Slide 1 - "Cover page"

Hi, and welcome to my talk on "Building a vertical search site"

Slide 2 - "Just the Facts, Ma'am

Krugle is system for using search to help developers.

I'll talk about the details of what we do in the next slide, but the key points are that we built three different products in a short amount of time, and on a limited startup budget - and the way we did that was to use almost exclusively open source.

Along the way we learned some things about both architecture and open source, and that's what I'll be covering today. I'll focus mostly on the search aspects, specifically how we use Lucene.

Slide 3 - "Three Faces of Krugle"

I'd mentioned that we built three different products, or what we call the three faces of Krugle. It's not as schizophrenic as it sounds, since all three build on the same core technologies, and they provide the same key functionality for search-driven development.

What do I mean by "search driven development"? It's using search to help solve development problems, by making the right information available at the right time. The reason I bring this up is that it's important to understand what problems we're solving, and thus what functionality we need. And from that functionality flows the architectural requirements.

I think it's a too-common problem that we, meaning developers who are making technology decisions, focus on an architecture that we like. And that then in turn defines what's easy and what's hard, so the architectural decision winds up having a lot of say about functionality. And that then in turn defines what problems you can or can't solve.

We tried to flip that around, and make sure we focused on the problem space first, and used that to drive features and architecture. Now we did, early on, make the decision to use open source as much as possible. This wasn't an architectural decision so much as one based on cost and time-to-market, but it did wind up influencing many of the subsequent architectural choices.

Slide 4 - "Krugle.org free public site"

So what's the problem we solve with the public site? There are three:

First, there's a lot of open source code out there, but it wasn't easily searchable.
Second, there's a lot of open source projects too, but it's hard to find what you're looking for.
Third, there's a lot of technical information, but it's sometimes hard to find the good stuff.

The common word in all three of the above is "lot". We knew we'd have to support fast search on many bytes of all three types of data.

Slide 5 - Krugle.org demo

<switch to web site>

When I do a search on "lucene" in code, I'm going to get a lot of hits.

But I'm really looking for input on performance, so I can restrict to just comments that mention performance.

But I see a mix of languages, and I'm only interested in Java. And I don't want to get hits in Lucene itself, just code that uses Lucene.

Then I do a search on "lucene" in projects, where I also want it to be about databases.

So there's the requirement of fast, flexible search over lots of data. And you can see that we also have a requirement to provide a very rich, dynamic browser UI for exploratory search. So that also creates architectural requirements. For example, portlets are an interesting technology, but didn't seem right for this particular UI. We were going to need to write a lot of Javascript, where we were in control of the presentation layer. And we'd want to be able to access the services using HTTP requests.

<switch back to PowerPoint>

Slide 6 - Krugle.org Architecture (web)

We've covered the problem space, and touched on functionality, so let's start talking about the actual nuts-and-bolts of the architecture.

This is a pretty standard design, so I won't spend a lot of time on it.

There's the public-facing stuff, which we call the dirty side of the system. This consists of the top three pieces you see here - industry-strength firewalls from Juniper, load balancers from F5, and then four mid-range web servers.

The load balancers support sticky sessions using IP addresses, so a user keeps getting sent to the same web server once they've established a connection. This would cause problems if we had lots of AOL users coming from the same IP address, but fortunately that's not the case.

When we first went live, we only had two web servers, and we were exec-ing Perl code. This didn't work so well. Things backed up pretty quickly under load, mostly due to not using mod_perl. Though we made things worse by generating authentication tokens using "real" randomness, which means we quickly ran out of truly random bits. So then logins stacked up as the system waiting for cosmic rays to generate more randomness. Pretty quickly we realized that we didn't need that level of security.

The problem of having only two web servers was that when we had to take a web server out of production to fix something, that left us with just one live server, which is something you want to avoid if your ops team is going to sleep well at night.

Slide 7 - Krugle.org Architecture (API)

The API layer is more interesting, as here we get into things specific to the service that we provide.

As I mentioned earlier, requests to the API tier come from the web servers over HTTP. This "Krugle API" is reasonably RESTful. Requests to read or query "things" are received as HTTP GET requests, and the response is XML. A "thing" is a file, a project, a user-generated note, a codespace, and so on.

This REST approach has worked well for us, in general. We've benefitted from having these loosely coupled web apps, and it's been pretty easy to integration something like Solr into the mix.

