PeepCode press

$9

Ruby on Rails

CODE REVIEW

by Geoffrey Grosenbach

CONTENTS

4 Store Sessions in the Database

7 Use Custom Configuration Files

11 Use Constants for Repeated Strings

14 Keep Time in UTC

16 Don't Loop Around ActiveRecord

22 Beware of Binary Fields

27 Cache Your Reports

32 Store Country Lists

36 Avoid Bloated Controllers

41 Keep Your Controllers and Views Skinny

47 Don't Store Objects in the Session

51 Avoid Heavy Response Processing

56 Use ActiveRecord Mailing

63 Monitor Your Servers

72 Don't Cut Costs on Hardware

76 Test-Drive

83 The Rest of the List

Don't Loop Around ActiveRecord

CHAPTER 5

THE WRONG WAY

ActiveRecord makes database access quite simple. With a few lines of Ruby, you can jump across tables and drill down into associated objects. It's so easy that you may be tempted to do something like this:

```
# The wrong way
@league = League.find_by_name "Northwest"

@all_stars = []
@league.teams.each do |team|
  team.players.each do |player|
    @all_stars << player if player.all_star?
  end
end

@all_stars
```

Unfortunately, that will quickly become a very expensive operation. To see why, look at the Rails logfile after this query:

```
League Load (0.000331)   SELECT * FROM leagues WHERE
(leagues."name" = 'Northwest') LIMIT 1
Team Load (0.000573)   SELECT * FROM teams WHERE (teams.
league_id = 1)
Player Load (0.000693)   SELECT * FROM players WHERE
(players.team_id = 1)
Player Load (0.000658)   SELECT * FROM players WHERE
```

```
(players.team_id = 2)
Player Load (0.000623)   SELECT * FROM players WHERE
(players.team_id = 3)
```

What happened? We started with a single `league` object, then issued a separate SQL command to retrieve the teams for that league. Additional SQL commands were required to retrieve the players in each of those teams.

In this case there were only a few teams, but 5 queries were generated. A league with more teams, more players, or further nesting could easily result in dozens of SQL queries and a very slow response.

THE RIGHT WAY

There are two easy ways to improve this code. First, you could add a single @include@ directive to load all the teams and players with a single SQL command.

```
# A better way, with eager loading
@league = League.find_by_name("Northwest", {
  :include => {:teams => :players}
})

@all_stars = []
@league.teams.each do |team|
  team.players.each do |player|
    @all_stars << player if player.all_star?
  end
end

@all_stars
```

:include Options

The `include` argument is very powerful but can also be confusing. Here are a few issues to be aware of.

PLURALITY

The plurality of the `include` needs to match the plurality of the association you're joining on.

For example, here is a `Farm` that has many chickens (and also cows):

```
class Farm < ActiveRecord::Base
  has_many :chickens
  has_many :cows
end
```

If you wanted to get all the farms and all their chickens, you would use the plural `chickens` because that is how it is used in the `has_many` statement in the model:

```
Farm.find(:all, :include => :chickens)
```

In reverse, you would use the singular. `Chicken` belongs to the singular `farm`:

```
class Chicken < ActiveRecord::Base
  belongs_to :farm
  has_many :eggs
end
```

Use the singular `farm` to include a join from the `Chicken` model back to the `Farm` model.

```
Chicken.find(:all, :include => :farm)
```

HASHES AND ARRAYS

The include starts with a hash, but it can go either way depending on how many associations are being included. We already saw that a `Farm` has both `chickens` and `cows`. To include both, use an `Array` as the value for the `include` hash.

```
Farm.find(:all, {
  :include => [:chickens, :cows]
})
```

ALL THE WAY DOWN

You can go even further (within reason). We could retrieve all the farms, all their cows and chickens, and each of the chickens' eggs.

```
Farm.find(:all, {
  :include => [
    :cows,
    {:chickens => :eggs}
  ]
})
```

Looking at the log now shows a single query:

```
League Load Including Associations (0.002438)   SELECT
leagues."id" AS t0_r0, leagues."name" AS t0_r1, teams."id"
AS t1_r0, teams."name" AS t1_r1, teams."league_id" AS
t1_r2, players."id" AS t2_r0, players."name" AS t2_r1,
players."team_id" AS t2_r2, players."all_star" AS t2_r3 FROM
leagues LEFT OUTER JOIN teams ON teams.league_id = leagues.
id LEFT OUTER JOIN players ON players.team_id = teams.id
WHERE (leagues."name" = 'Northwest')
```

Even better, there will never be more than one query, no matter how many rows are in the database.

But this isn't the best way. We're still using Ruby to filter the records, which can be slow. We're still returning more records than we need, which results in extra traffic between your application server and your database server.

An even better way is to write a small amount of custom SQL that returns only the records we need. To do this, we'll need to rethink the query. It's not really about the league, it's about players. We still want to return the name of the team and the name of the league for each player, but we can start by filtering the players by their all-star status.

```
# A better way, with custom SQL
@all_stars = Player.find(:all, {
  :conditions => [
    "leagues.name = ? AND players.all_star = ?",
    "Northwest", true
  ],
  :include => {:team => :league}
})
```

Because we're writing a snippet of SQL, we have to break the veil of plurality. ActiveRecord table names are plural by default. When referring to the names of joined tables directly (as with the first value passed to `:conditions`), you'll need to use the plural. In the example above, `leagues` and `players` are named directly in the conditional, so we have to use the plural table name, not the singular model name.

The log here is still only one query, but now the database is doing the work of filtering the players, which will usually be much faster than asking Ruby to do it.

```
Player Load Including Associations (0.002233)   SELECT
players."id" AS t0_r0, players."name" AS t0_r1,
players."team_id" AS t0_r2, players."all_star" AS t0_r3,
teams."id" AS t1_r0, teams."name" AS t1_r1, teams."league_
id" AS t1_r2, leagues."id" AS t2_r0, leagues."name" AS t2_r1
FROM players LEFT OUTER JOIN teams ON teams.id = players.
team_id LEFT OUTER JOIN leagues ON leagues.id = teams.
league_id WHERE (leagues.name = 'Northwest' AND players.
all_star = 't')
```

RESOURCES

- RailsCasts on include (http://railscasts.com/episodes/22)
- Include by Default plugin (http://blog.jcoglan.com/includebydefault)

:select and :include

There is a tricky interaction between the `:select` and `:include` options. If you are constructing special queries for reports, you'll need to understand how they work together.

:INCLUDE CLOBBERS :SELECT

If a query contains an `:include` directive, any `:select` options will be ignored. For example you might expect this to return an array of ids only. Not so!

```
# This won't work as you expect
Farm.find(:all, {
  :select  => "farms.id, chickens.id",
  :include => :chickens
})
```

The `:include` will force all fields to be returned, no matter what you pass to `:select`.

USE :JOIN WITH :SELECT

The way around this is to code the `:join` explicitly.

```
Farm.find(:all, {
  :select  => "farms.id, chickens.id",
  :join    => "LEFT JOIN ..."
})
```

Rails has softened my brain so that in my (not very) old age I often forget the exact SQL join syntax that I want to use. To be honest, I always had to look it up online even before I had access to a nice ORM library.

My modus operandi is to type my query into the Rails console and look at the output in the development log. I can then copy and paste the proper join string back into my code in the right place.

```
LEFT OUTER JOIN chickens ON chickens.farm_id =
farms.id
```