Speedshop Tune: Slope Clinical

@nateberkopec

Prepared with care

@nateberkopec

2024-09-17

Contents

Hello Slope!

Thanks for having me take a look at your application. I've identified several areas for investment in performance. Some of the things I've discussed in this report are actually just simple configuration changes you can make today that will save you a lot of money. Others are more long-term projects and skills that you can improve on over the next six months.

This document is organized at the top level by our desired Outcomes, which are my goals for your performance improvements over the next six months. Underneath that are specific Recommendations to achieve those outcomes. Each Recommendation has an associated cost and benefit, rated subjectively on a 5 point scale.

At the end of the document, I will present again the Outcomes and Recommendations without commentary. This is intended to be a quick reference to help you turn this document into action and assist during planning your sprints, etc.

I hope you enjoy this document and find it a useful guide for the next 6 months of performance work on your application.

Nate Berkopec, owner of The Speedshop

Outcome: Create an anti-N+1 workflow

When I started this engagement and got onboarded, it was my impression that the biggest problem you wanted to solve was that the site starts to "feel" slow for customers when they get a moderate-to-large amount of data in their account. Solving this problem was the top priority of this audit, and comprises a majority of the recommendations and outcomes here.

The root cause of "more data means more slow" in Slope is N+1 queries (and in general, disorganized data access). This is not uncommon in apps in Slope's domain. In general, the actual row counts for each customer are not "big data" and there are very few issues with slow database queries (there's only one specific one actually). Instead, what we have is controller actions with hundreds of database queries that could have been done in just a handful.

In my experience, apps get into a low-performance state because their engineering loop is broken:

1. No one sets any requirements for what "performant" means. What latency is good enough, and what is failing?
2. Because there is no requirement, no one works on slow pages. Everyone understands the requirement for bugs: there shouldn't be any. If we find one, we put it in JIRA. How does performance get into JIRA?
3. No one builds the "muscle" of how to find/fix performance issues, performance deteriorates until customers complain (the ultimate performance monitoring system: support@mydomain.com).

Because of Slope's particular domain, the loop has created a backlog of N+1 issues.

I will introduce specific suggestions for specific N+1s later in the report. In this section, I want to focus specifically on the engineering conditions which created this situation: treat the disease, not the symptoms.

Finally, while I *say* "N+1", which refers speficially to a query pattern like:

```
SELECT * FROM `users` WHERE `users`.`id` = 1
SELECT * FROM `users` WHERE `users`.`id` = 2
```

```
SELECT * FROM `users` WHERE `users`.`id` = 3
...
```

What I really *mean* is "inefficient" SQL, a.k.a. any SQL that we could have done in fewer queries.

For example, consider a boolean column like "ready", which can be true or false:

```
SELECT * FROM `items` WHERE `items`.`ready` = true
SELECT * FROM `items` WHERE `items`.`ready` = false
```

Why do this in two queries (assuming the column cannot be NULL)? Just select all!

Recommendation: Set performance requirements

Engineering is the process of creating something which meets a set of requirements while working within a set of constraints.

Performance work is one of the most "pure engineering" things we get to do as web application developers. The system - the web application - must load the page within a certain amount of time to deliver an acceptable customer experience.

At Slope, where there is little-to-no frontend Javascript (more on that later), the performance experience on a day-to-day basis is driven by server response times. Slow server response, slow customer experience. The relationship is direct.

Any response from Slope.io that takes over 1 second is definitely a poor customer experience. If one customer experiences several of those in the course of working with the app, they're going to feel it. Of course, a 10 second response is 10x worse - at that point you're wondering if the app is even up at all.

In addition, in my experience, almost any server response that takes more than 1 second is probably deeply unoptimized. There's low hanging fruit there to be picked.

This means that any response over 1 second is "high benefit, low cost" to address.

This "1 second" threshold needs to be "operationalized" somehow. We need a red-light green-light system where something indicates failure, and something indicates success. This is how bugs work. There either are bugs, or there aren't. We know how to fix bugs. You see one, you put it in JIRA. Setting clear requirements allows us to use the same workflow for performance.

Requirements are magical for a perf work process. They allow us to say when perf work is required, but also when it is done. Everything in the green? Carry on - don't work on perf!

I recommend that Slope.io decide that **any web transaction with 100 or more 1 second responses in the last 7 days** is failing.

There's a little bit of art in this: you want the requirement to be float "lots of traffic, lots of pain" to the top. It also depends on the sample rate - if you double the amount of samples, you should also double this requirement. You also want the size of the "failed transaction" list that comes out the other end to feel tractable and not overwhelming. It needs to represent an achievable goal.

My recommended requirement generates the following list:

1. InventoriesController#show
2. WorkspacesController#catalog
3. InventoriesController#create
4. SitesController#clinical_trials
5. ItemsController#edit_multiple
6. SitesController#show
7. API::V1::SearchController#api_search
8. ItemsController#create

If you read that list and went "oh yeah, that one's definitely bad" at least once, that's good. Requirements should capture what's already floating around in our heads.

Each day, about 100 to 150 users of Slope experience a > 1 second load time. Each day, Sentry samples about 250 to 300 unique users. That mean about 40-50% of users experience at least one >1 second load time. Let's bring that number down.

I recommend you set other performance requirements which feel relevant to the business based around this "1 second" rallying call: no one should have to wait one second or more for anything! It should be a rare event, not something 50% of your users experience every day.

Cost: 1, Benefit: 2. This is the foundation of all performance work. Without this, perf work never makes it into eng's workload.

Recommendation: Invest in a solid Sentry workflow

You currently have both Sentry and Elastic installed. Elastic is neat, but it's too difficult for an application developer to use and understand (similar to Datadog, in that way). Elastic will probably fulfill your "infrastructure monitoring" needs, but Sentry is a far better "application monitor".

You need to create some "workflows" around "how to use Sentry effectively", and invest a bit in the tool to make it truly useful for you.

