02

Performance e otimizações

Performance e otimizações

Durante o capítulo anterior, adicionamos a associação entre *companies* e *jobs* juntamente com filtros de controle de acesso, garantindo que empresas consigam manipular apenas os *jobs* criados por elas. Deixamos a aplicação em um estado totalmente funcional mas com uma questão de performance que é bastante simples de resolver, e vamos atacá-la no decorrer deste capítulo.

Evitando o problema de performance N+1

Na página que exibe a lista de *jobs* em nossa aplicação, além de mostrarmos o título de cada *job*, adicionamos também o nome da empresa à que ele pertence. Dessa forma, temos 1 consulta ao banco de dados para obter todos os *jobs* existentes, que retornará um número N de *jobs*, e como para cada *job* executamos uma nova consulta para obter sua empresa, criamos o problema de consultas N+1. Podemos confirmar isto através do log da aplicação: com o servidor rodando, acesse a listagem de *jobs* em *http://localhost:3000/jobs (http://localhost:3000/jobs)*, depois verifique o log do servidor no terminal e você deverá ver algo como:

```
Started GET "/jobs" for 127.0.0.1 at 2013-02-18 10:18:46 -0300
Processing by JobsController#index as HTML
  Job Load (0.3ms)  SELECT "jobs".* FROM "jobs" ORDER BY created_at DESC
  Company Load (0.1ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" = 3 LIMIT 1
   (0.1ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."job_id" = 4
  Company Load (0.1ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" = 2 LIMIT 1
  CACHE (0.0ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" = 3 LIMIT 1
   (0.1ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."job_id" = 3
  CACHE (0.0ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" = 2 LIMIT 1
   (0.1ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."job_id" = 2
  CACHE (0.0ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" = 2 LIMIT 1
   (0.1ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."job_id" = 1
  Rendered jobs/_job.html.erb (8.9ms)
  Rendered jobs/index.html.erb within layouts/application (9.5ms)
Completed 200 OK in 42ms (Views: 39.8ms | ActiveRecord: 0.8ms)
```

Veja que para cada um dos 4 *jobs* que temos no banco de dados, estamos executando uma nova consulta pela empresa relacionada. O Active Record é esperto o suficiente para tentar otimizar isso ao máximo, fazendo *cache* de consultas que já foram feitas durante a mesma requisição. Por exemplo, se existem 2 *jobs* pertencendo à mesma empresa, somente 1 consulta baterá no banco de dados, a outra retornará antes pois o Active Record identificará que ela já foi feita anteriormente (veja as linhas que contém `CACHE`). Mesmo assim, o problema de performance existe e será agravado conforme o número de *jobs* e *companies* no banco de dados cresce, pois aumentamos o número de consultas por empresas diferentes, degradando assim a performance da aplicação. Mas então, como resolvemos isto sem encher nosso código de lógica?

Simples! É muito mais rápido executarmos uma consulta que retorne todas as empresas de uma só vez, do que executarmos várias pequenas consultas para retornar empresa por empresa. Então ao invés de deixarmos as empresas serem carregadas quando exibimos os *jobs*, vamos instruir o Active Record a pré-carregar todas as empresas para nós de uma só vez! Abra o *app/controllers/jobs_controller.rb* e altere as *actions index* e *premium* para adicionar o comando *includes(:company)* ao buscar os *jobs*, como abaixo:

```
def index
  @jobs = Job.most_recent.includes(:company).all

  # more...
end

def premium
  @jobs = Job.where(premium: true).most_recent.includes(:company).
    paginate(page: params[:page], per_page: 10)
end
```

