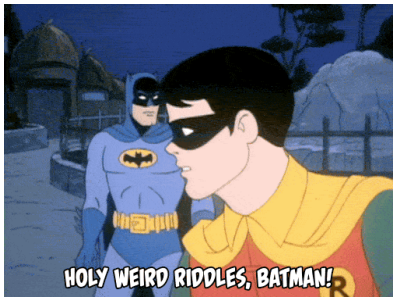
3 ActiveRecord Mistakes That Slow Down Rails Apps: Count, Where and Present

by **Nate Berkopec** (@nateberkopec) of ***speedshop*** (who?), a Rails performance consultancy.

Summary: Many Rails developers don't understand what causes ActiveRecord to actually execute a SQL query. Let's look at three common cases: misuse of the count method, using where to select subsets, and the present? predicate. You may be causing extra queries and N+1s through the abuse of these three methods. *(2778 words / 12 minutes)*

ActiveRecord is great. Really, it is. But it's an abstraction, intended to insulate you from the actual SQL queries being run on your database. And, if you don't understand how ActiveRecord works, you may be causing SQL queries to run that you didn't intend to.



"When does ActiveRecord execute queries? No one knows!"

Unfortunately, the performance costs of many features of ActiveRecord means we can't afford to ignore unnecessary usage or treat our ORM as just an implementation detail. We need to understand exactly what queries are being run on our performance-sensitive endpoints. Freedom isn't free, and neither is ActiveRecord.

One particular case of ActiveRecord misuse that I find is common amongst my clients is that ActiveRecord is executing SQL queries that aren't really necessary. Most of my clients are completely unaware that this is even happening.

Unnecessary SQL is a common cause of overly slow controller actions, especially when the unnecessary query appears in a partial which is rendered for every element in a collection. This is common in search actions or index actions. This is one of the most common problems I encounter in my performance consulting. It's a problem in nearly every app I've ever worked on.

One way to eliminate unnecessary queries is to poke our heads into ActiveRecord and understand its internals, and know exactly

how certain methods are implemented. **Today, we're going to look at the implementation and usage of three methods which cause lots of unnecessary queries in Rails applications: `count`, `where` and `present?`.**

How Do I Know if a Query is Unnecessary?

I have a rule of thumb to judge whether or not any particular SQL query is unnecessary. Ideally, a Rails controller action should execute **one SQL query per table**. If you're seeing more than one SQL query per table, you can usually find a way to reduce that to one or two queries. If you've got more than a half-dozen or so queries on a single table, you almost definitely have unnecessary queries. [1]

The number of SQL queries per table can be easily seen on NewRelic, for example, if you have that installed.

[1] Please don't email or tweet with me with 'Well ackshually…' on this one. It's a guideline, not a rule, and I understand there are circumstances where more than one query per table is a good idea.

Another rule of thumb is that **most queries should execute during the first half of a controller action's response, and almost never during partials**. Queries executed during partials are usually unintentional, and are often N+1s. These are easy to spot during a controller's execution if you just read the logs in development mode. For example, if you see this:

I keep an eyewash station next to my desk for really bad N+1s

```
User Load (0.6ms)  SELECT  "users".* FROM
"users" WHERE "users"."id" = $1 LIMIT 1  [["id",
2]]
Rendered posts/_post.html.erb (23.2ms)
User Load (0.3ms)  SELECT  "users".* FROM
"users" WHERE "users"."id" = $1 LIMIT 1  [["id",
3]]
Rendered posts/_post.html.erb (15.1ms)
```

... you have an N+1 in this partial.

Usually, when a query is executed halfway through a controller action (somewhere deep in a partial, for example) it means that you haven't `preload`ed the data that you needed.

So, let's look specifically at the `count`, `where` and `present?` methods, and why they cause unnecessary SQL queries.

.count executes a COUNT every time

I see this one at almost every company I contract for. It seems to be little-known that calling `count` on an ActiveRecord relation will *always* try to execute a SQL query, every time. This is inappropriate in most scenarios, but, in general, **only use `count` if you want to always execute a SQL COUNT *right now***.

