Project Report and Technical Documentation

Thomas Jund <info@jund.ch>
Andrew Mustun <andrew@mustun.com>
Laurent Cohn <info@cohn.ch>

24th May 2004

Version 1.0

Abstract
In this paper we present ***quaneko***, a tool to efficiently find data on the local computer system. The purpose of this document is the technical specification and description of the tool. Please note that this is not a user manual. You can find the user manual on the project web site http://quaneko.sf.net.

To engineer an efficient index, which is the heart of the project, is a challenging task. Besides the importance of good performance for search queries and fast indexing, there has to be a flexible handling for file types that are used in a typical modern office environment. Adding support for individual file formats has to be as simple as possible and must not need any programming skills.

Compared to other search tools like the built-in Windows Search, ***quaneko*** can be configured to parse any file formats.

After reading this document you will know which indexing system was implemented in ***quaneko*** and how it was developed. To learn about the way how to use the tool we recommend to read the manual mentioned above.

More information and file downloads for ***quaneko*** are available from http://quaneko.sf.net.

Contents

1 Introduction

Whenever you find that you are on the side
of the majority, it is time to reform.
(Mark Twain)

This chapter briefly introduces the project and defines the scope of this documentation.

1.1 Motivation

In the information age we live in, the skill to retrieve information in a fast and accurate manner has become crucial to most people who work with computers. However, while searching and browsing the Internet has become a matter of course, finding the right file on the local hard disk can still prove to be a day to day challenge.

The growing amount of personal and downloaded information on the computer makes it hard to keep track of the various directories and files. Especially finding older files can become time consuming and frustrating.

Of course it is possible to search for files using the standard search tools provided by the operating system (e.g. 'Search for Files and Folders' under Windows or the find / grep commands under Unix systems). In most cases ***quaneko*** is superior to these methods of information retrieval. Mainly because of these aspects:

- ***quaneko*** creates an index over the files and directories that need to be searched. This means that once an index is generated, searching for a word takes only seconds, no matter how many files need to be searched.
- ***quaneko*** can search any format type as long as there is a tool available and configured for converting the format into plain text. Supporting your own individual file format is as easy as configuring a command that converts the format into plain text.

In a nutshell, ***quaneko*** was built with search performance and flexible file type handling in mind. ***quaneko*** is designed to keep track of user specified directories and files. Based on a given keyword, ***quaneko*** lists all files which contain that word.

1.2 Task Description

This task outline is based on a description that was originally worded by Karl Rege:

"Disk capacity grows steadily and so does the need for tools that allow to organize and search for files by individual criteria. There are tools available that can be used to assign keywords to documents and search for those keywords. However, finding the appropriate keywords isn't trivial. The text search that many systems have built in fails with file formats that store the text in Unicode or in a compressed format. A tool is to be provided to search, possible with generation of index, over files in common formats (doc, pdf, html, etc.) for keywords similar like Google does it (single keywords only)."

1.3 Objective and Structure of this Document

The documentation is structured in three parts:

- In the first part we analyze the problem (section 2).
- The second part is about the design and implementation of ***quaneko*** (sections 3 - 9).
- Finally we discuss the evaluation and conclusion of the project in the third part (section 13).

The objective of this document is to reveal the internal structure and design of ***quaneko***. Further, it contains the specification for the C++ API which programmers of other applications can use to communicate with the ***quaneko*** search and index engine.

2 Problem Analysis

How do I type
*"for i in *.dvi; do xdvi $i; done"*
in a GUI?
(Discussion in comp.os.linux.misc
on the intuitiveness of interfaces.)

2.1 User Requirements

This section describes the problem from the user's point of view. It briefly describes the user's goals, how they currently solve the problem of finding documents, what they expect to be able to do and how they wish to do it.

2.1.1 User Groups

Typical ***quaneko*** users:

A Home and professional users who work with many documents.
B Professional users who need to index files in specific formats (e.g. a company internal format or a format used in a certain profession such as a CAD file format).
C Software developers who want to incorporate search capabilities into their own products.
D Users who need to automate indexing / searching tasks.

2.1.2 User's Goals

The user can recursively search folders of personal interest for documents which contain a given word. Important issues are the performance (a few seconds for a search query) and that the search can be configured to include the user's favorite file types such as doc, html, pdf, txt, etc.

2.1.3 Existing Solutions

quaneko focuses on users who currently use tools provided by the operating system to search their data. These tools are not very flexible and require a lot of time to search large amounts of data.

2.2 Usability Requirements

This section focuses on the system and software requirements needed to implement the user requirements and lists the functionality of the user interfaces.

The analysis of user groups who will use ***quaneko*** has lead to the definition of three interfaces:

- A GUI for user groups *A* and *B*.
- A command line interface (CLI) for user group *D*.
- A simple C++ API for group *C*.

2.2.1 Common Requirements

All interfaces (GUI, CLI, API) must offer the following functionality:

- Creating new indexes.
- Parsing files and directories into the index.
- Updating existing indexes.
- Querying an index for a simple word.

2.2.2 GUI

The GUI is kept simple to allow quick and effective access to the available search indexes. Further it offers a user friendly way to adjust application options and to configure the format filters.

In addition to the common requirements, the GUI must allow the user to accomplish the following tasks:

- Configuring format filters (Add/Remove/Edit).

Optional:

- Preview of search results.
- Open search results in a specified application.

2.2.3 CLI

In addition to the common requirements, the command line interface must offer switches for:

- Removing individual files or directories from an existing index.
- Validating the consistence of an existing index.

2.2.4 API

The API provides equal functionality like the command line interface in the form of a C++ library for programmers. For a detailed documentation of the API, please refer to the API reference documentation in appendix A, page 27.

2.3 Supportability Requirements

2.3.1 Operating Systems

quaneko is intended to work under all major Unix systems as well as Windows (32bit) and Mac OS X systems.

2.3.2 Programming Language and Libraries

The core parts of ***quaneko*** are implemented in C++ using the STL. For the GUI, the Qt toolkit [8] is chosen to offer the required portability in combination with C++.

3 Design

3.1 Architecture Overview

The architecture of ***quaneko*** follows the three-tier standard. The first tier consists of the user interfaces. The second tier processes the requests coming from the user interface tier. The third tier stores the index and holds the data files that are indexed. ***quaneko*** uses the normal file system as the third tier.

