Debian Code Search

Michael Stapelberg

Bachelor-Thesis
Studiengang Informatik

Fakultät für Informatik
Hochschule Mannheim

2012-12-19

Betreuer: Prof. Dr. Jörn Fischer
Zweitkorrektorin: Prof. Dr. Astrid Schmücker-Schend

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 14. Dezember 2012

Thank you

I want to say thanks to Dr. Jörn Fischer, Dr. Astrid Schmücker-Schend, Axel Wagner, Michael Sommer, Stefan Breunig, Hannes Frederic Sowa, Russ Cox, Matthias Schütz, Cian Synnott, Kristian Kraljic, Maik Fischer and Tollef Fog Heen for numerous tips, valuable discussions, proof-reading and any other kind of support.

Abstract

Since the shutdown of Google Code Search in January 2012, developers of free/libre open source software (FOSS) have lacked a source code search engine covering the large corpus of open source computer program code.

Without such a tool, developers could not easily find example code for poorly documented libraries. They could not quickly determine the scope of a problem — for example, figuring out how many packages call a specific library function with a bug in it.

This thesis discusses the design and implementation of Debian Code Search, a search engine based on a re-implementation of the ideas behind Google Code Search, but with Debian's FOSS archive as a corpus, and using Debian's clean meta data about software packages to improve search results.

The work presented includes optimizing Russ Cox's re-implementation to work with a large corpus of data, refactoring it as a scalable web application backend, and enriching it with various ranking factors so that more relevant results are presented first. Detailed analysis of these main contributions as well as various smaller utilities to update, index and rank Debian packages are included.

With the completion of this thesis, Debian Code Search is working and accessible to the public at `http://codesearch.debian.net/`. Debian Code Search can be used to search 129 GiB of source code, typically within one second.

Abstract

Seit der Abschaltung von Google Code Search im Januar 2012 herrschte ein Mangel an Suchmaschinen für Programmcode, die genügend Open Source Software (FOSS) durchsuchen. Open Source-Entwickler waren nicht mehr in der Lage, bei der Arbeit mit schlecht dokumentierten Bibliotheken schnell Beispielcode zu finden. Sie konnten nicht mehr mit wenig Aufwand die Auswirkung eines Problems abschätzen, z.B. wie viele Pakete eine Funktion aufrufen, in der sich ein Fehler zeigte.

Diese Bachelor-Arbeit diskutiert die Architektur und Implementierung von Debian Code Search, einer Suchmaschine, die Debians FOSS-Archiv als Datengrundlage nutzt und Debians große Menge an sauberen Metadaten um die Suchergebnisse untereinander zu sortieren.

Debian Code Search baut auf den Codesearch tools, einer Implementation derselben Technologie, die auch Google Code Search nutzte, auf. Diese Implementation von Russ Cox wurde optimiert um große Datenmengen zu verarbeiten und so verändert, dass sie in einer Web-Anwendung genutzt werden kann. Weiterhin wurde sie durch verschiedene Bewertungskriterien angereichert, die dafür sorgen, dass die relevantesten Treffer zuerst angezeigt werden.

Neben dem Optimieren der Codesearch tools für den Einsatz in einer großen Suchmaschine wurden im Rahmen dieser Arbeit auch kleine Hilfsprogramme entwickelt, die das Aktualisieren, Indizieren und Bewerten der Debian-Pakete übernehmen.

Mit der Fertigstellung dieser Arbeit funktioniert Debian Code Search und steht der Öffentlichkeit unter `http://codesearch.debian.net/` zur Verfügung. Debian Code Search kann genutzt werden, um 129 GiB an Quelltext in üblicherweise weniger als einer Sekunde zu durchsuchen.

Contents

1 Introduction **1**

2 Debian Code Search: An Overview **3**
2.1 Target audience and use cases 3
2.2 Other search engines 4

3 Architecture **6**
3.1 Architecture and user interface design principles 6
3.2 Modifications to the Codesearch tools by Russ Cox 7
3.2.1 Comparison to Google Code Search 7
3.3 High level overview 8
3.4 Architecture overview 9
3.5 Resource requirements and load-balancing 10
3.6 Programming language and software choice 13
3.7 Source corpus size 14
3.8 The trigram index 15
3.8.1 Trigram index limitations 16
3.8.2 Looking up trigrams in the index 17
3.8.3 Lookup time, trigram count and index size 18
3.8.4 Posting list decoding implementation 19
3.8.5 Posting list encoding/decoding algorithm 21
3.8.6 Posting list query optimization 22
3.9 Updating the index 24
3.10 Logging and monitoring 25
3.10.1 Logging 25
3.10.2 Monitoring 26
3.11 Caching 27
3.11.1 Implicit caching (page cache) 27
3.11.2 Explicit caching 27

4 Search result quality **30**
4.1 Metric definition 30
4.2 Ranking 31
4.2.1 Ranking factors which can be pre-computed per-package 31
4.2.2 Ranking factors which depend on the query 31
4.2.3 Ranking factors which depend on the actual results 32
4.2.4 Popularity contest 33
4.2.5 Reverse dependencies 34

4.2.6 Modification time of source code files . . . 35
4.3 Sample search queries and expected results . . . 36
4.4 Result quality of Debian Code Search . . . 38
4.5 Result latency of Debian Code Search . . . 41
4.5.1 Trigram lookup latency . . . 41
4.5.2 Source matching latency . . . 42
4.6 Language bias . . . 44
4.7 Duplication in the archive . . . 45
4.8 Auto-generated source code . . . 46
4.9 Test setup, measurement data and source code . . . 47
4.10 Overall performance . . . 48
4.10.1 Measurement setup . . . 48
4.10.2 Queries per second by query term . . . 50
4.10.3 Queries per second, replayed logfile . . . 50
4.10.4 Real-world response times . . . 51
4.11 Performance by processing step (profiling) . . . 52
4.11.1 Before any optimization . . . 52
4.11.2 After optimizing the trigram index . . . 53
4.11.3 After re-implementing the ranking . . . 53

5 Conclusion **54**

6 Future Work **55**

7 Appendix A: Screenshots **56**

Bibliography **58**

List of Figures **61**

List of acronyms **63**

1 Introduction

This work describes the motivation, architecture and evaluation of Debian Code Search, a web search engine for computer program source code.

With the shutdown of Google Code Search in January 2012, a very useful tool for many Open Source (FLOSS[1]) developers vanished. Debian Code Search aims to bring back very fast (much less than one second) regular expression search across all 129 GiB of FLOSS software currently included in Debian.

Having access to a quick search engine over the entire source code empowers Debian developers and developers in general: researching which packages use a certain function is suddenly feasible without first downloading and extracting tens of gigabytes of source code. Finding and referring to the implementation of a library function in bug reports becomes easy. Overall packaging quality might increase due to easy access to a vast corpus of packaging examples.

Other currently available source code search engines do not have up-to-date and high quality results. Some even target only one specific programming language. Furthermore, none are available as open source and therefore cannot be used in Debian. Having access to the code means that Debian-specific improvements can be made and there is no risk of the search engine disappearing because of a single company's decision.

The goal of this bachelor thesis is to build and evaluate a FLOSS source code search engine based on Russ Cox's Codesearch tools[2]. This involves modifying the tools in such a way that they can be used within the backend of a web application. Furthermore, since the Codesearch tools were not specifically designed for use in a large-scale public search engine, performance bottlenecks and architectural problems or limitations need to be identified and fixed or circumvented. It is desirable that the resulting system is scalable across multiple machines so that high demand can be satisfied.

Since the Codesearch tools are written in Go, Debian Code Search is written in Go, too. While it is not the primary topic of this thesis, the reader will get an impression of how well Go is suited for this kind of application and where this young programming language still has weak spots.

Chapter 2 identifies the target audience for Debian Code Search (DCS), lists several use cases and explains why the Debian project provides a good setting for this work.

Chapter 3 explains the principles and architecture of DCS. This not only includes a high level overview but also an explanation of the different processes which are involved and how they can be distributed on multiple machines to handle more load. This chapter also covers the trigram index and the various optimizations thereof which were necessary to ensure DCS

[1] FLOSS stands for Free/Libre Open Source Software

runs fast enough.

Chapter 4 explains how the search engine ranks search results to display the most relevant results first. It also defines a metric for search engine quality and evaluates Debian Code Search based on that metric. Finally, it covers limitations of the system and presents overall performance results, both with synthetic work loads and replayed real-world logfiles.

Chapter 5 summarizes the achievements of this work and chapter 6 gives an outlook on what can be improved with future work.

2 Debian Code Search: An Overview

> “Debian is a computer operating system composed of software packages released as free and open source software primarily under the GNU General Public License along with other free software licenses.”[21]

Debian Code Search, built during this thesis, is a search engine covering all source code contained in the Debian archive. By using an existing archive of free software[1], the time- and resource-consuming crawling of the public internet for source code is avoided. Additionally, Debian Code Search takes advantage of the large amount of high-quality metadata which is readily available within Debian for ranking search results.

2.1 Target audience and use cases

A public instance of Debian Code Search is available for members of the Debian project and anyone who is interested in using it. The target audience is primarily any FLOSS developer or user (especially using Debian) and everybody who deals with Debian packaging.

Typical use cases are:

- The documentation of many open source libraries is too sparse to be helpful. By entering the function name of a library function like `XCreateWindow`, you can find either the function’s definition, its implementation or source code which uses the function (to be used as an example).
- Given the name of an algorithm, find an implementation of that algorithm.

  Due to the search result ranking, the implementations that are presented first are very likely to be widely used and well-tested implementations unless the algorithm is very specific.
- Given a backtrace with debug symbols of a crashed program, quickly locate and browse the source code referred to in the backtrace to understand and solve the problem at hand.

  This frees the person looking into the bug report from having to download the relevant source code first. Instead, the source code can be browsed online without having to wait for downloads to finish and without having to use disk space for storing source code. This lowers the barrier and hopefully leads to more people working on these kind of bug reports.
- Find fragments within Debian packaging files. E.g. find all Debian-specific patches which use the old `gethostbyname` for resolving DNS names instead of its replacement

[1] Free software as defined by the Debian Free Software Guidelines (DFSG)

getaddrinfo[2]; Or finding all calls of the old ifconfig and route utilities[3].

- Judge whether a function is in wide-spread use before depending on it in a project.

  An example is the function pthread_mutexattr_setpshared. After having read the documentation, you might still wonder whether this function is widely supported, for example on other UNIX operating systems such as FreeBSD, and whether you are using it as its authors intended. Debian Code Search reveals that the Apache web server uses the function, so depending on it should not lead to bad surprises.

Being a Debian project, it is important that all parts of the system are available as FLOSS themselves. This ensures that anyone can send improvements and prevents single points of failure: when the current maintainer cannot continue for some reason, other people can take over.

2.2 Other search engines

Of course, the idea of a code search engine is not a novel idea. There are several code search engines available on the web currently, and others have been available in the past. Unfortunately, many are too specific because they only support one programming language, have a very small source code corpus or do not deliver high-quality results.

Nullege http://nullege.com is a language-specific code search engine for Python. Its corpus contains 21 630 open-source projects[4] as of 2012-12-07, typically hosted on github or SourceForge. Nullege uses its own search engine[5].

Unfortunately, more detailed information about nullege is sparse. There is no research paper about it. Searching for "nullege" on nullege yields no results, therefore it has to be assumed that nullege is not open source.

Sourcerer sourcerer is a language-specific code search engine for Java. It is a research project from University of California, Irvine. sourcerer's corpus contains 1555 open source Java projects.

Unfortunately, the public version of sourcerer is broken at the moment and only returns an HTTP 404 error.

Koders http://koders.com is a code search engine for open source code. After being acquired by Black Duck Software in 2008, the public site remains free to use[6], but (private) code search is a component of the Black Duck Suite. Koders' corpus contains 3.3 billion lines of code, but is not actively updated at the moment because the whole Koders software is in flux.[7]

[2] This is worthwhile because getaddrinfo is required for correctly implementing IPv6. Full IPv6 support is a Debian Release Goal: http://wiki.debian.org/ReleaseGoals/FullIPv6Support
[3] Done without a search engine by Adam Borowski in this discussion on the Debian development list: http://lists.debian.org/debian-devel/2012/08/msg00163.html
[4] Nullege searches open source projects according to http://nullege.com/pages/about. The exact number of projects is visible on http://nullege.com
[5] https://twitter.com/nullege/status/7637551310
[6] http://en.wikipedia.org/wiki/Koders
[7] http://corp.koders.com/about/

Koders uses ctags, a widely used FLOSS tool for indexing source code, to build its search index[7]. Due to Koders' use of ctags, only code which is written in one of the 41 languages which ctags supports[8] is present in Koders' index. Searches are always case-insensitive and regular expressions are not supported. However, one can search specifically for class names (e.g. `cdef:parser`), method names, interface names and combinations thereof.

Koders uses a proprietary heuristic to rank code: "Essentially, the more reuse a code file receives, the more likely it will appear higher in your search results."[7]

ohloh code `http://code.ohloh.net` is the successor to Koders. ohloh code indexes the projects which are registered at the open source directory site ohloh, currently also owned by Black Duck Software.

The search features are very similar to those of Koders.

Krugle `http://krugle.org` is the public code search engine of Krugle, a code search company which was acquired by Aragon Consulting Group in 2009. Krugle's product is Krugle Enterprise, a searchable source code library[9].

Features of Krugle include searching for function calls, function definitions, class definitions, within certain projects, by document type, within a specific time period and by license.

Krugle's corpus consists of a mere 450 open source projects[10].

Google Code Search Starting out as the intern summer project of Russ Cox[2], Google Code Search was the first public search engine to offer regular expression searches over public source code. It was launched in October 2006 and was shut down in January 2012 "as part of Google's efforts to refocus on higher-impact products"[2].

Search features include filtering by license, file path, package, class or function (or any combination)[11]. Every filter supports regular expressions[12].

A version of Google Code Search which operates on all repositories hosted at Google Code is still available at `http://code.google.com/codesearch`.

The description of its inner workings and the availability of an open-source re-implementation of its tools were the main motivations for this work.

[8] `http://ctags.sourceforge.net/languages.html`
[9] `http://venturebeat.com/2009/02/17/aragon-buys-software-code-analysis-co-krugle/`
[10] `http://www.krugle.org/projects/`
[11] `http://code.google.com/codesearch`
[12] `http://www.google.com/intl/en/help/faq_codesearch.html#regexp`

3 Architecture

The best way to bring Debian Code Search to everybody who wants to use it is to implement it as a web application. A user only needs to have a web browser installed, and a computer without a web browser is unthinkable for many people. Since there are libraries for HTTP available in every popular programming language, extending Debian Code Search to offer an API — for example for Integrated Development Environment (IDE) integration — is easy.

Creating and running a modern web application or web service touches a lot of different areas and technologies, such as DNS, HTTP servers, load-balancers for scalability and redundancy, HTML and CSS, caching, browser compatibility, logging and monitoring.

This chapter first describes the goals and principles after which the architecture was designed and then describes the actual architecture and how it adheres to these principles.

3.1 Architecture and user interface design principles

1. The service should be a pleasure to use and helpful for its users. This includes good search result quality, a visually pleasing and consistent design and low latency. Marissa Mayer (formerly at Google, now CEO of Yahoo) says that a delay of 0.5 s caused a 20% drop in search traffic [10]. At Amazon, A/B tests showed that very small delays (increments of 100 milliseconds) would result in substantial revenue drops.
2. The architecture should follow current best-practices and as such be easy to understand and maintain. This is an important point to consider when starting a new Debian project. Ultimately, the goal is to have the software running on machines administered by Debian System Administration (DSA). Using well-known software as building blocks reduces the workload for DSA and lowers the barrier for external contributions.

   Also, it is very likely that the author of this work will not be able to spend his time on this project as the single administrator for a long time. Therefore, making it easy for other people to grasp the system is a good thing for attracting contributors.
3. Stateless design should be preferred wherever possible. In such a design, components can be stopped/started or exchanged at runtime without a lot of work or negative consequences for users. This encourages small improvements and a more engaging workflow. It also decreases the number of ways in which things can go wrong, because there is no corruption of state leading to undefined behavior.
4. Leverage concurrency and parallelism. For the most part, modern CPUs are being designed with more cores rather than faster clock speeds [5]. Having a software architecture which works well on such systems allows for performance increases in the future by simply replacing the hardware, just like it used to be with increasing clock speeds.

3.2 Modifications to the Codesearch tools by Russ Cox

This entire work would not have been possible without Russ Cox's article "Regular Expression Matching with a Trigram Index or How Google Code Search Worked"[2]. In this article, he describes how he built Google Code Search as a summer intern at Google in 2006.

He also released the Codesearch tools with that article, an implementation of the original concepts in the Go programming language. Debian Code Search (DCS) uses large parts of the code. Essentially, DCS is a web frontend for the Codesearch tools with a large index.

Several changes have been made to the code:

- The original tools directly printed their results to standard output because they worked under the assumption that they are always run interactively from a terminal. Functions have been changed to return an array of results instead. New data types to hold the results were added.
- DCS displays source code context lines in its search results, while the original Codesearch tools did not. This required changing the algorithm for matching a regular expression so that it stores previous lines and subsequent lines.
- Posting list decoding was profiled, rewritten in hand-optimized C code and retrofitted with a custom query optimizer to satisfy performance requirements.
- The indexing tool has been replaced with a DCS-specific indexing tool that skips some files which are not interesting and writes the index into multiple shards due to maximum file size limitations of the Go programming language.
- DCS ranks filenames before letting the original Codesearch tools search for results in these files. Also, the search results are ranked before they are displayed. These changes are not directly in the code of the Codesearch tools, but they are absolutely necessary for running a good search engine.

3.2.1 Comparison to Google Code Search

Since both Debian Code Search (DCS) and Google Code Search (GCS) are using the same technique, it is worthwhile to point out the differences.