The most common cause of unnecessary `count` queries is when you `count` an association you will use later in the view (or have already used):

"How many queries do we want per table?"

```
# _messages.html.erb
# Assume @messages = user.messages.unread, or
something like that

<h2>Unread Messages: <%= @messages.count %></h2>

<% @messages.each do |message| %>
blah blah blah
<% end %>
```

This executes 2 queries, a `COUNT` and a `SELECT`. The COUNT is executed by `@messages.count`, and

`@messages.each` executes a SELECT to load all the messages. Changing the order of the code in the partial and changing `count` to `size` eliminates the `COUNT` query completely and keeps the `SELECT`:

```
<% @messages.each do |message| %>
blah blah blah
<% end %>

<h2>Unread Messages: <%= @messages.size %></h2>
```

Why is this the case? We need not look any further than the actual method definition of `size` on ActiveRecord::Relation:

```
# File
activerecord/lib/active_record/relation.rb, line
210
def size
  loaded? ? @records.length : count(:all)
end
```

If the relation is loaded (that is, the query that the relation describes has been executed and we have stored the result), we call `length` on the already loaded record array. That's just a simple Ruby method on Array. If the ActiveRecord::Relation *isn't* loaded, we trigger a `COUNT` query.

On the other hand, here's how `count` is implemented (in ActiveRecord::Calculations):

```
def count(column_name = nil)
  if block_given?
    # ...
    return super()
  end

  calculate(:count, column_name)
end
```

And, of course, the implementation of `calculate` doesn't memoize or cache anything, and executes a SQL calculation every time it is called.

Simply changing `count` to `size` in our original example would have still triggered a `COUNT`. The record's wouldn't be `loaded?` when `size` was called, so ActiveRecord will still attempt a `COUNT`. Moving the method *after* the records are loaded eliminates the query. Now, moving our header to the end of the partial doesn't really make any logical sense. Instead, we can use the `load` method.

```
<h2>Unread Messages: <%= @messages.load.size %>
</h2>

<% @messages.each do |message| %>
blah blah blah
<% end %>
```

`load` just causes all of the records described by `@messages` to load immediately, rather than lazily. It returns the ActiveRecord::Relation, not the records. So, when `size` is called, the records are `loaded?` and a query is avoided. Voilà.

What if, in that example, we used `messages.load.count`? We'd still trigger a COUNT query!

When *doesn't* `count` trigger a query? Only if the result has been cached by `ActiveRecord::QueryCache`.[2] This could occur by trying to run the same SQL query twice:

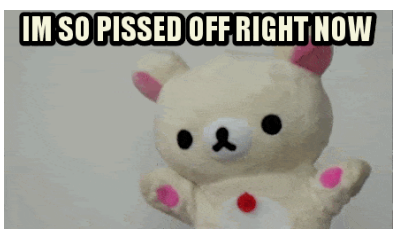
```
<h2>Unread Messages: <%= @messages.count %></h2>

... lots of other view code, then later:

<h2>Unread Messages: <%= @messages.count %></h2>
```

[2] I have some Opinions on the use of QueryCache, but that's a post for another day.