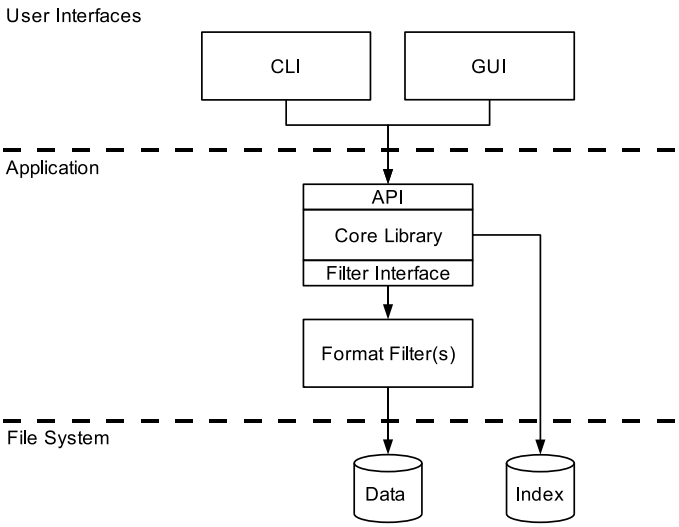


Figure 1: Architecture Overview

For a detailed description of the User Interfaces, please refer to the **Quaneko User Manual** [1]. The data files and the index over the data files reside on the local hard disk.

The following sections describe the application tier of ***quaneko***.

3.2 Core Library

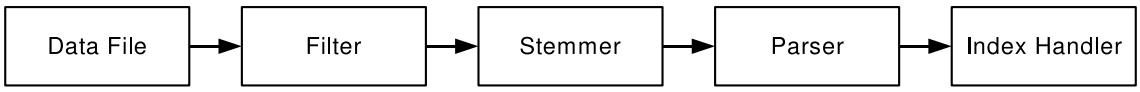


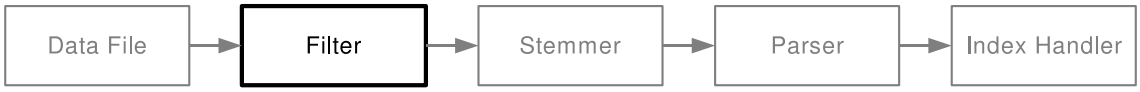
Figure 2: Core Library, Components

The core library contains all core functionality of ***quaneko***. The following list describes the most important components of the Core Library:

- The **Filter** module converts files in various formats into plain text (section 4).
- The **Stemmer** module applies stemming to words (section 5).
- The **Parser** extracts words from files and directories through the Filter Interface and uses the Index Handler to store the information about words and files. Further, the Parser is responsible to manage the file register and the word register (section 6).

- The **Index Handler** persistently manages the links between words and files (section 7).
- The **API** builds the interface between the middle tier and the user interfaces. For a detailed documentation of the API, please refer to the API reference documentation in section A.
- The **Settings** module stores and handles meta information about every index as well as user preferences (section 9).

4 Filter Module



The filter module manages the format filters for converting a non-text file format into plain text. For example PDF files can be converted into plain text with a command line utility called 'pdf2txt'. The filter module maintains a list of all available format filters and calls the third party utilities to do the conversion.

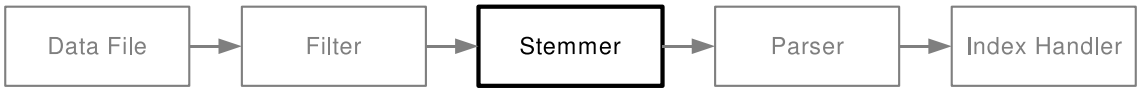
4.1 Input

- The path to a file in any kind of format (txt, doc, html, pdf, ..).
- A set of format filters (read from the application wide configuration file).

4.2 Output

- If a filter is available to convert the given file into plain text, a plain text file is produced.
- If there is no filter available for that format, an error is handed back to the caller and no output is produced.

5 Stemming Module



quaneko has stemmers for various languages built in. The stemmers are based on a set of stemmers that are available from the Snowball [3] project page.

In ***quaneko***, stemming can be enabled for exactly one language or disabled at the time a new index is created. Enabling stemming reduces the number of unique words in large texts significantly 1.

Type of Text	Language	Unique Words before stemming	Unique Words after stemming	Reduction
RFC (technical)	en	~47000	~39500	16%
Moby Dick	en	~19500	~13000	33%
Goethes Faust	de	~13500	~10000	26%

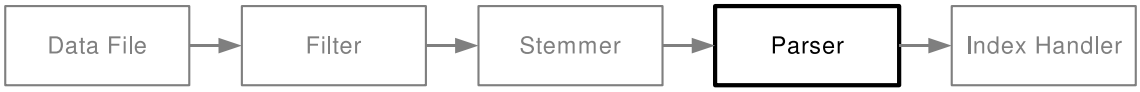
Table 1: Reduction of unique words found in a text with stemming enabled.

Stemming is applied to all words that are read from the data files as well as to the search word entered by the user. If the user searches for example for the word 'cycling', ***quaneko*** looks in the index for 'cycl' which will find all files containing 'cycle', 'cycling', 'cycles' and 'cycled'.

At the time of writing this document, stemming is supported for the following languages:

- Danish (da)
- German (de)
- English (en)
- Spanish (es)
- Finnish (fi)
- French (fr)
- Italian (it)
- Dutch (nl)
- Norwegian (no)
- Portuguese (pt)
- Russian (ru)
- Swedish (sv)

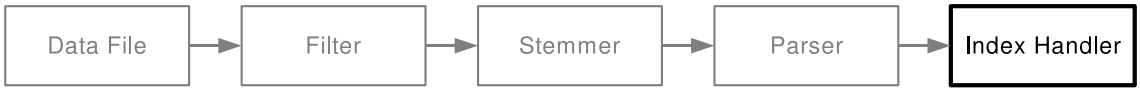
6 Parser Module



The parser is responsible for the actual indexing process. It reads the plain text output of a filter and adds all words that are found to the index. The parser converts all words to lower case and applies stemming if activated. Further, it can ignore all numbers and always strips away any white space or special characters from the beginning and end of a word.

