How to crawl a quarter billion webpages in 40 hours

by Michael Nielsen on August 10, 2012

More precisely, I crawled 250,113,669 pages for just under 580 dollars in 39 hours and 25 minutes, using 20 Amazon EC2 machine instances.

I carried out this project because (among several other reasons) I wanted to understand what resources are required to crawl a small but non-trivial fraction of the web. In this post I describe some details of what I did. Of course, there's nothing especially new: I wrote a vanilla (distributed) crawler, mostly to teach myself something about crawling and distributed computing. Still, I learned some lessons that may be of interest to a few others, and so in this post I describe what I did. The post also mixes in some personal working notes, for my own future reference.

What does it mean to crawl a non-trivial fraction of the web? In fact, the notion of a "non-trivial fraction of the web" isn't well defined. Many websites generate pages dynamically, in response to user input – for example, Google's search results pages are dynamically generated in response to the user's search query. Because of this it doesn't make much sense to say there are so-and-so many billion or trillion pages on the web. This, in turn, makes it difficult to say precisely what is meant by "a non-trivial fraction of the web". However, as a reasonable proxy for the size of the web we can use the number of webpages indexed by large search engines. According to this **presentation** by Googler **Jeff Dean**, as of November 2010 Google was indexing "tens of billions of pages". (Note that the **number of urls** is in the trillions, apparently because of duplicated page content, and multiple urls pointing to the same content.) The now-defunct search engine **Cuil** claimed to index **120 billion pages**. By comparison, a quarter billion is, obviously, very small. Still, it seemed to me like an encouraging start.

Code: Originally I intended to make the crawler code available under an open source license at GitHub. However, as I better understood the cost that crawlers impose on websites, I began to have reservations. My crawler is designed to be polite and impose relatively little burden on any single website, but could (like many crawlers) easily be modified by thoughtless or malicious people to impose a heavy burden on sites. Because of this I've decided to postpone (possibly indefinitely) releasing the code.

There's a more general issue here, which is this: who gets to crawl the web? Relatively few sites exclude crawlers from companies such as Google and Microsoft. But there are a *lot* of crawlers out there, many of them without much respect for the needs of individual siteowners. Quite reasonably, many siteowners take an aggressive approach to shutting down activity from less well-known crawlers. A possible side effect is that if this becomes too common at some point in the future, then it may impede the development of useful new services, which need to crawl the web. A possible long-term solution may be services like **Common Crawl**, which provide access to a common corpus of crawl data.

I'd be interested to hear other people's thoughts on this issue.

Follow Michael

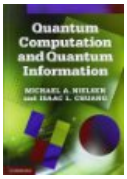
Subscribe to this blog
Michael's main blog
Follow Michael on Twitter

Other places on the web

GitHub
Delicious
Delicious research links for book project
Delicious research links for last book
Google Plus

Books

Reinventing Discovery: The New Era of Networked Science (Errata)



Quantum Computation and Quantum Information

Other projects

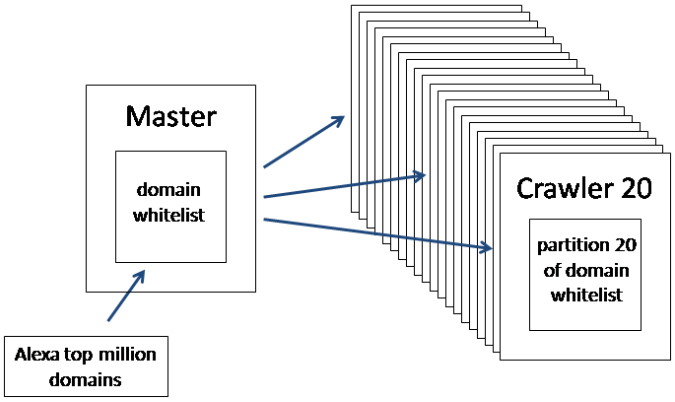
Essays
TEDxWaterloo video about open science
Quantum computing for the determined
Polymath wiki

Recent Posts

How the backpropagation algorithm works
Reinventing Explanation
How the Bitcoin protocol actually works
Neural Networks and Deep Learning: first chapter now live

(*Later update:* I get regular email asking me to send people my code. Let me pre-emptively say: I decline these requests.)

Architecture: Here's the basic architecture:



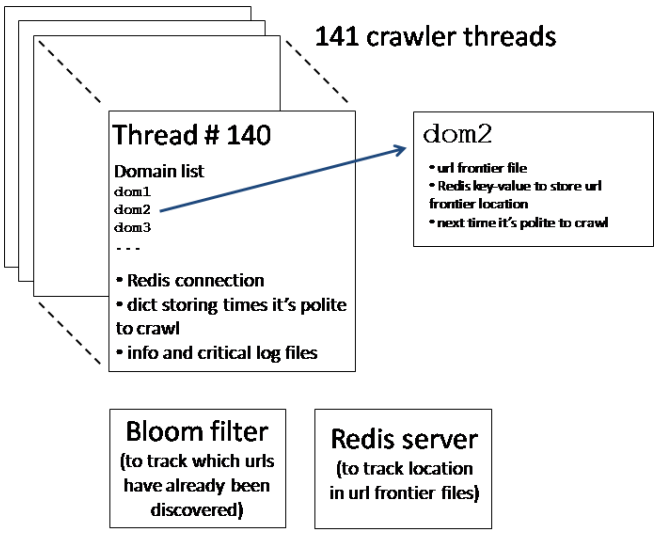
The master machine (my laptop) begins by downloading **Alexa's** list of the **top million domains**. These were used both as a domain whitelist for the crawler, and to generate a starting list of seed urls.

The domain whitelist was partitioned across the 20 EC2 machine instances in the crawler. This was done by numbering the instances $0, 1, 2, \ldots, 19$ and then allocating the domain `domain` to instance number `hash(domain) % 20`, where `hash` is the standard Python hash function.

