The Pragmatic Programmers

Rails Scales!

Practical Techniques for Performance and Growth

Cristian Planas

edited by Michael Swaine

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 3

Optimizing Data Access with ActiveRecord

You've heard the claim: "Rails doesn't scale." Like most generalizations, it's false, but that's small comfort when you're trying to figure out why this particular Rails application doesn't scale.

And in general, it's impossible to diagnose why a Rails application doesn't scale without performing a thorough investigation. It's the Anna Karenina principle: Every poorly performing system fails in its own unique way, having gotten there following its own peculiar path to failure.

Still, some paths are more common than others, so you can usually make an educated guess of why a system is performing poorly. Because *typically*, the performance bottleneck is going to be the database, and most commonly, certain data accesses executed by the Rails app.

This is completely normal. Nowadays, the vast majority of applications spend a lot of time moving data around: exposing information to end-users or to other applications via multiple interfaces, or just moving it from one data cluster to another. Given this, you can see why the tool you use to interact with the data is crucial. It will affect performance, reliability, and developer experience.

Ruby on Rails has a specific philosophy on how to interact with data that affects the application as a whole, including its performance and scalability. In this chapter, we will explore what this philosophy entails, while learning its quirks and the best way to use it to make our applications highly scalable.

Computer Science or Information Technology?

In 2017, The Economist published a story titled *The world's most valuable resource is no longer oil, but data*. The article argued that, by feeding the machine learning models that have disrupted countless industries, data has become a new kind of fuel,

one that powers the whole global economy. That may very well be true, but we software engineers know that data -in particular, digitized data- has been a crucial component of the vast majority of businesses long before the dawn of the AI revolution. In the 80s and 90s, when the field that is now commonly referred to as "computer science" or "software engineering" started to rise as a force to be reckoned with, the most commonly used term to refer to this new area was “IT”, Information Technology. While the term in the US has become associated with the least glamorous part of software engineering, to this day still in some languages a computer scientist is referred to with a word that plainly links the discipline with information: "Informatik" in German or "Informático” in Spanish are only two examples.

Referring to the systems we build as "information technology" may sound very old-fashioned, but I do like the term. It reminds software engineers of what is the real focus of our work: to give access to information. A very small percent of us work on parts of the field that don’t involve the direct manipulation of data: the amount of computer scientists that spent their days designing a new processor or even a new programming language is way smaller than the ones that work on extending or maintaining a set of APIs supported by an application written in Spring, Django, or Rails.

Managing Data the Rails Way

There are many different kinds of abstractions that can be used on the data layer; some with a lower level of abstraction and therefore “closer” to the way the database works, others with a higher one. Ruby is a high-level Object Oriented Programming (OOP) language, and therefore it makes sense that the most popular data-mapping abstractions would match its high-level, OOPish spirit. This type of data abstraction is called ORM; ORM standing for Object-relational Mapping. In essence, ORMs map run-time objects to a set of data; in a traditional RDBMS, each row would be mapped to a different object, with a class corresponding to each table.

The most popular Ruby ORM, and the one used by Rails, is ActiveRecord. AR is one of the best -if not the best- tool in the whole Ruby ecosystem, managing to hit some sweet spot: capturing the magic that is commonly associated with Rails while being expressive enough to feel somewhat transparent. AR offers a layer of abstraction on top of the database that significantly simplifies access to our data, additionally solving many of the issues related to interacting directly with the database.

Nevertheless, the use of any ORM, even one as good as ActiveRecord, comes with its own drawbacks. So let’s look at the most typical problems that arise with the use of any ORM and see how to fix them. After that, we will move

deeper into the data layer of our Rails application, hitting the database itself: We will learn how to analyze a given query to understand how the database is executing it, and how to optimize queries to maximize performance.

Removing n+1s and Preloading Data

Of all the issues in database access, there is one that is particularly notorious. Performance monitoring services give it special attention; it's even used as a common question in software engineering interviews. This problem is the n+1. It is particularly important, if, as with Rails, we use an ORM. In this section, you'll learn what n+1s are and how to detect them, and you'll learn techniques to avoid them.

Meeting Your Enemy: the n+1

N+1s arise when trying to fetch an N amount of records in a specific and highly inefficient way: instead of fetching all of them in one query, you go about fetching them one by one, thereby ending up executing N queries. Obviously, running `SELECT * FROM movies WHERE ID IN (1, 3, 4, 5, 6, 7, 8, 9, 10)` performs significantly better than executing 10 separate queries. Despite being a very well-known and even fairly obvious issue to the experienced developer, it's still one of the most frequent sources of problems in Rails applications, to the point that some of the most popular performance monitoring platforms (like New Relic) have specific mechanisms to detect it.