This recommendation encompasses a few different things:

1. Sample less.
2. Filtering out Sidekiq
3. Learn how to find problem transactions
4. Reading "Slowest Functions" dialog

5. Reading slow transaction traces
6. Reading slow profiles
7. Spans sort by throughput to find N+1

Let's take each in turn.

Sample less.

Sentry's "old" pricing, what you're currently on, is prohibitively expensive for transaction monitoring.

In the one week period prior to when I asked you to up your sample rate (thanks!), you accepted 200,000 transactions. So that's about 860,000 a month. Your client was sampling at about a 30% rate, so in reality you were doing 3 million transactions or so. According to sentry's pricing, 3 million performance units (I'm leaving out the profiling part for now) are 0.000130 * 3 million, or about $400. That's not bad. You should correct me if I'm wrong here, I can't see the actual spend in your dashboard.

In the new span quota system, you're charged per span rather per transaction (transactions have many spans). You could be doing about 100 million spans per month. However, given how complicated your controller actions are, each transaction contains a LOT of spans. I'm not sure the new pricing would be a better deal for you.

I think about $500/month in spend here is reasonable for a business of this size + the scope of your performance issues.

Sampling is particularly painful at Slope because your dataset is so small to begin with. You only have a couple thousand users doing low request volume.

Not sampling would also allow to target specific customer experiences. Joe from Joe Clinical complaining about site speed? Go check his transactions for the last week and see EXACTLY what was going FOR HIM. This is incredibly powerful, particularly for the kinds of issues you're having.

Learn how to filter out Sidekiq

Sidekiq job execution times are not your biggest issue. You need to know how, in Sentry, to filter them out of your 'transactions' lists and focus on web requests only.

In many searchboxes/filter boxes on Sentry, you can filter by `transaction.op`, op as in "operation". For web, it's `transaction.op=http.server`. Add this filter anytime you're working on perf. Unfortunately, you actually can't do this on the actual Backend Performance page. I've complained to the Sentry CEO about this on twitter (I know him personally/used to work for him on the Sentry ruby client), we'll see how quickly they fix it. On every other page in Sentry, you can use this filter.

Learn how to find problem transactions

Slope's biggest problem is that certain web transactions are delivering bad experiences.

There are a couple of ways to quantify a "bad experience". All are valid, none are perfect:

1. "Miserable" users. I talked in a previous recommendation about the "1 second threshold". In Sentry, you can say "a user is miserable if they experience 1 second web server response times" by using the "miserable(250)" function (misery = experiencing 4x the threshold). You can change the whole-project setting to use "250" as the threshold value (currently 300), but also can use this value specifically in graphs. I like this metric because it quantifies the number of humans experiencing a problem. Of course, the ideal is zero!
2. Apdex. Again, currently the threshold is not set to 250, but when you change it to 250, it again correlates with our 1 second threshold. In Apdex, a user experiencing 4x the threshold value is having a "terrible" experience and greatly affects the apdex score. What's a "good" apdex? To be honest, that's arbitrary. That makes this one a bit weaker as a metric. "Has an apdex score much lower than the rest of the site" is a good heuristic, i.e. "10 worst transactions by apdex".
3. p95. If the p95 is higher than 1 second, people are having "miserable" experiences > 5% of the time. That's bad.
4. Number of responses > 1 second. I've made a couple of dashboards for you that do this. This is basically equivalent to "number of miserable users" but it's based on "number of responses" instead "number of users", therefore capturing workflows that a user may do multiple times.

You can create dashboards for any/all of these or just explore the data using the Discover tab on the left. I use any/all of these to discover which transactions are performing poorly.

Reading "Slowest Functions" dialog

Once you're in Performance -> Backend -> click on an individual transaction in the list (example: inventories, at the bottom of the page if you scroll down you get a "slowest functions" list. This is based on aggregates of profiles, which I think is one of Sentry's coolest/best features.

I found this list very useful for understanding the *parent* of the N+1s in the app. For example, you may have a lot of Role N+1s caused by a line in user.rb, but what higher-level function is causing that? It shows up in this list.

I generally just look at the top 5 and try to look for outliers sorted by "total self time" (which is occurrences multiplied by average self time). For example, as I write this, Workspace#inventories_with_configurations is the top function on the list, 3x more "total self time" than 2nd place. That's really interesting,

and tells me this is the slowest method on the page and where we're spending a lot of time. I can click example profiles to dig into exactly why.

Reading slow profiles

Once you've clicked the profile, you're presented with a flamegraph from a single run of the transaction (don't bother with the aggregated profiles: I don't find them useful. instead, use profiles only for 1 specific customer request).

Reading flamegraphs is a bit of an art, and covered in a lot of my educational material. It's beyond the scope of this report. What I can tell you is: it's a skill, and you build skills through practice. Stare at flamegraphs, try to get insight from them, repeat.

In the case of Workspace#inventories_with_configurations, after looking at a profile, I learn that it's called in two different places, and basically just spends all of its time in ActiveRecord doing queries. Probably an N+1 issue. I'll learn what those queries are exactly from a different tool: traces.

Reading slow transaction traces

Profiles and traces work fundamentally differently:

1. Profiles are based on a "sampling profiler" interrupting the process every 5 ms and asking "what's your current stack?" and recording it.
2. Traces are based on hooks inserted in the code. For example, every time you start/end an activerecord query, we start and end a "span" in the trace. This is mostly implemented through ActiveSupport notifications.

Profiles are based on the statistical principle of sampling, while traces capture less detail but do not sample at all.

This means traces work really well for things like "show me all the DB queries that were run here"

On the individual transaction view page, you will also see a list of slow transaction traces. I love looking at these. It's maybe the most valuable thing in all of Sentry. Seeing exactly what queries ran for a particular user will probably be your most useful "N+1" tool.

Spans: sort by throughput to find N+1

Finally, on the same transaction summary page, there's a tab on the top called "spans" (NOT "aggregate spans").

