Speedshop Tune: Mitchells

@nateberkopec

Prepared with care for Mitchells Stores

@nateberkopec

3/1/2025

Contents

- Outcome: Reduce TTFB on Products and Taxons
  - Recommendation: Fix the Cache multi-get on Products#show
  - Recommendation: Remove (or batch?) ImageType#exists? and corresponding HTTP/HEAD requests.
  - Recommendation: Fix Role#exists? "N+1"
  - Recommendation: Fix ContentOption/*OptionMatch N+1 for Taxon#show
    * References/Bibliography
    * Summary

Hello Mitchells!

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.

Summary

You told me that your mission was to **improve your Web Vitals results**. The three web vitals are:

- **Largest Contentful Paint (LCP)**: The time it takes for the largest content element (almost always an image, but sometimes a text block) to become visible in the viewport.
- **Interaction to Next Paint (INP)**: Measures how quickly your page responds to all user interactions - clicks, taps, and keyboard presses - by monitoring the time from the interaction until the next frame is painted on screen.
- **Cumulative Layout Shift (CLS)**: A measure of how much movement of page content occurs during page load.

You currently don't have any issues with INP, so I'm ignoring it in the is report. That is not unexepcted, because typically only extremely heavy SPA-style apps have problems with INP. Your Javascript usage is quite light overall, so it's not surprising that your data here shows that this web vital is doing perfectly fine.

With the two remaining vitals, I found that:

- **CLS has some minor but specific bugs that need to be fixed**. You have a few specific issues, mostly around image heights, that lead to some layout shifting. You can probably fix the specific issues in a day or two, and install a more permanent fix to prevent regression here in about a week.
- **LCP has a more wide and varied set of issues, but can probably still be mostly fixed on a short timeline**. The highest impact fixes mostly revolve around marking images as either "important" or "not important", which, much like your CLS stuff, can be done quickly and

manually for important pages like the homepage, and then fixed on a more permanent basis for dynamic pages like taxons and products.

- **There are a number of other web performance things that can be addressed on longer horizons**. Some of these are pretty big lifts but with big payoffs, like moving your Javascript to async. Others are more medium-sized, like paring down your webfont usage and serving them firstparty instead of from Typekit.

To immediately and quickly get LCP and CLS back into the "green" zone, my action plan would be to pluck the following recommendations out of this report:

1. Mark LCP image with fetchpriority=high on the homepage, product and taxon pages.
2. On the homepage, manually mark all other images with loading=lazy.
3. Manually add image heights on the homepage
4. On mobile product pages, add image heights for the image display.

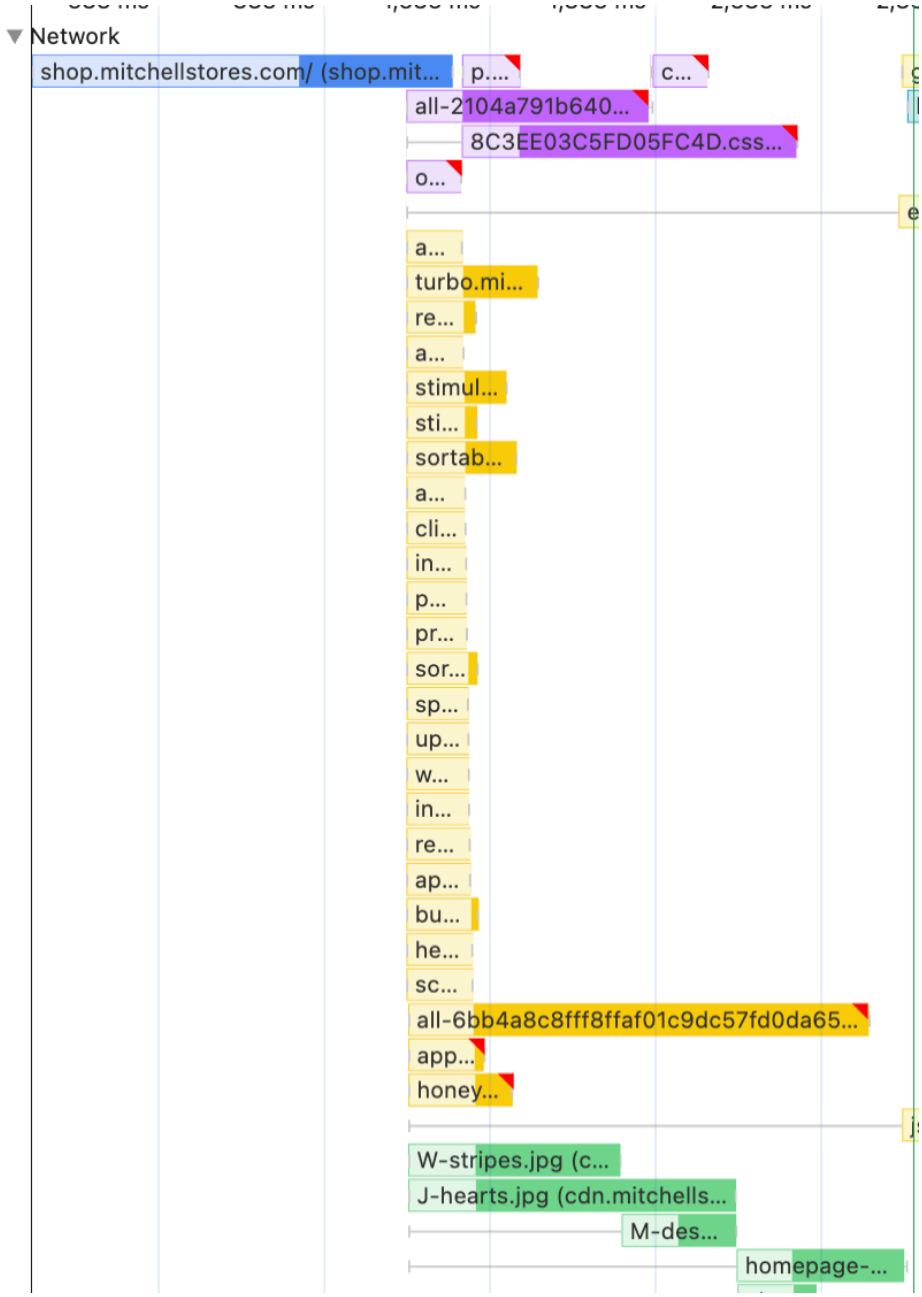
This workstream would probably take one person less than a week. I think it would address ~80% of the problem, though admittedly in a very brittle way (manually adding image heights means they're no longer valid when the image changes, for exampe). However, you can use that as a starting point to "backfill" in a more long-term solution with the other recommendations here.

How To Use This Document

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.Ratings are designed mostly to be relative to each other, i.e. a 5 is always harder or more valuable than a 4, etc. Even cost/benefit roughly means I think it's a toss-up whether or not you should do it, while a cost higher than the benefit rating means I think it's not worth doing in the near term future.

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: Reduce p75 LCP to 2.5 seconds for mobile devices, improve Lighthouse Score from 25 to 75

In preparing this report, I used the CruX visualizer from Google. It's important to know that Web Vitals are collected from user telemetry in Chrome browsers, and the data is published in the open for all URLs indexed by Google. While having an in-house Lighthouse audit like you do is nice, it's not reflective of the final signal actually used by Google.

