Building blocks of a scalable web crawler

Marc Seeger
Computer Science and Media
Stuttgart Media University

September 15, 2010

A Thesis Submitted in Fulfilment of the Requirements for a Degree of
Master of Science in Computer Science and Media
Primary thesis advisor: Prof. Walter Kriha
Secondary thesis advisor: Dr. Dries Buytaert

Abstract

The purpose of this thesis was the investigation and implementation of a good architecture for collecting, analysing and managing website data on a scale of millions of domains. The final project is able to automatically collect data about websites and analyse the content management system they are using.
To be able to do this efficiently, different possible storage back-ends were examined and a system was implemented that is able to gather and store data at a fast pace while still keeping it searchable.
This thesis is a collection of the lessons learned while working on the project combined with the necessary knowledge that went into architectural decisions. It presents an overview of the different infrastructure possibilities and general approaches and as well as explaining the choices that have been made for the implemented system.

Acknowledgements

I would like to thank Acquia and Dries Buytaert for allowing me to experience life in the USA while working on a great project. I would also like to thank Chris Brookins for showing me what agile project management is all about.
Working at Acquia combined a great infrastructure and atmosphere with a pool of knowledgeable people. Both these things helped me immensely when trying to find and evaluate a matching architecture to this project.
Finally, I would like to thank Professor Kriha who helped with the bureaucratic process and fueled my interest in scalable architectures.

Statement of originality

Marc Seeger , September 2010

Contents

Contents **i**

List of Figures **iv**

1 Introduction to the Project **1**

1.1 Acquia 1

1.2 Requirements 1

1.3 Existing codebase 2

2 Architectural decisions and limitations **5**

2.1 Estimated back-end load 5

2.2 Ruby 7

2.2.1 The choice for Ruby 7

2.2.2 Language Features 8

2.2.3 C Extensions 8

2.2.4 VMs 10

2.3 I/O model (async vs threading) 15

2.4 Amazon Elastic Compute Cloud 17

2.4.1 Instance Types 17

2.4.2 Elastic Block Store 19

2.4.3 Performance 20

3 Back-end and Search **23**

3.1 Datastores 23

3.1.1 Categorization 23

3.1.2 Data store scalability 27

3.2 Datastructures 28

3.2.1 Hash-based 29

3.2.2 R-tree-based 31

3.2.3 Merkle-tree-based 32

3.2.4 Trie-based 34

3.2.5 Bitmap-based 36
3.3 Generic Problems 38
3.3.1 HTTP persistent connections 38
3.3.2 Locking 39
3.3.3 Append-only storage and compacting 40
3.4 Search possibilities 41
3.4.1 Classification 41
3.4.2 Indexation 43
3.4.3 Map/Reduce 44
3.4.4 Search and the dynamo model 45
3.5 Evaluation 47
3.5.1 MongoDB 47
3.5.2 CouchDB 52
3.5.3 MySQL and PostgreSQL 57
3.5.4 Tokyo Cabinet 59
3.5.5 Riak 62
3.5.6 Cassandra 66
3.5.7 Miscellaneous 69
3.6 External search 72
3.6.1 Sphinx 72
3.6.2 Solr 75
3.6.3 Elasticsearch 77
3.7 Work Queues 80
3.7.1 Redis 80
3.7.2 Conclusion for the project 82
3.7.3 Beanstalkd 83
3.7.4 Conclusion for the project 84

4 Crawler **85**
4.1 System architecture 85
4.1.1 Components 85
4.1.2 Data schema 88
4.2 Problems 91
4.2.1 HTTP redirects and standards 91
4.2.2 wildcard subdomains and spam 92
4.2.3 www cname records 93
4.2.4 top level domain detection 93
4.2.5 Balancing parallelism 94
4.2.6 File descriptors 95
4.3 Link collection 96
4.3.1 RegExp vs HTML Parsing 96

4.3.2 Adaptive crawl depth 97
4.3.3 Twitter 98
4.3.4 Seed sites 99
4.3.5 DMOZ and Wikipedia 100
4.3.6 Zone files 100
4.3.7 Recrawling 101
4.4 Distribution 101

5 Profiling-Methods 103
5.1 lsof 103
5.2 dtrace and strace 104
5.3 curl 105
5.4 mtop 106
5.5 JRuby based profiling 106

6 Fingerprinting 109
6.1 CMS detection 109
6.1.1 The Generator meta tag 110
6.1.2 Included files 110
6.1.3 Javascript variables 111
6.1.4 Comments 111
6.1.5 Special paths 111
6.1.6 Images 112
6.1.7 HTTP headers 112
6.1.8 Cookies 113
6.1.9 Problems 113
6.2 Web servers 114
6.3 Drupal specific data 115
6.3.1 Modules 115
6.3.2 Versions 115
6.4 Hosting Provider 117
6.5 Geolocation 118
6.6 Industry 118

7 Conclusion and Outlook 121

Bibliography 123

List of Figures

3.1 B^+-Tree 29
3.2 Merkle-Tree 33
3.3 Trie datastructure 35
3.4 Range query in a Trie 36
3.5 Bitmap-Index 37
3.6 Reverse Index 43
3.7 Redis parallel access 82

4.1 Architecture overview 88

Chapter 1

Introduction to the Project

1.1 Acquia

The project discussed in this thesis was created for Acquia Inc. Acquia is a startup company based near Boston, Massachusetts. They provide services around the content management system Drupal ranging from hosting to consulting. The original idea for this project can be found in a blog post by Dries Buytaert titled the "Drupal site crawler project"[1]. Dries Buytaert is the original creator of the content management system Drupal as well as the CTO and Co-founder of Acquia.

1.2 Requirements

Acquia would benefit from knowing the current distribution of the Drupal content management system. As an open source project, tracking the number of Drupal installations can't be done by looking at licensing numbers. Instead, an actual crawling of web sites needs to be done to accurately count Drupal's real world usage. The initial idea for the project was the creation of a web crawler that would analyse websites for certain properties. Most importantly, the crawler should be able to detect if a website is using a content management system. Especially for

[1]`http://buytaert.net/drupal-site-crawler-project`

websites running on the Drupal CMS, the application should be able to detect details like version numbers, modules that are in use, and other specifics. Over the course of the project, the possibility of searching the collected data for certain combinations of features, e.g. "all Drupal sites that run on the IIS webserver and use a '.edu' toplevel domain", was added to the requirements. A specific focus was set on getting the crawler infrastructure scalable enough to collect details about hundreds of millions of domains while still keeping the the data searchable. The initial version of the crawler was designed by Dries Buytaert himself. He describes his experience in designing the initial crawler like this:

> Thanks to my engineering background and my work on Drupal, scalability issues weren't new to me, but writing a crawler that processes information from billions of pages on the web, is a whole different ball park. At various different stages of the project and index sizes, some of the crawler's essential algorithms and data structures got seriously bogged down. I made a lot of trade-offs between scalability, performance and resource usage. In addition to the scalability issues, you also have to learn to deal with massive sites, dynamic pages, wild-card domain names, rate limiting and politeness, cycles in the page/site graph, discovery of new sites, etc.

During the course of this project, I have encountered similar trade-offs and problems. This thesis is a collection of the lessons I learned while working on the project combined with the necessary knowledge that went into architectural decisions. The thesis should be able to give an overview of possibilities and the choices I have made. This way, it should be possible to use these evaluations of building blocks in other projects.

1.3 Existing codebase

When starting the project, there was already a small amount of code available. It was an initial version of the crawler that had been created by an intern over the course of a few weeks. It was based on the Hadoop framework. Hadoop is a Java-based framework to create and manage Map-Reduce jobs. The framework consists of several architectural components such as a distributed file system (HDFS), a central locking service (Zookeeper), a database (HBase) and a job distribution service (Jobtracker). Using this infrastructure, developers are able to split their tasks into several map and reduce phases and distribute these jobs over a number of servers in a transparent way.

While the Map-Reduce implementation that Hadoop provides is a good option for analysing large amounts of data, it added a large overhead to a web crawler infrastructure. Most of the first productive week of the project was spent trying to set up the complete Hadoop infrastructure and getting all of the components to work together. The wrapping of data collection phases into map and reduce phases added an extra layer of complexity on top of the actual application logic which seemed unnecessary. After discussing these issues with other engineers working at Acquia, I came to the conclusion that taking a fresh start with a worker queue based approach was a viable alternative to the current system design.

Chapter 2

Architectural decisions and limitations

2.1 Estimated back-end load

The first step in designing the system architecture is to accurately predict the production load on the back-end systems. Knowing these numbers will heavily influence the selection of caching layers and data stores that provide the foundation for the project. Let's assume our initial use-case is the simple crawling of a single website. Our crawler would go through a process involving these steps:

1. Get the domain URL from a queue
2. Download the front page (path: `"/"`) and robots.txt file
3. Parse all links to other domains
4. Check if the discovered links to external domains have already been analysed
5. If they haven't, put them on a queue.If they have, update an incoming link counter
6. Save the collected information about the site to the database.

7. Optionally: Save the sites HTML and HTTP headers to the database

Looking at I/O operations, this means that for every link to a domain we encounter, we have got to:

1. Check if this domain is already in the data store
2. Insert the link into the queue OR increment a incoming link counter

That means at least two operations for every discovered external link, given that our back-end has atomic increments of integer values. As an average load, this means that the back-end systems would have to withstand:

$$ops = dps * (2 * elpd) + is$$

ops = back-end i/o operations per second
dps = number of processed domains per second
elpd = external links per domain
is = amount of operations needed for storing the collected information. depending on the back-end, this might only be 1 write operation (e.g. a single SQL INSERT)

Let us see what this means if we put in conservative values:

dps: The initial target for the crawler are 10 domains per second (about 864.000 domains per day).

elpd: Information about this number can be found in several other papers. For our calculation, we assume an average of 7.5 external links per web page. [1]

[1] Broder et al.[2] estimated the average degree of external links per page at about 7. Ola Ågren talks in his paper "Assessment of WWW-Based Ranking Systems for Smaller Web Sites"[4] about "8.42 outgoing hyperlinks per HTML page." with a sample size of 7312 pages A group of Brazilian researchers set the number at 6.9 links per page with a sample size of about 6 Million pages and documented the data in the paper "Link-Based Similarity Measures for the Classification of Web Documents"[5]

is: In our case, we assume that our back-end data is completely denormalized and that we can write the fingerprint information to the data store in one single write operation.

With these numbers, we end up with approximately: $10 * (2 * 7.5) + 1 + 2 * (0.005) \approx 151$ operations per second on our back-end system just for crawling the front page alone. Additional operations will be necessary for making the data searchable, backups and other tasks. Even without those additional tasks, our back-end would have to complete an operation in under 7 milliseconds if we want to be able to process 10 domains per second.

With our current estimations, our back-end would end up having to deal with a minimum of 16 back-end operations for every domain we analyse. Additional features such as monitoring CMS changes when recrawling a domain or crawling more than just the front page would drive up the load on the system.

2.2 Ruby

2.2.1 The choice for Ruby

While the already existing source code was Java based, I decided to implement the new crawler in the programming language Ruby. Because of the size of the project and only me working on it, my main goal was to keep the code simple and rely on library support wherever possible. Ruby has strong roots in Perl and seemed to be a good fit for the analytical part of the process. Fingerprinting content management systems and dealing with HTML is greatly simplified by the support of regular expressions on a language level. and the availability of well documented and designed HTML parsers such as Nokogiri[2] and Hpricot [3]. A downside of Ruby is the set of problems that occur when using threads as a means of parallelizing

[2]`http://nokogiri.org/`

[3]`http://wiki.github.com/hpricot/hpricot/`

processes. The differences between the different available Ruby VMs in terms of threading support will be discussed in the VM subsection (2.2.4) of this chapter.

Another downside that people usually see when talking about Ruby is the low performance in terms of execution speed. Since our main performance intensive operation is the parsing of the incoming HTML, this problem is solved by Ruby's usage of libraries with c-extensions (see: 2.2.3).

2.2.2 Language Features

Ruby is a multi-paradigm programming language. It allows:

object orientation : every data type is an object. This also includes classes and types that many other languages implement as primitives (e.g. booleans, integers or null/nil)

procedural programming : when defining functions or variables outside of classes, it makes them part of the root, 'self' Object

functional programming : Ruby supports anonymous functions, closures, and continuations. All statements have values and all functions return the last evaluation implicitly

Besides these paradigms, Ruby is also a dynamic language in that it supports introspection, reflection as well as meta programming. Ruby uses a dynamic type system (so called "duck typing"). In terms of object orientation, Ruby supports inheritance and singleton methods. Although Ruby does not support multiple inheritance it allows the import of functionality using modules. These imports are called "mixins".

2.2.3 C Extensions

A good way of differentiating between the leading Ruby VMs is looking at the way they support C extensions. C extensions are a way for ruby-libraries to speed up computation extensive operations(e.g. XML parsing). By implementing these operations in C, library creators allow developers to interact with Ruby and still harvest the performance of raw C code. This performance advantage was especially important in the early days of Ruby. While Ruby's performance has increased, the usage of C based extensions is still beneficial. This holds especially true when it comes to gaining the functionality of many of the stable and well tested C-libraries. Wrapping these libraries into a Ruby layer is especially helpful when it comes to the large C-based libraries such as "libxml" (used in the Ruby XML/HTML parser "Nokogiri") or ImageMagick, for which RMagick provides a Ruby interface.

The downside of C extensions from a language point of view is, that they offer direct pointer access. This complicates the implementation of e.g. better garbage collection and in general, holds back VM development. It has to be noted that not all of the Ruby VMs support the use of C extensions. The level of support will be discussed in the respective VM's chapter. There are two different kinds of C extensions. The "regular" C extensions pose a problem for alternate implementations like JRuby, because of the complexity involved when exposing internals of the Ruby implementation or the usage of expensive (de)serialization. These extensions connect to Ruby's native API as exposed through ruby.h and libruby.

With the foreign function interface "FFI", programmers do not have to write a lot of C code and can stay in Ruby for most of the work. Charles Nutter, the project lead for Ruby, describes FFI in one of his blog posts[4] as follows:

> FFI stands for Foreign Function Interface. FFI has been implemented in various libraries; one of them, libffi, actually serves as the core of JNA, allowing Java code to load and call arbitrary C

[4]`http://blog.headius.com/2008/10/ffi-for-ruby-now-available.html`

libraries. libffi allows code to load a library by name, retrieve a pointer to a function within that library, and invoke it, all without static bindings, header files, or any compile phase.

2.2.4 VMs

One of the interesting things about Ruby is the number of Virtual Machines that are able to execute Ruby source code. From a feature point of view, the VMs differ in the way they implement threads (green threads vs. native threads), their general performance, the garbage collection algorithms they use, and weather or not the VM uses a global interpreter lock to synchronize threads. Some of them also employ different techniques such as JIT compilers to gain execution speed. To look at different performance numbers, The "great Ruby Shootout"[5] offers a lot of comparisons between the current Ruby VMs. The series of articles mainly focuses on performance in micro-benchmarks, but also pays attention to RAM consumption.

2.2.4.1 Global interpreter lock and threading

Especially when trying to implement a crawler that has a good domain throughput, it is important to look at the way that the execution environment handles threads. When a VM implements a global interpreter lock, it forces threads to acquire a lock before executing code. This lock is shared between all of the threads inside a VM. This means that only one thread can run at a time. While this seems to defeat the purpose of threads, it still enables a program to gain performance compared to a single threaded alternative. The way that the VM switches between threads has a large impact on the possible gains. Sasada Koichi, creator of YARV (the VM that is powering Ruby 1.9), explains on the ruby mailing list [6] how developers of C extensions can unlock the GIL before calling blocking functions by using the rb_thread_blocking_region() API. This allows

[5]http://programmingzen.com/2010/07/19/the-great-ruby-shootout-july-2010/
[6]http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/10252

parallel execution of code in a native OS threads. Doing this however results in some limitations:

1. The called function must be thread safe
2. You cannot call ruby functions from within your extension
3. System calls like thread_mutex_lock() can't be interrupted. This means that timeout() won't affect a function using these calls

While this is only useful for C extensions and does not allow "pure" Ruby threads to run in parallel, it solves most of the "big" performance problems for computationally expensive operations since most of them are implemented as 3rd party libraries in C. This especially holds true in our case where the external network calls are all handled by C-based libraries. Guido van Rossum, inventor of the Python programming language, even goes a step further and has this to say about the existence of a global interpreter lock in modern Programming languages such as Ruby and Python in one of his posts on the python-3000 mailing list [7]:

> Nevertheless, you're right the GIL is not as bad as you would initially think: you just have to undo the brainwashing you got from Windows and Java proponents who seem to consider threads as the only way to approach concurrent activities. Just because Java was once aimed at a set-top box OS that did not support multiple address spaces, and just because process creation in Windows used to be slow as a dog, doesn't mean that multiple processes (with judicious use of IPC) aren't a much better approach to writing apps for multi-CPU boxes than threads. Just Say No to the combined evils of locking, deadlocks, lock granularity, livelocks, nondeterminism and race conditions.

Especially when looking at languages made for highly parallel applications such as Erlang or Mozart/Oz, the existence of lightweight user-level

[7] `http://mail.python.org/pipermail/python-3000/2007-May/007414.html`

threads is widespread. For scaling over several CPUs, the current idea is to just launch several processes or fork the VM. The fork() system call allows for cheap duplication of a running process which is mostly implemented using copy-on-write semantics. Copy operations are implemented on a page-level and make fork() an alternative to investigate. It has to be noted though, that fork() is highly dependant on its implementation in the operating system. Here is a short excerpt from the Linux man page for fork:

> fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited. *Under Linux, fork() is implemented using copy-on-write pages*, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

2.2.4.2 MRI - Ruby 1.8

The VM that is currently most widely deployed is called "MRI", short for Matz's Ruby Interpreter. It was implemented by Ruby's inventor Yukihiro Matsumoto. It was the official VM for Ruby over the last few years and has the version number 1.8.

Threads: MRI implements threads as lightweight green threads and synchronizes them using a global interpreter lock.

C Extensions: MRI supports C Extensions

Garbage Collection: MRI uses a simple "mark and sweep" garbage collection algorithm.

Misc: MRI is probably still the most compatible version for third party libraries, although most library-developers are slowly moving to 1.9 compatibility. The changes between 1.8 and 1.9 are not that big, so it is an easy port for most projects.

2.2.4.3 YARV - Ruby 1.9

YARV (Yet Another Ruby VM) aka. Ruby 1.9 is the current, official Ruby VM and the successor of Ruby 1.8 (MRI). It has a vastly improved performance compared to 1.8.

Threads: YARV implements threads as native threads. They are, however, still synchronized with a global interpreter lock.

C Extensions: YARV supports C Extensions

Garbage Collection: YARV still uses a simple "mark and sweep" garbage collection algorithm.

Misc: Further performance improvements are available in the current development version. Ruby 1.9 added coroutines called "Fibers" for lightweight concurrency.

2.2.4.4 JRuby

JRuby is an implementation of Ruby running on the Java Virtual machine. JRuby essentially compiles ruby source code down to Java bytecode. For some of the dynamic features of Ruby, JRuby has to go through great lengths to imitate them using the given set of JVM bytecodes and thus does not reach native Java performance. At the time of writing, it ties with Ruby 1.9 in terms of performance. An added bonus is the ability to easily interact with Java libraries. This way, it can be used as "glue code" to keep the verbosity of Java to a minimum while still being able to leverage the high performance of the JVM.

Threads: JRuby uses operating system threads and doesn't have a global interpreter lock

C Extensions: JRuby can use C-extensions that use the Foreign Function Interface (FFI)[8]. Some of the bigger libraries ship Java-based extensions to create compatibility with JRuby. There is work on the way to fully

[8] `http://github.com/ffi/ffi#readme`

support C extensions. A good recap of this can be found on Charles Nutter's Blog[9]. He is one of the Project leads for JRuby and has valuable insight on the possible support for regular C extensions in JRuby:

> There's a very good chance that JRuby C extension support won't perform as well as C extensions on C Ruby, but in many cases that won't matter. Where there's no equivalent library now, having something that's only 5-10x slower to call – but still runs fast and matches API – may be just fine. Think about the coarse-grained operations you feed to a MySQL or SQLite and you get the picture.

Garbage Collection: JRuby is able to use the generational garbage collector available on the JVM.

Misc: JRuby is able to use a JIT compiler to enhance code execution performance.

2.2.4.5 Rubinius

The Rubinius virtual machine is written in C++. It uses LLVM to compile bytecode to machine code at runtime. The bytecode compiler and vast majority of the core classes are written in pure Ruby. The achieved performance and functionality gains over the last few months make Rubinius one of the most promising Virtual Machines for the Ruby programming language.

Threads: Rubinius uses operating systems threads in combination with a global interpreter lock

C Extensions: Rubinius supports C-extensions (with or without FFI)

Garbage Collection: Rubinius uses a precise, compacting, generational garbage collector

[9] `http://blog.headius.com/2010/07/what-jruby-c-extension-support-means-to.html`

Misc: Rubinius features a JIT compiler. At the time of writing, it is a bit behind Ruby 1.9 or JRuby in terms of performance.

2.2.4.6 Misc

At the time of writing, there are other Implementations such as IronRuby, a .NET implementation of the Ruby programming language, or Maglev, a Ruby implementation with integrated object persistence and distributed shared cache. While Maglev shows some nice performance numbers, it is still considered alpha and should not be used in a production environment at the moment.

