Postgres

The Best Tool You're Already Using

- Adam Sanderson
- LiquidPlanner

Adam Sanderson

I have been a full stack engineer at LiquidPlanner for 5 years.

- I got off in Kansas*, and that's ok!
- Github: adamsanderson
- Twitter: adamsanderson
- Blog: http://monkeyandcrow.com

* Seattle

LiquidPlanner®

Online project management with probabilistic scheduling.

- Started in 2007 with Rails 1.x
- Used Postgres from the beginning
- We have learned some great techniques along the way

Topics

- Tagging
- Hierarchy
- Custom Data
- Full Text Search

Method

For each topic, we'll cover the SQL before we cover its use in ActiveRecord.

We will use Postgres 9.x, Ruby 1.9 syntax, and ActiveRecord 4.0.

If you understand the SQL you can use it in any version of ActiveRecord, 4.0 just makes it easier.

Backstory

You just built a great new social network for hedgehog lovers around the world, HedgeWith.me.

Everything is going well. You have a few users, but now they want more.

My hedgehog is afraid of grumpy hedgehogs, but likes cute ones how can I find him friends?

hedgehogs4life

Tagging

People want to be able to tag their hedgehogs, and then find other hedgehogs with certain tags.

Defining Arrays in SQL

```sql
CREATE TABLE hedgehogs (
    id      integer primary key,
    name    text,
    age     integer,
    tags    text[]
);
```

Defining Arrays in ActiveRecord

```
create_table :hedgehogs do |t|
  t.string  :name
  t.integer :age
  t.text    :tags, array: true
end
```

ActiveRecord 4.x introduced arrays for Postgres, use `array:true`

Heads Up

Define array columns as `t.text` instead of `t.string` to avoid casting.

Postgres assumes that `ARRAY['cute', 'cuddly']` is of type `text[]` and will require you to cast, otherwise you will see errors like this:

```
ERROR: operator does not exist: character varying[] && text[]
```

Boolean Set Operators

You can use the set operators to query arrays.

- `A @> B` A contains all of B
- `A && B` A overlaps any of B

Querying Tags in SQL

Find all the hedgehogs that are spiny or prickly:

```
SELECT name, tags FROM hedgehogs
WHERE tags && ARRAY['spiny', 'prickly'];
```

`A && B` A overlaps any of B

Querying Tags in SQL

name	tags
Marty	spiny, prickly, cute
Quilby	cuddly, prickly, hungry
Thomas	grumpy, prickly, sleepy, spiny
Franklin	spiny, round, tiny

Querying Tags in SQL

Find all the hedgehogs that are spiny and prickly:

```
SELECT name, tags FROM hedgehogs
WHERE tags @> ARRAY['spiny', 'prickly'];
```

`A @> B` A contains all the B

Querying Tags in SQL

name	tags
Marty	spiny, prickly, cute
Thomas	grumpy, prickly, sleepy, spiny

Querying Tags in ActiveRecord

Find all the hedgehogs that are spiny and prickly

```
Hedgehog.where "tags @> ARRAY[?]", ['spiny', 'prickly']
```

Querying Tags in ActiveRecord

Create scopes to encapsulate set operations:

```
class Hedgehog < ActiveRecord::Base
  scope :any_tags, -> (* tags){where('tags && ARRAY[?]', tags)}
  scope :all_tags, -> (* tags){where('tags @> ARRAY[?]', tags)}
end
```

Querying Tags in ActiveRecord

Find all the hedgehogs that are spiny or large, and older than 4:

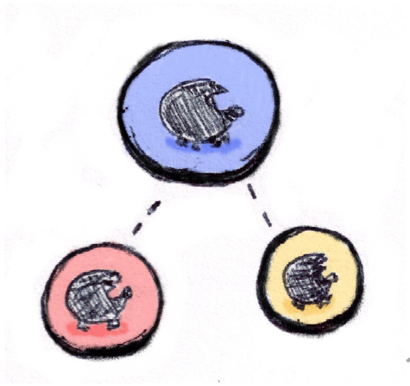
```
Hedgehog.any_tags('spiny', 'large').where('age > ?', 4)
```

“ *Hi, I run an influential hedgehog club. Our members would all use HedgeWith.me, if they could show which hogs are members of our selective society.*

Boston Spine Fancy President

Hierarchy

Apparently there are thousands of hedgehog leagues, divisions, societies, clubs, and so forth.