Looking through the CruX visualizer data for shop.mitchellstores.com, we can get some important information:

1. Desktop LCP is fine (2.2 seconds), but mobile LCP is very high (3.7 seconds average for the origin). So, we have more work to do for mobile devices.
2. Mobile LCP is particularly bad on the homepage, where it can be 5 seconds or more. Other pages, such as the men's taxon or product pages (where CRuX data is available) are not particularly bad.
3. ~50% of origin traffic is mobile, but 70% of traffic to the homepage is on mobile. Are mobile users abandoning the site when they see the homepage is slow to load?

Getting **mobile LCP to below Google's 'green' threshold of 2.5 seconds p75** is therefore our goal. Our improvements here will inevitably also have some impact on the desktop too, but since that's already "green" we don't have to focus on it specifically.

One more point on web vitals that will become important: you'll notice all the CRuX data can be viewed for a *specific URL* or for *an entire origin (domain)*. For search rankings, the *specific URL* data is what matters. If there is not enough URL-specific data, the origin is used instead (you'll note for some URLs, like say a specific product page, the 'url' tab disappears in CRuX visualizer).

Recommendation: (Manually) mark LCP images with fetchpriority=high

Mitchells is basically a catalog site, so it's not surprising that a majority of its performance issues revolve around images. There are so many images! The homepage loads 63 images, the men's taxon page loads over 90, and the individual product page loads 75.

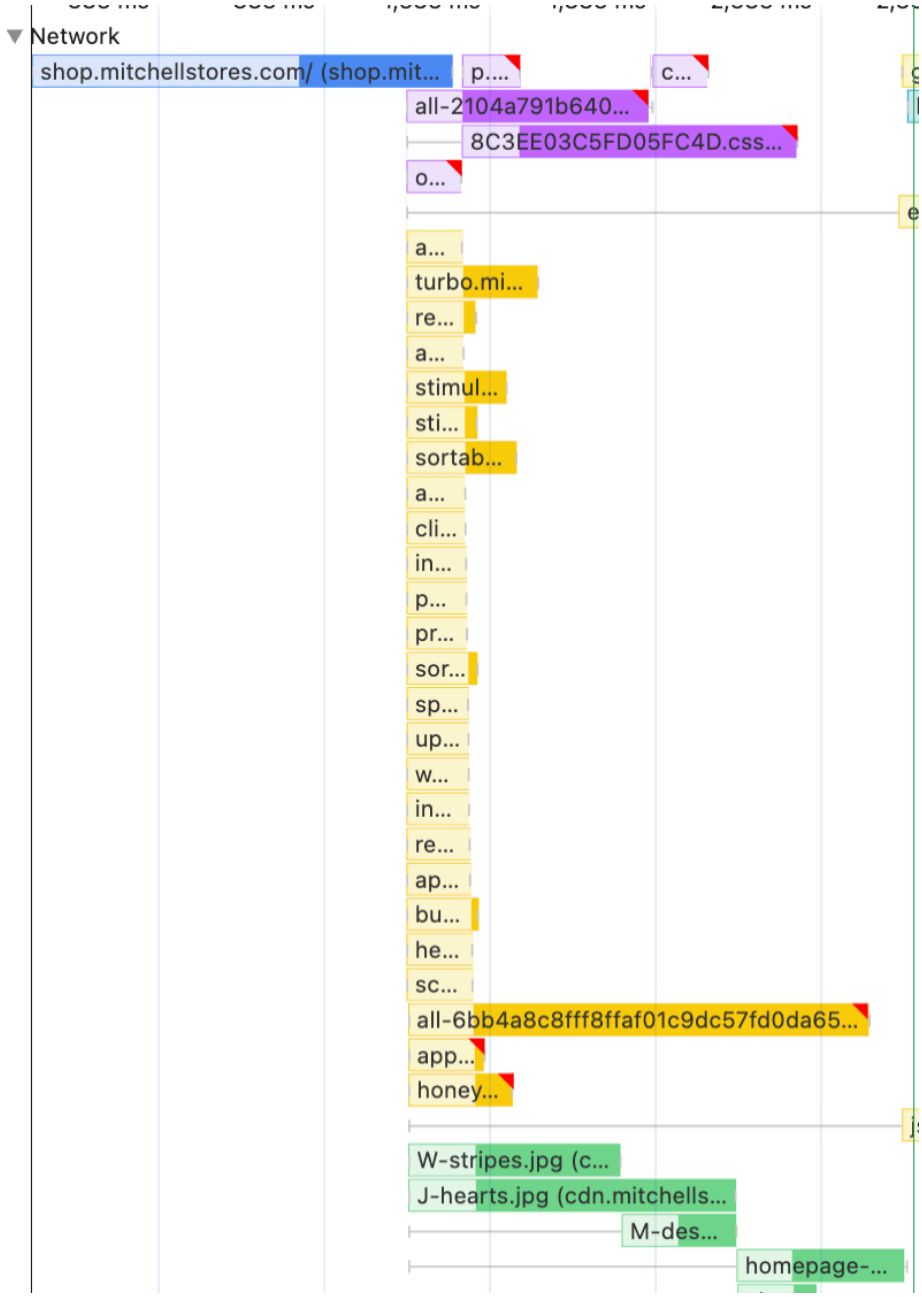
LCP is primarily an image-driven metric, because the LCP element is almost always an image.

For debugging LCP issues, my workflow is usually:

1. Open DevTools. Click the "responsive" icon and change the view to a

phone, e.g. the iPhone 14 Pro Max. Under the network tab, I have a custom preset that limits bandwidth to 20 mbps. Select that, and select "disable cache". Generally, I will adjust my network bandwidth preset *downward* until I can replicate the numbers I'm seeing in CrUX. The "Fast 4G" preset got me close enough.
2. Go to the performance tab. Click the gear icon, and select a 4x CPU slowdown. This will help simulate mobile devices. Hit CMD-SHIFT-E to trigger a full page reload.
3. In the Performance timeline, look for the LCP timing. You can click it and see which element is the LCP element. For everywhere I looked at Mitchells, it's an image.
4. Look at the image's network request and work backwards. Why didn't it finish earlier?

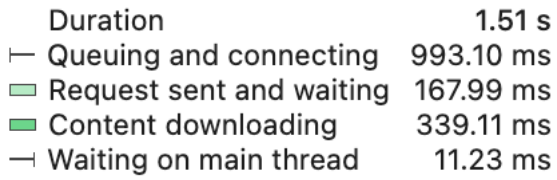
Doing this on the homepage, you'll see the LCP image is, helpfully, called `homepage-hero`. However, it starts downloading rather late:



The "whisker" of the boxplot here notes when this image was first discovered in the document and when it was "queued" to be downloaded. When the green "box" starts is where we actually made the request. Light green is time spent waiting for the first byte, and dark green is the time spent actually downloading

it.

You can also click the network request for a more detailed breakdown:



So, **why are we spending 1 second queueing for this image?**.

You can also see in the network information that this image's priority was `medium`.

HTTP/2 is based on the principle of one connection per origin. Therefore, we might have several resources we want to load at the same time, so we need to prioritize their ordering as to what we get first.

