Indexlib
Report on the Implementation of an Indexing Library

Luís Pedro Coelho
luis@luispedro.org

Abstract

With the widespread use of search engines and considering the success of gmail as a search based email interface and the number of requests of user for fast text and particularly email search, there is definitely a need for indexing email for fast access and search.

This need was badly provided for by open-source email clients. Therefore, the purpose of this project was to provide email indexing for kmail, an open-source email client.

As underlying indexing structure, inverted files were used. The project is broadly divided into a core library and its use by kmail.

1 Problem

Email indexing is a very desirable feature of an email client. This allows one to construct search-based interfaces, be they built around concepts such as *virtual folders*[1], search engine-like message search and filtering or others ...

This is all possible, of course, without text-indexing and has been implemented in all major mail clients for some time, but the performance is so poor that the value of these features is much diminished.

2 Semantic Details

2.1 Words

The first step in indexing a file is to break it into words. The library allows a way to change the definition of word easily, requiring no more than that it be a string of non-null bytes[2].

Right now, the following, admittedly European-centric, interpretation is used: A word is a string of alphabetic characters. These characters are then *normalized* which involves the removal of diacritics (like the conversion from $\acute{E}$ to E or $\c{C}$ to C) and the conversion to upper case.

[1] aka *search folders*, these are analogous to *views* in databases, in that they look like traditional folders, but contain a group of messages fulfilling a certain criteria which are physically stored in other *real* folders. Unlike traditional folders with automatic filters, these allow a message to be present in more than one folder at once

[2] Using the UTF-8 encoding, this allows the full Unicode set to be used.

2.2 Use of Quotes

The basic search method works as follows: searching for *one two* will return elements where both the words *one* and *two* appear even if they do not appear consecutively. The use of quotes around the expression, allows one to search for the exact expression.

This is implemented by keeping a copy of the original text, the general inverted file structure is used to return a list of files where the search items appear ch and then, exact case-insensitive matching is done using *shift-and*. The need for keeping a copy of the original text, could, of course, be removed if there was sufficient integration with the application to allow this process to be done over the original text without a copy.

3 Design

3.1 Underlying structure

The system developed is an implementation of *inverted files*. Inverted files were chosen as having a right balance between disk-space usage, search time, updatability and feature set. Also considered were suffix trees, which have a disk-space cost which is too large and suffix arrays which are difficult to update.

3.2 Design Decisions

The implementation is divided into a *core library* and the *kmail integration glue code*. The core is the major part of this project and where the bulk of this report will be focused.

3.3 From the Ground Up

A new implementation was written from the ground up even though several alternatives existed. These are however, not stand-alone library implementations and rely on databases and often other services, even webservers. Therefore, an application which wants to use indexing will need to open a connection to these services and negotiate with them. Some efforts are under way to hide this complexity behind simple Application Programming Interfaces, but these do not take away the need to run a separate server with a potential impact on startup time and may incur in a performance penalty as possibly leading to maintainability issues as the application needs to deal with an external service whose version and settings it does not control.

This implementation achieves the following:

Ease of use To use it, one just needs to *#include ¡index/index.h¿* and link in the appropriate library (using *-lindex*, or the platform's equivalent)

No external run time dependencies No external service needs to be installed. The library is sufficiently small to be bundled with the application if desired.

Limited compile time dependencies Besides the basic C++ environment, this library needs only boost's library headers. These are currently bundled with indexlib. There is no need for any of boost libraries to be installed, it is a header only dependency[3].

No startup penalties As will be seen, only a small startup time is needed.

The code was implemented in C++. This language also allows C bindings to be easily written. C being the native tongue of Unix, this allows almost universal access to the features.

3.4 Library Interface

The Library has a simple interface. Forgetting language details, here is the main interface

index* create_index(string name) Returns a new index identified by name. Name should be a filesystem path.

void index::add(string docname, string doc) Add the file doc to the index, calling it *docname*.

void index::remove(string docname) Remove the file *docname* from the index

vector<string> index::search(string pattern) Returns the set of files where pattern occurs.