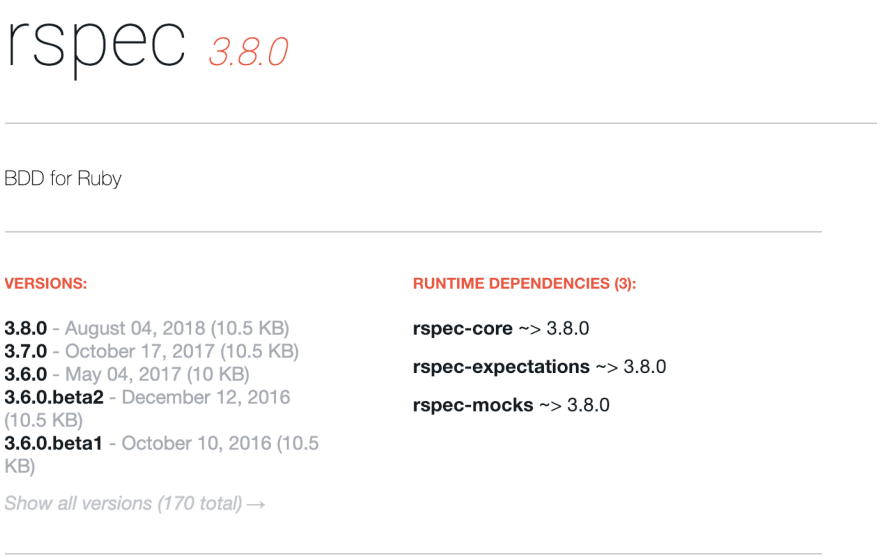
In my opinion, most Rails developers should be using `size` in most of the places that they use `count`. I'm not sure why everyone seems to write `count` instead of `size`. `size` uses `count` where it is appropriate, and it doesn't when the records are already loaded. I think it's because when



Every time you use count when you could have used size

you're writing an ActiveRecord relation, you're in the "SQL" mindset. You think: "This is SQL, I should write count because I want a COUNT!"

So, when do you actually want to use `count`? Use it when you won't actually *ever* be loading the full association that you're `count`ing. For example, take this view on Rubygems.org, which displays a single gem:



In the "versions" list, the view does a `count` to get the total number of releases (versions) of this gem.

Here's the actual code:

```
<% if show_all_versions_link?(@rubygem) %>
  <%= link_to t('.show_all_versions', :count =>
@rubygem.versions.count),
rubygem_versions_url(@rubygem), :class =>
```

```
"gem__see-all-versions t-link--gray t-link--has-
arrow" %>
<% end %>
```

The thing is, this view *never* loads *all* of the Rubygem's versions. It only loads five of the most recent ones, in order to show that versions list.

So, a `count` makes perfect sense here. Even though `size` would be logically equivalent (it would just execute a COUNT as well because `@versions` is not `loaded?`), it states the intent of the code in a clear way.

My advice is to grep through your `app/views` directory for `count` calls and make sure that they actually make sense. If you're not 100% sure that you really need a real SQL `COUNT` right then and there, switch it to `size`. Worst case, ActiveRecord will still execute a `COUNT` if the association isn't loaded. If you're going to use the association later in the view, change it to `load.size`.

.where means filtering is done by the database

What's the problem with this code (let's say its `_post.html.erb`)

```
<% @posts.each do |post| %>
  <%= post.content %>
  <%= render partial: :comment, collection:
post.active_comments %>
<% end %>
```

and in Post.rb:

```
class Post < ActiveRecord::Base
  def active_comments
    comments.where(soft_deleted: false)
  end
end
```

If you said, "this causes a SQL query to be executed on every rendering of the post partial", you're correct! `where` always causes a query. I didn't even bother to write out the controller code, because *it doesn't matter*. You can't use `includes` or other preloading methods to stop this query. `where` will always try to execute a query!

This also happens when you call scopes on associations. Imagine instead our Comment model looked like this:

```
class Comment < ActiveRecord::Base
  belongs_to :post
```

```
  scope :active, -> { where(soft_deleted: false)
}
end
```

Allow me to sum this up with two rules: **Don't call scopes on associations when you're rendering collections** and **don't put query methods, like `where`, in instance methods of an ActiveRecord::Base class**.

Calling scopes on associations means we cannot preload the result. In the example above, we can preload the comments on a post, but we can't preload the *active* comments on a post, so we have to go back to the database and execute new queries for every element in the collection.

This isn't a problem when you only do it once, and not on every element of a collection (like every post, as above). Feel free to use scopes galore in those situations - for example, if this was a PostsController#show action that only displayed one post and its associated comments. But in collections, scopes on associations cause N+1s, every time.