HTTP/2 implements a priority system where resources (like images, scripts, CSS) can be assigned different priority levels (high, medium, low, or idle). When a browser makes multiple requests, the server uses these priorities to determine which resources to send first.

This *particular* image fetch tends to be "in-flight" while these requests are also in flight:

1. A `high` priority request for all.js (360kb)
2. 2 or 3 other image requests, also medium, but for some reason it consistently prefers downloading W-stripes.jpg, J-hearts.jpg and 1 or 2 other random images first.
3. A handful of other JS requests.
4. A handful of `high` priority CSS requests.

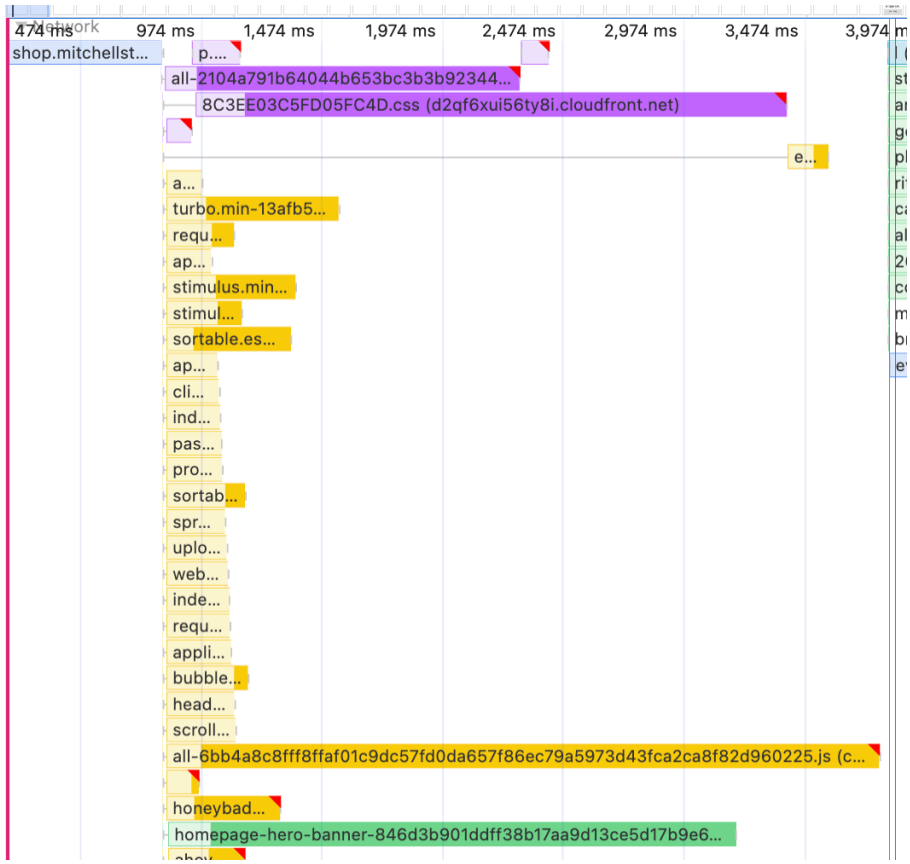
That's a *lot* going on and competing for bandwidth. We'll address how to move some of this other stuff around, but the easiest thing we can do is simply to tell the browser that the LCP image is important!

The `fetchpriority` attribute on an HTTP element helps us do this.

I routinely use DevTools overrides to test the effect of simple HTML changes like this one.

On my test setup using a 4x CPU slowdown and a network preset of 3 mbps, my baseline is an LCP of about 4.5 seconds.

Adding fetchpriority=high to the hero element along decreased LCP by about 0.75 seconds. Look how it removed all of the time spent queueing:



This network graph makes the next steps pretty obvious: the image download is taking too long now! How can we address this?

Cost: 0, Benefit: 4 A solid LCP bump with a ~15 character change. I've already opened the PR.

Recommendation: Pare down Gotham and Didot styles "A & B" to "A". Remove Didot?

We now see that there's a few requests happening in parallel with our image request. One is for fonts. I audited what's in the font file and what's in the page and noticed a few things.

I had Claude write me a script to check where these fonts were actually used. I'll reproduce that script here just so you can use the same thing:

```
// Find all elements using any Gotham or Didot font variant
const fontPatterns = ["Gotham", "Didot"];
const elementsWithFont = [];
```

```
// Check all elements on the page
Array.from(document.querySelectorAll("*")).forEach((el) => {
  const style = window.getComputedStyle(el);
  const fontFamily = style.fontFamily;

  // Check if the font family matches any of our patterns
  for (const pattern of fontPatterns) {
    if (fontFamily.includes(pattern)) {
      elementsWithFont.push({
        element: el,
        fontFamily: fontFamily,
        text:
          el.textContent.trim().substring(0, 50) +
          (el.textContent.length > 50 ? "..." : ""),
      });
      break; // Stop checking patterns once we find a match
    }
  }
});

// Group results by font family
const groupedByFont = {};
elementsWithFont.forEach((item) => {
  if (!groupedByFont[item.fontFamily]) {
    groupedByFont[item.fontFamily] = [];
  }
  groupedByFont[item.fontFamily].push(item);
});

// Log results
if (Object.keys(groupedByFont).length === 0) {
  console.log("No elements using Gotham or Didot font families found");
} else {
  console.log("Found elements using Gotham or Didot font families:");

  for (const [font, elements] of Object.entries(groupedByFont)) {
    console.log(`\n Font Family: "${font}" (${elements.length} elements)`);
    console.log("Sample elements:");
    elements.slice(0, 5).forEach((item, index) => {
      console.log(
        `  ${index + 1}. ${item.element.tagName.toLowerCase()}${
          item.element.id ? "#" + item.element.id : ""
        } - "${item.text}"`
      );
    });
  }
```

```
}
```

The results for the homepage were that most elements used a font-family of `"Gotham A", "Gotham B"`. This is weird, because we're specifying this "Gotham B" fallback but it's contained in the same file as Gotham A. I'm not sure under what circumstances, if any, Gotham B could possibly be used. **I think you should remove the B variation** from all fonts. I'm struggling to see how doubling font download size could have *any* possibly good payoff.

In addition, we notice **Didot is not used**. I only see it in a few places in the codebase. **You should remove it from the default font bundle and only include it on those pages**, or put it in a different CSS file with a lower fetchpriority.

Combining these recommendations should reduce the CSS for fonts on the homepage by 75%, or about 200kb.

As an experiment, I like to block URLS in DevTools to see what the best possible impact of "fixing" the issue for perf might be. Blocking the Didot/Gotham CSS improved LCP for me by an additional 500ms over the previous recommendation.

Cost: 2, Benefit: 3 Removing Didot might be a bit of a pain, since it's used on some random static pages I think in a few spots. This will probably only increase LCP by a bit, as you're still going to spend a lot of bandwidth downloading all.js.

Recommendation: Change CDN to Cloudflare or something else that will convert to WebP easily

The next easiest thing we can do is to make the image itself smaller. Today, the hero image is about 200kB. However, it's a JPEG.

