

Master Thesis

The Holumbus Framework

**Creating scalable and highly customized crawlers
and indexers.**

Submitted on:
September 4th, 2008

Submitted by:
Sebastian M. Schlatt
Dipl.-Wirtschaftsinform. (FH)
Parkallee 11
22926 Ahrensburg, Germany
Phone: +491638392847
E-Mail: sebastian@schlatt.com

Supervised by:
Prof. Dr. Uwe Schmidt
Fachhochschule Wedel
Feldstraße 143
22880 Wedel, Germany
Phone: +494103804845
E-Mail: uwe@fh-wedel.de

The Holumbus Framework

Creating fast, flexible and highly customized search engines with Haskell

Master's Thesis by Sebastian M. Schlatt

Abstract

Full text search is a key technology for today's information society. The vast accessible amount of information available in public and also privately used media creates the need for efficient search engine technologies. The Holumbus framework helps users with the creation of highly customized search engine applications over structured data sets. This work covers the development and implementation of the crawler and indexer modules of Holumbus.



Copyright © 2008 Sebastian M. Schlatt

This work is licensed under the Creative Commons Attribution-NonCommercial 2.0 Germany License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.0/de/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Layout done with the help of Timo B. Hübel's template, \LaTeX , KOMA-Script and \BibTeX .

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Integration	2
1.3	Outline	2
2	Fundamentals	4
2.1	Search Engine Structure	4
2.2	Forward Index	5
2.3	Inverted Index	6
2.4	Index Construction & Maintenance	7
2.4.1	In-Memory Inversion	7
2.4.2	Sort-Based Inversion	8
2.4.3	Merge-Based Inversion	8
2.4.4	Index Maintenance	9
2.5	Parallel and Distributed Crawling and Indexing	10
2.6	Case Folding, Stemming and Stop Words	12
2.7	Duplicate Document Detection	15
3	Analysis	16
3.1	Requirements	16
3.1.1	Fast Search Operations	16
3.1.2	Prefix Search	16
3.1.3	Scalability	17
3.1.4	Configurability	17
3.2	Basic functionality	20
3.2.1	Crawler State	20
3.2.2	Caching Temporary Data	20
4	Implementation	22
4.1	Environment	22
4.2	Framework	22
4.3	Crawler	25
4.3.1	Configuration and Internal State	25
4.3.2	Algorithm	27
4.4	Indexer	28
4.4.1	Configuration	28

4.4.2	Index Construction	31
4.5	Parallelization and Distribution with MapReduce	35
4.5.1	Memory-based Parallel Implementation	35
4.5.2	Parallel Implementation with Hard-disk-stored Intermediary Data	38
4.6	Monadic Indexes	39
4.6.1	Holumbus.Index.Inverted.OneFile	41
4.6.2	Holumbus.Index.Inverted.Persistent	41
4.6.3	Holumbus.Index.Inverted.Database	42
4.6.4	Synopsis	42
5	Evaluation	43
5.1	Memory Usage	44
5.2	Query Processing	45
5.3	Index Construction	46
5.4	Janus Webserver	47
6	Example Applications	48
6.1	Hayoo! Haskell API Search	48
6.2	FH Wedel	51
6.3	CGI-based Search Engine	52
6.4	Sitemap Tool	52
7	Conclusion	54
7.1	Synopsis	54
7.2	Implications for distributed MapReduce	56
7.2.1	Intermediary Data	56
7.2.2	Shared Dictionary	57
7.2.3	Distributed Reduce-phase	57
7.3	Future Work	58
7.3.1	Current Projects	58
7.3.2	Support Different Filetypes	58
7.3.3	Evaluate More Index Types	59
7.3.4	Index Maintenance	59
7.3.5	Stemming	60
7.3.6	Sophisticated Index Merging	60
	Bibliography	61
	Affidavit	64

List of Figures

2.1	Typical search engine architecture (from [BYRN99])	5
4.1	Holumbus framework architecture	23
4.2	Holumbus.Index.Inverted.Memory	39
5.1	Memory usage of inverted indexes with different implementations II .	44
5.2	Index construction time with different thread count in seconds	45
5.3	Index construction time with different thread count in seconds	46