2.2.4.7 Conclusion for the project

The complete support for C extensions and the performance enhancements over MRI (Ruby 1.8) make YARV (Ruby 1.9) the main deployment platform for the project. Remaining compatibility with JRuby and Rubinius is an interesting option, but it is not a high priority. Especially with the upcoming support for C Extensions in JRuby and the progress in Rubinius, the compatibility between the different VMs should reach almost 100% in the near future.

2.3 I/O model (async vs threading)

I/O related waits are one of the major performance problems when designing a crawler. Network I/O is several orders of magnitude slower than disk I/O and almost unpredictable in terms of request duration. A single request to a website involves not only an HTTP request, but also DNS resolution. Both of these steps can take from a few milliseconds to several seconds, depending on the location of the target URL and the performance of the respective servers.

An additional problem is that some of the domains that show up as links on the web are either non-existent or in a segment of the internet that is not reliably reachable from the Amazon EC2 network. Doing requests to these systems in a serial manner will result in very low throughput. A solution to this problem is running more than one request in parallel. While it does not solve the problem of web servers that have a large roundtrip time, it keeps connections to these servers from completely blocking other operations.

The two ways of introducing parallelism to the crawling process are either the use of threads or some form of non-blocking, asynchronous I/O conforming to the reactor/proactor pattern. A discussion of the implications of asynchronous I/O can be found in my paper "Event-Driven I/O - A hands-on introduction"[9]. Especially because of the VM limitations concerning threads, an asynchronous approach using the eventmachine framework[10] and libraries such as em-http-request[11] , em-redis[12] and em-mysqlplus[13] seemed beneficial to optimizing the system's throughput. There are voices in the developer community that question the usefulness of non-blocking I/O in comparison to the top of the line Virtual Machine and Operating System threading models. Especially for Java, Paul Tyma has an excellent presentation up on his blog called "Thousands of Threads and Blocking I/O - The old way to write Java Servers is New again (and way better)"[10].

While his presentation is controversial, it offers good insights in the changes to thread-costs that happened over the last few years. In the case of the crawler project, however, these findings cannot be 100% transferred. As mentioned in the Ruby section of this thesis (2.2), the existence of a global interpreter lock makes threading a bit of a problem. While quasi-parallel execution in C-based extensions is still possible by unlocking the GIL, a single extension not doing this can stop the whole crawler just because of, for example, an unresponsive DNS server.

[10]`http://rubyeventmachine.com/`

[11]`http://github.com/igrigorik/em-http-request`

[12]`http://github.com/madsimian/em-redis`

[13]`http://github.com/igrigorik/em-mysqlplus`

A series of great posts about this topic can be found on Ilya Grigorik's blog[14]. Especially his presentation "No Callbacks, No Threads: Async & Cooperative Web Servers with Ruby 1.9"[11] does a great job explaining the problems and drawing comparisons to other popular frameworks that deal with asynchronous I/O (such as Node.JS). For the project, the decision was to go asynchronous where needed and stay synchronous and maybe introduce threading where performance is not the major issue. While running synchronous code does stop the reactor loop in eventmachine, CPU-bound operations take so little time that the overhead in comparison to I/O operations can be ignored.

2.4 Amazon Elastic Compute Cloud

One of the requirements for the project was the ability for it to run on Amazon Elastic Compute Cloud (also known as "EC2"). Amazon EC2 is a service by Amazon that allows companies (and private users) to create and boot up virtual servers. These servers are hosted in Amazon data centres and based on the XEN virtualization technology.

2.4.1 Instance Types

Amazon offers several different hardware configurations. These are the configurations from the Amazon Instance Types Webpage[15] at the time of writing. Note: According to Amazon[16], "One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor."

[14]`http://www.igvita.com`
[15]`http://aws.amazon.com/ec2/instance-types/`
[16]`http://aws.amazon.com/ec2/`

Name	**RAM**	**EC2 Compute Units**	**storage**	**bit**	**I/O**	**$/h**
m1.small	1.7 GB	1	160 GB	32	moderate	0.085
m1.large	7.5 GB	4 (2 cores *2 units)	850 GB	64	high	0.34
m1.xlarge	15 GB	8 (4 cores * 2 units)	1690 GB	64	high	0.68
m2.xlarge	17.1 GB	6.5 (2 cores * 3.25 units)	420 GB	64	moderate	0.50
m2.2xlarge	34.2 GB	13 (4 cores * 3.25 units)	850 GB	64	high	1.00
m2.4xlarge	68.4 GB	26 (8 cores * 3.25 units)	1690 GB	64	high	2.00
c1.medium	1.7 GB	5 (2 cores * 2.5 units)	350 GB	32	moderate	0.17
c1.xlarge	7 GB	20 (8 cores * 2.5 units)	1690 GB	64	high	0.68
cc1.4xlarge	23 GB	2*Xeon X5570,quad-core	1690 GB	64	very high	1.60

One of the important limitations compared to other solutions is the lack of small instances with a 64 bit architecture in Amazon's lineup. A lot of back-end systems that use memory mapped I/O need a 64 bit operating system to work with large amounts of data. The MongoDB developers for example had this to say[17]:

> 32-bit MongoDB processes are limited to about 2.5 gb of data. This has come as a surprise to a lot of people who are used to not having to worry about that. The reason for this is that the MongoDB storage engine uses memory-mapped files for performance. By not supporting more than 2gb on 32-bit, we've been able to keep our code much simpler and cleaner. This greatly reduces the number of bugs, and reduces the time that we need to release a 1.0 product. The world is moving toward all 64-bit very quickly. Right now there aren't too many people for whom 64-bit is a problem, and in the long term, we think this will be a non-issue.

This makes MongoDB basically unusable on anything below a EC2 large instance. While most other solutions tend to work on 32 bit, using a EC2 large instance is highly beneficial for I/O throughput, provides a

[17] `http://blog.mongodb.org/post/137788967/32-bit-limitations`

64 bit environment and thus offers better database performance. The Cassandra wiki describes the situation as follows when it comes to the data access mode[18]:

> mmapped i/o is substantially faster, but only practical on a 64bit machine (which notably does not include EC2 "small" instances) or relatively small datasets. "auto", the safe choice, will enable mmapping on a 64bit JVM. Other values are "mmap", "mmap_index_only" (which may allow you to get part of the benefits of mmap on a 32bit machine by mmapping only index files) and "standard". (The buffer size settings that follow only apply to standard, non-mmapped I/O.)

The downside of this is that the smallest 64bit-Instance available on EC2 today is the m1.large instance. This instance is four times as expensive as the m1.small. While other "cloud hosting" providers offer 64 bit systems for smaller VM instances (e.g. Rackspace), using Amazon EC2 allows the project to leverage the knowledge of a large developer community. In this case, using Amazon EC2 is also beneficial because of Acquia's previous experience with the service and the existing infrastructure within the company.

2.4.2 Elastic Block Store

Amazon Elastic Block Store (EBS) provides block level storage volumes for use with Amazon EC2 instances. In contrast to an instances "ephemeral storage", EBS survives a shutdown or crash of the machine. This is why Important data should always be saved on Amazon's Elastic Block Store when using EC2. An EBS device can be mounted just like a regular blockdevice (even as a boot volume) and has the ability to create snapshots, provided that the file system supports it (XFS for example does). Using it as a a boot volume also allows switching between instance sizes (e.g. from small to medium) without having to set up all services again. The

[18] `http://wiki.apache.org/cassandra/StorageConfiguration`

problem with these upgrades is, that switching from a small EC2 instance to a large EC2 instance also means switching from 32 bit to 64 bit. This is usually not a good idea and a reinstallation is highly recommended.

Another interesting possibility when using EBS is the ability to use more than one volume and add them together in a RAID configuration. There are several benchmarks that prove the performance increase when doing so. The MySQL performance blog has a comparison between a single EBS volume and an EBS RAID configuration[19], victortrac.com offers a comparison of EBS against the ephermal disks[20] and the heroku blog[21] offers more EBS specific benchmarks. More about the problems with benchmarking EBS and ephemeral storage can be found in the performance section (2.4.3) of this chapter. Many articles about EBS highlight, that access times and throughput vary greatly depending on the load of other instances on the same host node. This means that while EBS can be faster than ephemeral storage, it is not consistent. Therefore its main advantage is the ability to have persistent data and the possible usage of a snapshot mechanism.

2.4.3 Performance

One major factor in the design of the crawler is the way that the EC2 platform virtualizes I/O operations. The back-end load against the database is one of the key limitations for big parts of the project. While most database storage engines are able to cache frequently requested data, the amount of domains that will be collected will result in frequent cache misses.

A good overview of the I/O behaviour of virtualized "cloud" servers

[19]http://www.mysqlperformanceblog.com/2009/08/06/ec2ebs-single-and-raid-volumes-io-bencmark/

[20]http://victortrac.com/EC2_Ephemeral_Disks_vs_EBS_Volumes

[21]http://orion.heroku.com/past/2009/7/29/io_performance_on_ebs/

can be found at the cloudharmony blog[22]. In our case, using a single EC2 large instance seems to be the right choice when it comes to I/O throughput. In general, an actual non-virtualized hard disc would be beneficial when it comes to performance.

When reading benchmarks, it is important to pay attention to the official Amazon EC2 documentation[23]. It mentions a penalty for the first write to a block on the virtualized I/O devices. If people don't pay attention to this, benchmarks will produce invalid results. This is the matching quote from the documentation:

> Due to how Amazon EC2 virtualizes disks, the first write to any location on an instance's drives performs slower than subsequent writes. For most applications, amortizing this cost over the lifetime of the instance is acceptable. However, if you require high disk performance, we recommend initializing drives by writing once to every drive location before production use.

[22]`http://blog.cloudharmony.com/2010/06/disk-io-benchmarking-in-cloud.html`

[23]`http://docs.amazonwebservices.com/AWSEC2/latest/DeveloperGuide/index.html?instance-storage-using.html`

Chapter 3

Back-end and Search

This chapter goes into detail about the different possibilities of persisting data and the algorithms behind it. It looks at ways to make the saved data searchable and evaluates current software-solutions to both problems.

3.1 Datastores

3.1.1 Categorization

One of the biggest problems in designing a crawler is the actual storage of data. An estimated dataset with well over 100 million domains that has a high number of reads and writes requires some further research when it comes to possible storage back-ends. The main idea of this chapter is to outline the possible technologies that store data in a persistent way. When it comes to persistently storing data, we can currently distinguish among 5 main categories of data-stores:

1. Relational database management systems
2. Column stores
3. Document stores
4. Key-Value stores

5. Graph databases

All of them offer a way to persist a document over an application restart and make it network accessible. When quoting examples, this paper will mostly concentrate on open-source projects that have an active community.

3.1.1.1 Relational database management systems

The family of RDBMS[1] are based on the idea that data can be represented in a collection of related tables that store their data in columns and rows. This idea was first coined by Edgar F. Codd in his paper "A Relational Model of Data for Large Shared Data Banks"[1]. Most RDBMS focus on being able to provide consistent data and the ability to enforce specific rules on data types, relational constraints, and the ability to do ad-hoc queries. For larger datasets, this is usually enabled by indexing the entries using a data structure such as a b-tree. The implications of the specific data structures will be discussed in chapter 3.2.

Most relational database management systems use SQL, the Structured Query Language, as the primary way for developers to filter out a specific subset of data. While features such as transactions or full consistency for data can be fine-tuned, they still add to the systems complexity and have a negative impact on the performance of writes and reads. The major free implementations of RDBMS are MySQL and ProstgreSQL and will be discussed in section 3.5.3. While there are also relational databases that are available as software as a service (e.g. Amazon RDS), using them would require access over a network connection. While this would help with distributing the project, it would also limit the maximum throughput compared to a local instance of a database.

[1]short for: Relational database management systems

3.1.1.2 Column Stores

Column Stores manage their data inside of columns. They differ from the relational schema of RDBMs in that they do not use rows, meaning that the different saved records don't need to share a common schema. They still offer some internal structure when compared to pure Key-Value stores. Another difference between Column stores and RDBMs is, that most of the available open-source Column Store solutions don't focus on ACID properties or constraints between different sets of data. A detailed explanation of these differences can be found in the paper "Column-Stores vs. Row-Stores: How Different Are They Really?" by Abadi, Madden and Hachem[6] This largest open-source projects providing a column store are Hadoop/HBase[2], Cassandra[3] and Hypertable[4]

3.1.1.3 Document Stores

Systems that can be described as "document stores" actually have knowledge about the data that they store. They usually are able to do basic operations (e.g. increment an integer inside of a document) and usually support map/reduce-type operations. Some (e.g. MongoDB) even offer "advanced" data structures such as arrays and hashes. The most popular Document stores at the time of writing are CouchDB[5] and MongoDB[6].

3.1.1.4 Key-Value Stores

Key-Value stores offer the simplest model of data storage. They basically provide a networked persistence to what is commonly known as an associative array (Ruby/Perl hashes, Python dictionaries, Java Hashtable). Most of them offer only three operations:

1. put

[2]`http://hbase.apache.org/`
[3]`http://cassandra.apache.org/`
[4]`http://hypertable.org/`
[5]`http://couchdb.apache.org/`
[6]`http://www.mongodb.org/`

2. get
3. delete

One of the bigger examples often quoted in literature is Amazon's Dynamo, a distributed key-value store described in [23]. It is also cited as a design inspiration by a lot of the bigger Key-Value store projects. Popular open-source projects include Project Voldemort[7], Riak[8], Scalaris[9], Berkley DB[10] and to some extend Redis[11] (Redis also offers arrays and some other data structures with the respective operations). Amazon SimpleDB is a commercial offering by Amazon that provides a Key-Value interface to data storage as software as a service.

Since Key-Value stores do not offer features such as complex query filters, joins, foreign key constraints, sorting or triggers, their performance is easily predictable and mainly a function of the amount of stored keys (given that there is enough RAM to store the basic index structures). Since the keys themselves don't have an internal relation to one another, scaling key value stores vertically is an easier task than doing the same for their RDBMS counterparts. Key-Value stores are often used in combination with an object serialization format such as Protocol Buffers, Thrift, BERT, BSON, or plain JSON. These formats help storing complex objects as a regular value.

3.1.1.5 Graph Databases

While there is a certain subset of graph databases that have specialized in storing only certain graphs (triplestores, network databases...), we are only paying attention to generic graph databases that can store arbitrary graphs. These graph databases usually consist of 3 basic building blocks:

[7] `http://project-voldemort.com/`
[8] `https://wiki.basho.com/display/RIAK/Riak`
[9] `http://code.google.com/p/scalaris/`
[10] `http://www.oracle.com/technology/products/berkeley-db/index.html`
[11] `http://code.google.com/p/redis/`

- Nodes
- Edges
- Properties

These items are combined to represent real world data. Graph databases are optimized for associative data sets. They are a good fit for uses such as tagging or metadata annotations. The crawler data would be an interesting fit since it is basically just a tagging of domain-nodes with several different data attributes (cms name, web server name, top level domain, ...). Querying the data inside of a Graph Database usually means traversing the graph along the edges of the nodes' relationships. This works fine for general "has this TLD" or "is powered by" relations, but it is harder to model things like the amount of incoming links. While setting it as a property does work, a search for every domain with more than 300 links would require an extended data schema or the support of a dedicated search library. One of the biggest open-source projects implementing a graph database is Neo4j[12].

3.1.2 Data store scalability

A key metric when evaluating data stores is the way they are able to deal with a large number of documents. There are mainly two ways of doing this:

1. Vertical scaling
2. Horizontal scaling

Vertical scaling is the ability to enhance performance by simply running on a faster processor, more RAM, or faster storage. This scalability completely focuses on a single-node operation. The higher the throughput and the more optimized the algorithms behind the system, the less urgent it is to actually have the need for horizontal scalability. In our

[12] `http://neo4j.org/`

case, being able to store, search, and query 100 million documents with a modest amount of fields is a target that would allow us to keep all of the back-end operation on a single machine, and only add additional machines when we want a higher HTTP/analytical throughput. Horizontal scalability describes the ability of the system to distribute data over more than a single computer. In our project, this should happen in a transparent way that does not require time-intensive or overly complex administrative tasks when dealing with backups or software updates.

3.2 Datastructures

This section of the thesis is devoted to giving a high-level overview about data structres that can usually be found in storage systems and the implication of using them.

3.2.0.1 B-trees

In general, B-trees are balanced trees that are optimized for situations in which there is not enough RAM to keep all of the data structure in memory, and parts of it have to be maintained on a block device (e.g. a magnetic hard-disc). B-Trees allow efficient insertion, updates, and removal of items that are identified by a key. In our case, this would most likely be the domain name.

The most common representative of the B-Tree family in data-storage systems is the B^+-Tree. The main difference between a B-Tree and a B^+-Tree is that one is not allowed to store keys in one of the in a B^+-Tree's leaves. They are reserved for data only. The advantage of a B^+-Tree over a regular B-Tree is that it tends to have a large fan-out (number of child nodes). This results in fewer indirections and thus fewer I/O operations to reach a certain piece of data. This is especially helpful with block based devices and one of the reasons why many file systems (NTFS, ReiserFS, XFS, JFS) use B^+-Trees for indexing metadata.

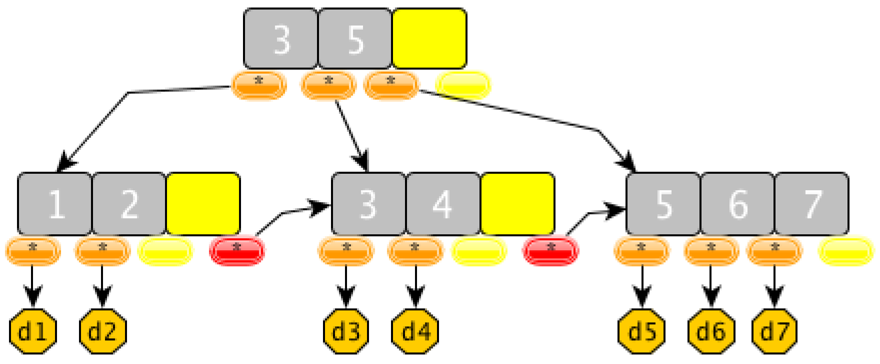

Figure 3.1: B^+-Tree mapping the numbers 1..7 to d1..7

According to Shaffer[17], when it comes to database systems, the B-tree and its variations (B^+-Tree, B*Tree, ...) are:

> [...] the standard file organization for applications requiring insertion, deletion, and key range searches. B-trees address effectively all of the major problems encountered when implementing disk-based search trees: 1. B-trees are always height balanced, with all leaf nodes at the same level. 2. Update and search operations affect only a few disk blocks. The fewer the number of disk blocks affected, the less disk I/O is required. 3. B-trees keep related records (that is, records with similar key values) on the same disk block, which helps to minimize disk I/O on searches due to locality of reference.

3.2.1 Hash-based

The idea behind a hash-based index is that the position of an arbitrary document in a key-value store can be calculated easily. By giving the key as the input to a special hashing function, it returns the documents/values position allowing the system to jump directly to the memory-page in question. After it arrives at the page, it does a simple linear scan to find the key. The page scan is necessary because storing only a single value per page (hence per key) would lead to extremely small pages, which would lead to a huge amount of necessary management information. Feature-wise, hash-indexes usually do not support range queries like "greater than". There are exceptions to this rule though: Cassandra uses an order-preserving hash system that allows these query elements. An interesting performance comparison between Hash and B^+-Tree can be found in the Berkley DB documentation[13]:

> There is little difference in performance between the Hash and Btree access methods on small data sets, where all, or most of, the data set fits into the cache. However, when a data set is large enough that significant numbers of data pages no longer fit into the cache, then the Btree locality of reference described previously becomes important for performance reasons. For example, there is no locality of reference for the Hash access method, and so key "AAAAA" is as likely to be stored on the same database page with key "ZZZZZ" as with key "AAAAB". In the Btree access method, because items are sorted, key "AAAAA" is far more likely to be near key "AAAAB" than key "ZZZZZ". So, if the application exhibits locality of reference in its data requests, then the Btree page read into the cache to satisfy a request for key "AAAAA" is much more likely to be useful to satisfy subsequent requests from the application than the Hash page read into the cache to satisfy the same request. This means that for applications with locality of reference, the cache is generally much more effective for the Btree access method than the Hash access method, and the Btree access method will make many fewer I/O calls. However, when a data set becomes even larger, the Hash access method can outperform the Btree access method. The reason for this is that Btrees contain more

[13] `http://www.oracle.com/technology/documentation/berkeley-db/db/programmer_reference/am_conf_select.html`

> metadata pages than Hash databases. The data set can grow so large that metadata pages begin to dominate the cache for the Btree access method. If this happens, the Btree can be forced to do an I/O for each data request because the probability that any particular data page is already in the cache becomes quite small. Because the Hash access method has fewer metadata pages, its cache stays "hotter" longer in the presence of large data sets. In addition, once the data set is so large that both the Btree and Hash access methods are almost certainly doing an I/O for each random data request, the fact that Hash does not have to walk several internal pages as part of a key search becomes a performance advantage for the Hash access method as well.

The Postgres 8 documentation is a little bit more definitive in its wording when it comes to Hash Indexes[14]:

> Note: Testing has shown PostgreSQL's hash indexes to perform no better than B-tree indexes, and the index size and build time for hash indexes is much worse. For these reasons, hash index use is presently discouraged.

As can be seen, the specific implementation of the indexes, the amount of stored data, and the access patterns all change the performance of the solution in question. This is why benchmarking with actual data and real-world load is an important step in the progress of developing this project. Synthetic benchmarks will probably not provide exact results in this case.