Another important part of the parser is the ability to update existing indexes. The parser removes files that are no longer available on disk, adds new files in directories that were previously parsed and updates files that have changed.

7 Index Handler



The index handler deals with the indexing system. The indexes link each word with a number of files in which the word occurs.

The indexing system is built on three different types of indexes:

- Direct Index.
- Inverted Index.
- Array Index.

Each index type has its own advantages in means of disk space and performance .

In addition to the indexes, two registers keep track of the words and files that have been indexed:

- Word Register
- File Register

Figure 3 shows an overview of the whole indexing system. The components are described in detail in the following sections.

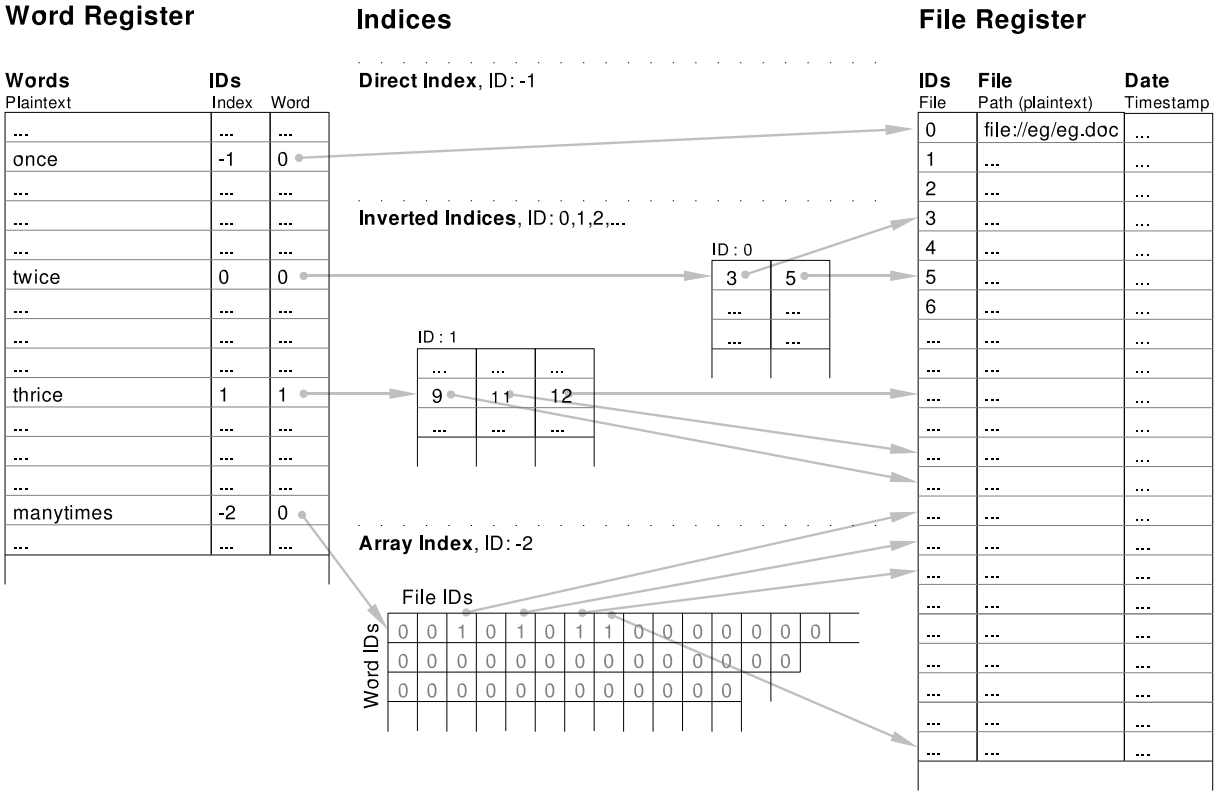


Figure 3: Overview of the complete indexing system.

7.1 Word Register

The Word Register contains all unique words available in the index. Each word is saved with an **Index ID** and a **Word ID** . The Index ID specifies which index links the word to a number of File IDs. The Word ID is used to point to the entry in the index. Each combination of Index ID and Word ID is unique and referred to as **Full Word ID** . The register is alphabetically sorted. For better performance, the word register is kept in the memory.

Word	Index ID	Word ID
aardvark	-1	37
abandon	-2	4
abby	2	4
...	..	..

Table 2: Example for a Word Register.

7.2 File Register

The File Register saves the URIs or paths to the files and directories that have been indexed. Every file or directory has a unique **File ID**. The modification time of the file is also stored to detect changes of the file. The update function relies on that time stamp. The time stamp is measured in seconds since the Epoch (00:00:00 UTC, January 1, 1970). The File Register does not have any particular order.

URI or Path	File ID	Time Stamp
/home/tux/data	0	1083350910
/home/tux/data/animals.txt	37	1083360539
/home/tux/data/fish	3	1083360145
/home/tux/data/fish/recipes.txt	42	1083360561
...	..	..

Table 3: Example for a File Register.

7.3 Direct Index

Word Register		
Word	**Index ID**	**Word ID**
aardvark	-1	37

File Register		
File ID	**Path / URI**	**Timestamp**
37	/.../animals.txt	...

Figure 4: In this example, the word 'aardvark' is directly linked to the only file it appears in. For this case no intermediate lookup tables are needed.

Direct indexing is used for words which occur in one file only.

In this case the Word ID equals the File ID and the Index ID is set to -1 (see Figure 3). Our research has shown that between 30% and 60% of all words appear in only one file and can therefore be indexed with this efficient method (see Figure 7).

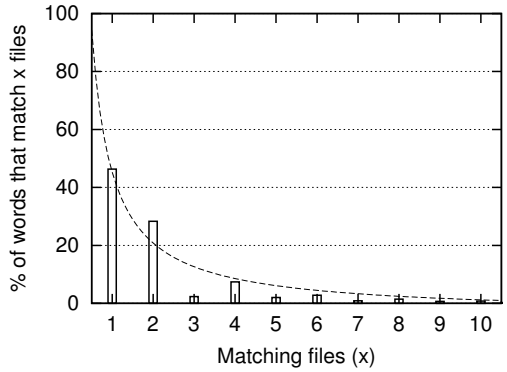


Figure 5: An example for the distribution of words in 1000 RFC files. About half of all words are only found in one file.

