# Chapter 6
# Krugle Code Search Architecture

Ken Krugler

**Abstract** Krugle was one of the earliest commercial portals for searching open source software. This chapter reviews the history of Krugle from initial inception to present day. It follows the search engine from the initial public version to the enterprise offering, with a particular focus on lessons learned from design decisions on topics such as web crawling, indexing, system architecture, and deployment.

## 6.1 Introduction

Krugle is a search engine for searching in source code and related technical information. There is a public site at Krugle.org, which has information on the top 3,500 open source projects, including project descriptions, licenses, software configuration management activity, and most importantly the source code—more than 400 million lines and growing.

There is also an enterprise version, which runs inside of company firewalls and provides the same search functionality against internal code and technical information.

In this chapter, I'll be describing the Krugle architecture, how it evolved over time, and the lessons we learned during that process.

## 6.2 Background

In 2004 I got actively involved in my first open source project, the ill-fated Chandler PIM. It slowly dawned on me that there were literally billions of lines of open source

K. Krugler (✉)
Scale Unlimited, 14860 Uren St, Nevada City, CA, USA
e-mail: kkrugler@scaleunlimited.com

code available, but no good way to find error messages, examples of API usage, or even the source for an open source component being used.

At the same time, a friend at a startup mentioned to me that the most powerful tool their developers had wasn't a debugger, or an IDE, or a build system—it was Google. This then led me to start thinking about how you'd create the ultimate programmer's search tool, and (in a broader sense) what "search-driven development" would look like.

In 2005 I started development of Krugle with a small team of friends. We got our first round of venture capital funding in September, and unveiled the Krugle site at the 2006 DEMO conference in February.

In 2006 we started work on the enterprise version of the product, which had a very different architecture. This was released to beta customers in April 2007.

In 2008 we switched the public site from the original distributed architecture to a version that uses a single large Krugle Enterprise appliance.

In 2011 the public site switched to the third version of our architecture, though still running on a Krugle Enterprise appliance.

## 6.3 Initial Public Version

We collected three types of information—web pages, source code and project descriptions. Each had its own collection, processing and searching infrastructure.

The web page and source code processing infrastructure was based on an early version of Hadoop, running on a cluster of 14 slave servers and one master.

The project descriptions were extracted and processed on a single server, coordinating with a MySQL database running on another server.

### *6.3.1 Web Page Crawling*

We used Nutch to crawl technical web pages, collecting and extracting information that would be of use to software developers. Examples of such "interesting" pages included open source projects, programming tutorials, mailing lists, and bug databases.

We modified Nutch to implement a "focused crawler." The diagram below explains the fundamental steps in a focused crawl—the key point is that each "fetch loop" only fetches a percentage of all known URLs, and the pages with the highest estimated score are fetched first. This allows us to focus the crawler on areas of the web that would prove most likely to contain quality pages (Fig. 6.1).

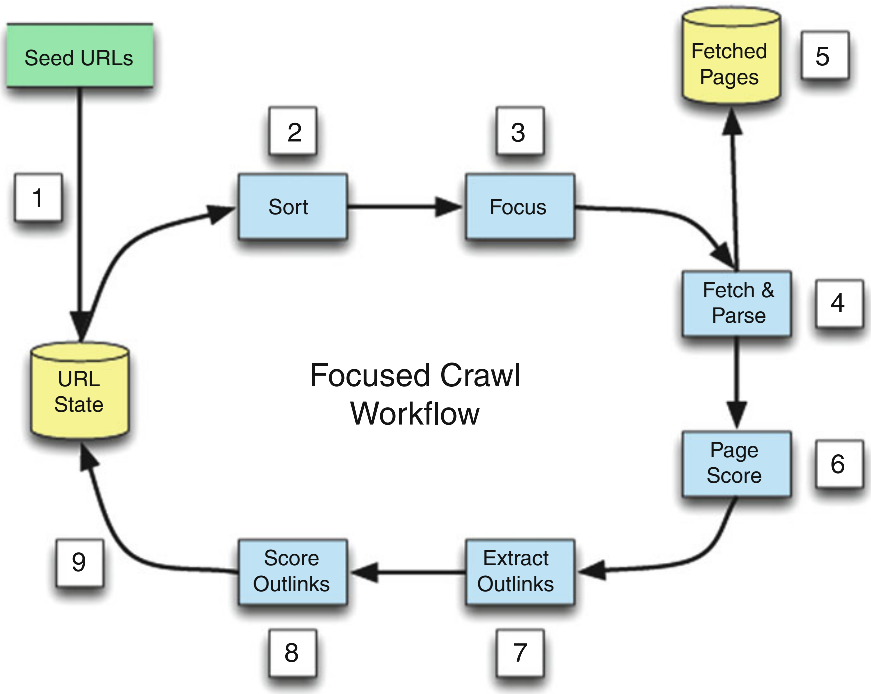


**Fig. 6.1** Focused crawl in initial public version

**URL State Database** This database (often called a "CrawlDB") maintains one entry for each unique URL, along with status (e.g. "have we fetched this page yet?"), the page score of fetched pages, and the link score. The actual approach used for this database varies, depending on the scale of the crawl, the availability of a scalable column-based DB or key/value store, etc.