Isto vai fazer com que todas as empresas ligadas aos *jobs* que estamos buscando sejam carregadas com apenas uma consulta adicional ao banco de dados. Volte ao navegador e atualize a página, depois cheque novamente o log:

```
Started GET "/jobs" for 127.0.0.1 at 2013-02-18 10:31:15 -0300
Processing by JobsController#index as HTML
  Job Load (0.1ms)  SELECT "jobs".* FROM "jobs" ORDER BY created_at DESC
  Company Load (0.2ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" IN (3, 2)
   (0.1ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."job_id" = 4
  Company Load (0.1ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" = 2 LIMIT 1
   (0.1ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."job_id" = 3
   (0.1ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."job_id" = 2
   (0.1ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."job_id" = 1
  Rendered jobs/_job.html.erb (12.1ms)
  Rendered jobs/index.html.erb within layouts/application (12.8ms)
Completed 200 OK in 58ms (Views: 17.9ms | ActiveRecord: 1.9ms)
```

Veja que agora o Active Record buscou por todas as empresas de uma só vez, com a cláusula SQL `companies.id IN (3, 2)`. Excelente, evitamos de forma simples essa questão de performance da aplicação com a inclusão de apenas um comando, deixando todo o trabalho sujo para o Active Record.

No entanto, se observarmos esse log ainda veremos que existem algumas consultas sendo executadas para calcular a quantidade de comentários para cada *job* sendo exibido. Isso acontece porque, juntamente com as informações do *job* na listagem, estamos mostrando a quantidade de comentários, causando outro problema similar de *N+1*. Porém, não faz muito sentido para nós pré-carregarmos os comentários da mesma forma que fizemos com a empresa, pois não exibimos **nada** dos comentários nessa página além da contagem, e provavelmente teremos muito mais comentários do que vagas em si, então seria um desperdício de recursos. Vamos resolver isso utilizando outra funcionalidade do Active Record: **counter cache**.

Fazendo cache de contadores com counter_cache

O Active Record possui uma funcionalidade bem interessante chamada **counter cache** que nos permite criar um *cache de contagem*, que traduzindo para o nosso caso, é uma maneira de armazenarmos o número de comentários em cada *job*, ao invés de calcularmos este número toda vez executando uma consulta no banco de dados.

O *counter cache* funciona da seguinte maneira: cada vez que um comentário é criado, o contador de comentários do *job* relacionado é incrementado, e cada vez que um comentário é removido, esse contador é decrementado. Dessa forma, o contador permanece sempre atualizado com o número total de comentários associados ao *job*, e poderemos simplesmente exibir este contador ao invés de executarmos a consulta que fazemos hoje.

Criar um *counter cache* é bastante simples, primeiramente devemos adicionar um campo à tabela *jobs* para armazenar esse contador, contudo esse campo deve seguir a convenção do Rails para facilitar nossa vida: como dizemos *a job has many comments*, o campo será chamado **comments_count**, ou seja, o nome da associação **comments** com o sufixo **_count**. Você pode criar o campo com outro nome se desejar, mas preferimos seguir a convenção não é? Execute o comando para gerar a *migration*:

```
$ rails g migration add_comments_count_to_jobs comments_count:integer
```

É sempre uma boa prática garantir que os contadores sejam inicializados com `0`, então vamos abrir essa *migration* e adicionar a opção `default: 0`, indicando para o banco de dados que o valor `0` será usado para inicializar esse campo sempre que um novo *job* for inserido:

```
class AddCommentsCountToJobs < ActiveRecord::Migration
  def change
    add_column :jobs, :comments_count, :integer, default: 0
  end
end
```

Atualize então o banco de dados rodando as *migrations* em seguida:

```
$ rake db:migrate
```

Perfeito, o banco de dados está pronto! Agora precisamos informar ao modelo *Comment* que esse contador existe, para que ele execute o trabalho de mantê-lo atualizado para nós. Abra o modelo *app/models/comment.rb* e adicione a opção `counter_cache: true` à linha `belongs_to :job`, como abaixo:

```
class Comment < ActiveRecord::Base
  belongs_to :job, counter_cache: true
  # more...
end
```

Nota: caso você decida utilizar um nome diferente de *comments_count* para o campo, você precisa identificar este nome com a opção *counter_cache*, como em `counter_cache: "special_comments_count"`.