- GCS crawled the public internet while DCS indexes source code in Debian.
- GCS had to use heuristics for determining metadata such as license, authors, and so on, while DCS can use the metadata existing in Debian.
- GCS's user interface was more sophisticated than DCS's is currently, likewise for the search query keywords.
- GCS is closed source, so it cannot be deployed/modified/used anywhere, while DCS is licensed under the BSD license, which allows for using it within enterprises.

A direct performance or search result quality comparison of DCS with GCS is not possible, since the latter is offline since January 2012. Even if it were still online, it would not be possible to run it in a controlled environment for reliable measurements.

3.3 High level overview

Figure 3.1 shows the flow of a request in Debian Code Search. Section 3.4, page 9, covers a request's life in more detail, including the various processes which are involved.

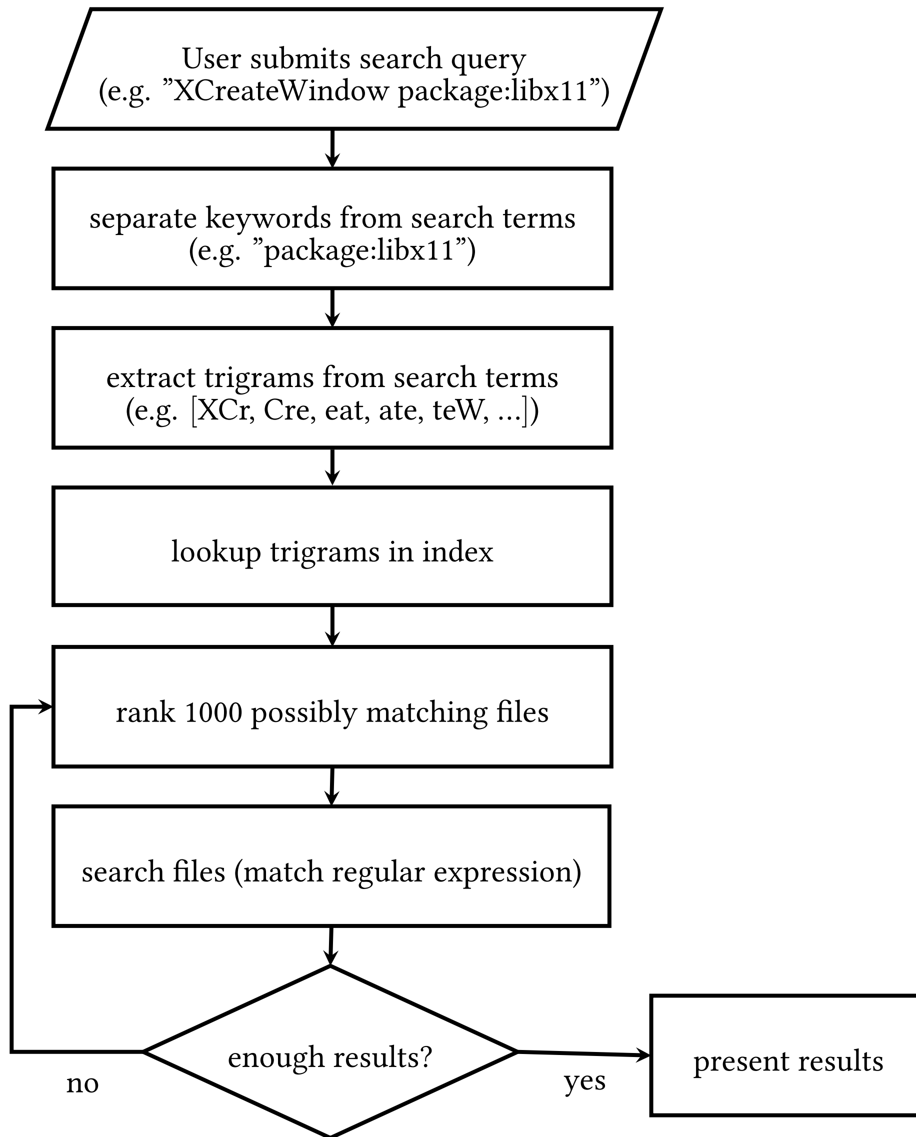

Figure 3.1: High level overview of a request's flow in Debian Code Search.

3.4 Architecture overview

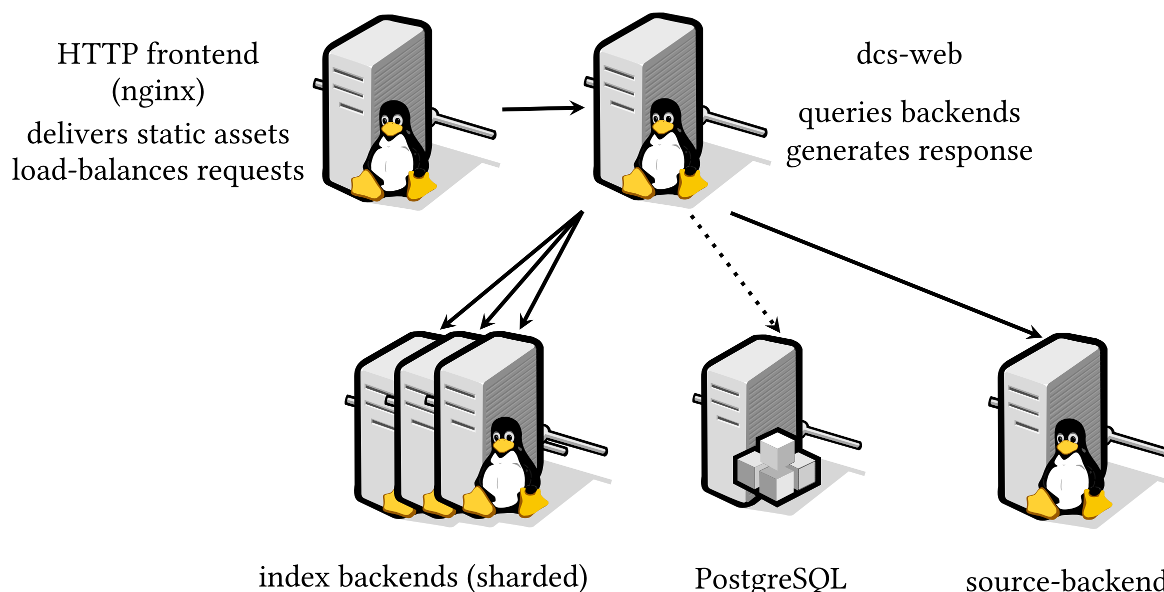

Figure 3.2: Architecture overview, showing which different processes are involved in handling requests to Debian Code Search.[1]

Debian Code Search consists of three different types of processes (dcs-web, index-backend, source-backend) running "behind" an NGINX webserver and accessing a POSTGRESQL database when starting up.

When a new request comes in to `http://codesearch.debian.net/`, NGINX will deliver the static index page. However, when the request is not for a static page but an actual search query, say `http://codesearch.debian.net/search?q=XCreateWindow`, the request will be forwarded by NGINX to the dcs-web process.

dcs-web first parses the search query, meaning it handles special keywords contained in the query term, e.g. "`filetype:perl`", and stores parameters for pagination. Afterwards, dcs-web sends requests to every index-backend process and gets back a list of filenames which possibly contain the search query from the index-backends. See section 3.5 on why there are multiple index-backend instances. See section 3.8, "The trigram index", on page 15 for details of the index-backend lookup process.

These filenames are then ranked with ranking data loaded from POSTGRESQL in such a way that the filename which is most likely to contain a good result comes first. Afterwards, the list of ranked filenames is sent to the source-backend process, which performs the actual searching using a regular expression matcher, just like the UNIX tool `grep(1)`.

As soon as the source-backend has returned enough results, dcs-web ranks them again with the new information that was obtained by actually looking into the files and then presents the results to the user.

[1] This figure has been created with dia. The icons are gnomeDIAicons, licensed under the GPL.

3.5 Resource requirements and load-balancing

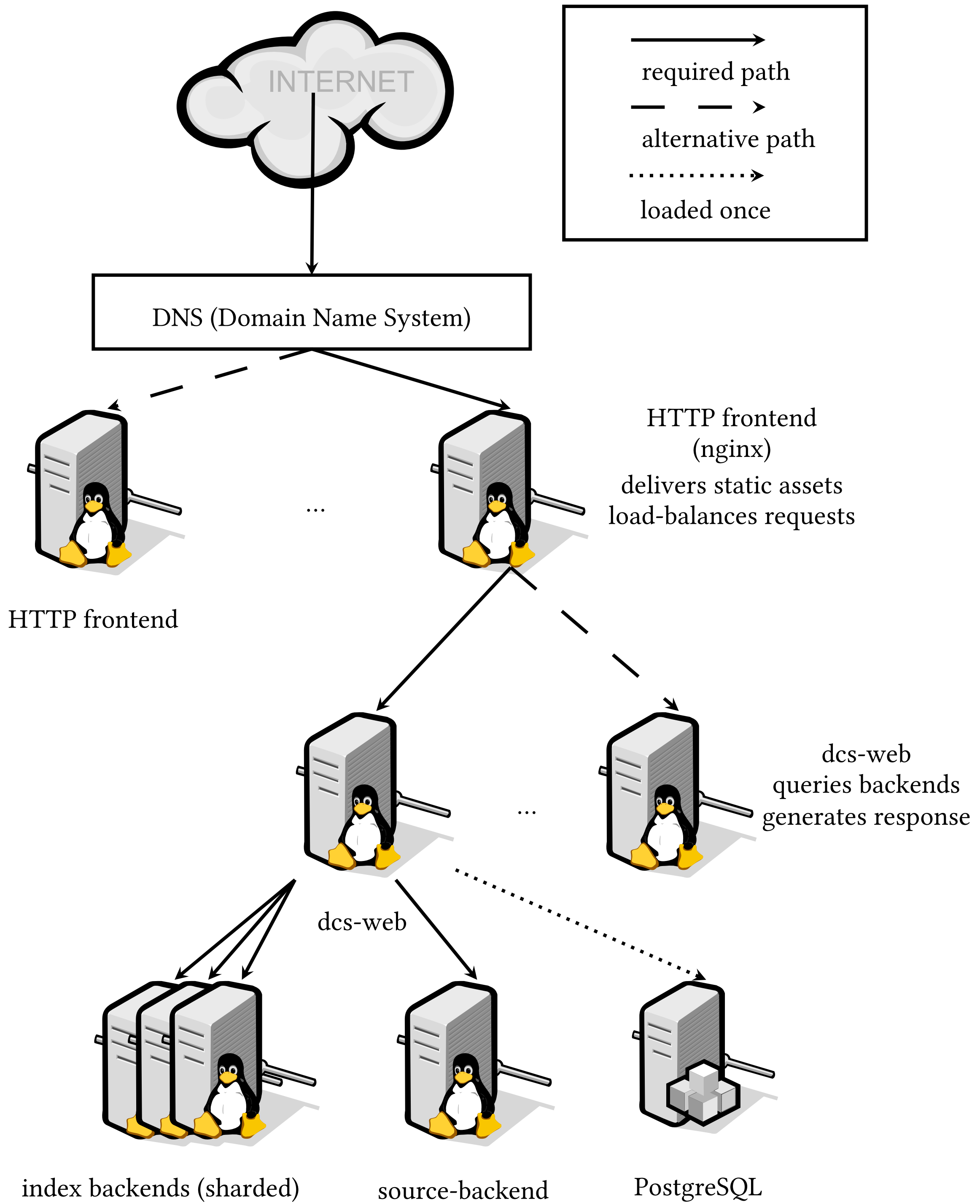

Figure 3.3: Architecture overview with load-balancing possiblities.

As you can see in figure 3.2, when the user requests `http://codesearch.debian.net/`, the browser first needs to resolve the name `codesearch.debian.net` to an IP address using the Domain Name System (DNS). This is the first step where the load can be balanced between multiple servers: the browser will connect to the first IP address it gets, so a DNS server can just return all IP addresses in a different order (e.g. round-robin). DNS for debian.net is hosted by the Debian project, so Debian Code Search doesn't have to setup or maintain any software or hardware for that.

After resolving the hostname, the browser will open a TCP connection on port 80 to the resolved IP address and send an HTTP request. This request will be answered by the HTTP frontend webserver, which is the second step where the load can be balanced and redundancy can be added: The frontend can split the load between the available backends and requests can still be answered if a certain number of backends fail.

Furthermore, the backend only has to communicate with the frontend, therefore the burden of handling TCP connections — especially slow connections — is entirely on the frontend.

Requests which can be answered from the cache (such as static pages, images, stylesheets and JavaScript files) can be served directly from the frontend without causing any load on the backend. The HTTP frontend runs on a Debian Code Search machine.

`dcs-web` receives actual search requests and runs on a Debian Code Search machine. This might be the same machine as the frontend runs on, or a different, dedicated machine, if the demand is so high that this is necessary to maintain good performance. To answer a request, dcs-web needs to perform the following steps:

1. Query all index backends. The index is sharded into multiple index backend processes due to technical limitations, see section 3.8.1, page 16.
2. Rank the results.
3. Send the results to one of the source backends, which performs the actual searching.
4. Format the response.

Each index backend and source backend corresponds to one process, which typically will run on the same machine that dcs-web runs on. Should the index size grow so much that it cannot be held by one machine anymore, index backends can also run on different machines which are connected by a low-latency network.

Should it turn out that disk bandwidth is a problem, one can run multiple source backends, one for each disk. These source backend processes can be run on the same machine with different disks or on different machines, just like the index backend processes.

Index backends, if all deployed on a single machine, need to run on a machine with at least 8 GiB of RAM. Not keeping the index in RAM means that each request needs to perform a lot of additional random disk accesses, which are particularly slow when the machine does not use a solid state disk (SSD) for storing the index[28].

Source backends profit from storing their data on a solid state disk (SSD) for low-latency, high-bandwidth random file access. Keeping the filesystem metadata in RAM reduces disk access even further. The more RAM the machine which hosts the source backend has, the better: unused RAM will be used by Linux to cache file contents[24], so search queries for popular files might never even hit the disk at all, if the machine has plenty of RAM. 16 GiB

to 128 GiB of RAM are advisable. More is unnecessary because Debian currently contains 129 GiB of uncompressed source code and it is very unlikely that 100 % of it needs to be held in RAM.

Additional metadata used for ranking is stored in PostgreSQL but only loaded into memory by dcs-web on startup, thus the dotted line.

Every component in this architecture, except for PostgreSQL, communicates with each other using HTTP, which makes deploying different parts of the architecture on different machines on-demand as easy as changing a hostname in the configuration file.

3.6 Programming language and software choice

Main programming language

Since this work is based on the existing codesearch tools published by Russ Cox[2], which are written in Go[16], it is a very natural decision to choose Go for building the web service, too.

Additionally, Go was designed in the first place to be used as a language for building server software, especially for the web. This is not surprising given that the origin of the language is Google, where Go is used in production[2]. Other companies have generally found their experience with Go to be positive[3] and the author of this work has done several successful projects with Go himself.

Despite being a relatively new language, Go's performance is good enough for the volume of requests expected in the near future for Debian Code Search: a typical HTTP service implemented in Go handles about 1500 to 2000 requests/second[19].

Note that not every little tool used within this project has to be written in Go. Where appropriate, other languages such as UNIX shell script have been used, for example for unpacking the Debian source mirror.

Software building blocks

The webserver NGINX is used to ease the load on the search backend by serving cached or static assets, to terminate HTTP connections for end users and to distribute the load between multiple servers. NGINX has very compelling performance[8] with a small memory and CPU footprint even for thousands of connections. While there are alternatives with similar performance such as LIGHTTPD, the author prefers NGINX for its simpler and cleaner configuration. Also, NGINX seems to be the fastest most popular webserver[18].

To store computed information like the per-package ranking without inventing a custom file format for every little piece of information, the relational database PostgreSQL is used. Again, this choice is because of personal preference and familiarity with the software.

Development of the software uses GIT as the source control management system (SCM). Experience has shown that it is the most popular SCM for Open Source projects and thus removes another barrier for contributors.

[2] http://golang.org/doc/go_faq.html#Is_Google_using_go_internally
[3] https://developers.google.com/events/io/sessions/gooio2012/320/

3.7 Source corpus size

The largest amount of data in the whole project is taken up by the source code of all Debian packages. Due to space and bandwidth considerations, source code is usually distributed in compressed archives. After mirroring the Debian archive, you end up with 36 GiB of data[4]. Note that this is only the `main` distribution of Debian, not the `non-free` or `contrib` distributions. The latter were not included in this project because of licensing.

Since the source code is mirrored in compressed form, it is necessary to decompress it before being able to search through it. Since DCS needs to search through the files for every result that is displayed, it makes sense to keep the uncompressed source code permanently.

Filtering out unwanted data in an early stage saves resources in every subsequent step. Therefore, all packages whose names end in "-data" are not unpacked[5]. With this step, the amount of data to process is reduced by 2316 MiB to 33 GiB.

Unpacking the remaining packages[6] results in 129 GiB of source code[7] in 8 117 714 files. It is hard to predict the growth of the Debian archive, but reserving 250 GiB of storage for uncompressed source code should leave some room for future growth while being commonly available in today's server hardware[8].

In comparison to a typical web search engine corpus, our source corpus is small: In 1998, Google crawled 24 million web pages resulting in 147 GiB of data[1].

---

[4] More precisely: 37 226 384 B as of 2012-07-25. The Debian archive was mirrored with `debmirror -a none --source -s main -h ftp.de.debian.org -r /debian /debian`.

[5] It is a Debian convention that packages like games that come with a large amount of data (levels, images, etc.) are split up into two packages. Being only a convention, removing all packages ending in "-data" also removes some false-positives such as console-data. A better way of distinguishing data packages from regular packages would be beneficial.

[6] To properly unpack each Debian source package, the tool `dpkg-source` was used. It takes care of Debian-specific patches and the various different Debian package formats.

[7] With "source code" meaning everything that is contained in the source package. This might include documentation or other files which are not actually indexed later on.