There have been two significant downsides to this approach.
The first is that during development, you often wind up needing to hit one of these APIs to really test out your code. So we have a "development" API server that's available, but only if you're on VPN, and sometimes it's not available. People are changing code, so things can and do break. And when that happens, dependent services become harder to work on.

The other is that automated integration testing is also harder, for the same reason. We wind up baking a lot of pre-canned data into the build process so that we can deploy the system to a webapp container with some reasonable set of test data, versus having dependencies on the development cluster.

So how did a Krugle API request actually get processed?

There's a box on this diagram labeled "API Cluster". A cluster consists of one API server, four code search servers, four page search servers, and a filer. All of these machines are commodity hardware (that's a code phrase for "really cheap") running Red Hat Linux.

We have multiple clusters of servers, which simplifies testing for our data pushes and more importantly makes it easier to upgrade the software. During the first year you can expect to be quickly iterating on the design, which means you often have to revise dependent pieces of code throughout the system. Keeping things in clusters made that easier, at the expense of some under-utilization of hardware.

Currently we have four clusters. There's the live cluster, which is handling all the requests coming from the outside. Then there's a fail-over or standby cluster, which we use to swap in servers if any one fails on the live cluster. There's also a "deck" cluster that's in testing, and will be the next live cluster. These three clusters are constantly being rotated, as we release new versions of the software and do data updates.

Since the design has settled down, we're moving away from this architecture, to something more common where there's a pool of similar types of servers - for example, multiple API servers that are round-robin-dispatched to by the load balancer. We can then do the same thing for the code and page searchers.

Slide 8 - API Server

The API server a reasonably fast machine with some redundancy - things like dual power supplies, NICs, and spare drives. It turns out the API server isn't the bottleneck for performance, given how we use remote servers to do the searching. So we don't need a fire-breathing box here, just something that isn't likely to die.

On this API server, all of the services are running as Java webapps inside of Resin. Yes, we could have used Tomcat, but at the time when we were making this decision back in 2005, there was significant discussion on the mailing lists about Tomcat having stability problems. Mostly these seemed to be around it hanging at random times, under load. Resin got high marks here, so that's what we went with.

As you'll see later, we're using Jetty for part of the enterprise product, and we might wind up switching to it for everything. One of the advantages of Jetty is that it's better integrated with Maven and Eclipse, which we use internally.

But in general Resin has done well for us. Occasionally when we're having problems in the back end due to memory pressure, Resin can go into a mode where it returns bogus responses to requests - for example, we'll get an HTTP 200 status code, but no content. And that makes the middleware Perl code, the XSLT and Javascript all very unhappy.

And surprisingly enough, the API server typically isn't the bottleneck, it's the remote search servers. If we threw enough search servers at the problem, I'm sure that eventually we would be limited by this one server, but that hasn't happened yet.

One more bit of info...we monitor this API server pretty carefully, using automated tests that generate Big Brother alerts when things get out of bounds. By "things" I mean how many active threads are running, the responsiveness of the system to queries, and pressure on the heap. This last one we calculate by parsing the Resin JVM log to determine the number of full GCs in the past ten GC events...when this gets too high, then it's a good sign that the situation is about to turn ugly.

Slide 9 - The Life of a Files Query

Going back to the code search we ran, the /files webapp service uses bits of Nutch to leverage the Nutch support for distributed searching. The query gets sent out, using Hadoop RPC, to four code searchers running a modified version of the Nutch remote searcher code. Each searcher is a 4GB server with two fast disks, so we run two 32-bit JVMs on each box, and two remote code searchers.

This lets us split our code index into 8 pieces, each with more than 5M documents. We randomly distribute the documents, to avoid skewing the inverse document frequencies. If, for example, we had one of these code searchers with nothing but Java source, and another searcher with only one Java file, then hits from the first server would get lower scores than what we want, and that one file from the second searcher would get a very high score.

Lucene's remote searcher implementation takes care of adjusting for this potential skew, but Nutch doesn't. If you take care when building the indexes, then this isn't a problem, and makes things faster. You can avoid another remote call required to gather the info you need to adjust for unbalanced document-level term frequencies.

