The Mythical IO-Bound Rails App

Jan 23, 2025

I want to write a post about Pitchfork, explaining where it comes from, why it is like it is, and how I see its future. But before I can get to that, I think I need to explain a few things.

When the topic of Rails performance comes up, it is commonplace to hear that the database is the bottleneck, so Rails applications are IO-bound anyway, hence Ruby performance doesn't matter that much, and all you need is a healthy dose of concurrency to make your service scale.

But how true is this in general?

Conflating scale and performance

First, it is true that when scaling a Rails application, the first major bottleneck you will encounter will generally be the database.

Rails, like the overwhelming majority of modern web frameworks, is stateless, hence it is trivial to scale it horizontally to handle as much load as you want. As long as you can keep adding server capacity, Rails will keep scaling. It might not be the cheapest stack to scale, but it will scale as well as any other stateless web framework. To handle 10x more traffic, you need roughly 10x more server capacity, that's simple and not what teams who need to scale a Rails application will struggle with.

Relational databases, however, are much harder to scale. By default, you can't scale them horizontally unless you implement data sharding in some way, and depending on your data model, it can sometimes be very challenging, so generally relational databases are initially scaled vertically, meaning migrating to increasingly more powerful servers. Vertical scaling allows you to scale pretty far, much farther than most Rails users will ever need, but the cost increase won't be linear, and if you are successful enough, it won't be viable anymore and you'll have to shard or use another type of data store.

That's what the database being the bottleneck means. Not that querying the database is slow, nor that it is the most significant factor in overall service latency, but that it's the part of your infrastructure you'll have to care about the most when your service witnesses increasingly more usage.

That's what a lot of public discourse gets wrong, they conflate scale and performance, or more precisely, throughput and latency.

Being able to scale means maintaining some level of service while serving more users with close-to-linear costs. It doesn't mean being particularly fast, cheap, or efficient, it simply means being able to grow without hitting a bottleneck, and without costing you exponentially more

money.

As such, the database being the bottleneck is true, but it doesn't imply that the application is spending the majority of its time waiting on IO.

Most Rails performance issues are database issues

Another fact that is often stated to explain how Rails applications are IO-bound, is that the most common performance issues with Rails applications are missing database indexes, N+1 queries and other data access issues.

From my personal observations, that definitely rings true, but these are bugs, not a given property of the system. They're supposed to be identified and fixed, as such, it doesn't really make sense to design your infrastructure to accomodate that characteristic.

When properly indexed, and assuming the database isn't overloaded, the vast majority of queries, especially the mundane lookups by primary key, take less than a couple of milliseconds, often just a fraction of a millisecond. If the application does any substantial amount of transformations on that data to render it in HTML or JSON, it will without a doubt spend as much or more time executing Ruby code than waiting for IOs.

Evidence: YJIT effectiveness

Now of course, every application is different, and I can only really speak with confidence about the ones I've worked on.

However, over the last couple of years, many people reported how YJIT reduced their application latency by 15 to 30%. Like Discourse seeing a 15.8-19.6% speedup with JIT 3.2, Lobsters seeing a 26% speedup, Basecamp and Hey seeing a 26% speedup or Shopify's Storefront Renderer app seeing a 17% speedup.

If these applications were really spending the overwhelming majority of their time waiting on IO, it would be impossible for YJIT to perform this well overall.

Even on very JIT-friendly benchmarks with no IO at all, YJIT *only* speeds up Ruby by 2 or 3x. On more realistic benchmarks like `lobsters`, it's more around 1.7x. Based on this, we can assume with fairly good confidence that all these applications are certainly not spending 80% of their time waiting on IO.

To me, that's enough to consider that most Rails applications aren't IO-bound. Some applications out there most definitely are IO-bound, and I spoke with a few people maintaining such applications. But just like flying fishes, they exist, yet they don't constitute the majority of the genus.

CPU starvation looks like IO to most eyes

One thing that can cause people to overestimate how much IO their app is doing is that, in most cases, CPU starvation will look like Ruby is waiting on IO.

If you look at how the overwhelming majority of IO durations are measured, including in Rails logs and in all the most popular application performance managers, it's generally done in a very simple, obvious way:

```ruby
start = Time.now
database_connection.execute("SELECT ...")
query_duration = (Time.now - start) * 1000.0
puts "Query took: #{query_duration.round(2)}ms"
```

Logically, if this code logs: `Query took: 20.0ms`, you might legitimately think it took 20 milliseconds to perform the SQL query, but that's not necessarily true.

It actually means that performing the query **and** getting the thread scheduled again took 20 milliseconds, and you cannot possibly tell how much each part took individually (Edit: At John Duff's request, I wrote a very quick guide on how you can tell if your application is experiencing some form of CPU starvation).