[8] e.g. Hetzner's `http://www.hetzner.de/hosting/produkte_rootserver/ex4` comes with 3 TB of disk space

3.8 The trigram index

In order to perform a regular expression match, it is impractical to search through the whole corpus for each query. Instead, an inverted n-gram index is required:

> "[...] [an] n-gram is a contiguous sequence of n items from a given sequence of text or speech."[23]

In our case, the text is either the program source code when indexing, or the user's search terms when processing a query. An n-gram of size 3 is called a trigram, a good size for practical use:

> "This sounds more general than it is. In practice, there are too few distinct 2-grams and too many distinct 4-grams, so 3-grams (trigrams) it is."[2]

As an example, consider the following simple example of a search query:

`/Google.*Search/` (matching Google, then anything, then Search)

Obviously, only files which contain both the terms "Google" and "Search" should be considered for the actual regular expression search. Therefore, these terms can be translated into the following trigram index query:

`Goo` *AND* `oog` *AND* `ogl` *AND* `gle` *AND* `Sea` *AND* `ear` *AND* `arc` *AND* `rch`

This process can be performed with any query, but for some, the translation might lead to the trigram query matching more documents than the actual regular expression. Since the trigram index does not contain the position of each trigram, these kinds of false positives are to be expected anyway.

The full transformation process is described in Russ Cox' "Regular Expression Matching with a Trigram Index"[2].

3.8.1 Trigram index limitations

This work is based on Russ Cox' Codesearch tools[2] (see also 3.2, page 7), which define an index data structure, index source code and support searching the previously indexed source code using regular expressions.

Google Code Search was launched in 2006, but Russ Cox' Codesearch tools are implemented in Go, a language which was created in late 2007[6]. Therefore, they cannot be the original Google Code Search implementation, and have not been used for a large-scale public search engine.

On the contrary, Russ recommends them for local repositories, presumably not larger than a few hundred megabytes:

> "If you miss Google Code Search and want to run fast indexed regular expression searches over your local code, give the standalone programs a try."[2]

One obvious limitation of the index implementation is its size limitation of 4 GiB, simply because the offsets it uses are 32-bit unsigned integers which cannot address more than 4 GiB. Another not so obvious limitation is that even though an index with a size between 2 GiB and 4 GiB can be created, it cannot be used later on, since Go's mmap implementation only supports 2 GiB[9].

To circumvent this limitation while keeping modifications to Russ Cox' Codesearch tools to a minimum[10], a custom indexing tool to replace Codesearch's `cindex` has been implemented. This replacement, called `dcsindex`, partitions the index into multiple files. Additionally, it tries to reduce the index size in the first place by ignoring some files entirely.

Partitioning does not only work around the index size limitation, it also comes in handy when distributing the index across multiple machines. Instead of running six index shards on one machine, one could run three index shards on each of two machines.

---

[9] This is because `len()` always returns an `int` to avoid endless loops in a `for i:=0; i<len(x); i++` expression (i is always signed, so `len()`'s results needs to be, too). An `int` currently is 32 bits big, even on 64-bit platforms. See `http://code.google.com/p/go/issues/detail?id=2188`.

[10] The more modifications are made, the higher the amount of future maintenance work will be. Since Debian projects are volunteer-driven[11] most of the time, keeping maintenance work low is a very desirable goal.

3.8.2 Looking up trigrams in the index

In order to modify the index data structure in such a way that it is more suited for our purposes, we first need to understand how it works. Figure 3.4 is an illustration of the different parts of the index. On the left, you can see each section within the index file. On the right, an example entry for each of the different sections is provided (except for LIST OF PATHS, which is unused in our code). Each section of the index is sorted, and the number of entries of the NAME INDEX and POSTING LIST INDEX are stored in the TRAILER.

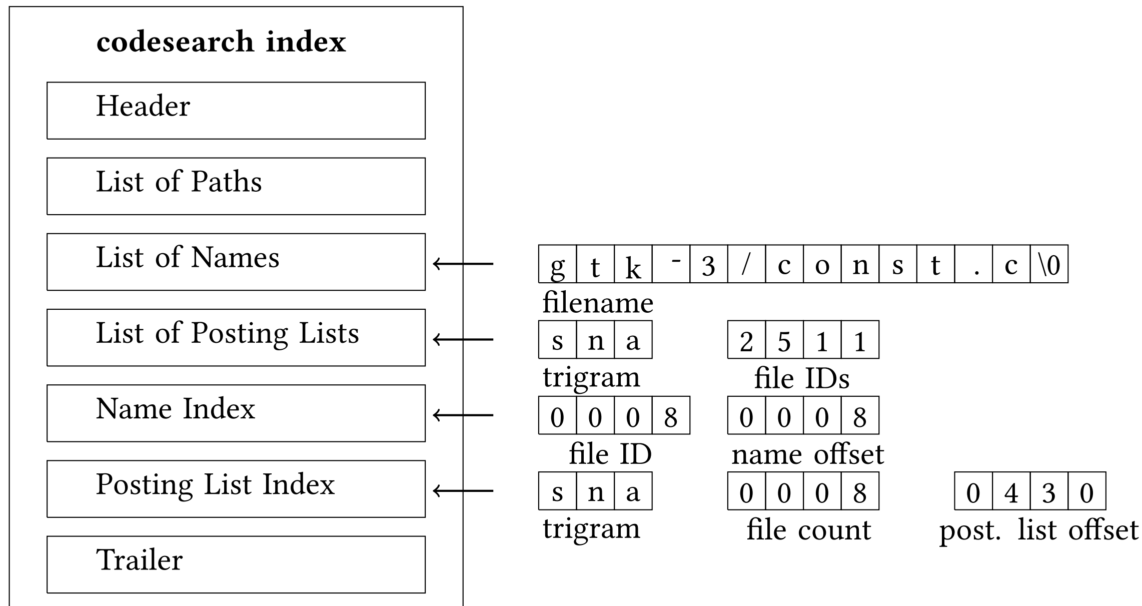

Figure 3.4: The Codesearch index format. Trigram lookups are performed as described below.

Assuming the list of all files which contain the trigram “sna” needs to be obtained, the following steps have to be performed:

1. Seek to the POSTING LIST INDEX and perform a binary search to find the entry for trigram “sna”. The entry reveals the file count and a byte offset (relative to the first byte of the index) pointing at the entry for “sna” in the LIST OF POSTING LISTS.
2. Seek to the entry for “sna” in the LIST OF POSTING LISTS and decode the varint [7]-encoded list of file IDs.
3. For each file ID, seek to the appropriate position in the NAME INDEX. The byte offset pointing to the filename in the LIST OF NAMES is now known.
4. For each name offset, seek to the appropriate position in the LIST OF NAMES and read the filename (NUL-terminated).

3.8.3 Lookup time, trigram count and index size

As you can see in figure 3.5, the trigram lookup time increases linearly with the trigram count. This is expected: if you look up four trigrams instead of two, it takes about twice as long, no matter how fast the algorithm for the lookup is.

You can also observe that the index size plays a role: since the average complexity of binary search is $\mathcal{O}(\log n)$ (with n being the number of trigrams stored in the index), there is a logarithmic connection between the lines representing different index sizes.

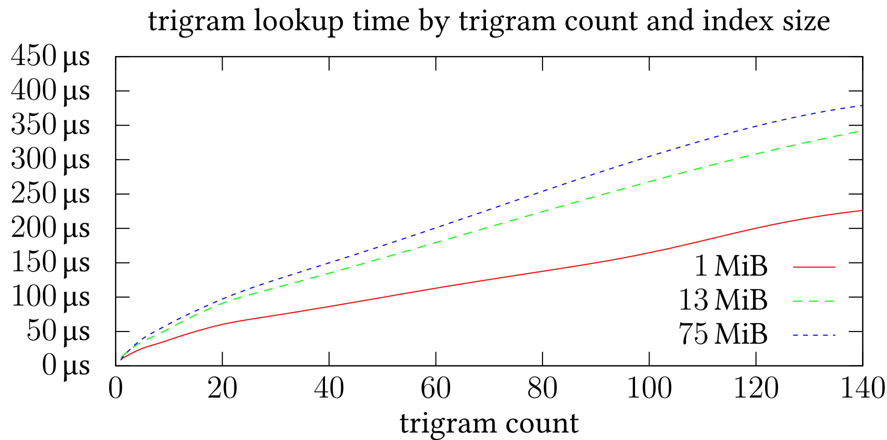

Figure 3.5: Trigram lookup time grows linearly with the query's trigram count and logarithmically with the index size. 33 queries with varying length were examined.

Figure 3.6 contains measurements for the same queries as figure 3.5, but sorted by combined posting list length. E.g. if the query is "foob", and the trigram "foo" has a posting list containing 300 files, and the trigram "oob" has a posting list containing 200 files, the combined posting list length is 500. Looking at the figure, one realizes that as the combined posting list length increases, the posting list lookup time increases linearly.

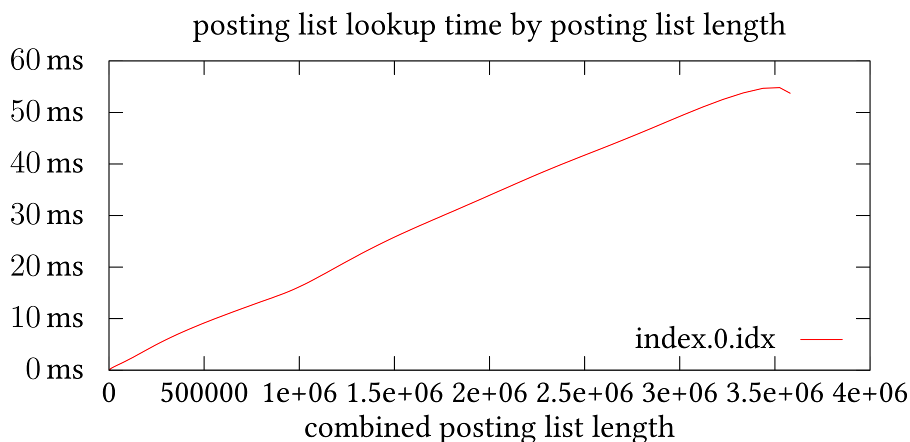

Figure 3.6: Posting list decoding time increases linearly with combined posting list length.

3.8.4 Posting list decoding implementation

Section 3.8.3 states that the overall posting list lookup time increases linearly with the posting list length. This is very comprehensible, but the absolute time required for lookups of large posting lists intuitively seems very high (≈ 40 ms).

Profiling the code revealed that the majority of time is spent actually decoding the posting lists — the binary search to find the offset and length is absolutely negligible. By dividing the time through the amount of bytes decoded, our hunch is confirmed by the realization that 7 MiB s^{-1} is far too slow for an Intel Core i7 2600K CPU.

The Go implementation of `Uvarint`, which comes as part of the standard library in the package `encoding/binary`[11], is fairly straightforward (see listing 3.1), though using it obviously leads to a lot of function call overhead and it uses 64-bit integers which have to be converted to 32-bit integers.

Listing 3.1: Go varint decoding

```
func Uvarint(buf []byte) (uint64, int) {
  var x uint64
  var s uint
  for i, b := range buf {
    if b < 0x80 {
      if i > 9 || i == 9 && b > 1 {
        return 0, -(i + 1) // overflow
      }
      return x | uint64(b)<<s, i + 1
    }
    x |= uint64(b&0x7f) << s
    s += 7
  }
  return 0, 0
}

func (r *postReader) next() bool {
  for r.count > 0 {
    r.count--
    delta64, n := binary.Uvarint(r.d)
    delta := uint32(delta64)
    r.d = r.d[n:]
    r.fileid += delta
    return true
  }
  r.fileid = ^uint32(0)
  return false
}

func postingList(r *postReader)
    []uint32 {
  x := make([]uint32, 0, r.max())
  for r.next() {
    x = append(x, r.fileid)
  }
  return x
}

x := postingList(r)
```

Listing 3.2: DCS C varint decoding

```
static __attribute__((hot)) const uint32_t
    uvarint(const uint8_t *restrict*data) {
  uint32_t b, c, d;
  if ((b = *((*data)++)) < 0x80) {
    return b;
  } else if ((c = *((*data)++)) < 0x80) {
    return (uint32_t) (b & 0x7F) |
           (uint32_t) (c << 7);
  } else if ((d = *((*data)++)) < 0x80) {
    return (uint32_t) (b & 0x7F)       |
           (uint32_t)((c & 0x7F) << 7) |
           (uint32_t) (d << 14);
  } else {
    return (uint32_t) (b & 0x7F)        |
           (uint32_t)((c & 0x7F) << 7)  |
           (uint32_t)((d & 0x7F) << 14) |
           ((uint32_t)(*((*data)++)) << 21);
  }
}

int cPostingList(const uint8_t *restrict
    list, int count, uint32_t *restrict
    result) {
  int fileid = ~0;
  while (count--) {
    fileid += uvarint(&list);
    *(result++) = fileid;
  }
}

func myPostingList(data []byte, count int)
    []uint32 {
  result := make([]uint32, count)
  C.cPostingList((*C.uint8_t)(&data[0]),
      C.int(count),
      (*C.uint32_t)(&result[0]))
  return result
}

x := myPostingList(r.d, r.max())
```

[11] See http://golang.org/src/pkg/encoding/binary/varint.go, licensed under the BSD license.

Both code excerpts were shortened for brevity. Support for `restrict` lists and checks for index consistency have been removed.

The C implementation makes heavy use of compiler hints (such as the `hot` attribute, the `restrict` keyword and marking variables as `const`) to enable compiler optimizations. It uses 32-bit integers and is hand-unrolled. Instead of repeatedly calling `r.next()` like the Go code does, the C function `cPostingList` is called once and all calls of the `uvarint` function within `cPostingList` are inlined by the compiler.

To compare the Go and C versions in a fair manner, figure 3.7 not only shows the Go code from listing 3.1 above, but also an optimized version which closely resembles the C code from listing 3.2 (that is, it also uses an inlined, hand-unrolled, 32-bit version of `Uvarint`). This optimized Go version has been benchmarked compiled with 6g (the x86-64 version of the gc compiler) and with `gccgo`[12]. Since `gccgo` uses GCC[13], the resulting machine code is expected to run faster than the machine code of the not-yet highly optimized 6g.

As you can see in figure 3.7, optimizing the Go version yields a speed-up of $\approx 3\times$. Using `gccgo` to compile it yields another $\approx 2\times$ speed-up, but only for large posting lists, interestingly. The optimized C implementation achieves yet another $\approx 2\times$ speed-up, being an order of magnitude faster than the original Go version for large posting lists.

These results confirm that it is worthwhile to re-implement the posting list lookup in C.

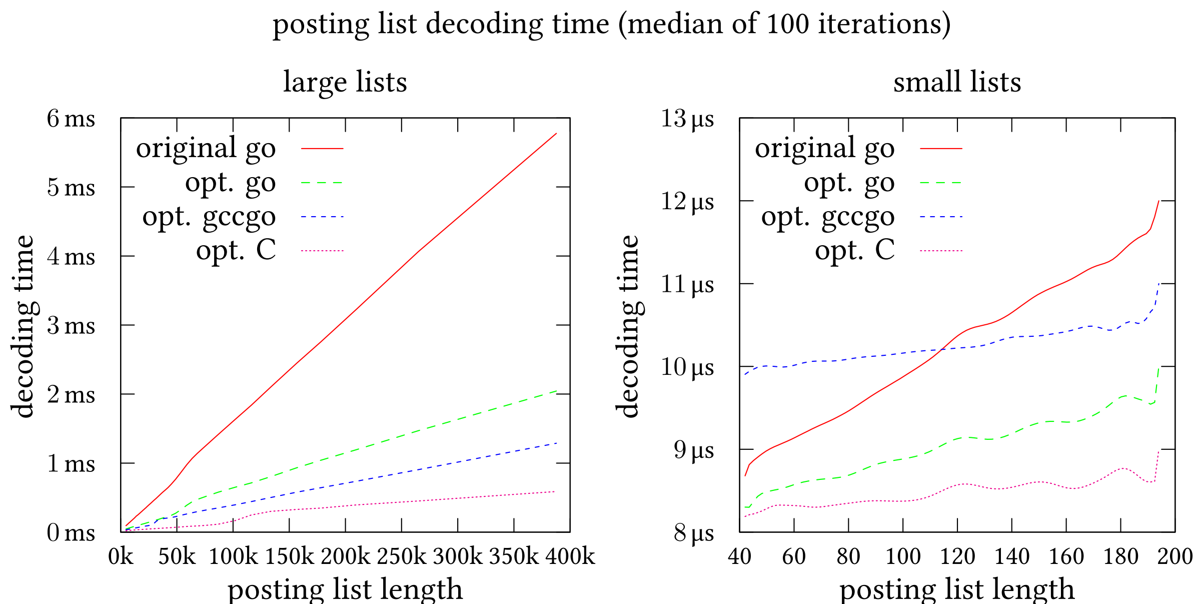

Figure 3.7: The optimized version written in C is faster for both small and large posting lists. At sufficiently large posting lists, the C version is one order of magnitude faster.

[12] using `go build -compiler gccgo -gccgoflags '-O3 -s'`
[13] The GNU Compiler Collection, a very mature collection of compilers with lots of optimizations

3.8.5 Posting list encoding/decoding algorithm

Jeff Dean (Google) presented an optimized version of the `varint` algorithm in 2009, called "Group Varint Encoding"[4] (`group-varint` from here on). The idea is to avoid branch mispredictions by storing four continuation markers in one byte before the next four values. A continuation marker indicates the next byte belongs to the current value and is usually stored by setting the highest bit of the value to 1. Storing four continuation markers allows for more efficient decoding because the CPU can read one byte and then decode four integers without any branches.