The best way I've found to fix this particular problem is to **create a new association**. Justin Weiss, of "Practicing Rails", taught me this in this blog post about preloading Rails scopes. The idea is that you create a new association, which you *can* preload:

```
class Post
  has_many :comments
  has_many :active_comments, -> { active },
class_name: "Comment"
end

class Comment
  belongs_to :post
  scope :active, -> { where(soft_deleted: false)
}
end

class PostsController
  def index
    @posts = Post.includes(:active_comments)
  end
end
```

The view is unchanged, but now executes just 2 SQL queries, one on the Posts table and one on the Comments table. Nice!

```
<% @posts.each do |post| %>
  <%= post.content %>
  <%= render partial: :comment, collection:
post.active_comments %>
<% end %>
```

The second rule of thumb I mentioned, **don't put query methods, like where, in instance methods of an ActiveRecord::Base class**, may seem less obvious. Here's an example:

```
class Post < ActiveRecord::Base
  belongs_to :post

  def latest_comment
    comments.order('published_at desc').first
  end
```

What happens if the view looks like this?

```
<% @posts.each do |post| %>
  <%= post.content %>
  <%= render post.latest_comment %>
<% end %>
```

That's a SQL query on every post, regardless of what you preloaded. In my experience, **every instance method on an ActiveRecord::Base class will eventually get called inside a collection**. Someone adds a new feature and isn't paying attention. Maybe it's by a different developer than the one who wrote the method originally, and they didn't fully read the implementation. Ta-da, now you've got an N+1. The example I gave could be rewritten as an association, like I described

earlier. That can still cause an N+1, but at least it can be fixed easily with the correct preloading.

Which ActiveRecord methods should we *avoid* inside of our ActiveRecord model instance methods? Generally, it's pretty much everything in the `QueryMethods`, `FinderMethods`, and `Calculations`. Any of these methods will usually *try* to run a SQL query, and are resistant to preloading. `where` is the most frequent offender, however.

any?, exists? and present?

Rails programmers have been struck by a major affliction - they're adding a particular predicate method to just about every variable in their applications. `present?` has spread across Rails codebases faster than the plague in 13th century Europe. The vast majority of the time, the predicate adds nothing but verbosity, and really, all the author needed was a truthy/falsey check, which they could have done by just writing the variable name.

Here's an example from CodeTriage, a free and open-source Rails application written by my friend Richard Schneeman:

```
class DocComment < ActiveRecord::Base
  belongs_to :doc_method, counter_cache: true

  # ... things removed for clarity...

  def doc_method?
    doc_method_id.present?
  end
end
```

What is `present?` doing here? One, it transforms the value of doc_method_id from either `nil` or an `Integer` into `true` or `false`. Some people have Strong Opinions about whether predicates should return true/false or can return truthy/falsey. I don't. But adding `present?` also does something else, and we have to look at the implementation to figure out what:

```
class Object
  def present?
    !blank?
  end
end
```

`blank?` is a more complicated question than "is this object truthy or falsey". Empty arrays and hashes are truthy, but `blank`,

and empty strings are also `blank?`. In the example above from CodeTriage, however, the only things that `doc_method_id` will *ever* be is `nil` or `Integer`, meaning `present?` is logically equivalent to `!!`:

```
def doc_method?
  !!doc_method_id
  # same as doc_method_id.present?
end
```

Using `present?` in cases like this is the wrong tool for the job. If you don't care about "emptiness" in the value you're calling the predicate on (i.e. the value cannot be `[]` or `{}`), use the simpler (and much faster) language features available to you. I sometimes see people even do this on values *which are already boolean*, which means you're just adding verbosity and making me wonder if there's some weird edge cases I'm not seeing.

Alright, that's my style gripe. I understand that you may not agree. `present?` makes more sense when dealing with strings, which can frequently be empty (`""`).

Where people get into trouble is calling predicates, such as `present?`, on

ActiveRecord::Relation objects. Let's say you need to know if an ActiveRecord::Relation has any records. You can use the English-language synonyms any?/present?/exists? or their negations none?/blank?/empty?. Surely it doesn't matter which method you choose, right? Just pick the one that sounds the most natural when read aloud? Nope.