3.2.2 R-tree-based

While this data-structure is not of primary interest for our project, R-Trees show up in data-storage systems from time to time, so knowing what they do without going into too much detail seems beneficial. R-trees are

[14]`http://www.postgresql.org/docs/8.0/interactive/indexes-types.html`

data structures similar to B-Trees. They are primarily used for indexing multi-dimensional data such as geo-coordinates. They got described by Antonin Guttman (UC Berkley) in his 1984 paper "R-Trees - A Dynamic Index Structure for Spatial Searching"[22]. CouchDB, for example, has a fork called "GeoCouch" that uses R-Trees for geospatial indexing. SQLite also offers R*Tree support[15] as a compile time option. There are also other options for indexing geo-spatial data. MongoDB, for example, offers geospatial indexing[16] using a geographic hash code on top of standard MongoDB B^+-Trees. However, the MongoDB manual entry mentions that:

> The problem with geo-hashing is that prefix lookups don't give you exact results, especially around bit flip areas. MongoDB solves this by doing a grid by grid search after the initial prefix scan. This guarantees performance remains very high while providing correct results.

3.2.3 Merkle-tree-based

Merkle trees (also known has "Hash-trees") are not primarily used to store data efficiently. They are a type of data structure that encodes summary information about a larger piece of data in tree form. This information can be used to verify data. Especially in a distributed data-storage systems, this can be used to detect inconsistencies between replicas faster, while also minimizing the amount of transferred data. This is often described as an "anti entropy" process. More information is provided by the Amazon Dynamo paper[23] in chapter 4.7 "Handling permanent failures: Replica synchronization". The dynamo-based data-stores Cassandra and Riak also provide this feature. The Riak team mentions merkle trees in their architecture document[17]:

[15]http://www.sqlite.org/rtree.html
[16]http://www.mongodb.org/display/DOCS/Geospatial+Indexing
[17]http://riak.basho.com/edoc/architecture.txt

> Riak attempts to move data toward a consistent state across nodes, but it doesn't do so by comparing each and every object on each node. Instead, nodes gossip a "merkle tree", which allows them to quickly decide which values need comparing.

The Cassandra documentation mentions a difference between the original dynamo model and the one in Cassadra as follows[18]:

> The key difference in Cassandra's implementation of anti-entropy is that the Merkle trees are built per column family, and they are not maintained for longer than it takes to send them to neighboring nodes. Instead, the trees are generated as snapshots of the dataset during major compactions: this means that excess data might be sent across the network, but it saves local disk I/O, and is preferable for very large datasets.

Other uses of hash trees can be found in the ZFS filesystem[19] and also in the Google Wave protocol[20]

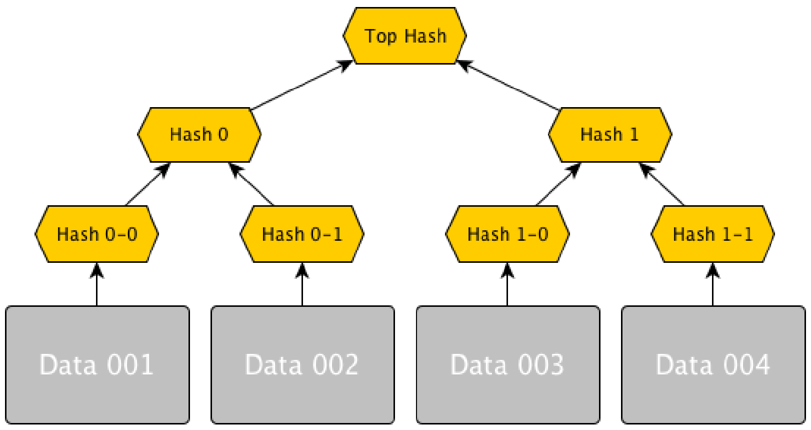

Figure 3.2: Merkle-Tree

[18]http://wiki.apache.org/cassandra/AntiEntropy
[19]http://blogs.sun.com/bonwick/entry/zfs_end_to_end_data
[20]http://www.waveprotocol.org/whitepapers/wave-protocol-verification

3.2.4 Trie-based

A trie[21], also known as a prefix-tree, is an ordered tree data structure optimized for storage of key-value data. The basic idea behind storing strings in a trie is that each successive letter is stored as a separate node in the tree. To find out if the word ’car’ is in the list of stored words, one starts at the root and looks up the ’c’ node. After having located the ’c’ node, c’s children are searched for an ’a’ node, and so on. To differentiate between ’car’ and ’carpet’, each completed word is ended by a previously defined delimiter. The definition of a Trie by the Information Technology Laboratory by the US National Institute of Standards and Technology (NIST) is the following[22]:

> Definition: A tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes.

A typical use case is the storage of language dictionaries used for spell checking or input fields that should auto-complete (e.g. eMail addresses in a mail program). An example of a software using tries to manage data is Lucene, the library powering the Solr search service. Since Lucene treats most data internally as strings, working with numeric ranges can be greatly accelerated by using the Trie data structure to store the values. Figure 3.4 shows how a range query can be mapped to a Trie. For further details, Edward Fredkin’s original paper about Tries called "Trie Memory"[20] should be a good starting point.

[21]The origin of the term *trie* stems from the word "retrieval"

[22]`http://www.itl.nist.gov/div897/sqg/dads/HTML/trie.html`

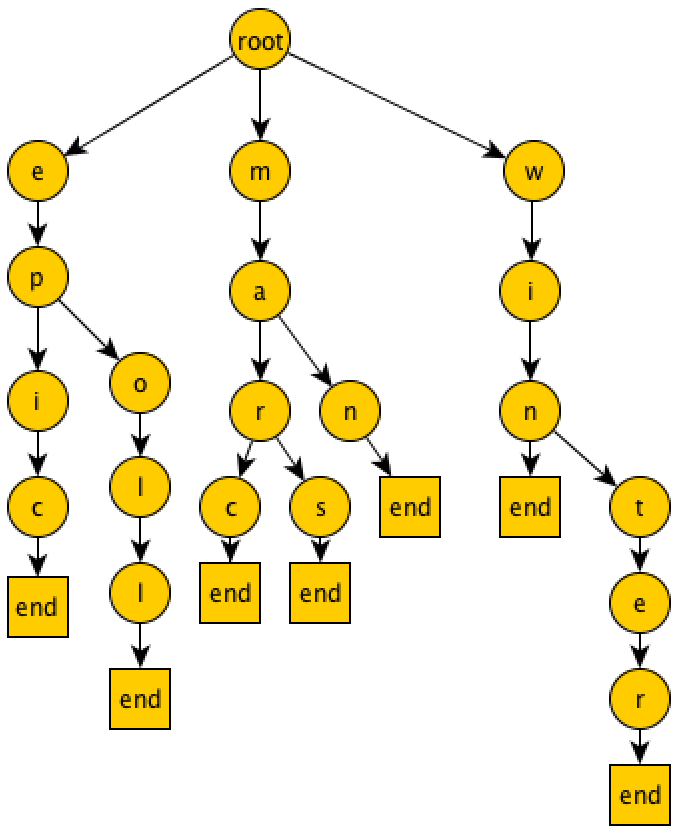

Figure 3.3: Trie storing the words: epic, epoll, marc, mars, man, win, winter

Compared to regular Binary Search Trees, the key lookup in a Trie is faster. The Trie takes a maximum of O(m) operations where m is the length of the key while a Binary Search Tree uses O(m * log n) also depending on n, the amount of entries in the tree. Another advantage is the space savings that a Trie has in comparison to a Binary Search Tree when using a large number of small strings as keys. This is because the keys can share common prefixes, and only use that space once, no matter how many strings there are.

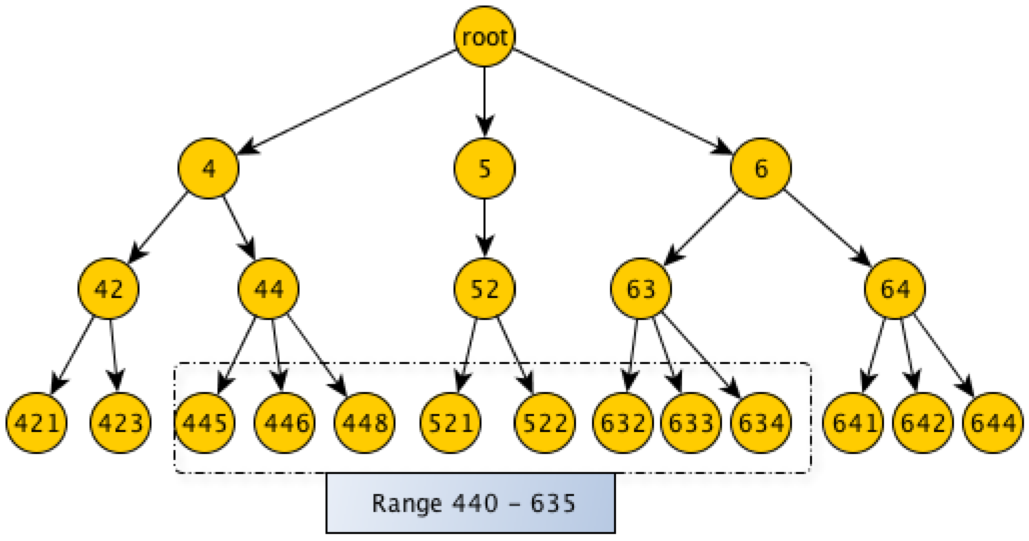

Figure 3.4: Range query in a Trie

3.2.5 Bitmap-based

A bitmap index is a special kind of index that has traditionally been used for data-entries that only have a limited amount of possible values. A bitmap index uses bit arrays as its main data-structure and can answer most queries by performing bitwise logical operations on them. They have the nice property that multiple bitmap indexes can be merged into one combined index to resolve simple queries over multiple fields. An example of a simple AND query can be seen in figure 3.5. Bitmap indexes were usually used for low cardinality values such as gender, weekday etc. In this case, a bitmap index can have a large performance advantage over commonly used B$^+$-Trees. In "An Adequate Design for Large Data Warehouse Systems: Bitmap index versus B-tree index" by Zaker et al [19], the authors go even as far as saying that for certain situations, the cardinality of the values does not matter. Their conclusive statement is as follows:

> Thus, we conclude that Bitmap index is the conclusive choice for a DW designing no matter for columns with high or low

cardinality.

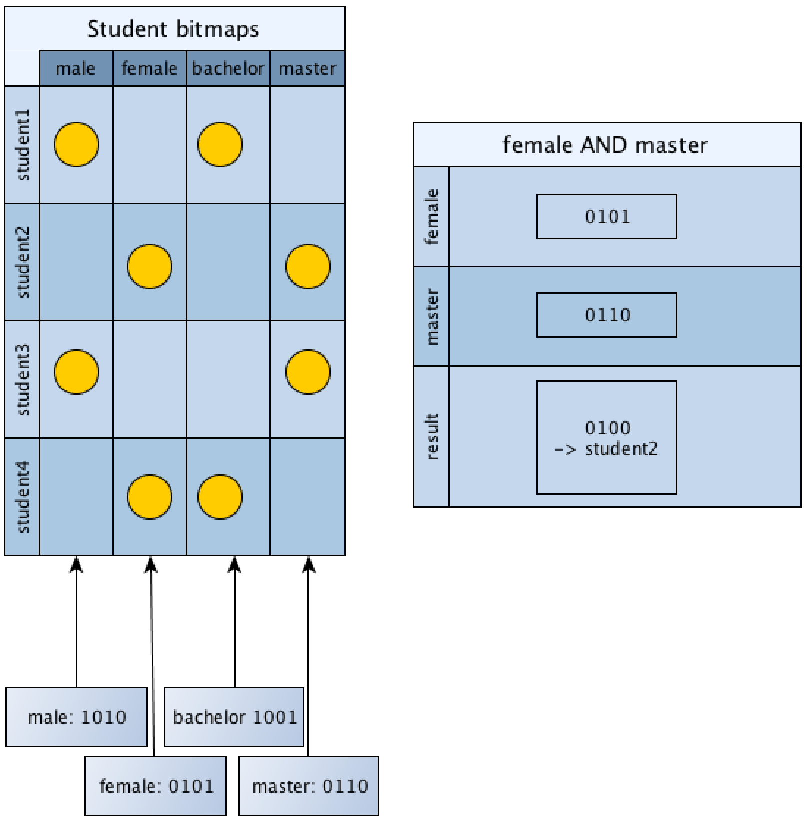

Figure 3.5: Bitmap-Index

Interestingly, bitmap indexes are barely implemented in current data-storage systems. While there are some exceptions within the SQL space (e.g. Oracle[23]), and internal usage in special search solutions (e.g. Lucene's OpenBitSet class). Sadly, none of the common open-source data stores in the SQL space (e.g. MySQL, PostgreSQL) or the NoSQL space offer user-definable bitmap indexes to the end user so far. Some search solutions (e.g. Solr, Sphinx, PostgreSQL) are using them internally, but

[23]`http://download.oracle.com/docs/cd/B13789_01/server.101/b10736/indexes.htm`

offer only limited user control at the time of writing. A typical use-case of bitmap indexes can be found in PostgreSQL, which uses them to combine B-Tree indexes when using AND or OR queries[24]. More details about bitmap indexes can also be found in Chan and Ioannidi's "Bitmap index design and evaluation"[18].

An interesting project that provides a highly optimized bitmap index implementation is "FastBit"[25] created by UC Berkley in connection with the U.S. Department of Energy. The project has a lot of available publications[26] explaining the details of its implementation such as its compression algorithm for bitmaps and other technical properties of the system.

3.3 Generic Problems

This section of the paper is dedicated to describing some commonly seen problems with storage solutions that could be found while evaluating specific software products. They seem to be common problems and thus should be checked for every new product.

3.3.1 HTTP persistent connections

HTTP persistent connections are also known as "HTTP keep-alive". The idea behind them is to use the same TCP connection to send and receive multiple HTTP requests/responses, as opposed to opening a new connection for every single request/response pair. Especially when working with data stores that have an HTTP based interface (e.g. CouchDB), it is important to rely on a HTTP library that is able to keep the TCP connection to the data store open. Some libraries establish a new connection for each request, resulting in a lot of overhead. For Ruby, the Patron

[24]http://www.postgresql.org/docs/current/static/indexes-bitmap-scans.html

[25]https://sdm.lbl.gov/fastbit/

[26]http://crd.lbl.gov/~kewu/fastbit/publications.html

project[27] is an HTTP library that is based on libcurl and supports the creation of session objects that use libcurl's persistent HTTP connections.

3.3.2 Locking

One of the biggest problems with storage systems, in the case of our project, is that the system will have a lot of mixed writes and reads in a highly parallel fashion. Some systems like MongoDB or MySQL with the MyISAM storage engine are locking the whole database (or Database table) when updating a single document/row. This leads to a situation where a lot of parallel processes pile up requests for hundreds of other domains just because of an update to a single domain. This cuts down the performance of the whole system. Especially when using distributed crawlers, this could lead to latency even worsening that situation. Possible solutions for this are:

- Using a storage back-end that has some form of optimistic locking or multi-version concurrency control. Systems which provide this include CouchDB , Riak, and the InnoDB storage engine (used with MySQL).
- Using a queueing system that can buffer write requests. While it would put the whole system into an eventually consistent state, it would at least help to take wait-time away from the crawling processes by buffering some operations (e.g. writing updates, inserting results)
- Using caches for the high-volume requests. Links to prominent domains like twitter.com that are encountered (and thus checked) frequently could be kept in a LRU-cache, a cache that discards the **l**east **r**ecently **u**sed items first, and be answered locally instead of reaching out to the database. This would keep the total amount of read-requests down. The problem with this approach is that the cached requests can't be recorded to the data-storage without

[27] `http://github.com/toland/patron`

adding an additional write. This makes it hard to keep track of the number of incoming links to a certain domain. Incoming links are a good metric for the domain's popularity. A possible solution is to keep track of hits locally and write out the new number in exponential intervals.

It has to be noted that looking at the internals of the databases' locking mechanism is an important step in evaluating the storage back-end. While MySQL using MyISAM does have table level locks, MyISAM offers things like concurrent Inserts[28] that allow new inserts to run while a 'SELECT" call is in progress. These options might make the usage of large grained locks tolerable. Especially with MySQL, the amount of tuning options is overwhelming at first, and a lot can be gained by understanding them. A good source for this kind of information is "High Performance MySQL, Second Edition" that got published by O'Reilly Media[21]. More discussion about MySQL can be found in the respective section (3.5.3) of this thesis.

3.3.3 Append-only storage and compacting

Some databases use an append-only file format that, instead of changing an old document, will create a new document when writing updates to the database. This allows the database to keep the old data in a consistent state in case of a crash during the write process. The problem with some of these systems is that they have to "compact" the data-files on disc from time to time to clean the old versions of the file that are not in use any more. CouchDB, for example, uses a simple approach by just traversing over the whole file and copying the most recent version of each record into a new file. After the traversal is done, an atomic switch to the new file happens. The problem with this approach is that it slows down performance heavily when being used on a system with a lot of updates (e.g. incoming link counters) and a slow I/O subsystem (e.g. virtualized EC2 storage). In my experiments, inserts were coming in at

[28]http://dev.mysql.com/doc/refman/5.0/en/concurrent-inserts.html

such a fast rate that the compaction process took days to finish while still doubling the space requirements of the data on disk. This holds especially true when other I/O intensive operations (e.g. backups) are running at the same time.

3.4 Search possibilities

This section will give an overview about the different techniques to make a large data collection searchable and the resulting implications.

3.4.1 Classification

One of the important features when choosing a storage back-end is the ability to filter our data according to certain criteria. There are three major categories of searches:

1. Specific domain Searches for a specific domain and its connected data. Example: A search for all saved details about "example.org"

2. Property combinations. Searches for a combination of saved properties. Example: All Drupal sites running on the IIS web server that are ranked among the top 1 million websites worldwide according to Alexa

3. Fulltext search. A search for all documents that have a certain substring in a specified field. An example would be "All domains that have the word ’cake’ in their domain name"

When searching for a specific domain, all of the previously discussed data-stores offer a primary key lookup. In our case, this lookup would be a way of getting the data referenced by a website’s domain name. This is the minimum a system should be able to do for us.

Property combinations require a bit more of a data schema or computing capacity to work efficiently. While using map-reduce to simply iterate over all of the gathered domains would result in the correct answer, it would also take a long time unless being done on a large cluster. This can be optimized by only looking at a certain subset of domains. In our example case, simply keeping a list for each Content Management System with the matching domain names would greatly reduce the total amount of documents that would need to be scanned from "all domains" to "all domains in the drupal-collection". The downside of this approach is, that managing these collections adds extra complexity to the application logic. An automated way of doing this internally is provided by some of the back-ends (most RDBMS, search solutions like lucene/solr or sphinx, some document stores like MongoDB). The RAM requirements for keeping these indexes in memory grow with the total amount of collected domains. The more "free form" your queries are, the more indexes one has to add to the data in order to keep query performance at an acceptable limit. An interesting solution for "fixed" queries is offered by CouchDB's incrementally updated views which require only one index per view and spread the processing power that is needed over every insert/update/delete operation. Aside from RDBMS and Document Stores, Graph databases are also a nice fit when querying for single attributes, since their natural data-model is highly optimized for this kind of situation.

Fulltext search: Filtering data for a field that has a specific substring with a wildcard in the end of the term (example*) usually just requires a back-end that organizes the searchable data in B+Trees. This is available in most data-stores that go beyond the simple key-value model. Examples would be MongoDB, MySQL or Tokyo Cabinet. If there is a leading wildcard, the data-store would usually have to reverse the field in question ("example.com" -> "moc.elpmaxe") and save the inverted field into a seperate B+Tree. This behaviour is offered by Solr with the use of the "reversedwildcardfilterfactory".

3.4.2 Indexation

One of the most common ways to keep data searchable is the usage of so called "indexes". The "inverted index" is a data structure that is used to store a mapping from content (e.g. a word or a primary key) to a position inside of the database. This enables a database system to move away from full table scans to simply accessing the term that is looked for directly with as few redirections as possible. More about the commonly used data structures to implement these reverse indexes (e.g. B$^+$-Trees, Bitmap-Indexes) can be found in section 3.2. An interesting use case of these indexes is the ability to combine them using boolean logic. A simple example would be the search for "term A AND term A". To solve this query, the database only has to look up the resulting positions for a search for term A and simply do an AND merge of this list with the results for term B. After merging the intersection of these lists using a logical AND, the resulting list points to documents that feature both terms.

