QUESTIONS

How do we ask questions of our data?

WE NEED A MEDIATOR

What is the best mediator between us and our data?

SQL

SQL is the best way to talk to our SQL database

HOW MANY POSTS ARE THERE?

```
sql> select count(id) from posts;
```

```
count
```

```
-----
```

```
1066
```

```
(1 row)
```

WHAT ABOUT

Elixir AND Ecto?

ECTO

Ecto is a domain specific language for writing queries and interacting with databases in Elixir.

HOW MANY POSTS ARE THERE?

```
iex> from(p in "posts",  
         select: count(p.id))  
|> Repo.one()
```

1066

```
17:16:36.573 [debug] QUERY OK source="posts" db=10.8ms queue=0.2ms  
SELECT count(p0."id") FROM "posts" AS p0 []
```

**QUERIES ARE JUST
DATA**

QUERIES AS DATA

- ▶ **#Ecto.Query Struct**
- ▶ **You build them up as you go**
- ▶ **You can inspect them**

DATA INTEGRITY

SCHEMALESS QUERIES

ECTO'S ESCAPE HATCH

ENHANCING ECTO WITH CUSTOM FUNCTIONS

LAYING A SOLID FOUNDATION FOR OUR DATA

OUR DATA IS ONLY ANY GOOD
if it is correct

WE LIKE PUTTING DATA IN OUR
database

**WHO MAKES SURE WE DON'T
PUT BAD DATA IN OUR
DATABASE?**

VALIDATIONS, RIGHT?

App → validations → DB

Mobile → validations → DB

Services → validations → DB

YOUR DATABASE IS THE ULTIMATE

gatekeeper

DATABASE AS THE *Gatekeeper*

"A database actively seeks to maintain the correctness of all its data."
– Joe Celko

DATA INTEGRITY - DATA TYPES

Data types as constraints

```
create table(:developers) do
  add :email, :varchar
  add :admin, :boolean
  add :created_at, :timestamp
end
```

DATA INTEGRITY - DATA TYPES

Using better, custom data types (e.g. `uuid`, `bigint`, and `citext`)

```
execute("create extension if not exists citext;")
```

```
create table(:developers, primary_key: false) do
  add :id, :uuid, primary_key: true
  add :email, :citext
  add :admin, :boolean
  add :created_at, :timestamp
end
```

DATA INTEGRITY - NOT NULL

Enforce Presence with Not Null Constraints

```
create table(:developers, primary_key: false) do
  add :id, :uuid, primary_key: true
  add :email, :citext, null: false
  add :admin, :boolean, null: false
  add :created_at, :timestamp
end
```

DATA INTEGRITY - NOT NULL

The most important column constraint is the NOT NULL. Use this constraint automatically and then remove it only when you have good reason. This will help you avoid the complications of NULL values when you make queries against the data.

– Joe Celko

DATA INTEGRITY - FOREIGN KEYS

Enforce Relationships with Foreign Key Constraints

```
create table(:posts) do
  add :title, :varchar, null: false
  add :body, :text, null: false
  add :developer_id, references(:developers, type: :uuid)
end
```

DATA INTEGRITY - CHECK CONSTRAINTS

Enforce More General Relationships with Check Constraints

```
create table(:posts) do
  add :title, :varchar, null: false
  add :body, :text, null: false
  add :likes, :smallint, null: false, default: 0

  add :developer_id, references(:developers, type: :uuid)
end

create constraint(:posts, "ensure_positive_likes", check: "likes >= 0")
```

DATA INTEGRITY SCHEMALESS QUERIES ECTO'S ESCAPE HATCH ENHANCING ECTO WITH CUSTOM FUNCTIONS

SCHEMALESS QUERIES

```
iex> Repo.one(from p in "posts", select: count(p.id))  
1066
```

```
iex> Repo.one(from p in MyApp.Posts, select: count(p.id))  
1066
```

**SO, WHY
SCHEMALESS?**

QUERY TIME

HOW MANY DEVELOPERS ARE THERE?

HOW MANY DEVELOPERS ARE THERE?

```
iex> from(d in "developers",  
           select: count(d.id))  
|> Repo.one()
```

```
17:19:01.195 [debug] QUERY OK source="developers" db=1.0ms queue=2.9ms  
SELECT count(d0.id) FROM "developers" AS d0 []
```

32

FROM CLAUSE

The **FROM** specifies one or more source tables for the **SELECT**.