E pronto, temos o nosso contador! Contudo, como é possível que você já tenha criado alguns comentários no seu banco de dados (mesmo que apenas testando a aplicação), precisaremos gerar uma nova *migration* para atualizar o contador dos *jobs* relacionados a estes comentários, caso contrário eles estarão incorretamente zerados. Execute:

```
$ rails g migration update_comments_count_in_jobs
```

Então abra essa *migration* em seu editor, e altere-a como abaixo:

```
class UpdateCommentsCountInJobs < ActiveRecord::Migration
  def up
    Job.find_each do |job|
      Job.reset_counters job.id, :comments
    end
```

```
    end

    def down
    end
  end
```

No método `up` que será chamado quando a *migration* for executada, percorremos a lista de *jobs* existentes, e para cada *job* chamamos `reset_counters`, passando o `id` do *job* e a associação que possui o *counter_cache*, nesse caso *comments*. Execute então a *migration*:

```
$ rake db:migrate
```

Desta forma garantimos que os contadores estarão corretos desde o início.

Nota: Não precisamos adicionar nada ao método `down` pois não há necessidade de revertermos essa alteração caso a *migration* seja revertida.

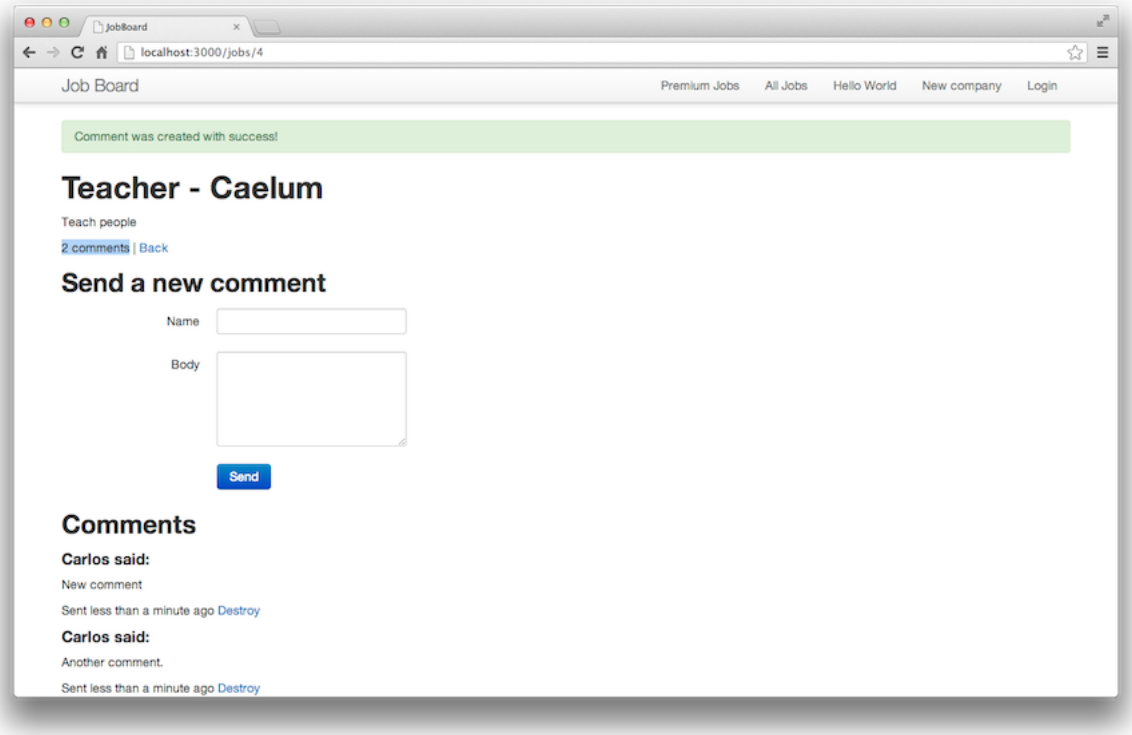
Vamos verificar se nosso contador está funcionando como esperado. Volte ao navegador, atualize a listagem de *jobs*, verifique os contadores de comentários e cheque novamente o log do Rails:

```
Started GET "/jobs" for 127.0.0.1 at 2013-02-18 12:00:28 -0300
Processing by JobsController#index as HTML
  Job Load (0.1ms)  SELECT "jobs".* FROM "jobs" ORDER BY created_at DESC
  Company Load (0.2ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" IN (3, 2)
  Company Load (0.2ms)  SELECT "companies".* FROM "companies" WHERE "companies"."id" = 2 LIMIT 1
  Rendered jobs/_job.html.erb (9.7ms)
  Rendered jobs/index.html.erb within layouts/application (10.5ms)
Completed 200 OK in 40ms (Views: 16.6ms | ActiveRecord: 1.2ms)
```