Hierarchy

We need to efficiently model a club hierarchy like this:

- North American League
    - Western Division
        - Cascadia Hog Friends
        - Californian Hedge Society

How can we support operations like finding a club's depth, children, or parents?

Materialized Path in SQL

Encode the parent ids of each record in its `path`.

```
CREATE TABLE clubs (
    id              integer primary key,
    name            text,
    path            integer[]
);
```

Querying a Materialized Path

id	name	path
1	North American League	[1]
2	Eastern Division	[1,2]
4	New York Quillers	[1,2,4]
5	Boston Spine Fancy	[1,2,5]
3	Western Division	[1,3]
6	Cascadia Hog Friends	[1,3,6]
7	California Hedge Society	[1,3,7]
…		

Materialized Path: Depth

The depth of each club is simply the length of its path.

- `array_length(array, dim)` returns the length of the array

`dim` will always be 1 unless you are using multidimensional arrays.

Materialized Path: Depth

Display the top two tiers of hedgehog clubs:

```
SELECT name, path, array_length(path, 1) AS depth
FROM clubs
WHERE array_length(path, 1) <= 2
ORDER BY path;
```

`array_length(path, 1)` is the depth of record

Materialized Path: Depth

name	**path**	**depth**
North American League	[1]	1
Eastern Division	[1,2]	2
Western Division	[1,3]	2
South American League	[9]	1

Materialized Path: Children

Find all the clubs that are children of the California Hedge Society, ID: `7`.

```
SELECT id, name, path FROM clubs
WHERE path && ARRAY[7]
ORDER BY path
```

`A && B` A overlaps any of B

Materialized Path: Children

id	name	path
7	Californian Hedge Society	[1,3,7]
8	Real Hogs of the OC	[1,3,7,8]
12	Hipster Hogs	[1,3,7,12]

Apparently it is illegal to own hedgehogs in California

Materialized Path: Parents

Find the parents of the California Hedge Society, Path: `ARRAY[1,3,7]`.

```
SELECT name, path FROM clubs
WHERE ARRAY[id] && ARRAY[1,3,7]
ORDER BY path;
```

`A && B` A overlaps any of B

Materialized Path: Parents

id	name	path
1	North American League	[1]
3	Western Division	[1,3]
7	Californian Hedge Society	[1,3,7]

ActiveRecord: Arrays & Depth

With ActiveRecord 4.x, `path` is just ruby array.

```
class Club < ActiveRecord::Base
  def depth
    self.path.length
  end
  ...
```

Querying in ActiveRecord

Encapsulate these conditions as instance methods:

```ruby
class Club < ActiveRecord::Base
  def children
    Club.where('path && ARRAY[?]', self.id)
  end
  def parents
    Club.where('ARRAY[id] && ARRAY[?]', self.path)
  end
```

Querying in ActiveRecord

Now we have an easy way to query the hierarchy.

```
@club.parents.limit(5)
@club.children.joins(:hedgehogs).merge(Hedgehog.any_tags('silly'))
```

These features can all work together.

Mind blown?

“

I need to keep track of my hedgehogs' favorite foods, colors, weight, eye color, and shoe sizes!

the Quantified Hedgehog Owner

If I am forced to enter my hedgehog's shoe size, I will quit immediately!

the Unquantified Hedgehog Owner

Custom Data

Your users want to record arbitrary data about their hedgehogs.

Hstore

Hstore provides a hash column type. It is a useful alternative to ActiveRecord's `serialize` where the keys and values can be queried in Postgres.

Hstore

Hstore needs to be installed manually. Your migration will look like this:

```
class InstallHstore < ActiveRecord::Migration
  def up
    execute 'CREATE EXTENSION hstore'
  end
  ...
```

Heads Up

Although hstore is supported by ActiveRecord 4.x, the default schema format does not support extensions.