SELECT CLAUSE

SELECT retrieves rows from zero or more tables

HOW MANY POSTS BY CHANNEL?

HOW MANY POSTS BY CHANNEL?

JOIN CLAUSE

A JOIN clause combines two FROM items

HOW MANY POSTS BY CHANNEL?

```
iex> posts_and_channels =  
  from(p in "posts",  
    join: c in "channels",  
    on: p.channel_id == c.id)
```

HOW MANY POSTS BY CHANNEL?

GROUP BY CLAUSE

With group by, output is combined in groups of rows that match the grouping value

HOW MANY POSTS BY CHANNEL?

```
iex> from([p,c] in posts_and_channels,  
         group_by: c.name,  
         select: c.name)
```

HOW MANY POSTS BY CHANNEL?

```
iex> from [p,c] in posts_and_channels,  
        group_by: c.name,  
        select: c.name)  
|> Repo.all()
```

```
["clojure", "react", "rails", "vim", "workflow", "command-line", "sql",  
 "elixir", "erlang", "design", "testing", "go", "mobile", "javascript",  
 "devops", "ruby", "html-css", "git", "emberjs"]
```

HOW MANY POSTS BY CHANNEL?

AGGREGATES

Aggregate functions are computed across all rows making up each group, producing a separate value for each group.

HOW MANY POSTS BY CHANNEL?

```
iex> from([p,c] in posts_and_channels,  
         group_by: c.name,  
         select: {  
           count(p.id),  
           c.name  
         })
```

HOW MANY POSTS BY CHANNEL?

```
iex> from([p,c] in posts_and_channels,  
         group_by: c.name,  
         select: {  
           count(p.id),  
           c.name  
         })  
|> Repo.all()
```

```
[{13, "clojure"}, {5, "react"}, {102, "rails"}, {201, "vim"}, {59, "workflow"},  
 {110, "command-line"}, {121, "sql"}, {73, "elixir"}, {1, "erlang"},  
 {6, "design"}, {28, "testing"}, {5, "go"}, {15, "mobile"}, {67, "javascript"},  
 {32, "devops"}, {125, "ruby"}, {17, "html-css"}, {63, "git"}, {23, "emberjs"}]
```

HOW MANY POSTS BY CHANNEL?

ORDER BY CLAUSE

If the ORDER BY clause is specified, the returned rows are sorted in the specified order.

HOW MANY POSTS BY CHANNEL?

```
iex> from([p,c] in posts_and_channels,  
         group_by: c.name,  
         order_by: [desc: count(p.id)],  
         select: {  
           count(p.id),  
           c.name  
         })
```

HOW MANY POSTS BY CHANNEL?

```
iex> from([p,c] in posts_and_channels,  
         group_by: c.name,  
         order_by: [desc: count(p.id)],  
         select: {  
             count(p.id),  
             c.name  
         })  
|> Repo.all()
```

```
[{201, "vim"}, {125, "ruby"}, {121, "sql"}, {110, "command-line"},  
 {102, "rails"}, {73, "elixir"}, {67, "javascript"}, {63, "git"},  
 {59, "workflow"}, {32, "devops"}, {28, "testing"}, {23, "emberjs"},  
 {17, "html-css"}, {15, "mobile"}, {13, "clojure"}, {6, "design"}, {5, "go"},  
 {5, "react"}, {1, "erlang"}]
```

HOW MANY POSTS ON AVERAGE PER DEVELOPER?

HOW MANY POSTS ON AVERAGE PER DEVELOPER?

```
iex> post_counts =  
  from(p in "posts",  
       group_by: p.developer_id,  
       select: %{  
         post_count: count(p.id),  
         developer_id: p.developer_id  
       })
```