Tente também criar e remover alguns comentários através da interface e observe o log, você deverá ver comandos *SQL* para atualizar o contador, como esses:

```
Started POST "/jobs/1/comments" for 127.0.0.1 at 2013-02-18 15:50:45 -0300
...
  SQL (0.1ms)  UPDATE "jobs" SET "comments_count" = COALESCE("comments_count", 0) + 1 WHERE "jobs"."id" = 1
...
Redirected to http://localhost:3000/jobs/1


Started DELETE "/comments/1" for 127.0.0.1 at 2013-02-18 15:49:55 -0300
...
  SQL (0.4ms)  UPDATE "jobs" SET "comments_count" = COALESCE("comments_count", 0) - 1 WHERE "jobs"."id" = 1
...
Redirected to http://localhost:3000/jobs/1
```



Veja que não há mais consultas para calcular o número de comentários, excelente! Eliminamos mais um problema de performance com uma funcionalidade que o Active Record nos proporciona, sem muito esforço. Mas devemos nos perguntar: não tivemos que fazer absolutamente nenhuma alteração nas *views* para exibir o *comments_count* ao invés da contagem que era calculada anteriormente, como isso é possível? Novamente o Active Record está sendo esperto e nos ajudando, mas vamos entender isto mais de perto.

Diferenças entre count, length e size em associações has_many

Para entendermos como o Active Record utilizou corretamente o *counter cache* que definimos, vamos fazer alguns testes e conferir os resultados através do console do Rails. Mas antes disto abra a *partial app/views/jobs/_job.html.erb*, e identifique a linha que exibe a quantidade de comentários:

```
<%= pluralize(job.comments.size, "comment") %>
```

O importante aqui é notar que o número de comentários está sendo exibido através do código `job.comments.size`. Existem basicamente 3 maneiras de se obter essa contagem, através dos métodos `count`, `size`, e `length`, porém cada um deles atua de maneira diferente dependendo do estado da associação. Vamos utilizar a associação *Company has many Jobs* para entendermos melhor como esses métodos funcionam, apenas para facilitar pois não existe *counter cache* envolvido, e voltaremos neste código em instantes. Abra um console do Rails, e siga os passos abaixo:

```
>> company = Company.last
  Company Load (0.2ms)  SELECT "companies".* FROM "companies" ORDER BY "companies"."id" DESC LI
=> #<Company id: 3, name: "Caelum", email: "contato@caelum.com.br", encrypted_password: "$2a$10
>> company.jobs.count
   (0.1ms)  SELECT COUNT(*) FROM "jobs" WHERE "jobs"."company_id" = 3
=> 2
>> company.jobs.size
   (0.2ms)  SELECT COUNT(*) FROM "jobs" WHERE "jobs"."company_id" = 3
=> 2
>> company.jobs.length
```

```
  Job Load (0.2ms)  SELECT "jobs".* FROM "jobs" WHERE "jobs"."company_id" = 3
=> 2
```

