

Yet another way to connect to more than one database from Ruby on Rails

La Coctelera
ruby@the-cocktail.com

April 2007

Abstract

The *share-nothing* architecture allows to easily scale a Rails application by *simply* adding more application servers. But, as Twitter developer Alex Payne's interview¹ pointed out last week, this approach will eventually take the bottleneck to the database server. In this paper we discuss how we were hit by that issue in La Coctelera², and how we solved it.

1 Scaling the application layer

Scaling the application layer on a Ruby on Rails site is simple, so there is no point in explaining it again; there are many good references out there, e.g. in the Ruby on Rails wiki and in the *Agile*³ book. You may use Mongrel or use FastCGI, Add More Servers, putting them in your Lighttpd configuration, or in your Apaches's mod_proxy.

2 Scaling the database layer

This is not trivial, but it shouldn't be difficult either.

Setting up database replication depends heavily on the database engine you are using and it's far beyond the scope of this paper, but the basic idea is simple: having one master database server which deals with all the write queries (and, eventually, with some reads), and one or more slaves, which will only be queried in read-only mode. The way you achieve this is up to you, but it should not be very difficult. For MySQL, for example, the process is detailed in the book *High performance MySQL*⁴

¹<http://www.radicalbehavior.com/5-question-interview-with-twitter-developer-alex-payne/>

²<http://www.laocotelera.com/>

³<http://www.pragmaticprogrammer.com/titles/rails/index.html>

⁴<http://www.oreilly.com/catalog/hpmysql/index.html>

If you have more than one slave server, you will need to do balancing between them. We have successfully tested `pen`⁵ (but not intensively because we have only one slave server in production) for this task.

3 Rails app configurations

In our approach, you will have to add the configuration for your read-only database server in your `config/database.yml`. This could be the real server (if you have only one) or a balancing service such as `pen` (if you have more than one), and with the name `production_ro`:

```
[...]  
  
production:  
  adapter: mysql  
  database: lacocstelera  
  host: papalagi  
  username: lacocstelera  
  password: secretsecretsecret  
  
production_ro:  
  adapter: mysql  
  database: lacocstelera  
  host: maitai  
  username: lacocstelera  
  password: secretsecretsecret
```

For development and testing purposes, you can also add `development_ro` and `testing_ro`.

4 Filtering write and read queries

If you inspect the code, you will see that there is only one point where Rails establishes the connection with the database, which is a method called (not surprisingly) `ActiveRecord::Base.establish_connection`. When our code takes control on the request, the connection has already been established, but we can call this method again on a different database, and this is the connection that will be used. Before every database query we could (re-)establish the connection to the database we want, but that would be tedious, inefficient, error prone and tightly coupled (not the kind of code we want in our application). We think it is better done at the request level instead. When we start processing a request, we establish a connection to either the master or the slave database (or balancer), which is the one that gets used. The code looks like this:

⁵<http://siag.nu/pen/>

```

class ApplicationController < ActionController::Base
  def select_database
    db_conf = ActiveRecord::Base.configurations
    env = if <your logic>
            RAILS_ENV
          else
            RAILS_ENV + '_ro'
          end
    config = db_conf[env] || db_conf[RAILS_ENV]
    ActiveRecord::Base.establish_connection(config)
  end
end

```

Your logic is replaced with statements used to identify the requests in which writes to the database are needed. This will of course depend on your application. If you are being totally RESTful (or at least semantic), you can just check for *create*, *update*, *delete* (or so). In our case, we needed a finer control, so we coded a constant array with the actions we were balancing⁶ with the format `controller#action` in our `config/environment.rb`:

```

APP_RO_ACTIONS = %w{
  tags#index tags#show
  frontpages#show
  posts#list posts#search
  posts#index posts#category
  [some more actions...]
  comments#list comments#index
}

```

So this is the real code of our filter:

```

def select_database
  db_conf = ActiveRecord::Base.configurations
  action = "#{params[:controller]}\##{params[:action]}"
  env = ( APP_RO_ACTIONS.index(action).nil? ?
          RAILS_ENV : RAILS_ENV + '_ro' )
  config = db_conf[env] || db_conf[RAILS_ENV]
  ActiveRecord::Base.establish_connection(config)
end
end

```

This might not be the prettiest solution, and is far simpler than the one proposed by DrNic⁷, but for the single requirement we had, which is querying two different databases depending on the controller and the action, it does the job fairly well.

⁶We preferred to make the default the **write** actions, as to try to write, by error, in a read-only database would result in a 500 error.

⁷http://magicmodels.rubyforge.org/magic_multi.connections/