HOW MANY POSTS ON AVERAGE PER DEVELOPER?

```
iex> Repo.all(post_counts)
```

```
[%{developer_id: 14, post_count: 6}, %{developer_id: 25, post_count: 43},  
%{developer_id: 32, post_count: 1}, %{developer_id: 27, post_count: 2},  
%{developer_id: 8, post_count: 332}, %{developer_id: 17, post_count: 1},  
%{developer_id: 15, post_count: 23}, %{developer_id: 1, post_count: 1},  
%{developer_id: 10, post_count: 18}, %{developer_id: 26, post_count: 78},  
%{developer_id: 11, post_count: 15}, %{developer_id: 4, post_count: 130},  
%{developer_id: 18, post_count: 14}, %{developer_id: 30, post_count: 10},  
%{developer_id: 16, post_count: 3}, %{developer_id: 33, post_count: 1},  
%{developer_id: 6, post_count: 3}, %{developer_id: 19, post_count: 9},  
%{developer_id: 29, post_count: 82}, %{developer_id: 2, post_count: 236},  
%{developer_id: 23, post_count: 10}, %{developer_id: 31, post_count: 5},  
%{developer_id: 20, post_count: 8}, %{developer_id: 5, post_count: 3},  
%{developer_id: 13, post_count: 3}, %{developer_id: 22, post_count: 12},  
%{developer_id: 9, post_count: 10}, %{developer_id: 24, post_count: 4},  
%{developer_id: 7, post_count: 3}]
```

HOW MANY POSTS ON AVERAGE PER DEVELOPER?

```
iex> Repo.aggregate(subquery(post_counts), :avg, :post_count)
```

```
#Decimal<36.7586206896551724>
```

SCHEMALESS QUERIES

Let's try something a bit more complex

What is the channel and title of each developer's most liked post in 2016?

COMPLEX QUERIES

**Writing complex queries is all about
building the solution from the ground up**

PIECE BY PIECE

**WHAT IS THE CHANNEL AND TITLE OF
EACH DEVELOPER'S MOST LIKED POST IN 2016?**

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

```
iex> posts_devs_channels =  
  from(p in "posts",  
    join: d in "developers",  
    on: d.id == p.developer_id,  
    join: c in "channels",  
    on: c.id == p.channel_id)
```

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

```
iex> top_of_2016 =  
  from([posts, devs, channels] in posts_devs_channels,  
  order_by: [desc: posts.likes],  
  select: %{  
    dev: devs.username,  
    channel: channels.name,  
    title: posts.title  
  })
```

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

```
iex> top_of_2016 |> Repo.all()
```

```
[%{channel: "javascript", dev: "Frank Rossitano", title: "Because JavaScript"},  
 %{channel: "vim", dev: "Kenneth Parcell",  
   title: "Highlight #markdown fenced code syntax in #Vim"},  
 %{channel: "command-line", dev: "Kenneth Parcell",  
   title: "Homebrew is eating up your harddrive"},  
 ...]
```

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

DISTINCT CLAUSE

If **SELECT DISTINCT** is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates).

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

```
iex> top_of_2016 =  
  from([posts, devs, channels] in posts_devs_channels,  
    distinct: devs.id,  
    order_by: [desc: posts.likes],  
    select: %{  
      dev: devs.username,  
      channel: channels.name,  
      title: posts.title  
    })
```

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

posts.created_at > ^Ecto.DateTime.cast!({{2016, 1, 1}, {0, 0, 0}}),

and

posts.created_at < ^Ecto.DateTime.cast!({{2017, 1, 1}, {0, 0, 0}}),

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

WHERE CLAUSE

If the WHERE clause is specified, all rows that do not satisfy the condition are eliminated from the output.

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

```
iex> top_of_2016 =  
  from([posts, devs, channels] in posts_devs_channels(),  
    distinct: devs.id,  
    order_by: [desc: posts.likes],  
    where: posts.created_at > ^Ecto.DateTime.cast!({{2016,1,1},{0,0,0}}),  
    where: posts.created_at < ^Ecto.DateTime.cast!({{2017,1,1},{0,0,0}}),  
    select: %{  
      dev: devs.username,  
      channel: channels.name,  
      title: posts.title  
    })
```

WHAT IS THE CHANNEL AND TITLE OF EACH DEVELOPER'S MOST LIKED POST IN 2016?

```
iex> top_of_2016 |> Repo.all()
```

```
[%{channel: "elixir", dev: "Liz Lemon",
  title: "Invoke Elixir Functions with Apply"},  
%{channel: "workflow", dev: "Jack Donaghy", title: "Ternary shortcut in PHP"},  
%{channel: "vim", dev: "Tracy Jordan",
  title: "Use colorcolumn to visualize maximum line length"},  
%{channel: "ruby", dev: "Pete Hornberger",
  title: "Ruby optional arguments can come before required"},  
%{channel: "ruby", dev: "Jenna Maroney",
  title: "Using pessimistic gem version to catch betas"},  
...]
```