You're using Cloudfront as your CDN. More full-featured CDNs, such as Cloudflare or Cloudinary, will do some basic image optimization for you. You can also do this with AWS Cloudfront, but it's going to require some fancy DIY stuff, like a Lambda to convert images on the fly. It's *far* easier to just pay someone to do this for you. I personally use Cloudflare here, as it will just convert everything to WebP where possible.

Locally, I ran `cwebp -q 80 input.jpg -o output.webp` to try converting the hero image to a WebP format image with a quality factor of 80. It reduced the size to 137kb, a ~25% savings. 60kB may seem small, but your entire CSS is about 90kB on the wire. So, we could try to eliminate 66% of your CSS, or just figure out how to serve next-gen image formats. That's going to be a lot easier.

To test the impact of this change, I did a local override with a ~140kb image I found on a different area of the site. I was not able to create a measurable LCP difference.

Cost: 2, Benefit: 3: 60kB is honest work. Changing your CDN to something that converts to webP on-the-fly will improve image loads for *everything* across the site *all the time*, which should have a benefit even if the benefit for the homepage header is dubious. This suggests the "bandwidth stealing" effect of all.js and all.css is higher than the effect of freeing up bandwidth with a smaller hero.

Recommendation: Split all.css into 3 to 5 smaller files.

`all.css` gets the special `highest` fetch priority, because it's render-blocking CSS. This hampers our efforts to improve the download speed of the hero, because any extra bandwidth we free up gets "claimed" by the higher fetch priority CSS.

This makes reducing the size of our CSS download a high priority. It's currently about 90 kB.

In DevTools, if you hit CMD-SHIFT-P and then type "coverage", you'll see an option for "start instrumenting coverage and reload page". This is essentially a more detailed version of the analysis Lighthouse can do, and will show for each JS and CSS file how much of it you actually used. The Sources view will now always have red lines for lines of files which are not used.

For me, the homepage shows only **7.3%** usage. So, 92.7% of your CSS is downloaded but not used.

Unfortunately the process here is extremely manual. Your pages are dynamic (so we can't just run a CSS purge step in the build), and there's of course tradeoffs between loading CSS beforehand versus only-as-needed.

I recommend aiming to split all.css into 3-5 files:

1. homepage.css
2. products.css
3. all.css, which more or less exists as it does today.

For the homepage and for product pages, remove all.css and replace it with a specific CSS file, aiming for more like 50%+ usage.

You can create more "specific" CSS views for endpoints which you feel are worth the squeeze re: how much traffic they get.

Cost: 3, Benefit: 3: It's probably the most valuable ~45 kB you can remove from the products and homepage loads.

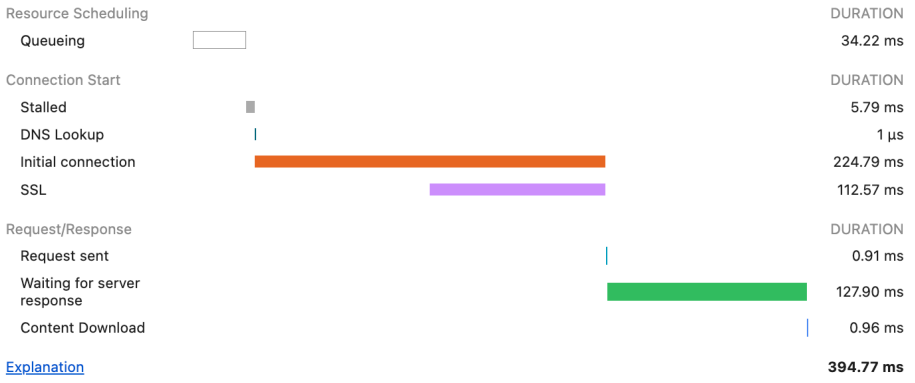
Recommendation: Move cdn.mitchellstores.com to shop.mitchellstores.com

You currently serve your Rails app and static assets from different domains. This makes your site slightly slower to load because we have to establish a new

connection to the CDN domain.

You can see this if you flush all socket pools and then reload the page in DevTools. To flush sockets, visit chrome://net-internals and click "sockets", then click "close" and then "flush". For some reason I always have to hit it multiple times.

Then, reload the page in DevTools.



Moving cdn.mitchellstores.com to www.mitchellstores.com/assets would remove this step completely, reducing LCP by 1 to 2 network round-trips for first time users. This should knock another 1 to 300 milliseconds off your LCP, particularly on mobile where network latency can be pretty high on cell networks.

Cost: 3, Benefit: 3

Recommendation: Everything on the homepage which isn't the hero should be loading=lazy

Looking at the network diagram, once you add `fetchpriority=high` to the hero image, the page load gets split into two phases:

1. We download the CSS, JSS and hero image in parallel.
2. Once all those three steps are done, every other image downloads.

The reason is because everything in step 1 has a fetchpriority of high or highest, and everything in step 2 is medium or lower. So, we don't *really* need to optimize step 2 further, because it happens after LCP.

However, we can still do additional things here to more accurately reflect the priority of various images, even if I don't think it will have a direct effect on LCP. It's sort of like a "backstop", a second technique that also helps ensure an optimal pageload. Pageloads aren't deterministic (because network conditions aren't) and we can only really do detailed tests in Chrome, so multiple techniques helps make sure our vision is actually achieved.

Since the homepage is relatively static, we can manually add the `loading` attribute to any image which is not the hero set to a value of `lazy`. This has the effect of *not even starting the image download* unless the image is visible. This frees up bandwidth for other stuff.

Cost: 1, Benefit: 1 A 15-minute PR for the homepage.

Recommendation: Manually walk through templates and mark images as loading=lazy.

Since the homepage is so important, I think it's fine to go in and hand-optimize that. However, your other dynamic layouts are perhaps a bit of a bigger lift.

Google doesn't recommend putting `loading=lazy` on *every* image because you might put it on an LCP image by accident. That's good advice. However, 90% of the images you put on a page, you know they're never going to be the LCP image.

Using "pages which receive lots of organic search traffic" as a guide, I would burn down the images on these pages and mark as many as loading=lazy as I could. You might find a way to come up with abstractions here to make this easier.

Cost: 1, Benefit: 1 Just spend a few hours on it and move on.

Outcome: Reduce CLS to 0.10 on all devices

This is the second web vital you're not doing so hot on.

Returning to the CrUX data for a moment, CLS is perfectly fine on the Origin level for desktop viewers. On the homepage, however, it's above 0.1 even for those desktop-ers.

For mobile users, CLS on the homepage is actually quite good, although it's high (0.18) for the origin. Pages such as the taxon or popular product pages show high CLS.

Recommendation: Homepage: manually add image heights

The homepage contributes a large amount to the origin score due to accounting for 10-20% of total visits, so it's worth special attention. It may seem strange that CLS is so low for mobile but high for desktop here, which is the reverse for the rest of the origin, but it actually makes perfect sense.

On desktop, the following load order occurs:

1. CSS and JS download simultaneously, and rather quickly, due to a high-quality network connection.

2. As soon as the JS and CSS are downloaded, we layout the page for the first time. This happens before the images have downloaded at all.
3. The images download and get added to the layout. Because the images do not have pre-computed heights, they cause layouts shifts when added in.