Deployment and management of the cluster was handled using **Fabric**, a well-documented and nicely designed Python library which streamlines the use of ssh over clusters of machines. I managed the connection to Amazon EC2 using a **set of Python scripts** I wrote, which wrap the **boto** library.

I used 20 Amazon EC2 **extra large** instances, running Ubuntu 11.04 (Natty Narwhal) under the **ami-68ad5201 Amazon machine image** provided by Canonical. I used the extra large instance after testing on several instance types; the extra large instances provided (marginally) more pages downloaded per dollar spent. I used the US East (North Virginia) region, because it's the least expensive of Amazon's regions (along with the US West, Oregon region).

Single instance architecture: Each instance further partitioned its domain whitelist into 141 separate blocks of domains, and launched 141 Python threads, with each thread responsible for crawling the domains in one block. Here's how it worked (details below):



The reason for using threads is that the Python standard library uses blocking I/O to

Why Bloom filters work the way they do

Recent Comments

Rich on How the Bitcoin protocol actually works
Kaitlyn on If correlation doesn't imply causation, then what does?
Kaitlyn on If correlation doesn't imply causation, then what does?
What I learned at Build Peace (by Jonathan Stray) | Build Peace on If correlation doesn't imply causation, then what does?
What I learned at Build Peace (Jonathan Stray) | Build Peace on If correlation doesn't imply causation, then what does?

Linklog

OSI: The Internet That Wasn't - IEEE Spectrum
How TCP/IP won over the OSI. Interesting in part for the discussion of what openness means, exactly, and when it is advantageous for a process to be open.

Lester Dent's Master Plot Formula
More in the Moorcock vein. It's easy to imagine the reaction of the critic, holding their nose at writing to formula. But you can turn that around, regarding Dent (and, more plausibly, Moorcock) as a student and theoretician of structure. And that's a pretty powerful point of view. Of course, word-by-word Dent is a poor […]

How to Write a Book in Three Days: Lessons from Michael Moorcock
Fascinating both intrinsically, and for the commentary. The commentary first: part of the interest is from people who desire an easy way to write (or, more accurately, to have written). But there is also clearly a genuine interest on the part of many: what does this guy know that I don't about storytelling? You may […]

Gossip is Philosophy
Kevin Kelly interviews Brian Eno. Slow to get going, but fascinating. Eno proposes "process, not product", says that it's his "ease of seduction" that means he often gets things first, talks about putting more "Africa" into computers, and generally makes many interesting comments.

Susan Sontag interview
Conveys well how much force of personality and intellect Sontag brought to her writing.

Documentaries

Visual Storytelling: The Digital Video Documentary
A brief skim suggests that this is a very good basic guide to making documentaries.

handle http network connections. This means that a single-threaded crawler would spend most of its time idling, usually waiting on the network connection of the remote machine being crawled. It's much better to use a multi-threaded crawler, which can make fuller use of the resources available on an EC2 instance. I chose the number of crawler threads (141) empirically: I kept increasing the number of threads until the speed of the crawler started to saturate. With this number of threads the crawler was using a considerable fraction of the CPU capacity available on the EC2 instance. My informal testing suggested that it was CPU which was the limiting factor, but that I was not so far away from the network and disk speed becoming bottlenecks; in this sense, the EC2 extra large instance was a good compromise. Memory useage was never an issue. It's possible that for this reason EC2's high-CPU extra large instance type would have been a better choice; I only experimented with this instance type with early versions of the crawler, which were more memory-limited.

How domains were allocated across threads: The threads were numbered $0, 1, \ldots, 140$, and domains were allocated on the basis of the Python hash function, to thread number `hash(domain) % 141` (similar to the allocation across machines in the cluster). Once the whitelisted domains / seed urls were allocated to threads, the crawl was done in a simple breadth-first fashion, i.e., for each seed url we download the corresponding web page, extract the linked urls, and check each url to see: (a) whether the extracted url is a fresh url which has not already been seen and added to the url frontier; and (b) whether the extracted url is in the same seed domain as the page which has just been crawled. If both these conditions are met, the url is added to the url frontier for the current thread, otherwise the url is discarded. With this architecture we are essentially carrying out a very large number of independent crawls of the whitelisted domains obtained from Alexa.

Note that this architecture also ensures that if, for example, we are crawling a page from TechCrunch, and extract from that page a link to the Huffington Post, then the latter link will be discarded, even though the Huffington Post is in our domain whitelist. The only links added to the url frontier will be those that point back to TechCrunch itself. The reason we avoid adding dealing with (whitelisted) external links is because: (a) it may require communication between different EC2 instances, which would substantially complicate the crawler; and, more importantly, (b) in practice, most sites have lots of internal links, and so it's unlikely that this policy means the crawler is missing much.

One advantage of allocating all urls from the same domain to the same crawler thread is that it makes it much easier to crawl politely, since no more than one connection to a site will be open at any given time. In particular, this ensures that we won't be hammering any given domain with many simultaneous connections from different threads (or different machines).

Problems for the author

> For some very large and rapidly changing websites it may be necessary to open multiple simultaneous connections in order for the crawl to keep up with the changes on the site. How can we decide when that is appropriate?

How the url frontiers work: A *separate* url frontier file was maintained for each domain. This was simply a text file, with each line containing a single url to be crawled; initially, the file contains just a single line, with the seed url for the domain. I spoke above of the url frontier for a thread; that frontier can be thought of as the combination of all the url frontier files for domains being crawled by that thread.

Each thread maintained a connection to a **redis** server. For each domain being crawled by the thread a redis key-value pair was used to keep track of the current position in the url frontier file for that domain. I used redis (and the **Python bindings**) to store this information in a fashion that was both persistent and fast to look up. The persistence was important because it meant that the crawler could be stopped and started at will, without losing track of where it was in the url frontier.

The Great Ecstasy of the Woodcarver Steiner - Werner Herzog

Documentary of Wolfgang Steiner, one of the world's top ski-jumpers in the 1970s. The spine of the documentary is a sequence of extraordinary shots of Steiner's jumps, taken with a pair of high-speed cameras.