While this sounds like a reasonable optimization to make and the findings of D. Lemire and Leonid Boytsov[9] seem to confirm the optimization, a test with the Debian Code Search posting lists reveals that `group-varint` is not faster than `varint`, see figure 3.8. This is most likely because in our data set, the fast-path is taken most of the time: one byte represents the entire value and there is no need for bit shifting and multiple loop iterations. Therefore, as the fast path takes the same branch over and over again, DCS's decoding does not suffer from branch mispredictions.

Due to this benchmark, the posting list encoding algorithm was not changed from the original `varint`. It would be an interesting area of further research to vary the index shard size, thus generating different posting list deltas, and benchmark several different algorithms, such as Lemire's FastPFOR or others mentioned in the same paper[9]. Unfortunately, Lemire's scarcely documented source code is written in C++, which cannot be used with Go at the moment, and no C implementation is available.

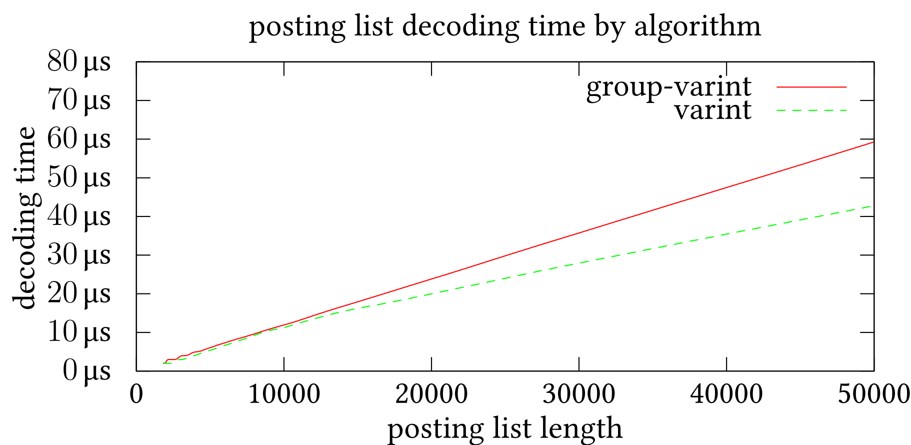

Figure 3.8: The simpler `varint` algorithm is as fast as `group-varint` for short posting lists and even faster than `group-varint` for longer posting lists.

3.8.6 Posting list query optimization

> "The query optimizer is the component of a database management system [DBMS] that attempts to determine the most efficient way to execute a query." [25]

Just like a DBMS, our trigram lookup benefits from query optimization. To understand the benefits which a very simplistic query optimizer brings (for *AND* queries only), it is helpful to work with an example. The user input of this example is the query `XCreateWindow`, which will be translated into the 11 trigrams $T =$ [`Cre`, `Win`, `XCr`, `ate`, `dow`, `eWi`, `eat`, `ind`, `ndo`, `rea`, `teW`] (not sorted, directly as read from the index). See section 3.8 (page 15) for more information about the trigram index.

The original code search tools execute *AND* queries in the order in which the trigrams are produced, so they would first seek to the posting list P_1 for $T_1 =$ `Cre`, decode it, then seek to the posting list P_2 (`Win`), intersect it with the previous posting list, and so on.

To reduce the amount of memory which is used for intersecting the lists, it is beneficial to first sort the trigrams ascendingly by the length of their posting lists. In this particular example, this reduces memory usage from 315 KiB to 3 KiB.

The final result R of an *AND* query is the intersection of all posting lists:

$$R = P_1 \cap P_2 \cap \cdots \cap P_{11}$$

Since $R \subset P_1$, DCS could use P_1 instead of R and match the regular expression in all files of P_1. The obvious problem is that P_1 contains more entries than R, that is, it contains some false positives: files which do not contain the search term, but still need to be searched. The $\Delta_{P_{i-1}}$ column of table 3.1 shows how many false positives are ruled out in each intersection.

trigram	$\#P_i$	i	$\#\cap$	$\Delta_{P_{i-1}}$	Δ_R	decoding time
`Xcr`	763	1	763		503	12 µs
`teW`	5732	2	266	497	6	38 µs
`eWi`	15 568	3	263	3	3	58 µs
`Win`	46 968	4	261	2	1	283 µs
`Cre`	78 955	5	260	1	0	209 µs
`dow`	97 453	6	260	0	0	344 µs
`ndo`	107 002	7	260	0	0	249 µs
`eat`	192 107	8	260	0	0	573 µs
`ind`	234 540	9	260	0	0	513 µs
`rea`	299 415	10	260	0	0	813 µs
`ate`	419 943	11	260	0	0	896 µs

Table 3.1: *AND* query execution for `XCreateWindow`. After decoding two of 11 posting lists, the difference to result R is very small, after reading 5 of 11 posting lists, there is no difference anymore. The decoding time for posting lists 6 to 11 is wasted time.

Looking at table 3.1, it is obvious that decoding could stop after reading the posting list of trigram $T_3 =$ `eWi`, but the program cannot know that since it doesn't know the final result R.

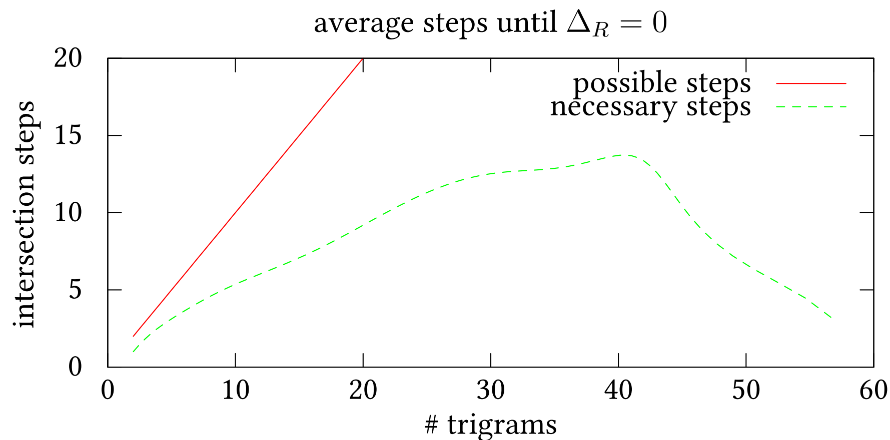

Figure 3.9: It takes only a fraction ($\approx \frac{1}{3}$) of intersection steps to get to the final result R. 1500 random function and variable names have been tested.

Instead, DCS can only use some kind of heuristic to decide when decoding can stop because the amount of false positives will not be reduced significantly by reading more posting lists.

Figure 3.9 confirms the hunch: It is (on average) not necessary to perform all intersections to get to the final result R, or very close.

A heuristic which yields a low number of false positives but still saves a considerable number of steps (and thus time) is:

Stop processing if $\Delta_{P_{i-1}} < 10$ and $i > 0.70 \times n$ (70 % of the posting lists have been decoded). As figure 3.10 shows, the amount of false positives does not exceed one or two files on average, while the total speed-up for executing the *AND* query is $\approx 2\times$.

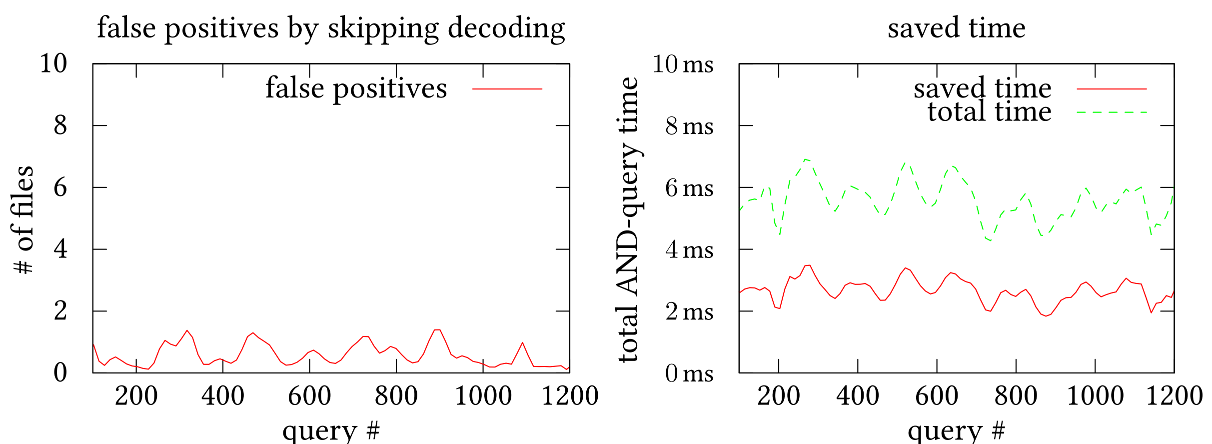

Figure 3.10: With the heuristic explained above, the amount of false positives does not exceed two files on average; the total speed-up is $\approx 2\times$.

3.9 Updating the index

The Debian archive changes all the time. New package versions get uploaded, some of them containing new releases of the software itself and some of them changing the packaging or adding patches to fix bugs.

To make Debian Code Search pick up these changes, the source mirror has to be synchronized and unpacked and then the DCS index has to be updated.

These updates should be performed at least once per week. Due to the indexing being very time consuming (≈ 6 h), performing updates once a day puts more load on the server without adding value. Every three days seems like a good compromise.

The full procedure to update the index goes as follows. First, two temporary folders are created: one will hold the new files, the other will hold the old files once the new files are moved to their destination. Then, the source mirror is updated:

```
mkdir /dcs/{NEW,OLD}
debmirror -a none --source -s main -r /debian /dcs/source-mirror
```

Now, the tool `dcs-unpack` takes the source mirror and the old `unpacked` folder to create the `unpacked-new` folder, which contains an unpacked version of the source mirror. All files that are shared between `unpacked` and `unpacked-new` are hardlinked[14] to save disk space and unpack time:

```
dcs-unpack                          \
  -mirrorPath=/dcs/source-mirror/   \
  -oldUnpackPath=/dcs/unpacked/     \
  -newUnpackPath=/dcs/unpacked-new/
```

Afterwards, the `unpacked-new` folder is indexed into files stored in `NEW`:

```
dcsindex                            \
  -mirrorPath=/dcs/NEW/             \
  -unpackedPath=/dcs/unpacked-new/  \
  -shards=6
```

Finally, the old and new files are swapped and all index-backend processes are restarted:

```
mv /dcs/index.*.idx     /dcs/OLD/
mv /dcs/NEW/index.*.idx /dcs/
mv /dcs/unpacked        /dcs/OLD/unpacked
mv /dcs/unpacked-new    /dcs/unpacked
for i in $(seq 0 5); do
    systemctl restart dcs-index-backend@$i.service
done
```

After verifying that no human mistake was made by confirming that Debian Code Search still delivers results, the `OLD` and `NEW` folders can be deleted.

[14] Hard links make a file available under more names, see http://en.wikipedia.org/wiki/Hard_link

3.10 Logging and monitoring

Web server logfiles are usually plain text files, containing one line for each request processed by the web server.

It is important to start with good logging infrastructure from the very beginning because logfiles can highlight potential problems with the web application, like requests resulting in an HTTP Error 500 (Internal Server Error) or requests which are resource-intensive and therefore cause latency spikes for subsequent requests.

3.10.1 Logging

Since in Debian Code Search's architecture every request passes through NGINX, the HTTP frontend webserver, this is the place where logging makes most sense so that the administrator can get a good overview of the whole system.

HTTP requests are either served directly by NGINX (static assets) or passed on to one of the backends. Requests for static assets are logged into a standard HTTP access logfile, as used by default in almost all webservers. This logfile will tell us how many people or computer programs such as web search crawlers accessed Debian Code Search, but not how many of them actually searched for something.

Requests for the backends are logged to a separate file. This logfile contains the timestamp, request URL and the response code, but also which backend handled the request and how long each processing step of the backend took.

```
log_format upstream '$remote_addr - - [$time_local] "$request" '
    '$status 1 upstream [$upstream_addr] '
    '[$upstream_response_time]=response request $request_time '
    't0 $upstream_http_dcs_t0 '
    # similar entries omitted for brevity

location = /search {
    access_log /var/log/nginx/dcs-upstream.log upstream;
    proxy_pass http://dcsweb/search;
}
```

A resulting logfile entry looks like this:

```
87.198.192.202 - - [14/Oct/2012:14:38:40 +0200] "GET
   /search?q=entry%5C.S HTTP/1.1" 200 1 upstream
   [188.111.72.14:28080] [0.512]=response request 0.512 t0
   237.24ms t1 106.08ms t2 4.42ms t3 49.57ms numfiles 4538
   numresults 40
```

The slightly peculiar looking format `[0.512]=response` is necessary to be able to parse the logfile with POSIX regular expressions later on because NGINX will put comma-separated values in there if it tries multiple backends. When one backend fails to return a reply with a

HTTP 200 status code and NGINX falls back to another backend, the entry looks like this:

```
2001:4d88:100e:23:3a60:77ff:feab:d3ea - - [13/Oct/2012:17:13:18
    +0200] "GET /search?q=AnyEvent%3A%3AXMPP HTTP/1.1" 200 1
    upstream [188.111.72.14:28080, 127.0.0.1:28080] [0.026,
    0.243]=response request 0.269 t0 30.46ms t1 12.42ms t2 0.02ms
    t3 199.31ms numfiles 4 numresults 9
```

The total amount of time necessary to handle this request is 0.269 s. The numbers after t0, t1, and so on are custom HTTP header values which are filled in from the backend's HTTP response. All timing information which is measured is included in this form, thus automatically ending up in the same logfile as the HTTP requests without any need to cross-correlate two logfiles from different machines or even implement logging in the backend at all.

3.10.2 Monitoring

In this case, monitoring means looking at graphs which are generated periodically or on-demand based on the logging data described above.

The goal is to detect and analyze bugs, performance problems or abuse of the service. For example, huge spikes in search result latency are very easy to spot even with a cursory glance. Based on the graph, the corresponding logfiles can then be consulted to find the query which causes the problem.

To generate graphs from the logfiles, the program `collectd` was used because it is a light-weight tool to collect all sorts of information and the author is familiar with it already. `collectd` uses round-robin databases via the RRDtool library to store data over a fixed period of time, typically weeks or months. Graphs are generated with RRDtool's graphing tool.

To parse the logfiles, `collectd`'s "tail" plugin was used. The tail plugin follows a (log)file and parses each line according to a configurable specification. Here is the `collectd` configuration excerpt for the tail plugin:

```
<Plugin "tail">
    <File "/var/log/nginx/dcs-upstream.log">
        Instance "dcs-upstream"
        <Match>
            Regex    "([0-9.]*)\]=response"
            DSType   GaugeAverage
            Type     delay
            Instance "response"
        </Match>
        # similar entries omitted for brevity
    </File>
</Plugin>
```

The tail plugin uses POSIX regular expressions[26] which do not allow for non-greedy matching. Therefore, the peculiar log format mentioned above is necessary.

3.11 Caching

Modern web applications often use caching to trade memory for CPU time or network latency. As an example, Wikipedia's featured article of the day is requested a lot, so each of Wikipedia's cache servers will have a copy in memory to avoid querying the Wikipedia database servers and rendering the page. This section discusses which kinds of caching are applicable and useful in Debian Code Search.

The efficiency of a cache is measured with the cache hit ratio. If, over the time period of one day, the cache hit ratio is 2 % and there were 100 requests, that means 2 search queries were answered from the cache while 98 search queries were not in the cache. In this section, not only the cache hit ratio is considered, but also the amount of memory spent on the cache. For example, if the cache hit ratio could be increased from 2 % using 10 MiB of memory to 4 % using 16 GiB of memory, that would not be worthwhile.

3.11.1 Implicit caching (page cache)

Apart from explicitly configured caching, the Linux kernel also provides a page cache [24]. Whenever a block of data is read from the hard disk, it ends up in the page cache, and subsequent requests for the same block can be answered from memory.

For DCS, a good starting point after booting the server is to load the entire trigram index into the page cache[15] as trigram index lookups are necessary for all queries, so all queries will profit from that.

Linux uses a Least-Recently-Used (LRU) strategy for its page cache [15]. For DCS, that means often-queried trigrams will stay in the page cache while other file contents (source code that needs to be displayed/searched) will evict rarely-used pieces of the trigram index.

The behavior of the page cache is the reason why DCS does not attempt to lock the entire trigram index in memory: On machines with little RAM (e.g. 8 GiB), this would lead to nearly no RAM available for non-trigram caching, while on machines with a lot of RAM, it is unnecessary.

While the page cache is a necessary feature to make modern computing as fast as it is, the obvious problem is that while the data is present, all algorithms still need to process it. That is, the search engine still needs to decode the posting lists, search through all files, rank the results and format the page. Another downside of the page cache is that Linux does not provide an interface to measure the page cache hit ratio.

3.11.2 Explicit caching

Debian Code Search consists of multiple processes which use HTTP to communicate with each other. A welcome side effect is that sophisticated caches can very easily be added before every such process. That is, for caching whole search result pages, the frontend NGINX web

[15] This can be done by simply reading all files:

```
for f in /dcs-ssd/index.*.idx; do dd if=$i of=/dev/null bs=5M; done
```

server can simply cache requests to `/search`. To cache index query results, a new cache could be added in front of requests to `index-backend` processes.

When Debian Code Search was launched on 2012-11-06, no explicit caching was configured. Three sample queries were included in the launch announcement[16]: `"XCreateWindow"`, `"workaround package:linux"` and `"AnyEvent::I3 filetype:perl"`. These queries were selected to demonstrate certain features, but also because they are served quickly.