So getting back to that /files search request. It was sent out to each of the eight code searchers. They've returned their top N hits to the files webapp running on the API server. The files service picked out the top hits, then made another request back to some number of the code searchers to get more info, like summaries, for the hits that made the cut.

Then the files service uses Dom4J to turn these results into a standard XML response like what we saw, and that gets returned to the caller. We also cache this locally, using ehcache.

Finally, in order to display a file when the user clicks on a hit, the files service has to return the contents of the file. Here we cheat a bit. When you make a read request to the files service, and specify the file URI, what you get back is a bunch of meta-data about the file. Things like the license, the fingerprint value, and so on. One of these bits of info is the filerURI. This is what you use to make a secondary request, to get the actual contents of the file. This request gets proxied in the middleware to an instance of Lighty running on the filer server. This way we can use Lighty to efficiently serve up static content, without bogging down the API server.

Slide 10 - Page queries

For searching our 40M page tech page crawl, it winds up being very similar to files service. We using a pretty stock version of Nutch here, other than converting the Nutch results into our standard XML response format.

Page query results, like the files service, are cached on the API server using ehcache.

For the remote page searchers, we only run one JVM on each server, and we split up the index differently. Here we have one fast disk for the index, and a bigger, slower drive for the actual page data.

Slide 11 - Search Hardware requirements

Now why did we go with 8 code searchers running on 4 servers? We set up a load test, and tried to figure out where the performance elbow existed. The question is when does adding servers stop improving performance significantly, for our target index and load? And for us, this was what we wound up with. The general rule of thumb seems to be that you want to have less than 10M documents per index, but that can vary widely. Why is that?

Note that I said "target index" previously. One of the changes we made, that let us get away with only 4 servers, was using the same technique on source code that Nutch uses on common words. If you leave in common words on a web page, your index size gets bigger and your search performance drops. But if you index combinations of common words, then you can avoid this problem. We did that type of thing for code like "i = 0", and that was a big win.

We'd get an even bigger win if we sorted our index by the static score we have for each file, and then do early search termination. That's a contrib that Doug Cutting made to Lucene a while back, but I haven't heard too much about people using it - or at least not using it successfully. Since we're currently fast enough, that's on the back burner.

We could also improve speed by combing more of our fields into combo fields. Because we parse the code, we wind up with multiple fields for each file. For example, there's a code field, a comment field, a function call field, a function def field, a class def field, a filename field, and so on. When you do a search, this currently generates a very big query. If we combined fields that used similar analyzers, we could have a more efficient query at the expense of a slightly bigger index.

And finally, Nutch doesn't support replication currently, where the index is duplicated on multiple remote search servers. But we could easily implement that ourselves, by using round-robin dispatching from pools set up in our load balancer, where each pool has N instances of code searchers with the same index.

Now my point here isn't to dive into the details of Lucene indexing. It's to point out the very big dependency your architecture and scaling requirements will have on your index. And not just the index size in terms of document count, but how it's organized, and what techniques you can use to improve the speed of the search.

And finally, it's hard to know where problems will pop up. The current performance bottleneck for us in code search is the generation of summaries. This is because all of the code searchers get the actual file data via an NFS cross-mount to the filer. Given the size of the data, we didn't want to keep it on the local disk, but this means that multiple threads from all 8 code searchers can be hitting the filer at the same time during summarizing, so we wind up being I/O bound by the filer disk seek time. We could shift the bottleneck to a different location by putting the code on the code

searchers, or by hooking up a faster and more expensive storage area network.

As a side point, if I was going to do this again, I'd have each box configured with four fast SATA-II drives from Western Digital, four dual-core CPUs, and 8GB of RAM. Then I could run four JVMs on each server, and get equivalent performance with half the number of physical servers to rack, cable, maintain, and eventually replace.

One of the things we learned during performance work was that you wanted to avoid having multiple searchers using the same spindle, as that led to lots of contention, and thus lots of extra seeks, which killed performance.

I know so of you are wondering about keeping the index in RAM, and thus avoiding any disk issues. We tried that on a server with lots of memory, using Lucene's RAMDirectory, and were surprised by the results. Once the file system buffers are loaded with the index, a RAMDirectory or a Memory mapped directory was about the same speed as a vanilla FSDirectory. In fact, without a lot of extra memory, things were actually slower. It seemed as though the RAM directory approach wound up chewing a lot of memory to instantiate all of the Java objects containing the data.