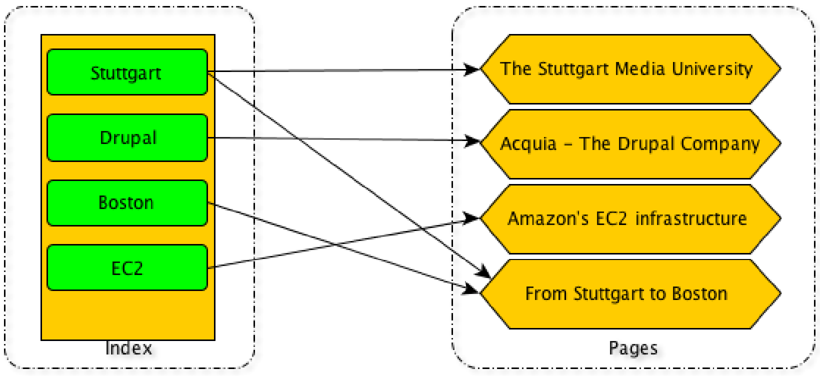
Stuttgart
Drupal
Boston
EC2
Index
The Stuttgart Media University
Acquia - The Drupal Company
Amazon's EC2 infrastructure
From Stuttgart to Boston
Pages

Figure 3.6: Reverse Index

One of the important things to keep in mind is the RAM and CPU usage of these data-structures. For every added or deleted entry to the data-store, all corresponding indexes have to be updated. Especially when trying to make all of the fields in the data searchable, RAM usage might become a problem when operating on a single node. In early experiments with MongoDB, the use of a complete set of indexes led to MongoDB having to swap indexes in and out of RAM every few seconds, keeping a database lock in the process and decimating the possible operations per second. Possible solutions for this problem are:

- Relying on a data store that supports sparse indexes. These indexes are able to only insert a certain subset of data (e.g. fileds that are not NULL). This can help if a lot of the values are not available in every document (in our case: CMS specific data). Otherwise, indexation of a seldom used column would lead to a data structure having to carry NULL values for the documents that do not have the field (our worst case: a B^+-Tree with 100 million entries compared to one with 10000). The downside of this is, that searches for entries that are in the not-indexed part of the sparse index don't benefit from it (e.g. every document where the 'cms_version' field is NULL). In our case, these queries probably won't matter all that much to the end user.
- Using a caching solution for writes. While the storage back-end would still need to manage the data structures, updating the data structures would not keep the actual crawling processes waiting.
- Using an external search service. This would allow us to only update the search index every few hours. Inserts into the actual data store will not have to update several data structures on every write. The downside is that new results won't be searchable instantly and that data would be stored in two different locations.

3.4.3 Map/Reduce

Map reduce is a programming model that allows the processing of large data sets by distributing the workload onto different distributed clients. This usually goes hand in hand with distributed storage of the data. Some solutions like CouchDB offer a map-reduce interface, but primarily use it to generate "views" of the data. Using incremental updates, these views can be kept current. The problem with incremental views is that the possible queries have to be known in advance and are limited by the generated view.

The more common use-case of map-reduce are distributed ad-hoc queries over several servers. While using map-reduce over several machines would scale almost linearly, it is also a batch oriented system and not designed for real-time searches. The amount of servers needed to filter a subset of more than 100 million documents (in our case: 1 document = 1 domain) and return a result in an interactive search context (<10 seconds) make it not a feasible option. While real-time search is not a proper fit, it could still be used to distribute load to worker processes. The downside of this would be the need to package all application logic into map and reduce phases.

3.4.4 Search and the dynamo model

An interesting topic is the combination of Amazon Dynamo based systems, such as Riak or Cassandra, and the ability to search data stored inside of them. The general idea is to create an inverted index (as described in section 3.4.2) of the existing documents and somehow store it inside of the data-store itself. Since dynamo based systems usually partition data using a consistent hashing algorithm, the only thing left to do is to split the index up according to certain criteria and store the broken down pieces. There are basically two different approaches to splitting up this index:

1. Document based partitioning
2. Term based partitioning

A good additional source of information about this topic is a presentation[29] that Basho employee Rusty Klophaus gave at the Berlin Buzzwords

[29]`http://berlinbuzzwords.blip.tv/file/3813596/`

conference in 2010. In this presentation, he explains the design behind "Riak Search", the soon-to-be-released addition of Lucene on top of the dynamo based Riak data-store.

3.4.4.1 Document based partitioning

The idea behind this is to store an inverted index for every document in the partition where the document itself is stored. This means that a query for a certain term has to hit every partition to check weather or not any documents contain that term. One advantage in a multi-server environment is that a single query tends to have a lower latency because it can hit all available nodes in parallel. After the slowest node has answered, you have got all possible results. It is also nice in terms of index distribution, since it disperses the index the same way the documents are dispersed over the system, evening out the storage requirements. The problem is that the total query-throughput of the system is smaller because every single query will be passed on to all of the available nodes. Running more than one query in parallel means that query 2 has to basically wait for query 1 to finish. This way of partitioning the inverted index also leads to a lot of seeks on the disks of the nodes, and to a lot of network connections.

3.4.4.2 Term based partitioning

The idea behind term based partitioning is to split the document into its terms, and store the inverted index according to its terms (e.g. single words in a text). This means that the terms of a single document will be scattered around the whole system. If two documents have the same term, both of them will be referenced by the partition holding this term. This means that finding all documents containing a certain term is relatively easy. The system just has to use its consistent hashing function on this term and will be able to immediately find the inverted index pointing to all of the documents containing this term. The downside is that putting a new document into the inverted index requires writes to a lot of partitions, compared to the document based partitioning approach. Another downside is that the query latency tends to be higher because there is only one single node answering the query. The upside of this is that parallel queries will probably be split between several nodes of the system and not have to hit all of them at once. One of the biggest problems with this approach is the "hot spot" problem. Since this approach will always map the same term to the same partition, a popular term will always end up on the same partition. This means that all searches for this term will end up on that single node.

3.5 Evaluation

This section of the paper should give an overview of the possible candidates for back-end storage of the collected data. The main focus in the selection of the participants was on open-source projects that have an active community and that still show frequent progress in development.

3.5.1 MongoDB

3.5.1.1 Features

One of the most promising data storage back-ends these days is MongoDB. It is a document store that has an extremely rich data-model that includes data structures such as arrays, associative arrays and supports atomic operations[30] like integer increments etc. Internally, MongoDB saves data in the BSON format. BSON is a binary encoded serialization of JSON-like documents with an open specification available at bsonspec.org.

MongoDB uses B^+-Trees to be able to allow ad-hoc queries that are comparable in features to an SQL SELECT. Atomic modifiers allow things like incrementing integers on the server side or pushing data into arrays. There is also the possibility of running Javascript code on the server and

[30] `http://www.mongodb.org/display/DOCS/Atomic+Operations`

MongoDB even provides an interface for Javascript-based map/reduce tasks. MongoDB also has a feature called "GridFS" that allows applications to store files of any size inside the database without adding extra complexity to the user's source code. Internally, the usual data structures are used to save the chunks of data. Accessing and modifying data happens using a pretty intuitive syntax. The developer basically constructs a document with the things that he is looking for. As an example, a search for a student named John that is enrolled would result in a query like this (Javascript syntax):

```
db.my_collection.find({"name":"John"}, {"enrolled":true})
```

It also has advanced query operators like greater than or less than:

```
db.my_collection.find({ "name":"John", "age" : { $gt: 25} } )
```

It even supports search using regular expressions, e.g. a case-insensitive search for John or Jon:

```
db.my_collection.find( { "name" : /^joh?n/i } );
```

Updates work basically the same way, only that you have to also supply a second document with the update instructions. Atomically incrementing the semester of everybody named John would look like this

```
db.my_collection.update( { "name":"John" }, { $inc: { "semester" : 1 } } );
```

A simple update of a field would look like this:

```
db.my_collection.update( {"student_id":62123 }, { "enrolled": false } );
```

MongoDB offers an interactive shell that allows one to do simple searches like this on the fly. There are also tools like "MongoHub"[31] that allow easy introspection and visual navigation through the stored data.

MongoDB officially provides drivers for C, C++, Java, Javascript, Perl, PHP, Python, and Ruby. There is also a wealth of community supported

[31] http://mongohub.todayclose.com/

drivers for nearly any programming language available. Links to them can also be found in the official MongoDB wiki[32]. They also feature an alpha quality REST interface[33].

3.5.1.2 Scalability

The single node performance of MongoDB is more than impressive. Even without trying to horizontally scale MongoDB, it offers better performance than all of the other comparable systems (RDBMS / Document stores). It has to be noted that MongoDB, by default, has data durability constraints that are a little bit more relaxed than some of the other database solutions. This, however, is not a real problem for the data in our project. Inserts and updates in MongoDB also offer the possibility of setting a "safe" and "fsync" option that will make the call return either when the data has been written to the server or when the server has fsynced the data to disk. The official MongoDB blog contains a good article[34] about MongoDB's approach to single-node durability.

Concerning the need for vertical scalability, the official MongoDB website collects independent third party benchmarks in their wiki[35]. Horizontal scaling with replication and sharding is in heavy development and looks promising. The downside is that it requires different categories of machines, and does not follow a uniform approach as such as the dynamo based Cassandra, Riak or Project Voldemort. Servers have to specifically be configured as a "Master", marked as routing instances, and also manually removed from the cluster, which drives up complexity in operation of the system.

[32]`http://www.mongodb.org/display/DOCS/Drivers`
[33]`http://www.mongodb.org/display/DOCS/Http+Interface`
[34]`http://blog.mongodb.org/post/381927266/what-about-durability`
[35]`http://www.mongodb.org/display/DOCS/Benchmarks`

3.5.1.3 Limitations

MongoDB uses memory-mapped I/O, this limits its ability to store data bigger than 2.5 GB on 32 bit systems. A big problem for our project is the fact that MongoDB currently uses global locking when manipulating data. This means that a long running write will block all other reads and writes. MongoDB is developing rapidly and adding "yield" mechanisms to allow long-running operations to give over control to other waiting operations. There is however the problem that the locking occurs on process level and writes to database A can block reads on database B.

3.5.1.4 Search

As previously mentioned, MongoDB offers its own internal query mechanism. To be able to deal with millions of documents, however, the need for putting indexes on a lot of fields would slow down inserts/updates and also require a lot of RAM to be able to update the matching parts of the B^+-Tree of every index. Another downside is the fact, that MongoDB does not support sparse indexes at the moment. Especially for fields that don't show up in every document, including a NULL value in the index drives up memory usage in the generated B^+-Trees and also uses up disk space. Faceting is possible by using a map/reduce task, this however does not perform particularly well with a large number of documents and results in slowdowns. During the last few releases, the yielding of locks during the map-reduce phase was introduced which helped a bit with the problem of slowdowns.

Projects also exist that try to integrate MongoDB into external search solutions like Solr. Photovoltaic[36], for example, uses MongoDBs replication internals to pipe changes into a Solr Index as they happen. This is not as convenient as Solr's internal update mechanism that supports some data-stores such as MySQL as a data source, but it is at least a possibility.

[36] `http://github.com/mikejs/photovoltaic`

3.5.1.5 Operations

Backups can be easily achieved by calling the "fsync" command on the database, setting a write lock and then creating a file system snapshot (using XFS and EBS). The MongoDB homepage offers a good overview of the different backup possibilities [37]. The sharding is still more complicated than dynamo based solutions (namely Riak or Cassandra), but development is moving fast and the active mailing lists and IRC channel are a big plus compared to other solutions with less active communities (e.g. tokyo cabinet).

3.5.1.6 Conclusion for the Project

MongoDB would be a great fit in terms of data storage. Especially the atomic operations such as increments help to keep the code base small and clean. A downside is the need for 64 bit instances in connection with EC2. This drives up costs when compared to other solutions because the minimum configuration would be an EC2 large instance.

Horizontal scaling would only be necessary if we actually wanted to use the database for search operations and create indexes that would take up more RAM than a single machine could provide in a cost effective manner. The effect of the global locking would have to be re-evaluated over time. Almost every new major version of the database over the last months eased the locks by introducing yielding operations and fine tuning the databases behaviour.

Search for specific subsets of the data would require a third party tool like Solr This, however, might change if sparse indexes are introduced to MongoDB. In that case, all of the Drupal specific fields could be indexed without including millions of documents that do not have the field. Another solution to this problem might be splitting the data into several databases according to the detected CMS. This would require manual

[37] `http://www.mongodb.org/display/DOCS/Backups`

management of data which would increase code complexity. The current locking policy (process wide, not database wide) also complicates this solution.

3.5.2 CouchDB

3.5.2.1 Features

Apache CouchDB is one of the older "NoSQL" solutions. It features a RESTful JSON API that makes it easy to access using HTTP. Another interesting feature is the ability to use Javascript or Erlang based incremental MapReduce operations to create something similar to "views" in RBDMS. There are also different third party implementations of "view servers"[38] that allow the user to write map-reduce functions in other languages (Python[39], Ruby[40] or Java[41]). Using the third party view servers requires some additional setup. Since views can be generated incrementally on every insert, update, or delete, they have a pretty constant (low) overhead during regular operations, yet still allow complex transformations and filtering of the stored data.

CouchDB's internal multi-version concurrency control design automatically provides snapshot capabilities but also adds a disk space penalty that makes the database require compaction from time to time. CouchDB uses B-trees internally, but adds some slight modifications. This is described in the official CouchDB book[42]:

> CouchDB's B-tree implementation is a bit different from the original. While it maintains all of the important properties, it adds Multi Version Concurrency Control (MVCC) and an append-only design. B-trees are used to store the main database

[38]`http://wiki.apache.org/couchdb/View_server`
[39]`http://code.google.com/p/couchdb-python/`
[40]`http://github.com/candlerb/couchdb_ruby_view`
[41]`http://github.com/cloudant/couchjava`
[42]`http://books.couchdb.org/relax/appendix/btrees`

> file as well as view indexes. One database is one B-tree and one view index is one B-tree. MVCC allows concurrent reads and writes without using a locking system. Writes are serialized, allowing only one write operation at any point in time, for any single database. Write operations do not block reads and there can be any number of read operations at any time. Each read operation is guaranteed a consistent view of the database, How this is accomplished, is at the core of CouchDB's storage model. The short answer is that because CouchDB uses append-only files, the B-tree root node must be rewritten every time the file is updated. However, old portions of the file will never change, so every old B-tree root, should you happen to have a pointer to it, will also point to a consistent snapshot of the database.

Another interesting feature of the CouchDB project is "futon". It is an included web interface that helps the user to administer CouchDB. It allows the user to view and manipulate stored data. It can also be used to administer most aspects of CouchDB, such as compaction, creation of views, or setting up replication.

3.5.2.2 Scalability

Thanks to the previously mentioned MVCC append-only approach and Erlang as a programming language, CouchDB offers one of the best concurrency behaviours in the field allowing good vertical scalability. Horizontal scalability in CouchDB basically consists of incremental replication with bi-directional conflict detection and resolution. Setting up replication is easily available over CouchDB's REST interface. A simple HTTP POST request to "/_replicate" with this body is all that is necessary:

```
"source":"$source_database","target":"$target_database"
```

Sharding is not supported on the server side. There are, however, third party frameworks that provide clustering/partitioning for CouchDB (an example would be couchdb-lounge[43]).

3.5.2.3 Limitations

CouchDB has no limitations for database sizes on 32 bit systems such as MongoDB. However, it does have other problems when it comes to large amounts of data. In my tests, running CouchDB on an small EC2 instance resulted in problems when it came to compacting its append-only storage format. Under production load, the compaction process could not follow up with updates to stored data and was not able not finish compaction in a reasonable amount of time (a day). This has also been described as a generic problem of some append only storage back-ends in section 3.3.3.

3.5.2.4 Search

Thanks to its copy on write nature, CouchDB offers an interesting approach to indexes. CouchDB supports the creation of "views" according to a custom search query. These views will be saved as a collection and are updated incrementally on every access. It is also possible to update them at fixed intervals or after every document update. The views can be created in Javascript or Erlang. An example of a Javascript based view from the CouchDB wiki[44] would be this:

```
Map:
function(doc) {
  emit(null, doc);
}


Reduce:
function(doc) {
  if (doc.Type == "customer") {
    emit(doc.LastName, {FirstName: doc.FirstName, Address: doc.Address});
```

[43]http://tilgovi.github.com/couchdb-lounge/

[44]http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views

```
    emit(doc.FirstName, {LastName: doc.LastName, Address: doc.Address});
  }
}
```

This would create a view that includes all customers from the database with their full names and addresses. The interesting thing is that this would allow one to look up customers by first OR last name because both of them were emitted as a key. The resulting view would look like this:

```
{
   "total_rows":4,
   "offset":0,
   "rows":
   [
     {
       "id":"64ACF01B05F53ACFEC48C062A5D01D89",
       "key":"Katz",
       "value":{"FirstName":"Damien", "Address":"2407 Sawyer drive, Charlotte NC"}
     },
     {
       "id":"64ACF01B05F53ACFEC48C062A5D01D89",
       "key":"Damien",
       "value":{"LastName":"Katz", "Address":"2407 Sawyer drive, Charlotte NC"}
     },
     {
       "id":"5D01D8964ACF01B05F53ACFEC48C062A",
       "key":"Kerr",
       "value":{"FirstName":"Wayne", "Address":"123 Fake st., such and such"}
     },
     {
       "id":"5D01D8964ACF01B05F53ACFEC48C062A",
       "key":"Wayne",
       "value":{"LastName":"Kerr", "Address":"123 Fake st., such and such"}
     },
    ]
}
```

While this would allow us to create certain fixed views and even to keep faceted searches up-to-date easily, ad hoc queries are still impossible unless all of them are known in advance and have been prepared as a view. Another (third party) option of searching within CouchDB is the "CouchDB-lucene"[45] project. CouchDB-lucene integrates the lucene search service on top of CouchDB. This would allow ad-hoc searches using the lucene syntax.

3.5.2.5 Operations

Since CouchDB uses an append only storage back-end, just creating snapshots (even without an fsync command) will still result in a valid data structure. This is one of the advantages of CouchDB when it comes to the operation of the database. Notes on this can be found over at the CouchDB WIki[46].

3.5.2.6 Conclusion for the Project

While CouchDB is a great choice in terms of parallel access and performance, the amount of updates (e.g. for link counters) that we need to be able to do would require constant compacting of the back-end. This would slow down other operations because of the heavy I/O load and almost double the disk space requirements. If CouchDB ever optimized the back-end compaction, it would be worth a second evaluation, but at the time of writing, it is a bad choice from an operations perspective.

[45]http://github.com/rnewson/couchdb-lucene
[46]http://wiki.apache.org/couchdb/FilesystemBackups

3.5.3 MySQL and PostgreSQL

3.5.3.1 Features

One of the differences between MySQL and PostgreSQL is their design philosophy. While PostgreSQL is a single, unified database server, MySQL is a network front-end to a number of different storage engines. Feature wise, the two open source RDBMS offer a wide range of options. While most of the transactional features are not really important for our project, there are some things that help keep the code clean and enhance performance, such as:

- Ability to fine-tune data types (e.g. using an unsigned integer for link counters since they are by definition not negative)
- Atomic increments (e.g. "UPDATE MyTable SET LinkCount=LinkCount+1 Where domain=example.org")
- On-the-fly compression (if we wanted to save the HTML code of certain sites)
- Ability to fine-tune indexation (e.g. only indexing the first 5 letters of the cms name. After the fifth letter, it is usually unique enough)
- Support for partial indexes that disregard NULL values (only in Postgres, not MySQL)

3.5.3.2 Scalability

Both PostgreSQL and the InnoDB storage back-end in MySQL rely on MVCC and are able to provide non-locking parallel access. When it comes to MySQL, the choice of database back-end (MyISAM, InnoDB, MariaDB, ...) is one of the most important choices when it comes to performance. Especially with InnoDB, the database also requires extensive tuning of cache-sizes to get to a good performance level. A great resource to read up on the different MySQL storage options is the "High Performance MySQL" book published by O'Reilly[21]. PostgreSQL and MySQL performance lags behind the other solutions when it comes to simple key-value access of data.

Both of them support replication and sharding as a form of horizontal scalability either internally or using third party tools, but it is nowhere near the level of ease of use that most of the NoSQL solutions offer. In general, more thought has to go into the creation of a data-schema and fine-tuning parameters than for most other data-stores.

3.5.3.3 Limitations

Besides the comparatively low performance numbers, the solutions don't have any direct limitations concerning operations or general use in our project. Looking at operational constraints of other data-stores, the maturity of the RBDMS projects seems to be beneficial.

3.5.3.4 Search

While both MySQL and Postgres support the creation of indexes (usually B^+-Tree, sometimes hash-based), optimized search solutions such as Solr offer better performance and flexibility. MySQL and Postgres are supported by SOLR's data import handler and can be indexed without the need for another third party tool.

An interesting feature of the internal indexation implementations is the use of adaptive hash indexes in MySQL's InnoDB back-end[47]. This feature automatically generates Hash indexes in addition to the regular B^+-Tree when it would benefit performance. With the amount of data we handle, however, this might not come into action. Another interesting feature is the support for partial Indexes when using PostgreSQL[48]. The main idea behind this is that a column that is seldom used (e.g. the version number that is only collected for Drupal sites) will not be indexed as a "NULL"

[47]`http://dev.mysql.com/doc/refman/5.0/en/innodb-adaptive-hash.html`
[48]`http://www.postgresql.org/docs/8.4/static/indexes-partial.html`

value but would simply be ignored. This way, the size of the B$^+$-Tree is only connected to the amount of rows that include this column, and not to the total amount of columns. The use of partial indexes could make it possible to rely on the internal indexing mechanisms of PostgreSQL to index our collected data.

3.5.3.5 Operations

MySQL and Postgres are mature and stable solutions that offer a great amount of information and tooling. On-line backups can be done in a variety of ways and relying on already existing solutions limits the need for extra documentation. For InnoDB databases, Percona's Xtrabackup[49] provides on-line backup functionality. The included "mysqldump" tool can also do on-line backups of InnoDB tables using the "single-transaction" switch. For PostgreSQL, the included pg_dump tool offers similar functionality. Backups using Amazon EBS and file system snapshots are also possible.

3.5.3.6 Conclusion for the Project

The use of a relational database system allows for seamless integration with external search services such as Solr or Sphinx, while offering a great amount of tooling and available expertise. If it is possible to keep up with the necessary amount of operations per second, long term storage in one of the systems seems to be a good compromise between ease of use from an operational standpoint and performance. The only downside is the overhead of having to use SQL for simple operations and the performance penalty of having to support unneeded ACID constraints.