I don't know why they chose to quantify it as "throughput/sec", but if you sort this by "throughput/sec", you get a list of the "most frequent spans" in the transaction. This effectively gives you a "N+1 priority list", as the spans at the top of this list are basically showing up the most often in every trace.

Cost: 2, Benefit: 3 Costs the wallet, and takes a bit of training time. If it's useful, I could record a video here to show my process.

Recommendation: Run rack-mini-profiler in all environments.

`rack-mini-profiler` is my favorite Rails performance tool of all time. On every request to your application, you get:

1. Response time displayed on-page
2. A list of every SQL query, and the in-app stacktrace for it.
3. A flamegraph
4. More - if you need it!

It's absolutely indispensable. It's also possible to run in production. You can turn it on for all your admin users.

It's important for devs to have performance tooling always available, always at hand, always on. Rack-mini-profiler is the foundational tool for that.

Cost: 1, Benefit: 3 Put it in the gemfile, add 10 lines of code for safely deploying it to prod. In prod, you will need to configure storage in Redis.

Recommendation: Use prosopite, remove bullet.

Over the years, I've learned that bullet promotes the wrong mindset in developers. With bullet, they think "as long as bullet has no warnings, I have no N+1s", and in addition, these warnings are easy to ignore.

Prosopite has several advantages over bullet:

1. You can run it in tests. Use tests to certify that your code has no N+1s!
2. The N+1 detection algorithm is much better and tuned more towards "no false positives", which leads to a high signal v noise experience.
3. It can `raise` on failure. I like this. I want things to go bang when they're wrong, like exceptions when code is incorrect. Obviously, don't do this in prod, but in other environments it works great.

Cost: 2, Benefit: 3 Setup could be a bit painful, as you have to probably set it up so that existing N+1s are "allowed" but you don't ship new ones. But I used this tool extensively at Gusto and loved it.

Recommendation: Build a better development seed

An effective performance workflow goes like this:

1. Discover issues in production (Sentry).
2. Reproduce them locally (rack-mini-profiler)

The trouble is, *especially* with N+1s, you need the data to cause the reproduction. Currently, your seed is insufficient to do that. Seed files which require developer maintenance inevitably go out of date (who can be bothered to write a feature, tests for it, AND update seed data?) and don't create enough records required to make N+1s obvious (it's different when you have 1 item per inventory vs 100).

Only production data really has that ability to show you all the crazy things your users are trying to do with your product.

Unfortunately, I think the options here are all pretty bad.

1. Build your own seed tool based on production data. This works well but would be a massive project for a team of this size.
2. Re-purpose an OSS tool like snaplet
3. Get a vendor who purports to do this for you. I have none to recommend.

Probably the only realistic path forward for Slope is something like Snaplet. You don't need to be able to pull the entire DB down to local (although you could), but just some slice of it (some group of suppliers/sites/studies).

Cost: 5, Benefit: 3. Probably my most difficult suggestion.

Recommendation: Set logs to debug in development

I noticed they're currently set to `:info`.

With `:info`, SQL queries are not logged to console. I don't like this. Personally, I always develop with my logs open. I can see them at all times. This means that when I navigate aroudn the site, I can *see* and *feel* N+1s, because I see them scroll by in the logs.

I realize not everyone wants to or will develop this way. But it makes N+1s *visible*, which is your main issue right now.

Cost: 0, Benefit: 1.

Recommendation: Treat ActiveRecord::QueryCache as a "mini" N+1.

When you run a query twice in a row, it hits ActiveRecord::QueryCache on the 2nd and subsequent runs. In Sentry, you'll see a `cache: true` tag on the span, and in your logs it looks like this:

```
CACHE Holler Load (0.0ms)  SELECT "hollers".* FROM "hollers" ORDER BY "hollers"."id" ASC LIN
```

The QueryCache is basically a Hash, limited to 1000 keys, that is cleared out on every request (or job run). However, the "0.0ms" is misleading.

QueryCache only prevents the query itself from calling the database. You still have to generate the SQL string and you still have to create an ActiveRecord

object from the result. This is far from free. In my testing, a QueryCache "hit" costs about 1/10th of a millisecond for a single record. This is fine to do once or twice, but most actions in your app hit the query cache hundreds of times. This is adding up.

Treat QueryCache hits as a "mini" n+1. They're not as important as real ones that go to the DB, but they also shouldn't be ignored "because they're cached".

Cost: 2, Benefit: 1

Recommendation: Access every table one time.

In general, my "heuristic" is that I want to access every table once. Usually I can craft a SQL query that gets everything this page needs from a table. I always want to reduce the number of queries-per-table to something close to 1. It's not always possible (or even always optimal), but it's a good method for reducing time spent accessing data.

Things like this:

```
SSODomain Load (0.4ms)  SELECT "sso_domains".* FROM "sso_domains" WHERE "sso_domains"."allow
SSODomain Load (0.1ms)  SELECT "sso_domains".* FROM "sso_domains" WHERE "sso_domains"."allow
```

Could be done in one query (true OR false).

You can also use things like `GROUP BY` to remove multiple queries (saw this one on the admin homepage):

```
ItemStateTransition Count (0.1ms)  SELECT COUNT(*) AS "count_all" ... WHERE (event = 'report
ItemStateTransition Count (0.1ms)  SELECT COUNT(*) AS "count_all" ... WHERE (event = 'report
```

Multiple queries will always be more expensive that doing it once in a single query.

Cost: 2, Benefit: 2 One of the easier refactorings to do in most cases. Can be tricky sometimes to figure out how to pass data around/between different parts of the code. Usually the answer is "@variable in the controller".

Recommendation: Never use query methods in instance methods.

You have a few ActiveRecord models that do things like this:

```
def actionable_supplier_orders_for(supplier)
  self.supplier_orders_for(supplier).where(state: ["approved", "acknowledged", "processing
end
```

This method will always cause a SQL query when called. The most common ActiveRecord methods that cause queries are `where`, `order`, and `find`.