Slide 12 - The Life of a Projects Query

Beyond code files and tech pages, there's a separate set of services that basically are front-ends to instances of Solr webapps that are also running inside of Resin.

When the projects service gets a request, it converts it into a standard Solr query, forwards it to the Solr webapp, then converts the response into our standard XML response and sends that back to the caller.

This seems inefficient, but given the query rate, the efficiency with which Resin handles local HTTP requests, and the performance of Solr/Lucene, this hasn't been an issue for us. We're using an index with about 150K entries, but I've heard stories on the list of much bigger indexes running without performance issues under high load.

Solr has been good to us in several different ways. It's easy to set up and get running, especially with the admin UI that gives you a view onto the index and lets you easily run test queries.

There are lots of useful analyzers that you can easily configure, using Solr's support for a Lucene index "schema".

It's never crashed.

And Solr also makes it pretty easy to do safe and efficient updates to the live index.

We've run into four challenges with Solr.

First, it wasn't possible to embed Solr into an existing webapp. There have been contribs made that now enable this, so that limitation is going away.

Second, Solr didn't support distributed searchers. We want to put the project meta-data with the files that belong to it, on the remote code searchers. But this requires being able to send a Solr request out to multiple searchers, and merge the results. This would be like what we do using Nutch currently. Same as with embedded Solr, there's a recent contrib that adds this functionality.

The third issue isn't really a Solr problem. Solr uses the Lucene query parser, and we wound up

having to spend time figuring out how to "escape" words and characters that a user might enter which would confuse the query parser. We wanted to still have '+' and '-', but we didn't need to support grouping of queries using parenthesis, for example. I know Hoss has a more forgiving parser that's part of his DismaxRequestHandler, but that wasn't available when we were working on this problem.

We also ran into problems getting HTML data into Solr, due to all of the escaping/unescaping that needs to happen for correct round-trip handling. Solr requests and responses are XML, so if you have markup data inside of the Solr XML wrapper, you need to escape it so it doesn't confuse things. But our services return XML too, and the XML we return needs to be processed in some cases using XSLT, so we had fun getting all of the conversion steps set up properly.

Slide 13 - Krugle.org Architecture (CPI)

A key point to remember here is that we've got three main sources of data that we use for our public site.

There's code, projects, and tech pages, and we handle each in a different way.

The diagram on the screen is for the page crawl, since the other two pieces, while interesting to me, don't really demonstrate exciting uses of Apache software.

For the page crawl, we use an 11 machine cluster running an older version of Nutch. It's version 0.8.2 with some customizations, and Hadoop 0.9.2. Why are we on an older version? Well, more than once we got bit by really bad bugs when updating to a newer version that "only had bug fixes". So last January we made the decision to stick with what we had, because it was good enough.

There's a lot of info on Nutch and Hadoop at the Apache web site, so I'm going to focus on the customizations we made, and the things we learned while using Nutch for a medium size web crawl.

First, a quick overview. A "crawl" in Nutch consists of multiple loops. A loop is a sequence of using the Nutch crawl database, the crawldb, to generate a sorted list of pages to fetch, then fetching some number of these, parsing the resulting pages, scoring the pages, extracting outlinks, and updating the crawldb.

When we've crawled enough, whatever that means, we index the resulting pages, generate a set of four indexes, and save these off to a filer. Remember how I mentioned that we'd run into some really bad Nutch and Hadoop bugs? Well, this filer is our insurance policy, and it's saved our butts more than once. That includes incidences of friendly fire, like when somebody accidentally deleted everything from the Hadoop distributed file system.

Now our goal is to crawl the "technical web". This is the subset of all 20 gazillion pages out there that contain technical information which we think would be useful to a developer. The way we do this is by analyzing each page, using some fairly straightforward techniques, to decide how "technical" it is. We use term vector similarity, where the models we use are the result of harvesting training data from various sections of Wikipedia. Using this approach, we can come up with a number of different classifiers, for different technical areas, and thus derive a more accurate score for a page.

That page score can still be wildly wrong, but because this score flows into the OPIC algorithm, individual page errors aren't such a big deal.