3.5.4 Tokyo Cabinet

3.5.4.1 Features

Tokyo cabinet is developed and sponsored by Mixi, the Japanese equivalent of Facebook. It is a fast and feature-rich library that offers several

[49] `https://launchpad.net/percona-xtrabackup`

different data-structures:

B^+-Tree: A regular B^+-Tree, allowing several values for a single key. It supports prefix and range matching on a key.

Hash: A hash-based database allowing only a single value per key.

Fixed-length: High performance data-structure containing an array of fixed-length elements.

Table: A column store with support for indexation and queries.

Tokyo cabinet has excellent single-node performance numbers and is leading the field together with MongoDB. It also provides on-the-fly compression of data (bzip, gzip) and does fine-grained record level locking. Tokyo cabinet is only responsible for data-storage; a network interface is provided by "Tokyo Tyrant". Tokyo Tyrant supports asynchrnous I/O with epoll/kqueue and can be used in a highly parallel manner.

An interesting feature of Tokyo Tyrant is the ability to extend it using scripts in the Lua programming language. Ilya Grigorik has covered this extendibility in a great blog post[50]. The use of lua allows the easy creation of server-side atomic commands such as the incrementing of integer numbers. The scripting extensions can be either record-locking or use a global lock.

Bindings for Tokyo-Cabinet and Tokyo-Tyrant are available for almost any major programming language (including Java, Ruby, Erlang, Lua, Python, and Haskell). A subset of Tokyo Tyrant functions is also available via the memchached protocol and HTTP in case a native library is not available in the language of choice.

[50]`http://www.igvita.com/2009/07/13/extending-tokyo-cabinet-db-with-lua/`

3.5.4.2 Scalability

The single node performance of Tokyo Cabinet is one of its biggest advantages. It delivers comparable numbers to MongoDB and usually is faster than other data-stores. For large amounts of data, tuning of the b$^+$-Tree parameters is necessary. Horizontal scalability would need to be part of Tokyo Tyrant. Tokyo Tyrant is the network interface to Tokyo Cabinet. It offers Master-Master and Master-Slave replication. Automatic sharding is not part of Tokyo-Tyrant itself, however there are third party libraries like LightCloud[51] that use Tokyo Tyrant as a back-end. Access to LightCloud from Ruby is still in development[52] and adds further dependencies to the project.

3.5.4.3 Limitations

Sadly, the community surrounding Tokyo Cabinet is not as vibrant as the one around projects like MongoDB or CouchDB. A lot of the engineering blog posts are in Japanese and the only central point for the english-speaking community is a mailing list[53] and a wiki[54].

While there are reports of corrupted data-files from time to time, I haven't encountered any problems when evaluating the database with real world data. In case of a problem, Toru Maesaka highlights low-level recovery tools in his blog post "How to recover a Tokyo cabinet database file"[55]. Tokyo Cabinets successor Kyoto Cabinet has even less available information. For both of them, tuning of startup parameters is necessary when trying to access more than a few million documents without heavy speed degradation. The only real information about important parameters and their meaning is currently available at James Edward

[51]`http://opensource.plurk.com/LightCloud/`
[52]`http://github.com/mitchellh/lightcloud`
[53]`http://groups.google.com/group/tokyocabinet-users`
[54]`http://tokyocabinetwiki.pbworks.com/`
[55]`http://torum.net/2010/01/how-to-recover-a-tokyo-cabinet-database-file/`

Gray II's blog in his series on Tokyo Cabinet[56].

3.5.4.4 Search

Tokyo Cabinet offers support for indexation using the "Table" database. The performance while testing it with 60 million documents isn't usable in production, even when indexing the columns. A second option would be the use of "Tokyo Dystopia", a full-text search system that integrates within Tokyo Cabinet and should provide better performance. Sadly, the missing community behind the solution is one of the main reasons for not evaluating the Tokyo Dystopia in detail.

3.5.4.5 Operations

Backup and replication are included with Tokyo Tyrant and the "tcrmgr" tool that is shipped with it, but are not an optimal solution. Using "tcrmgr copy", a backup copy of the database is being created. Sadly, this blocks incoming connections to the server. Another possibility would be the booting of a process that acts as a read-slave for the backup procedure.

3.5.4.6 Conclusion for the Project

While the Tokyo-family of products offers interesting functionality and great performance, the missing community and resources for operating it make it a sub-par choice for the project.

3.5.5 Riak

3.5.5.1 Features

Riak is an open-source Erlang-based implementation of Amazon's Dynamo[23] created by a company called "Basho". One of the differences from some of the other open-source Dynamo implementations (e.g. Cassandra) is that Riak actually tries to stay close to the original paper and uses a

[56]http://blog.grayproductions.net/articles/tokyo_cabinets_keyvalue_database_types

vector clock implementation for conflict resolution. Details about Vector clocks can be found on the official Basho Blog in the excellent posts "Why Vector Clocks are Easy"[57] and "Why Vector Clocks are Hard"[58].

In addition to the Dynamo paper, Riak offers Javascript-based Map/Reduce to filter out certain subsets of data or do simple calculations (such as maximum, minimum or average). They also provide built in map-reduce helpers in Erlang[59] that assist in the creation of high performance versions of these tasks. Riak also has the ability to use HTTP Link Headers (as defined in RFC 2068 [14], section 19.6.2.4) to create connections between different records and allow something called "Link walking" which helps to annotate data with links, and thus enabling more than simple key-value queries. "Link walking" has been described in the "Link walking by example" post[60] on the basho blog. Interaction with Riak can be done with a REST API or Google's Protocol Buffers (which have less of an overhead). Ruby libraries exist for both protocols. Another interesting feature is Riak's support for pluggable back-end storage systems. While it originally used InnoDB, the developers switched over to their own system called "Bitcask". There are also experimental back-ends such as a Redis backend[61] for Riak.

3.5.5.2 Scalability

Concerning horizontal scalability, Riak supports elastically adding and removing machines from a server-cluster while automatically balancing the load on each machine. When adding a new node to a running cluster, it automatically starts taking an equal share of the existing data from the other machines in the cluster, as well as an equal share of all new requests and data. The mechanism behind this is the automatic division of the cluster's data space into partitions (64 by default). Upon adding

[57]http://blog.basho.com/2010/01/29/why-vector-clocks-are-easy/
[58]http://blog.basho.com/2010/04/05/why-vector-clocks-are-hard/
[59]http://hg.basho.com/riak_kv/src/tip/priv/mapred_builtins.js
[60]http://blog.basho.com/2010/02/24/link-walking-by-example/
[61]http://github.com/cstar/riak_redis_backend

a new node to the cluster, it claims an equal share of the partitions, meaning that every other node has a fewer partitions to worry about. When removing a node, the reverse happens. The removed node hands partitions back to the remaining nodes in the cluster.

Vertical scalability is enhanced by Riak's back-end storage engine called "Bitcask". The design principles are described in the paper "Bitcask - A Log-Structured Hash Table for Fast Key/Value Data"[15] by Justin Sheehy and David Smith. Jeff Darcy has run some benchmarks and presented the results in his "Bitcask Rocks" blogpost [62]. The current memory usage is 32 bytes per key (20 bytes hash + 4 bytes file id + 8 bytes offset). This would allow us to store 100 million keys in less than 3 GB of RAM. According to the riak IRC channel, this number will be brought down over time using optimizations such as Burst tries[63]. More details about the possible optimizations can be found in a chat transcript from the official IRC channel[64].

3.5.5.3 Limitations

When using map-reduce operations to filter data, the map operation requires a list of keys that should be used with the map operation. Since our goal is to select a subset of all of our data, we would have to pass all available keys (-> domain names). Riak allows the passing of a whole collection of keys (a so called "bucket"), but the manual has this to say about the process:

> You may also pass just the name of a bucket ("inputs":"mybucket",...), which is equivalent to passing all of the keys in that bucket as inputs (i.e. "a map/reduce across the whole bucket"). You should be aware that this triggers the somewhat expensive "list keys" operation, so you should use it sparingly.

[62]`http://pl.atyp.us/wordpress/?p=2868`
[63]`http://portal.acm.org/citation.cfm?id=506312`
[64]`http://gist.github.com/438065`

Since we are dealing with several million keys, using the "list keys" operation is sadly not something that we can consider in a real-time search environment. While finishing the thesis, Justin Sheehy put up a post to the Riak mailing list titled "list_keys is less bad"[65] in which he describes some optimizations that promise improved performance of up to 10 times. Evaluating the performance of the map/reduce queries might be something to consider once the changes have been merged into a stable release.

3.5.5.4 Search

Since map-reduce operations are currently not feasible, and Riak does not offer any other options so far, an interesting approach could be the use of an external search service (Solr or Sphinx) together with Riak's "Post-Commit hooks"[66]. These hooks allow the execution of Erlang code after a successful write (update or deletes are also considered writes). This way, the data could be posted to the web-service part of the search service and real-time searches could become possible. Basho is working on their own implementation of this called "Riak Search"[67]. At the time of writing, however, this feature is not available to the general public.

3.5.5.5 Operations

Riak is a nicely designed piece of software from an operations perspective. It does not require special nodes; all of them are equal. Updating software versions is as easy as taking down a node, updating it, and then bringing it back online. This can be done one by one until the whole cluster runs the updated version. Centralized backups (and restores) can be done using the Riak command line tools[68]. Since Riak is a distributed system, doing file system snapshots might work if there were only one machine

[65] http://lists.basho.com/pipermail/riak-users_lists.basho.com/2010-August/001811.html

[66] http://wiki.basho.com/display/RIAK/Pre-+and+Post-Commit+Hooks

[67] http://www.basho.com/riaksearch.html

[68] https://wiki.basho.com/display/RIAK/Command-Line+Tools#Command-LineTools-backup

in the cluster and the file system storage back-end (riak_kv_fs_backend) is used. This is defiantly not recommended for production use.

3.5.5.6 Conclusion for the Project

While Riak would be a good solution in terms of pure data storage and scalability, there are some deficits. The missing ability to filter out certain subsets of data easily, the limitations for map-reduce operations, and the missing integration with third party search services make it a unsatisfactory choice for our project.

3.5.6 Cassandra

3.5.6.1 Features

Cassandra is a Java-based open source implementation of the system described in Amazon's dynamo paper[23] and was initially implemented by the Facebook engineering team. However, it does differ in some aspects from the original paper. For example, it does not use vector clocks for conflict resolution (such as Riak). Instead, it uses timestamps. This requires all of the *client machines* (not servers!) to be NTP synced. Cassandra also moved away from the original partition-based consistent hashing over to key ranges for replication and sharding, while adding functionality like order-preserving partitioners and range queries. Cassandra can also function as a back-end for the map-reduce framework Hadoop, and provide data stored in Cassandra to Hadoop jobs.

3.5.6.2 Scalability

Concerning horizontal scalability, dynamo based systems distribute data well over several machines in a cluster. While Cassandra is really similar to Riak in this regard, there are some differences between the two. When adding a machine to a Cassandra cluster, by default Cassandra will take on half the key range of whichever node has the largest amount of data stored. This can be fine-tuned, but in general leads to a way of balancing that is less smooth than Riak's partition based approach. It also results in a problem when lots of machines in a cluster are overloaded by a tiny amount. If all nodes in an N-node cluster were overloaded, you would need to add N/2 new nodes. It has to be said that Cassandra also provides a tool ('nodetool loadbalance') that can be run on each node to rebalance the cluster manually.

3.5.6.3 Limitations

Cassandra's data-model consisting of Keyspaces and Column Families is defined in an XML file. Currently, changing the data-model at this level requires a rolling reboot of the entire cluster. Its competitor Riak allows the creation of new "buckets" with a changed data-model on the fly. The upcoming version 0.7, however, will feature live schema updates, according to the official Cassandra wiki[69]. Something that is common with current dynamo based systems is the missing support for atomic operations (e.g. incrementing Integer numbers). Although there is a story for this in the Cassandra bugtracker[70], it would require the inclusion of the open-source coordination service Apache Zookeeper[71], a close clone of Google's "Chubby" system[72].

3.5.6.4 Search

An interesting project called "lucandra"[73] allows one to store the data structures used by the Lucene/Solr search service inside Cassandra itself. While this does not help with the actual indexation of content, it does help to distribute the disk requirements of the data structures, while also allowing more than one search service accessing the same data. This could help upgrading a cluster node without losing the ability to search the data. The storage of a reverse index was actually the original use-case

[69]http://wiki.apache.org/cassandra/LiveSchemaUpdates
[70]https://issues.apache.org/jira/browse/CASSANDRA-721
[71]http://hadoop.apache.org/zookeeper/
[72]http://labs.google.com/papers/chubby.html
[73]http://github.com/tjake/Lucandra

of Cassandra at Facebook. The indexation of content would still have to be the task of custom code that pushes changed records to Solr.

3.5.6.5 Operations

Cassandra is a fairly low maintenance tool when it comes to operations because of its auto-balancing features. Backups can be done in two ways according to the operations part of the Cassandra wiki[74]. The first way of doing backups is to either use the "nodetool snapshot" command or to create file systems snapshots of the single nodes themselves. The wiki mentions this about the implications of snapshotting:

> You can get an eventually consistent backup by flushing all nodes and snapshotting; no individual node's backup is guaranteed to be consistent but if you restore from that snapshot then clients will get eventually consistent behaviour as usual.

Another option is using the "sstable2json" and "json2sstable" commands that will take a node's local Cassandra datafile as an input argument and export the saved data in JSON format. This also has to be done on every node in the cluster. Currently there is no "central" backup solution comparable to Riak's "riak-admin backup" command. Concerning monitoring, it has to be noted that Cassandra exposes internal metrics as JMX data, the common standard in the JVM world.

3.5.6.6 Conclusion for the Project

Cassandra is an interesting project in that it allows the easy distribution of back-end data over several nodes. The downside is that it does not offer any direct search possibilities. The only advantage to Riak that can be currently found for our project is the already existing third party integration for storing Lucene/Solr data and the possible interaction with Hadoop (although Riak offers limited map/reduce too).

[74] `http://wiki.apache.org/cassandra/Operations`

3.5.7 Miscellaneous

There are a few other interesting storage back-end systems that, at the time of writing, look promising but either do not offer any advantages over already discussed solutions, or need some more time to build up a community before using them in production. This section will give a short overview of other projects that were considered.

3.5.7.1 Terrastore

Terrastore[75] is a Java-based document store built on top of the Terracotta Plattform[76]. It automatically partitions data over several cluster nodes and has a HTTP REST interface. It supports atomic server-side operations such as integer increments, or even the execution of Javascript functions. Documents uploaded to the storage engine have to be encoded as JSON data.

An interesting feature is the integration of Elastic Search into Terrastore which would allow fulltext-queries within the stored data. From an operations perspective, Terrastore distinguishes between "master" and "server" nodes. While master nodes provide cluster management and storage services, server nodes provide actual access to Terrastore data and operations. This means that the setup involves more work than with systems like Cassandra or Riak which have "equal" nodes.

Terrastore has a nice technical basis, but is still a very new project and mainly developed by a single lead-developer (Sergio Bossa). Terrastore itself is also a pretty Java-centric project and while the HTTP interface is usable from Ruby, a dedicated client library would be a nice addition. In my limited performance measurements, the initial single-node performance with smaller data samples (< 1 million) was comparable to other databases, but when testing samples growing into the tens of millions of documents,

[75]`http://code.google.com/p/terrastore/`
[76]`http://www.terracotta.org/`

there was a steep drop in insertion performance. This is understandable since neither Terrastore nor the included ElasticSearch have been optimized for single node operation. In general, the project shows great potential and should be re-evaluated after another few months.

3.5.7.2 Amazon SimpleDB

SimpleDB is a commercial web service provided by Amazon that allows storage and retrieval of schemaless rows of data. While SimpleDB would be an interesting fit for our data, there are some limitations[77]. There is a maximum response size for "SELECT" calls (1 MB, 2500 items) and a maximum of 10 GBs of data per domain that would require additional code.

Another limitation detailed in the "Cloud Tips: How to Efficiently Forklift 1 Billion Rows into SimpleDB" Blogpost[78] by practicalcloudcomputing.com is a throttling that occurs at about 70 singleton PUT-operations per SimpleDB domain per second. This would require batching access to simpleDB using batched PUTs[79] which, in turn, would require additional code.

While the additional costs of using SimpleDB might even out with the ability to use smaller EC2 instances when offloading data-storage to SimpleDB, the extra code required to work arround the SimpleDB limitations does not seem to be worth the effort for a one person project.

3.5.7.3 Project Voldemort

Project Voldemort is another Java-based dynamo clone that offers a simple key-value interface to data. While it is a perfectly fine solution,

[77]http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/index.html?SDBLimits.html

[78]http://practicalcloudcomputing.com/post/284222088/forklift-1b-records

[79]http://aws.amazon.com/about-aws/whats-new/2009/03/24/write-your-simpledb-data-faster-with-batch-put/

the community behind Riak and Cassandra is more active these days and provides better integration with the Ruby programming language. Riak and Cassandra also offer more features on top of the dynamo model (e.g. map/reduce functionality with Riak or range queries with Cassandra).

3.5.7.4 Neo4j

Neo4J is a graph database that operates on nodes, edges, and relationships instead of the usual rows, columns, or key-value pairs. Treating relationships as "first class citizens" could be an interesting fit for our data schema that mainly consists of a domain and its various properties ("is a Drupal site", "is located in Germany", etc.). Neo4j seems to be mainly suggested as an embedded database rather than a standalone process that is accessible over a network connection. It offers an impressive vertical scalability in that it can handle billions of nodes on a single server configuration. While it is hard to find proper benchmarks on the web, initial tests showed these numbers to be true. Another interesting property is that access speed while traversing the graph does not depend on the size of the graph, but remains nearly constant. Since the vertical scalability is impressive, the lack of "proper" horizontally scaling is not a high priority item. The homepage[80] mentions that it "can be sharded to scale out across multiple machines", but there is no support for automatic server-side sharding. Sharding has to be done manually by the client.

Neo4j also offers Lucene integration[81] to do a fulltext indexation of nodes. Since our main query use-case does not require full text indexation, but could be satisfied by the relationships provided in the graph, the usage of this feature for the project would have to be evaluated further. There is a feature complete ruby library called Neo4j.rb[82] that allows access from ruby to Neo4j. Sadly, Neo4j.rb it is limited to JRuby and can't be used from any of the other Ruby VMs. While there is some work on the way to create

[80] `http://neo4j.org/`
[81] `http://components.neo4j.org/index-util/`
[82] `http://github.com/andreasronge/neo4j`

a REST server for neo4j and details are available on the projects wiki[83] and in the introducing blog post[84], it is currently considered unstable and not a good choice for the project.

In general, Neo4j would be an interesting fit, but the small size of its community and the missing network interface exclude it as primary choice for the project.

3.6 External search

While it is easy to find a current "nosql"-data-store that offers great performance or scalability, it is hard to find one that allows easy ad-hoc search functionality. High performance ad-hoc searches require additional data structures to be able to keep CPU usage and search time down to a minimum. Something good about our use-case is that real-time indexation of incoming data is not a big priority, and updating the index with newly added domains once a day is more than enough. Specialized search services can help with high-performance searches through millions of domains. This chapter should give a short overview over the possible options.

3.6.1 Sphinx

3.6.1.1 Overview

Sphinx is an open-source full-text search server written in C++. While it operates as a standalone server, it can also be integrated into MySQL. "SphinxSE" offers a MySQL storage engine that, despite the name, does not actually store any data itself. The official documentation[85] describes it as follows:

[83] `http://wiki.neo4j.org/content/Getting_Started_REST`

[84] `http://blog.neo4j.org/2010/04/neo4j-rest-server-part1-get-it-going.html`

[85] `http://www.sphinxsearch.com/docs/current.html`

> It is actually a built-in client which allows MySQL server to talk to searchd, run search queries, and obtain search results. All indexing and searching happen outside MySQL. Obvious SphinxSE applications include:
>
> - easier porting of MySQL FTS applications to Sphinx;
> - allowing Sphinx use with programming languages for which native APIs are not available yet;
> - optimizations when additional Sphinx result set processing on MySQL side is required (including JOINs with original document tables or additional MySQL-side filtering).

3.6.1.2 Data input

One of the (minor) downsides of the Sphinx search server is the restriction it poses on source data. This is a direct quote from the official Sphinx documentation:

> There are a few different restrictions imposed on the source data which is going to be indexed by Sphinx, of which the single most important one is:
> ALL DOCUMENT IDS MUST BE UNIQUE UNSIGNED NON-ZERO INTEGER NUMBERS (32-BIT OR 64-BIT, DEPENDING ON BUILD TIME SETTINGS).
> If this requirement is not met, different bad things can happen. For instance, Sphinx can crash with an internal assertion while indexing; or produce strange results when searching due to conflicting IDs. Also, a 1000-pound gorilla might eventually come out of your display and start throwing barrels at you. You've been warned.

While this is not a unsolvable problem, it does require changes to the source data. In our case, an unsigned auto-incrementing integer offers more than enough room, but would add another index to the system. This would increase the RAM usage when dealing with large numbers of documents and require additional CPU cycles on every insert to the database.

3.6.1.3 Search queries

When using Sphinx in connection with MySQL, Sphinx offers an SQL dialect called "SphinxQL" that allows filtering out certain subsets of the collected data. The official manual describes it like this:

> SphinxQL is our SQL dialect that exposes all of the search daemon functionality using a standard SQL syntax with a few Sphinx-specific extensions. Everything available via the SphinxAPI is also available SphinxQL but not vice versa; for instance, writes into RT indexes are only available via SphinxQL.