Inge Druckrey: Teaching to See

A reminder of the diversity of life: the passion and learning and insight people pour into calligraphy, font design, and design more generally.

Indie Game: The Movie (2012) - IMDb

Striking for the level of emotional commitment and uncertainty (and turmoil) on the part of the creators!

Step Into Liquid (2003)

Remarkable survey of the cutting edge of surfing. We see the origins of tow-rope surfing (where surfers are pulled by jet skis into waves that are too big to paddle out to), the use of hydrofoil designs that put the board a foot or two _above_ the wave, and even the use of weather stations [...]

Search

Search

Archives

April 2014
January 2014
December 2013
November 2013
September 2012
August 2012
June 2012
April 2012
March 2012
February 2012
January 2012
July 2011
June 2011

Each thread also maintained a dictionary whose keys were the (hashed) domains for that thread. The corresponding values were the next time it would be polite to crawl that domain. This value was set to be 70 seconds after the last time the domain was crawled, to ensure that domains weren't getting hit too often. The crawler thread simply iterated over the keys in this dictionary, looking for the next domain it was polite to crawl. Once it found such a domain it then extracted the next url from the url frontier for that domain, and went about downloading that page. If the url frontier was exhausted (some domains run out of pages to crawl) then the domain key was removed from the dictionary. One limitation of this design was that when restarting the crawler each thread had to identify again which domains had already been exhausted and should be deleted from the dictionary. This slowed down the restart a little, and is something I'd modify if I were to do further work with the crawler.

Use of a Bloom filter: I used a **Bloom filter** to keep track of which urls had already been seen and added to the url frontier. This enabled a very fast check of whether or not a new candidate url should be added to the url frontier, with only a low probability of erroneously adding a url that had already been added. This was done using **Mike Axiak**'s very nice C-based **pybloomfiltermmap**.

Update: Jeremy McLain **points out in comments** that I've got this backward, and that with a Bloom filter there is a low probability "that you will never crawl certain URLs because your bloom filter is telling you they have already been crawled when in fact they have not." A better (albeit slightly slower) solution would be to simply store all the URLs, and check directly.

Anticipated versus unanticipated errors: Because the crawler ingests input from external sources, it needs to deal with many potential errors. By design, there are two broad classes of error: *anticipated errors* and *unanticipated errors*.

Anticipated errors are things like a page failing to download, or timing out, or containing unparseable input, or a `robots.txt` file disallowing crawling of a page. When anticipated errors arise, the crawler writes the error to a (per-thread) informational log (the "info log" in the diagram above), and continues in whatever way is appropriate. For example, if the `robots.txt` file disallows crawling then we simply continue to the next url in the url frontier.

Unanticipated errors are errors which haven't been anticipated and designed for. Rather than the crawler falling over, the crawler simply logs the error (to the "critical log" in the diagram above), and moves on to the next url in the url frontier. At the same time, the crawler tracks how many unanticipated errors have occurred in close succession. If many unanticipated errors occur in close succession it usually indicates that some key piece of infrastructure has failed. Because of this, if there are too many unanticipated errors in close succession, the crawler shuts down entirely.

As I was developing and testing the crawler, I closely followed the unanticipated errors logged in the critical log. This enabled me to understand many of the problems faced by the crawler. For example, early on in development I found that sometimes the html for a page would be so badly formed that the html parser would have little choice but to raise an exception. As I came to understand such errors I would rewrite the crawler code so such errors become anticipated errors that were handled as gracefully as possible. Thus, the natural tendency during development was for unanticipated errors to become anticipated errors.

Domain and subdomain handling: As mentioned above, the crawler works by doing lots of parallel intra-domain crawls. This works well, but a problem arises because of the widespread use of subdomains. For example, if we start at the seed url `http://barclays.com` and crawl only urls within the `barclays.com` domain, then we quickly run out of urls to crawl. The reason is that most of the internal links on the `barclays.com` site are actually to `group.barclays.com`, not `barclays.com`. Our crawler should also add urls from the latter domain to the url frontier for `barclays.com`.

We resolve this by stripping out all subdomains, and working with the stripped domains when deciding whether to add a url to the url frontier. Removing subdomains turns out to be a surprisingly hard problem, because of variations in the way domain names are formed. Fortunately, the problem seems to be well solved using **John Kurkowski's tldextract library**.

On the representation of the url frontier: I noted above that a separate url frontier file was maintained for each domain. In an early version of the code, each crawler thread had a url frontier maintained as a *single* flat text file. As a crawler thread read out lines in the file, it would crawl those urls, and append any new urls found to the end of the file.

This approach seemed natural to me, but organizing the url frontier files on a per-thread (rather than per-domain) basis caused a surprising number of problems. As the crawler thread moved through the file to find the next url to crawl, the crawler thread would encounter urls belonging to domains that were not yet polite to crawl because they'd been crawled too recently. My initial strategy was simply to append such urls to the end of the file, so they would be found again later. Unfortunately, there were often a *lot* of such urls in a row – consecutive urls often came from the same domain (since they'd been extracted from the same page). And so this strategy caused the file for the url frontier to grow very rapidly, eventually consuming most disk space.

Exacerbating this problem, this approach to the url frontier caused an unforseen "domain clumping problem". To understand this problem, imagine that the crawler thread encountered (say) 20 consecutive urls from a single domain. It might crawl the first of these, extracting (say) 20 extra urls to append to the end of the url frontier. But the next 19 urls would all be skipped over, since it wouldn't yet be polite to crawl them, and they'd also be appended to the end of the url frontier. Now we have 39 urls from the same domain at the end of the url frontier. But when the crawler thread gets to those, we may well have the same process repeat – leading to a clump of 58 urls from the same domain at the end of the file. And so on, leading to very long runs of urls from the same domain. This consumes lots of disk space, and also slows down the crawler, since the crawler thread may need to examine a large number of urls before it finds a new url it's okay to crawl.