If you were writing your own queries, it's unlikely that you would fall into this particular pitfall since it would be obvious that the same query was being rewritten over and over. The n+1 query problem is an issue in ORMs because of that abstraction layer between database and application.

The companion application for this book has an n+1 problem, and in this section, we are going to fix it. Take a look at the app now. The issue is in the `stores#show` action. The n+1 is triggered by the following view, that you can find in `app/views/stores/show.slim`:

```
table
  - @films.each do |film|
    tr
      td = film.title
      td = film.language.name
```

Looks harmless, doesn't it? Unfortunately, it isn't. The last line of this view will run N queries, where N is the number of films owned by the store: this means hundreds of unnecessary queries. There's a good chance this won't be detected until the code hits production, as the datasets that are typically

used in testing and staging environments tend to be smaller. In mature systems, production data can be big enough for an n+1 to cause serious trouble, including outages.

Next, you are going to find proof of the problem by calling that action in your own instance of the application. Access the URL of stores/1/movies, so the view we have just seen gets rendered. Check the log, and you will find something similar to this:

```
Started GET "stores/1/movies" for ::1 at 2022-08-15 23:58:35 -0400

Film Load (100.6ms)  SELECT "films".* FROM "films"
INNER JOIN "inventories" ON "films"."id" = "inventories"."film_id"
WHERE "inventories"."store_id" = $1  \[\["store_id", 1\]\]
    ↳ app/views/stores/show.html.slim:2

Language Load (1.2ms)  SELECT "languages".* FROM "languages"
WHERE "languages"."id" = $1 LIMIT $2  \[\["id", 105\], \["LIMIT", 1\]\]
    ↳ app/views/stores/show.html.slim:5

Language Load (0.9ms)  SELECT "languages".* FROM "languages"
WHERE "languages"."id" = $1 LIMIT $2  \[\["id", 106\], \["LIMIT", 1\]\]
    ↳ app/views/stores/show.html.slim:5

Language Load (0.2ms)  SELECT "languages".* FROM "languages"
WHERE "languages"."id" = $1 LIMIT $2  \[\["id", 103\], \["LIMIT", 1\]\]
    ↳ app/views/stores/show.html.slim:5

# Lots of very similar looking queries here...

Rendered stores/show.html.erb within layouts/application
(Duration: 3923.8ms | Allocations: 6551627)

Rendered layout layouts/application.html.erb
(Duration: 3945.4ms | Allocations: 6586873)

Completed 200 OK in 4054ms
(Views: 3777.1ms | ActiveRecord: 229.4ms | Allocations: 6601655)
```

Executing this extremely simple action took almost 4 seconds! Of that time, the majority was spent in the view. Moreover, the n+1 is also causing havoc in our memory usage. Check the "Allocations" part in the three last lines; we are allocating over 6.5 million objects in memory every time we render this view. This is bananas!

The log presented us with a very bleak situation, but also one that is fairly typical of an n+1 problem. The log showed how our Rails application is executing hundreds of different SELECT calls on the languages table, which is a waste. N+1 is a problem that leaves a very recognizable pattern in the logs, with a flood of extremely similar-looking SELECT statements, one after the other. While there are more sophisticated ways to catch n+1s (many perfor-

mance monitoring systems auto detect them), in most cases a quick glance at the log of a slow request will reveal the culprit association. Unfortunately, complex applications can make n+1 detection difficult, but that's when we can use more sophisticated tools to catch n+1s. The Bullet gem[1] has been around for more than 10 years, and not only detects n+1s but also unnecessary eager loads and opportunities to use counter-caching. Prosopite (https://github.com/charkost/prosopite) is a popular gem of very recent creation that is able to detect n+1 in some edge cases that are not currently supported by Bullet. Moreover, most APM systems (APM stands for Application Performance Monitoring), like Scout or New Relic, feature n+1 autodetection.

As you see, there are plenty of options to catch those pesky n+1s. It's not surprising, because an n+1 is a bad problem to have in your application, one that can become even worse in certain situations: for example, if the database is in a different host than the Rails application, something fairly typical in production environments. In this case, each query will imply another trip around the network, increasing the gravity of the problem. N+1s can become even worse in certain situations: for example, if the database is in a different host than the Rails application, something fairly typical in production environments. In this case, each query will imply another trip around the network, increasing the gravity of the problem.

Fortunately, fixing n+1s is extremely simple thanks to ActiveRecord.

1. https://github.com/flyerhzm/bullet