List of Tables

2.1	Document-word-matrix (Forward Index)	5
2.2	Input data for MapReduce job “word count”	11
2.3	Intermediary data for MapReduce job “word count”	12
2.4	Result for MapReduce job “word count”	13
4.1	MapReduce types for index construction	33
4.2	Basic Holumbus index construction algorithm	35
5.1	Query processing benchmarks (702 queries, 571535 unique words) . .	46
5.2	Time (in seconds) to construct an index over precomputed HolDocu- ments on a local hard disk with different thread counts	47
6.1	FH Wedel contexts	50
6.2	Fh Wedel contexts	51

Listings

2.1	Inverted index in Haskell syntax	6
2.2	Basic MapReduce interface	10
4.1	Crawler configuration and state	25
4.2	Indexer configuration and state	28
4.3	Context configuration	29
4.4	Memory-based inverted index	31
4.5	Inserting words into an inverted index	32
4.6	Inserting occurrences into an inverted index	32
4.7	Basic MapReduce interface	33
4.8	Basic MapReduce interface	34
4.9	Basic MapReduce interface	36
4.10	Monadic index interface	40

1

Introduction

1.1 Motivation

The amount of available information is growing rapidly in the information age. [Gan08] is estimating that 281 exabytes of information were available in 2007 and predicting further growth to at least 1,800 exabytes in 2011. Most of the information is unstructured and distributed over different media.

Search engines have become essential tools in today's information retrieval since the classic concept of tables of contents is not applicable to many information sources. To retrieve information from large data sets, data structures that allow the fast retrieval of highly relevant documents in very little time have to be used. Users of internet search engines for example would not allow a search engine to process a query for more than a few seconds before closing the site and using another search engine.

Internet search engines usually aim to cover as much information as possible that is publicly available on the internet and index every word they can retrieve from a document. In the course of providing better search results, a common concept is that words in certain scopes are rated more important than others. For example the words in the title-tag of a html document are considered to be very important since a title usually describes the page's content accurately. Nowadays many commercial search

engine companies offer customizable search technology with restricted scope which can be used on company websites (e.g. [yah]).

Many websites offer search applications, that cover only the contents of the pages of one web project. For companies that sell their products over the internet, this can be the most important part of the web site since it helps customers find the products they want - or other products that the company wants them to find.

The goal of this work is to develop a highly customizable and configurable framework for the creation of crawlers and indexers to collect data for search engine applications.

1.2 Integration

This thesis describes the development of parts of the Haskell search engine framework *Holumbus*. The framework is designed for the implementation and comparison of different datastructures on which full text searches can be performed.

While the implementation of the searching part is [Hüb08] and more information about distributed computations will be available in [Sch08], this thesis describes the design and implementation of flexible and customizable crawlers and indexers with the *Holumbus* framework. Additionally, the already implemented inverted index is examined and an alternative implementation with less memory usage at the cost of longer query execution times is introduced and the performance of both index types is measured and compared.

1.3 Outline

In chapter 2 basic information about search engines in general, the construction and maintenance of indexes and techniques to reduce the memory usage of indexes is given. Because many *Holumbus*-related fundamentals have already been explained in [Hüb08], some parts are kept brief to avoid unnecessary repetitions.

Chapter 3 describes the requirements that were defined before and during the development of the *Holumbus*-searchengine modules. Additionally some considerations about crawler- and indexer-library internals are presented.

The next chapter 4 describes the implementation of the crawler and indexer libraries together with the necessary utility modules that were developed. The later-added monadic interface for indexes and three index types that were developed and tested with the new extended interface are also described.

In Chapter 5 some tests are described that were carried out to evaluate the implemented libraries. This includes also the libraries that are described in [Hüb08] and could not be tested with larger data sets as the crawler and indexer libraries were finished later than the search libraries and thus only smaller indexes, that were built with beta-libraries were available.

Chapter 6 deals with the example applications that were built to test and demonstrate the capabilities of the Holumbus framework.

In Chapter 7 conclusions that could be drawn from the development of the libraries are expressed. Emphasis is put on implications for the implementation of a distributed MapReduce framework since the development is already in progress and some basic problems that will have to be solved became obvious during the implementation of the searchengine libraries. The chapter also contains a section that lists some ideas, how the Holumbus libraries can be extended in the future. In the last chapter, a summary of the work on the Holumbus-searchengine libraries is given.