The problem with doing this in an instance method is that **every instance method will inevitably be called in a loop**, triggering N+1.

This kind of code can also occur in a view helper. It's an instance method using a method *which always causes a sql query* when called.

Memoization can sometimes help, but not always. Memoization assumes you're actually calling the method twice on the same object, which is not always true. Only insert memoization when you can verify it actually fixes an N+1, observing the behavior in rack mini profiler or in the logs.

Instead of doing this kind of code in the model, it really should live in the controller layer or in a scope (class method) called by a controller. The controllers job is to determine which instances of the model need to be passed to the view, the models job is to describe behavior. Try not to mix these concerns.

Cost: 3, Benefit: 4 High cost because this pattern is particularly entrenched in code. High benefit because it is the main cause of your N+1 issues.

Recommendation: Never use "fake" ActiveRecord associations.

Throughout the codebase, but particularly on user, you have instance methods whose job it is to return an array of associated ActiveRecord models. A good example is `User#organizations`.

```
def organizations
  if self.is_organization?
    organizations =
      self.roles.where(name: %i[organization_admin organization_staff])
        .includes(:resource).collect(&:resource)
  elsif self.is_admin? || self.is_observer?
    return Organization.all
  else
    return Organization.none
  end
end
```

1. You don't get memoization by default. Calling this multiple times is a multiple N+1, compounded by queries in is_organization and is_admin/is_observer (although caught by query cache)
2. You cannot preload/eager load this.
3. You don't get any of the other niceties given to you by an ActiveRecord association.

For this example, all the required data is in the `roles` and `users_roles` tables. We essentially have a list of names we care about: `admin`, `observer`, `organization_admin`, and `organization_staff`.

I spent a long time trying to find a SQL-powered solution here. It does exist (essentially: for each role in the users_roles/roles join, use a subquery to check

for is_admin?/is_observer?) but try as I might, I couldn't make it work as an ActiveRecord association at the same time.

It's actually pretty easy to get an association working for the non-admin/non-observer case. But this behavior of "admins/observers return all" is tough to express in SQL.

I see a few options here, going forward:

1. Change table layout, such that this is easy to express with SQL (not sure what this is yet)
2. Change behavior, so that admins/observers do not return all

This is really a fundamental problem in the code, and leads to a huge amount of your SQL querying issues. I believe it cannot be punted/ignored, and must be solved.

Cost: 4, Benefit: 5

Recommendation: Stop caching

This may sound like a weird one. Specifically, I'm referring to your use of Rails.cache, and the `cache` view helper (essentially: everything in memcache).

Application caching is for making fast sites faster. It's like the frosting on top of a cake. It's how you go from 200ms responses to sub-100ms. It's not how you fix 500ms+ responses, which is Slope's biggest issue.

Slope's use of caching exhibits all the problems I typically see with sites trying to use application caching for the wrong reasons.

Your **hitrate is very low**. The cache hitrate is hovering around ~66%. That means 1/3 of the time, the cache is doing nothing but adding latency. Ouch. It also means you're going to have a bigger cache (lower hitrate = higher memory use).

You're **not using multifetch** in a lot of places you could be. Calls like:

```
<%= render partial: 'sponsors/skus_table_row', collection: @skus, as: :sku, cached: true %>
```

… result in a single cache fetch, as Rails calculates and combines the cache fetches for each element in the collection. However, calls like:

```
<% @users.each do |user| %>
<% cache user do %>
```

… result in a single round-trip to the cache store for every @user.

You're using **highly complex cache keys** with more than 1 or 2 elements, which directly will result in a low hitrate as these keys are constantly changing. For example, in `turbo_table.html.erb`, you're using 5 different elements: `[current_user, current_user.updated_at, item, table_context, item.updated_at]`.

You're **introducing bugs** caused by expiration time. In this example in search.rb, a user's authorizations are cached for an hour. If their authorizations change during this time, they will still be able to access their content using their old auth:

```
Rails.cache.fetch(
  "policies/User:#{user.id}/#{object.class.name}:#{object.id}",
  expires_in: 1.hour
)
```

It's also **costing you time with writes**. The amount of `touch` relationships is quite high, resulting in lots of `UPDATE` load for even the most minor record changes.

Caching is not a good way to paper over poorly performing underlying data access. It's for making already fast-paths faster.

Cost: 1, Benefit: 1. You should still fix the underlying data access issues first, but with a hitrate this low, the cache could probably just be turned off with little impact.

Outcome: Lower p95 response times

Now, we move on to specific fixes for specific pages. These pages were ones I identified using the techniques in the previous section (high p95, high user misery) and the specific fixes found by reading through Sentry transaction traces and profiles.

Almost any page can and should have a p95 below 1 second. For each page in this list, I've listed the highest-leverage fix for getting you to that goal.

Recommendation: Fix Workspace#inventories_with_configurations

This method is used in `site.rb` in `any_workspaces_using_slsm?` and also directly in `_workspaces_table_row.html.erb`. It negatively impacts the performance of two important controllers: WorkspacesController#catalog, and InventoriesController#create.

In just one profile that I looked at, this method took >200ms to work through the N+1s it caused.

```
class Workspace < ApplicationRecord
  def inventories_with_configurations
    self.inventories.joins(:inventory_template_configurations).distinct.reject{|inventory| i
  end
end
```

As mentioned in a previous recommendation, this is an instance method on a ActiveRecord model which will generate a query every time it is run. It

also will load every `inventory` associated with the Workspace, and uses a fake ActiveRecord association:

```
class Inventory < ApplicationRecord
  def configurations
    self.inventory_template_configurations
  end

  def inventory_template_configurations
    InventoryTemplateConfiguration.where(site: self.site, inventory_template: self.inventory
  end
end
```

The solution in this particular case is making `inventories_with_configurations` into a true ActiveRecord association, and then preloading it:

```
has_many :inventories_with_configs, -> {
    joins(:inventory_template_configurations).distinct
  }, class_name: 'Inventory'
```

Like most of your other issues, I think that `join` will only work if the correct has_many/has_one/belongs_to relationships are set up on Inventory, InventoryTemplateConfiguration, Site, and InventoryTemplate. You may instead just create a new foreign key on InventoryTemplateConfigurations for the Inventory instead.