**Page Score** Every fetched page is processed by a page scorer. This calculates a numeric value for the page, where higher values correspond to pages that are of greater interest, given the focus of the crawl. A page scorer can be anything from a simple target term frequency calculation to a complex NLP (natural language processing) analyzer.

**Link Score** Every URL has a score that represents the sum of page scores from all pages that contain an outbound link that matches the URL. Page scores are divided up across all outbound links, thus the increase in a URL's link score from a page with many outbound links will be minimized.

**Fetched Pages Database** This is where all fetched pages are stored, using the URL as the key. Typically this isn't a real database, but rather an optimized, compressed read-only representation.

With terminology out of the way, we can discuss the steps in a focused crawl workflow.

1. The first step is to load the URL State database with an initial set of URLs. These can be a broad set of top-level domains such as the 1.7 million web sites with the highest US-based traffic, or the results from selective searches against another index, or manually selected URLs that point to specific, high quality pages.
2. Once the URL State database has been loaded with some initial URLs, the first loop in the focused crawl can begin. The first step in each loop is to extract all of the unprocessed URLs, and sort them by their link score.
3. Next comes one of the two critical steps in the workflow. A decision is made about how many of the top-scoring URLs to process in this loop. The fewer the number, the "tighter" the focus of the crawl. There are many options for deciding how many URLs to accept—for example, based on a fixed minimum score, a fixed percentage of all URLs, or a maximum count. More sophisticated approaches include picking a cutoff score that represents the transition point (elbow) in a power curve.
4. Once the set of accepted URLs has been created, the standard fetch process begins. This includes all of the usual steps required for polite and efficient fetching, such as robots.txt processing. Pages that are successfully fetched can then be parsed.
5. Typically fetched pages are also saved into the Fetched Pages database.
6. Now comes the second of the two critical steps. The parsed page content is given to the page scorer, which returns a value representing how closely the page matches the focus of the crawl. Typically this is a value from 0.0 to 1.0, with higher scores being better.
7. Once the page has been scored, each outlink found in the parse is extracted.
8. The score for the page is divided among all of the outlinks.
9. Finally, the URL State database is updated with the results of fetch attempts (succeeded, failed), all newly discovered URLs are added, and any existing URLs get their link score increased by all matching outlinks that were extracted during this loop.

At this point the focused crawl can terminate, if sufficient pages of high enough quality (score) have been found, or the next loop can begin.

In this manner the crawl proceeds in a depth-first manner, focusing on areas of the web graph where the most high scoring pages are found.

In the end we wound up with about 50 million pages, and a "crawlDB" that contained around 250 million URLs, of which about half were scored high enough such that we would eventually want to crawl them.

### *6.3.2 Web Page Processing*

Once we had fetched a web page (or document, such as a PDF) then we'd parse it, to extract the title and text. Again, we leveraged the support that was already there in Nutch.

We also extracted information about source code repositories during the crawl, which allowed us to build a large list of CVS and SVN repositories for the source code crawler.

### *6.3.3 Web Page Searching*

Finally, we used Nutch's search support (built on top of Lucene) to support searching these web pages. The actual indexes were stored on multiple page searchers, since (at that time) a typical 4 core box with 8 GB of ram could comfortably handled 10–20 M pages, and our index was bigger than that. Nutch provided the support to distribute a search request to multiple searchers, each with a slice ("shard") of the index, then combine the results.

### *6.3.4 Source Code Crawling*

The source code crawler was also based on Nutch. We added "protocol handlers" for CVS and SVN, which let us leverage the distributed fetching and parsing support that was built into Nutch.

The "crawlDB" for the source code crawler contained HTTP-based URLs to SVN and CVS repositories. We manually entered many of these URLs that were found via manual searching, but we also included URLs that were discovered during the web page crawl as described previously. Finally, the project processing code (see below) also provided us with repository information.

One of the challenges we ran into was deciding whether to only get the trunk of project's code, or some number of the tags and branches as well. Initially we just went after the trunk, but eventually we settled on logic that would fetch the trunk, plus the "most interesting" tags, which we defined as being the latest point release for each major and minor version. Thus we'd go for the 1.0.4, 2.0 and 2.1.3 tags, but not 1.0.1, 1.0.2, 1.0.3, 2.1.0, etc.

