How I would design… a Web Crawler!

A system design demonstration

Searching high and low!

Audience

This article is the next in my series of how I would design popular applications. It is recommended (although not entirely necessary) to read the previous posts I've helpfully compiled in a list here. We will expect a basic familiarity with architecture principles and AWS, but hopefully this post is approachable for most engineers.

Some of the ideas are taken from the YouTube channel here, which I've found very useful in learning about system design!

Argument

Initially, let's look at our problem statement.

The System to Design

A web crawler, or spider, is responsible for trawling the internet and indexing the pages that reside there. They wander around the web collecting information on sites. This information is used by companies such as search engines, price comparison tools, copyright detection applications and data scientists.

A useful analogy I pinched from here describes a web crawler as similar to a librarian in a disorganised library. They tootle around and read titles, blurbs, and a few pages of each book to figure out what it's about. They then use this to categorise and organise the library, providing an index card to make books easy to find!

Within our system the requirements are:

- We need to be able to provide a number of seed URLs to begin our search. The crawler will take these pages, follow all of the links within them, and reiterate this process. Using this process we will slowly cover more and more of the web.
- We also need to be able to heed politeness policies. We will assume we only need to set a delay for a certain website before visiting it again.
- We need to be able to separate links to files and links to other pages.
- We need to offer some way of prioritising our crawling. For example, news pages will need to be crawled regularly, while more static content we can leave longer between crawls.
- We also need to prevent indexing the same page twice.

Our usual non-functional requirements stand. It must be reliable, scalable and available.

The Approach

We have a standard approach to system design which is explained more thoroughly in the article here. However the steps are summarised below:

1. **Requirements clarification:** Making sure we have all the information before starting. This may include how many requests or users we are expecting.

2. **Back of the envelope estimation:** Doing some quick calculations to gauge the necessary system performance. For example, how much storage or bandwidth do we need?
3. **System interface design:** What will our system look like from the outside, how will people interact with it? Generally this is the API contract.
4. **Data model design:** What our data will look like when we store it. At this point we could be thinking about relational vs non-relational models.
5. **Logical design:** Fitting it together in a rough system! At this point I'm thinking at a level of 'how would I explain my idea to someone who knows nothing about tech?'
6. **Physical design:** Now we start worrying about servers, programming languages and the implementation details. We can superimpose these on top of the logical design.
7. **Identify and resolve bottlenecks:** At this stage we will have a working system! We now refine the design.

With that said, let's get stuck in!

Requirements Clarification

Initially I would have a few questions. How many websites are we expecting to crawl, how many pages are on a website and roughly how big is a page?

Additionally we may wonder roughly how often we're looking to refresh a page. As we have a sense of priority we will need an average as some are refreshed more often than others.

Back of the Envelope Estimation

Let's say we have a billion websites, each with 100 pages. The average size of a page is 100KB and we're going to refresh, on average, once a month.

This means `1,000,000,000 x 100 x 100KB = 10PB` a month, which is `120PB` a year! We're looking at needing to store a lot of data! This is `3.85GB/s` bandwidth, which is not to be sniffed at.

System Interface Design

We don't have any direct requirements for search, which means we don't have any places for clients to interact with our application. However, we can think about how we want to download pages from the internet. This is our first challenge.

For those of you familiar with React.js, you will know retrieving a web page's HTML is not quite enough. When you load a page there is often some rendering to be done in order to view the full content. Some developers are beginning to use server side rendering which aids with this issue, but you can still expect a number of single page applications.

We can use a tool like Puppeteer to render our pages for us, allowing us to crawl their HTML and add it to our indexes. A first problem solved!

Data Model Design

Now let's look at our data model design. We have two main things we would like to store.

1. A queue of URLs we would like to crawl. We can initialise this with our seed URLs.
2. Pages we have downloaded from the internet in the form of HTML.

Our queue of URLs we will store along with a next crawl time. This will allow us to respect the politeness policy set up by sites, as well as the priority we have assigned a URL. It should be enough to store something similar to:

- `url VARCHAR`
- `nextCrawlTime TIMESTAMP`

We can then periodically poll this data store to see which URLs are ready to be examined.

The pages we have downloaded from the internet are best kept in file storage. As most of my expertise lies in AWS, I would suggest S3. We will see in a later section how this fits into our overall design.