7.4 Inverted Index

Word Register		
Word	**Index ID**	**Word ID**
linus	0	3

Inverted Index 0	
File ID	**File ID**
...	...
...	...
...	...
21	74
...	...

File Register		
File ID	**Path / URI**	**Timestamp**
21	/.../friends.txt	...
...	...	...
74	/.../phonebook.txt	...

Figure 6: An Inverted Index links a word to a number of files that contain that word.

quaneko uses Inverted Indexes to index words that occur in more than one file (note: for words that occur in many files, the Array Index is used instead, see 7.5).

Each row of an Inverted Index stores a number of pointers to File IDs. The row number (not stored in the file) indicates the Word ID the row refers to. Every Inverted Index has a maximum number of File IDs it can store per row.

If a word occurs in two files, it is indexed in Inverted Index 0, which can store two File IDs for one word. At the time a third file is found which also contains the same word, the word moves to inverted index 1 which can store up to four File IDs and so on.

There can be any number of Inverted Indexes in ***quaneko***. However, in most cases, having more than 8 Inverted Indexes proves to be inefficient.

An Inverted Index in ***quaneko*** can store up to $2^{(\text{Index ID}+1)}$ File IDs per row. Tests with different values have shown that this formula performs best with an acceptable overhead of unused spaces for File IDs.

The Inverted Index is optimized for finding all File IDs to a belonging Word ID. It is not optimized to find all words that occur in a certain file.

Inverted Indexes are not kept in memory.

7.5 Array Index

The Array Index is used for words which occur in many different files. The strength of the Array Index lies in indexing very common words with less disk space than an Inverted Index. Every word requires as many bits of memory as there are files indexed. For every file that does not contain the word, a '0' is stored and for every files that contains the word a '1'.

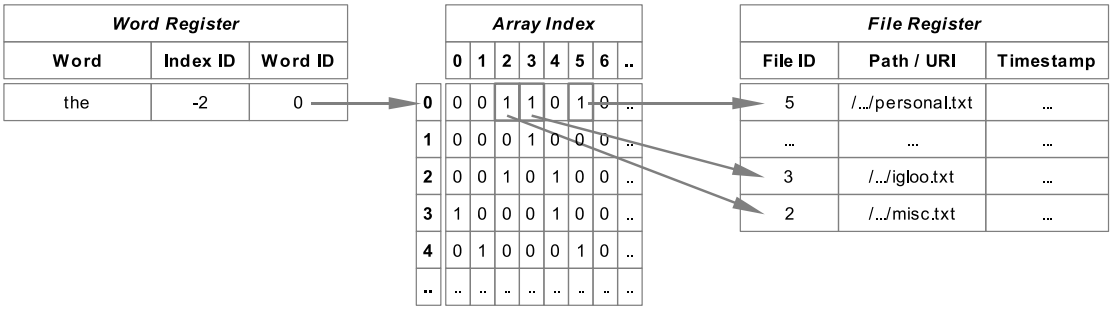


Figure 7: The Array Index is a bitmap in which every column accounts for a File ID and every row for a Word ID. If a word occurs in a file, the bit at position (Word ID/File ID) is set to 1, otherwise to 0.

The index file needs to be horizontally resized when no File IDs are available anymore. Every such resize doubles the amount of File IDs.

Words that occur in so many files that they end up in the Array Index are never removed from it anymore. This leads to the unlikely possibility that a row in the Array Index could become orphaned. Preventing this would significantly reduce the performance of the Array Index.

8 Optimization

8.1 Combining Different Types of Indexes

Our first approach of creating an index that links each word to a number of files was a simple bit map (see 7.5). When tests showed that a lot of words appear in only one or two files (see Figure 7, page 14), we have introduced Inverted Indexes.

The following diagram shows the impact of Inverted Indexes on the total index size. For this statistic, 2000 RFC files with a total size of 65.6MB were indexed.

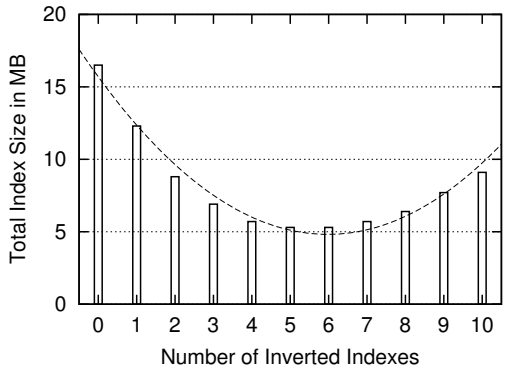


Figure 8: Impact of the inverted indexes on the total index size.

Note, that each Inverted Index can link up to $2^{(n+1)}$ files to a word. For example the first Inverted Index (0) can link each word in it to up to $2^1 = 2$ files. The Index with ID 10 can link each word to up to $2^{11} = 2048$ files (for this example, this would mean that the Array Index is never used). Table 4 shows a more complete list of the relationship between index IDs and index width.

Index ID	Max. file pointers
Inverted Index 0	2
Inverted Index 1	4
Inverted Index 2	8
Inverted Index 3	16
Inverted Index 4	32
Inverted Index 5	64
...	...

Table 4: Preconfiguration Indexing System.

By default, only 6 Inverted Indexes are used. This number can be configured in the settings file of ***quaneko*** (see section 9.2).

9 Settings

The configuration of ***quaneko*** is saved in a settings file. Under Unix and Mac OS X, this file is placed in the user's home directory. E.g.

```
/home/tux/.quanekorc
/Users/tux/.quanekorc
```

Under Windows systems, it is not clearly defined where the user's home directory is. ***quaneko*** looks for the folders specified in either of the environment variables: HOME, USERPROFILE, HOMEPATH. One possible location for the settings file is:

```
C:\Documents and Settings\Tux\.quanekorc
```

9.1 File Format

The settings file contains one key / value pair per line. Key and value must be separated by a tabulator.

