Big or small,
proprietary or
open source,
Web or intranet,
it's a tough job.

Why writing your own Search Engine is hard

ANNA PATTERSON, STANFORD UNIVERSITY

There must be 4,000 programmers typing away in their basements trying to build the next "world's most scalable" search engine. It has been done only a few times. It has never been done by a big group; always one to four people did the core work, and the big team came on to build the elaborations and the production infrastructure. Why is it so hard? We are going to delve a bit into the various issues to consider when writing a search engine. This article is aimed at those individuals or small groups that are considering this endeavor for their Web site or intranet. It is fun, but a word of caution: not only is it difficult, but you need two commodities in short supply—time and patience.

SUPER-SHORT SEARCH ENGINE OVERVIEW

OK, let's do it. Let's write a search engine.

A crawler gets the Web pages off of that pesky Web and onto your beautiful disks. You'll need lots of disks.

Then you need to index these pages—say which page has which words. This will tell you that Janet Jackson was found on the www.superbowl.com page. Usually, indexing happens locally on the disks where your crawler dumped these Web pages. Hey, why move them?

In most architectures, now you need to merge these indices so that you have one place to go to in order to find all the pages mentioning Janet Jackson's Super Bowl performance. When you merge all these small indices, the final index will be so big that it won't fit on one machine. This means that you'll have to merge these small indices in such a way as to split the final big index across many machines.

Now you are ready to serve queries? Wrong. Now you build the runtime system that gets users' queries, retrieves the results out of the index from the right machine(s), and re-ranks them according to the query. All this, while people are drumming their fingers on their desks waiting—hopefully, lots of people and, hopefully, not enough time for much drumming.

Why writing your own Search Engine is hard

RESOURCES

People talk a lot about the thousands of machines needed to build a search engine. This sounds very scary. All search engines, however, started with a lot more thought and design than they did machines. So let's see what is fact and what is fallacy.

Bandwidth. Legend has it that venture capitalists used to buy hard disks for young entrepreneurs to prove that their ideas would work. Now disks are cheap—but the new bottleneck is bandwidth. Usually that takes capital. You need this bandwidth to get the pages from the Web in the first place. The "CPU-ness" or memory of the machines that you use doesn't really matter. All that matters is how much bandwidth you have (can afford) and can use because crawling is not a CPU endeavor—crawling is a bandwidth monster.

There are lots of ways around this issue, but the most useful is to realize that you won't get the indexer and the servers working right (if at all) for six months, anyway, so crawl slowly and index what you have as you go along. Bugs will show up in the later phases, so the lack of pages won't be the thing holding you up; instead, it'll be those nasty bugs slowing you down. So crawl continuously at whatever rate you can afford (down to 1-megabyte DSL), and the rest will take care of itself. By the time you have a search engine that works on the pages you have and can keep up with your super-slow crawl, perhaps you'll be in a position to afford big bandwidth by raising capital.

Big bandwidth is usually found at a collocation facility (or colo). I want to warn against this if you are a super-small company. Get the bandwidth to the office! If you have a small team, the last thing you can afford is people on the highway all day long running to the colo. This is another big reason that I recommend small bandwidth for the development phase. You can't afford the loss of a person for half a day to go exchange a disk. Another reason to avoid a colo is that it's hugely expensive. Just throw the stack of machines under your desk and consider it a space heater.

CPU Issues. People argue all day about which types of CPUs to use for which phase of a search engine. Most people argue that the ideal is to get stupid CPUs for crawling and fast CPUs for indexing and serving. Why is this?

You don't need a lot of thinking to do crawling; you need bandwidth, so any old CPU will do. For indexing, you are doing a lot of I/O and a lot of thinking/analyzing the page, so the bigger the better. At serve time, you're going to need to re-rank the URLs in response to a query, so again, the bigger the better.