Cost: 2, Benefit: 3 Needs some AR unraveling but will have a good impact on two of the most important controller actions to fix.

Recommendation: InventoriesController#show: Count collections by loading and partitioning, or GROUPing

This particular controller suffers from an N+1 regarding COUNTs on the ITEMs table.

It looks like:

```
SELECT COUNT(*) FROM "items" WHERE "items"."inventory_id" = $1 AND ("items".state = 'dispat
```

This is then repeated for about a dozen different `state`s, likely most or all of them.

Then, you N+1 again when you actually SELECT each of these from the DB.

```
SELECT "items".*
FROM "items"
WHERE "items"."inventory_id" = $1 AND "items"."reserved" = $2 AND ("items"."state" IN (
  'ready'
)) AND NOT ("items"."state" IN (
  $3, $4, $5
```

```
) AND "items"."reserved" = $6)
/* Varous combinations of filters are applied to items over many queries */
```

In this case, you're basically just ending up running ~20+ queries over items attached to this inventory_id.

You should just fetch every single `item` associated with this `inventory_id`. It seems like most of the time, inventories don't have that many items, and even if they did, the combination of SQL queries here is basically loading them all in to memory *anyway*. So let's avoid doing that over 20 round trips to the database and do it in one.

Looking at profiles, its a bit hard to follow where exactly these queries are being generated from. It may be a variety of areas, which will require loading the items associated with the inventory and then passing them around.

Cost: 2, Benefit: 1. AFIACT this is limited to having an impact only on this controller. May require unravelling in several locations in the view.

Recommendation: Limit or consolidate link calls in jump_to

`WorkspacesController#catalog`, `ShipmentsController#show` and `SitesController#show` profiles (and slow traces) show this thing where it says there are long chunks of "missing span instrumentation". If you look at the profile itself (example user who encountered this:), you see that you're actually spending *loads* of time in the `_jump_to.html.erb` partial and ApplicationHelper#jump_to_for generating URLs. This can take several seconds for some users. This means that their jump to list is incredibly long.

There are probably a number of things you could do here, but the most obvious is just to limit the number of generated links. If you're generated hundreds of links here (which I think you are!), that's just a bad UX to begin with. Optimizing bad UX is a classic perf mistake: let's just fix the UX issue.

Most of the people actually encountering this are on staging with slope.io emails, but it also affects production users to a lesser extent.

Cost: 1, Benefit: 2 Seems to benefit mostly slope.io emails. Poor Mckenzie. It's really causing almost every page load they experience to be 2 sec+.

Recommendation: InventoriesController#create, ItemsController#edit_multiple, ItemsController#create: Background the alert updating callback

The number of callbacks executed on creating an Inventory or editing items is quite heavy. On one example profile, running eveyr callback took about 1.5 seconds. This seems common with bigger users.

After looking at a profile, I determined most of the time is being spent in running `Alert#updated`, which mostly spends it's time running `ExpiredItemAlert.make`. It kicks off a massive amount of writes. This is also true for `ItemsController#create`.

`ItemsController#edit_multiple`, this controller spends it's entire time in `ItemsController#transition_multiple_items`, also running the `Alert.updated/Expired/ExpiringItemAlert` dance.

This Alert infrastructure seems pretty complex in a lot of different ways. It almost seems like you've reimplemented an event stream in SQL and ActiveRecord.

Probably the simplest/easiest solution here is background this callback. It doesn't seem like Alerts need to be 100% up-to-date on the next pageload? It's possible I don't understand what Alerts are being used for.

Cost: 1, Benefit: 3 Cost is based on "just put it in background". Actually coming up with a lower-impact solution for Alerts is probably much more intensive, on the order of a 4 or 5. I would need a meeting to understand what the goal is with Alerts.

Recommendation: SitesController#clinical_trials: N+1 on Alert

There's a repeated N+1 here surrounding Workspace#has_expiring_items_alert? and orkspace#has_inventory_alert?, which is called for each row in `app/views/shared/_workspaces_table_row.html.erb`.

These relations are fundamentally just looking for a `ExpiringItemAlert`, `LowInventoryAlert` or `NoInventoryAlert` associated with the current Workspace. Again, the problem is that these are not really associations, and could not be preloaded with something like:

```
Workspace.preload(:expiring_item_alerts, :low_inventory_alerts, :no_inventory_alerts)
```

It's compounded by the fact that the alerts table is quite large, which makes these queries slow to satisfy (in the 30-50ms range), making this N+1 10x more expensive than other kinds. This page definitely suffers from it the worst due to the repeat calls in this row.

Make alert loading a true relation, and preload it.

Cost: 2, Benefit: 3: Would fix p95 issues w/this page entirely, but the relations look a bit complicated and would require some work to unwind I think.

Recommendation: Reduce overhead of ShipmentPolicy#scoped_shipment_role on API::V1::SearchController#api_search

Returning to the caching issue - every slow trace I found here was hitting the cold/uncached path for `Search#user_can_view?`, which would take 1-2 seconds to run for all the times it was called.

In the particular example I looked at, lots of time was spent in `ShipmentPolicy#show?` and `scoped_shipment_role`, which is a very complex method that maps through every single user role and every single resource associated with that user role (!!!) to gather every shipment associated with every single one of those resources (!!!!!).

This scheme is not scaling. I'm not sure how much of this controllers problems are associated with Shipment policies *specifically*, but I think fixing this single policy is a big enough bottleneck that we can at least apply our learnings to other policies.

Essentially, you want to know for each shipment:

- Is this user associated with this shipment somehow?
- If they are, through what role?