These problems could have been solved in various ways; moving to the per-domain url frontier file was how I chose to address the problems, and it seemed to work well.

Choice of number of threads: I mentioned above that the number of crawler threads (141) was chosen empirically. However, there is an important constraint on that number, and in particular its relationship to the number (20) of EC2 instances being used. Suppose that instead of 141 threads I'd used (say) 60 threads. This would create a problem. To see why, note that any domain allocated to instance number 7 (say) would necessarily satisfy `hash(domain) % 20 = 7`. This would imply that `hash(domain) % 60 = 7` or 27 or 47, and as a consequence all the domains would be allocated to just one of three crawler threads (thread numbers 7, 27 and 47), while the other 57 crawler threads would lie idle, defeating the purpose of using multiple threads.

One way to solve this problem would be to use two **independent** hash functions to allocate domains to EC2 instances and crawler threads. However, an even simpler way of solving the problem is to choose the number of crawler threads to be coprime to the number of EC2 instances. This coprimality ensures that domains will be allocated reasonably evenly across both instance and threads. (I won't prove this here, but it can be proved with a little effort). It is easily checked that 141 and 20 are coprime.

Note, incidentally, that Python's `hash` is not a true hash function, in the sense that it doesn't guarantee that the domains will be spread evenly across EC2 instances. It turns out that Python's `hash` takes similar key strings to similar hash values. I talk more about this point (with examples) in the fifth paragraph of **this post**. However, I

found empirically that `hash` seems to spread domains evenly enough across instances, and so I didn't worry about using a better (but slower) hash function, like those available through Python's `hashlib` library.

Use of Python: All my code was written in Python. Initially, I wondered if Python might be too slow, and create bottlenecks in the crawling. However, profiling the crawler showed that most time was spent either (a) managing network connections and downloading data; or (b) parsing the resulting webpages. The parsing of the webpages was being done using **lxml**, a Python binding to fast underlying C libraries. It didn't seem likely to be easy to speed that up, and so I concluded that Python was likely not a particular bottleneck in the crawling.

Politeness: The crawler used Python's **robotparser library** in order to observe the **robots exclusion protocol**. As noted above, I also imposed an absolute 70-second minimum time interval between accesses to any given domain. In practice, the mean time between accesses was more like 3-4 minutes.

In initial test runs of the crawler I got occasional emails from webmasters asking for an explanation of why I was crawling their site. Because of this, in the crawler's **User-agent** I included a link to a **webpage** explaining the purpose of my crawler, how to exclude it from a site, and what steps I was taking to crawl politely. This was (I presume) both helpful to webmasters and also helpful to me, for it reduced the number of inquiries. A handful of people asked me to exclude their sites from the crawl, and I complied quickly.

Problems for the author

> Because my crawl didn't take too long, the `robots.txt` file was downloaded just once for each domain, at the beginning of the crawl. In a longer crawl, how should we decide how long to wait between downloads of `robots.txt`?

Truncation: The crawler truncates large webpages rather than downloading the full page. It does this in part because it's necessary – it really wouldn't surprise me if someone has a terabyte html file sitting on a server somewhere – and in part because for many applications it will be of more interest to focus on earlier parts of the page.

What's a reasonable threshold for truncation? According to **this report** from Google, as of May 2010 the average network size of a webpage from a top site is 312.04 kb. However, that includes images, scripts and stylesheets, which the crawler ignores. If you ignore the images and so on, then the average network size drops to just 33.66 kb.

However, that number of 33.66 kb is for content which may be served compressed over the network. Our truncation will be based on the uncompressed size. Unfortunately, the Google report doesn't tell us what the average size of the uncompressed content is. However, we can get an estimate of this, since Google reports that the average uncompressed size of the *total* page (including images and so on) is 477.26 kb, while the average network size is 312.04 kb.

Assuming that this compression ratio is typical, we estimate that the average uncompressed size of the content the crawler downloads is 51 kb. In the event, I experimented with several truncation settings, and found that a truncation threshold of 200 kilobytes enabled me to download the great majority of webpages in their entirety, while addressing the problem of very large html files mentioned above. (Unfortunately, I didn't think to check what the *actual* average uncompressed size was, my mistake.)

Storage: I stored all the data using EC2's built-in **instance storage** – 1.69 Terabytes for the extra-large instances I was using. This storage is non-persistent, and so any data stored on an instance will vanish when that instance is terminated. Now, for many kinds of streaming or short-term analysis of data this would be adequate – indeed, it might not even be necessary to store the data at all. But, of course, for many applications of a crawl this approach is not appropriate, and the instance

storage should be supplemented with something more permanent, such as S3. For my purposes using the instance storage seemed fine.

Price: The price broke down into two components: (1) 512 dollars for the use of the 20 extra-large EC2 instances for 40 hours; and (2) about 65 dollars for a little over 500 gigabytes of outgoing bandwidth, used to make http requests. Note that Amazon does not charge for incoming bandwidth (a good thing, too!) It would be interesting to compare these costs to the (appropriately amortized) costs of using other cloud providers, or self-hosting.

Something I didn't experiment with is the use of Amazon's **spot instances**, where you can bid to use Amazon's unused EC2 capacity. I didn't think of doing this until just as I was about to launch the crawl. When I went to look at the spot instance pricing history, I discovered to my surprise that the spot instance prices are often a factor of 10 or so lower than the prices for on-demand instances! Factoring in the charges for outgoing bandwidth, this means it may be possible to use spot instances to do a similar crawl for 120 dollars or so, a factor of five savings. I considered switching, but ultimately decided against it, thinking that it might take 2 or 3 days work to properly understand the implications of switching, and to get things working exactly as I wanted. Admittedly, it's possible that it would have taken much less time, in which case I missed an opportunity to trade some money for just a little extra time.