On mobile, you instead get:

1. CSS and JS download simultaneously. However, they're pretty slow. In addition, the JS has to actually execute, which, on mobile processors, takes a long amount of time.
2. Because of the lengthened step 1 plus a smaller viewport, there's less opportunity for late-loading images to shift the layout around.

On the homepage specifically, you probably only need to added a dozen or so heights to various images to eliminate the layout shifts.

Cost: 1, Benefit: 2: Slightly low benefit here because improving CLS on the desktop isn't really necessary, but this will take about an hour.

Recommendation: On taxon pages (mens/womens), use aspect-ratio to specify image height

You have several full-width and 25-percent-width images here, which, when they load in, cause a layout shift. I would add an ID here and specify the aspect ratio of the image that's going to load in:

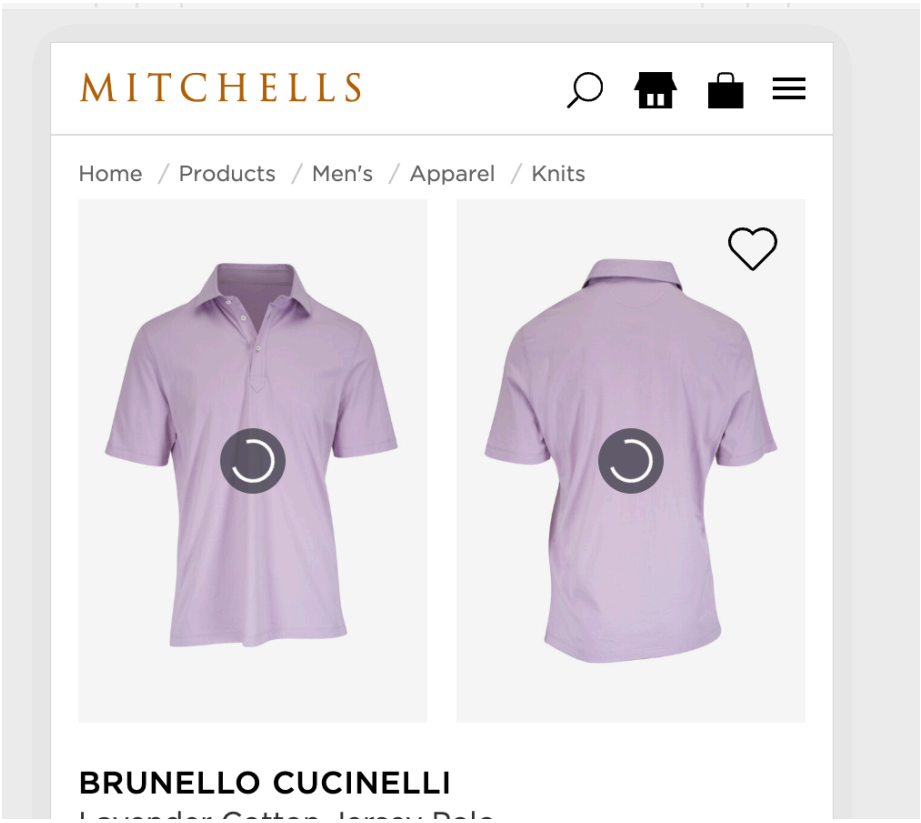
```
[data-title="12215::mitchells-welcomes-a-new-store"] {
  aspect-ratio: 16 / 9;
  object-fit: cover;
}
```

The width is calculated from the CSS, and since it knows the aspect ratio, it can also calculate the final display height. You'll have to do this manually for marketing images *or* write a new image tag helper and store image heights and widths so you can insert that into markup dynamically.

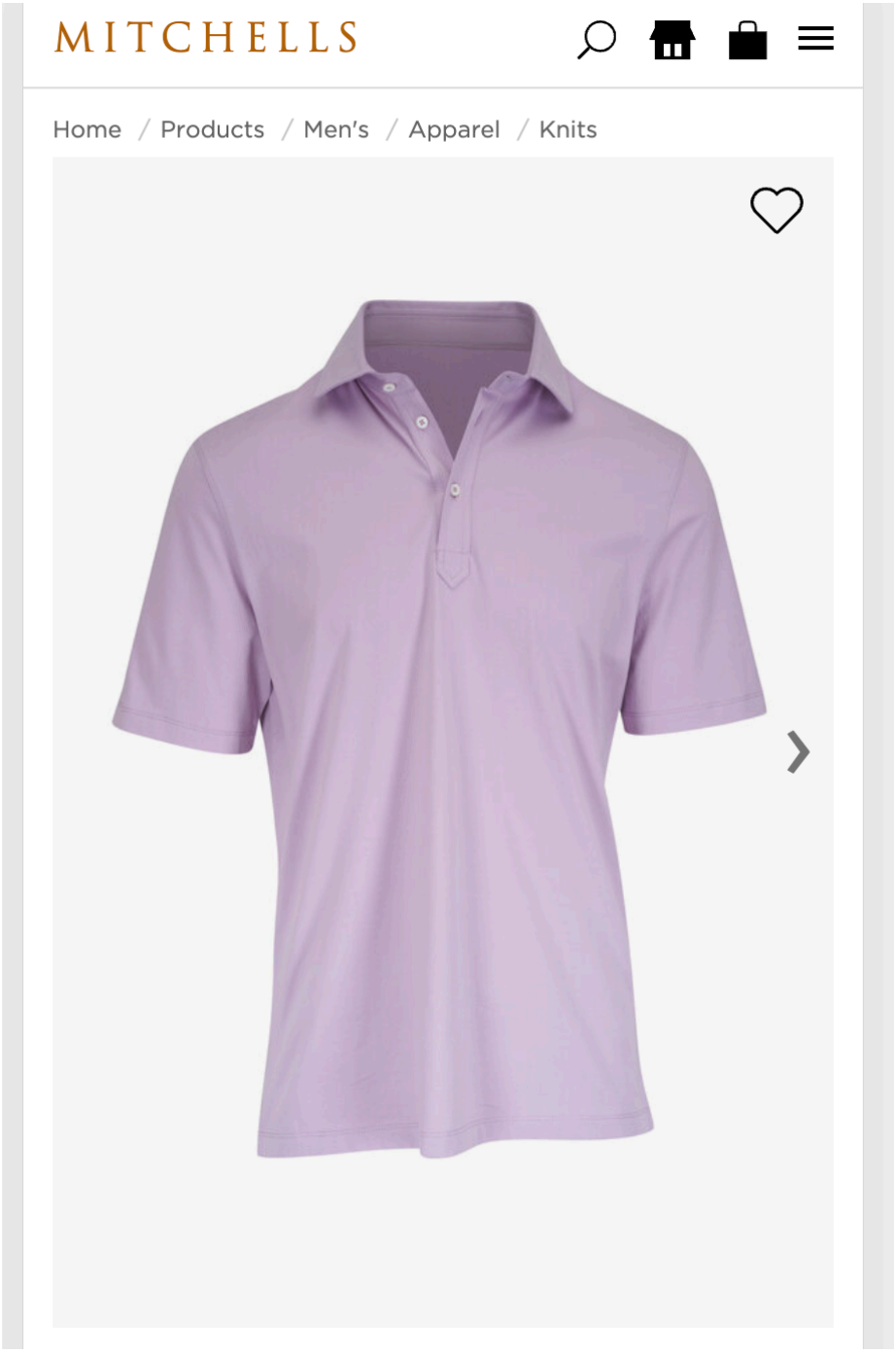
Cost: 2, Benefit: 3 Taxon pages account for a lot of views, so this will need to be addressed.