Since you're writing the search engine yourself, however, it has to be one size fits all. Most indexing algorithms worth their salt will probably peg any CPU. So the same advice goes: it doesn't matter, get what you can afford; the bugs you write will slow you down more than the cheap CPUs. If you have to look around your local Fry's or CompUSA for CPUs, however, more on-board cache will be key for the indexing algorithms because more of the page will be kept onboard.

If your algorithm doesn't peg a Pentium 4, then rethink the game plan of building a better search engine, because yours will not be the one that wins.

Disk Issues. SCSI is faster, but IDE is bigger (and cheaper). If you are writing a search engine yourself, use IDE. This will save money in many ways. You get bigger disks, so one machine can hold 1 terabyte for IDE disks easily, but this just isn't the case for SCSI. Secondly, SCSI disks are a lot more expensive—also not a good idea for four guys in the garage.

At runtime, you'll be disk-bound. You have two tasks: get the index entries off disk and re-rank these for relevancy. For getting the index entries off disk, you might think the faster the disk the better. But users will not see the performance increase you get from SCSI in the disk transfer rate, because it takes a lot of practice with the search engine end game (the runtime architecture) for this difference to be an issue. Instead, use parallelism and multiple cheap disks to achieve this speed-up. This will still save you money in buying fewer machines and give you practice with the key tool of search engine architectures—parallelism.

Ah, but SCSIs are hot-swappable, you say. Get over it. Remember, no colo. You cannot afford it and you don't want it. So if you're worried about disk failures since you picked your disks out of a Dumpster, then my advice is

don't screw the covers onto your machines and don't use four screws per disk. This makes IDEs pretty easy to repair, but certainly not hot-swappable.

Storing Files. Old-fashioned file systems used to have a limit on file size—some of them had a 2-gigabyte limit. These file systems also used to have an issue with storing lots and lots of files in one directory. For these reasons, the prevailing wisdom has been to crawl a bunch of URLs and stuff them into one big file (up to the limit) and then start on the next file. Even though current operating systems don't have the same number-of-file restrictions they used to, putting lots of pages in one file is still a good idea. Stuff them in—up to the limit of good performance of your operating system.

Why? When indexing, or laying down the crawl, a big continuous file saves a whole lot of disk seeks—the fewer files the better. Disk seeks will kill you even if your disk transfer rate is high. You cannot afford the time to seek to a file to process a Web page. Web pages right now average around 10 kilobytes per page (I'm such an oldtimer, I remember when they were 2 KB, and others remember when they were 1 KB). You don't want to seek to a disk to read 10 KB when we are talking about millions, if not billions, of Web pages. Essentially, this will almost double your processing time, as well as fry your disks from the Dumpster.

While you might think that it is conceptually cleaner to store one Web page per one file, this will become a management pain—and it will also slow down your processing.

Networking. With real estate they say "location, location, location." Well, a good search engine rule that I've learned the hard way is: Don't use NFS. Don't use NFS. Don't use NFS (network file system). NFS might seem like a great idea for an index that won't fit on one machine (and yours probably won't). It seems like the perfect solution. If you put the index on multiple machines, then NFS will make it seem like your index is on one machine. Sound good? That way you don't have to do or learn any networking yourself. Wrong! You'll have to do real distributed systems work for the serving architecture, anyway, so get it over with and do the work now.

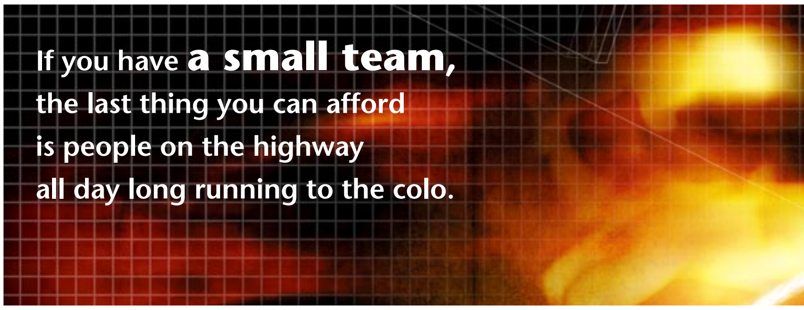
Current NFS implementations can't stand the punishment inflicted by the runtime system, or the indexing phase without using "spendy" specialized hardware.