We also found out quickly that we needed to be careful about monitoring and constraining the load that we put on CVS and SVN repositories. Due to a bug in the code, we accidentally wound up trying to download the complete Apache.org SVN repository—the trunk and all branches and tags from every project. This was crushing their infrastructure, and the ops team at Apache wisely blocked our crawler IP addresses, to prevent melt down. After some groveling and negotiations, plus more unit tests, we were unblocked and could resume the crawl at a more reasonable rate.

For some of the larger repositories, we looked into mirroring them, and eventually did set up an rsync of several. Unfortunately we were never able to negotiate a mirroring agreement with SourceForge, which was the biggest single repository that we needed to crawl. And the total number of unique repositories (more 100) made it impractical to negotiate that many data sharing deals, especially since many of these repositories only consisted of a few projects.

The situation in 2012 would be better suited to specific data mirroring agreements with a few repositories, given the larger number of code hosting sites that have significant number of projects (e.g. Google Code, GitHub). Note, though, that we would still need to crawl project descriptions independent of the code, as very few hosting sites or projects have adopted the use of standardized project metadata such as DOAP (description of a project) or fully specified Maven pom.xml files.

In the end, we wound up crawling over 130,000 projects found on more than 100 sites.

### *6.3.5 Source Code Processing*

The meat of our system was the parsing infrastructure that we built, using ANTLR 3.0 grammars. We developed over 30 grammars, which we use to turn source code into something internally we refer to as a "use-def tree."

As an example of what ANTLR grammar looks like, heres a snippet from the python.g file:

```
classdef
scope EnclosingScope;
    :   'class' (NAME->class(name={$NAME.text},
                begin={start($NAME)},
                end={stop($NAME)}))
        { $EnclosingScope::st = $classdef.st; }
        (LPAREN testlist RPAREN)? COLON
        { $st.setAttribute("comments", comments()); }
        suite
        // catch comment in classes without func defs
        {$st.setAttribute("containedComments",
        comments());}
    ;
```

This tree, which we saved as XML, essentially tagged text in the source file as being comments, code, or whitespace. In addition the code sections would be further tagged as class definitions, function definitions, and function calls.

An example of the resulting XML for a Java file looks like:

```
<krugleparse version="0.3">
<uri>test/EndianUtils.udt</uri>
<language>Java</language>
<udt>
<c b="0" e="803"><![CDATA[/*  * Licensed
to ï£¡  */]]></c>
```

```
<pkg n="org.apache.commons.io" b="813" e="833">
  <im n="java.io.EOFException" b="844" e="863"/>
  <im n="java.io.IOException" b="873" e="891"/>
  <im n="java.io.InputStream" b="901" e="919"/>
  <c b="952" e="1636"><![CDATA[/**  * Utility code
  ï£¡  */]]></c>
  <im n="java.io.OutputStream" b="929" e="948"/>
  <cd n="EndianUtils" b="1651" e="1661">
    <c b="1670" e="1748"><![CDATA[/* Instances
    should ...*/]]></c>
    <fd n="swapShort" b="2038" e="2046">
    </fd>
```

In order to pick the right parser, we had a preliminary analysis step that used the file name and regex patterns to determine the programming language.

We also ran additional tools over the code, including one that calculated actual lines of code, and another that extracted open source license information from source file comments and non-code text files. These results were then fed back into the project processing phase (see below) to add additional metadata to each project.

As part of a “virtuous cycle,” we also found URLs in comments, and added these to the crawlDB. This in turn helped us find pages with references to additional code repositories, which we could then crawl and parse—and the cycle repeats.

Finally, we processed the parse trees to create Lucene documents. Each XML document for one source file would become one Lucene document, with many of the fields (e.g. “function call”) being multi-valued, since one XML file could have many separate sections for each type of entity identified in the source.

We stored the actual source code separately, in regular files. One of our challenges became the management of large amounts of data—we were using custom-built filers (servers with lots of drives) to store many terabytes of source code, and keeping everything in sync, up to date, backed up, and available for processing was one of the major daily headaches. At this time Hadoop was not yet mature enough for us to trust it with our data—in fact we lost **all** of our crawl data once due to a bug in one of the first releases.

In the end we wound up with 2.6 billion lines of real source code in about 75 million source files.

### *6.3.6 Source Code Searching*

Finally, we again used Nutch’s search support to handle searching the source code. A search request would be distributed to multiple code searchers, each with a slice of the total index. Nutch would then handle combining the results.