By monitoring the web server log files after sending the announcement, it quickly became clear that explicit caching of the `/search` URL would be helpful: people shared interesting search queries, such as `http://codesearch.debian.net/search?q=fuck`[17] or `http://codesearch.debian.net/search?q=The+Software+shall+be+used+for+Good%2C+not+Evil`[18]. As table 3.2 shows, these shared queries are the most popular queries. At least some of them also profited from explicit caching, e.g. “The Software shall be used for Good, not Evil” with a cache ratio of 80.2 %.

search term	hits	cached	cache ratio
The Software shall be used for Good, not Evil	2066	1657	80.2 %
fuck	1247	423	33.9 %
workaround package:linux	683	128	18.7 %
XCreateWindow	528	71	13.4 %
idiot	265	8	3.0 %
AnyEvent::I3 filetype:perl	255	35	13.7 %
shit	130	10	7.7 %
FIXME	116	30	25.9 %
B16B00B5	105	45	42.9 %
(babefee1|B16B00B5|0B00B135|deadbeef)	94	38	40.4 %

Table 3.2: Top 20 search queries and their cache hit ratio from 2012-11-07 to 2012-11-14. The cache was 500 MiB in size and entries expire after 15 minutes; see listing 3.3 (page 29).

The varying cache hit ratios are caused by the different time spans in which the queries are popular. The top query was so popular that it always stayed in the cache, while other queries did not stay in the cache for very long.

[16] `http://lists.debian.org/debian-devel-announce/2012/11/msg00001.html`

[17] `https://twitter.com/antanst/status/266095288266153984` and `http://www.reddit.com/r/programming/comments/12sni3/debian_code_search/c6xz82h`

[18] Referred to in `http://apebox.org/wordpress/rants/456/`, a blog post about a harmful software license

Table 3.2 suggests that even higher cache hit ratios can be achieved by increasing the expiration time, which is currently configured to 15 minutes. That is, if a cached search result page is 15 minutes old, the query needs to be re-executed. The expiration time is a trade-off: if it is too high, users will be served old content for longer than necessary. If it is too low, the cache does not lead to decreased server load. Of course, in Debian Code Search, content does not get refreshed that often because the index rebuilds are computationally expensive, so the expiration time could be much higher. Typically, the index will be updated every three days.

For the moment, the caching configuration is left as-is and can be improved once traffic patterns stabilize.

Listing 3.3: nginx cache configuration

```
proxy_cache_path /var/cache/nginx/cache levels=1:2
    keys_zone=main:50m
    max_size=500m inactive=15m;

proxy_temp_path /var/cache/nginx/tmp;

location = /search {
    # omitted for brevity ...
    set $cache_key $scheme$host$uri$is_args$args;
    proxy_cache main;
    proxy_cache_key $cache_key;
    proxy_cache_valid 15m;

    proxy_pass http://dcsweb/search;
}
```

4 Search result quality

In order for Debian Code Search to become a useful tool for many developers, it is important that the quality of search results is high. While humans can intuitively tell a good search engine from a bad one after using it for some time, this chapter defines a metric with which the quality of a code search engine can be measured. It also covers the ranking factors used by DCS, evaluates how well they work, and describes its current limitations.

4.1 Metric definition

Search engines generally present a list of results, ordered by their ranking so that the supposedly best result is at the top. If the user is satisfied with the result that is presented first, the search engine did the best possible job.

To properly define a metric, some terms need to be clarified first:

u is the user who sends the search query.

q is the search query, e.g. `"XCreateWindow"`.

r is an individual search result. There is typically more than one result.

$|r|$ is defined as the position of r within the list of all results. The first result has $|r_{\text{first}}| = 0$, the second result has $|r_{\text{second}}| = 1$ and so on.

Our penalty for how good the search results for a given query and user are is defined as

$$P(u, q) = |r_{\text{satisfactory}}|$$

This penalty P depends on the user and on the query. The earlier the search result with which the user is satisfied shows up in the search results, the lower the penalty is. Therefore, if the user is presented with a result that immediately satisfies her, then $P(u, q) = 0$. If the user has to skip 40 results before she finds one with which she is satisfied, then $P(u, q) = 40$.

The best search engine is the search engine for which

$$\text{median}_{q \in Q, u \in U} \left(P(u, q) \right)$$

is minimal. That is, the median of all penalties for a set of queries Q and a set of users U is minimal. This median defines our metric.

4.2 Ranking

When a search engine gets a search query, it is not feasible to compute all matches. Doing so would take a long time: for `XCreateWindow`, computing the matches within all files which are returned by the index query takes about 20 s to complete.

Given that typical web search engine queries are answered in the fraction of a second, this is clearly unacceptable. Therefore, the ranking is not only used for presenting the matches in the best order to the user, but also before even computing any matches this reduces the amount of data which needs to be processed. This kind of ranking is called "pre-ranking" from now on. Since results with low ranking are unlikely to be requested by the user, they are not even computed in the first place.

Each of the ranking factors which are presented in the subsequent chapters aims to assign a single number to each package or source code file.

4.2.1 Ranking factors which can be pre-computed per-package

Popularity contest installations See section 4.2.4 (page 33).

Number of reverse dependencies See section 4.2.5 (page 34).

Number of bugreports While a relation between bug frequency and package popularity may exist [3], the two previous ranking factors also indicate popularity. Aside from popularity, it is not intuitively clear whether the number of bug reports should be weighted in favor of or against any particular package. The number of bugreports has therefore not been considered as part of the ranking within this work.

File modification time See section 4.2.6 (page 35).

Version number cleanliness A very low version number like `0.1` might be an indication for an immature FOSS project. Therefore, one might intuitively think that it would be a good idea to use version numbers as a ranking factor: The higher the version, the better.

However, the perception of what a version number means differs from project to project. Some projects use the release date as version number[1] while others have version numbers that asymptotically approach π[2].

Due to the wildly varying meaning of version numbers, they have not been used for ranking within Debian Code Search.

4.2.2 Ranking factors which depend on the query

Query contained in file path When the query string is contained in the file path, the file should get a higher ranking. This is very simple to implement and computationally cheap.

[1] e.g. git-annex, TeXlive, SLIME, axiom, batctl, caspar, ...
[2] TeX approaches π, Metafont approaches e, there might be others

Query is a library symbol When the query matches one of the library's exported function symbols, the package should get a much higher ranking. This is computationally cheap but works only for C-like programming languages and libraries.

Query is in the package's description The package description is a concise summary of what the package does. Library packages often list their main areas or available functions, so it is likely that the user's search query matches the description, especially if she is searching for an algorithm name.

4.2.3 Ranking factors which depend on the actual results

Indentation level of the result The vast majority of programming languages represents different scopes by indentation[3]. Scopes represent a hierarchy: higher scopes mean the symbol is more important (e.g. a function definition at the top-level scope in C is more important than a variable declaration in a lower-level scope). To quickly and language-independently decide whether one line is more important than another, the amount of whitespace in front of it is counted.

Word boundary match To prefer more exact results, the regular expression features `\b` can be used (match on word boundaries). As an example, `\bXCreateWindow\b` will match `Window XCreateWindow(` but not `register XCreateWindowEvent *ev =`. Since the position of the matching portion of text is known, it is used to further prefer earlier matches over later matches.

Since programming languages are structured from left-to-right, earlier matches (such as function definitions) are better than later matches (such as function parameter types).

[3] Of the most popular languages in Debian, the first 10 (C, Perl, C++, Python, Java, Ruby, ocaml, LISP, Shell, PHP, see section 4.6 (page 44)) use indentation.

4.2.4 Popularity contest

The Debian "popcon" (short for Package Popularity Contest) project was registered in 2003 to generate statistics regarding which packages are in use in Debian[13]. Users of Debian who have internet access can opt-in to this project by installing the package `popularity-contest`. The Debian installer also offers to install this package when installing a new machine. The `popularity-contest` package contains a script which is run once a week and uploads statistics about the installed packages to Debian servers.

In this work, only one aspect of the popcon data has been used: the package installation count. This figure describes how many different machines have installed the package in question. Each machine is identified by a 128-bit UUID[14].

From here on, $\text{inst}(x)$ is defined as the number of installations of package x.

Since popcon is an opt-in mechanism, the numbers are to be taken with a grain of salt. The amount of submissions as of 2012-07-13 is about $120\,000$[4]. This is a substantial number, but not necessarily representative.

While there are similar databases in other Linux distributions, most notably Ubuntu, which also uses popcon, these have not been used in this work for various reasons: other data is not as easily available as Debian popcon data is. The latter has been taken from UDD, the Ultimate Debian Database[12]. Furthermore, when using data from other Linux distributions, package availability[5] and package names often differ.

[4] `http://popcon.debian.org/stat/sub-***.png` [sic!]

[5] Software installed without using the package system is not tracked by popcon or any other such database that I know of.

4.2.5 Reverse dependencies

Each Debian (binary) package includes dependency information in its `Depends` field. Assuming package A and B depend on package x, the reverse dependencies rdep(x) are defined as the list $[A, B]$. Intuitively, the longer the list of reverse dependencies of any package is, the more important the package is.

Since both inst(x) and rdep(x) indicate the importance of package x, it has to be ensured that they are not simply interchangeable. Table 4.1 shows that there is no obvious correlation between the inst(x) and rdep(x) rankings. Therefore, the number of reverse dependencies was used to calculate the final ranking.

The formula used to transform the number of reverse dependencies into a ranking factor is:

$$R_{\text{rdep}}(C) = 1 - \frac{1}{|\text{rdep}(x)|}$$

rdep-rank	package	rdep(x)	inst-rank	Δ
1	eglibc	17 854	47	46
2	gcc-4.7	10 224	25	23
3	qt4-x11	4806	582	579
4	perl	3911	8	4
5	glib2.0	3161	132	127
6	python-defaults	3128	83	77
7	kde4libs	2297	929	922
8	zlib	2109	10	2
9	libx11	1957	86	77
10	cairo	1806	202	192
11	mono	1589	705	694
12	pango1.0	1550	203	191
13	gtk+2.0	1489	231	218
14	gdk-pixbuf	1437	922	908
15	freetype	1336	102	87
16	atk1.0	1326	230	214
17	libxml2	1242	84	67
18	fontconfig	1196	151	133
19	ncurses	1128	2	17
20	python2.6	1127	180	160
$\overline{x}_{\text{arithm}}$				236.9

Table 4.1: Top 20 packages sorted by rdep with their rdep-rank and inst-rank. As can be seen, there is no obvious correlation between the rdep and inst metric.

4.2.6 Modification time of source code files

In general, newer source code is more interesting than old source code. Either the code was modified to fix a bug, in which case the user is interested in obtaining the latest version of the code with as many bugfixes as possible. Or the code was written more recently, meaning that it is more relevant to the user because it solves a current problem.

In Debian and Linux in general, there are three different timestamps for each file: The access time (`atime`), the change time (`ctime`) and the modification time (`mtime`). The difference between change time and modification time is that changes to the inode (permissions or owner for example) are recorded in the former, while the latter only gets set when the actual file contents are modified.

One concern with using the mtimes of individual files within a source package is that — due to packaging mechanisms — the mtimes might be set to the time at which the package was created, rendering them useless for our purpose. To determine whether this effect has a measurable impact, for each source package the following analysis was run: Find each file within the source package which ends in `.c` or `.h` (a quick heuristic for recognizing a portion of our source code), then calculate the difference between each file's mtime and the package's mtime. For each source package, the average difference was stored.

This analysis can be expressed with the following formula (with f being the list of files for each source package s):

$$d_s = \frac{\sum_f |\text{mtime}(f) - \text{mtime}(s)|}{|f|}$$

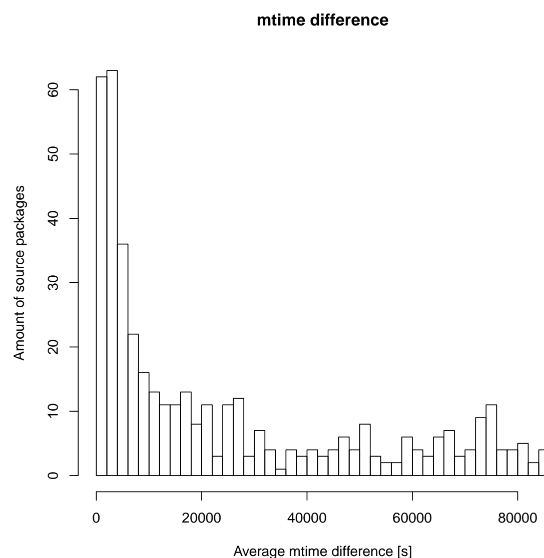

Figure 4.1: Average mtime difference between the source package and its `.c` or `.h` files

A little over 60 packages ($0.6\,\%$ of all analyzed source packages) have an average mtime difference near zero. In these cases, mtime is not a good ranking factor, but 60 packages is a negligible number.

4.3 Sample search queries and expected results

The sample queries and expected results listed in this section have been used to evaluate Debian Code Search. An expected result is a search result which the user would list as the best result after carefully examining all possible results for the given query. Of course, the expected results listed here are only a portion of all possible results for each search term.

XCreateWindow This function is a low-level function to create an X11 window. Programs which use toolkits such as GTK or Qt will never call it directly, but every low-level X11 program with a GUI will call it.

Expected results:
```
libx11_1.5.0-1/include/X11/Xlib.h:1644
libx11_1.5.0-1/src/Window.c:100
```

PNG load A generic query which you could type if you wanted to display a PNG image and didn't know how to approach the question at all. There are no restrictions on programming language in this query.

Expected results:
```
sdl-image1.2_1.2.12-2/IMG_png.c:35
pekwm_0.1.14-2/src/PImageLoaderPng.cc:64
stella_3.7.2-1/src/common/PNGLibrary.cxx:56
```

strftime is a function provided by the standard C library which stores the specified time in a string formatted according to the given format. You might search for the implementation if you are wondering about a small detail or search for an example if you are not satisfied with the available documentation.

Expected results:
```
eglibc_2.13-35/time/strftime.c:25
eglibc_2.13-35/time/time.h:199
eglibc_2.13-35/time/strftime_l.c:498
```

(?i)bloomfilter A bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set[20]. This search query represents the search for a data structure to find a library, an implementation or a usage example.

Expected results:
```
webkit_1.8.1-3.1/Source/JavaScriptCore/wtf/BloomFilter.h:38
chromium-browser_20.0.1132.57~r145807-1/src/chrome/browser/ \
    safe_browsing/bloom_filter.cc:155
git-annex_3.20120721/Command/Unused.hs:204
eiskaltdcpp_2.2.6-4/dcpp/BloomFilter.h:27
gnunet_0.9.3-2/src/util/container_bloomfilter.c:107
python-bloomfilter_1.0.3-2/pybloom/pybloom.py:87
```

smartmontools is the name of a package, not a command, which displays the S.M.A.R.T.[6]

[6] "S.M.A.R.T. […] is a monitoring system for computer hard disk drives to detect and report on various indicators of reliability, in the hope of anticipating failures."[27]

data of hard disks.

Expected results:
`smartmontools_5.42+svn3561-3/smartctl.cpp:4`

ifconfig is the name of an old tool to configure network interfaces. This search query has been added to represent searching for a tool which gets invoked by Debian maintainer scripts.

Expected results:
`net-tools_1.60-24.1/ifconfig.c:177`
`busybox_1.20.0-6/networking/ifconfig.c:4`

_NET_WM_PID is the name of an X11 atom which is used to store process IDs on an X11 window to allow correlating an X11 window with a running process. It is very similar to XCreateWindow above, but there is no definition, so the expected search results are only usage examples.

Expected results:
`libsdl1.2_1.2.15-5/src/video/x11/SDL_x11video.c:429`
`qt4-x11_4.8.2-1/src/gui/kernel/qwidget_x11.cpp:846`
`mplayer_1.0 rc4.dfsg1+svn34540-1/libvo/x11_common.c:742`

pthread_mutexattr_setpshared is the name of a function which will mark a pthread mutex as shared between different processes. One might search for this query to figure out if this functionality is in widespread use and if it is thus considered safe to use it in new code.

Expected results:
`apache2_2.2.22-9/test/time-sem.c:321`

conv.*pam_message is a regular expression which should match the signature of a PAM conversation callback. The author searched for this to find an example which clarifies how to allocate memory for the responses which are used in this callback.

Expected results:
`pam_1.1.3-7.1/xtests/tst-pam_unix4.c:53`
`openssh_6.0p1-2/auth-pam.c:555`

dh_installinit.*only is a regular expression which should match the utility dh_installinit within the Debian packaging of some packages. It was used by the author in a discussion about systemd support in dh_installinit when a question about how to deal with packages shipping their own systemd service files came up (they have to call `dh_installinit --onlyscripts` manually).

Expected results:
`busybox_1.20.0-6/debian/rules:170`

4.4 Result quality of Debian Code Search

This section examines the result quality of Debian Code Search for the sample queries defined in section 4.3. After analyzing how well each ranking factor performs, a weighted combination of all ranking factors is constructed which is used in Debian Code Search as the default ranking.

For every ranking factor described in section 4.2 (page 31), each of the sample queries listed in section 4.3 (page 36) was run with a special flag causing DCS to list all possible results, not just the first 40. From the full results, the penalty to the expected results was extracted (penalty as defined in section 4.1, page 30). The results are listed in table 4.2.

In case there was more than one expected result for a query, the median of the penalties was used.

query	# res	none	rdep	inst	path	pkg	indent	boundary
XCreateWindow	3145	1674	57	10	1689	1673	491	1551
PNG load	73	10	11	1	11	10	43	12
strftime	40 048	31 320	50	1455	2952	31 336	8819	11 040
(?i)bloomfilter	1852	920	213	698	424	920	338	357
smartmontools	868	79	174	141	91	75	389	563
ifconfig	9164	6924	1130	90	775	6985	5689	3918
systemctl	553	145	112	39	2	147	59	468
_NET_WM_PID	286	141	50	31	157	153	173	164
pthread…setpshared	348	330	45	46	321	324	294	129
conv.*pam_message	213	104	21	13	104	101	91	105
dh_installinit.*only	40	35	12	0	35	35	24	35

Table 4.2: Ranking penalties per search query. Lower values are better.