9.2 Settings Reference

Global Settings

Key	Description
/config	
../numinvertedindexes	The number of inverted indexes to use for newly created indexes by default. Example: 4
/filters	
../filter1_type	File types for filter 1. The types are the file extensions this filter can handle separated by a space. Example: htm html
../filter1_app	The application used to convert the file type to plain text. In the command, %f marks the place where the data file path is inserted. %o can optionally mark the place in the command where the temporary plain text output is generated. If %o is omitted, the output of the command to stdout is redirected into the temporary plain text file. Examples: html2text ”%f” -o ”%o” pdf2txt ”%f” cp ”%f” ”%o”
../filter1_open	The application used to execute and open an indexed file. In the command, %f marks the place where the data file path is inserted. Example: notepad.exe

Index Specific Settings

Key	Description
/index/TestIndex	
../arrayindexfile	Points to the file which contains the array index for the specific index. Example: /tmp/TestIndex/arrayindex.dat
../arrayindexwidth	The current width of the array index in Bytes. This value can grow over time. Example: 128
../fileindexfile	Points to the file which contains the file register. Example: /tmp/TestIndex/fileindexfile.dat
../wordindexfile	Points to the file which contains the word register.
../ignorenumbers	If this value is 1, numbers are ignored and not parsed into the index. Otherwise they will be treated like normal words.
../invertedindex/0/file	Points to the file which contains the inverted index with the given ID (in this case 0).
../invertedindex/0/max	Maximum number of file IDs stored for one word in the index with the given ID.
../invertedindex/0/min	Minimum number of file IDs stored for one word in the index with the given ID (currently not used).
../lock	1 indicates that this index is currently in use (locked). 0 means the index is not locked and can be opened.
../stemming	The language used for stemming words. The language is indicated with its ISO 639 two letter code. Supported language codes are: da, de, en, es, fi, fr, it, nl, no, pt, ru and sv. If left empty, stemming is disabled for this index.

10 Testing Documentation

A program that has not been tested does not work.
Bjarne Stroustrup [9, pg. 712]

The testing took approximately 50% of the total development effort. A lot of the functions and modules were tested in the implementation phase.

10.1 Application Testing

We have tested the whole application (black box) using various test cases to cover different situations. Each test script automatically ensures that its postconditions are fulfilled. The following test cases are defined in the testing script:

```
(1)  ...  Test --update        File deletion
(2)  ...  Test --parse         File deletion
(3)  ...  Test --parse         All indexes
(4)  ...  Test --update        All indexes, files deleted
(5)  ...  Test --parse         All indexed, files deleted
(6)  ...  Test --update        File content exchange, modification time
(7)  ...  Test --update        File content exchange
(8)  ...  Test --update        Files emptied
(9)  ...  Test --update        File added
(10) ...  Test --parse         File added
```

An integral part of application testing is the validate switch on the CLI which ensures the correctness of the indexes.

10.2 Performance Testing

Timer functions are used on source code level to show runtime and number of calls of functions. Timer tests are run and evaluated on functions from the core library. Critical functions are easily evaluated. This is an example output of a performance test:

```
TIMER[27]: started 1 times. Total time for LDS_ArrayIndex::getFileSize: 0:000086
TIMER[28]: started 0 times. Total time for LDS_ArrayIndex::resizeFile: 0:000000
TIMER[29]: started 6296576 times. Total time for LDS_ArrayIndex::getChar: 78:546969
```

10.3 Memory leaks

To test for memory leaks, the tool Valgrind [10] was used. Further information and a detailed description of the tool is available on the project page. Command used for testing:

```
valgrind --leak-check=yes --show-reachable=yes ./cli ...
```

11 Installation

11.1 Installation of Binaries

Precompiled ***quaneko*** binaries are available from http://quaneko.sf.net. The packages can be extracted and should be ready to run.

11.2 Installation from Sources

11.2.1 Unix, Linux, Mac OS X

Under Unix compatible systems with gcc>=3.x installed, building ***quaneko*** from sources is straight forward:

- Run 'make' in the ***quaneko*** directory to build the core and CLI
- Run 'make qtgui' for the GUI (requires Qt>=3.3)

11.2.2 Windows (32bit)

This section describes the installation of ***quaneko*** from sources on a 32bit Windows platform (e.g. Windows 95, 98, ME, XP, 2000).

quaneko was tested with the free Borland command line compiler version 5.5. Although it should compile with any other ANSI compliant C++ compiler, we have experienced problems with Microsoft Visual C++ version 6.

Installing the Free Borland Command Line Tools

The tools are available for download after registering as a user from the Borland web page:

`http://www.borland.com/bcppbuilder/freecompiler/`

At the time of writing this document, the latest version was:

```
File: freecommandLinetools.exe
Version: 5.5
Release Date: 08/24/2000
Size: 8.7MB
```

The Command Line Tools can be installed by running the executable file that was downloaded from the Borland page. Make sure that you remember the path in which you install the tools.

Installing *quaneko*

The latest ***quaneko*** sources are available from http://quaneko.sf.net. Download and extract the source package to your hard disk. In the subsequent sections of this installation guide we assume that the ***quaneko*** source package was extracted into "C:\qnk".

Setting up the Environment

Launch a command prompt ("DOS Box", "MS DOS Prompt"). Set the following environment variables according to your Borland command line tools installation (we assume that the tools are installed in "C:\Borland\BCC55"):

```
set BCB=C:\Borland\BCC55
set PATH=%PATH%;%BCB%\Bin
```

Compiling the Sources

In the command prompt, change to the directory where you have installed ***quaneko***. E.g.:

```
c:
cd c:\qnk
```

Run the batch script that comes with the ***quaneko*** source package:

```
cd scripts
build_bcc55.bat
```

The command line interface "cli.exe" should now be compiled and ready to use in your installation directory. E.g. "C:\qnk\cli".

12 Project Management

12.1 Project Initiation

12.1.1 Choosing a License

From the very beginning of the project it was clear to us that ***quaneko*** will be released under an open source license. The GPL[1] was chosen primarily because it is well known and the most widely used open source license. The stemming library is licensed under the BSD License[2].

12.1.2 Hosting

SourceForge.net is the world's largest Open Source software development web site and provides free services to Open Source developers. Towards the end of the development of ***quaneko*** we decided to move the whole project to SourceForge.net to give other developers access to it and to keep the project alive.

The ***quaneko*** homepage can be accessed under: http://quaneko.sf.net.

12.2 Documentation

12.2.1 Source Code Documentation

The source code is documented in the sources themselves. Only an automated way of extracting this documentation and generating a set of HTML pages from it can be considered a solution. doxygen[4] was chosen for its compatibility with the well known JavaDoc utility and for the fact that it supports C++ and runs under Linux.