We did wind up having to add a few enhancements to the search process, specifically enabling time limits on queries. The problem was that certain complex queries could take up to minutes to get results, and during this time they would cause all other queries to “stack up,” leading to poor search performance for all users. Our solution was to enable early search termination at a low level (in Lucene), where after a specified amount of time the search would terminate even if it hadn’t reached the end of the index. This would then return potentially different results, if the set of documents found prior to termination didn’t include all of the top results that would normally be returned, but we felt this was an acceptable tradeoff.

Source code searching required a special query parser, to support custom tokenization and other tweaks that we did to improve code searching. This query parsing was handled by the master search server, before queries were sent out to the four search slaves.

## *6.3.7 Project Crawling*

Initially we were only “crawling” projects (scraping pages) to find the repository information needed for our source code crawler. In early 2006, however, we realized that rich project metadata was critical to providing context for source code search results.

This changed the nature of our project crawling support, as it was no longer sufficient to use a page to extra repository data—instead, we needed to create a web mining system that could determine things like the project license.

The project crawl was all about discovering the URLs to project home pages. This typically involved custom Python code to do a very specific “discovery crawl” on an open source hoster site, e.g. Java.net. We also used data dumps from SourceForge.net and a few other sites that provided project listings in a format where we could then easily construct project home page URLs.

## *6.3.8 Project Processing*

Once the discovery crawl had found a number of project home page URLs, we would web mine those pages to extract the project name, license, and other useful metadata.

Each open source hosting site had its own HTML page format, which meant writing detail extractors (again, in Python) for each hoster. As you might imagine this wasn’t the most interesting thing to be working on, but it was critical for us to have structured data about projects to help augment code search results, as otherwise it was very challenging for end-users to understand and evaluate code search results. Providing project details added context that significantly improved the user experience.

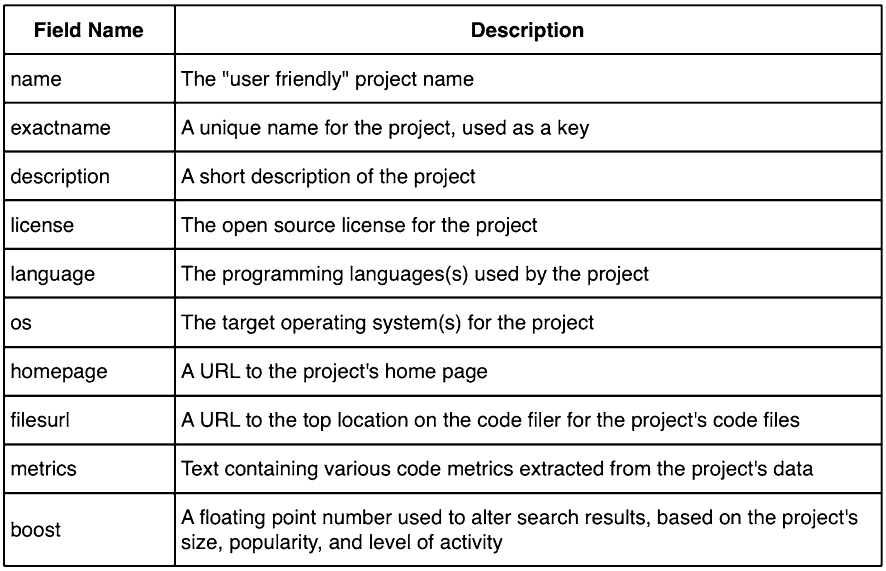
The results of this web mining would be saved to a MySQL database, where our librarian could review and correct any obvious errors. With over 130,000 projects it wasn't possible to review each one, so we ranked projects by their size and level of activity.

Packaging up the results included getting statistics out of the source code project (see the source code processing section), determining the location of the source on the code filer, and building a searchable index.

### *6.3.9 Project Searching*

By the time we added support for projects as separate entities in the system, CNet had open-sourced their enterprise search application ("Solr"). Solr provided a nice layer on top of Lucene, which simplified the work we had to do to add project-centric search support.

We created a Solr index schema that had the following fields, among others (Table 6.1):



| Field Name | Description |
|---|---|
| name | The "user friendly" project name |
| exactname | A unique name for the project, used as a key |
| description | A short description of the project |
| license | The open source license for the project |
| language | The programming languages(s) used by the project |
| os | The target operating system(s) for the project |
| homepage | A URL to the project's home page |
| filesurl | A URL to the top location on the code filer for the project's code files |
| metrics | Text containing various code metrics extracted from the project's data |
| boost | A floating point number used to alter search results, based on the project's size, popularity, and level of activity |

Table 6.1: Excerpt from solr index schema for projects

## 6.4 Public Site Architecture

The above describes a mixture of ad hoc back end systems, and the components used to provide search services. The actual architecture used to handle both end user (browser-based) queries and partner/enterprise API requests is a combination of a few additional systems (Fig. 6.2).