Update `config/application.rb` to use the SQL schema format, otherwise your tests will fail.

```ruby
class Application < Rails::Application
  config.active_record.schema_format = :sql
end
```

Defining an Hstore in SQL

```
CREATE TABLE hedgehogs (
    id      integer primary key,
    name    text,
    age     integer,
    tags    text[],
    custom  hstore DEFAULT '' NOT NULL
);
```

Defining an Hstore in ActiveRecord

`hstore` is supported in ActiveRecord 4.x as a normal column type:

```
create_table :hedgehogs do |t|
  t.string  :name
  t.integer :age
  t.text    :tags, array: true
  t.hstore  :custom, :default => '', :null => false
end
```

Heads Up

Save yourself some hassle, and specify an empty hstore by default:

```
t.hstore  :custom, :default => '', :null => false
```

Otherwise new records will have null hstores.

Hstore Format

Hstore uses a text format, it looks a lot like a ruby 1.8 hash:

```
UPDATE hedgehogs SET
custom = '"favorite_food" => "lemons", "weight" => "2lbs"'
WHERE id = 1;
```

Be careful of quoting.

Hstore Operators

Common functions and operators:

- `defined(A, B)` Does A have B?
- `A -> B` Get B from A. In ruby this would be A[B]

Query Hstore in SQL

Find all the favorite foods of the hedgehogs:

```
SELECT name, custom -> 'favorite_food' AS food
FROM hedgehogs WHERE defined(custom, 'favorite_food');
```

`defined(A, B)`	Does A have B?
`A -> B`	Get B from A. In ruby this would be A[B]

Query Hstore in SQL

name	food
Horrace	lemons
Quilby	pasta
Thomas	grubs

Query Hstore in ActiveRecord

Create scopes to make querying easier:

```
class Hedgehog < ActiveRecord::Base
  scope :has_key,   -> (key){ where('defined(custom, ?)', key) }
  scope :has_value, -> (key, value){ where('custom -> ? = ?', key, value) }
  ...
```

Query Hstore in ActiveRecord

Find hedgehogs with a custom `color`:

```
Hedgehog.has_key('color')
```

Query Hstore in ActiveRecord

Find hedgehogs that are brown:

```
Hedgehog.has_value('color', 'brown')
```

Query Hstore in ActiveRecord

Find all the silly, brown, hedgehogs:

```
Hedgehog.any_tags('silly').has_value('color', 'brown')
```

Updating an Hstore with ActiveRecord

With ActiveRecord 4.x, hstore columns are just hashes:

```
hedgehog.custom["favorite_color"] = "ochre"

hedgehog.custom = {favorite_food: "Peanuts", shoe_size: 3}
```

Heads Up

Hstore columns are always stored as strings:

```
hedgehog.custom["weight"] = 3
hedgehog.save!
hedgehog.reload
hedgehog.custom['weight'].class #=> String
```

Someone commented on my hedgehog. They said they enjoy his beady little eyes, but I can't find it.

hogmama73

Full Text Search

Your users want to be able to search within their comments.

Full Text Search in SQL

```
CREATE TABLE comments (
    id              integer primary key,
    hedgehog_id     integer,
    body            text
);
```

Full Text Search Data Types

There are two important data types:

- `tsvector` represents the text to be searched
- `tsquery` represents the search query

Full Text Search Functions

There are two main functions that convert strings into these types:

- `to_tsvector(configuration, text)` creates a normalized `tsvector`
- `to_tsquery(configuration, text)` creates a normalized `tsquery`

Full Text Search Normalization

Postgres removes common stop words:

```
select to_tsvector('A boy and his hedgehog went to Portland');
-- boy, hedgehog, portland, went

select to_tsvector('I need a second line to fill space here.');
-- fill, line, need, second, space
```

Full Text Search Normalization

Stemming removes common endings from words:

term	**stemmed**
hedgehogs	hedgehog
enjoying	enjoy
piping	pipe

Full Text Search Operators

Vectors:

- `V @@ Q` Searches V for Q

Queries:

- `V @@ (A && B)` Searches V for A and B
- `V @@ (A || B)` Searches V for A or B

Full Text Search Querying

Find comments about "enjoying" something:

```
SELECT body
FROM comments
WHERE to_tsvector('english', body)
  @@  to_tsquery('english','enjoying');
```

`V @@ Q` Searches V for Q

Full Text Search Querying

- Does he enjoy beets? Mine loves them
- I really enjoy oranges
- I am enjoying these photos of your hedgehog's beady little eyes
- Can I feed him grapes? I think he enjoys them.