3.5 Use of Memory Mapped Files

Memory-mapped files (or mmapped files, after the system call *mmap()*) is an operating systems concept which allows one to use a disk-based file in the same manner one uses memory. The whole implementation will be based around memory-mapped files. This has the following advantages:

- Fast startup time. Since there is no load time from the disk to the memory on load, there is no time penalty on application startup related to converting from a disk to a memory format.
- Intelligent page-in and page-out algorithms. Since the use of mmapped files relies on the base operating system algorithms for on-demand paging, these algorithms are likely to be better than any that might be developed by this project.

This decision forces the manual management of memory and the reimplementation of functions equivalent to *malloc()* and *free()*. These are reimplemented on the basis of free-lists. These are not generic implementations, but rely on specific knowledge about the types used. This is mostly a space optimization to avoid using extra accounting overhead, since they could easily be adapted to allocate anything.

[3]This may seem odd to traditional C programmers, but in modern C++ a lot of things are implemented using templates which mangles the difference between implementation and interface.

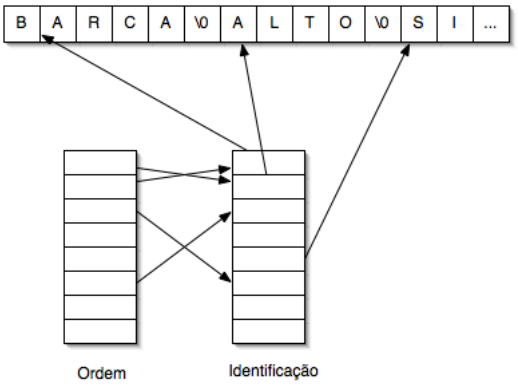

Figure 1: Term List

4 Implementation Overview

4.1 Term Lists

Term lists are implemented as shown in Figure 1.

The string data is kept in a contiguous array of null-terminated strings. There are also two auxiliary arrays. The first one, marked *Identification* contains an indeces into the string data, to the start of strings. By transversing this array, we can transverse the array of strings. To add a string, one needs only to add it to the end of the string data and add a pointer to it in the first auxiliary array. This also keeps a unique numeric identifier for each string in the form of its order of insertion which is the offset in this *identification array*.

To speed up the search in this set of strings, a second auxiliary array is used which is ordered by the lexicographical order of the strings. When a string is added, that might mean that a new entry needs to be added to this array somewhere inside it as well, which results in all the other entries being down shifted (except in the unlikely case that the string is added to the end of the array).

Removing a string is a potentially expensive operation. Also, removing a string might change the *ids* for all the strings below it in the *identification array*, which would force other changes to the index to maintain consistency since (as will be illustrated below) other structures rely on this value. Therefore, the removal of a string is implemented simply by deleting its entry in the *ordered array*, so that it will not be found anymore. Later, an expensive *cleanup operation* over the whole index will remove all the stale data and readjust the changed references.

To look up a word, we need to use binary search over the *ordered array*. A use of a trie-like structure might speed things up, but the speed obtained by binary search is already good enough and such a structure might be added on top of the present implementation at a later time.

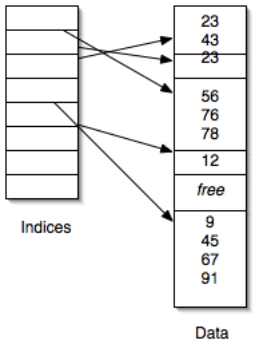

Figure 2: File Reference Lists

4.2 Index Lists

At the heart of the inverted file structure is an array of lists of file identifiers. This basic structure is shown in Figure 2.

Externally each file is identified by a string. Internally, this string is kept in an array and each file is identified by its index into this array of document names in much the same way as terms can be mapped to numbers using the term list above.

To look up the files where a certain word appears, we use its numeric id as an index into the first level array and follow the pointer there to the list of files where the string appears.