Iterating through *every single owned resource* to *run a SQL query looking for shipments* cannot be the fastest possible way to to do that.

Cost: 3, Benefit: 2 Somewhat limited benefit to just this search action. Cost high because I think it could entail rethinking how authorization works completely (though this is probably something you need to do anyway)

Recommendation: WorkspacesController#shipments N+1s

There's a few N+1s in this controller that have pretty big impact, as a result of looping through the `shared/shipments_table`.

Here's a pseudo-association of `Shipment#skus` which, called in a loop in the partial, causes N+1:

```
def skus
  skus = []
  self.manifests.each do |manifest|
    skus << manifest.source.sku if manifest.source.is_a?(Stock)
  end
  skus.uniq
end
```

Others are real associations that simply weren't preloaded, such as `origin` and `destination` and `protocols`.

Preload the associations and remove pseudo-associations here and this would be a fast action.

I think this action represents a good "place to get started", as the N+1s here are more straightforward than in other places of the codebase.

Cost: 1, Benefit: 1 Limited impact on fixing this, but easy to do.

Outcome: Lower p50 response times

The previous section focused on outlier responses: what happens to big users with "big" accountants that encounter big problems. This section focuses more on every-day, every-page problems. Fixing these problems lowers the "average" or "most-of-the-time" experience, rather than the extreme experiences.

Recommendation: Overhaul authorization

We've already talked a lot about the ways that authorization causes N+1s indirectly, because authorization checks on associations (`is_admin?`) causes additional lookups/creates pseudo-associations.

However, it also creates "regular old N+1s", particularly at the beginning of an action.

This is caused by various calls to methods like `User#is_sponsor_admin?` and `User#is_protocol_manager?`:

```
SELECT 1 AS one
FROM "roles"
INNER JOIN "users_roles" ON "roles"."id" = "users_roles"."role_id"
WHERE "users_roles"."user_id" = $1 AND (resource_type IN (
  'ProtocolSupplier'
)) AND "roles"."name" = $2
LIMIT $3
/* This is repeated for various Resource Types */
```

```
def is_workspace_manager?
  Workspace.find_roles(:workspace_manager, self).any?
end

def is_protocol_manager?
  Protocol.find_roles(:protocol_manager, self).any?
end

def is_sponsor_lab_manager?
  Lab.find_roles(:sponsor_lab_manager, self).any?
end
```

Instead of doing an existence query for every resource type, we should just look up all users roles once and store them in a single place, accessible by all parts of the app that need it (probably something like `current_user.roles`).

In my head, the methods above should look something like:

```
def is_workspace_manager?
  roles.workspace_manager.any?
end
```

I suspect in this case `roles` cannot be an ActiveRecord association (in this case - it would still cause an N+1) but instead will have to be a specialized in-memory object (i.e. a Repository pattern), containing all relevant `role` and `users_roles` records, with accessors that get the relevant info w/o further DB queries.

Cost: 3, Benefit: 3 Conceptually complex to setup (though not much code).

Recommendation: Fix alerts lookups, both in speed and N+1

Almost every action in the application has 2-3 lookups on the Alerts table, each of which takes 30-50ms.

Fundamentally, most of the slow query is caused by looking for large amounts of `container_id`s in an `IN` clause, i.e.:

```
SELECT COUNT(*)
FROM "alerts"
WHERE "alerts"."container_type" = $1 AND "alerts"."container_id" IN (
  $2, $3, ... $165
) AND "alerts"."type" NOT IN (
  $166, $167, $168
)
```

That's just not going to be a fast query. I'm fairly sure this is from `VisibleSidebars#count_alert_groups(alerts)`.

For perf reasons, I would simply reduce this feature to an existence check rather than an exact count. It's enough to know if you *have* an alert versus how many there exactly.

Cost: 1, Benefit: 1 Simple fix.

Recommendation: Log jobs upon completion, not enqueueing.

You've got this gem for logging ActiveJobs which runs on every job enqueueing (which is quite frequent for some actions).

I understand the desire to create audit trails, but I don't see this is as necessary for job enqueueing.

If you're worried about data consistency in general, then I think Redis is not for you. "Double-entry job bookkeeping" to avoid what you see as shortcomings of Redis as a backing store means you should just be queueing your jobs in SQL instead, using SolidQueue. IMO, you should just drop whatever this gem is doing on enqueueing, and keep it on completion of the job if you wish. It's adding *lots* of write load on job insertion, which will evenutally bite you.

Cost: 2, Benefit: 1 It's not the biggest impact on overall latency. I think the requirement here should be reassessed.

Outcome: Remove deployment footguns

I noticed a handful of minor issues which could cause bigger problems in the future if not eventually addressed. Rather than shoot ourselves in the foot, these "footguns" should be fixed at some point.

Recommendation: Instrument request queue time

Every single request to your backend from a browser client experiences latency along the following steps:

1. Time spent routing from the client to your server (network RTT, not really under your control)
2. Time spent routing inside your infra to a container (internal RTT, under your control but quite minimal time)
3. Time spent queueing, waiting for an empty Puma process (0 if any Puma process is available, but can be substantial if all are busy)
4. Time spent actually running the request (what Sentry reports).

Step 3 is a common snafu. You currently don't have this number recorded or reported anywhere. It can be a massive component in user resposne times if the deployment is under heavy load or is misconfigured. Imagine running a grocery store, but no one knew how long the lines for checkout were. That's what this is like.

Unfortunately Sentry **does not yet track this number**. It's kind of excusable because it's clearly not an "infra APM". But you need this number. Probably with your setup, you should be reporting it to Elastic.

Ideally, you scale up or down based on this number. When request queue time goes up, scale up. However, your load is so low and predictable, I don't yet recommend autoscaling. But you do need this number still so you can make accurate manual adjustments.

1. **Insert a time-since-epoch-in-milliseconds header** to each request at the first available point you can in your infrastructure. Usually this is the load balancer, sometimes the nginx/traefik/reverse-proxy in front of Puma.