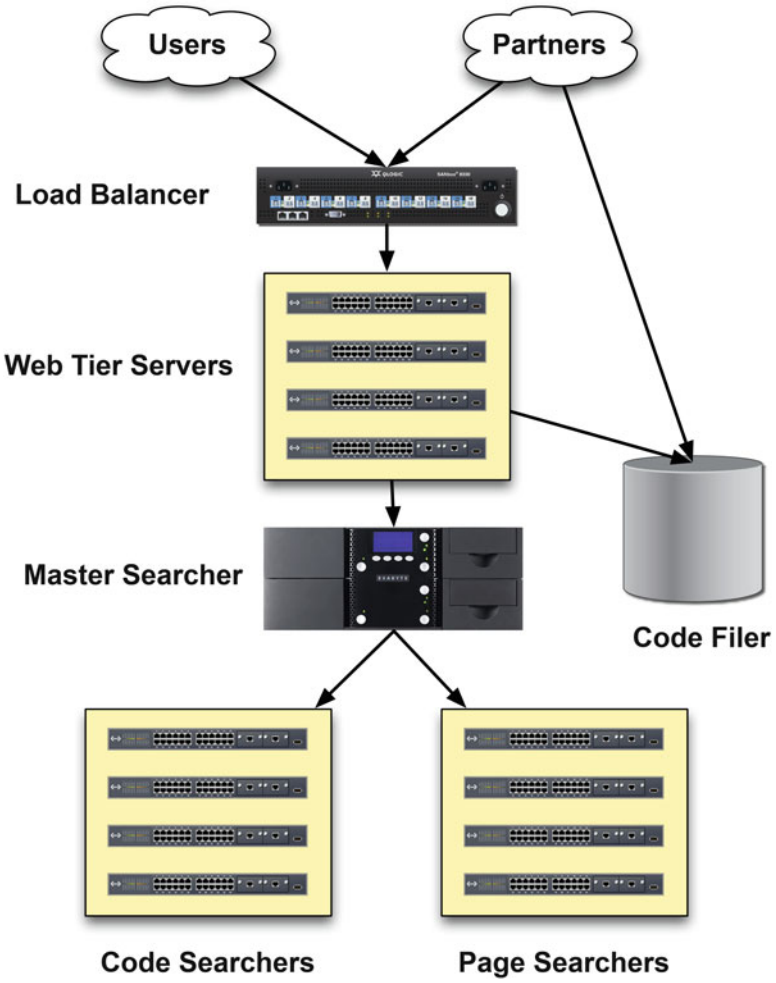


**Fig. 6.2** Architecture of public Krugle site

### *6.4.1 Load Balancer*

At the very top of the stack we had a load balancer, coupled with a firewall. All requests, either from end users (browsers) or via the Krugle API would go through a firewall and then to a load balancer, which would distribute the request to one of four servers sitting in the web tier.

### *6.4.2 Web Tier*

The four web tier servers all ran identical versions of Perl code that acted as an intermediary between the lower-level Krugle API (implemented in Java, uses XML) and higher-level requests coming from web pages, partners, and external Krugle Enterprise boxes.

This Perl layer gave us extra flexibility, which meant we could quickly fix problems in the lower-level API and add additional functionality without having to rebuild and redeploy the search infrastructure sitting below the web tier.

In retrospect I think we should have spent more time making sure the Krugle API was correct and complete, versus back-filling and patching via Perl code in the web tier. We didn't have a good way to test the web tier layer, versus the many unit tests we implemented against the Krugle API.

### *6.4.3 Master Searcher*

The single main searcher used a bigger hardware configuration, with more memory. The master searcher was responsible for the Krugle API, which was used by everybody (web tier, partners, and Krugle enterprise systems) to interact with the search indexes and data maintained by the public site.

The main searcher also ran the Solr search server responsible for project search requests.

### *6.4.4 Page Searchers*

We had four servers, each with one large/slow disk to store the web page content, and a second faster disk to store to the search index. These four "slave" searchers provided distributed search across the index, and a separate front-end process running on a master search server then combined the results.

### *6.4.5 Code Searchers*

Similar to web pages, we had four servers, though each of these had two fast disks to store to the search index. The actual code files were stored on a separate file server called the code filer.

### *6.4.6 Code Filer*

All of the actual code files were stored on a separate “code filer”, which was a big server stuffed with drives that were RAIDed together.

These files were fronted by the Lighttpd web server, which was configured to require a time-limited token for authorized access.