2

Fundamentals

Indexing large data sets leads to two of the most important problems in computer science: complexity of data structures and algorithms concerning memory usage and processing time. If a search engine is designed to cover large sets of documents, the indexing should lead to a memory reduction without causing long query processing times. Search engine users usually expect a search engine interface to present the results within few seconds.

2.1 Search Engine Structure

A common search engine architecture can be seen in [2.1](#). The underlying data is stored in the central index which is built by an indexer. Users' queries entered in a search engine interface are processed by the query engine which extracts the pages that seem to fulfill the information needs of the user based on the entered query best.

The pages which have to be added to the index are identified by a crawler.¹ Even though it is possible to combine crawling and indexing in one organizational entity, searchengines usually split these subsystems.

In reality, the subsystems of a search engine can be arbitrarily complex. A crawler is usually at least a multithreaded if not distributed program. This can help to reduce

¹also: bot, spider, harvester

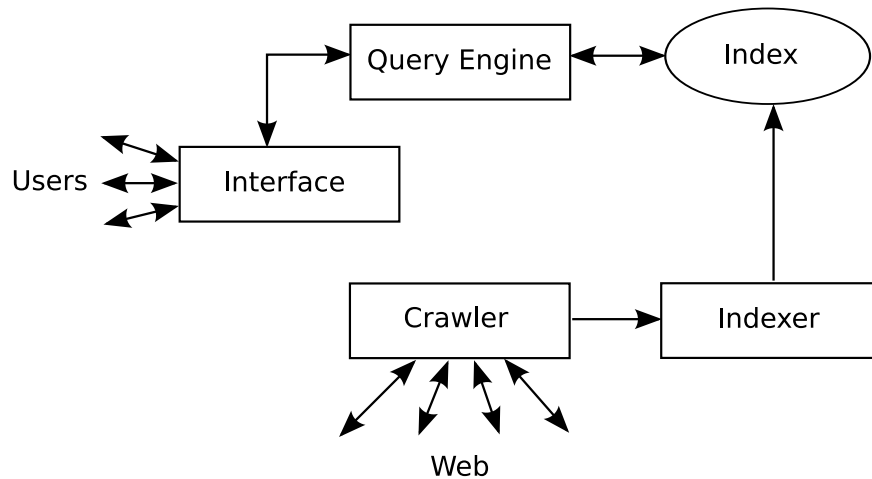


Figure 2.1: Typical search engine architecture (from [BYRN99])

the impact of the bottleneck “data access over network connections”. Indexers can consist of several tasks that are executed sequentially. For example, the parsing of input documents and creation of temporary data structures can be the first phase, while the second phase can consist of rearranging the temporary data to ease the third phase, the “real” index construction.

2.2 Forward Index

A document set that has to be indexed can be represented as a matrix in which each row corresponds to a document and each row represents one word that appears in any document. The function $f(d_i, w_j)$ represents data that is needed for the search

	w_1	w_2	...	w_j
d_1	$f(d_1, w_1)$	$f(d_1, w_2)$...	$f(d_1, w_j)$
d_2	$f(d_2, w_1)$	$f(d_2, w_2)$...	$f(d_2, w_j)$
...
d_i	$f(d_i, w_1)$	$f(d_i, w_2)$...	$f(d_i, w_j)$

Table 2.1: Document-word-matrix (Forward Index)

engine purpose. If it is only intended to identify documents that contain a word, it can be *True* if document d_i contains w_j and *False* if not. For better ranking possibilities it could also be an Integer value that represents the the frequency of w_j in d_i . To enable

phrase searches and improve the ranking², it can be a set of positions on which w_j occurs in d_i .

To invert the text, the matrix has to be transposed so that the table rows contain the words and can be accessed during query processing. For larger datasets containing hundreds of thousands of documents and millions of unique words, this matrix can become very large and the index construction a time- and memory-intensive task.

Of course, human-readable documents are not presented in a matrix-style as in Table 2.2. But a data representation like this is actually often used as an intermediate result during the inversion of documents, like for example described in [BP98]. This kind of intermediate result is often called forward index since it is different from the original document presentation, but the terms are still sorted in a forward manner.