2. **In a rack middleware, subtract Time.now - request_start_time**. Report to elastic. You now have request queue time.

Cost: 1, Benefit: 1. Traffic isn't a big deal for you, but I think this would be a short project.

Recommendation: Any job over 1 minute should be broken into chunks

Jobs that take over 1 minute, *particularly* in small applications with small amounts of resources, can be dangerous. There's a number of reasons:

1. They're usually not idempotent and are dangerous to kill midway through. What happens if a 30-minute job is killed at 15 minutes? Probably the last 15 minutes of work is never done. Yikes.
2. They hog queue resources for long amounts of time. Relevant to my concerns here, long-running jobs tend to "take down" Sidekiq threads as a result of just running so long. This is an issue when there aren't that many workers to begin with.

There are two good solutions:

1. The new Iterable job support in Sidekiq 7.3 allows you to solve the "correctness" concern by at least making every job idempotent and "checkpoint-able".
2. You can use a "fan-out" structure so that work is paralellized over many workers. Parent jobs fan out over many child jobs. Use queue weighting or sharding (see my Sidekiq in Practice course, which you should have access to) to avoid any issues regarding these new jobs clogging up all workers in the queue.

Cost: 2 Benefit: 1 These migrations can be a bit of work, but there's only a handful of jobs that need this.

Recommendation: Don't use $redis, use a connection pool. Remove "cache" usage from $redis.

You have a few problems here:

```
$redis = Redis.new(url: uri, ssl_params: { verify_mode: OpenSSL::SSL::VERIFY_NONE })
```

The most obvious is no SSL.

The next is that you're assigning a single Redis connection to a global variable. **This is not threadsafe**. You use this in a multithreaded environment today in your cache supplier counts jobs:

```
$redis.set "#{supplier.to_param}", supplier_data.to_json
```

Luckily, you don't immediately try to read this value in the job itself, so I think you're safe from any race conditions.

You should be using a connection pool from the `connection_pool` gem instead.

However, to be honest, this data should just be in Rails.cache anyway. Simply remove it, delete this Redis instance, and in the future, don't assign a single Redis connection to a global.

Cost: 0 Benefit: 0: It's basically dead code, but could cause multithreading bugs if built upon or expanded.

Recommendation: Install jemalloc.

Almost every client I've ever worked with has installed jemalloc and immediately saved ~10% of their memory usage. It's a no-brainer.

Adding it to your Dockerfile will take about 2-3 lines of code.

Cost: 0 Benefit: 1 Takes 15 minutes, but I don't think you're suffering from poor memory use.

Outcome: Add robustness to background jobs

I noticed a couple of issues regarding background jobs. Again, I feel this are mostly "future-proofing", much lower priority than the N+1 and web transaction latency issues.

Recommendation: Use SLA queues

You have essentially 2 main queues today, default and counts.

As applications grow, they start to try to increase the number of queues they have, as a response to various problems:

1. The default queue is overrun by a specific type of job. (Oh no, somebody enqueued 3 million Searchkick reindexes and all other jobs are not running!)
2. Specific jobs have weird needs, like high memory or low/no concurrency

Every single job in your system **implicitly has an SLO**. You expect AddPartsToDrugJob to execute in X amount of time or less, otherwise the system won't work as expected.

The idea of an "SLA queue" is that all queues in the system should simply be named after the service level they provide. For example:

1. asap
2. within_5_minutes
3. within_1_hour
4. within_1_day

This is an incredibly powerful idea. It:

- **Makes SLOs explicit**. Every time you write a job, you have to decide what the expectation is for its total time to run.
- **Makes alerts obvious**. How do you know when to scale up the default queue today? Hmmm. Well, if the queue was named "within 5 minutes" and queue latency was over 5 minutes, it's pretty obvious!
- **Allows load shedding** In a high-db-load scenario, you can pause the `within_1_day` queue for 23 hours and still be golden!

Your job load is low today, but I would look to migrate to this system now while overall job load isn't a big deal, rather than migrate 100s of jobs later when you feel the pain.

Cost: 3 Benefit: 1

Recommendation: Debounce Searchkick reindexing

You end up running the Searchkick reindexing job *a lot* (once every 2 seconds).

The `searchkick` gem shares a strategy for batching reindexes that would benefit you to implement.

Essentially it just involves changing to:

```
searchkick callbacks: :queue
```

and then running a ProcessQueueJob on a regular interval.

Cost: 1, Benefit: 1 Not a big deal for latency, just would reduce background job load substantially.

Recommendation: Set up Sentry Queues when supported

A new PR just got opened to make the Sentry ruby gem use the Sentry Queues product properly with Sidekiq. This would be a huge improvement on the observability of your queues (how long do jobs spend in the queue today? no idea.)

Set a notification on this one and upgrade as soon as released! It would be a great improvement.

Cost: 0, Benefit: 2 Huge improvement to your view of queue health.

Outcome: Prepare for future growth

Consider these backlog items/nice to haves as your client base increases.

Recommendation: Find ways to integrate Turbo more widely

The best opportunity for decreasing the latency that a customer experiences in a web-app is either to use Turbo/Hotwire or create an SPA (e.g. React). The reason for this is simple math:

1. Assume each session lasts 10 pages/navigations.
2. Each navigation takes 2 seconds.

That's 20 seconds of experienced latency. However, by re-using the page, including the CSSOM, parts of the DOM, and the JavaScript VM, we can reduce a page navigation from 2 seconds to 500 milliseconds. That's 1.5 seconds for each pageload beyond the first! That reduces experienced latency from 20 seconds to 6.5 seconds! A crazy high reduction.

There is nothing you can do on the server side that even comes close to the improvement possible by committing to Turbo or a SPA.

Since your team is small, not frontend-heavy, and the application itself would work quite well in the paradigm, Turbo/Hotwire seems the obvious choice. Remember, you don't have to get fancy here. Simply getting every page working with Turbo Drive would be 80% of the total improvement.