Precisamos prestar bastante atenção nas consultas sendo efetuadas ao banco de dados. Quando executamos `company.jobs.count`, o Active Record disparou uma consulta do tipo `COUNT` ao banco, para retornar **somente** o número total de *jobs*, mas **sem carregá-los**. O mesmo acontece quando executamos `company.jobs.size`, como podemos observar na segunda consulta `COUNT`. Contudo, quando chamamos `company.jobs.length`, o Active Record não disparou uma consulta `COUNT`, mas carregou para memória **todos** os *jobs* relacionados à empresa em questão antes de retornar o número total. Interessante não é? Vamos continuar com os testes, e executar os 3 comandos novamente na mesma ordem:

```
>> company.jobs.count
   (0.2ms)  SELECT COUNT(*) FROM "jobs" WHERE "jobs"."company_id" = 3
=> 2
>> company.jobs.size
=> 2
>> company.jobs.length
=> 2
```

Podemos notar agora que há uma diferença: o método `count` continuou executando uma consulta `COUNT`, porém o `size` não disparou nenhuma consulta! O que acontece é que o `size` é esperto o suficiente para identificar que os *jobs* já foram carregados do banco de dados, então ele sabe que não precisa ir até lá novamente executando uma nova consulta, ele simplesmente retorna o número de *jobs* já carregados. Esperto não? E o `length` vai simplesmente retornar o mesmo número de *jobs* que já foram carregados, exatamente como antes.

Em resumo, estes métodos funcionam da seguinte forma:

-
- `count`: **sempre** consulta o banco de dados para retornar o número de itens;
-
- `length`: **sempre** carrega a associação caso não esteja carregada, e então retorna o número de itens;
-
- `size`: quando a associação não está carregada, ele funciona como o `count`, indo até o banco de dados; já quando ela está carregada, ele é similar ao `length`, retornando o número de itens já em memória.

Ou seja, o `size`, que é o método que estamos utilizando para exibir o número de comentários, é o mais esperto de todos, pois ele identifica o estado da associação para saber como retornar o número de itens de forma mais performática, indo ou não ao banco de dados. E aqui é que está o pulo do gato! O `size` também é capaz de identificar que existe um *counter cache* envolvido na associação, e nesse caso, ele o utiliza e simplesmente retorna seu valor, ao invés de ir até o banco de dados. É exatamente isto que acontece em nosso código que exibe o número de comentários:

```
<%= pluralize(job.comments.size, "comment") %>
```

Por estarmos utilizando `job.comments.size`, ele já identificou o `counter_cache` que acabamos de adicionar e fez uso dele, evitando as consultas adicionais ao banco de dados. Bacana não é? Você pode também executar alguns testes alterando esse código para utilizar `count` e `length`, e verificar os resultados através do log do servidor.

Com isso finalizamos o capítulo 4! Aprendemos a identificar os problemas de performance mais vistos em aplicações Rails e a solucioná-los de forma bastante simples utilizando funcionalidades que o Active Record provê para nós. É importante salientar que otimizar prematuramente é normalmente considerado desnecessário na grande maioria das aplicações, pois durante as etapas iniciais de desenvolvimento dificilmente encontraremos problemas com performance. Contudo, identificar e solucionar um problema de *N+1* como o que vimos neste capítulo requer praticamente zero de código e é algo que conseguimos evitar facilmente graças ao Active Record, além de ser um conhecimento que pode vir a muito útil em diversas situações.

E para o próximo capítulo vamos tornar o envio de comentários mais dinâmico utilizando AJAX, vejo você lá!

Para saber mais

-
- Veja mais sobre *N+1* e o método `includes` do Active Record no Rails Guide (http://guides.rubyonrails.org/active_record_querying.html#eager-loading-associations)
-
- Consulte a documentação do método reset_counters (http://api.rubyonrails.org/classes/ActiveRecord/CounterCache.html#method-i-reset_counters) para atualizar *counter caches*.
-
- Conheça a *gem bullet* (https://rubygems.org/gems/bullet), que pode vir a ser uma mão na roda para auxiliar a identificar problemas de *N+1* em aplicações existentes.