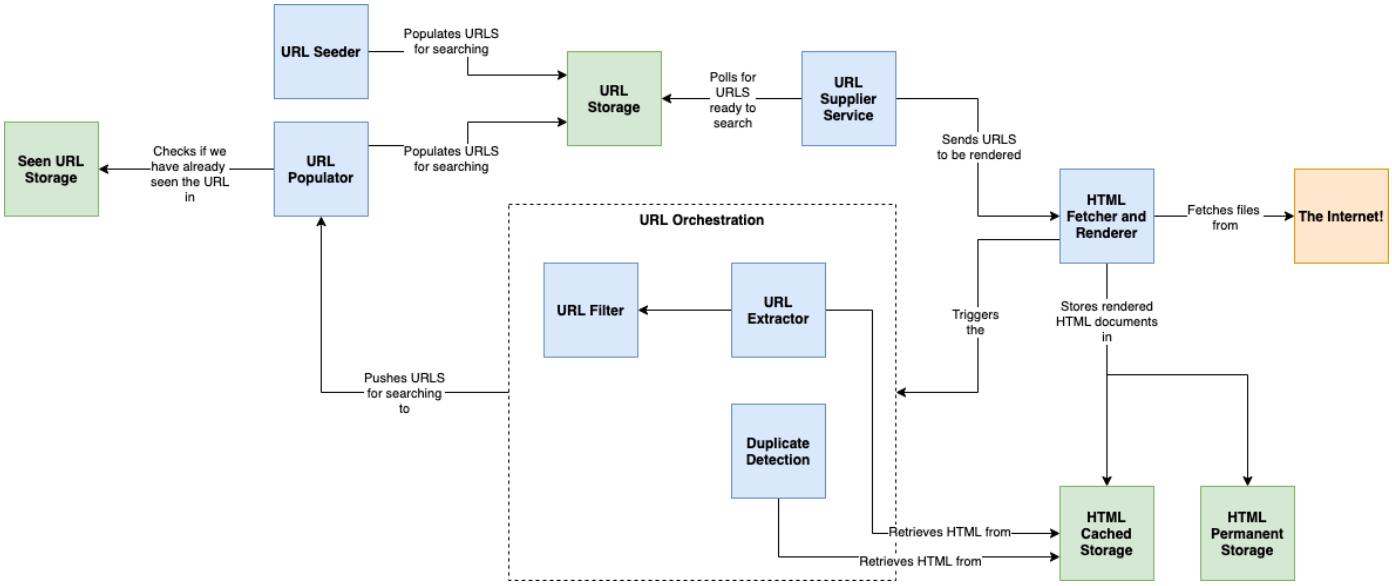
Logical Design

To begin our logical diagrams, let's think through what we would like the system to do. Initially, we want to be able to put a list of URLs somewhere, which are fed to a service which fetches and renders them, as well as storing their HTML.

We should pass the HTML along to another set of services which (as per our requirements):

- Checks to see if the page is a duplicate of one we have seen before.
- Filters out links to other pages and objects from that page.
- Extracts and persists the next URLs for searching.

Let's put this in a rough diagram.



A basic logical design

Let's step through this in slightly more detail. Initially we have a URL seeder, which we may choose to populate with the 100 most popular websites, or similar. This persists these URLs along with a next crawl time to the URL storage. This is polled by the URL supplier service which looks for entries whose next crawl time is less than the current time. It then sends these to the HTML Fetcher and Renderer.

The HTML Fetcher and Renderer goes out to the URL and retrieves and persists the HTML in two places. We can use a `HEAD` call to the page initially to see if it has been updated. We also need to set a user agent so the site knows who is crawling it. cache, for quick access by downstream services, and permanent storage. It then triggers a URL orchestration service, which contains all of our services for extracting URLs, deduplicating HTML, and filtering URLs dependent on if they are files or other pages.

Finally we pass to the URL populator which checks to see if we have seen this particular URL before, and if so discard it. If we haven't then we can add it to the URL storage along with an appropriate next crawl time according to the politeness policy and the priority. Notice how this is similar to a breadth first search.

At this point it's worth taking an aside to discuss bloom filters, duplicate detection and `robots.txt` files.

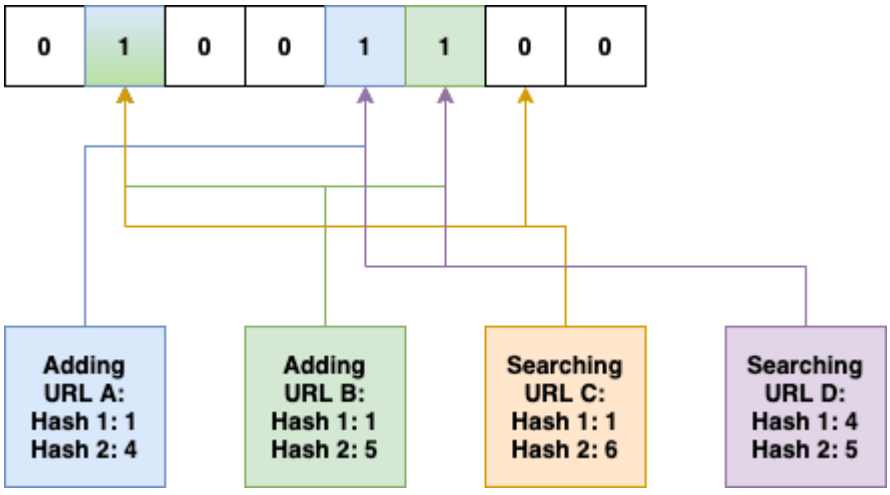
Bloom Filters

In our particular use case we need to check if we have seen a URL before. One way of doing this would be to check through the list one by one, seeing if the URL crops up. This is `O(n)` and begins to fall apart when we try and distribute the list of URLs across multiple servers.