Cost: 3 Benefit: 5 The main cost is updating all your JS behaviors and 3rd parties.

Recommendation: Service extraction via packwerk

They say the easiest way to improve performance in an app is to add network latency between every method call. Oh wait, that's not right.

Microservices essentially can cause N+1 problems *for every method call* rather than just for data access. You have to be extremely performance-oriented to make this work.

However, I also understand the desire to "split up" a monolith as teams grow, in an effort to allow each "one-pizza team" to own a part of the app completely, and allow others to ignore that part of the app.

`packwerk` seems a great way to do this. You get to extract and enforce boundaries inside the monolith itself, without adding performance overhead. It's worked pretty well at Gusto. You can always take an "extracted" pack later and put it behind a network call in another service anyway. In most ways, extraction into a "pack" is a *necessary prerequisite* to extracting a service.

Cost: 4 Benefit: 0 No performance benefit. Big migration.

Recommendation: Sidekiq volume is low, keep it simple, avoid capsules/lambdas

Your overall Sidekiq volume is quite low, in the realm of a few jobs per second. For that reason, it's best to keep it simple: have a few SLA queues, and just autoscale or even just overprovision a bit. It won't cost much. Solutions like running in a capsule or a lambda are overcomplicating things for not enough gain. Spend a couple hundred bucks a month to greatly overprovision yourself here and not spend any dev time worrying about it.

Cost: 0 Benefit: 0 I'm basically saying "don't change anything", so it's a 0/0.

Recommendation: Pursue "each customer in own DB" infrastructures (or just pay to shard)

Business like yours tend to encounter a problem as they grow: tables grow in row count, causing lookups to get slower. However, each customer's data really is quite independent from each other (or at least parts of it can be).

Because in many ways Slope is a *network* as much as it is a "CRUD management" app, I'm not sure exactly where the seams exist here. However, even for "networks" or "marketplaces", I've seen architectures where each customer "gets their own database" work!

You can start all new customers in a "shared" environment and move them into their own database later. New Rails features coming soon will even allow you to *join records across databases*, which is wild.

And these days, "sharding as a service" vendors like Planetscale are also quite good. You can decide to just punt here and pay someone to shard for you later (this is what Gusto did, it seems to be fine).

Cost: 0 Benefit: 0

Summary

- Outcome: Create an anti-N+1 workflow
  - Recommendation: Run rack-mini-profiler in all environments. *Cost: 1 Benefit: 3*
  - Recommendation: Never use "fake" ActiveRecord associations. *Cost: 4 Benefit: 5*
  - Recommendation: Never use query methods in instance methods. *Cost: 3 Benefit: 4*
  - Recommendation: Set logs to debug in development *Cost: 0 Benefit: 1*
  - Recommendation: Use prosopite, remove bullet. *Cost: 2 Benefit: 3*
  - Recommendation: Invest in a solid Sentry workflow *Cost: 2 Benefit: 3*

  - Recommendation: Set performance requirements *Cost: 1 Benefit: 2*
  - Recommendation: Stop caching *Cost: 1 Benefit: 1*
  - Recommendation: Access every table one time. *Cost: 2 Benefit: 2*
  - Recommendation: Treat ActiveRecord::QueryCache as a "mini" N+1. *Cost: 2 Benefit: 1*
  - Recommendation: Build a better development seed *Cost: 5 Benefit: 3*
- Outcome: Lower p95 response times
  - Recommendation: InventoriesController#create, ItemsController#edit_multiple, ItemsController#create: Background the alert updating callback *Cost: 1 Benefit: 3*
  - Recommendation: SitesController#clinical_trials: N+1 on Alert *Cost: 2 Benefit: 3*
  - Recommendation: Limit or consolidate link calls in jump_to *Cost: 1 Benefit: 2*
  - Recommendation: Fix Workspace#inventories_with_configurations *Cost: 2 Benefit: 3*
  - Recommendation: WorkspacesController#shipments N+1s *Cost: 1 Benefit: 1*
  - Recommendation: Reduce overhead of ShipmentPolicy#scoped_shipment_role on API::V1::SearchController#api_search *Cost: 3 Benefit: 2*
  - Recommendation: InventoriesController#show: Count collections by loading and partitioning, or GROUPing *Cost: 2 Benefit: 1*
- Outcome: Lower p50 response times
  - Recommendation: Fix alerts lookups, both in speed and N+1 *Cost: 1 Benefit: 1*
  - Recommendation: Overhaul authorization *Cost: 3 Benefit: 3*
  - Recommendation: Log jobs upon completion, not enqueueing. *Cost: 2 Benefit: 1*
- Outcome: Remove deployment footguns
  - Recommendation: Install jemalloc. *Cost: 0 Benefit: 1*
  - Recommendation: Don't use $redis, use a connection pool. Remove "cache" usage from $redis. *Cost: 0 Benefit: 0*
  - Recommendation: Instrument request queue time *Cost: 1 Benefit: 1*
  - Recommendation: Any job over 1 minute should be broken into chunks *Cost: 2 Benefit: 1*
- Outcome: Add robustness to background jobs
  - Recommendation: Set up Sentry Queues when supported *Cost: 0 Benefit: 2*
  - Recommendation: Debounce Searchkick reindexing *Cost: 1 Benefit: 1*
  - Recommendation: Use SLA queues *Cost: 3 Benefit: 1*
- Outcome: Prepare for future growth
  - Recommendation: Find ways to integrate Turbo more widely *Cost: 3 Benefit: 5*
  - Recommendation: Pursue "each customer in own DB" infrastruc-

tures (or just pay to shard) *Cost: 0 Benefit: 0*
- Recommendation: Sidekiq volume is low, keep it simple, avoid capsules/lambdas *Cost: 0 Benefit: 0*
- Recommendation: Service extraction via packwerk *Cost: 4 Benefit: 0*