### *6.4.7 Live Versus Stand-by Versus Dev Systems*

The actual infrastructure was even more complex than what has been described above. The code searchers, page searchers, master searcher and code filer all comprised a snapshot of the state of the system, for both code and data. As such, we had three copies of these—one in production, one on standby, and one being provisioned as the next “live” system.

### *6.4.8 Release Process*

Whenever our “dev” system was sufficiently tested, we would do a “flip.” This consisted of switching the VIPs (virtual IP addresses) for the master server and code filer to the new system, which would then cause the web tier to start using the new code and data. The previously live system would become the backup, which we could easily flip back to if the new live system had serious problems. And the previous backup system would become the new dev system, which we would start provisioning as the next live system.

This continuous rotation of systems worked well, but at the cost of lots of extra hardware. We also spent a lot of time pushing data around, as provisioning a new server meant copying many terabytes of data to the searchers and the filer from our back-end systems.

### *6.4.9 Post Mortem*

The public search system worked, but required a lot of manual labor to build and deploy updates to the data, and had a lot of moving pieces. This complexity resulted in a lot of time spent keeping the system alive and happy, versus improving and extended the search functionality.

In retrospect it would have made more sense to first focus on the Krugle Enterprise system, and handle the public site via setting up multiple instances of these servers, each with a slice of the total projects. Then the delta between a stock Krugle

Enterprise system and the public site would have been a top-level "master" that handles distributing search requests and combining the results.

We would still have needed a back-end system to handle crawling web pages and handling search requests, but that's a much more isolated problem, and one with better existing support from Nutch.

## 6.5 Krugle Enterprise

The architecture described above worked well for handling billions of lines of code, but it wasn't suitable for a stand-alone enterprise product that could run reliably without daily care and feeding. In addition, we didn't have the commit comment data from the SCM systems that hosted the project source code, which was a highly valuable source of information for both searches and analytics.

So we created a workflow system (internally called "the Hub") that handled the crawling and processing of data, and converted the original multi-server search system into a single-server solution ("the API").

The enterprise version doesn't support crawling web pages, and it relies on users manually defining projects—specifying the repository type and location, the description of the project, etc. This information is still stored in a MySQL database.

### *6.5.1 SCM Comments*

We added support for fetching, parsing and searching SCM comments that we retrieved from SCM systems. These comments were stored in the same Solr search server used for project search, but in a different Solr "core."

We created a Solr index schema that had the following fields, among others (Table 6.2):

### *6.5.2 SCMI Architecture*

Early on we realized that it would be impossible to install and run all of the many different types of source code management system (SCM) clients on the enterprise server. For example, a ClearCase SCM requires a matching client, which in turn has to be custom installed.

Our solution was to define a standard protocol between the Hub and "helper" applications that could run on other servers. This SCM interface (SCMI) let us quickly build connectors to many different SCM systems, including ClearCase, Perforce, StarTeam, and git, as well as non-SCM sources of information such as Jira and Bugzilla.

| Field Name | Description |
|---|---|
| uid | A unique id generated for each SCM comment. |
| projectUid | The unique id (project name) for the corresponding project. |
| projectName | The "user friendly" project name |
| revision | The revision number for the corresponding SCM commit. |
| author | User name of the person who did the SCM commit. |
| date | Date of the commit. |
| description | User-provided description of the commit. |
| files | List of files that were changed, added, or deleted. |
| activityScore | A float that describes the impact of the commit, measured by number of files changed, added, or deleted. |

Table 6.2: Excerpt from Solr index schema for SCM data

## *6.5.3 Performance*

For an enterprise search appliance, a basic issue is how to do two things well at the same time—updating a live index, and handling search requests. Both tasks can require extensive CPU, disk and memory resources, so it's easy to wind up with resource contention issues that kill your performance.

We made three decisions that helped us avoid the above. First, we pushed a significant amount of work "off the box" by putting a lot of the heavy lifting work into the hands of small clients called Source Code Management Interfaces (SCMIs). These run on external customer servers instead of on our appliance, and act as collectors for information about projects, SCM comments, source code and other development-oriented information. The information is then partially digested before being sent back to the appliance via a typical HTTP RESTful protocol.

Second, we use separate JVMs for the data processing/indexing tasks versus the searching/browsing tasks. This let us better control memory usage, at the cost of some wasted memory. The Hub data processing JVM receives data from the SCMI clients, manages the workflow for parsing/indexing/analyzing the results, and builds a new "snapshot." This snapshot is a combination of multiple Lucene indexes, plus all of the content and other analysis results. When a new snapshot is ready, a "flip" request is sent to the API JVM that handles the search side of things, and this new snapshot is gracefully swapped in.

On a typical appliance, we have two 32-bit JVMs running, each with 1.5 GB of memory. One other advantage to this approach is that we can shut down and restart each JVM separately, which makes it easier to do live upgrades and debug problems.