What SQL queries do you think the following code will execute? Assume `@comments` is an ActiveRecord::Relation.

```
- if @comments.any?
  h2 Comments on this Post
  - @comments.each do |comment|
```

The answer is *two*. One will be an existence check, triggered by `@comments.any?` (`SELECT 1 AS one FROM ... LIMIT 1`), then the `@comments.each` line will trigger a loading of the entire relation (`SELECT "comments".* FROM "comments" WHERE ...`).

What about this?

```
- unless @comments.load.empty?
  h2 Comments on this Post
  - @comments.each do |comment|
```

This one only executes one query - `@comments.load` loads the entire relation right away with `SELECT "comments".* FROM "comments" WHERE ....`

And this one?

```
- if @comments.exists?
  This post has
  = @comments.size
  comments
- if @comments.exists?
  h2 Comments on this Post
  - @comments.each do |comment|
```

Four! `exists?` doesn't memoize itself and it doesn't load the relation. `exists?` here triggers a `SELECT 1 ...`, `.size` triggers a `COUNT` because the relation hasn't been loaded yet, and then the next `exists?` triggers ANOTHER `SELECT 1 ...` and finally `@comments` loads the entire relation! Yay! Isn't this fun? You could reduce this down to just 1 query with the following:

```
- if @comments.load.any?
  This post has
  = @comments.size
  comments
- if @comments.any?
  h2 Comments on this Post
  - @comments.each do |comment|
```

And it just gets better - this behavior changes depending if you're on Rails 4.2, Rails 5.0 or Rails 5.1+.

Here's how it works in Rails 5.1+:

method	**SQL generated**	**memoized?**	**implementation**	**Runs query if `loaded?`**
present?	SELECT "users".* FROM "users"	yes (`load`)	Object (!blank?)	no
blank?	SELECT "users".* FROM "users"	yes (`load`)	`load`; `blank?`	no
any?	SELECT 1 AS one FROM "users" LIMIT 1	no unless `loaded`	`!empty?`	no
empty?	SELECT 1 AS one FROM "users" LIMIT 1	no unless `loaded`	`exists?` if `!loaded?`	no
none?	SELECT 1 AS one FROM "users" LIMIT 1	no unless `loaded`	`empty?`	no

method	SQL generated	memoized?	implementation	Runs query if `loaded?`
exists?	SELECT 1 AS one FROM “users” LIMIT 1	no	ActiveRecord::Calculations	yes

Here’s how it works in Rails 5.0:

method	SQL generated	memoized?	implementation	Runs query if `loaded?`
present?	SELECT “users”.* FROM “users”	yes (`load`)	Object (!blank?)	no
blank?	SELECT “users”.* FROM “users”	yes (`load`)	`load` ; `blank?`	no
any?	SELECT COUNT(*) FROM “users”	no unless `loaded`	`!empty?`	no
empty?	SELECT COUNT(*) FROM “users”	no unless `loaded`	count(:all) > 0	no

method	SQL generated	memoized?	implementation	Runs query if `loaded?`
none?	SELECT COUNT(*) FROM “users”	no unless `loaded`	`empty?`	no
exists?	SELECT 1 AS one FROM “users” LIMIT 1	no	ActiveRecord::Calculations	yes

Here’s how it works in Rails 4.2:

method	SQL generated	memoized?	implementation	Runs query if `loaded?`
present?	SELECT “users”.* FROM “users”	yes	Object (!blank?)	no
blank?	SELECT “users”.* FROM “users”	yes	to_a.blank?	no
any?	SELECT COUNT(*) FROM “users”	no unless `loaded`	`!empty?`	no

method	SQL generated	memoized?	implementation	Runs query if `loaded?`
empty?	SELECT COUNT(*) FROM “users”	no unless `loaded`	count(:all) > 0	no
none?	SELECT “users”.* FROM “users”	yes (`load` called)	Array	no
exists?	SELECT 1 AS one FROM “users” LIMIT 1	no	ActiveRecord::Calculations	yes

`any?`, `empty?` and `none?` remind me of the implementation of `size` - if the records are `loaded?` do a simple method call on a basic Array, if they’re not loaded, *always run a SQL query*. `exists?` has no caching or memoization built in, just like other ActiveRecord::Calculations. This means that `exists?`, which is another method people like to write in these circumstances, is actually much worse than `present?` in some cases!