12.2.2 Technical Documentation

For the technical documentation of the project LaTeX and OpenOffice were considered. Since we've used LaTeX on various occasions before and made very good experiences with it, the choice was rather clear. LaTeX also has the advantage of being a plain text document format which can be managed by CVS. Further, it is very practical to automatically generate and include plots from gnuplot[3] and documentation parts from doxygen. For diagrams and graphics, OpenOffice was chosen due to its good graphical capabilities and its built-in EPS support (a requirement needed for integration in LaTeX documents).

[1] `http://www.opensource.org/licenses/gpl-license.html` - the GPL.
[2] `http://www.opensource.org/licenses/bsd-license.html` - the BSD License
[3] Gnuplot is a portable command-line driven interactive datafile and function plotting utility.[5]

Description	Priority	LaTeX	OpenOffice
Mandatory			
Availability on Linux		✓	✓
Desired			
Version Control	10	10	1
Automated generation and inclusion of documentation parts	8	10	3
Efficiency	8	8	6
Know-how in the team	7	6	5
Points ($\sum$ Priority · Points)		286	117

Table 5: Value benefit analysis for the technical documentation.

12.2.3 User Manual

Choosing the right way to produce the user manual was the most difficult decision when it came to documentation. The requirement to be able to produce a high quality HTML output without actually having to code in plain HTML, combined with the desire to be able to produce a printable manual from the same sources was hard to fulfill. Docbook[6] with XSL, FO was considered but seemed to be time-consuming to set up. Also the printed output was not fully satisfying and flexible enough.

ManStyle was chosen for its easy handling. Results can be produced with little effort and writing the documentation sources is straight forward. ManStyle documentations can be easily integrated into Qt applications in the form of an online help system. Alternatively the same manual can be viewed with any normal HTML browser. Last but not least ManStyle can produce PS files from the same document sources for printing the manual or converting it to PDF.

12.3 Choosing a Development Platform

Possible choices for a development platform were Windows and Linux. Both were available to all project members. The value benefit analysis in table 7 is based on personal experiences, requirements and know-how. It is not a generic, representative analysis to compare Windows and Linux. The analysis is correct for this very project and our team.

Description	Priority	**Windows**	**Linux**
Mandatory			
Availability at school		✓	✓
Availability at home		✓	✓
Desired			
Tools (CVS, ssh, shell, vim, distcc, LaTeX, ManStyle, doxygen, compiler)	8	3	10
Stability, reliability	5	6	9
Effort to set up the working environment	7	3	10
Personal efficiency when working with this platform	8	3	10
Points ($\sum$ Priority · Points)		75	195

Table 6: Value benefit analysis for the development platform.

12.4 Choosing a Programming Language

Choices for the programming language were rather limited since only Java and C++ are known to all project members and learning a new language would have been an additional risk that could have compromised the schedule.

Description	Priority	**Java**	**C++**
Mandatory			
Availability on Unices, Linux, Windows, Mac OS X		✓	✓
Desired			
Performance	5	4	8
Know-how in the team	7	6	7
Overall user experience for applications written in that language (installation, snappy GUI, ..)	8	6	10
Points ($\sum$ Priority · Points)		110	169

Table 7: Value benefit analysis for the development platform.

The decisive factors for choosing C++ were the overall user experience that we find to be generally better with a C++ application and the better performance since it could prove to be crucial for ***quaneko***.

12.5 Choosing a GUI Toolkit

Portability was the main issue when choosing a GUI toolkit for ***quaneko***. The application GUI must run under Linux but should also be available for Windows users. Coding the GUI twice was not considered to be an acceptable solution.

Qt[8] was chosen mainly for being a well designed and highly portable C++ GUI toolkit. There is also a good level of experience with Qt in our team. Qt is licensed under the GPL for Linux, Unix systems and Mac OS X. However, the Qt version for Windows is proprietary software.

12.6 Version Control System

CVS was chosen over the relatively new Subversion version control system. CVS was already known and used before by all team members. Further, CVS installations are available on public development platforms such as SourceForge.

12.7 Generation of Executables

The use of makefiles is the preferred way over using IDEs to build ***quaneko***. Makefiles can be generated with qmake (part of Qt) for a variety of platforms and compilers. qmake was chosen due to the simple configuration an because it was available with the Qt installation.

12.8 Project Organization

12.8.1 Process Model

The *quaneko* development follows an evolutionary approach. Functionality is implemented in prototypes and then transferred or rewritten into the final modules.

12.8.2 Project Responsibilities and Deliverables

Laurent Cohn

- Implementation and Design of:
  - Filter Module
  - GUI
- Documentation
  - Technical
  - User Manual
  - Web site

Thomas Jund

- Implementation and Design of:
  - Array Index
  - Inverted Index
- Testing
- Deployment
- Documentation
  - Technical
  - User Manual

Andrew Mustun

- Implementation and Design of:
  - Index handler
  - Parsing module
  - Stemming module
  - Settings class
  - Debugging class
  - API
  - CLI
- Testing
- Documentation
  - Technical
  - User Manual

12.9 Milestones

- **April, 5:** Creating and querying indexes, Functional CLI.
- **May, 4:** Fully functional CLI, API and core.

- **May, 10:** Fully functional GUI, User manual.