Especially for simple queries, there is basically no difference between SQL and SphinxQL. A simple example from the manual looks like this:

```
SELECT *, AVG(price) AS avgprice, COUNT(DISTINCT storeid)
FROM products
WHERE MATCH('ipod')
GROUP BY vendorid
```

Another alternative is the use of the query API from Ruby using one of the available libraries. Thinking Sphinx[86] allows the usage of Active Record with Sphinx, and Riddle[87] provides ruby-wrapped raw access to Sphinx. Riddle also offers a nice way of updating data that is currently in the sphinx search index.

3.6.1.4 Conclusion for the project

The performance numbers and features would be a good fit for the project. The need for an extra integer ID in the SQL schema is suboptimal, but still an acceptable modification. A downside of using Sphinx for the project is the missing expertise inside of Acquia compared to Solr.

[86] http://freelancing-god.github.com/ts/en/
[87] http://github.com/freelancing-god/riddle

3.6.2 Solr

3.6.2.1 Overview

Apache Solr is a Java-based search platform built on top of the Apache Lucene library. It provides a REST interface which allows interaction for both querying and application specific tasks such as reindexing.

3.6.2.2 Data input

There are several ways of getting data from external sources into Solr. The one offering the broadest support is probably Solr's ability to accept XML over a restful HTTP connection. In the default configuration, the interface listens on port 8983 and expects XML in the following format:

```
<add>
  <doc>
    <field name="id">example.com</field>
    <field name="cms_name">drupal</field>
    <field name="cms_version">7.0</field>
  </doc>
</add>
```

While this interface can be accessed from basically any language that has an HTTP library, Solr offers a second possibility that is more suited to the crawlers infrastructure: a JDBC import. By specifying the connection to the database and a mapping to the current data-schema, Solr will import the data directly from the database. This results in faster imports. Solr also allows incremental "delta" imports. Solr saves the timestamp of the last import and can pass this value to a custom query. This allows Solr to only add records to the index that changed since the last update. This way, indexation can happen at night using a cronjob and the direct communication between storage back-end and search back-end keeps down the complexity. It does, however, make it harder to switch technologies in case there are unforeseen problems with Solr.

A big advantage is that Solr can map document input to its internal data schema. This allows us to keep our back-end data unchanged and just use SQL and the schema mapping mechanism to determine which parts of the data have to be searchable, and which parts of the data should actually be saved in the index.

3.6.2.3 Search queries

In general, Solr supports multiple query syntaxes through its query parser plugin framework. The default syntax is the Lucene query parser syntax[88] which is a simple text string that can be enhanced with boolean operators, and even allows boosting certain search terms in priority and wildcards, among other features.

The data can be accessed using Solr's HTTP interface. A more convenient way for accessing the data from Ruby is the rsolr-ext[89] library that wraps the communication in ruby-esque datatypes. A sample-query using the rsolr-ext library and employing a variety of features such as faceting and range queries looks like this:

```
require 'rsolr-ext'
 solr = RSolr.connect
 solr_params = {
   :page=>2,
   :per_page=>10,
   :phrases=>{:name=>'This is a phrase'},
   :filters=>['test', {:price=>(1..10)}],
   :phrase_filters=>{:manu=>['Apple']},
   :queries=>'ipod',
   :facets=>{:fields=>['cat', 'blah']},
   :echoParams => 'EXPLICIT'
 }
```

[88] http://lucene.apache.org/java/2_9_1/queryparsersyntax.html
[89] http://github.com/mwmitchell/rsolr-ext

```
  response = rsolr.find solr_params
```

The rsolr gem also provides the ability to add or update data currently in Solr.

3.6.2.4 Conclusion for the project

The huge feature-set of Solr makes it a good choice for the project. Since there is a lot of internal knowledge about Solr available at Acquia, it seems to be the best solution for the problem at this point in time. While the indexation is slower than in sphinx, this is not a problem since we do not target real time search of newly analysed domains.

3.6.3 Elasticsearch

3.6.3.1 Overview

Elastic Search is a fairly new project that started in early 2010. Its main goal is to provide an automatically sharded search solution based on the Lucene project (the same is true for Solr, see section 3.6.2). A pretty accurate differentiation from the Solr project has been given by the author himself when he was asked "How is Elastic Search different from Solr?" in an interview posted on the Sematext blog[90]:

> To be honest, I never used Solr. When I was looking around for current distributed search solutions, I took a brief look at Solr distributed model, and was shocked that this is what people need to deal with in order to build a scalable search solution (that was 7 months ago, so maybe things have changed). While looking at Solr distributed model I also noticed the very problematic "REST" API it exposes. I am a strong believer in having the product talk the domain model, and not the other way around. ElasticSearch is very much a domain driven search engine, and I explain it more here: `http://www.`

[90]`http://blog.sematext.com/2010/05/03/elastic-search-distributed-lucene/`

`elasticsearch.com/blog/2010/02/12/yourdatayoursearch.html`.
You will find this attitude throughout elasticsearch APIs.

3.6.3.2 Data input

Elasticsearch currently offers three different ways for a developer to add data to the index:

1. A REST API
2. A Java API
3. A groovy API

Since our project is neither Java nor Groovy based, the most interesting interface for the project's specific use-case is the REST API. The project offers both interaction with the index (indexation), and the service itself (administration). The usage is refreshingly simple and consists of an API that expects JSON encoded parameters, and can be accessed using either HTTP or the memcached protocol. The returned results are also encoded in JSON. The interaction is simple and straight forward as can be seen by the REST API overview:

API	Descripton
index	Index a typed JSON document into a specific index and make it searchable.
delete	Delete a typed JSON document from a specific index based on its id.
get	Get a typed JSON document from an index based on its id.
search	Execute a search query against one or more indices and get back search hits.
count	Execute a query against one or more indices and get hits count.
create_index	Creates an index with optional settings.
delete_index	Deletes an index.
put_mapping	Register specific mapping definition for a specific type against one or more indices.

3.6.3.3 Search queries

Elastic Search offers a query DSL[91] that is accessible over the REST interface. A search for all sites running on the Drupal CMS would consist of sending this JSON string to the REST interface:

```
{
    "term" : { "cms_name" : "drupal" }
}
```

There are two ruby libraries (rubberband[92] and elasticsearch[93]) that allow easy ruby-like access to the search interface.

3.6.3.4 Conclusion for the project

Elastic search is a very promising project that targets deployment on multiple machines. It would be a nice fit in combination with one of the

[91]http://www.elasticsearch.com/docs/elasticsearch/rest_api/query_dsl/
[92]http://github.com/grantr/rubberband
[93]http://github.com/adrpac/elasticsearch

dynamo based systems (Riak or Cassandra). Since it is not specifically designed for single-node operation and still early in development, more mature projects like Solr seem to be a better solution for our problem.

3.7 Work Queues

While there are a lot of queuing and messaging systems available on the market, the main focus of the project was on systems that allow an easy configuration, and do not add a lot of operations overhead. While the discussed systems do not provide complex clustering or replication, they have more than enough throughput, and most interestingly, are almost configuration free. For bigger projects that do not have a big focus on simplicity and a higher degree of parallelization across a large number of servers, alternative job distribution systems like Gearman[94] or even Hadoop[95] might be worth an evaluation.

3.7.1 Redis

3.7.1.1 Features

Redis is not strictly speaking a work queue; it is a network-accessible data-structure server. Using a Key-Value interface, a developer can work with strings, lists, sets, and ordered sets. Each specific type features a large array of atomic operations. The "list" type, for example, is basically a stack that allows the developer to push to and pop values from it. Since both of the operations can operate in a LIFO or FIFO manner, it allows the usage of a Redis list as a simple queue. Even more complex operations such as "RPOPLPUSH" exist, allowing to "Return and remove (atomically) the last element of the source List stored at srckey and push the same element to the destination List stored at dstkey"[96]. Using an ordered sets type, this simple list can be turned into a persistent queue

[94]`http://gearman.org/`

[95]`http://hadoop.apache.org/`

[96]`http://code.google.com/p/redis/wiki/CommandReference`

with priories.

When it comes to performance, Redis is one of the fastest data-stores available. Even with a small EC2 instance (1,7 GB RAM, 1 EC2 Compute unit, 32 bit), the numbers gathered by the c-based Redis-benchmark tool are impressive:

```
SET: 9363 requests per second
GET: 8940 requests per second
INCR: 8547 requests per second
LPUSH: 9141 requests per second
LPOP: 8713 requests per second
```

It has to be noted, however, that network latency and not using C might slow down individual clients. Redis is also really impressive when it comes to parallel access. Jak Sprats ran a series of benchmarks[97] and showed that the performance degradation with 27,000 parallel accessing clients is minimal when compared to just 10 parallel clients. His results in Figure 3.7 depict the amount of parallel connections on the horizontal axis and the GET requests per second on the vertical axis.

Libraries to interact with Redis exist for most languages (Java, Erlang, Scala, Ruby, Lua, Perl, Python, Actionscript, PHP, Haskel and many more[98]). For the interaction with the asynchronous Eventmachine library in Ruby, em-redis[99] provides library support. A disadvantage of using Redis as a queue is that it does not provide a native "reserve" functionality. Most specialized queueing systems allow clients to "reserve" a job. This results in the job not being available for other clients, but still staying in the queue. This can usually be coupled with a timeout, after which the job will be reservable again by other clients. This helps with retaining the job data in the case of a client crash.

[97]http://groups.google.com/group/redis-db/browse_thread/thread/1fc6fd6c8937dda7

[98]http://code.google.com/p/redis/wiki/SupportedLanguages

[99]http://github.com/madsimian/em-redis

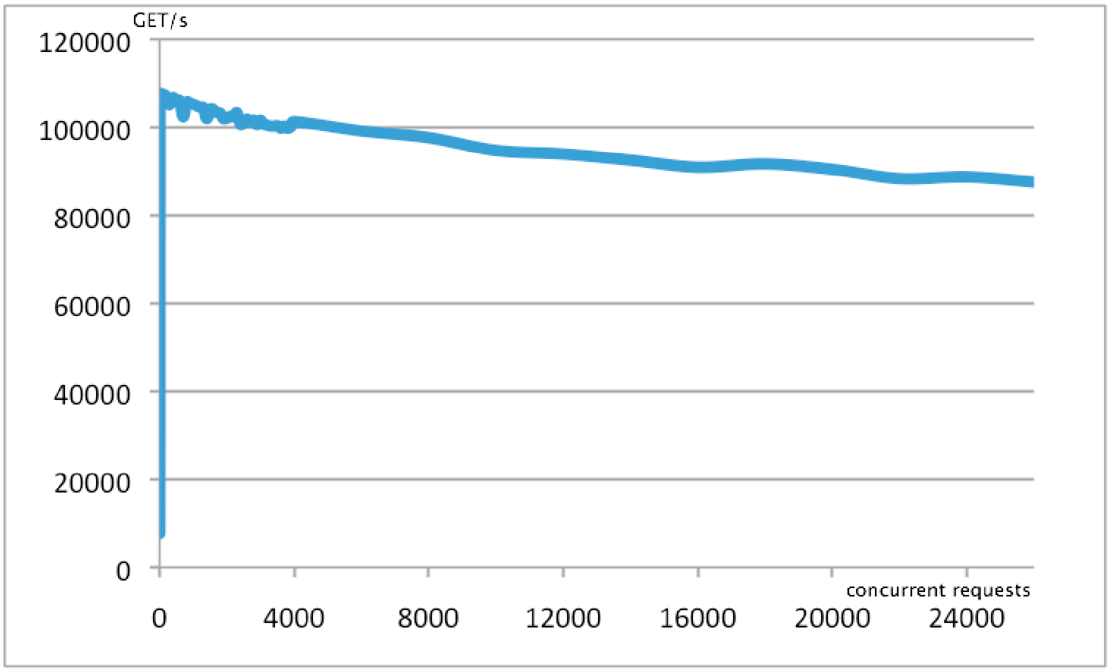

Figure 3.7: Redis parallel access

When using blocking push and pop commands ("BLPOP", "BRPOP"), it is a good idea to open up a second connection to Redis which is only used for writing. Especially when working in a threaded environment, BLPOP on an empty list will lock the connection until somebody pushes a value to the list. The problem when only using a single connection is that you cannot push a value because of BLPOP's lock.

3.7.2 Conclusion for the project

Redis seems a great fit for usage in the project. While it lacks certain optional functionality, if used it in other parts of the system (caching for example) it would help to reduce the amount of infrastructure, while still providing great performance.

3.7.3 Beanstalkd

3.7.3.1 Features

Beanstalkd was, according to the official website[100], originally designed for "reducing the latency of page views in high-volume web applications by running time-consuming tasks asynchronously". Ilya Grigorik describes beanstalk in his blog post "scalable work-queues with beanstalk"[101] like this:

> A single instance of Beanstalk is perfectly capable of handling thousands of jobs a second (or more, depending on your job size) because it is an in-memory, event-driven system. Powered by libevent under the hood, it requires zero setup (launch and forget, ala memcached), optional log based persistence, an easily parsed ASCII protocol, and a rich set of tools for job management that go well beyond a simple FIFO work queue.

Beanstalkd offers internal priorities. By setting a priority variable (default: 65536) when adding a job to the queue, the user can decide how important a certain job is. Higher priorities (higher numbers) will be picked up and processed faster by workers requesting a job. In general, beanstalkd offers several different states for jobs. Jobs will move between the following states:

ready: job can be picked up by worker. When this will happen depends on the job's relative priority

reserved: the job is being worked on at the moment

delayed: A job in this state has been inserted with a delay. When this time period is over, it will automatically switch over to the "ready" state

[100] `http://kr.github.com/beanstalkd/`
[101] `http://www.igvita.com/2010/05/20/scalable-work-queues-with-beanstalk/`

burried: This state is usually for jobs that crashed while being processed. Worker clients can mark the job as burried if the developer wants to check the cause of the crash later in time.

When interacting with beanstalkd from ruby, beanstalkd's ruby client ("beanstalk-client"[102]) still lacks a bit when it comes to a multi-threaded environment as the projects current readme file[103] will tell you:

> If you want to use this library from multiple concurrent threads, you should synchronize access to the connection. This library does no internal synchronization.

While synchronizing access manually does work, it adds lots of extra, unnecessary code to the project. Interaction with the asynchronous Eventmachine library from Ruby can be achieved using em-jack[104]. Other client libraries are available for most major programming languages[105]. Performance wise, beanstalkd is comparable to Redis as it can deal with several thousand job operations per second.

3.7.4 Conclusion for the project

While Beanstalk is a great and simple queueing system, it does not have a lot of advantages over Redis in our specific use-case. The ability to monitor crashed jobs is a nice feature, but it does not weigh heavier than adding another piece of infrastructure to our system.

[102] `http://beanstalk.rubyforge.org/`
[103] `http://github.com/kr/beanstalk-client-ruby/blob/master/README.rdoc`
[104] `http://github.com/dj2/em-jack`
[105] `http://wiki.github.com/kr/beanstalkd/client-libraries`

Chapter 4

Crawler

4.1 System architecture

This section will describe the architecture of the crawler at the end of the project. This architecture is the result of the infrastructure evaluation documented in the other sections of this thesis.

4.1.1 Components

The architecture consists of 4 main components:

1. Redis
2. MySQL
3. Solr
4. Ruby scripts

A graphical overview of the connections between the several components can be seen in figure 4.1. The main components fulfil the following tasks:

Redis is a distributed data structure server who's main task is to function as a queueing system and write through cache. Instead of writing results directly to MySQL, the data is written to a list data structure inside of Redis. This happens using an Eventmachine-based (asynchronous) library. A second Ruby application is continuously pushing the data from the Redis insertion queue to MySQL. Redis also works as a cache for all detected links. Instead of querying MySQL every time we hit a new link, and thus stop the crawling process, we push the links into a Redis list. Another process is continuously getting the links from the queue, checking their existence in the database or a local cache, and adding newly detected domains to the crawl queue.

MySQL is mainly used as a persistent back-end storage. The InnoDB storage engine allows concurrent access to the database. Keeping MySQL on EBS allows a simple backup process using file system snapshots. Since MySQL does not support partial indexes (see 3.5.3.4), indexation of all of the available columns is simply not possible without needing large amounts of RAM. A switch to PostgreSQL might help with this situation, but the time constraints and the missing expertise concerning Postgres sadly did not make a further evaluation possible. Another reason to chose MySQL as a back-end is the seamless interaction with the Solr search service. MySQL can be natively accessed by Solr and incremental data imports work without the need of custom scripting.

Solr allows to search the collected data. In our case, it even stores a copy of the data itself to help separate the storage back-end (MySQL) from the web interface. Its faceting features allow the easy generation of statistical data like the CMS distribution in the Alexa top 1 Million sites. It also supports incremental imports by using a timestamp in the SQL data that updates whenever a row is changed. This way, a simple cron job can launch a daily import of new data. Since it currently saves all of the indexed data, it can function as the only source of information for the web interface.

Ruby scripts function as glue code between all of the systems and contain the application logic. They collect the data (crawling), insert it into the database, and manage the crawling queue. The internal web interface is also powered by a minimalist ruby web framework called Sinatra[1] for now. Sooner or later, this will be probably be transferred to a Drupal powered web page that accesses Solr. The HTTP requests during the crawling-phase are powered by em-http-request[2], an asynchronous HTTP library based on the asynchronous Ruby I/O framework Eventmachine[3].

When it comes to the crawling and link collection process itself, a single step per domain seems to be beneficial compared to splitting the different steps (analytical processing, link extraction, etc.) into several independent processes. Having a completely decoupled system with a lot of workers has a certain architectural charm to it, but passing the data between the several processes results in a big network overhead, while saving all of the data into a temporary storage results in a lot more load on the storage back-end.

In the end, the best solution seemed to keep all of the application logic that does not need to access any additional external network data (e.g. fingerprinting or geo-locating) in one process and just write the results of this step into a fast Redis cache for further insertion into permanent Storage. This also gives us a central place to filter or introspect data running through the system, without the need to modify any application code. It also gives us the advantage of being able to write all of the data to the database in one big write, compared to incrementally filling up a record in several small updates.

[1]http://www.sinatrarb.com
[2]http://github.com/igrigorik/em-http-request
[3]http://rubyeventmachine.com/

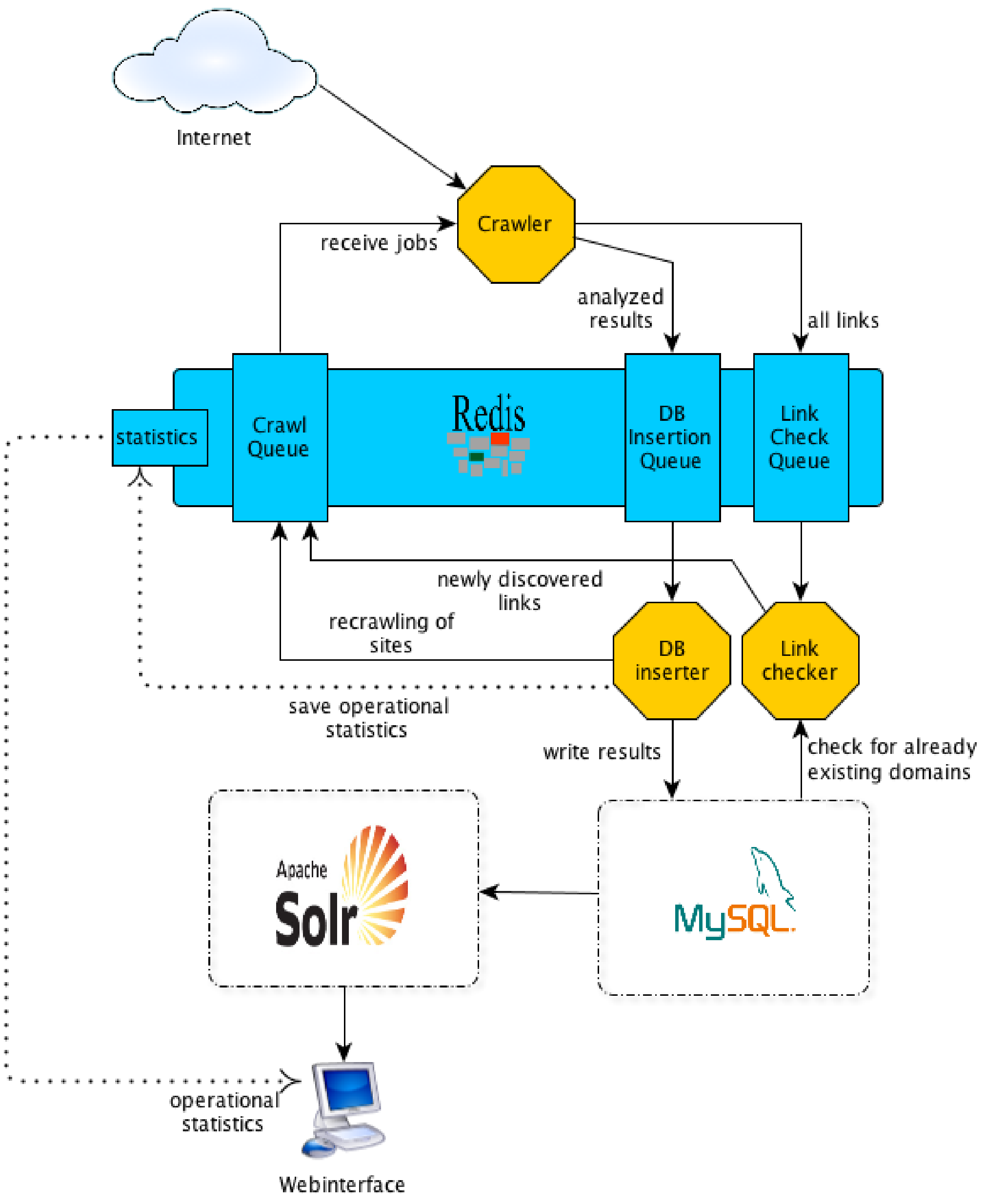

Figure 4.1: Architecture overview

4.1.2 Data schema

The data schema is kept in denormalized form. While adding degrees of normalization to it might help with space usage, keeping it denormalized helps us to easily switch between back-end storages, keeps queries simple, and allows us to retrieve humanly readable data easily. It also means that updates and deletions only operate on a single document in a single location, and do not need to deal with several tables. It also allows us to keep the amount of indexes in RAM to a minimum. For every extra table, we would need another primary key which in turn would need another B^+-Tree with millions of entries.