I'll talk about the OPIC algorithm in a bit, if there's enough interest. But key point for us is that each page's "tech score" flows via outlinks from that page to all the other pages. And for unfetched pages, this score then is used to sort pages for fetching. We typically put a pretty tight cap on the number of pages we fetch during each loop. By doing this, we get a more focused crawl, at the expense of our overall crawl rate in terms of pages per minute.

The major crawl problems aren't related to Nutch, Hadoop, or our operations. It's dealing with things like honey pot sites that suck a crawler in to an endless maze of artificial links. Or really badly broken sites that trickle multi-megabyte PDFs back to us at 5 kilobits/second. We seem to come across this latter problem more often than I was expecting, probably because there are a lot of wanna-be geeks running their "Big House of Code" server in a basement with a dial-up connection.

So it often becomes a personnel issue - how do you find that anal retentive crawlmeister who actually enjoys baby-sitting the web crawl for days on end? In our case, one of the developers made the mistake of taking a bio break at the wrong time, so he got the raw deal.

Slide 14 - What's OPIC?

If you're going to use Nutch, you need to understand at least a little bit about OPIC.

It's the "on-line page importance computation" that Nutch uses both during the crawl, and as the final static page score. In theory this is an incremental link analysis score that converges to something similar to a PageRank score if you recrawl enough. In practice, Nutch's implementation of this has some serious flaws.

For example, in a stable page crawl, page scores tend to "leak" out of any leaf pages. So the scores of pages continue to drop. Now since they're all dropping together, that's not so bad. But then when you inject new pages, these pages have scores that wind up being much higher than the older pages.

OPIC is also very sensitive to link farms, As you add new pages, the total energy of the web graph keeps going up. And so this winds up fighting with the leakage from leaf pages, which means that you wind up with spammy, highly linked pages having ridiculously high scores, and leaft pages have very low scores.

If you aren't recrawling, and you can stay away from link farms, then it's good enough to help guide the crawl. And the resulting scores are good enough for us, but we wind up having to essentially start each recrawl from scratch.

Slide 15 - Krugle.org Partner Sites

This is the second face of Krugle, where we provide code search for partners who have developer networks or communities and code that they want to make searchable.

These are hosted by us, with customized look and feel. In some cases we restrict searches to the subset of projects hosted by or associated with the partner.

Each partner has their own set of challenges:

Sourceforge obviously has the most projects, and the biggest traffic.

Amazon has two types of projects, some of their own, and some hosted elsewhere that use Amazon web services. They want these searched as one set of "Amazon plus friendlies" projects.

IBM developerWorks has rapid update requirements. So they send us regular updates, via an HTTP feed, for new or modified projects. These need to be processed and pushed to the live site.

Slide 16 - Krugle.org Architecture (partners)

Our partners make use of some specialized APIs provided by Perl code running in the middleware.

These requests first get round-robin dispatched to one of three LightTPD servers that use mod_cache to create a high performance cache. This
significantly reduces the load on the API server, by up to 95%. For example, instead
of 20 requests/second from a partner hitting the API layer, it's only 1 request/second.

Requests that are cache misses get sent on the web tier, where mod_perl executes Perl code that "wraps" our low-level APIs. Because of the standard API to services, it's easy in Perl to create customized functionality on top of these APIs.

Slide 17 - Cache as cache can

Caching is clearly a big topic, so I won't go into this in depth. I'm just going to point
out the many places where caching does occur throughout the system.

We've got the automatic file block caching that Linux does for us. As I mentioned earlier, this works surprisingly well for keeping key parts of a Lucene index in memory. In fact I know of one major company that warms up their Lucene-based searchers by cat'ing the index directory to /dev/null, as a way of forcing it all into the file system cache.

We cache search results at the service level in the API server using ehcache. And Solr has its own cache for queries.

There's also the LightTPD-based cache for partner APIs that I just mentioned.

And the web browser UI caches results during a user session, to avoid re-fetching content.

One problem that happens is when you start to depend on the cache to achieve target performance levels, and then the cache goes down or needs to be reloaded. Suddenly the back-end system gets hit with a huge load spike, and as we discussed when things start backing up, they back up in a hurry. So whenever possible we try to gracefully age the cache entries.

Now with all these different caches, keeping them in sync could be a problem. But luckily for us, having caches that are out-of-sync generally isn't a problem, because the browser UI gracefully handles cases where an expected resource suddenly isn't available. And that's the most common issue that occurs. So rather than worrying about 100% consistency, we handle it downstream, which winds up being a lot easier.