12.10 Schedule

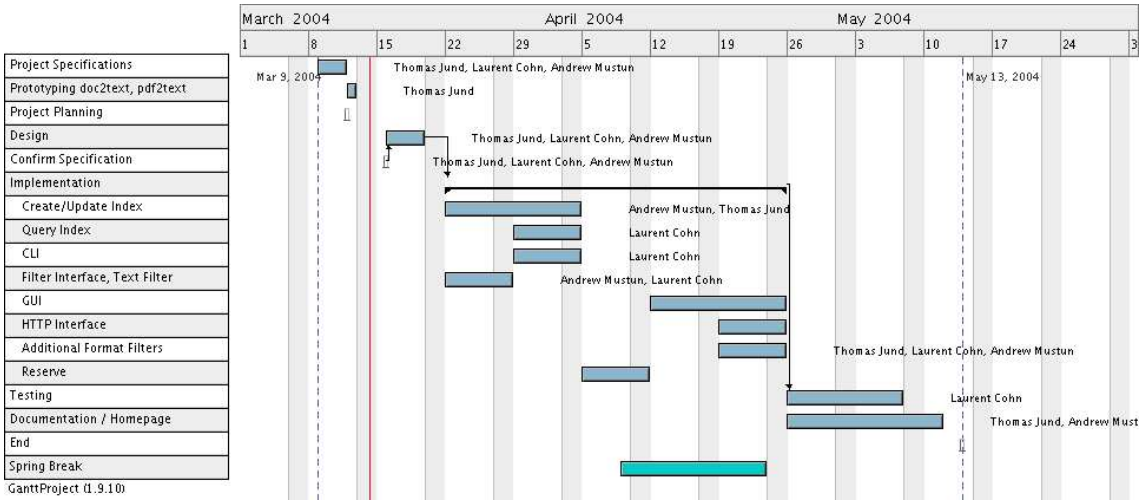


Figure 9: Planned schedule.

12.11 Management

12.11.1 Risk Management

Prototypes were implemented for modules that contain high risks:

- Stemming
- Indexing
- Calling commands (filters) from C++ under Windows

13 Conclusion

In this section we look back on the first implementation of ***quaneko*** and discuss some thoughts about the technology we have used.

13.1 Parsing

Generally we think that the approach of parsing different document types by using the plain text output of existing converters proved to be a good idea. There are many plain text filters available and configuring them for ***quaneko*** is easy enough that we can expect the users to do so.

However, the fact that the user has to install and configure something after downloading ***quaneko***, might also be a reason for some people not to use ***quaneko***.

13.2 Indexing

We think that one type of index can not offer a good enough solution for indexing both very common and very rare words. The hybrid solution we have implemented in ***quaneko*** tries to apply the right technology to each individual case.

For the example data we have used when testing ***quaneko***, we were able to show that this approach leads to better results than using one single index type.

13.3 Room for Improvements

The current ***quaneko*** implementation has some room for improvements in the following areas:

- Performance: for updates, parsing, queries
- Usability: it would be possible make adding filters easier by automate the install / setup process.
- Portability: ***quaneko*** should be ported to other platforms like Solaris and various Linux distributions.

A C++ Application Programming Interface

A.1 qnk Namespace Reference

The C++ API supports methods to create, remove, query and update an index and to parse directories (recursively) or individual files to an index.

Functions

- bool **isIndexLocked** (const std::string &name)
  Checks if the given index is locked.
- void **removeIndexLock** (const std::string &name)
  Unlocks the given index.
- vector< string > **getAvailableIndexes** ()
- vector< string > **getStemmingLanguages** ()
- vector< LDS_Filter > **getFilters** ()
- void **setFilters** (const vector< LDS_Filter > &f)
  Sets the filter vector to f.
- LDS_Filter * **getFilterForType** (const string &t)
  Searches for a filter for the given file extension.
- void **addFilter** (LDS_Filter &f)
  Adds the given filter to the filter list.
- void **removeFilter** (int id)
  Removes the filter with the given id from the filter list.
- bool **createIndex** (const string &name, const string &path, bool ignoreNumbers, const string &stemmingLanguage)
  Creates a new index with the given name (id) at the given path.
- bool **removeIndex** (const string &name)
  Removes the index with the given name.
- bool **validateIndex** (const string &name)
  Validates the index with the given name.
- vector< string > **queryIndex** (const string &name, const string &word, LDS_Progress-Listener *pl)
  Searches for the given word in the given index.

- vector< string > **complexQueryIndex** (const string &name, const string &expr, LDS_ProgressListener *pl)

  Searches for the given expression in the given index.

- bool **updateIndex** (const string &name, LDS_ProgressListener *pl)

  Updates the given index.

- void **updateFile** (const string &name, const string &file, LDS_ProgressListener *pl)

  Updates the given file or directory in the given index.

- bool **parseFiles** (const string &name, const vector< string > &files, LDS_ProgressListener *pl)

  Parses all files and directories in the given index.

A.1.1 Detailed Description

The C++ API supports methods to create, remove, query and update an index and to parse directories (recursively) or individual files to an index.

All API functions are state-less.

Author:
Andrew Mustun

A.1.2 Function Documentation

A.1.2.1 void qnk::addFilter (LDS_Filter & *f*)

Adds the given filter to the filter list.

Parameters:
f The filter to add.

A.1.2.2 vector<string> complexQueryIndex (const string & *name*, const string & *expr*, LDS_ProgressListener * *pl*)

Searches for the given expression in the given index.

Parameters:
name Unique name of the index.

expr The expression to search for.

pl Pointer to an object that is interested in progress events.

A.1.2.3 bool createIndex (const string & *name*, const string & *path*, bool *ignoreNumbers*, const string & *stemmingLanguage*)

Creates a new index with the given name (id) at the given path.

'path' must exists and be writable. A subdirectory 'name' will be created in 'path'. All files belonging to the index and its internal structure are created in that subfolder.

Example: name = "TestIndex", path = "/tmp". The index files are created in /tmp/TestIndex

Parameters:

name Index name. Must be unique for one user on a system.

path Path where the index is stored.

ignoreNumbers true: Don't index numbers

stemmingLanguage Language used to do stemming or empty to disable stemming. Supported languages are: da, de, en, es, fi, fr, it, nl, no, pt, ru, sv.

A.1.2.4 std::vector< std::string > qnk::getAvailableIndexes ()

Returns:

A vector of the names of all currently available indexes.

A.1.2.5 LDS_Filter* getFilterForType (const string & *t*)

Searches for a filter for the given file extension.

Parameters:

t The type (file extension). E.g. "html", "pdf", ..

Returns:

The first filter that is found for handling the given file extension or NULL if no filter can be found.

A.1.2.6 std::vector< LDS_Filter > qnk::getFilters ()

Returns:

A vector of all available filters.

A.1.2.7 std::vector< std::string > qnk::getStemmingLanguages ()

Returns:

A vector of languages available for stemming.

A.1.2.8 bool qnk::isIndexLocked (const std::string & *name*)

Checks if the given index is locked.

Parameters:

name Index name. Must be unique for one user on a system.

Return values:

true The given index is locked.

false The given index is not locked.