SCHEMALESS QUERY FUNCTIONS IN ECTO 2.0

- ▶ `Ecto.Repo.update_all/3`
- ▶ `Ecto.Repo.insert_all/3`
- ▶ `Ecto.Repo.delete_all/3`

DATA INTEGRITY SCHEMALESS QUERIES ECTO'S ESCAPE HATCH ENHANCING ECTO WITH CUSTOM FUNCTIONS

ESCAPE HATCH

Ecto can't do it all, sometimes we need an *Escape Hatch*

ONE-OFF QUERIES

Using Ecto.Repo.query

```
iex> Repo.query("select * from generate_series(1,5);")
```

```
12:00:14.801 [debug] QUERY OK db=1.5ms
select * from generate_series(1,5); []
{:ok,
%Postgrex.Result{columns: ["generate_series"], command: :select,
connection_id: 59379, num_rows: 5, rows: [[1], [2], [3], [4], [5]]}}
```

FRAGMENTS

The Ecto.Query.API.fragment function

FRAGMENTS IN QUERIES

```
iex> from(d in "developers",  
         select: fragment("count(*)"))  
|> Repo.one()
```

```
17:19:01.195 [debug] QUERY OK source="developers" db=1.0ms queue=2.9ms  
SELECT count(*) FROM "developers" AS d0 []
```

FRAGMENTS IN QUERIES

```
iex> from(d in "developers",  
         select: fragment("count?", d.id)  
     |> Repo.one()
```

```
17:19:01.195 [debug] QUERY OK source="developers" db=1.0ms queue=2.9ms  
SELECT count(d0.id) FROM "developers" AS d0 []
```

FRAGMENTS IN QUERIES

```
iex> top_of_2016 =  
  from([posts, devs, channels] in posts_devs_channels(),  
    distinct: devs.id,  
    order_by: [desc: posts.likes],  
    where: posts.created_at > ^Ecto.DateTime.cast!({{2016,1,1},{0,0,0}}),  
    where: posts.created_at < ^Ecto.DateTime.cast!({{2017,1,1},{0,0,0}}),  
    select: %{  
      dev: devs.username,  
      channel: channels.name,  
      title: posts.title  
    })
```

FRAGMENTS IN QUERIES

BETWEEN PREDICATE

The **BETWEEN** predicate simplifies range tests

a between x and y

FRAGMENTS IN QUERIES

```
fragment "? between ? and ?",
  posts.created_at,
  ^Ecto.DateTime.cast!({{2016, 1, 1}, {0, 0, 0}}),
  ^Ecto.DateTime.cast!({{2017, 1, 1}, {0, 0, 0}})
)
```

FRAGMENTS IN QUERIES

```
iex> top_of_2016 =  
  from([posts, devs, channels] in posts_devs_channels(),  
    distinct: devs.id,  
    order_by: [desc: posts.likes],  
    where: fragment("? between ? and ?",  
      posts.created_at,  
      ^Ecto.DateTime.cast!({{2016, 1, 1}, {0, 0, 0}}),  
      ^Ecto.DateTime.cast!({{2017, 1, 1}, {0, 0, 0}})  
    ),  
    select: %{{  
      dev: devs.username,  
      channel: channels.name,  
      title: posts.title  
    }})
```

FRAGMENTS IN MIGRATIONS

```
create table(:developers, primary_key: false) do
  add :id, :uuid, primary_key: true, default: fragment("gen_random_uuid()")
  add :email, :citext, null: false
  add :created_at, :timestamptz, null: false, default: fragment("now()")
  add :updated_at, :timestamptz, null: false, default: fragment("now()")
end
```

ONE STEP FURTHER

Reusable fragments with custom functions

DATA INTEGRITY SCHEMALESS QUERIES ECTO'S ESCAPE HATCH ENHANCING ECTO WITH CUSTOM FUNCTIONS

CUSTOM FUNCTIONS

```
fragment("? between ? and ?",
posts.created_at,
^Ecto.DateTime.cast!({{2016, 1, 1}, {0, 0, 0}}),
^Ecto.DateTime.cast!({{2017, 1, 1}, {0, 0, 0}})
)
```