These six predicate methods, which are English-language synonyms all asking

the same question, have completely different implementations and performance implications, and these consequences depend on which version of Rails you are using. So, let me distill all of the above into some concrete advice:

- `present?` and `blank?` should not be used if the ActiveRecord::Relation will never be used in its entirety after you call `present?` or `blank?`. For example, `@my_relation.present?; @my_relation.first(3).each`.
- `any?`, `none?` and `empty?` should probably be replaced with `present?` or `blank?` unless you will only take a section of the ActiveRecord::Relation using `first` or `last`. They will generate an extra existence SQL check if you're just going to use the entire relation if it exists. In essence, change `@users.any?; @users.each...` to `@users.present?; @users.each...` or `@users.load.any?; @users.each...`, but `@users.any?; @users.first(3).each` is fine.
- `exists?` is a lot like `count` - it is never memoized, and always executes a SQL query.

Most people probably do not actually want this behavior, and would be better off using `present?` or `blank?`

Conclusion

As your app grows in size and complexity, unnecessary SQL can become a real drag on your application's performance. Each SQL query involves a round-trip back to the database, which entails, usually, at *least* a millisecond, and sometimes much more for complex `WHERE` clauses. Even if one extra `exists?` check isn't a big deal, if it suddenly happens in every row of a table or a partial in a collection, you've got a big problem!

ActiveRecord is a powerful abstraction, but since database access will never be "free", we need to be aware of how ActiveRecord works internally so that we can avoid database access in unnecessary cases.

App Checklist

- Look for uses of `present?`, `none?`, `any?`, `blank?` and `empty?` on objects which may be ActiveRecord::Relations. Are you just going to load the entire array later if the relation is present? If so, add `load` to the call (e.g. `@my_relation.load.any?`)
- Be careful with your use of `exists?` - it ALWAYS executes a SQL query. Only use it in cases where that is appropriate - otherwise use `present?` or any other the other methods which use `empty?`
- Be extremely careful using `where` in instance methods on ActiveRecord objects - they break preloading and often cause N+1s when used in rendering collections.
- `count` always executes a SQL query - audit its use in your codebase, and determine if a `size` check would be more appropriate.

Want a faster website?

I'm Nate Berkopec (@nateberkopec). I write online

about web performance from a full-stack developer's perspective. I primarily write about frontend performance and Ruby backends. If you liked this article and want to hear about the next one, click below. I don't spam - you'll receive about 1 email per week. It's all low-key, straight from me.

yourawesomeemail@cool.com

SUBSCRIBE!

The Complete Guide to Rails Performance

Look what I wrote! The Complete Guide to Rails Performance is a full-stack course that gives you the tools to make Ruby on Rails applications faster, more scalable, and simpler to maintain. It includes a 361 page PDF, private Slack, and over 15 hours of video content.

LEARN MORE

More Posts

The World Follows Power Laws: Why Premature Optimization is Bad

Programmers vaguely realize that 'premature optimization is bad'. But what is premature optimization? I'll argue that any optimization that does not come from observed measurement, usually in production, is premature, and that this fact stems from natural facts about our world. By applying an empirical mindset to performance, we can...

Read more

Why Your Rails App is

Slow: Lessons Learned from 3000+ Hours of Teaching

I've taught over 200 people at live workshops, worked with dozens of clients, and thousands of readers to make their Rails apps faster. What have I learned about performance work and Rails in the process? What makes apps slow? How do we make them faster?

Read more

The Complete Guide to Rails Performance, Version 2

I've completed the 'second edition' of my course, the CGRP. What's changed since I released the course two years ago? Where do I see Rails going in the future?