Notice how "enjoying" also matched "enjoy" and "enjoys" due to stemming.

Full Text Search Wildcards

- `to_tsquery('english','cat:*')` Searches for anything starting with cat

Such as: cat, catapult, cataclysmic.

But not: octocat, scatter, prognosticate

Full Text Search Wild Cards

Find comments containing the term "oil", and a word starting with "quil" :

```
SELECT body
FROM comments
WHERE to_tsvector('english', body)
  @@ ( to_tsquery('english','oil')
    && to_tsquery('english','quil:*')
  );
```

`V @@ (A && B)` Searches V for A and B

Full Text Search Querying

- What brand of oil do you use? Have you tried QuillSwill?

Heads Up

`tsquery` only supports wildcards at the end of a term.

While `quill:*` will match "QuillSwill", but `*:swill` will not.

In fact, `*:swill` will throw an error.

Even More Heads Up!

Never pass user input directly to `to_tsquery`, it has a strict mini search syntax. The following all fail:

- `http://localhost` `:` has a special meaning
- `O'Reilly's Books` Paired quotes cannot be in the middle
- `A && B` `&` and `|` are used for combining terms

You need to sanitize queries, or use a gem that does this for you.

Full Text Search With ActiveRecord

We can wrap this up in a scope.

```
class Comment < ActiveRecord::Base
  scope :search_all, -> (query){
    where("to_tsvector('english', body) @@ #{sanitize_query(query)}")
  }
```

You need to write `sanitize_query`, or use a gem that does this for you.

Full Text Search With ActiveRecord

Find the comments about quill oil again, and limit it to 5 results:

```
Comment.search_all("quil* oil").limit(5)
```

Since `search_all` is a scope, we chain it like all the other examples.

Full Text Search Indexing

Create an index on the function call `to_tsvector('english', body)`:

```
CREATE INDEX comments_gin_index
ON comments
USING gin(to_tsvector('english', body));
```

The `gin` index is a special index for multivalued columns like a `text[]` or a `tsvector`

Heads Up

Since we are indexing a function call, `to_tsvector('english', body)`, we must call it the same way every time.

You don't have to use `english`, but you do need to be consistent.

In Summary

- Arrays can model tagging and hierarchies
- Hstore can be used to model custom data
- Postgres supports full text search

You can now enjoy the happy hour!

```
SELECT * FROM beers WHERE

traits @> ARRAY['hoppy', 'floral']
```

Any Questions?

Possible suggestions:

- Why not normalize your database instead of using arrays?
- Can I see how you implemented `sanitize_query`?
- What is a good gem for full text search?
- What about ActiveRecord 2 and 3?
- Why hstore instead of JSON?
- Can I buy you coffee?

Extra Resources

- ActiveRecord Queries & Scopes
- Postgres Array Operators
- Postgres Hstore Documentation
- Postgres Full Text Search
- Ruby Gems for Full Text Search
    - Textacular Supports Active Record 2.x and 3.x
    - pg_search Supports Active Record 3.x, but has more features
- My Blog, Github, and favorite social network
- How to draw a hedgehog.

Bonus

Here's `sanitize_query`:

```
def self.sanitize_query(query, conjunction=' && ')
  "(" + tokenize_query(query).map{|t| term(t)}.join(conjunction) + ")"
end
```

It breaks up the user's request into terms, and then joins them together.

Bonus

We tokenize by splitting on white space, `&`, `|`, and `:`.

```
def self.tokenize_query(query)
  query.split(/(\s|[&|:])+/)
end
```

Bonus

Each of those tokens gets rewritten:

```
def self.term(t)
  # Strip leading apostrophes, they are never legal, "'ok" becomes "ok"
  t = t.gsub(/^'+/,'')
  # Strip any *s that are not at the end of the term
  t = t.gsub(/\*[^$]/,'')
  # Rewrite "sear*" as "sear:*" to support wildcard matching on terms
  t = t.gsub(/\*$/,':*')
  ...
```

```ruby
  ...
  # If the only remaining text is a wildcard, return an empty string
  t = "" if t.match(/^[:* ]+$/)

  "to_tsquery('english', #{quote_value t})"
end
```