Improvements to the crawler architecture: Let me finish by noting a few ways it'd be interesting to improve the current crawler:

> For many long-running applications the crawler would need a smart crawl policy so that it knows when and how to re-crawl a page. According to a **presentation** from **Jeff Dean**, Google's mean time to index a new page is now just minutes. I don't know how that works, but imagine that notification protocols such as **pubsubhubbub** play an important role. It'd be good to change the crawler so that it's pubsubhubbub aware.
>
> The crawler currently uses a threaded architecture. Another quite different approach is to use an **evented architecture**. What are the pros and cons of a multi-threaded versus an evented architecture?
>
> The instances in the cluster are configured using fabric and shell scripts to install programs such as redis, pybloomfilter, and so on. This is slow and not completely reliable. Is there a better way of doing this? Creating my own EC2 AMI? Configuration management software such as **Chef** and **Puppet**? I considered using one of the latter, but deferred it because of the upfront cost of learning the systems.
>
> Logging is currently done using Python's `logging` module. Unfortunately, I'm finding this is not well-adapted to Python's threading. Is there a better solution?
>
> The crawler was initially designed for crawling in a batch environment, where it is run and then terminates. I've since modified it so that it can be stopped, modifications made, and restarted. It'd be good to add instrumentation so it can be modified more dynamically, in real time.
>
> Many interesting research papers have been published about crawling. I read or skimmed quite a few while writing my crawler, but ultimately used only a few of the ideas; just getting the basics right proved challenging enough. In future iterations it'd be useful to look at this work again and to incorporate the best ideas. Good starting points include a **chapter** in the book by Manning, Raghavan and Sch\"utze, and a **survey paper** by Olston and Najork. Existing open source crawlers such as **Heritrix** and **Nutch** would also be interesting to look at in more depth.

From → Uncategorized

67 Comments

harsh permalink

very thorough. thanks for sharing your deep experiments. it helps.

Tim McNamara permalink

Threading is quite likely to have been a very expensive option. Celery's documentation[0] provides an example of using 1000 concurrent eventlets per machine.

With the right architecture, you can provide a far-higher level of concurrency than ~140 per EC2 instance.

[0] http://docs.celeryproject.org/en/latest/userguide/concurrency/eventlet.html

Michael Nielsen permalink

Thanks for the tip – I need to look into Celery and eventlets.

Ask Solem permalink

1000 per machine depends on the number of CPU's though, I would try to use several 'celery worker' instances on each machine, maybe as many as there are CPU cores with 1000 greenthreads each. You will be able to do far more many requests/s with this setup than using the multiprocessing/threads pool. Of course the down side is if you need to do anything blocking that will drag the overall performance down considerably.

Ask Solem permalink

Oh, and there's an eventlet crawler example that also happens to be using a bloomfilter (though the example is pretty stupid since the bloomfilter is passed along, so it's more of a 'lucky if we don't hit the same page twice' approach, just used for illustration (the bloomfilter would need be a separate service).
https://github.com/celery/celery/blob/master/examples/eventlet/webcrawler.py

Curious_Hacker permalink

Hello Mr Celery,

Looking at your crawler example, I see the following comment:

"A BloomFilter with a capacity of 100_000 members and an error rate of 0.001 is 2.8MB pickled, but if compressed with zlib it only takes up 2.9kB(!)."

If you're implementing a bloom filter correctly, and its populated to about at least 1/2 its capacity, then compression at the levels described in the comments would be highly unlikely.

Furthermore, Bloom filters are only useful if they can completely reside in l2 or l3, if they are so large that they end up being flushed to RAM…they

are essentially useless – 2.8MB is rather large even for a server class I3.

David Higgins permalink

Tutorials on how to crawl aside, there has to be a motivation? So you throw money at machines that can do this. You write your script(s), and you get your data. So what?

Are you going to do something with this data? Where may you upload it?

I want:

– Multiple paste-bins of this data.
– On many paste-bins, and not one paste-bin service.

http://r3versin.com/blog/bins/

– A Crawl-able version of the data split into chunks so the data is digestible by search engines

– An analysis of keywords in the data. (What's the status quo regarding keywords, excluding sort-tail queries like "the", and other sentence clauses.

– A machine readable dump of links contained within the crawl data (Not just links to the data that has been crawled)

– I want the data to be available for download. In GZip format / RAR / Tarball format (Preferably in small chunks so I don't have to seek out programs that can read 1GB text files)

Jorge permalink

Does this guy work for you?

Daeyun permalink

Does this guy work for you?

Dave permalink

What an impolite request.

Jonathan permalink

"I want, I want, I want…" It's obvious you're going nowhere in life. fast.

niko_gramophon permalink

Post the code or it didn't happend 

AWS permalink

Thanks for a great write up! You don't need to see the code to understand the concepts here. I was also wondering how Google and others are handling javascript apps or pages using div instead of links, etc. Web builders should know better and use proper hooks but it's an imperfect world. Appreciate the info!

Duncan permalink

Did you consider using gevent (http://www.gevent.org/) for networking? This gets around Python's problem of blocking io by using non-blocking sockets, libev and coroutines. It's also extremely fast.

This would allow a single-threaded process to have open connections to thousands of pages, with no time wasted in io-wait. It's hard to know for certain without benchmarks, but I would expect your performance to improve significantly using one single-threaded process per core with non-blocking io.

Michael Nielsen permalink

No, I didn't, I'll look into it for the future. Thanks for the tip.

Zhenya permalink

Duncan, how do you plug gevent into scrapy? It doesn't seem like scrapy itself supports it so I guess certain customization is required in order to use gevent for the crawler, right?

François Beausoleil permalink

Great work, I like your comments. I was tasked to write a crawler once, and faced many similar challenges. In my case though I never respected robots.txt, nor imposed a limit on the frequency of crawling a domain. In my defense, I was getting random URLs from the Twitter spritzer, so I wasn't too worried about hitting the same URL multiple times.

Regarding Puppet, Chef or an AMI, if you want an instance up fast, the only way to go is to build your own instance. That's the only way to come up and be fully ready to start. In all other cases, you have to install software manually. To use spot instances, I believe the AMI option would have been best.