A noteworthy feature of our schema is the way that IP addresses are handled. They are converted and saved as integer values. Although MySQL offers the inet_aton()[4] function to natively convert IP addresses into a 4 byte integer, using the "IPAddr" class that is part of the Ruby core libraries and the .to_i() method to convert the address into an integer representation seems like a more portable solution without any performance penalties. While arrays are an optional part of the SQL99 standard and implemented by e.g. Oracle[5] and PostgreSQL[6], MySQL does not currently support them. This is why we had to serialize an array of CMS modules (also known as "plugins") into a single string. Using the pipe character ("|") to delimit the modules, gives us an easy way to save and recall them without too much overhead.

[4]http://dev.mysql.com/doc/refman/5.0/en/miscellaneous-functions.html#function_inet-aton

[5]http://download.oracle.com/docs/cd/B10501_01/appdev.920/a96624/05_colls.htm

[6]http://www.postgresql.org/docs/8.0/interactive/arrays.html

key	**type**	**note**	**example data**
domain	varchar(255)	primary key	acquia.com
ts	timestamp	indexed and automatically updated on document change. Used by Solr delta indexation	2010-09-06 11:11:15
domain_suffix	varchar(32)		com
domain_root	varchar(255)		acquia
domain_subdomain	varchar(255)		www
cms_name	varchar(100)		drupal
websrv_name	varchar(100)		apache
ip	int(25)	saving as integer saves space	1264949760
country	varchar(25)	country code	us
incoming_link_count	int(20)	unsigned	180225
response_time	smallint(20)	unsigned	1501
provider	varchar(50)	according to ip range	amazon
cms_version	varchar(15)		n/a
cms_modules	text	"|"-seperated	jstools|wiki|[...]
alexa_rank	int(10)	unsigned	16119
industry	varchar(255)		software & internet
subindustry	varchar(255)		e-commerce and internet businesses
last_crawl	timestamp	to determine necessity for recrawling	2010-08-28 11:22:13

4.2 Problems

4.2.1 HTTP redirects and standards

One of the interesting things to discover while writing a crawler was how most HTTP redirects that can be encountered in the wild do not conform to the standards. The main problem is the values that can usually be found in the "Location" field of the HTTP redirect header.

The W3C RFC 2616[7] is pretty clear about the fact that the location field should have an absolute URI. Here is a direct quote from the matching part of the document:

> 4.30 Location
> The Location response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource. For 201 (Created) responses, the Location is that of the new resource which was created by the request. For 3xx responses, the location SHOULD indicate the server's preferred URI for automatic redirection to the resource. The field value consists of a single absolute URI.
> Location = "Location" ":" absoluteURI
> An example is:
> Location: http://www.w3.org/pub/WWW/People.html

Instead, here is what you tend to find in the wild:

- a file name (`index_new.php`)
- an absolute path (`/new/site.htm`)
- a relative path (`cms/index.php`)
- backslashes, preferably on IIS Webservers (`http:example.org`)
- local paths (`e:/mmbm-web/404b.htm`)

To be sure to properly identify sites even if their redirect is broken, it is beneficial to implement a cleanup method. This method can take the domain and the received redirect string as an input and return a proper URI.

4.2.2 wildcard subdomains and spam

Another thing that one has to pay attention to is the possibility of domains to use a Wildcard DNS record. This allows a server to answer requests for all possible sub-domains. Details can be found in section 4.3.3 of RFC 1034 [8]. One of the big problems with this is keeping down the amount of time that we use on a single domain. This can especially be a problem for big sites like tumblr or deviantART that give each of their members a sub-domain. While it would be possible to crawl all possible sub-domains, crawling the millions of tumblr or deviantART sub-domains does not lead to any new information, but only takes up a lot of time. We also have to pay attention to the frequency with which we crawl the domains. Hammering on the sub-domains of a single provider will usually trigger some rate control mechanism that either blocks our IP or results in things like the server answering with a HTTP 503 status code ("Service Unavailable").

Another big problem are domains that use a wildcard cname entry and generate hundreds of links. These links can be found as spam in forums, blog comments, or just on a simple HTML page. These are probably used in order to fake importance in search engines. A solution to this is to keep a whitelist of allowed subdomains ("www.", "blog.", "shop.", "web.", "cms.", etc.) and a whitelist of top level domains that usually do not serve up wildcard subdomains (e.g. ".edu", ".gov", ".museum", ".mil", ".int"). One interesting thing: while .pro domains are limited to a certain group of people[7], whitelisting this toplevel domain would also whitelist services like dyndns.pro. The problem is that dyndns.pro is

[7] `http://www.registry.pro/pro/eligibility.shtml`

a service that allows people to register arbitrary custom sub-domains. During the operation of the crawler, we encountered a network of spam sites utilizing thousands of dyndns.pro sub-domains.

4.2.3 www cname records

Another big problem is that a lot of domains do not actually serve their root domain, but only their "www." sub-domain. This means that when getting a link to example.com, doing a HTTP GET request to that site could time out or result in an error message. What a lot of web browsers will do in this case is simply check if the "www.example.com" sub-domain answers the request successfully. This means that in case a domain does not answer, the crawler should also check the matching www. sub-domain. This is especially expensive in the case of a synchronous DNS request in combination with a DNS server that takes long to answer. In this case, a thread will be blocked for twice as long as the timeout value just as it waits for a DNS answer. An alternative would be to pre-resolve the domains in the queue using an asynchronous DNS library and add the IP information to the queue before the domain actually reaches the http library. The other alternative is using asynchronous I/O for both the HTTP and DNS request. While this adds a bit of complexity to the project, the performance gains can be substantial.

4.2.4 top level domain detection

Another interesting quirk when dealing with the Internet is the fact that there is no algorithmic way of determining which part of a URL represents the toplevel domain. For a lot of domains it is easily recognizable. For instance the usual country toplevel domains:

- example.de
- example.fr
- example.us

Some of the more interesting cases are domains like these, where everything behind the "example." is actually a valid domain suffix:

- example.research.aero
- example.gv.at
- example.kiev.ua
- example.edu.vn
- example.pvt.k12.wy.us

Detecting the public suffix of a domain is necessary to be able to block generated subdomains (some-generated-domain.spam.com) while still allowing valid ones (university.edu.vn). Sadly, there is no algorithmic approach to detecting which part of an URL is an actual top level domain and which part is a sub-domain. Luckily, the Mozilla Foundation has a project called "The public suffix list"[8] that tries to categorize all public suffixes that are available on the web. Ruby libraries such as Domainatrix[9] or "Public suffix service"[10] are able to parse the list and analyse domains according to the rules of the list.

4.2.5 Balancing parallelism

Another interesting performance problem is the balancing of either crawling-threads or asynchronous i/o requests over time. In theory, we could just experiment with a different number of threads and find the one that utilizes our server's resources to a maximum without overloading it. The major problem with this approach is the differences in response times between different groups of domains. If we happen to currently crawl several sites in China, the I/O wait times will be much larger compared to sites hosted inside of the United States. Larger waiting times also means less CPU utilization. If we are running with a fixed

[8]http://publicsuffix.org/
[9]http://github.com/pauldix/domainatrix
[10]http://www.simonecarletti.com/code/public_suffix_service

amount of parallel threads, the server will either be underutilized in the case of the Chinese sites or overutilized in the case of the websites hosted in the United States. A seemingly easy solution would be to add additional processes once we drop under a certain throughput per second. The problem is that other processes running on the server (e.g. Solr indexation) could require a share of the CPU, and thus limit the throughput at this moment. Adding additional threads would kill server performance in this case.

Another point to pay attention to is the load that increasing the number of file descriptors and context switches adds to the system. This keeps the system from scaling linearly. The current practical solution is to simply find a good fixed amount of parallel operations that keep the throughput at a satisfactory level even when going through a large number of quickly answering target domains. While this is probably still improvable, the insertion of the collected data into the MySQL back-end currently seems to be the bottleneck, so there is no need for immediate action on this part.
An interesting approach for future improvements might be monitoring the time each request takes and act accordingly. For regular requests, launch a thread to replace the current one after its termination.
For slow requests, start an additional thread.
For very fast requests, let the thread terminate without launching a new one.

4.2.6 File descriptors

Especially when working with asynchronous I/O, the amount of available file descriptors for the process is of great importance. File descriptors are (among other things) used to track the availability of socket data in Unix like systems. Limiting the amount of file descriptors effectively limits the possible concurrency of the application. Especially when developing on Mac OS X, the default limit of only 256 file descriptors for user processes starts to be a problem quickly. Hitting this limit can cause the application to break at seemingly random points during execution. Setting the file descriptor limit to a higher value using the "ulimit" command before executing the application solves this problem. Another approach is to limit the amount of file descriptors that will be allocated by the framework. In Eventmachine, the set_descriptor_table_size() method can put an upper limit on the amount of descriptors. This, however, could harm performance and throughput.

4.3 Link collection

4.3.1 RegExp vs HTML Parsing

When it comes to extracting hyperlinks from an HTML document, there are basically two approaches:

- Use an HTML/XML parser
- Use regular expressions

Using an HTML parser and focusing on the "href" attribute of HTML <a> elements will only extract actual links, not domain names that are in text areas. Using a regular expression will usually extract anything that looks like a link, regardless if it is actually in the right position or not. Problems with html parsers might arise when it comes to HTML that is not well-formed. An example of a regular expression to detect links to external sites (in our case: links that do not start with a slash) inside of a href attribute looks like this:

```
/href.?=.?["'](["^\/].*?)["']/i
```

A simple benchmark comparing two c-based html parsers for Ruby (Nokogiri[11] and Hpricot[12]), both using CSS and XPATH selectors to the previously mentioned regular expression, gives the following results when parsing the HTML of the reddit.com frontpage 250 times:

[11] http://nokogiri.org/

[12] http://wiki.github.com/hpricot/hpricot/

Method	Duration in s
regexp	0.352365
hpricot (xpath)	3.808862
hpricot (xpath, no href)	7.473591
hpricot (css)	5.392564
nokogiri (xpath)	4.532942
nokogiri (xpath, no href)	4.223752
nokogiri (css)	4.958362

In our case, "no href" means that we were only looking for a elements ("//a") and not for a elements with href ("//a[@href]"). The extraction of the link is happening after we collected the results. Since the regular expression offers 10 times faster performance in Ruby, using it seems to be the best solution for the problem. Just in case that Ruby encounters strange encodings, a fallback leads to a second method using the Nokogiri HTML/XML parser.

4.3.2 Adaptive crawl depth

After processing a few million domains, collecting new ones becomes a problem. Websites tend to link only in a certain "circle" of similar sites. For example, technology sites will usually link to other technology websites, French domains tend to link to other French domains and so on. Some examples of previous work dealing with this are Cho, J.'s and Garcia-Molina's "Efficient Crawling Through URL Ordering"[24], Marc Najork and Janet L. Wiener's "Breadth-first crawling yields high-quality pages"[25], Baeza-Yates et al. with "Crawling a Country: Better Strategies than Breadth-First for Web Page Ordering"[26] and Daneshpajouh et al. with "A Fast Community Based Algorihm for Generating Crawler Seeds Set"[27].

Our initial strategy for this problem was adjusting the depth of a default crawl. Instead of just extracting links from the root of the domain ("/"), as soon as the domains in queue dropped below a certain number, the depth of the crawl increased. Visiting other links allowed us to gather more domains while still keeping a good performance when we have enough new domains to crawl in queue. This behaviour was discontinued with the introduction of other mechanisms like zone files (see 4.3.6) and seed sites(see 4.3.4) in favour of a cleaner and simpler code base. While the approach is an elegant solution to the problem that offers a trade-off between domain throughput and link-collection, it was a classical case of "reinventing the wheel". Just parsing links to external domains from the root of a website still allows us to gather enough new links to detect domains that are not part of the initial seed of domains, while keeping the code simple and the throughput high.

4.3.3 Twitter

Another good source of links (especially new and popular domains) is the micro-blogging service Twitter. Twitter offers a streaming API that delivers live JSON data from the service directly to the application. There are several different ways to gather new links. The most effective one would probably be the "links" stream available for special applications at `http://stream.twitter.com/1/statuses/links.json`. The official description of this resource is as follows:

> Returns all statuses containing http: and https:. The links stream is not a generally available resource. Few applications require this level of access. Creative use of a combination of other resources and various access levels can satisfy nearly every application use case.

Twitter had an amount of over 3000 tweets per second with an average of 750 tweets per second in June of 2010, according to the official company blog[13]. Even if only 5% of tweets have a link in them, that would mean that we would have to deal with over 160 links per second, which would require some caching and deduplication mechanisms to be able to

[13]`http://blog.twitter.com/2010/06/another-big-record-part-deux.html`

keep the impact on the back-end minimal. This is especially true if we consider the amount of time it takes to resolve all of the shortened URLs that are usually used on twitter. Services like bit.ly, TinyURL or is.gd allow users to post longer links on the character limited micro-blogging service. These can be resolved using HTTP head requests and checking the "location" field in the resulting HTTP redirect as can be seen in this example:

```
$ curl -I http://bit.ly/9vGKyw
HTTP/1.1 301 Moved
Server: nginx/0.7.42
[...]
Location: http://blog.marc-seeger.de
```

An easier alternative to the "full" stream is the usage of twitters regular non-streaming REST API. A simple request to `http://search.twitter.com/search.json?q=blog+filter:links&result_type=recent` is able to return all tweets that include the word "blog" as well as an http link. Using targeted searches like this, we are able to filter out an interesting subset of results without having to constantly work through the complete twitter timeline. This also allows us to include twitter as just another seed site.

4.3.4 Seed sites

Besides the previously mentioned service Twitter, there are a handful of other interesting websites that specialize in collecting links, while also providing a convenient free API. Reddit[14] and Delicious[15] are sites that specialize in helping their users collect links to interesting external sites. While Reddit has a big community aspect with comments and specific subsections, Delicious is more focused on providing a tagging solution for linked content. Both of the services offer a REST based JSON

[14]http://www.reddit.com/
[15]http://www.delicious.com/

API which we can use to automatically extract newly posted links from both services. While Delicious offers us a broader variety, Reddit seems to focus on high-priority websites. Both sites offer developer-friendly terms of service that allow easy access to the provided data within certain limitations. The allowed request-frequency is well within the bounds of being useful for this project.

4.3.5 DMOZ and Wikipedia

An interesting collection of links to bootstrap the crawling process can be found at the "DMOZ open directory project"[16]. DMOZ inherited its name from "directory.mozilla.org", its original domain name. The directory of domains provided by the project is available as a RDF dump file which can easily be fed into the crawler using a little script file. A comparable list of links to external domains is being provided by the Mediawiki foundation[17]. The files with the pattern "*-externallinks.sql" provide records of links to external domains found within the contents of the Wikipedia.

4.3.6 Zone files

One of the most interesting alternatives to manually gathering links is the usage of zone files. There are services (e.g. premiumdrops.com) that provide copies of the zone files of major top-level domains. At the time of writing, this includes the following TLDs:

[16]`http://www.dmoz.org/`
[17]`http://download.wikimedia.org/dewiki/latest/`

TLD	Domains
.com	89 million
.net	13 million
.org	8.4 million
.info	6.7 million
.biz	2 million
.us	1.6 million

Using these lists, it is possible to keep feeding new domains to the crawler without paying too much attention to link-collection beyond the parsing of the initial HTML page at the root of the domain, which has to be done for fingerprinting purposes anyway.

4.3.7 Recrawling

Another interesting possibility of making sure that the crawler does not run out of domains late in the crawling cycle is a certain recrawling of domains. If the crawler hits a link to a domain that has last been crawled more than a certain amount of time (e.g. 3 months) ago, just readding it to the crawl queue will keep the information of popular sites relatively current and also scrap some new domains from their front pages every now and then. Something to consider is that checking for the date of a domain's last crawl adds an additional database request for every single encountered link to an external domain. This additional load can be cut down by using local caches or checking in only 50% of the cases.

4.4 Distribution

One of the most interesting problems to solve was the possibility of distributing the crawling process to more than one machine. While the system design targets high-performance more than horizontal scalability, adding lightweight nodes that only deal with the crawling is a desired feature to enhance throughput. When looking at figure 4.1, it can be seen that the only connection to the back-end that the actual crawling process needs is the access to the Redis process. Jobs are gathered from Redis and results (information, discovered links) are immediately written back to it. There is no further interaction for the crawling process with any other back-end systems. This allows us to leave a "main" server to do the heavy back-end work (MySQL, Solr) while letting the actual crawling take part on machines that are much more lightweight when it comes to RAM and storage space. The main component for the crawling machines is available CPU power and network bandwidth. Since the final writes to Redis can be done in an asynchronous fashion and do not require any feedback, only getting the next job from the queue actually suffers from the higher network latency. Besides this read penalty, the only other thing suffering from a networked connection to Redis is the number of file descriptors and kernel processes that do the asynchronous I/O. Inserting the data, checking for the existence of links, and recrawling old sites can all be put into small processes on the server hosting Redis itself. This also helps keeping the actual storage back-end logic concentrated in two processes; the link checker and the database inserter.

The actual long term storage back-end can be switched easily by just replacing the specific code in these two modules. An additional change would be required to insert the newly found data into Solr.

Chapter 5

Profiling-Methods

This section of the thesis describes a few commonly used means of profiling/debugging several parts of the architecture that have been helpful in the development of the project. There are a lot of tools that come with a default Unix/Linux installation that allow for generic profiling of running applications or third party services. The advantage of analysing applications using these tools is that they allow one to gather information about any running process. There is no need need for a modified VM or special startup options of the application itself.

5.1 lsof

The ’lsof’ command lists information about files opened by processes. It has to be noted that ’files’ also includes sockets in Unix. This helps us gather information about network connections opened by an application. This little tool can help, among other things, to detect memory leaks. I noticed that the amount of RAM the crawler was using went up linearly with the amount of domains it crawled. Just by doing a simple listing of the files opened by the crawler process, I noticed that the file for the geo-ip database had been loaded hundreds of times. This file is used to determine the country a server is in. Seeing that this might be the cause of the memory consumption allowed me to quickly look through the sourcecode and notice that a reference to this file was leaked everytime a domain was analysed. An example output of lsof looks like this:

```
$ lsof -p 230 | grep REG
texmaker 230 mseeger  [...] /Applications/[...]/texmaker
texmaker 230 mseeger  [...] /Applications/[...]/QtXml
texmaker 230 mseeger  [...] /Applications/[...]/QtGui
texmaker 230 mseeger  [...] /Applications/[...]/QtNetwork
texmaker 230 mseeger  [...] /Applications/[...]/libqkrcodecs.dylib
texmaker 230 mseeger  [...] /Applications/[...]/libqjpeg.dylib
texmaker 230 mseeger  [...] /Applications/[...]/libqmng.dylib
```

5.2 dtrace and strace

Something similar to lsof can be done using dtrace (on OSX and Solaris) or strace (on Linux). To list all applications that open files on disk, dtrace allows to do a simple

```
#dtrace -n 'syscall::open*:entry { printf("%s %s",execname,copyinstr(arg0));
[...]
  0  18510    open:entry gedit-bin /Users/mseeger/Downloads/tyranttest.rb
  0  18510    open:entry gedit-bin /Users/mseeger/Downloads/tyrant_search.rb
  0  18510    open:entry gedit-bin /Users/mseeger/Downloads/Python.pdf
  0  18510    open:entry gedit-bin /Users/mseeger/Downloads/keanu_sad.jpg
[...]
```

In this case, you can see the files being opened by the "gedit-bin" application. In general, this can help to find unintended operations that should be taken out of a loop. Dtrace even offers tools like "iotop" that show you I/O utilization on a per process basis and many more. On OSX, Apple ships an application called "instruments" which provides a convenient user interface to dtrace and has many templates that help to debug running processes and monitor specific system events. Strace also allows to debug processes at runtime and to inspect their system calls. It is not as powerful and scriptable as dtrace, but is supported on Linux and works equally as well for simple tasks.

5.3 curl

When it comes to HTTP interaction, curl can be considered the Swiss army knife of tools. It allows one to easily create GET, HEAD or POST requests to arbitrary HTTP endpoints, and displays the results in an easily readable text view. The response to a simple HEAD request looks like this:

```
$ curl -I buytaert.net
HTTP/1.1 200 OK
Server: nginx/0.7.62
Content-Type: text/html; charset=utf-8
X-Powered-By: PHP/5.2.4-2ubuntu5.10
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Cache-Control: store, no-cache, must-revalidate, post-check=0, pre-check=0
Set-Cookie: SESS1d728079e066e9a00738cd212ea24f73=08un8nf9slbudvasb09it86pa4; [...]
Last-Modified: Sun, 05 Sep 2010 12:16:48 GMT
Vary: Accept-Encoding
Date: Sun, 05 Sep 2010 12:16:49 GMT
X-Varnish: 819908815
Age: 0
Via: 1.1 varnish
Connection: keep-alive
X-Cache: MISS
```

Curl is especially helpful when trying to find unique parts of a CMS that might be used to detect its usage in the fingerprinting stage of the crawler. In this case, the Expires HTTP header is set to "Sun, 19 Nov 1978 05:00:00 GMT" which usually indicates that a website is running on the Drupal CMS.