Read more

A New Ruby Application

Server: NGINX Unit

NGINX Inc. has just released Ruby support for their new multi-language application server, NGINX Unit. What does this mean for Ruby web applications? Should you be paying attention to NGINX Unit?

Read more

Malloc Can Double Multi-threaded Ruby Program Memory Usage

Memory fragmentation is difficult to measure and diagnose, but it can also sometimes be very easy to fix. Let's look at one source of memory fragmentation in multi-threaded CRuby programs: malloc's per-thread memory arenas.

Read more

Configuring Puma, Unicorn and Passenger for Maximum Efficiency

Maximum Efficiency

Application server configuration can make a major impact on the throughput and performance-per-dollar of your Ruby web application. Let's talk about the most important settings.

Read more

Is Ruby Too Slow For Web-Scale?

Choosing a new web framework or programming language for the web and wondering which to pick? Should performance enter your decision, or not?

Read more

Railsconf 2017: The Performance Update

Did you miss Railsconf 2017? Or maybe you went, but wonder if you missed something on the

performance front? Let me fill you

in!

Read more

Understanding Ruby GC through GC.stat

Have you ever wondered how the heck Ruby's GC works? Let's see what we can learn by reading some of the statistics it provides us in the GC.stat hash.

Read more

Rubyconf 2016: The Performance Update

What happened at RubyConf 2016 this year? A heck of a lot of stuff related to Ruby performance, that's what.

Read more

What HTTP/2 Means for Ruby Developers

Full HTTP/2 support for Ruby web frameworks is a long way off - but that doesn't mean you can't benefit from HTTP/2 today!

Read more

How Changing WebFonts Made Rubygems.org 10x Faster

WebFonts are awesome and here to stay. However, if used improperly, they can also impose a huge performance penalty. In this post, I explain how Rubygems.org painted 10x faster just by making a few changes to its WebFonts.

Read more

Page Weight Doesn't Matter

The total size of a webpage, measured in bytes, has little to do with its load time. Instead, increase network utilization:

make your site preloader-friendly, minimize parser blocking, and start downloading resources ASAP with Resource Hints.

Read more

Hacking Your Webpage's Head Tags for Speed and Profit

One of the most important parts of any webpage's performance is the content and organization of the head element. We'll take a deep dive on some easy optimizations that can be applied to any site.

Read more

How to Measure Ruby App Performance with New Relic

New Relic is a great tool for getting the overview of the performance bottlenecks of a Ruby application. But it's pretty

extensive - where do you start? What's the most important part to pay attention to?

Read more

Ludicrously Fast Page Loads - A Guide for Full-Stack Devs

Your website is slow, but the backend is fast. How do you diagnose performance issues on the frontend of your site? We'll discuss everything involved in constructing a webpage and how to profile it at sub-millisecond resolution with Chrome Timeline, Google's flamegraph-for-the-browser.

Read more

Action Cable - Friend or Foe?

Action Cable will be one of the main features of Rails 5, to be released sometime this winter.

But what can Action Cable do for Rails developers? Are WebSockets really as useful as everyone says?

Read more

rack-mini-profiler - the Secret Weapon of Ruby and Rails Speed

rack-mini-profiler is a powerful Swiss army knife for Rack app performance. Measure SQL queries, memory allocation and CPU time.

Read more

Scaling Ruby Apps to 1000 Requests per Minute - A Beginner's Guide

Most "scaling" resources for Ruby apps are written by companies with hundreds of requests per second. What about scaling for the rest of us?

Read more

Secrets to Speedy Ruby Apps On Heroku

Ruby apps in the memory-restrictive and randomly-routed Heroku environment don't have to be slow. Achieve <100ms server response times with the tips laid out below.

Read more

Speed Up Your Rails App by 66% - The Complete Guide to Rails Caching

Caching in a Rails app is a little bit like that one friend you sometimes have around for dinner, but should really have around more often.

Read more

100ms to Glass with Rails and Turbolinks

Is Rails dead? Can the old Ruby

Is Rails dead? Can the old Ruby web framework no longer keep up in this age of "native-like" performance? Turbolinks provides one solution.

Read more