Since the number of results differs greatly, instead of using the absolute penalties of each ranking factor R it is helpful to work with percentages, defined as follows:

$$P_R = \left(1 - \frac{R}{\#\text{res}}\right)$$

Note that the percentage is substracted from 1 to reverse its meaning: for penalties, lower is better, but for percentages, higher is better. This allows a more intuitive interpretation of the results.

The difference caused by each ranking factor can be expressed by the comparison of each ranking's percentage with the percentage when using no ranking at all: $\Delta(R) = P_R - P_{\text{none}}$ (in percentage points). The results are listed in table 4.3.

query	# res	P_{none}	Δ_{rdep}	Δ_{inst}	Δ_{path}	Δ_{pkg}	Δ_{scope}	Δ_{line}
XCreateWindow	3145	.468	+.514	+.529	−.005	.000	+.376	+.039
PNG load	73	.863	−.014	+.123	−.014	.000	−.452	−.027
strftime	40 048	.218	+.781	+.746	+.708	.000	+.562	+.506
(?i)bloomfilter	1852	.503	+.382	+.120	+.268	.000	+.314	+.304
smartmontools	868	.909	−.109	−.071	−.014	+.005	−.357	−.558
ifconfig	9164	.244	+.632	+.746	+.671	−.007	+.135	+.328
systemctl	553	.738	+.060	+.192	+.259	−.004	+.156	−.584
_NET_WM_PID	286	.507	+.318	+.385	−.056	−.042	−.112	−.080
pthread...setpshared	348	.052	+.819	+.816	+.026	+.017	+.103	+.578
conv.*pam_message	213	.512	+.390	+.427	.000	+.014	+.061	−.005
dh_installinit.*only	40	.125	+.575	+.875	.000	.000	+.275	.000
$\overline{x}_{arithm}$			+.395	+.444	+.168	−.001	+.096	+.046

Table 4.3: Improvements of each ranking over the “none-ranking”. Higher is better.

It is noteworthy that the Δ-values for a query such as `smartmontools` are mostly negative since the P_{none} percentage already is quite good by chance (90 %).

By normalizing the numbers that specify how the different rankings perform in relation to each other, the final ranking $R_{weighted}$ can be defined, which is a weighted sum of all rankings considered above:

$$\begin{aligned} R_{weighted} = & 0.3427 \times R_{rdep} + \\ & 0.3840 \times R_{inst} + \\ & 0.1460 \times R_{path} + \\ & 0.0008 \times R_{pkg} + \\ & 0.0841 \times R_{scope} + \\ & 0.0429 \times R_{line} \end{aligned}$$

To figure out how the weighted ranking performs in comparison to the other rankings, $\min(P)$ is defined as the minimum (best) penalty of all penalties for P. Then, the penalty achieved with $R_{weighted}$ is compared against the minimum penalty and the penalty of using no ranking at all. The results are listed in table 4.4.

As expected, in almost all cases $R_{weighted}$ performs much better than using no ranking at all (except when P_{none} is already good by chance). In about 50 % of the cases, $R_{weighted}$ is as good as the single best ranking or better. In the other cases, it performs worse — adding about three to four pages of search results (with 40 results per page) which the user has to dismiss before finding the expected result.

query	# res	min	none	weighted	Δ_{min}	Δ_{none}
XCreateWindow	3144	10	1674	2	−8	−1672
PNG load	72	1	10	2	+1	−8
strftime	40 074	50	31 320	142	+92	−31 178
(?i)bloomfilter	1852	213	920	499	+286	−421
smartmontools	868	75	79	524	+449	+445
ifconfig	9176	90	6924	56	−34	−6868
systemctl	552	2	145	65	+63	−80
_NET_WM_PID	300	31	141	35	+4	−106
pthread...setpshared	339	45	330	42	−3	−288
conv.*pam_message	212	13	104	14	+1	−90
dh_installinit.*only	40	0	35	0	0	−35

Table 4.4: Improvements of the weighted ranking over the best single ranking for each query. Lower is better.

To verify that the improvements are not just the result of optimizing $R_{weighted}$ for the specific test data, a second set of queries, the testing set, has been examined in the same fashion as the other queries. The results are listed in table 4.5.

query	# res	min	none	weighted	Δ_{min}	Δ_{none}
XkbWriteXKBKeymap	18	0	14	0	0	−14
struct request_rec	35	0	33	0	0	−33
AnyEvent::I3	20	0	14	2	+2	−12
getaddrinfo	18 691	34	14 605	0	−34	−14 605
file-exists-p	7153	70	4470	150	+80	−4320

Table 4.5: Improvements of the weighted ranking over the best single ranking for each testing set query. Lower is better.

As can be observed in table 4.5, $R_{weighted}$ also works well for our testing set of queries.

4.5 Result latency of Debian Code Search

As explained in section 3.1 (“Architecture and user interface design principles”, page 6), low latency is an important goal for any interactive computer application. This section presents result latency measurements.

The ideal goal is to not exceed 100 ms for the entire timespan from the user pressing the “Enter” key on his keyboard to the page having been rendered in his/her web browser. A delay of 100 ms is less than humans can perceive, thus making Debian Code Search feel instant, without any delay at all[17].

Of course, this work only considers the parts of latency which are realistically influencable. As an example, network latency cannot be optimized for everybody in the world, except by placing many servers in geographically close locations for the majority of users, but this is out of scope for this project.

4.5.1 Trigram lookup latency

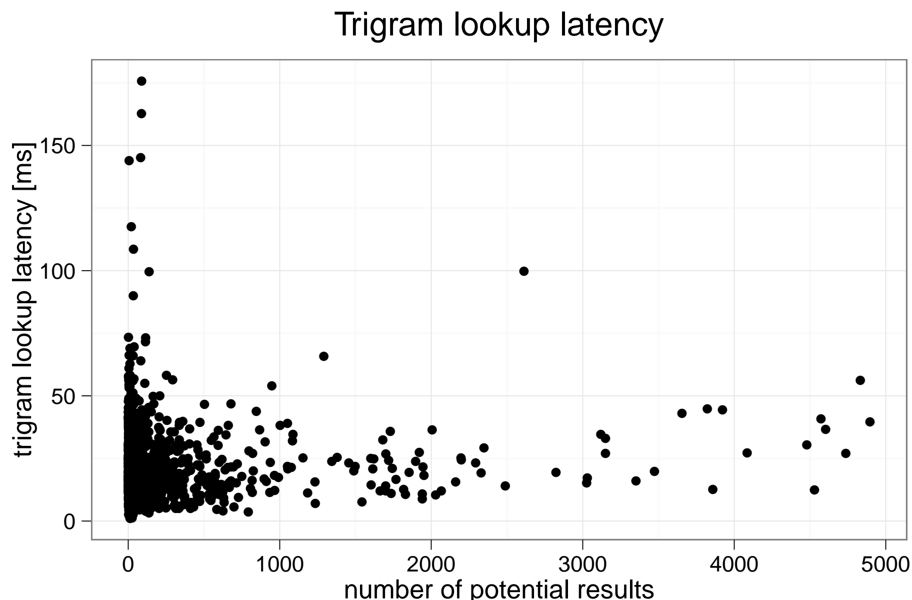

Figure 4.2: Trigram lookup latency distribution by number of results. Each point represents one out of 1500 randomly chosen function or variable names.

As you can see in figure 4.2, the lookup latency for most queries is lower than 50 ms, and only few queries exceed 100 ms. The number of potential results influences the latency somewhat, but not very strongly. This is in line with the results of section 3.8.4 (“Posting list decoding implementation”, page 19) and section 3.8.6 (“Posting list query optimization”, page 22), in which the trigram lookup step was optimized.

Figure 4.3 reveals that most queries actually have a trigram lookup latency between ≈0 ms and 30 ms, which leaves us with roughly 50 ms (plus network latency outside of our control)

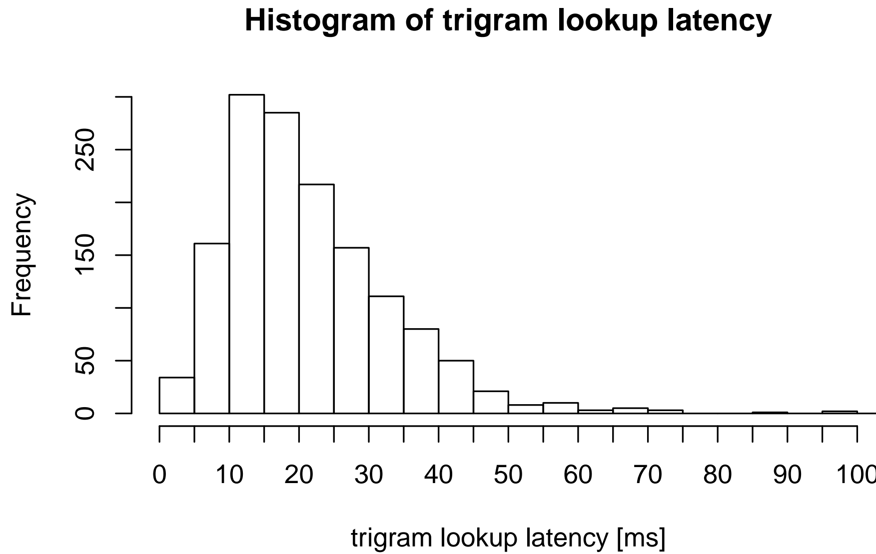

Figure 4.3: Histogram of trigram lookup latency

to stay within the goal of 100 ms.

4.5.2 Source matching latency

The source matching step is where the potentially matching files from the trigram lookup are actually searched/matched for the query string.

Due to the complex nature of the source code (a regular expression matcher), it is out of scope for this work to perform algorithmic optimizations or code optimizations on this step. This is unlike the trigram lookup, which was small and understandable enough to be re-implemented in optimized C. Therefore, the regular expression matching of the Codesearch tools has been used as-is.

Figure 4.4 (page 43) shows that the source matching latency spans a larger range (from 0 ms up to 200 ms) than the trigram lookup latency. Furthermore, it does not linearly correlate with the number of potential results (the amount of input files for source matching).

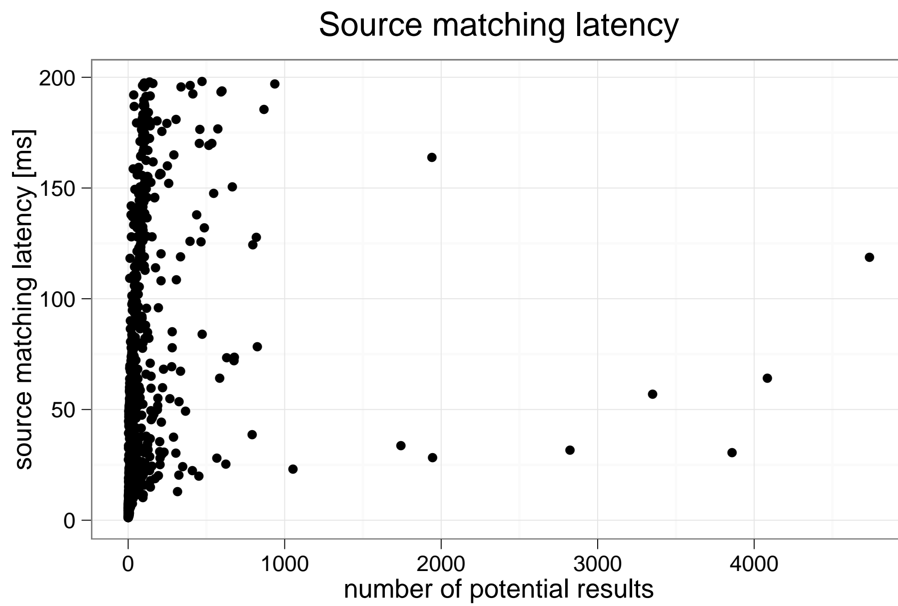

Figure 4.4: Source matching latency distribution by number of potential results (the number of files which are searched for matches)

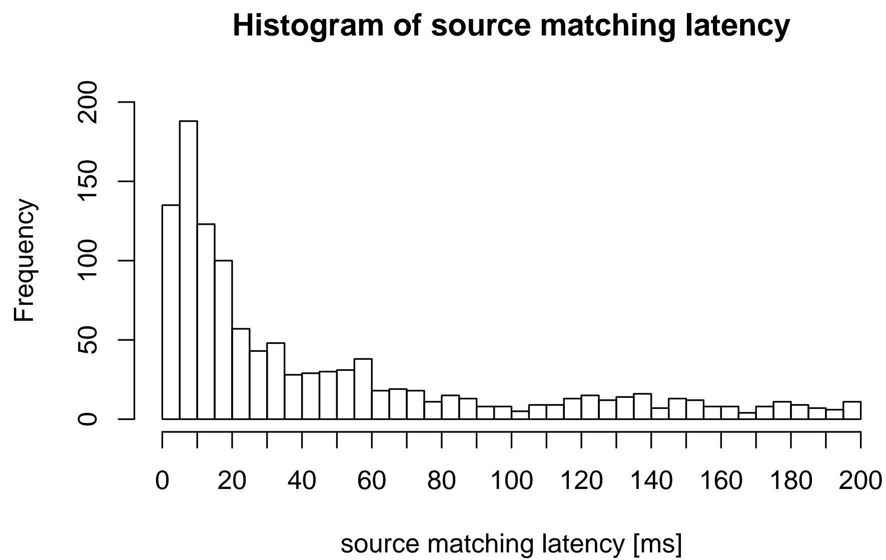

Figure 4.5: Histogram of source matching latency. Most source matching is done within 20 ms, but the tail is considerably longer than for trigram lookup latency.

4.6 Language bias

A consequence of choosing the Debian archive as corpus is that a lot of software is not included. Also, Debian Code Search clearly is biased in favor of certain languages.

To analyze the distribution of languages in the Debian archive, the debtags meta information[30] has been used. debtags uses faceted classification and allows us to count the number of packages tagged with the facet `implemented-in::perl` for example. The result of this analysis can be seen in table 4.6.

The whole ecosystem of Objective C makes for a striking example of language bias: Objective C is on position 14 in table 4.6 with only 62 packages being implemented in Objective C. Yet, Objective C ranks place 3 in the TIOBE index for August 2012[7]. The explanation for this observation is that Objective C is the native language for writing software in Apple's OS X and on Apple's iOS. It is therefore not surprising that it is a popular language as measured by TIOBE but not a popular language as per table 4.6.

Another observation is that the distribution of languages over all source code repositories hosted at github.com[8] does not match table 4.6 at all. The cause for this could be that programmers in different languages use source control management systems (SCMs) to a varying degree. However, it seems unlikely that this is the only reason. Instead, considering the fact that top github languages are JavaScript and Ruby, one can conclude that the rising amount of web applications is underrepresented in Debian and traditional desktop applications are underrepresented on github.

position	# pkg	language	position	# pkg	language
1.	4448	c	13.	77	c-sharp
2.	3300	perl	14.	62	objc
3.	1698	c++	15.	56	tcl
4.	1077	python	16.	48	fortran
5.	307	java	17.	43	ecmascript
6.	292	ruby	18.	42	vala
7.	207	ocaml	19.	42	scheme
8.	200	lisp	20.	36	lua
9.	199	shell	21.	19	ada
10.	156	php	22.	15	pike
11.	96	r	23.	3	ml
12.	93	haskell	24.	0	erlang

Table 4.6: Debian packages with implemented-in::language tag per language

[7] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
[8] `https://github.com/languages/`

4.7 Duplication in the archive

It is noteworthy that there is some duplication of soure code in the Debian archive. One example is the WebKit layout engine used by Apple Safari and Google Chrome, for example. WebKit code can be found in the WebKit package, in the Qt framework, in Chromium and in PhantomJS.

In general, such duplication is unfortunate because it makes maintenance of the projects harder, that is, bugs have to be fixed in more than one place. Then again, it sometimes reduces the potential for bugs, for example when Qt doesn't use whichever version of WebKit is installed on the system, but only uses its well-tested, bundled version.

From the perspective of Debian Code Search or any code search engine, duplication is unfortunate because users might get multiple search results containing the same source code for a single query (e.g. a user searches for "`(?i)bloomfilter`" and gets presented with `JavaScriptCore/wtf/BloomFilter.h` from qt4, webkit, etc.). This is undesirable. Let us assume the limit for source code results is fixed at, say, 10 results per page. When the same result is presented three times, the user effectively is presented with 8 results instead of 10, and he might get frustrated because he needs to realize that the presented results are identical.

Detecting duplicates (or very similar source code, e.g. a slightly different version of WebKit) can be done by applying metrics such as the Levenshtein distance[9]. The problem is not prevalent enough that it would be worthwhile (within this work) to design and apply such a technique on our entire corpus.

Within Debian, there is also a project to reduce the number of duplicates to avoid security issues from unapplied bugfixes to embedded copies of code[10].

[9] "[...] the Levenshtein distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is equal to the number of single-character edits required to change one word into the other." [22]

[10] `http://lists.debian.org/debian-devel/2012/07/msg00026.html`

4.8 Auto-generated source code

There are several open-source projects[11] which include automatically generated source code to a varying degree.

One example is XCB, the X11 C Bindings[12]. The project consists of an XML protocol description of the wire-level X11 protocol (`xcb-proto`) and a script to generate the X11 C Bindings themselves (`c_client.py`).

Where the very old `libX11` contains a lot of old hand-written code, XCB wants to simplify maintenance of such code by keeping the protocol description in an XML file.

Having a separate description from the code generator is also beneficial since it allows for X11 bindings for other programming languages to be generated rather easily. An example is xpyb, the X Python Binding.

The problem with automatically generated source code is that it cannot be indexed by Debian Code Search for multiple reasons:

- Running third-party code is a potential security risk. This is mitigated by the code being uploaded only by trusted Debian Developers, but nevertheless it might make Debian Code Search unstable.
- Running code before indexing makes the indexing process slow.
- Running code requires the dependencies to be present, which can be solved by installing the dependencies and running the code in a chroot-environment[13], but that requires a lot more resources than simply indexing files.