Since the file list might grow as new documents are added, it might overfill the space allocated to it and might need a reallocation.

Over this basic structure, three simple space optimizations were implemented. First, a special case was made for one element lists. Secondly, file references are 32 bit numbers, but, where possible, we only save 8 bit deltas which saves space in very dense file lists. Thirdly, a general purpose algorithm is run over the lists, using zlib, the library implementation of deflate, the algorithm used by *gzip*.

4.2.1 One Element Lists

Some words appear in only one document. Treating this as a special case allows a simple optimization: instead of allocating a vector to hold this, use the space where the pointer was held to save the reference there directly as shown in Figure 3.

Using positive and negative numbers allows one to distinguish between the two types of data kept in the same array: pointers and one-element lists.

This simple optimization carries neither run-time nor memory penalties and costs just a small complexity penalty in the implementation.

4.2.2 8-bit deltas

Some words appear very often, leading to very large lists. Given that element references are assigned sequentially, the values are always increasing. Therefore,

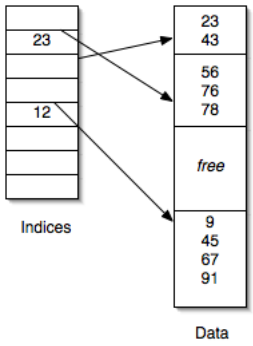

Figure 3: File Reference Lists With One Element Optimization

it is reasonable to assume that, for very dense lists, the difference between consecutive references will be small enough to fit in an eight bit number. Switching to eight-bit numbers and storing differences instead of absolute values allows a space saving. Of course, eight bits may not be enough in certain situations and there is a need to store a longer sequence: a zero byte, followed by the full four byte number.

This optimization carries both a slight run-time cost and a slight complexity penalty regarding the code to implement it.

4.2.3 ZLib Compression

A more general compression scheme is also implemented. This consists of using a general purpose compression algorithm (in this case, deflate, implemented by the zlib library), to compress and uncompress the data on a *as needed* basis. Due to the way that the deflate algorithm works, we compress the data by blocks (we chose a block of size $2^{12} = 4\,096$ which is the natural page size of the test machine).

This scheme achieves very good disk space compression at the cost of some run time penalty as well as RAM costs. This RAM cost might be tunable by specifying a write-back scheme. In the current implementation, the system writes everything back at the end of the session.

5 Limitations

5.1 Language Dependency

The first step in indexing is to break up the input text into words and these are turned into their uppercase version. This simple phrase hides a mountain of assumptions on the structure of the language. Defining how these concepts map into non-European languages are beyond this project.

5.2 Error Recovery

At the moment, there is no effort taken to try and recover an index recovered from a damaged form (be it from a non-correct exit, ie. crash, or other