In the indexing phase, you will get corrupted indices as you try to do lots of networked writes. Ask the contributors to NFS in Linux and they will tell you the same: not ready for serious punishment.

Next, using NFS in the runtime system, you will get machines that don't have fault tolerance. If one of the NFS'd machines is sick, then the rest just seize. Not good.

SOFTWARE TO WRITE/GET

Crawler. If you don't use an open source crawler, my advice is a super-simple multistep crawler. This is very important advice that will cut months off your development time, so if you ignore everything else, don't ignore this.


If you have **a small team,**
the last thing you can afford
is people on the highway
all day long running to the colo.

If you want to build a crawler yourself, then first get a list of URLs that you want to seed your crawler with (these need to be good starting points for exploring the Web—dmoz, Yahoo...). Then write any simple program that will get them. For instance, `(dolist (y list of URLs) GET y)` is essentially all you need.

When you get these pages, analyze the outgoing links in the pages to create a new list for your simple crawler and go get those. What about duplicates, you ask? `Sort | uniq` on Linux will do this for you; otherwise, I think you can handle it. This takes care of duplicate URLs, but what about duplicate content? My advice: find those at serve time.

The really hard problem with crawlers is to perform dynamic duplicate elimination—eliminating both duplicate URLs and duplicate content. With the system that I described, we've avoided getting a Ph.D. dissertation and instead have some piece of code you can hand off to your youngest sibling.

Indexing. Next you need to churn through the pages and build an index. This is tricky. Just don't do anything wrong, as the saying goes. One false step and those billions of pages are going to take too long to process and your 1-MB DSL crawling line is going to seem fast.

There is a major field of study about the different things to index on. Don't get a Ph.D.; just index on words. Words are what people search for; they don't search for N-Grams or letters or PTrees or locations in streams, so any other method other than the simplest will make you seem clever. But, hey, writing your own search engine is hard enough. Save what cleverness you own for ranking.

Two other pieces of key advice: First, just index the data you need to serve your kind of search results and do your kind of ranking. Don't write down everything and the kitchen sink—save that for when you go ultra-commercial. The first item of business is getting something presentable up. Correction—start by getting something up. Find out what went wrong and fix it.

Second, do not get attached to the "index format." The hallowed "index format" is not the end of the search engine, it is just the beginning. It is a tool to see results, so change it and change it often. Play with it, and you and your team will be on a winner to be able to improve search results quickly.

Why would you need to add things to the index? Perhaps you've just decided that it would be good idea to keep whether the indexed word is in the title. So now you need a space to annotate this fact. You might have other ideas that mean adding more data to the index.

Let's say that you've worked in the long dark until the proud day when you type in a search for *bug*, and pages that mention *Britney Spears* but not *bug* appear. All kinds of things like that happen. Do a dance—you're almost there. Just keep fixing.

A last word of advice: when in the development phase, keep a disk-based index architecture. You are not getting lots of traffic, you want flexibility regarding which items to place in the index, and mostly you want a happy team. A happy team does not fight over bits. A happy team does not see whose new feature is in and whose is out because there isn't enough memory. Buy disks, play with features, and have fun.

Dynamic versus Static Ranking. Don't do page rank initially. Actually don't do it at all. For this observation I risk being inundated with hate mail, but nonetheless don't do page rank. If you four guys in your garage can't get something decent-looking up *without* page rank, you're not going to get anything decent up *with* page rank. Use the source, Luke—the HTML source, that is.