Ryan permalink

Cool project — but I think it's rather naive to believe that open sourcing the code will do (or allow) any harm that can't be done already.

Federico Sasso permalink

Great article, with good in-depth explanations.
Thank you for referencing publicsuffix.org, I will use it to fix my home baked solution to the subdomain issue.
Besides, every detail you gave I found valuable and interesting.

As an author of a web crawler myself, I encountered many other problems attempting to make the crawler both robust and polite:
– fetching a pool of robots.txt per domain: each subdomain has to obey to its own robots.txt file (with also different crawl-delay); also http/https version should be treated as obeying to different robots.txt file
– correctly dealing with non-200 http response codes when fetching robots.txt
– supporting http compression (gzip/deflate), taking into account a possible 406 return code
– using a queue based implementation to implement the breadth first visit (I know it's

trivial, but I saw many using a resource consuming implementation based on a recursion algorithm)
– being resistent to spider traps (supporting robots.txt helps, but e.g. some calendars are involontary infinite-loop bot traps)
– using a robust html parser to deal with badly formed html
– efficiently extracting links from a dom to avoid regex false positives
– correctly transforming relative path to absolute urs (base attribute, weird characters in dotted path, …)
– dealing with url canonicalization (case sensitivity issues, tracking and session id querystring parameters, querystring parameters order, …)
– exit strategies when encountering repeated http 500 error codes, or 503 (under maintenance)
– …

Regards,

Michael Nielsen permalink

Thanks for the comments. Yes, these are all important. I dealt with several of these points, but not everything. In particular, in future iterations I'd like to solve the problems of duplicated content, finding canonical urls, and spider traps. My informal testing suggested that these problems weren't too bad for this crawl, mostly because of my approach of using a relatively shallow crawl of a large number of whitelisted domains. But for more general web crawls solving these problems will be essential.

Chris permalink

Simplest way I've found for finding dupe content is using a hash, but not ideal as even the slightest difference in page content will mean that they are no longer seen as duplicates – I find it useful enough at the crawler level though to discard dupes quickly.

For deeper analysis I have considered using the hash after boiler plate removal so that it only captures the main content (or some components of the page), but this still doesn't always capture crud content where there may just be a change in one keyword on the page; a more complex method – but very resource intensive – is to calculate the entropy between pages post indexing; I've used it when analysing some small to medium sized clients to highlight similar content but don't see it as being efficient when crawling.

Anyway much appreciate hearing about your experiences with AWS, as we are currently considering porting our own code to the cloud.

Bobby permalink

For identifying duplicate content, consider rabin chunking and storing the chunk ids (or store all the data indexed by chunk id)

ashish singh permalink

hey you can always give a link to your code on github so that people who are interested in learning things can take advantage of that i strongly suggest you that plus open sourcing also helps in improving the code and ideas sometimes. release

your code pls.

peter permalink

awesome write-up!

Manish permalink

Ignoring external links from whitelisted domains isn't cool.

Zheng Liu permalink

Thanks for the very nicely written article. For configuration management, I would suggest to look into salt (http://saltstack.org/). It is python based, and support remote execution on group of machines through 0MQ. I would also suggest to have the system running as daemon(s). Separate different functionalities into nodes, use some queueing mechanism (0MQ, RabbitMQ, beanstalk, etc) for the communication/coordination. This will allow you to handle politeness/retry, job distribution, prioritization, etc. in a much extensible fashion than file based approach. A well modulized system will make development and scaling much easier.

Michael Nielsen permalink

Thanks for the tips, I'll look into several of these.

Daeyun permalink

what can you do with the crawled pages? any cool projects in mind?

Jebus permalink

Thanks for this great write-up. Please don't post the code. It's not worth releasing to all the people who might just end up trying to abuse it. Anyone who can't already develop this shouldn't just get the leg up on your hard work.

Guru permalink

Great.Thanks for sharing your experience.

Xinxing permalink

Thank you for sharing the practice experience, very cool!

i_hack_sites permalink

Interesting read, i have experimented with some basic crawls of the Alexa top 1 million myself for various research projects. However, I used a $12 a month VPS and it took 2 weeks. Used Ubuntu, bash and curl. Not quite the same scale.

http://hackertarget.com/top-websites-analysis-alexa-rank/

David Mercer permalink

I think not using puppet or similar systems was wise, as at least puppet is terribly complex. Powerful, but takes a LARGE time investment to get going with.

I also second (third, fourth?) the ‘use a custom AMI image’ opinion. Your gut was right on that one. If you were crawling a much larger number of pages, a hand built image with just what you need in it would be ideal. Then you just need to update your image, and spin up new ones using that. Doing that with spot instances and using Elastic Block Storage rather than S3 would be perhaps the best way to go for performance and cost.

robots.txt handling is indeed problematic as it can be very difficult to get right for lots of webmasters….its format is, IMHO, ‘sucky’ to use the technical term.

Rohit permalink

Does you perform this exercise as part of any project? anyway thanks for sharing your experience, its great help.

Askinosie Dog permalink

This is brilliant. I would _love_ to see the code for this, to see how you stitched together so many technologies. You could castrate it in some way so evil people couldn’t use it, but I think it’s a sad thing to leave that code unpublished.

Karthik permalink

About the comment on Google: typically, the output of such a system is fed into three things ->

1. document storage which lets one query by URL, and by parameters like version, last changed etc. this is later read in a separate workflow by the parsing workers. The current version of the document is then mapped to the URL.
2. a document analyzer (HTML parser) which you have and the outputs of that go to. It can also perform diff on previous versions to see what was changed. This can be, with other header information, used to analyze how often this document needs to be crawled. The outputs of this go to:
3. an indexing system backend, which creates an index for words and URLs. Typically a Map – Reduce task.
4. task queuing / workflow, to initiate crawls on old/new URLs, which is already used.