Recommendation: On product pages, fix massive pop when high-res image gets loaded in.

On product pages on mobile, you see a huge CLS pop when the "large" product image gets loaded in. I see two steps in the page load:


MITCHELLS
Home / Products / Men's / Apparel / Knits
BRUNELLO CUCINELLI

... and then:



This massive two-step pop means a huge layout shift event. This is on an a

very important layout, so I think it's probably the most import CLS bug on the whole site.

You know the aspect ratio of this image, so I think just adding an `aspect-ratio` property here will fix this issue.

Cost: 2, Benefit: 4 Fix this one and I think you'll greatly reduce CLS on the origin.

Recommendation: Enforce image heights through code, burndown.

All of your CLS bugs boil down to the lack of known image heights, either through hard-coding the height property or through an aspect-ratio property.

It's very difficult to predict what images are important to fix here *in advance*, because whether or not an image causes a layout shift depends on what the parent container element looks like, what order the image appears in the document, and network conditions.

For that reason, I prefer logging this sort of event from real users, and then burning down problems as reported from real users.

You already have New Relic in your stack, so if you pay for Browser Pro, you can get all warning level logs. That means you can log layout shifts as they occur and also what caused them. To identify the image, I would probably use the image alt or the id/class of the parent container, rather than the URL (which is a bunch of ActiveStorage gobbledygook).

Logging layout shifts as they occur is an exercise easily tackled by your favorite intern Claude, but basically all you do is use the `LayoutShift` API to subscribe to these events.. Once you get a shift, log important debug information and NewRelic will pick it up. Aggregate it in NewRelic and "burn down" the list.

Cost: 3, Benefit: 4 A long term solution for finding and fixing specific CLS issues.

Outcome: Improve FCP to 1.8 seconds on all devices

First Contentful Paint is an important performance event, even if it's not directly a Web Vital. It contributes greatly to the perception of performance (by loading *anything* earlier we perceive the load to be faster) and is a necessary precursor step to LCP (we have to paint ANYTHING before we can paint SOMETHING!).

It's also a big contributor to your Lighthouse score, which is an indirect goal for us.

Recommendation: Async all Javascript

This is the highest-cost, highest-benefit recommendation in the entire report.

Currently, your JavaScript is loaded synchronously, which means that when the browser encounters a script tag, it stops parsing the HTML until the script is downloaded and executed. This blocks the render of the page.

The `async` attribute on script tags tells the browser to download the script in parallel with HTML parsing. When the script finishes downloading, HTML parsing is paused again while the script executes. This means:

1. Scripts load in parallel with other resources
2. Scripts execute as soon as they finish downloading
3. Scripts may execute in any order
4. Scripts **don't block page rendering while downloading**

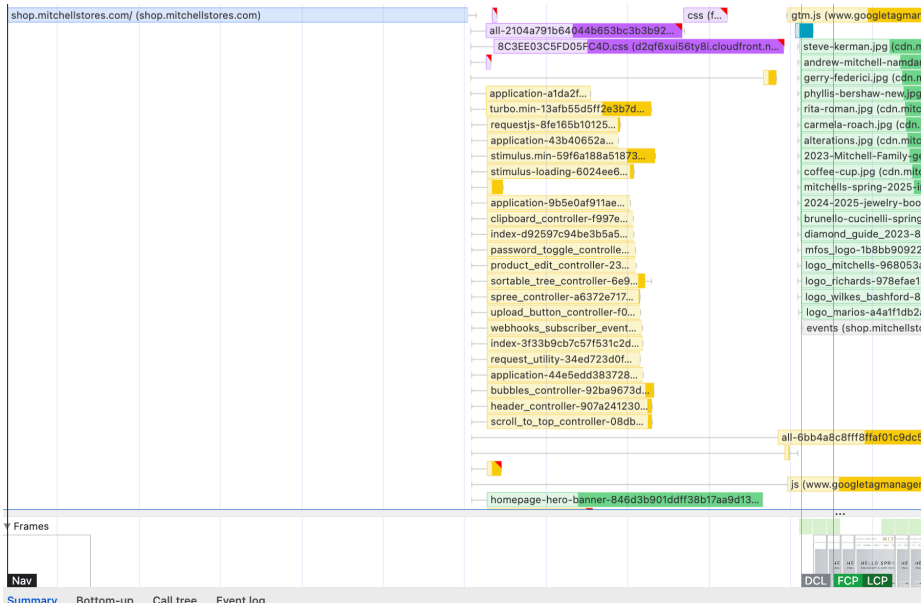
This would dramatically improve FCP because the browser wouldn't have to wait for script downloads before starting to render the page.

Your app uses jQuery and some other libraries that expect synchronous loading, so this would require some careful refactoring to make sure scripts execute in the correct order. You have several inline script tags littered around the page. That doesn't work in an `async` world. You'd need to:

1. Move all inline scripts to external files
2. Add dependency management (like ES modules)
3. Test thoroughly across all pages

However, there's one reason to have hope: if you `async` your Javascript today (via a DevTools override), the page doesn't break, and still renders correctly visually. That means you're using Javascript in a limited way to install certain behaviors, rather than render big chunks of the page. That means `async`-ing JS is possible!

With `async`, combined with the other recommendations in this report, **I was able to get LCP (and FCP) down to 1 second**. Note the waterfall here, and how all.js loads after the hero:



The biggest thing will be moving inlined script tags out into external scripts. There are probably 2-dozen plus behaviors installed on the homepage as inlined script tags.

Cost: 4, Benefit: 5 It's basically a re-architecture of your entire frontend approach, but it would be worth several seconds of FCP and LCP.

Recommendation: First-party all fonts

Third-party serving of fonts has a number of disadvantages:

1. We have to connect to a new origin. See previously how that costs us a couple hundred milliseconds.
2. It helps the browser to multiplex bandwidth more efficiently when it's across one connection on one domain.
3. We can subset the font.

I used to be pretty "high" on 3rd-partying fonts, and particularly I think Google Fonts is still quite good (since it will even subset for you), but typekit is pretty bad. It requires a new origin or two, and a couple of redirects, in order to just get the fonts downloaded. It's a slow design. You can simply buy Gotham and Didot for a couple hundred bucks and serve this yourself from your own domain, which will move up font rendering by a couple hundred milliseconds.

Cost: 3, Benefit: 2 Probably a pain to get off of a third party here, the benefit is admittedly limited.

Recommendation: Async Apple Pay, Ahoy and Honeybadger tags

You have three third party script tags that are blocking render, which could be async'd instead. They are very small and so have limited impact on page render (certainly, basically zero when all.js is considered!).

Apple Pay

This is just a normal script tag. You're not using any Apple APIs in internal script tags, and this appears *after* all.js, so I think this is just to get the Apple Pay button to show up? If so, just adding an `async` attribute here should Just Work.