CUSTOM FUNCTIONS

```
defmodule CustomFunctions do
  defmacro between(value, left_bound, right_bound) do
    quote do
      fragment("? between ? and ?",
        unquote(value),
        unquote(left_bound),
        unquote(right_bound))
    end
  end
end
```

CUSTOM FUNCTIONS

```
iex> import CustomFunctions

iex> from([posts, devs, channels] in posts_devs_channels(),
  distinct: devs.id,
  order_by: [desc: posts.likes],
  where: between(posts.created_at,
    ^Ecto.DateTime.cast!({{2016, 1, 1}, {0, 0, 0}}),
    ^Ecto.DateTime.cast!({{2017, 1, 1}, {0, 0, 0}})
  ),
  select: %{
    dev: devs.username,
    channel: channels.name,
    title: posts.title
  })

```

ONE MORE QUESTION TO ASK

WHAT ARE THE HOTTEST POSTS?

WHAT ARE THE HOTTEST POSTS?

Hottest TILs

How Rails Responds to `*/`*	#rails • 6 likes
Grep For A Pattern On Another Branch	#git • 9 likes
Viewing A File On Another Branch	#git • 8 likes
Get The pid Of The Session	#vim • 4 likes
New PostgreSQL 9.6 slice syntax	#sql • 6 likes
Default netrw To Tree Liststyle	#vim • 8 likes
Images in markdown	#workflow • 5 likes
What Changed?	#git • 9 likes
Check The Installed Version Of Phoenix	#elixir • 6 likes
Installing the Golang tools with Vim	#go • 5 likes

MEASURING HOTNESS

with a HackerNews-esque Ranking Algorithm¹

```
hotness_score = (likes / (age_in_hours ^ gravity))
```

¹ <https://medium.com/hacking-and-gonzo/how-hacker-news-ranking-algorithm-works-1d9b0cf2c08d#.3sdij412h>

MEASURING HOTNESS

with a HackerNews-esque Ranking Algorithm¹

```
hotness_score = (likes / (age_in_hours ^ 0.8))
```

¹ <https://medium.com/hacking-and-gonzo/how-hacker-news-ranking-algorithm-works-1d9b0cf2c08d#.3sdij412h>

WHAT ARE THE HOTTEST POSTS?

```
age_in_hours = age_in_seconds / 3600
```

WHAT ARE THE HOTTEST POSTS?

```
age_in_hours = age_in_seconds / 3600
```

```
age_in_seconds = (current_timestamp - published_at)
```

WHAT ARE THE HOTTEST POSTS?

```
age_in_hours = age_in_seconds / 3600
```

```
age_in_seconds = extract(epoch from  
                  (current_timestamp - published_at))
```

WHAT ARE THE HOTTEST POSTS?

```
age_in_hours = extract(epoch from  
                  (current_timestamp - published_at)  
               ) / 3600
```

WHAT ARE THE HOTTEST POSTS?

```
age_in_hours = extract(epoch from  
                  (current_timestamp - published_at)  
                 ) / 3600
```

```
age_in_hours = greatest(age_in_hours, 0.1)
```

WHAT ARE THE HOTTEST POSTS?

```
defmacro hours_since(timestamp) do
  quote do
    fragment(
      "extract(epoch from (current_timestamp - ?)) / 3600",
      unquote(timestamp)
    )
  end
end
```

WHAT ARE THE HOTTEST POSTS?

```
iex> posts_with_age_in_hours =  
  from(p in "posts",  
       select: %{  
         id: p.id,  
         hours_age: hours_since(p.published_at)  
       })
```

WHAT ARE THE HOTTEST POSTS?

```
iex> posts_with_age_in_hours |> Repo.all()
```

```
[%{hours_age: 16176.589612136388, id: 12},  
%{hours_age: 8308.070006305556, id: 657},  
%{hours_age: 7713.880550556667, id: 708},  
%{hours_age: 6054.369684539444, id: 833},  
%{hours_age: 6768.798842247777, id: 772},  
%{hours_age: 8315.479890300556, id: 654},  
%{hours_age: 5698.932204395278, id: 870},  
...]
```

WHAT ARE THE HOTTEST POSTS?

```
iex> posts_with_age_in_hours =  
  from(p in "posts",  
       where: not is_nil(p.published_at),  
       select: %{  
         id: p.id,  
         hours_age: hours_since(p.published_at)  
       })
```