Another solution could be to hash the URLs and see if the hash exists, however in the case of hash collisions we have a worst case complexity of `O(n)` . Additionally our table will become huge and will add latency as we read and write from it (especially if it is on disk).

Enter the bloom filter. This is a probabilistic data structure. Rather than giving a definite answer (this URL exists/ this URL doesn't exist), it gives an estimate (there is a 90% chance this URL exists/ there is a 100% chance this URL doesn't exist). As we explain how the bloom filter works we can see why we don't always know if a URL *does* exist, but we always know if a URL *doesn't* exist.

As always, we can elucidate with a confusing diagram of boxes and arrows.



Demonstrating a bloom filter

The bloom filter is based around a bit array. We have two hash functions, 1 and 2. We apply both hash functions to our URLs and come out with two separate numbers. We then set the bits of the array according to these numbers.

For example, we're adding URL A to the filter. The first hash function generates the number 1, the second the number 4. We then set the bits in the array accordingly. URL B works in a similar way.

Now let's look up two URLs, C and D. We run URL C through the hash functions and get 1 and 6. We can see 6 is not set, so we know it can't appear! However, if we run D through the functions we get 4 and 5, which are both set.

Notice we haven't actually seen D before! It just happens that another two URLs have combined to match its hash. This is the probabilistic element. We've *probably* seen it before, but in this case we haven't!

Next let's think about deduplication.

Deduplication

There are many sites which essentially publish the same page. Think aggregator sites, archival sites etc. We don't want to bother indexing the same page multiple times, but how do we detect these duplicates?

A naive approach would be to go word for word down the page comparing, or to generate a hash. However, this is slow and costly. Instead there are a number of algorithms that do a comparison for us: minhash and simhash (a method of seeing how close two sets are), fuzzy search (searching to see if the documents are roughly similar), latent semantic indexing (a technique for discovering the statistical relationship of words that appear together in documents) or the standard boolean model of information retrieval (another method of search).

We won't go into detail here as it's beyond the scope of the article (and probably beyond my intellectual scope), but it's enough to know they're there!

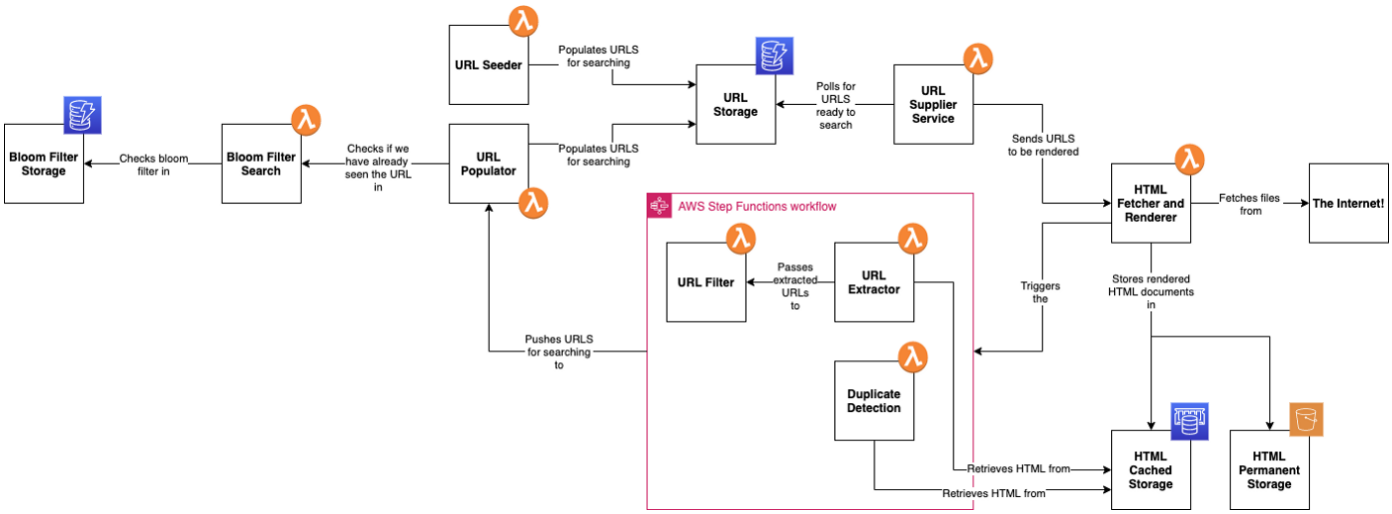
Robots.txt Files

A really useful introduction to `robot.txt` files is published on the page here. To offer a quick summary, these files sit at the root of a site and are used to provide crawlers information. Here's the one for the BBC.