Honeybadger

You define your `beforeNotify` hook as an internal script tag - simply change that around a bit to call only after the script is loaded:

```
<script src="//js.honeybadger.io/v6.10/honeybadger.min.js" async onload="initHoneybadger()">
<script>
  function initHoneybadger() {
    Honeybadger.configure({
      // ...
    });

    Honeybadger.beforeNotify((notice) => {
      // ...
    });
  }
</script>
```

Ahoy

This one is harder because there are numerous inline script tags in the document which expect Ahoy to be available, so async-ing this is blocked on removing those and moving to external async scripts first.

Cost: 3, Benefit: 1: The scripts here are so small compared to the tasks before you with all.js that it's probably not worth considering at the moment.

Outcome: Reduce TTFB on Products and Taxons

As a less important goal, you should try to reduce the TTFB (and, therefore, the response time) of the Taxons and Products show actions. The p95 on these is sometimes high (>500ms), so we could save a few hundred ms on all our

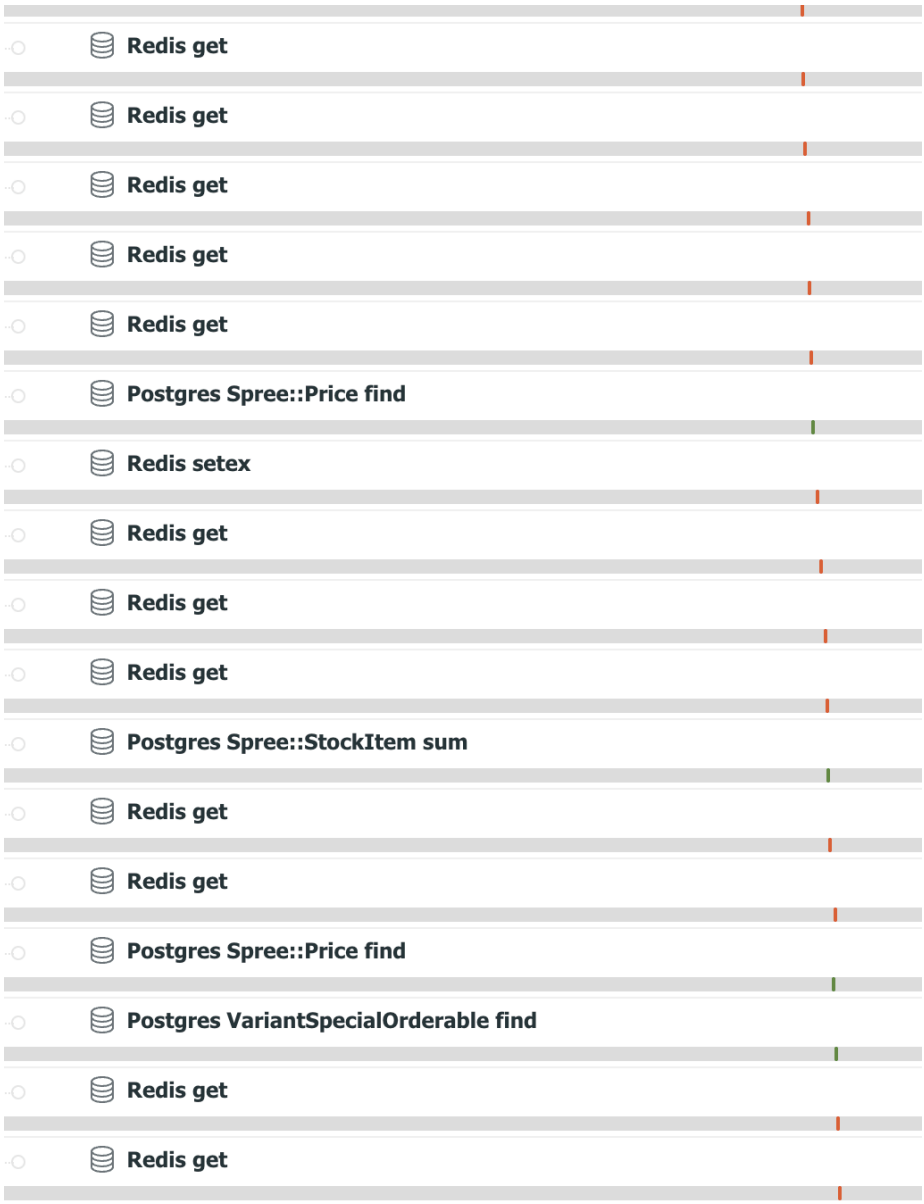
various metrics by addressing it through reduced response time. There's a few easy wins here.

Recommendation: Fix the Cache multi-get on Products#show

For NewRelic, my process is basically:

- Go to the transactions tab. Go the the "full view" list, then sort by time consumed and look at the p95 for each as I go down the list. If a controller action has more than 1% of total time, and a p95 over 500ms, it's a problem and we can fix it.
- Go to the traces tab on the left. Look for the controller again. Click it, then look for recent traces whose duration is *close* to the p95.
- Look at 3 to 5 traces, try to find patterns for why traces are this slow.

In this case, ProductsController#show meets the first criteria, and looking at traces showed a clear pattern:



...way too many cache round-trips. For reference, this was `/products/1421776-valentino-garavani-belts`.

The solution here is to wrap these up as multi-gets. Rails cache partials can make this incredibly easy:

```
# In your view
<%= render(partial: "product", collection: @products, cached: true) %>
```

This will generate a single multi-get request to your cache store instead of indi-

vidual gets for each product.

I find this is a lot easier than trying to manage your own multi-fetches.

To find the exact location of this redis cache get, I would use `rack-mini-profiler` in production (which you already have set up, hoo-ra!)

Cost: 2, Benefit: 3: Would reduce p95 by a lot on this page.

Recommendation: Remove (or batch?) ImageType#exists? and corresponding HTTP/HEAD requests.

Also in `ProductsController#show` requests, I noticed a slow loop which ends up making lots of HEAD requests to S3 checking for the existence of a file:

This strikes me as a bit weird, because I'm struggling to imagine the failure case. Are files frequently *not* present on S3 when they should be? I think

this is probably the result of naively checking for the presence of an image or attachement without realizing that it costs 10-15 ms to do so.

Use `rack-mini-profiler` and `flamegraph` mode to find and nuke the sources of these requests. You don't need to be making requests to S3 in-line during a web request.

Cost: 1, Benefit: 2. There aren't that many of these (it's per product image I assume) but the fix is probably straightforward.

Recommendation: Fix Role#exists? "N+1"

At the beginning of almost every trace, there are 5 to 10 Spree::Role#exists? checks in a row. This is the result of doing something like this:

```
if user.is_admin?
# ...
elsif user.is_manager?
# ...
elsif #...
```

This is a *super common* problem in user authorization. The thing is that almost every user will have a very small set of `spree_role_users`, so making a query to check just for the *existence* of a single one is a bit of a waste when you could have loaded *all roles* into memory.

The core reason is the implementation of `has_spree_role?`

```
def has_spree_role?(role_name)
  spree_roles.exists?(name: role_name)
end
```

… which should, in my opinion, be:

```
def has_spree_role?(role_name)
  spree_roles.any? { |role| role.name == role_name }
end
```

…this forces the load of the spree_roles association completely, and subsequent requests will use that memoized association instead.