WHAT ARE THE HOTTEST POSTS?

```
defmacro greatest(value1, value2) do
  quote do
    fragment("greatest(?, ?)", unquote(value1), unquote(value2))
  end
end
```

WHAT ARE THE HOTTEST POSTS?

ecto_pg_extras

```
# mix.exs
defp deps do
  [..., {:ecto_pg_extras, "~> 0.1.1"}]
end
```

```
# hot_posts.ex
import EctoPgExtras
```

WHAT ARE THE HOTTEST POSTS?

```
iex> posts_with_age_in_hours =  
  from(p in "posts",  
       where: not is_nil(p.published_at),  
       select: %{  
         id: p.id,  
         hours_age: greatest(hours_since(p.published_at), 0.1)  
       })
```

WHAT ARE THE HOTTEST POSTS?

```
iex> posts_with_age_in_hours =  
  from(p in "posts",  
       where: not is_nil(p.published_at),  
       select: %{  
         id: p.id,  
         likes: p.likes,  
         hours_age: greatest(hours_since(p.published_at), 0.1)  
       })
```

WHAT ARE THE HOTTEST POSTS?

```
iex> hot_posts =  
  from(p in subquery(posts_with_age_in_hours),  
    select: %{{  
      id: p.id,  
      hotness_score: fragment("? / (? ^ ?)",  
                                p.likes,  
                                p.hours_age,  
                                0.8)  
    } )
```

WHAT ARE THE HOTTEST POSTS?

```
iex> hot_posts =  
  from(p in subquery(posts_with_age_in_hours),  
    order_by: [desc: 2],  
    select: %{{  
      id: p.id,  
      hotness_score: fragment("? / (? ^ ?)",  
                                p.likes,  
                                p.hours_age,  
                                0.8)  
    }})
```

WHAT ARE THE HOTTEST POSTS?

```
iex> hot_posts =  
  from(p in subquery(posts_with_age_in_hours),  
    order_by: [desc: 2],  
    select: %{  
      id: p.id,  
      hotness_score: fragment("? / (? ^ ?)",  
                                p.likes,  
                                p.hours_age,  
                                0.8)  
    },  
    limit: 5)
```

WHAT ARE THE HOTTEST POSTS?

```
iex> hot_posts |> Repo.all()
```

```
[%{hotness_score: 0.07338486607688295, id: 1134},  
%{hotness_score: 0.0641696195616784, id: 1128},  
%{hotness_score: 0.06255221703215852, id: 1131},  
%{hotness_score: 0.05892805984356843, id: 1127},  
%{hotness_score: 0.056850664015716326, id: 1125}]
```

WHAT ARE THE HOTTEST POSTS?

```
iex> hot_posts_with_titles =  
  from(p in subquery(posts_with_age_in_hours),  
       join: posts in "posts", on: posts.id == p.id,  
       order_by: [desc: 2],  
       select: %{  
         title: posts.title,  
         hotness_score: fragment("? / (? ^ ?)",  
                               p.likes,  
                               p.hours_age,  
                               0.8)  
       },  
       limit: 5)
```

WHAT ARE THE HOTTEST POSTS?

```
iex> hot_posts_with_titles |> Repo.all()
```

```
[%{hotness_score: 0.07335796393712307, title: "Custom loaders for webpack"},  
%{hotness_score: 0.06415573399947418, title: "Rerun Only Failures With RSpec"},  
%{hotness_score: 0.06253343464917116,  
  title: "Rails on ruby 2.4: Silence Fixnum/Bignum warnings"},  
%{hotness_score: 0.05891816565837998, title: "Polymorphic Path Helpers"},  
%{hotness_score: 0.056841537315585514, title: "Clean untracked files in Git"}]
```

THAT'S IT

NOW THE *database* IS YOUR

BEYOND ECTO

moebius - <https://github.com/robconery/moebius>

defql - <https://github.com/fazibear/defql>

SOURCES AND LINKS

- ▶ github.com/hashrocket/ecto_pg_extras
 - ▶ til.hashrocket.com
- ▶ **Joe Celko's SQL for Smarties: Advanced SQL Programming, 5th Ed**
 - ▶ [PostgreSQL 9.6 Documentation](https://www.postgresql.org/docs/9.6/)

THANKS!

- ▶ Josh Branchaud
- ▶ Software Developer at Hashrocket
- ▶ Twitter: @jbrancha
- ▶ Github: @jbranchaud