5.4 mtop

When using MySQL as a back-end, there are a lot of available profiling tools that a developer can chose from. A simple tool allowing monitoring of a running MySQL process, even on the server, is mtop[1]. It allows looking at long-running queries, checking the cache hit ratios, and counting the amount of queries per second the system is dealing with. If development requires more than such a simplistic view, tools such as Jet Profiler[2] allow a more in-depth review of running queries compared to mtop.

5.5 JRuby based profiling

As mentioned in the Ruby section2.2.4, JRuby allows Ruby code to run on Hotspot, the Java Virtual Machine. Not only does it provide native threads without a global interpreter lock, it does also allows the usage of most of the Java Profiling tools. Charles "Headius" Nutter is a core developer for the JRuby project and has provided several excellent blog-posts that deal with the available tools and their usage with JRuby:

- "Browsing Memory the JRuby Way"[3] gives an introduction to jmap and jhat
- "Finding Leaks in Ruby Apps with Eclipse Memory Analyzer"[4] introduces the Eclipse Memory Analyzer, a more interactive way of finding memory leaks than the jmap/jhat combination
- "Browsing Memory with Ruby and Java Debug Interface"[5] introduces the Java Debug Interface that not only allows heap inspection,

[1] `http://mtop.sourceforge.net/`

[2] `http://www.jetprofiler.com/`

[3] `http://blog.headius.com/2010/07/browsing-memory-jruby-way.html`

[4] `http://blog.headius.com/2010/07/finding-leaks-in-ruby-apps-with-eclipse.html`

[5] `BrowsingMemorywithRubyandJavaDebugInterface`

but also allows installing breakpoints and single stepping through running code.

Chapter 6

Fingerprinting

One of the main features of the project is the ability to detect the software that powers a website in the background. This chapter will explain the different ways to detect this software and go into details about the gathering Drupal specific data (used modules and the Drupal release). While working on this thesis, Patrick S. Thomas released a paper called "Blind Elephant: Web application fingerprinting with static files"[12] that also deals with this topic. I suggest to also read his paper which focuses on version detection for different content management systems.

6.1 CMS detection

At the time of writing, the fingerprinting mechanism created for this project is able to detect 80 different content management systems. The interface used is pretty simple:

Input : HTML of the root ('/') page + HTTP Headers

Output : true or false

The main way of detecting a CMS is checking for specific things the generated pages include such as:

6.1.1 The Generator meta tag

Berners-Lee and Connolly described the meta element in RFC 1866 ("Hypertext Markup Language - 2.0")[13] as follows:

> The <META> element is an extensible container for use in identifying specialized document meta-information.

The "generator" meta tag is used to give information about which application created the HTML code on the page. While this tag can be removed without any problems, in the large majority of cases, it exists in the default installation state of a content management system. It can help us to determine the CMS and sometimes even its specific version. This is a sample from a default installation of the Wordpress blogging engine:

```
<meta name="generator" content="WordPress 2.9.2" />
```

While security through obscurity is not a proper way to keep an application safe, it might be a good idea to remove this tag just in case a vulnerability of a certain version of the CMS would make it easily detectable as a possible target.

6.1.2 Included files

Since the upcoming of the term "AJAX", most bigger content management systems use Javascript to allow their users to edit forms or interact with the website without reloading the complete page. For this, external Javascript files are usually required. In the case of Drupal, a file called "drupal.js" is included using the script tag:

```
<script type="text/javascript" src="/misc/drupal.js?3"></script>
```

Please pay attention to the problems section (6.1.9) of this chapter for further comments on this.

6.1.3 Javascript variables

After a javascript library such as jQuery has been loaded, it is sometimes necessary to set certain CMS specific parameters. In the case of Drupal, for example, it is often the case that the HTML code includes this call:

```
<!--//--><![CDATA[//><!--
jQuery.extend(Drupal.settings, {"basePath":"\/","googleanalytics":
{"trackOutgoing":1,"trackMailto":1,"trackDownload":1,
"trackDownloadExtensions":"7z|aac|avi|js|[...]|xml|zip"}});
//--><!]]>
```

6.1.4 Comments

Another way of a content management system to mark created pages is by inserting an HTML comment inside the page source. A nice example of this behaviour is the Kajona[3] CMS which features this text:

```
<!--
Website powered by Kajona Open Source Content Management Framework
For more information about Kajona see http://www.kajona.de
-->
```

And in the case of TYPO3, this comment can often be found:

```
<!--
This website is powered by TYPO3 - inspiring people to share!
TYPO3 is a free open source Content Management Framework initially created by Kaspe
TYPO3 is copyright 1998-2009 of Kasper Skaarhoj. Extensions are copyright of their
Information and contribution at http://typo3.com/ and http://typo3.org/
-->
```

6.1.5 Special paths

Besides the actual required files themselves, a lot of CMS systems use fixed internal paths that give out information about the software in use. Wordpress, for example, has a lot of paths that feature the "wp-" prefix.

```
<script type='text/javascript'
src='http://[...].de/wp-includes/js/jquery/jquery.js?ver=1.3.2'>
</script>
```

As with the included Javascript files, please also pay attention to the problems section (6.1.9) of this chapter for further comments on this. There is also the possibility of actually checking certain application specific paths that are available for some content management systems. A good example would be the path "/sites/all" for the Drupal CMS.

Another possibility to detect a CMS would be to check for the default admin interface URL. For example, Typo3 uses "/typo3" and Wordpress based blogs use "/wp-admin". While these additional checks would allow an even better accuracy in the detection process, they would also require additional HTTP requests. Especially when you consider the amount of available (and detectable) CMS systems, this would result in a huge performance loss.

6.1.6 Images

Another thing to look out for are "powered by" images that are also usually accompanied by matching "alt" tags in the "img" element. Some versions of the Alfresco CMS, for example, feature a "Powered by Alfresco" icon that looks like this in the HTML source code:

```
<img src="assets/images/icons/powered_by_alfresco.gif"
alt="Powered by Alfresco" width="88" height="32" />
```

6.1.7 HTTP headers

Some CMS are detectable by looking at the HTTP headers of the package being returned by the web server. One of the more prominent examples is Drupal which sets the "EXPIRES" header to "Sun, 19 Nov 1978 05:00:00 GMT". In this case, November 19th 1978 is the birthday of the original creator and project lead for Drupal. Before actually adding this to the fingerprinting engine, it is always a good idea to check the source code (if available) to be sure that it was not a load balancer or the web server that added a certain HTTP header.

6.1.8 Cookies

Another interesting part of the HTTP headers are cookies. A lot of content management systems use session-cookies to track user behaviour or restrict access using role based access control systems. An example of these cookies can be found in headers created by the WebGUI CMS:

```
Set-Cookie: wgSession=qmDE6izoIcokPhm-PjTNgA;
```

Another example would be the blogtronix plattform which also sets a cookie:

```
Set-Cookie: BTX_CLN=CDD92FE[...]5F6E43E; path=/; HttpOnly
```

6.1.9 Problems

While the techniques that have been showed so far are able to detect a large percentage of CMS systems, there are as always exceptions to those rules. When it comes to the detection of included Javascript/CSS files, "CSS/Javascript aggregation" can circumvent detection of specific files. When aggregating Javascript or CSS files, a single file containing all of the initially included files is generated and linked to. This will generally improve site load time and might look like this in HTML:

```
<script type="text/javascript"
src="http://[...].org/files/js/js_35664871d2d4304ed9d4d78267fc9ed5.js">
</script>
```

The filename is usually just generated, and without inspection of the file, it is hard to say which CMS (or other preprocessor) created the file. Another problem arises when it comes to using CMS-specific paths in the detection process. A good example of CMS paths are the "wp-" prefixed paths that Wordpress uses to store content. Most images posted by a user will usually have "/wp-content/" in the path-name. The problem is that "hotlinking", meaning the linking to images that are on another person's webspace is pretty common on the internet. By simply relying on paths, a lot of false positives could arise. Selecting paths carefully might help with this. In the case of Wordpress, the "wp-includes" path tends to be a good candidate for detection, as most people will not hotlink another person's Javascript/CSS.

6.2 Web servers

When it comes to detecting web servers, there are a lot of sophisticated methods to do so. Dustin Willioam Lee describes this in detail in his Master's Degree Thesis "HMAP: A Technique and Tool For Remote Identification of HTTP Servers" [[16]] Since the fingerprinting of web servers is not the main goal of the current project, I decided to simply rely on the "server" field in the HTTP header. A typical HTTP response including the "Server" field looks like this:

```
$ curl -I marc-seeger.de
HTTP/1.1 200 OK
Content-Type: text/html
Connection: keep-alive
Status: 200
X-Powered-By: Phusion Passenger (mod_rails/mod_rack) 2.2.15
Content-Length: 1398
Server: nginx/0.7.67 + Phusion Passenger 2.2.15 (mod_rails/mod_rack)
```

Using this response, we can deduct that the website in question is running on the nginx webserver or is at least load-balanced by it. While some sites try to keep their output to a minimum, the simple check for the "Server" header is enough to detect a large majority of web servers.

6.3 Drupal specific data

Since Drupal is of specific interest to this project, the detection of modules and version numbers is being processed in an additional step, since the combination of HTML + HTTP headers is usually not enough to detect all of it.

6.3.1 Modules

Drupal is a modular CMS and heavily relies on modules, to extend its functionality. There are thousands of modules and being able to detect the ones running on a specific site could be helpful when searching for specific combinations (e.g. sites using an online shop module such as 'Ubercart').

The initial idea for gathering modules is a simple check of the HTML code for included files that are in a directory called "modules". In this case, for example, the website uses modules such as "poll" (to create online polls) and "lightbox2" (a javascript image viewer):

```
[...]
<style type="text/css" media="all">@import "/modules/poll/poll.css";</style>
[...]
<script type="text/javascript" src="/sites/all/modules/lightbox2/js/lightbox.js"></
[...]
```

While the detection problems (6.1.9) with CSS/JS aggregation also hold true for this detection method, it still is able to analyze a large percentage of sites without any additional HTTP overhead.

6.3.2 Versions

When it comes to the detection of the Drupal version (e.g. "6.17"), however, the regular HTML source code does not provide enough information. While Wordpress, for example, provides version information within the generator tag, Drupal does not have a generator tag or comment in the generated HTML that would be able to tell us the version number.

There are some things that can be used to distinguish between different versions of the CMS: The CHANGELOG.txt file is often present and accessible in installations. it carries the exact version number of the running software. This is a sample from the official drupal.org page:

```
// $Id: CHANGELOG.txt,v 1.253.2.39 2010/06/02 18:52:32 goba Exp $

Drupal 6.17, 2010-06-02
----------------------
- Improved PostgreSQL compatibility
- Better PHP 5.3 and PHP 4 compatibility
```

While it is possible that people could upgrade to a newer version of Drupal but not replace the changelog.txt file in the process, the percentage is small enough to be neglected. If the changelog file is not accessible, there is another way to do a version check. The "Taxonomy" module is present in most Drupal installations. If it is accessible (/modules/taxonomy/taxonomy.inf
is also has the version information present:

```
[...]
; Information added by drupal.org packaging script on 2010-06-02
version = "6.17"
project = "drupal"
datestamp = "1275505216"
[...]
```

Using these two tests, we were able to detect version information on approximately 75% of detected Drupal sites (sample size: 300.000 domains running Drupal).

6.4 Hosting Provider

Another interesting task was to create the ability to detect the hosting provider for a certain domain. Usually, it's as easy as having the IP-Ranges of the big providers available, and checking weather or not the server's IP Address is inside their network. For larger providers such as Amazon EC2, the ranges can be found using a simple Google search. For other providers, this information is not always that easy to find. It can usually manually be extracted by just using the "whois" tool to query a server's IP:

```
$ whois 208.97.187.204
[...]
NetRange:       208.97.128.0 - 208.97.191.255
CIDR:           208.97.128.0/18
[...]
NetName:        DREAMHOST-BLK5
```

In this example, the network 208.97.128.0/18 seems to belong to the provider "Dreamhost". While it would be nice to be able to automatically extract this information, the whois protocol sadly does not define a formatting for this information. When querying for different IPs, it is soon evident that parsing this data would be a huge project on its own. The initial steps for such a project called "ruby-whois"[1] already exist, but the main focus is not on the network data and for the time being, contributing code that would change that is not within the direct scope of the crawler project. The pragmatic solution to this problem is to simply get a list of the biggest hosting providers using Google, and spend a few minutes finding out their network ranges.

[1] http://www.ruby-whois.org/

6.5 Geolocation

Another interesting property of a website is the location of its web server. This can help to determine the target audience for commonly used top level domains like .net without going into CPU intensive analysis of the website's language. Since IP Ranges are being distributed in a geographically consistent manner, databases which map an IP address to a country are publicly available and a reliable source of information. One example would be the "GeoLite" data by MaxMind. Their "open data license" is developer friendly and the only thing they request is this:

> All advertising materials and documentation mentioning features or use of this database must display the following acknowledgment: "This product includes GeoLite data created by MaxMind, available from http://maxmind.com/

Existing Ruby-bindings to this database exist[2] and work as expected.

6.6 Industry

One of the hardest requirements was the categorization of domains into different industries. An interesting approach would be the implementation of machine learning techniques to classify the data. There are several algorithms that allow classification of data. A neural network or a support vector machine would allow automatic classification, but this would probably be a project on its own and does not seem feasible.

An easy alternative to this is the usage of a web-service such as jigsaw.com[3]. Before using this data, it would be wise to study their API's terms of service[4] and be sure to comply in terms of what can be saved and which parts of the data are actually restricted. After enough domain +

[2]http://geoip.rubyforge.org/
[3]http://developer.jigsaw.com/
[4]http://developer.jigsaw.com/Developer_Terms_of_Use

industry combinations have been collected, using the already mentioned supervised learning algorithms might be an interesting follow up project.

Chapter 7

Conclusion and Outlook

The implemented architecture has been recorded processing up to 100 domains per second on a single server. At the end of the project the system gathered information about approximately 100 million domains. The collected data can be searched instantly and the automated generation of statistics is visualized in the internal web interface.

For the future, there are two areas of interest that could enhance the current design. The first one is the fast and steady pace of new developments in the "nosql" space. Being able to switch the back-end to something like Riak (with its announced search integration[1]) or newer versions of MongoDB (that might be using more finely grained locks) would be worth a future evaluation. The addition of one of these solutions as a native data-source to Solr would also help to improve the current architecture. The second area of interest would be the complete transformation of the last remaining synchronous parts of the system into an asynchronous and callback-driven workflow.

The project was a great exercise in applying my theoretical knowledge about scalability to a real-life scenario with large amounts of data, and I hope that this thesis allows others to learn from my results.

[1]`http://www.basho.com/riaksearch.html`

Bibliography

[1] E. F. Codd
A relational model of data for large shared data banks.
Communications of the ACM archive
Volume 13 , Issue 6 (June 1970)
Pages: 377 - 387
Year of Publication: 1970
ISSN:0001-0782

[2] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener.
Computer Networks: The International Journal of Computer and Telecommunications Networking archive
Volume 33 , Issue 1-6 (June 2000)
Pages: 309 - 320
Year of Publication: 2000
ISSN:1389-1286

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall and Werner Vogels
Dynamo: Amazon's Highly Available Key-Value Store
In the Proceedings of the 21st ACM Symposium on Operating Systems Principles, Stevenson, WA, October 2007.

[4] Ola Ågren, Department of Computing Science, Umeå University, SWEDEN
Assessment of WWW-Based Ranking Systems for Smaller Web Sites,
2006
`http://www.dcc.ufla.br/infocomp/artigos/v5.2/art07.pdf`

[5] Link-Based Similarity Measures for the Classification of Web Documents
Journal of the American Society for Information Science and Technology
Volume 57 , Issue 2
Pages: 208 - 221
Year of Publication: 2006
ISSN:1532-2882

[6] Column-stores vs. row-stores: how different are they really?
International Conference on Management of Data archive
Proceedings of the 2008 ACM SIGMOD international conference on Management of data
Vancouver, Canada
SESSION: Research Session 20: Tuning and Probing table of contents
Pages: 967-980
Year of Publication: 2008
ISBN:978-1-60558-102-6

[7] Request for Comments: 2616
Network Working Group
Hypertext Transfer Protocol – HTTP/1.1
June 1999
`http://www.ietf.org/rfc/rfc2616.txt`

[8] Request for Comments: 1034
Network Working Group
DOMAIN NAMES - CONCEPTS AND

FACILITIES
November 1987
`http://tools.ietf.org/html/rfc1034`

[9] Event-Driven I/O - A hands-on introduction
Marc Seeger
HdM Stuttgart
July 13, 2010

[10] Thousands of Threads and Blocking I/O
The old way to write Java Servers is New again (and way better)
Paul Tyma
Presented at the SD West 2008 conference
`http://paultyma.blogspot.com/2008/03/writing-java-multithreaded-servers.html`

[11] No Callbacks, No Threads: Async webservers in Ruby 1.9
Ilya Grigorik
Railsconf 2010: `http://www.oscon.com/oscon2010/public/schedule/detail/13709`
OSCON 2010: `http://en.oreilly.com/rails2010/public/schedule/detail/14096`
Video: `http://www.viddler.com/explore/GreggPollack/videos/40/`

[12] BLINDELEPHANT: WEB APPLICATION FINGERPRINTING WITH STATIC FILES
Patrick S. Thomas
BlackHat USA - July 28, 2010

[13] Request for Comments: 1866
Network Working Group, T. Berners-Lee & D. Connolly
1995
`http://www.ietf.org/rfc/rfc1866.txt`

[14] Request for Comments: 2068
Hypertext Transfer Protocol – HTTP/1.1
Fielding, et. al.
1997
`http://www.ietf.org/rfc/rfc2068.txt`

[15] Bitcask - A Log-Structured Hash Table for Fast Key/Value Data
Justin Sheehy and David Smith
2010
`http://downloads.basho.com/papers/bitcask-intro.pdf`

[16] HMAP: A Technique and Tool For Remote Identification of HTTP Servers
DUSTIN WILLIAM LEE B.S. (Gonzaga University)
1990
`http://seclab.cs.ucdavis.edu/papers/hmap-thesis.pdf`

[17] A Practical Introduction to Data Structures and Algorithm Analysis
Clifford A. Shaffer. Prentice-Hall
1997
`http://people.cs.vt.edu/~shaffer/Book/`

[18] Bitmap Index Design and Evaluation
Chee-Yong Chan und Yannis Ioannidis
Proceedings of the 1998 ACM SIGMOD Conference.
`http://www.comp.nus.edu.sg/~chancy/sigmod98.pdf`

[19] An Adequate Design for Large Data Warehouse Systems: Bitmap index versus B-tree index
Morteza Zaker, Somnuk Phon-Amnuaisuk, Su-Cheng Haw
International Journal of Computers and Communications, Issue 2, Volume 2, 2008 1 `http://www.universitypress.org.uk/journals/cc/cc-21.pdf`

[20] Trie Memory,
Edward Fredkin

CACM, 3(9):490-499
September 1960
`http://portal.acm.org/citation.cfm?id=367400`

[21] High Performance MySQL, Second Edition - Optimization, Backups, Replication, and More
ByBaron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, et al.
Publisher:O'Reilly Media
June 2008
ISBN: 978-0-596-10171-8

[22] R-trees: a dynamic index structure for spatial searching
International Conference on Management of Data archive
Proceedings of the 1984 ACM SIGMOD international conference on Management of data table of contents Boston, Massachusetts
SESSION: Physical database design table of contents
Pages: 47 - 57
Year of Publication: 1984
ISBN:0-89791-128-8

[23] Dynamo: Amazon's Highly Available Key-value Store
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall and Werner Vogels
In the Proceedings of the 21st ACM Symposium on Operating Systems Principles
Stevenson, WA, October 2007.
`http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf`

[24] Efficient Crawling Through URL Ordering
Cho, J. and Garcia-Molina, H. and Page, L.
Seventh International World-Wide Web Conference (WWW 1998)
1998
`http://ilpubs.stanford.edu:8090/347/1/1998-51.pdf`

[25] Marc Najork and Janet L. Wiener
Breadth-first crawling yields high-quality pages.
In Proceedings of the Tenth Conference on World Wide Web, pages 114–118, Hong Kong, May 2001.
`http://www10.org/cdrom/papers/pdf/p208.pdf`

[26] R., Castillo, C., Marin, M. and Rodriguez, A.
Crawling a Country: Better Strategies than Breadth-First for Web Page Ordering. In Proceedings of the Industrial and Practical Experience track of the 14th conference on World Wide Web, pages 864–872
2005
`http://www.dcc.uchile.cl/~ccastill/papers/baeza05_crawling_country_better_breadth_first_web_page_ordering.pdf`

[27] Shervin Daneshpajouh, Mojtaba Mohammadi Nasiri, Mohammad Ghodsi
A Fast Community Based Algorihm for Generating Crawler Seeds Set
In proceeding of 4th International Conference on Web Information Systems and Technologies (WEBIST-2008)
2008
`http://ce.sharif.edu/~daneshpajouh/publications/A%20Fast%20Community%20Based%20Algorithm%20for%20Generating%20Crawler%20Seeds%20Set.pdf`