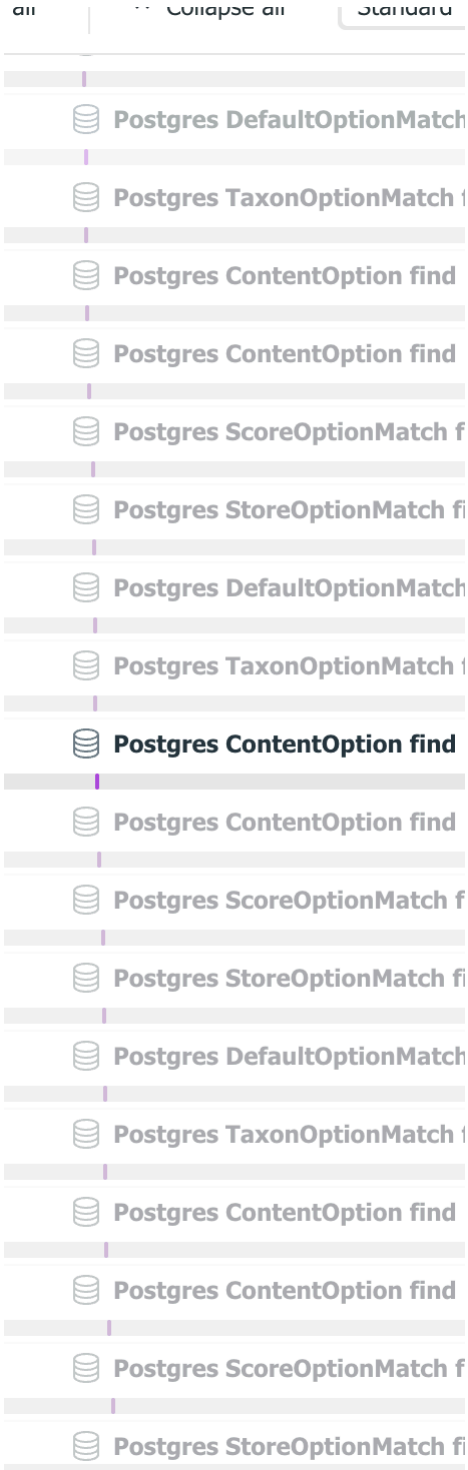
Again, this is a benefit because:

1. This data is almost always queried more than once per request.
2. Users have a limited number of spree_roles, and it costs about the same to load 1 as it does 20.

Cost: 1, Benefit: 1 Excessive SQL added on every request is a pet peeve of mine.

Recommendation: Fix ContentOption/*OptionMatch N+1 for Taxon#show

Taxon traces have the following funny N+1 pattern:


Collapse all
Standard
Postgres DefaultOptionMatch
Postgres TaxonOptionMatch
Postgres ContentOption find
Postgres ContentOption find
Postgres ScoreOptionMatch f
Postgres StoreOptionMatch fi
Postgres DefaultOptionMatch
Postgres TaxonOptionMatch
Postgres ContentOption find
Postgres ContentOption find
Postgres ScoreOptionMatch f
Postgres StoreOptionMatch fi
Postgres DefaultOptionMatch
Postgres TaxonOptionMatch
Postgres ContentOption find
Postgres ContentOption find
Postgres ScoreOptionMatch f
Postgres StoreOptionMatch fi

The queries here are very simple, so this is almost certainly fixable with `preload`, at least the `ScoreOptionMatch` and `DefaultOptionMatch`.

The queries for ContentOption are a little more interesting and instructive:

```
SELECT COUNT(*) FROM (SELECT DISTINCT "content_options".* FROM "content_options" LEFT OUTER
```

…followed by

```
SELECT DISTINCT "content_options".* FROM "content_options" LEFT OUTER JOIN default_option_ma
```

So, that tells me that you're unnecessarily *counting* before just loading the entire relation. Remove the count query and just count the records returned by the second query.

Use rack-mini-profiler to find the source location of all these and remove them.

Cost: 2, Benefit: 2 Good old N+1 squash.

References/Bibliography

- Roberts, Harry. "Core Web Vitals for Search Engine Optimisation." CSS Wizardry, July 3, 2023. https://csswizardry.com/2023/07/core-web-vitals-for-search-engine-optimisation/.
- Google. "Chrome UX Report." Chrome UX Report Dashboard. Accessed March 10, 2025. https://cruxvis.withgoogle.com/#/.
- Eq8. "Image Width and Height in Rails ActiveStorage." Eq8 Blog, March 23, 2021. https://blog.eq8.eu/til/image-width-and-height-in-rails-activestorage.html.
- Google. "Overrides." Chrome for Developers. Accessed March 10, 2025. https://developer.chrome.com/docs/devtools/overrides.
- Argyle, Katie Hempenius, and Barry Pollard. "Optimize resource loading with the fetchpriority attribute." web.dev, January 31, 2024. Accessed March 10, 2025. https://web.dev/articles/fetch-priority?hl=en#browser_priority_and_fetchpriority.

Summary

- Outcome: Reduce p75 LCP to 2.5 seconds for mobile devices, improve Lighthouse Score from 25 to 75
  - Recommendation: (Manually) mark LCP images with fetchpriority=high *Cost: 0 Benefit: 4*
  - Recommendation: Change CDN to Cloudflare or something else that will convert to WebP easily *Cost: 2 Benefit: 3*
  - Recommendation: Pare down Gotham and Didot styles "A & B" to "A". Remove Didot? *Cost: 2 Benefit: 3*
  - Recommendation: Manually walk through templates and mark images as loading=lazy. *Cost: 1 Benefit: 1*
  - Recommendation: Everything on the homepage which isn't the hero should be loading=lazy *Cost: 1 Benefit: 1*

  - Recommendation: Move cdn.mitchellstores.com to shop.mitchellstores.com *Cost: 3 Benefit: 3*
  - Recommendation: Split all.css into 3 to 5 smaller files. *Cost: 3 Benefit: 3*
- Outcome: Reduce CLS to 0.10 on all devices
  - Recommendation: On product pages, fix massive pop when high-res image gets loaded in. *Cost: 2 Benefit: 4*
  - Recommendation: Enforce image heights through code, burndown. *Cost: 3 Benefit: 4*
  - Recommendation: On taxon pages (mens/womens), use aspect-ratio to specify image height *Cost: 2 Benefit: 3*
  - Recommendation: Homepage: manually add image heights *Cost: 1 Benefit: 2*
- Outcome: Improve FCP to 1.8 seconds on all devices
  - Recommendation: Async all Javascript *Cost: 4 Benefit: 5*
  - Recommendation: First-party all fonts *Cost: 3 Benefit: 2*
  - Recommendation: Async Apple Pay, Ahoy and Honeybadger tags *Cost: 3 Benefit: 1*
- Outcome: Reduce TTFB on Products and Taxons
  - Recommendation: Remove (or batch?) ImageType#exists? and corresponding HTTP/HEAD requests. *Cost: 1 Benefit: 2*
  - Recommendation: Fix the Cache multi-get on Products#show *Cost: 2 Benefit: 3*
  - Recommendation: Fix ContentOption/*OptionMatch N+1 for Taxon#show *Cost: 2 Benefit: 2*
  - Recommendation: Fix Role#exists? "N+1" *Cost: 1 Benefit: 1*