Finally, we tune the disks being used to avoid seek contention. There are two drives devoted to snapshots, while one is serving up the current snapshot, the other is being used to build the new snapshot. The Hub also uses two other drives for raw data and processed data, again to allow multiple tasks to run in a multi-threaded manner without running into disk thrashing.

The end result is an architecture that looks like this (Fig. 6.3):

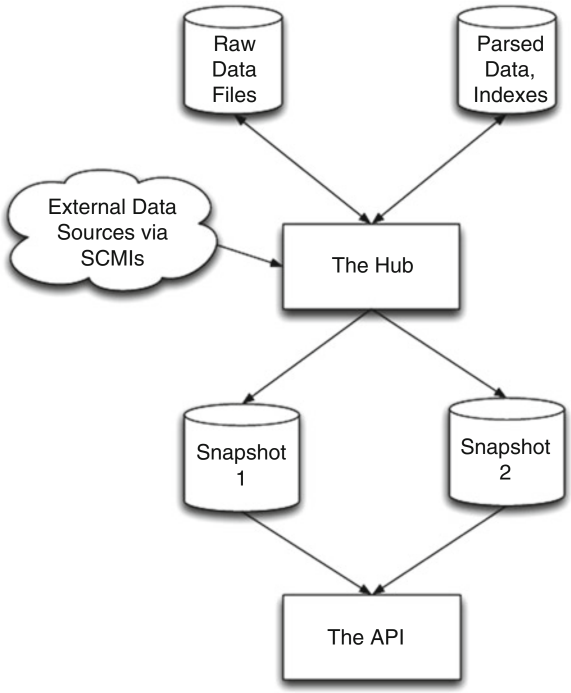


**Fig. 6.3** Architecture of Krugle enterprise

### *6.5.4 Parsing Source Code*

During early beta testing, we learned a lot about how developers search in code, with two in particular being important. First, we needed to support semi-structured searches, for example where the user wants to limit the search to only find hits in class definition names.

In order to support this, we had to be able to parse the source code. But "parsing the source code" is a rather vague description. There are lots of compilers out there that obviously parse source code, but full compilation means that you need to know about include paths (or classpaths), compiler-specific switches, the settings for the macro preprocessor in C/C++, etc. The end result is that you effectively need to be able to build the project in order to parse it, and that in turn means you wind up with a system that requires constant care and feeding to keep it running. Often that doesn't happen, so the end result is shelfware.

Early on we made a key decision, that we had to be able to process files individually, without knowledge of build settings, compiler versions, etc. We also had to handle a wide range of languages. This in turn meant that the type of parsing we could do was constrained by what features we could extract from a very fuzzy parse. We couldn't build a symbol table, for example, as that would require processing all of the includes/imports.

Depending on the language, the level of single-file parsing varies widely. Python, Java and C# are examples of languages where you can generate a good parse tree, while C/C++ are at the other end of the spectrum. Languages such as C/C++ that supports macros and conditional compilation are especially challenging. Dynamic languages like Ruby and Perl create their own unique problems, as the meaning of a term (is it a variable or a function) sometimes isn't determined until run-time.

So what we wind up with a best guess, where we're right most of the time but we'll occasionally get it wrong.

We use ANTLR to handle most of our parsing needs. Terr Parr, the author of ANTLR, added some memoization support to version 3.0, which allowed us to use fairly flexible lexer rules without paying a huge performance penalty for frequent back-tracking.

### *6.5.5 Substring Searching*

The second important thing we learned from our beta testing was that we had to support some form of substring searching. For example, when a user searches on "ldap" she expects to find documents containing terms like "getLDAPConfig" , "ldapTimeout" , and "find_details_with_ldap" .

We could treat every search term as if it had implicit wildcards, like "*ldap*" , but that is both noisy and slow. The noise (false positive hits) comes from treating all contiguous runs of characters as potential matches, so a search for "heap" finds a term like "theAPI" .

The performance hit comes from having to: (a) first enumerate all terms in the index to find any that contain <term> as a substring, and then (b) use the resulting set of matching terms in a (potentially very large) OR query. BooleanQuery allows a maximum of 1,024 clauses by default—searching on the Lucene mailing list shows many people have encountered this limit while trying to support wildcard queries.

There are a number of approaches to solving the wildcard search problem, some of which are covered in the "Lucene in Action" book. For example, you can take every term and index it using all possible suffix substrings of the text. For example, "myLDAP" gets indexed as "myl", "yld", "lda", and so on. This then lets you turn a search for "*ldap*" into "ldap*", which cuts down on the term enumeration time by being able to do a binary search for terms starting with "ldap", versus enumerating all terms. You still can wind up with a very large number of clauses in the resulting OR query, however. And the index gets significantly larger, due to term expansion.