These files allow sites to express their preferences. We can do things like disallow a user agent from crawling certain spaces, and expose a sitemap to aid crawling. We can also use the unofficial `crawl-delay` directive to let some crawlers know how long to wait before hitting it again.

Now we've covered these features, let's move onto the physical design.

Physical Design



Basic physical design

We can go mainly serverless for this! The URL Seeder can be an AWS Lambda function which we provide a number of URLs. These are then persisted in the URL storage. We then have another scheduled Lambda, which polls the URL storage and sends any required URLs to the fetcher and renderer.

The fetcher and renderer then spins up a lambda per request, downloads, renders and stores the page. Our permanent storage is S3 (as we're storing documents), and our cache storage is Elasticache.

When we've finished downloading our page we set off an AWS Step Function, an orchestration framework. This is really useful as it allows us to do things like retries when

components fail, and eases the integration of different services. I've written a previous article on it here.

Finally we push to the URL populator, which checks for duplicate URLs using a Lambda service containing the bloom filter. Notice that an array of bits can also be stored as just consecutive bits translating to a number! We store this number in a cache so it can be accessed by our Lambda. This completes the loop.

Identify and Resolve Bottlenecks

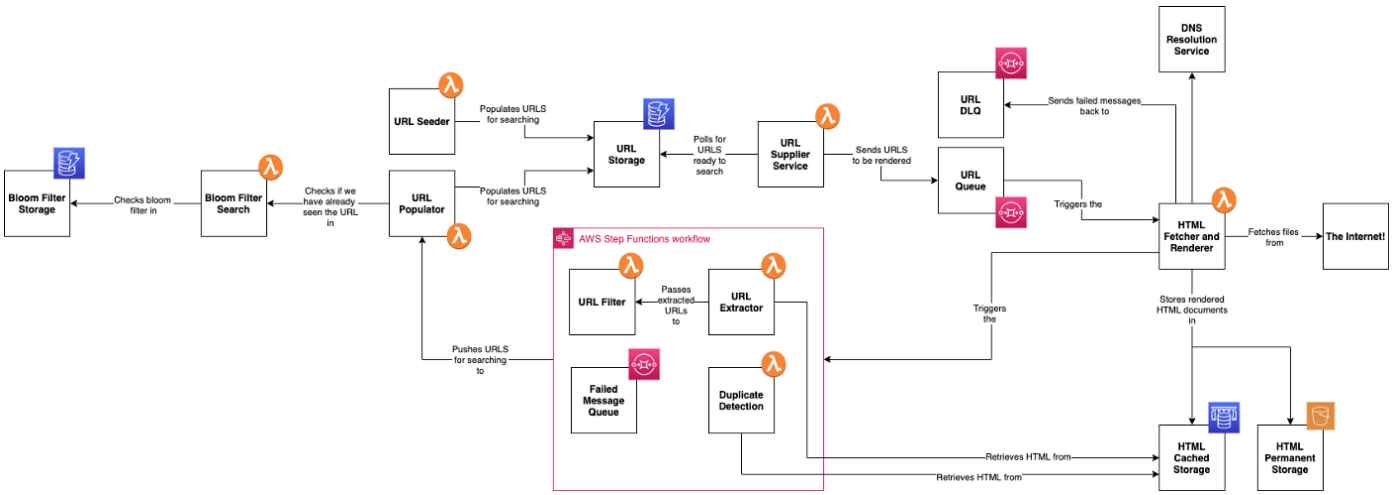
This design is far from perfect. The major choking point will be on DNS resolution. Each time we query a website we need to check the local cache to see if we have a DNS entry (which we might not even have as we're running serverless), then go and query a bunch of DNS servers to find the IP address of the site we're looking for. Not that speedy.

If instead we can maintain our own list of DNS resolutions we can immediately query this (especially helpful in our serverless setup) and save time.

The other issue I would be worried about would be resiliency. What if any of the various components failed? To aid this we could introduce a queue with retry logic and a dead letter queue between the supplier and the fetcher. We could also implement a failed message queue in the step function. We would send messages which have failed one of our orchestrated services and need replaying.

Additionally, I'm curious about how performant it would be to poll the DynamoDb instance. There are some solutions which use services, queue and a hashtable, but it seems overengineered unless I'm missing something. There is also a question over the best way to store a bloom filter.

Regardless, the final design is below.



Final web crawler design

Conclusion

In conclusion we have investigated, discussed and proposed a solution to designing a web crawler.