Page rank is lengthy analysis of a global nature and will cause you to buy more machines and get bogged down on this one complicated step—this one factor in ranking. Start by exploiting everything else you can think of: Is the word in the title? Is it in bold? etc. Spend your time thinking about anything you can exploit and try it out.

This again will give you the freedom and make you develop an architecture good for adding things and trying them out. This will become invaluable later.

Serving. Runtime systems are hard. Algorithms are hard. The hardest part about a search engine is that you have to do both. They have to work together, and both parts are absolutely critical.

At serve time, you have to get the results out of the index, sort them as per their relevancy to the query, and stick them in a pretty Web page and return them.

Now that you have the list of URLs, you have to rank them according to your **relevancy algorithm.** This has to be fast. People are waiting.

If it sounds easy, then you haven't written a search engine. Remember, first, that some queries have more than one word. This means that you have to intersect the index entries for the two words. My advice is to have them presorted in some canonical URL number order so that you can view the two (n) index entries as two stacks and pop until the tops are equal, in which case, you win the prize—the URL is in both index entries. These sorts of computations have to be run at query time and they need to be run quickly, so think hard about how you are going to do intersections.

Next problem, query time ranking. Now that you have the list of URLs, you have to rank them according to your relevancy algorithm. This has to be fast. People are waiting.

The fastest thing to do at runtime is pre-rank and then sort according to the pre-rank part of your indexing structure. This often results in generic (read not the best of breed) ranking algorithms. You need to take into account the actual query when you are ranking. Thus, you need some data in your index to help take the query into account and re-rank your a priori ranking quickly at runtime.

For the basic runtime architecture, you will find no end to people willing to argue about the "appropriate" way to do it. In practice, there are two basic disk-based methods and other memory-based methods. Since we're doing this on the cheap, we'll cover just the basic disk-based methods.

The first major method is this: after indexing the files locally—where your crawler deposited them—leave the little indices there. Yes, do nothing more. This means at runtime you ask all machines that have answers for the appropriate query to get back to you ASAP. You drum your fingers as long as you are willing, then gather these little lists into a *big* list and sort this list for relevancy.

The other method is to gather all results for a particular word together in a big list beforehand. Then when a query arrives, go to the appropriate machine, get the list, and then sort for relevancy. Without showing my bias too much, look on the bright side: for rare queries or obscure words, these are equivalent.

NO ROOM FOR ERROR

When you look at all these steps and all the complications, this process is rife with things that go can wrong. The hardest part about writing a search engine is that you're going to process billions of URLS and serve millions, if not billions, of queries. This does not leave a lot of room for error. One super-linear algorithm applied over the wrong-sized list of items and you are sunk. One lock inside another lock and you are sunk. There will be no code paths not explored. All of those comments in your code, which print out errors like "This will never happen," will happen.

When you think that you are done, there is still the load balancing, the caching, the DNS servers, the ad service, the image servers, the update architecture, and (to take off on a familiar tune) a cartridge in a tape drive. Oh, and if you would like to hear from someone who's already done it, read Mike Cafarella and Doug Cutting's article, "Nutch: Open Source Web Search," on page 54 of this issue.

Sadly, the biggest thing that goes wrong while writing your own search engine is running out of time. Real life often interferes and forces you to end your quest. In that case, cheer up; once the search bug gets you, you'll be back. The problem isn't getting any easier, and it needs all the experience anyone can muster. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

ANNA PATTERSON (anna@cs.stanford.edu) has written two search engines. Most recently she wrote the biggest index in the world by indexing 30 billion Web pages at the Internet Archive at Recall.Archive.org. In 1998 she coauthored a search engine at Xift, where she was a founder. She received her Ph.D. in computer science from the University of Illinois at Urbana-Champaign and was a research scientist at Stanford University, where she worked on phenomenal data mining. She is also the mother of three preschoolers, who let her hack sometimes.

© 2004 ACM 1542-7730/04/0400 $5.00