Another approach is to convert each term into n-grams, for example, using 3-g the term "myLDAP" would become "myl", "yld", "lda", "dap", and so on. Then a search for "ldap" becomes a search for "lda dap" in 3-g, which would match. This works as long as N (e.g. 3, in this example) is greater than or equal to the minimum length of any substring you'd want to find. It also significantly grows the size of the index, and for long terms results in a large number of corresponding n-grams.

Another approach is to pre-process the index, creating a secondary index that maps from each distinct substring to the set of all full terms that contain the substring. During a query, the first step is to use this secondary index to quickly find all possible terms that contain the query term as a substring, then use that list to generate the set of sub-clauses for an OR query, similar to above. This gives you acceptable query-time speed, at the cost of additional time during index generation. And you're still faced with potentially exceeding the max sub-clause limit.

We chose a fourth approach, based on the ways identifiers naturally decompose into substrings. We observed that arbitrary substring searches were not as important as searches for whole sub-words. For example, users expect a search for "ldap" to find documents containing "getLDAPConfig", but it would be very unusual for the user to search for "apcon" with the same expectation.

To support this, we implemented a token filter that recognizes compound identifiers and splits them up into sub-words, a process vaguely similar to stemming. The filter looks for identifiers that follow common conventions like camelCase, or containing numbers or underscores. Some programming languages allow other characters in identifiers, or indeed, any character; we stuck with letters, numbers, and underscores as the most common baseline. Other characters are treated as punctuation, so identifiers containing them are still split at those points. The difference is that the next step, sub-range enumeration, will not cross the punctuation boundary.

When we encounter a suitable compound identifier, we examine it to locate the offsets of sub-word boundaries. For example, "getLDAPConfig" appears to be composed of the words "get", "LDAP", and "Config", so the boundary offsets are at 0, 3, 7, and 13. Then we produce a term for each pair of offsets (i,j) such that $i < j$. All terms with a common start offset share a common Lucene index position value; each new start offset gets a position increment of one.

### *6.5.6 Query Versus Search*

One of the challenges we ran into was the fundamentally different perception of results. In pure search, the user doesn't know the full set of results, and is searching for the most relevant matches. For example, when a user does a search for "lucene" using Google, they are looking for useful pages, but they have little to no idea about the exact set of matching pages.

In what I'm calling a query-style search request the user has more knowledge about the result set, and expects to see all hits. They might not look at each hit, but if a hit is missing then this is viewed as a bug. For example, when one of our users searches for all callers of a particular function call in their company's source code, they typically don't know about every single source file where that API is used (otherwise they wouldn't need us), but they certainly do know of many files which should be part of the result set. And if that "well known hit" is missing, then we've got big problems.

So where did we run into this situation? When files are very large, the default setting for Nutch was to only process the first 10 K terms. This in general is OK for web pages, but completely fails the query test when dealing with source code. Hell hath no fury like a developer who doesn't find a large file they know should be a hit, because the search term only exists near the end.

Another example is where we miss-classified a file, for example, if file xxx.h was a C++ header versus a C header. When the user filters search results by programming language, this can exclude files that they know of and are expecting to see in the result set.

There wasn't a silver bullet for this problem, but we did manage to catch a lot of problems once we figured out ways to feed our data back on itself. For example, we'd take a large, random selection of source files from the http://www.krugle.org site, and generate a list of all possible multi-line ("code snippet") searches in a variety of sizes (e.g. 1–10 lines). We'd then verify that for every one of these code snippets, we got a hit in the original source document.

## *6.5.7 Post Mortem*

The Krugle enterprise search system is in active use today at a mixture of Fortune 100 and mid-size technology companies. The major benefits seen by customers are: (a) increased code re-use, primarily at the project level; and (b) a decrease in time spent fixing the same piece of code that exists in multiple projects.

A major challenge has been to provide potential customers with a way to quantify potential benefits. There's a general perception that search is important, e.g. it's easy to agree with statements like "If you can't find it, you can't fix it." It's difficult, though, to determine how much time and money such a system would save, and thus whether investing in a Krugle system is justified.

One additional and unexpected hurdle has been integration of Krugle systems into existing infrastructure, primarily for authentication and authorization. Many large enterprise customers are very sensitive about who can access source code, and even between groups in the same company a lack of trust means that providing enterprise-wide access control that all parties accept often leads to protracted engagements with significant profession services overhead.

**Acknowledgements** Portions of this chapter were adapted from a case study written by the author that was previously published in the book "Lucene In Action, 2nd Edition" by Michael McCandless, Erik Hatcher, and Otis Gospodnetić.