Slide 18 - Krugle enterprise server

Finally, we're at the third, and to us most important, face of Krugle. That's our enterprise product, which we've been working on for almost a year. It went into trials in May, and recently was

released for general availability.

The enterprise product provides the same type of search-based functionality as the public site, but inside a company's firewall. So there are additional data sources - files and comments from internal SCMs, as well as project meta-data defined using what

The servers we run this on have a fast 150GB Western Digital drive, 4GB RAM, and two dual-core CPUs.

Slide 19 - Krugle enterprise demo

<switch to web browser>

In some ways it looks very similar to the public site.

But you can see there are three more search channels across the top here.

The three on the right are the same as for the public project, then there are three more on the left, for internal code, internal projects, and SCM comments.

<switch to PowerPoint>

Slide 20 - Krugle architecture (enterprise)

On the public site we've got probably 20 servers actively handling requests. But for our enterprise product, this all needs to fit into one box. So what all did we do to squeeze it down?

There's still an Apache httpd server, and mod_perl, but obviously only one of these instances running.

We still have Resin running webapps to implement the Krugle API, but there's no remote searchers for code. That all happens inside of the files webapp. And we don't support page crawling, so we can whack out a bunch of the Nutch crawl infrastructure.

But we need to add a nice UI for people who administer the system. And we also need to automate the code and SCM comment processing. So there's a Jetty-based webapp that implements the "hub" GUI I showed you previously, as well as this crawl process.

The configuration, including project definitions, is saved in a MySQL database that we interact with via Hibernate.

The result of a new or updated crawl is something we call a snapshot, which is a self-contained set of data stored in a directory. This includes the Solr index, the code index, the code files, reports generated on the code, and anything else that we serve up via the Krugle API on an enterprise box.

By creating this static, self-contained, optimized view of the data we can efficiently handle search requests, and avoid some of the data synchronization issues that force us to currently rotate entire clusters of servers on the public site.

Slide 21 - Key lessons

This is my final summary slide. If I had to do it all over again, what would be the most important

things I wished I'd know in advance?

The first item is avoiding the powerful urge to get the latest released version. Most of our developers use Macs, and I blame Mac OS X as the source of that urge. There's the continuous PR blitz to get critical security fixes. Most of us upgrade regularly, without any problems. But the same isn't always the case for open source. That 1.3.2 version might fix some bugs, but it can also add in a gnarly thread lock bug.

I had to include the build system here, because we use Maven. And like many people who use Maven, it's a love/hate relationship. Sometimes more hate than love, actually. For example, we'd added a maven-enforcer-plugin to our build. In July a new version was made available on the Maven public repo, something like 1.0alpha3. That version wound up automatically being used, and it had a nasty bug that prevented inter-module dependencies from resolving correctly. So we starting generating builds with out-of-date modules. It's the kind of thing that makes you age faster than you want. The solution there was to really, really lock down the versions of everything we use that gets pulled from the public Maven repo, and disable auto-downloading to be extra safe.

On the subject of what free means, nothing is every really free, and if you expect to somehow magically reduce your software development costs to zero by using open source, you're going to be disappointed. Yes, by using open source it reduced our startup costs significantly. But the bigger advantage for us was that we could quickly get changes made and bugs fixed, either by doing it ourselves, or in many cases paying a consultant to do it for us.

This aspect of paying to get things done, or maybe done sooner than it would have otherwise, is an interesting trend that I think will accelerate over time. If you're a commercial company like Krugle, having the ability to figure out who knows their stuff by watching the mailing lists, and then contracting with the right person to make the critical change in a timely manner - that's nirvana.

If we were using a commercial package, what? we file a change request? there's no way it would happen in time for us.

And hiring a random consultant to implement the functionality from scratch? Talk about a crap shoot.

The nature of open source projects, especially at Apache, means we have the ability to identify the right person, and we have the right to modify the code, both of which are critical.

As a side note, many of the consultants we work with have used a two-tier pricing model. If they make changes that get contributed back, it's a lower rate. And in most cases that's the route we've taken. Not only is it cheaper, but if you're worrying about helping out a competitor by improving the project for everybody, then you're worrying about the wrong things. You should be focused on what you do, not what your competitors might do.