So for all you know, the query might have been performed in under a millisecond, and all the remaining time was spent waiting to acquire the GVL, running GC, or waiting for the operating system scheduler to resume your process.

Knowing which is which is crucial:

- If all this time was spent performing the query, it suggests that your application is IO-heavy and that you may be able to use more concurrency (processes, threads, or fibers) to get some extra throughput.
- If all this time was spent waiting on the scheduler, then you might want to do the absolute opposite and use less concurrency to reduce latency.

This issue can often lead people to believe their application is more IO-heavy than it really is, and it's particularly vicious because it's a self-fulfilling prophecy. If you assume your application is IO-bound, given that it is commonplace to hear so, you'll logically run it in production with a threaded or async server using a decent amount of concurrency, and production logs will confirm your assumption by showing large amounts of time waiting on IO.

This problem is even more common on shared hosting platforms because they don't always guarantee that you can use the "virtual CPU" at 100%. On these platforms, your application has to share the physical CPU cores with other applications, and depending on what these other applications are doing and how busy they are, you may be able to use substantially more or substantially less than the one "virtual CPU" you are paying for.

This issue is not unique to Ruby, whenever you have a workload that mixes IO and CPU work, you have to arbitrate between allowing more concurrency and risk degrading latency, or reducing concurrency to ensure a low latency but decrease utilization. The more heterogeneous your workload is, as typical in a monolith, the harder it is to find the best compromise. That's one of the benefits of micro-services, getting more homogenous workloads, making it easier to achieve higher server utilization without impacting latency too much.

However, given that the default implementation of Ruby has a GVL, this problem is even more pronounced. Instead of all threads on the server having to share all CPU cores, you end up with multiple small buckets of threads that each have to share one CPU, hence you can end up with threads that can't be resumed even though there are some free cores or the server.

The thing to keep in mind is that, as a general rule that doesn't only apply to Ruby, CPU-bound and IO-bound work loads don't mix well, and should ideally be handled by distinct systems. For small projects, you can likely tolerate the latency impact of collocating all your workloads in a one-size-fits-all system, but as you scale you will increasingly need to segregate IO-intensive workloads from CPU-intensive ones.

Job Queues Are Different

One important thing to note is that what I'm saying above is only aimed at the web server part of Rails applications. Most apps also use a background job runner, Sidekiq being the most popular, and background jobs often take care of lots of slow IO operations, such as sending e-mails, performing API calls, etc.

As such job runners are generally much more IO intensive, and latency is generally a bit less important for them, so they usually can get away with higher concurrency than web servers.

But even then, it's common for users to crank their job runner concurrency too high and cause all IOs to appear much slower. A good example of that is how Sidekiq's maintainer asked me to implement a way to measure the round trip delay in C so that it's not skewed by GVL contention.

Why does it matter?

At that point, you might wonder why it matters how much time Rails applications spend waiting on IO.

For the average Rails user, it is important to know this, because it is what defines which execution model is best suited to deploy their application:

- If an application is truly IO-bound, as in spending more than 95% of its time waiting for IO, then using an asynchronous execution model is likely what will get you the best results.
- If an application isn't fully IO-bound, but still is quite IO-heavy, then using a threaded server, with a reasonable number of threads per process, is probably what will get you the best

tradeoff between latency and throughput.
- If an application doesn't spend significantly more than half its time on IO, then it might be preferable to use a purely process-based solution.

What the ratio of IO to CPU is likely to be like in an average Rails app is also what drove Rails to change the default generated Puma configuration from 5 threads to only 3.

But also for the Ruby community at large, I think it's important not to disregard Ruby's performance under the pretext that it doesn't matter since it's all about the database anyway. This isn't to say that Ruby is slow, it is without a doubt more than fast enough for writing web applications that provide a good user experience.

But it is also possible to write Ruby code in a way that doesn't perform well at all, be it for the sake of usability or just because it's fun to use meta-programing or that no one took the time to use a profiler to see if it is possible to do the same thing more efficiently.

As someone who spends a considerable amount of time looking at production profiles of Rails applications, I can say with confidence, there are a number of things in Rails and other commonly used gems that could be significantly faster, but can't because their public API prevents any further optimization.

As Ruby developers, we naturally have the tendency to put developer happiness first, which is great, but we should also make sure not to disregard performance. Usability and performance don't have to be mutually exclusive. APIs can be both very convenient and performant, but both have to be considered right from the start for it to happen. Once a public API has been defined and is widely used, there's only so much you can do to make it perform faster unless you are willing to deprecate it in favor of a more performant one, but the community appetite for deprecations and breaking change is no longer what it used to be.