On the other hand, automatically generated source code is often, and certainly in the case of XCB, not very valuable to read. This is similar to how the generated assembler code of any advanced C compiler is likely to not be easy to read. The unavailability of auto-generated code in Debian Code Search is thus not a very big loss, but undesirable nevertheless.

An acceptable workaround for open-source projects might be to include a pre-generated version (if feasible) in their releases.

[11] Undoubtedly also closed-source projects, but that's out of scope for this work.
[12] `http://xcb.freedesktop.org/`
[13] see `pbuilder`

4.9 Test setup, measurement data and source code

All benchmarks have been performed on the following desktop computer:

- Intel® Core™ i7-2600K (3.40 GHz)
- Intel DH67GD motherboard
- 4 x G.SKILL F3-10666CL7-4GBXH ⇒ 16 GiB DDR3-RAM 1333 MHz
- Intel SSDSC2CT18 (180 GiB SSD)

The computer was running Linux 3.5 and CPU frequency scaling was disabled to not interfere with the measurements and to ensure the CPU runs with maximum performance:

```
for i in /sys/devices/system/cpu/cpu[0-9]; do
    echo performance > $i/cpufreq/scaling_governor
done
```

All measurement data used for generating graphs or tables is available at
`http://codesearch.debian.net/research/`[14].

The source code of Debian Code Search is available on Github:
`https://github.com/debiancodesearch/dcs`

[14] Mirrored at `http://michael.stapelberg.de/dcs/` in case the service will be shut down at some point in time.

4.10 Overall performance

Of course, outside of theoretical simulations, a search engine is not used by precisely one person at a time, but more people might want to access it. This section examines the overall performance of Debian Code Search by simulating real-world usage in various ways.

This section seeks to determine the number of queries per second (qps) that Debian Code Search can handle. Keep in mind that this number is a worst-case limit: Only actual search queries are measured, while the time which users normally spend looking at the search results or accessing static pages (the index or help page for example) is neglected. That is, assuming DCS could handle 5 queries per second, that means that in the worst case, when there are 6 people who want to search in the very same second, one of them has to wait longer than the others (or everyone has to wait a little bit longer, depending on whether requests are artificially limited to the backend).

4.10.1 Measurement setup

To measure HTTP requests per second for certain access patterns, there are multiple programs available:

ApacheBench Shipping with the Apache HTTP server, this tool has been used for years to measure the performance of the Apache server and others. While it supports a gnuplot output format, the gnuplot output is a histogram since it is sorted by total request time, not sequentially.

siege A multi-threaded HTTP load testing and benchmarking utility.

weighttp Developed as part of the lighttpd project, weighttp describes itself as a lightweight and small benchmarking tool for webservers. It is multi-threaded and event-based. weighttp's command-line options are similar to those of ApacheBench, but it does not support dumping the raw timing measurements.

httperf A single-threaded but non-blocking HTTP performance measurement tool. Its strength is its workload generator, with which more realistic scenarios can be tested rather than just hitting the same URL over and over again.

Unfortunately, none of these tools provides a way of obtaining the raw timing measurements. Therefore, an additional parameter has been implemented in `dcs-web` which makes `dcs-web` measure and save the time spent for processing each request. This measurement file can later be processed with gnuplot.

To verify that these measurements are correct, two lines of the very simple weighttp source code have been changed to make it print the request duration to stdout. Then, `dcs-web` was started and the benchmark was run in this way:

```
export DCS="http://localhost:28080"
./dcs-web -timing_total_path="dcs-total-1.dat" &
./weighttp -c 1 -n 100 "$DCS/search?q=XCreateWindow" \
    | grep 'req duration' > weighttp-vs-internal.raw
```

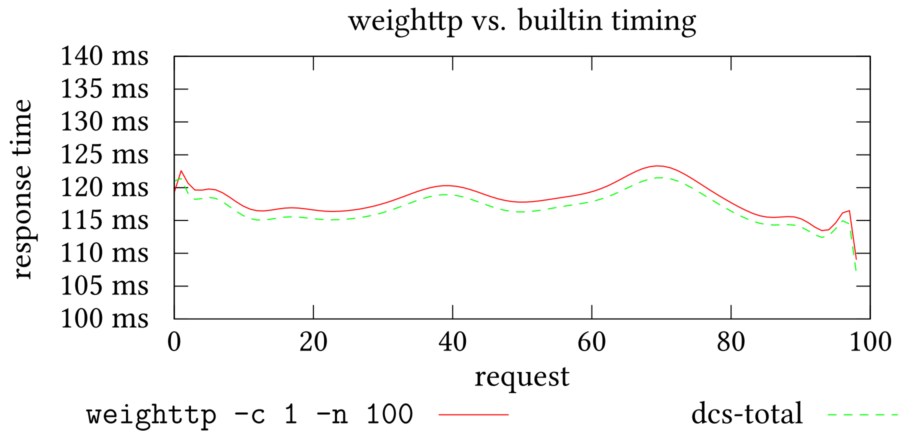

Figure 4.6: Comparison of response timing between HTTP benchmarking tool weighttp and the dcs-internal measurements. The figure confirms that weighttp and the internal measurements are working correctly. The lines differ slightly due to overhead for request parsing and receiving the reply, which is not contained in dcs-total.

Figure 4.6 shows that the measurements strongly correlate and from now on, it can be assumed that both, `dcs-web` measurements and weighttp work properly.

The aforementioned performance measurement tools allow for setting the number of concurrent requests. It is obvious that by using precisely one concurrent request (that is, the next request can only start after the current one finished), the benefits of a multi-core computer are not leveraged at all, and neither do requests in the real world wait for other requests to finish.

As you can see in figure 4.7, there are noticeable differences between one concurrent request and four concurrent requests, but afterwards the results stay the same except for queries which are answered very quickly. This fits our expectation since the machine on which the software is benchmarked uses a quad-core processor.

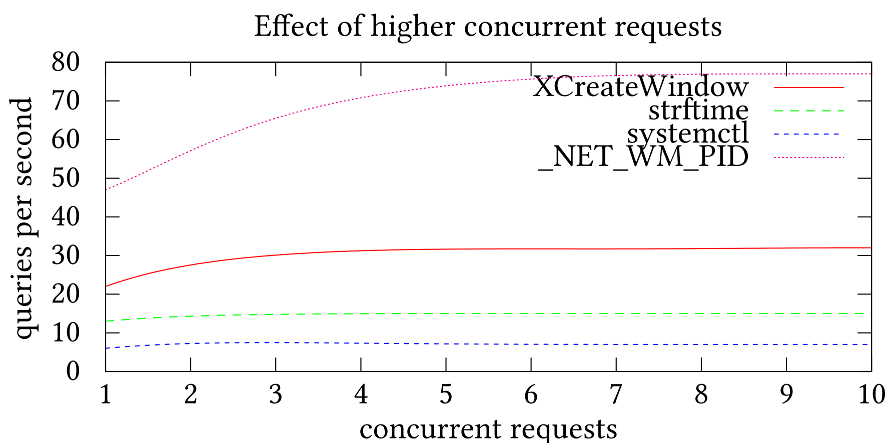

Figure 4.7: On a quad-core processor, using four concurrent requests yields best results.

4.10.2 Queries per second by query term

Since different queries have a different number of files which have to be searched, the time necessary to process each query depends on the query. Therefore, making a statement such as "DCS handles 14 queries per second on this machine" is not possible without also specifying which query was used to figure out this number.

query	# res	qps	time
XCreateWindow	3116	32	3 s
_NET_WM_PID	266	75	1 s
systemctl	739	7	12 s
ifconfig	8458	26	3 s
(?i)bloomfilter	1494	5	19 s
smartmontools	723	28	3 s
strftime	37 613	15	6 s
PNG load	71	6	16 s
pthread_mutexattr_setpshared	323	17	5 s
$\overline{x}_{\text{arithm}}$		23.4	

Table 4.7: Queries per second by query term. Higher is better.

The above queries do not necessarily reflect real-world queries. A much more interesting way of determining the queries per second rate is looking at a (popular) live system with real-world queries.

4.10.3 Queries per second, replayed logfile

The performance measurement tool httperf supports accessing URLs from a file. For this measurement, the first 250 queries of the real world query log during the timespan from 2012-11-06 to 2012-11-14 have been converted to httperf's wlog format, then httperf was started like this:

```
$ httperf --wlog=wlog --port 38080 --ra 4 --num-conn 250
Total: connections 250 requests 250 replies 250 test-duration 83.407 s
[...]
Request rate: 3.0 req/s (333.6 ms/req)
Request size [B]: 96.0

Reply rate [replies/s]: min 0.0 avg 3.1 max 11.2 stddev 3.0 (16 samples)
Reply time [ms]: response 5727.4 transfer 51.1
Reply size [B]: header 201.0 content 35557.0 footer 2.0 (total 35760.0)
Reply status: 1xx=0 2xx=236 3xx=0 4xx=14 5xx=0
```

As you can see, the rate of three queries per second is much lower than $\overline{x}_{\text{arithm}} = 23.4$ in table 4.7 with our synthetic way of benchmarking.

4.10.4 Real-world response times

This section presents different aspects of the same data as has been analyzed in section 4.10.3.

Figure 4.8 demonstrates that most requests are handled in less than one second.

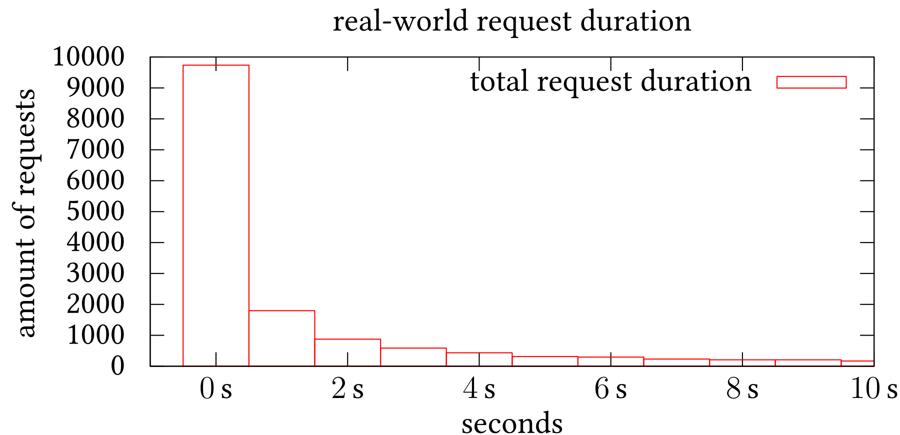

Figure 4.8: Most real-world requests are handled in less than one second.

Figure 4.9 reveals that the processing step which takes the most time is searching through the files to determine any actual matches. This is to be expected because the regular expression matching algorithm has not been optimized and the server on which Debian Code Search ran at the time during which the data was collected only has spinning hard disk drives, no solid state disks.

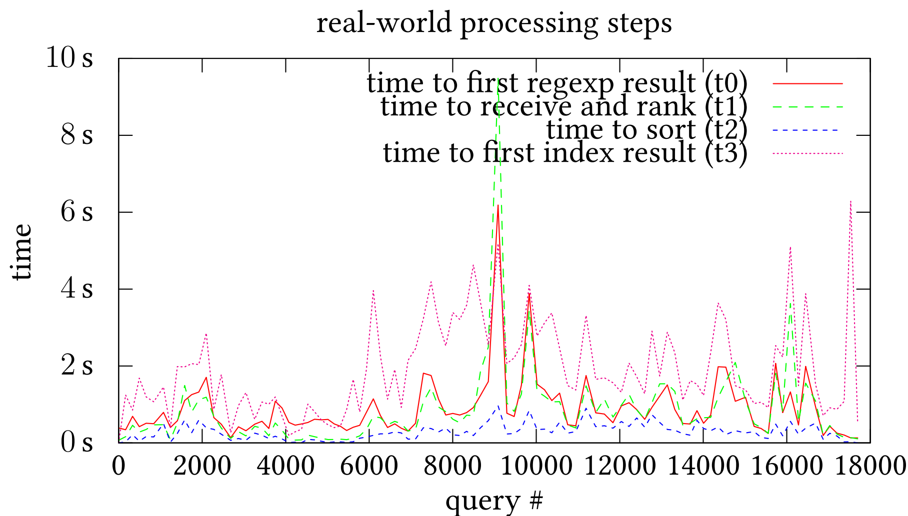

Figure 4.9: Searching files takes the most time, followed by trigram index lookups.

You can also observe several spikes. The biggest one in the middle of the graph was caused by a denial of service (DoS) attack[15], others are caused by computationally intensive queries.

[15] This is not obvious from the graph, but the log files confirm that it was a DoS attack.

4.11 Performance by processing step (profiling)

When optimizing software, the first step is to profile the software to see which part of it is actually slow. This is important because intuitively, one might have an entirely different part in mind, for example an algorithm which has some potential for optimization, while there are many other low-hanging fruit which can yield even greater performance improvements.

To profile which parts of Debian Code Search are slow, additional parameters have been added to `dcs-web`, just like in section 4.10.1. Since the code is so simple, the additional measurements do not influence the performance itself in any noticeable way:

```
func Search(w http.ResponseWriter, r *http.Request) {
    var t0, t1 time.Time
    t0 = time.Now()
    // ... send request to index-backend ...
    t1 = time.Now()
    // ...
    w.Header().Add("dcs-t0", fmt.Sprintf("%.2fms",
       float32(t1.Sub(t0).Nanoseconds())/1000/1000))
}
```

4.11.1 Before any optimization

Figure 4.10 shows that a substantial amount of time is used for the trigram index lookup: It takes about 40 ms until the first result appears. Note that results are ranked as they are received, so the “ranking” step covers receiving and ranking the files.

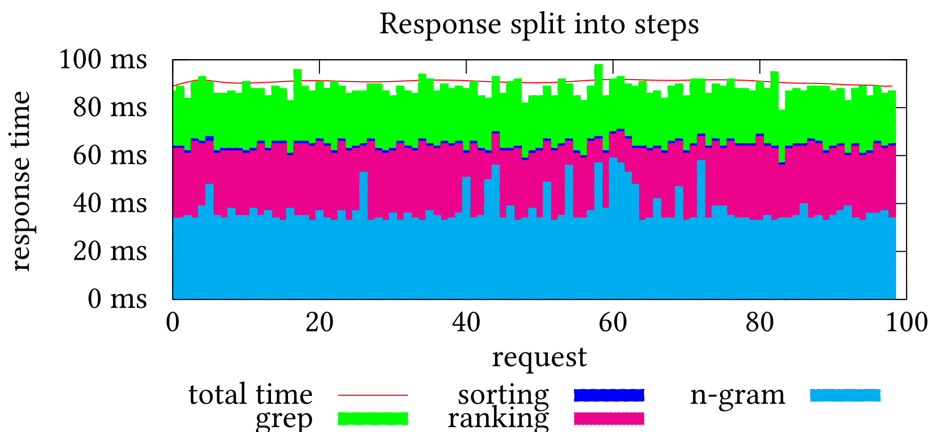

Figure 4.10: The trigram index lookup (“n-gram”) took almost as long as the actual searching of files containing possible matches (“grep”) before optimization.

4.11.2 After optimizing the trigram index

Section 3.8.4 ("Posting list decoding implementation") and 3.8.6 ("Posting list query optimization") explain the different optimizations to the trigram index in depth.

After these optimizations, the trigram lookup step is $\approx 4\times$ faster. Since the "ranking" step also contains trigram result retrieval, it is also faster.

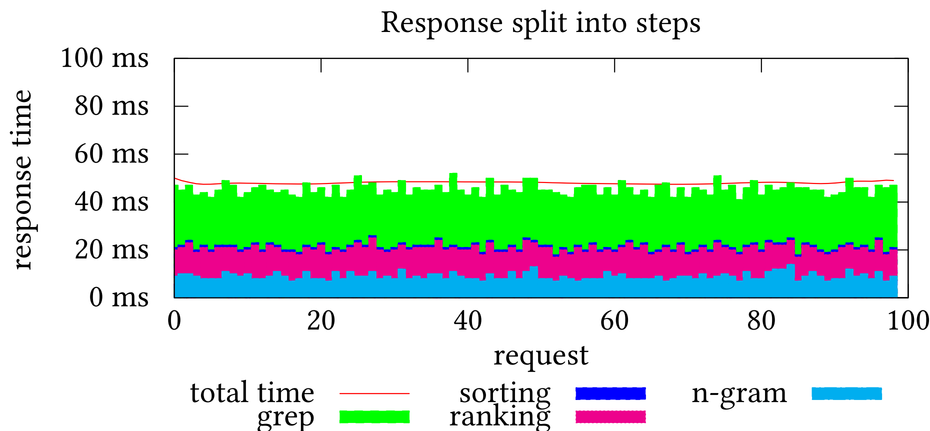

Figure 4.11: Optimizing the trigram lookup cuts the total time in half.

4.11.3 After re-implementing the ranking

It turns out that the cause for the long ranking time in figure 4.11 was that the implementation used regular expressions. By replacing the use of regular expression with hand-optimized, equivalent code, the time used for ranking could be cut to half, as you can observe in figure 4.12:

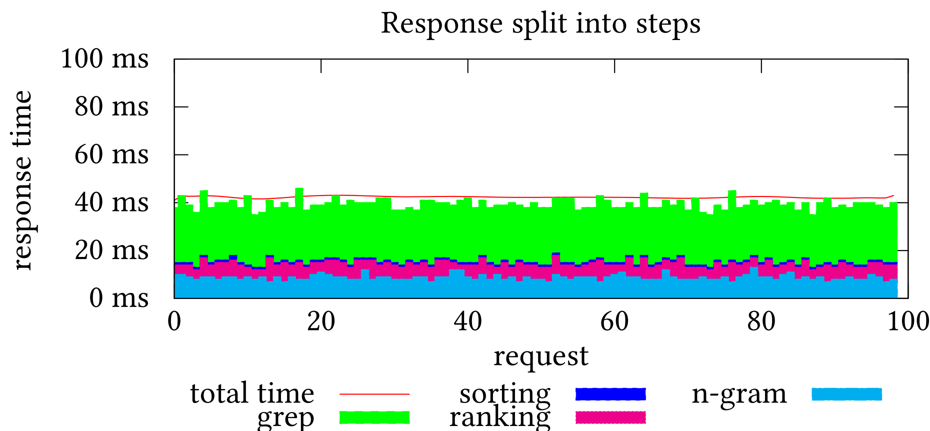

Figure 4.12: Replacing regular expression matching with a loop cuts the ranking time in half.