Based on this, things which I think can we written as reusable pieces of distributed software are:

DHTs for task 1 and 3.
Workflow Library – Specific instance being your crawler – for task 2 and task 4
Worker – Mapper or Reducer, which works with the Workflow above – for task 2 and 3

When you have written them like this, it’s easy to build multiple distributed systems with them.

Dean permalink

Thanks for sharing! This was really interesting and useful. We certainly use python for our scraping/parsing stack but usually use wget or curl to actually download the pages as they do a good job with obeying robots, etc. Thanks for posting the links to the various packages; I will certainly keep these in mind for future projects!

Cheers.

Peter Murray-Rust permalink

Very thorough write up Michael. Thanks – might refer to this in the future.

Not quite sure why you are doing this. There *is* a need for open crawling of the web (outside Google/Bing, etc.)

Not sure that closing the source is a good protection against misuse. People will simply try worse ideas that will eat up bandwidth and servers. I want to crawl parts of the web – you can guess which – and the best solution would be an agreed public/Open resource of crawled material and sites which would reduce duplication , error and bandwidth rather than everyone crawlling.

Michael Nielsen permalink

Peter – You may find it interesting to look at Common Crawl, who do maintain an open crawl of the web, at http://commoncrawl.org/

Sergey Shepelev permalink

Thank you very much, Michael, this is a rare case when one really did a hard work before writing a post.

Please hear opinion on these points raised in your writing, hopefully you or other readers find any useful:
– In a longer crawl, how should we decide how long to wait between downloads of robots.txt?
For the most technologically advanced part of webmasters, proper behavior here would be to comply with Expires/Cache-Control headers. Although, obviously many would not provide such headers or have them set at ‘do not cache’, so polite crawler must impose an artificial minimum (say 5 minutes).

– The crawler currently uses a threaded architecture. Another quite different approach is to use an evented architecture. What are the pros and cons of a multi-threaded versus an evented architecture?
For one, it’s only a matter of IO, which in your case is a relatively small part of whole crawler. However due to lack of good built in concurrency in Python, any solution would be radically intrusive and either hard to implement and maintain (manual handling of sockets, epoll, timeouts) or depend on libraries that play with core internals of C stack (uwsgi, greenlet (which is further presented by libraries like eventlet and gevent)) or depend on Twisted which implements concurrency using robust Python generators but is a problem in itself for various reasons, e.g. not compatible with standard HTTP libraries. You could as well go in a totally different direction and extract all IO to a separate program that could be written using more appropriate language/runtime/libraries, e.g. C/libcurl, C++/Boost/asio, Go, Haskell, or god forgive me Node.js. I am currently writing such external IO program for another crawler and would like to thank you for helpful insight on total size limit of page.
Second, to answer directly your question, it greatly depends on what is implied by “evented architecture”. If you mean code written in callback style, then pros for threaded are ease of writing, understanding and debugging, OS threads naturally exploit multiple cores but that’s not a problem since we’re talking about Python, the only place where multiple threads executing Python code would at least not interfere with each other is lxml (or other C libraries releasing GIL), the only possible pro for callback style is it’s the only way to write code that natively maps to high performance kernel syscalls. If on the other hand by “evented architecture” you mean “uses epoll/kqueue/select”, than i should note that this does not imply absence of threads, as noted before libraries like uwsgi, eventlet, gevent and even Twisted (to

some extent) make asynchronous IO look like sequential code with some notion of light threads, but again, while heavily tested (on smaller loads) they by themselves could possibly be a problem because they switch underlying C stack.

– Logging is currently done using Python's logging module. Unfortunately, I'm finding this is not well-adapted to Python's threading. Is there a better solution?
Try logbook or stdlib/syslog + (rsyslog or syslog-ng). Unfortunately, you may find many libraries already use logging; logbook provides monkey patching for that. Be aware that syslog path will lead you through several max message limits.

– It'd be good to add instrumentation so it can be modified more dynamically, in real time.
A better definition of "real time" could help. If you mean to make downtime lower, this may be accomplished generally, in two ways:
1) stop taking more jobs, fork+exec(argv) to load new code, terminate as soon as current jobs are finished; this approach serves well for network servers, for various job workers life is a lot easier since we don't need to maintain open client connections.
2) really update code dynamically using `reload` built in, though it puts hard restrictions on how you should write the code and is generally very hard to implement right.

Again, if not too bothering because such question was not asked, i would raise imporance of Karthik comment about separating pieces of software. This gives many benefits, such as weapon of choice, better control on resources at the cost of synchronization and communication which seem to be solved problems with software like Linux Futex :), ZeroMQ, crossroads, gearman, RabbitMQ, [Twitter] Storm.

Michael Nielsen permalink

Thankyou, Sergey, this is a terrifically informative comment, which has given me much to explore. Incidentally, your comment on reloading matches my experience: I made some trial experiments in this direction, and found it quite tricky to get right, while maintaining a clean design.

weidong permalink

Do you have encounter memory leak in your crawlers when the multithread processes run continuously more than one day?
I have a test with running single nomultithread process, single-thread process and multithread process continuously more than one day, the memory they used was expanded all. And I have checked for over one week, but have no breakthrough.
So do you have the same situation, and if it's convenient share me your code?
Any help will be highly appreciated, thanks very much.

Ovidiu Tatar permalink

First off all thanks for publishing you project, great job.

To your improvements I will say maybe the language "erlang" will solve a lot.
here some topics that erlang support:

fault-tolerant, soft-real-time, non-stop applications
It supports hot swapping, so that code can be changed without stopping a system…

While threads require external library support in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which

removes the need for locks.
(Source : Wikipedia)

Reference:
http://www.youtube.com/watch?v=OpYPKBQhSZ4

If I have enough time I will create a crawler base on erlang.

By // Ovi

SK permalink

Its nice write-up. thank you

question – is this architecture is based on centralized database? i.e each instance of EC2 saves the information to centralized DB whenever crawler encounters parses specific information from pages?

Where is database located – at your local machine? or onto the EC2?