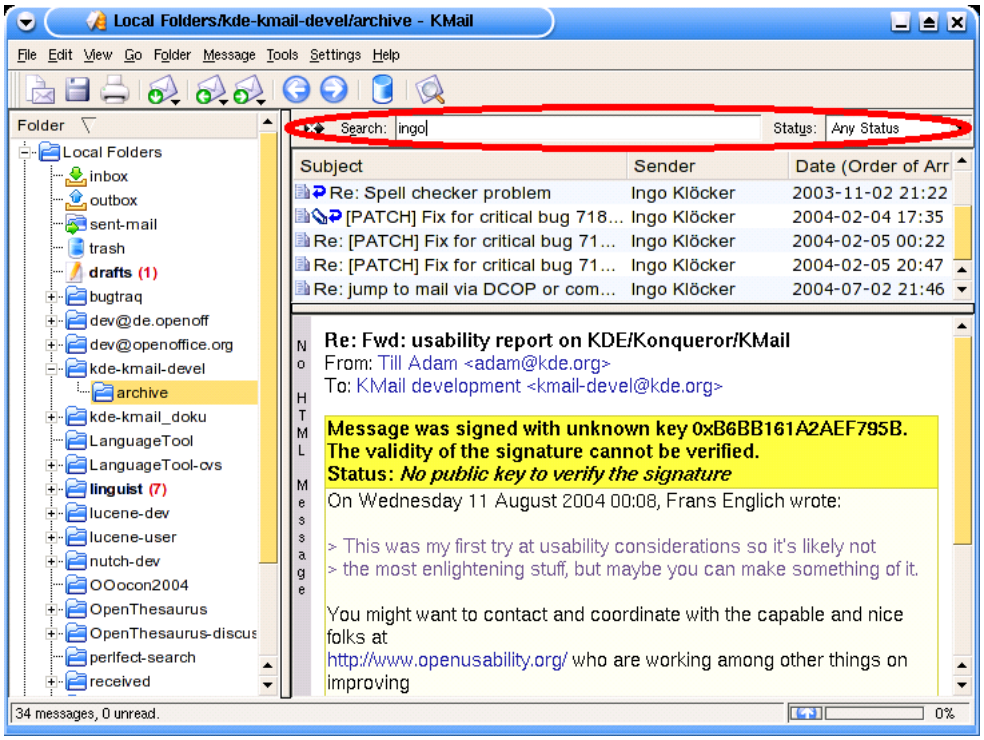

Figure 4: KMail Screenshot

problems).

6 KMail Integration

6.1 User Interface

The user interface is very simple. We reused an interface element which was already present as outlined in Figure 4. However, until now, this search bar filtered only on the headers and not on the contents of the message.

6.2 Integration Method

Each mailbox is indexed separately as the search bar makes sense for the current mailbox only. There is a script to index existing messages. New messages are handled by a filter on incoming email.

6.3 Upstream Integration

As of early March 2005, the whole KDE Project is in a freeze due to the imminent release of a new version. This means that it is impossible for this project to merged into the official tree before the new version is released and the code is *unfrozen*.

Disk Space (total)	389 MB
Disk Space (text)	7 MB
Number of messages	9,502
Number of mailboxes	15

Table 1: Size of Test Mailboxes

However, there is a great desire for this feature. Particularly, Don Sanders, one of the kmail maintainers, has shown great interest in this.

7 Results

7.1 Data Set

This are the results obtained over my private email. It is a small data set and I do not argue that it is a significant sample, but it is enough to provide an indication of the expected values.

In Table 1 the main values of this dataset are presented. Note that these are aggregate values, the messages are divided among 15 different mailboxes. The huge difference between the *total size* and the *text size* is due to attachments and headers. Also, the total space is the space used by the maildir representation where each individual message is a single file where the size is normally rounded up to a multiple of a page, while the text size was measured as only the text size.

7.2 Disk-space Usage

Using zlib compression, the index size is 3.5MB. Without this compression, it expands to 4.9MB. This is under the text size which is a typical value for inverted files.

7.3 Time

Indexing the mails takes 40 minutes if zlib compression is being used. Without this compression the time drops to 15 minutes. Both these times include the time taken for parsing the text of the raw messages, extracting the text only parts including any character encoding changes if necessary (email messages are normally sent in pure ASCII and non-ASCII characters need a special encoding). Given this large differential and the modest space savings, using this compression scheme is something which should remain an optional feature.

Search is very fast. Searching for any “algoritmo”, “inesc” or “pre” took less than one second (this last entry generates a large result set because it is prefix to a large number of words). Using zlib compression does not seem to affect these results.

The tests were run in an Apple Powerbook machine with a 1.3 GHz G4 Processor.

8 Future Work

The current integration with kmail while functional, has certain important limitations, especially on the need for manual installation. This work should proceed with the help of the kmail team.

As it stands, the code does not attempt to limit the damages in case of a crash. At least limiting and detecting the corruption of the index is a need.

A lock mechanism for access to the index is also a necessity for widespread adoptions. Kmail already limits itself to one instance accessing the mailboxes at any given time, so in this problem setting we can piggy-back on this existing mechanism.