Intuitively, one would now try to optimize the grep step. However, as it is rather complicated and not well-documented, this is beyond the scope of this work.

5 Conclusion

This thesis has demonstrated that it is possible to implement a search engine over a large amount of program source code in the setting of an Open Source Linux distribution such as Debian.

While optimizations of the trigram index were necessary, it turned out that the simple design of DCS which came to mind first was good enough: Debian Code Search could handle the load which the near-simultaneous announcement via multiple channels (Debian mailing lists, twitter, reddit, …) caused, with only a few hiccups during the 2-3 day period of high load. It should be stressed that in this initial deployment, only one server was used.

Debian Code Search uses metadata from Debian such as the package's popularity, package dependencies, its description, and others. This fact validates the usefulness of Debian's rigid quality assurance processes such as the recent switch to machine-readable copyright information.

After the launch of DCS, many people have written mails in which they thank me, and the general sentiment was very positive. But not only do Debian developers and users feel positive about it, they also use it in bug reports, when chatting and when discussing on mailing lists to prove their points, link to a specific line in the source, or as a way to analyze the scope of a problem.

6 Future Work

Of course, there is a lot of potential for future work. Depending on the acceptance and success of the search engine within the global developer community, one has to think about further optimizing the search engine and/or scaling its deployment. A few simple but time-consuming ideas are to further optimize the trigram index format for the purpose of DCS.

Depending on the actual real-world usage (more simple identifier/word lookups or more complex regular expressions?) it might be worthwhile to build a so-called lexicon and build a trigram index based on that (as described in "Partially specified query terms"[29]). One could then perform the regular expression matching on the lexicon instead of the actual files and save searching through a lot of data.

To ease the load on the trigram index for case-insensitive queries, one could look into extending the `dcsindex` program to write two different types of index: one with case-sensitive trigrams and one with case-insensitive trigrams. While this would increase the index size, although not by $2\times$, it would considerably speed up case-insensitive queries.

Furthermore, the search engine currently only works on text. It does not understand the different programming languages which exist and therefore cannot provide some more advanced features like 100% precise cross-links between implementation and definition. Modifying the code to closely work together with a compiler, which is the most obvious tool for language analysis, is a complex endavour which has to be done for each language one wants to support.

Another big and time-consuming task is to evaluate the ranking in a better way, perhaps with studies, by doing A/B testing or by data-mining the real-world query logs and signals. Possibly, the ranking could even be adjusted in reaction to the user behavior, without any manual interaction.

7 Appendix A: Screenshots

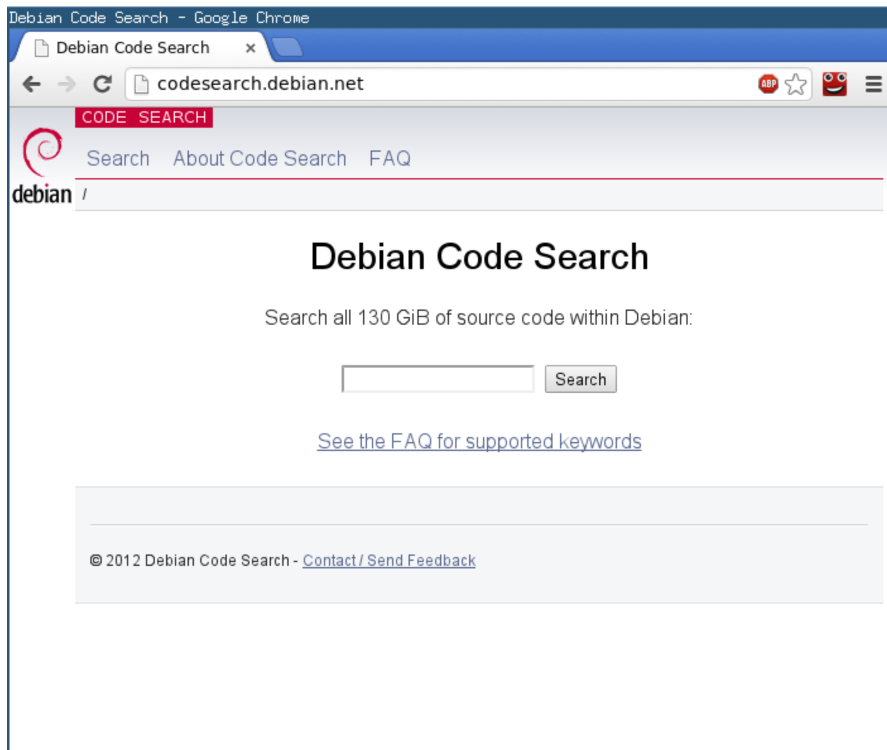

Figure 7.1: Start page of Debian Code Search

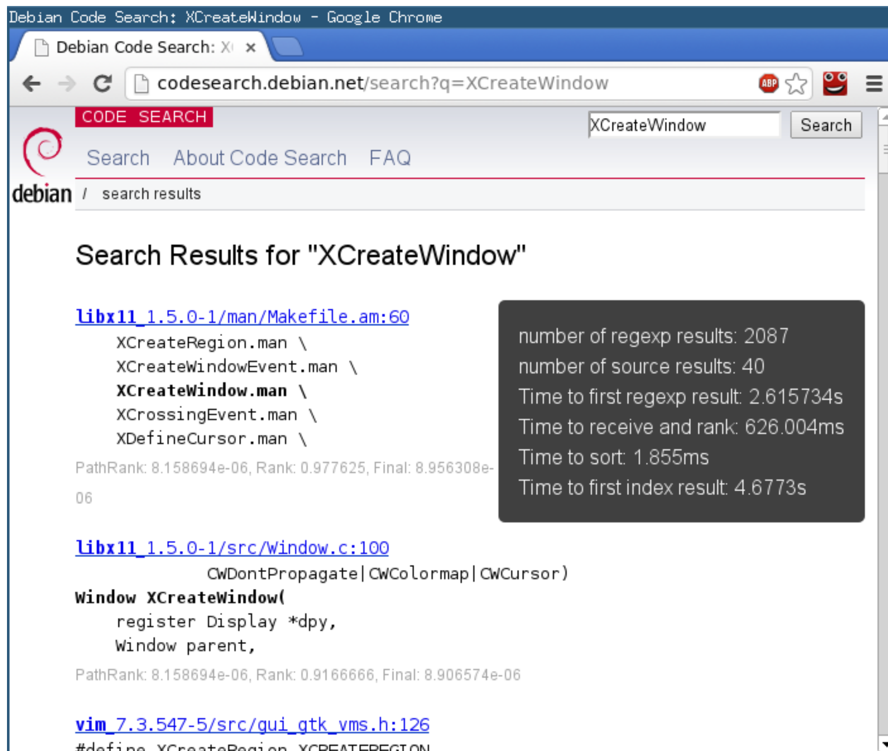

Figure 7.2: Search results for `XCreateWindow`

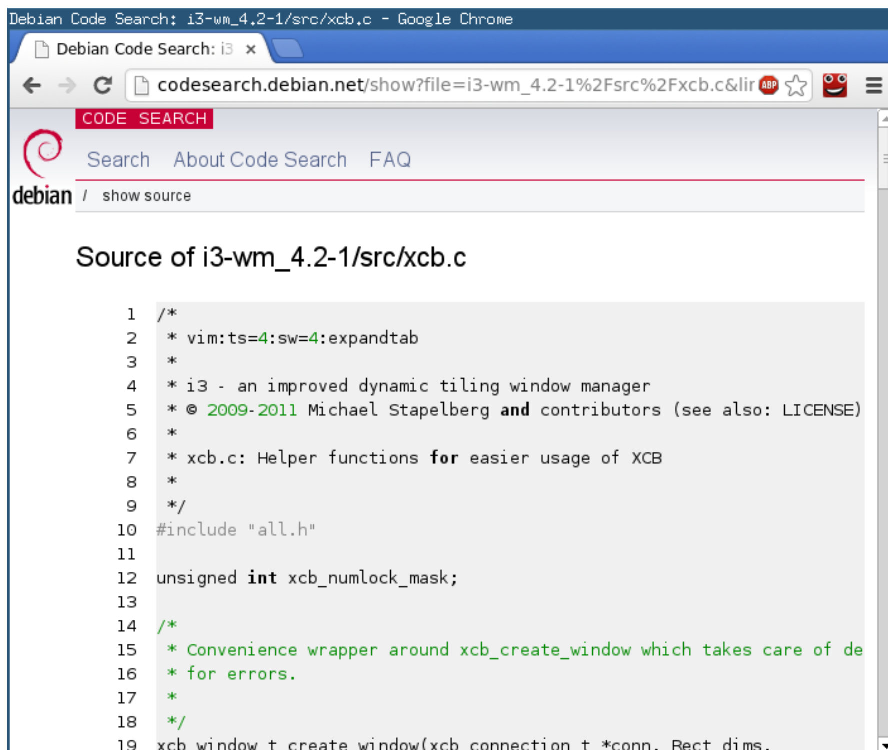

Figure 7.3: Displaying source code

Bibliography

[1] Brin, S. ; Page, L.: The anatomy of a large-scale hypertextual Web search engine. In: *Computer networks and ISDN systems* 30 (1998), Nr. 1-7, S. 107–117

[2] Cox, Russ: Regular Expression Matching with a Trigram Index. (2012). http://swtch.com/~rsc/regexp/regexp4.html

[3] Davies, J. ; Zhang, H. ; Nussbaum, L. ; German, D.M.: Perspectives on bugs in the debian bug tracking system. In: *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on* IEEE, 2010, S. 86–89

[4] Dean, Jeff: Challenges in Building Large-Scale Information Retrieval Systems. (2009). http://research.google.com/people/jeff/WSDM09-keynote.pdf

[5] Geer, D.: Chip makers turn to multicore processors. In: *Computer* 38 (2005), Nr. 5, S. 11–13

[6] golang.org: FAQ - The Go Programming Language. (2012). http://golang.org/doc/go_faq.html#history

[7] Google: Encoding - Protocol Buffers. (2012). https://developers.google.com/protocol-buffers/docs/encoding#varints

[8] Hidalgo Barea, A. et.al.: Analysis and evaluation of high performance web servers. (2011). http://upcommons.upc.edu/pfc/bitstream/2099.1/12677/1/memoria.pdf

[9] Lemire, Daniel ; Boytsov, Leonid: *Decoding billions of integers per second through vectorization.* http://arxiv.org/pdf/1209.2137v1.pdf. Version: 2012

[10] Linden, Greg: Geeking with Greg: Marissa Mayer at Web 2.0. (2006). http://glinden.blogspot.de/2006/11/marissa-mayer-at-web-20.html?m=1

[11] Mateos-Garcia, J. ; Steinmueller, W.E.: The institutions of open source software: Examining the Debian community. In: *Information Economics and Policy* 20 (2008), Nr. 4, S. 333–344

[12] Nussbaum, Lucas ; Zacchiroli, Stefano: The Ultimate Debian Database: Consolidating Bazaar Metadata for Quality Assurance and Data Mining. In: *7th IEEE Working Conference on Mining Software Repositories (MSR'2010).* Cape Town, South Africa, 2010

[13] Pennarun, Reinholdtsen Allombert: Alioth: Debian Package Popularity Contest. (2003). https://alioth.debian.org/projects/popcon/

[14] Pennarun, Reinholdtsen Allombert: Popularity-contest Frequently Asked Questions. (2004). http://popcon.debian.org/FAQ

[15] Riel, Rik van: PageReplacementDesign - linux-mm.org Wiki. (2010). http://linux-mm.org/PageReplacementDesign

[16] Schmager, F.: Evaluating the GO Programming Language with Design Patterns. (2011). http://researcharchive.vuw.ac.nz/bitstream/handle/10063/1553/thesis.pdf?sequence=1

[17] Thorpe, S. ; Fize, D. ; Marlot, C. et.al.: Speed of processing in the human visual system. In: *nature* 381 (1996), Nr. 6582, S. 520–522

[18] W3Techs: Historical trends in the usage of web servers, July 2012. (2012). http://w3techs.com/technologies/history_overview/web_server

[19] WHITEHEAD II, J.: Serving Web Content with Dynamic Process Networks in Go. (2011). http://www.cs.ox.ac.uk/people/jim.whitehead/cpa2011-draft.pdf

[20] Wikipedia: *Bloom filter*. http://en.wikipedia.org/w/index.php?title=Bloom_filter&oldid=506427663. Version: 2012. – [Online; accessed 14-August-2012]

[21] Wikipedia: *Debian — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Debian&oldid=518794222. Version: 2012. – [Online; accessed 26-October-2012]

[22] Wikipedia: *Levenshtein distance — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=525823323. Version: 2012. – [Online; accessed 7-December-2012]

[23] Wikipedia: *N-gram — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=N-gram&oldid=514951683. Version: 2012. – [Online; accessed 30-October-2012]

[24] Wikipedia: *Page cache — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Page_cache&oldid=516103102. Version: 2012. – [Online; accessed 11-November-2012]

[25] Wikipedia: *Query optimizer — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Query_optimizer&oldid=521108336. Version: 2012. – [Online; accessed 21-November-2012]

[26] Wikipedia: *Regular expression — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=525511640#POSIX. Version: 2012. – [Online; accessed 3-December-2012]

[27] Wikipedia: *S.M.A.R.T. — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=S.M.A.R.T.&oldid=525820381. Version: 2012. – [Online; accessed 7-December-2012]

[28] Wikipedia: *Solid-state drive — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=Solid-state_drive&oldid=525844401`. Version: 2012. – [Online; accessed 3-December-2012]

[29] Witten, I.H. ; Moffat, A. ; Bell, T.C.: *Managing gigabytes: compressing and indexing documents and images.* Morgan Kaufmann, 1999

[30] Zini, E.: A cute introduction to Debtags. In: *Proceedings of the 5th annual Debian Conference*, 2005, 59–74

Most of the citations refer to publications which are accessible in the world wide web since the subject matter is very recent.

List of Figures

3.1 High level overview of a request's flow in Debian Code Search. 8
3.2 Architecture overview, showing which different processes are involved in handling requests to Debian Code Search.[1] 9
3.3 Architecture overview with load-balancing possiblities. 10
3.4 The Codesearch index format. Trigram lookups are performed as described below. 17
3.5 Trigram lookup time grows linearly with the query's trigram count and logarithmically with the index size. 33 queries with varying length were examined. 18
3.6 Posting list decoding time increases linearly with combined posting list length. 18
3.7 The optimized version written in C is faster for both small and large posting lists. At sufficiently large posting lists, the C version is one order of magnitude faster. 20
3.8 The simpler `varint` algorithm is as fast as `group-varint` for short posting lists and even faster than `group-varint` for longer posting lists. 21
3.9 It takes only a fraction ($\approx \frac{1}{3}$) of intersection steps to get to the final result R. 1500 random function and variable names have been tested. 23
3.10 With the heuristic explained above, the amount of false positives does not exceed two files on average; the total speed-up is $\approx 2\times$. 23

4.1 Average mtime difference between the source package and its `.c` or `.h` files 35
4.2 Trigram lookup latency distribution by number of results. Each point represents one out of 1500 randomly chosen function or variable names. 41
4.3 Histogram of trigram lookup latency 42
4.4 Source matching latency distribution by number of potential results (the number of files which are searched for matches) 43
4.5 Histogram of source matching latency. Most source matching is done within 20 ms, but the tail is considerably longer than for trigram lookup latency. . . 43
4.6 Comparison of response timing between HTTP benchmarking tool weighttp and the dcs-internal measurements. The figure confirms that weighttp and the internal measurements are working correctly. The lines differ slightly due to overhead for request parsing and receiving the reply, which is not contained in dcs-total. 49
4.7 On a quad-core processor, using four concurrent requests yields best results. 49
4.8 Most real-world requests are handled in less than one second. 51
4.9 Searching files takes the most time, followed by trigram index lookups. . . . 51
4.10 The trigram index lookup ("n-gram") took almost as long as the actual searching of files containing possible matches ("grep") before optimization. 52
4.11 Optimizing the trigram lookup cuts the total time in half. 53

4.12 Replacing regular expression matching with a loop cuts the ranking time in half. 53

7.1 Start page of Debian Code Search 56
7.2 Search results for `XCreateWindow` 57
7.3 Displaying source code 57

List of Acronyms

BSD Berkeley Software Distribution

CSS Cascading Style Sheets, a language for describing how HTML content on a website should look.

DBMS Database management system

DCS Debian Code Search

DNS Domain Name System, used for resolving hostnames such as `www.heise.de` to IP addresses such as `2a02:2e0:3fe:100::8`

DoS Denial of Service

DSA Debian System Administration

FOSS Free/libre Open Source Software

GCC GNU Compiler Collection

GCS Google Code Search

GTK GIMP Toolkit, a toolkit for creating graphical user interfaces

HTML Hyper Text Markup Language

HTTP Hyper Text Transfer Protocol, a protocol used for accessing websites or webservices.

IDE Integrated Development Environment

LRU Least-recently Used

PAM Pluggable Authentication Modules

PNG Portable Network Graphics

POSIX Portable Operating System Interface

Qt pronounced "cute", a framework for creating graphical user interfaces

SCM Source Control Management

S.M.A.R.T. Self-Monitoring, Analysis and Reporting Technology

SSD Solid State Disk

UUID Universally Unique Identifier

XML Extensible Markup Language