Regards,
Shruti

Martin permalink

In my opinion architecture should be centralized as it makes sense from financial point of view to use EC2 spot-on instances for crawling. Temporary (and cheap) spot-on instances will crawl for urls and it will save all results into central DB (RDS instance) as can be interrupted in any moment (read more about spot-on instances).

Main EC2 instance will check all finished tasks in central DB and then it will decide what should be done next (again DB table or key-value store).

I would not mind to use extensively SQS (amazon queue service) or S3 to make implementation easier.

Ehab Heikal permalink

By not releasing your code I had to rely on non-well behaved crawlers for my project that needed a couple of million pages crawled

Jeremy permalink

Is there a reason you used Redis instead of Amazon SQS for the frontier?

Michael Nielsen permalink

I'm familiar with redis, and know nothing about SQS.

kovalson permalink

could you show your threading crawler programm (or a part of if the threading part)
thanks
kovalson

Bill permalink

Great article.

If you had to constantly crawl the pages over and over for freshness, would you have a suggestion for doing this without having to download all the content again and comparing for freshness. This is especially important for dynamic web sites. I read you can send a get modified to the URL stored but not all web sites would store this given they have backend database.

Any ideas would be helpful, I can't imagine google is constantly downloading the contents of all we sites over and over to compare what is changed as this would require significant bandwidth and load on all the end servers.

Michael Nielsen permalink

Bill,

Glad you enjoyed it.

On freshness, see the comment a few paragraphs from the end about pubsubhubbub. Using that protocol should definitely help, since it's explicitly a protocol to notify other services of changes. It's built into many popular CMSs. Similarly, things like being RSS aware would make a big difference, since there are services which will ping you when an RSS feed changes. There's also an academic literature on getting fresh pages, which I've only looked at briefly. I wouldn't be surprised if there is some useful stuff in there, but I'm not deeply enough read in that literature to be sure.

Ivan Voroshilin permalink

Great job Michael and it's not the worst way to spend almost 600 dollars
I'm practicing inverted indices/crawlers at the moment and studying different architectures presented on the net.

1. Some researchers say that crawling/indexing fits into Map-Reduce stuff very well. I don't have experience with Map-Reduce however but consider this paradigm is better for Internet indexing where billions of pages. What do you think?
For example:
http://km.aifb.kit.edu/ws/semsearch10/Files/indexing.pdf
2. I'm trying to implement auto sharding for crawling/indexing internet pages but haven't found much information on how to implement it. Would be grateful if you know something about it.

Thank you,
Ivan

Michael Nielsen permalink

Glad you enjoyed it, Ivan.

Unfortunately, I haven't tried using MapReduce to crawl and index. While writing the crawler, it occurred to me that my crawl was within a preset set of domains, and MapReduce would offer a relatively easy way of approaching the problem of adding extra domains to the crawl. But I never implemented it.

James permalink

Hello, I am looking to hire a programmer to develop a crawler for me.

My direct email is hfgappts@gmail.com

Email me with your info and I will give the specific details and project outline.

James

mohiul alam prince permalink

Is this only 1.69 Tera Bytes of data ??? 250Millions of pages might have 16/17 Tera Bytes of data. If your data is only 1.69 Tera Bytes then per pages size is only 7KB. And its really absurd.

Michael Nielsen permalink

I used 20 instances. So it was more like 30 Terabytes of data (I don't remember exactly anymore, but it was in that range). I downloaded and stored the html of most pages without any need for truncation.

Even if I had only been storing 7kb per page, it would not be "absurd" as you so politely put it: (a) it's easy to think of streaming applications which don't require more than very short-term storage; and (b) the technical interest is in designing the crawler, not in dealing with storage, which would be a much easier problem.

Senad permalink

Just a fantastic article. I was surfing around trying to find an article on how to use PHP for multi threaded scraping, and I came across this. I'm thinking if it's worth learning Python merely for this purpose? Do you think PHP could work well with multi threaded scraping?

Jeremy McLain permalink

I think you may be confused about the utility of the Bloom filter. In your article you state "I used a Bloom filter to keep track of which urls had already been seen and added to the url frontier. This enabled a very fast check of whether or not a new candidate url should be added to the url frontier, with only a low probability of erroneously adding a url that had already been added."

You've got it backwards. Instead there is a low probability of *not* adding a URL that has *not* already been added. This means that by using a Bloom filter there is a chance that you will never crawl certain URLs because your bloom filter is telling you they have already been crawled when in fact they have not.

With Bloom filters there is a possibility of false positives: This means the filter is erroneously saying that an item is in the set when in truth it is not. So if you are using a Bloom filter to determine if you have crawled a URL and the bloom filter is saying you have. There is a possibility that you haven't and you will forever miss that URL.

It would be great if the Bloom filter instead has a probability of False negatives. Meaning the Bloom filter is telling you that the URL has not been added to to frontier when in fact it already has. This would give you the behavior that you want. But that isn't the case.

Michael Nielsen permalink

You're quite right, this was my mistake. Embarassing, especially given this. A better solution in this case would be to simply store all the URLs (which you're likely to do anyway, I merely used the Bloom filter for speed).

I'll amend the post.

Jeremy McLain permalink

No worries. I read your other post too but it wasn't until I wanted to add a bloom filter to my own crawler that I discovered that it wouldn't exactly give me what I was expecting. I might combine or replace the the bloom filter with a Trie of the crawled URLs but I"ll need to experiment with performance a bit.

Trackbacks and Pingbacks

1. Sysadmin Sunday 92 - Server Density Blog
2. How to crawl a quarter billion webpages in 40 hours | DDI | Ryan W. Lessard
3. Web performance - Weekend must-read articles #28
4. A Smattering of Selenium #111 « Official Selenium Blog
5. How We Built Our 60-Node (Almost) Distributed Web Crawler | Blog - Semantics3
6. CommonCrawl and PHP – The Intro
7. Gearman: distributed workers | DEKIBA

Comments are closed.