A.1.2.9 bool parseFiles (const string & *name*, const vector< string > & *files*, LDS_ProgressListener * *pl*)

Parses all files and directories in the given index.

Directroies are recursively parsed.

Parameters:

name Unique name of the index.

files Vector of files and directories to parse into the given index.

pl Pointer to an object that is interested in progress events.

A.1.2.10 vector<string> queryIndex (const string & *name*, const string & *word*, LDS_ProgressListener * *pl*)

Searches for the given word in the given index.

Parameters:

name Unique name of the index.

word The word to search for.

pl Pointer to an object that is interested in progress events.

A.1.2.11 void qnk::removeFilter (int *id*)

Removes the filter with the given id from the filter list.

Parameters:

id ID of the filter to remove.

A.1.2.12 bool removeIndex (const string & *name*)

Removes the index with the given name.

Parameters:
name Unique name of the index.

Return values:
true Success.
false No such index found.

A.1.2.13 void qnk::removeIndexLock (const std::string & *name*)

Unlocks the given index.

Parameters:
name Index name. Must be unique for one user on a system.

A.1.2.14 void setFilters (const vector< LDS_Filter > & *f*)

Sets the filter vector to f.

Parameters:
f The new filter vector.

A.1.2.15 void updateFile (const string & *name*, const string & *file*, LDS_ProgressListener * *pl*)

Updates the given file or directory in the given index.

Entries are updated if a previously indexed file was removed or has changed. For directories, new files in the directory are added and obsolete files are removed.

Parameters:
name Unique name of the index.
file The file or directory to update.
pl Pointer to an object that is interested in progress events.

A.1.2.16 bool updateIndex (const string & *name*, LDS_ProgressListener * *pl*)

Updates the given index.

Entries are updated if a previously indexed file was removed or has changed. In indexed directories, new files are added.

Parameters:
name Unique name of the index.
pl Pointer to an object that is interested in progress events.

A.1.2.17 bool validateIndex (const string & *name*)

Validates the index with the given name.

Parameters:

name Unique name of the index.

Return values:

true Success.

false Index not valid (look at the application output for more information).

B Glossary

Array Index: An Array Index is a bitmap in which a Bit that is turned on (1) means that the file in this column contains the word in this row. See also page 14.

Direct Index: If a word occurs in only one file, its Word ID equals the File ID of that file. We refer to this as "direct indexing". See also page 13.

File Register: A list of absolute paths or URIs of all files that have been indexed.

Filter: A configured program which can convert files in a non plain text format into plain text. ***quaneko*** uses filters to index and preview non plain text files.

Full Word ID: A combination of an Index ID and a Word ID. Each such combination is unique for one index.

Index ID: The ID of an index. Each index has a unique ID. ID -1 is reserved for the Direct Index. Index -2 is reserved for the Array Index. All other IDs are positive IDs for Inverted Indexes.

Inverted Index: "An index into a set of texts of the words in the texts. The index is accessed by some search method. Each index entry gives the word and a list of texts, possibly with locations within the text, where the word occurs." [7] Inverted Indexes are used to lookup a key (file name) based on a value (word) rather than the other way around, hence 'inverted'. See also page 13.

Stemming: "A stemmer is a program or algorithm which determines the morphological root of a given inflected (or, sometimes, derived) word form – generally a written word form. A stemmer for English, for example, should identify the string 'cats' (and possibly 'catlike', 'catty' etc.) as based on the root 'cat', and 'stemmer', 'stemming', 'stemmed' as based on 'stem'." [2]

Word ID: The ID of a word in an index. This ID is unique for a particular index but the Word ID alone is not unique in the word register. The Word ID and Index ID together build the Full Word ID which is unique in the Word Register.

Word Register: An alphabetically sorted list of all words which can be queried.

C References

[1] *Quaneko User Manual, 2004*,
T. Jund, L. Cohn and A. Mustun, ZHW, Winterthur

[2] *Wikipedia*,
`http://www.wikipedia.org`

[3] *Snowball*,
`http://snowball.tartarus.org`, Dr. Martin Porter

[4] *doxygen*,
`http://www.doxygen.org`, Dimitri van Heesch

[5] *Gnuplot*
`http://www.gnuplot.info`

[6] *DocBook*,
`http://www.oasis-open.org/docbook`, OASIS

[7] *National Institute of Standards and Technology*,
`http://www.nist.gov`, Paul E. Black

[8] *Qt - Multi-platform, C++ Application Framework*,
`http://www.trolltech.com`, Trolltech AS

[9] *The C++ Programming Language, Third Edition*,
Bjarne Stroustrup, Addison Wesley

[10] *Valgrind*,
`http://valgrind.kde.org`

D Index

E About the Authors

Thomas Jund, Andrew Mustun and Laurent Cohn are undergraduate students at the Zurich University of Applied Sciences Winterthur (ZHW).

Thomas Jund received his certificate of ability as electrical draughtsman from the Swiss Federation in 1998. He worked in the area of computer aided design (CAD), documentation publishing and technical documentation services for a company specialized in building automation. In addition to his interests in Networking and CAD, he has also been active in web design and web programming.

The implementation of the inverted index and the array index as well as testing are the main activities covered by Thomas in ***quaneko***. He collaborated on parts of documentation and on the port to Windows.

Andrew Mustun has been heavily involved in various large and middle sized open source projects since 1995. He is the founder of well known projects like QCad, ManStyle and dxflib. Andrew's main interests are in the CAD / CAM area as well as in documentation and information management. He worked for 14 months on a collaboration solution for a UK based company which is specialized in document management for the building industry.

In ***quaneko***, Andrew was responsible for the parser, the stemmer module and parts of the indexing module as well as the CLI and the API of ***quaneko***. Further, his experience in developing multi-platform solutions contributed to keep ***quaneko*** portable.

Laurent Cohn has his main interests in web programming and web design. He has designed several web pages for companies and well-known non-profit organizations. Laurent has a lot of programming experience with PHP and MySQL. His experience with working with graphics and his intuition for good design, combined with his C++ skills make him a great GUI programmer.

In ***quaneko***, Laurent programmed the Filter Library and designed the GUI. He has also designed the ***quaneko*** logo printed on the title page and the ***quaneko*** program icons. Being the only active Windows user on the team, his Windows expertise kicked in handy when porting ***quaneko*** to Windows.