2.3 Inverted Index

There are several different types of indexes that have been tested over the years. For natural languages, inverted indexes perform best [ZM06; BYRN99; WMB99] and are the standard choice for search engine implementations. They can be used for the most common search types and are applicable for very big data sets. Basically an inverted index consists of a set of words and lists of occurrences for each word. In the easiest case an occurrence is an identifier of a document. It can be enhanced with the positions of the words in the original documents which is needed for the processing of phrase queries³.

```

type Inverted      = Map Word Occurrences
type Occurrences = Map DocId Positions
type Positions    = IntSet
type DocId        = Int
type Word         = String

-- short version
type Inverted'    = Map String (IntMap IntSet)

```

Listing 2.1: Inverted index in Haskell syntax

²For example, words that occur in the beginning of a document could be considered to be more important than those occurring at the end

³A phrase query searches multiple words in the correct order. Many search engine interfaces indicate phrase queries by wrapping phrase searches in quotation marks.

Inverted indexes offer good possibilities for compression to reduce the index size. Most techniques used for index compression base on the fact that the inverted occurrences lists can be sorted. This can be done for the lists of documents in which a term occurs as well as for the lists of positions on which a term can be found in a document. By replacing the real document ids or positions by the difference to the ancestor in the list, no information is lost but the values that have to be stored can be significantly smaller and thus they can be stored with less memory wastage. [WMB99] provides detailed explanations of concrete techniques for index compression and other index types. Further information is also available in the related work [Hüb08].

2.4 Index Construction & Maintenance

According to [WMB99] index construction / text inversion “is one of the most challenging tasks to be faced when a database is built”. There are three distinguishable basic methods for index construction: In-Memory Inversion, Sort-Based Inversion and Merge-Based Inversion. The following explanations for the three techniques are mainly based on the brief yet vivid descriptions in [ZM06]. More detailed information is given in [WMB99], where also index construction using compressed data is discussed.

For many index types, the dictionary, which stores the index terms and is the structure upon which searches are performed, and the inverted lists are separated from each other and stored in separate data structures or files. The dictionary usually contains pointers or other data that determines how an inverted list for a certain term can be accessed. The descriptions of the inversion techniques are all assuming that the constructed indexes uses this way of separation.

2.4.1 In-Memory Inversion

Index construction using in-memory inversion consists of three steps⁴. At first, the document collection is processed and for each term the document frequency is calculated. The length of the inverted lists - mainly influenced by the count of documents in which the term occurs - is determined. After that, the memory needs for the index construction is known; the memory is allocated and pointers to the start addresses of the lists are generated. As third step, the document collection is processed a second time. For each term in each document an entry determining the document is appended to the inverted list of the term. This again leads to a simple index type but adding for

⁴Writing the index to a file is an optional fourth step.

example term positions to the index and integrating it into the described algorithm can be done straightforwardly. [ZM06] specifies the memory needs for in-memory inversion to be 10-20% greater than the final memory usage of both dictionary and occurrence lists.

For data collections where the indexes exceed the available memory, the in-memory technique can be easily modified to store the index data on disk. Instead of allocating main memory to store the occurrence list, a binary file can be used and prepared in the same way. The occurrence lists can then be computed in a sequential manner where every construction step only processes as many terms as will fit into the main memory. The overhead of on-disk-indexes compared to in-memory-indexes is very little, even for multi-gigabyte input data according to [ZM06].

2.4.2 Sort-Based Inversion

Processing the documents twice as with the in-memory inversion technique adds additional costs to the indexing process as fetching and parsing account for about one half to two thirds of index construction time according to [ZM06]. Using sort-based inversion, the original documents are only processed once and an intermediate forward index is stored in a binary file. This forward index is sorted by documents and then becomes sorted by terms as the inversion is processed. When an in-place sorting algorithm is used, the memory overhead again can be held as little as about 10% of the final index's size. The limiting factor - as for file-aided in-memory inversion - is the dictionary size since it has to be held in the main memory during the whole index construction process.

2.4.3 Merge-Based Inversion

The third basic index construction variant is merge-based inversion. It is applicable for larger input data sets and it offers good opportunities for parallel and distributed indexing. Using merge-based inversion, the input documents are split into subsets. After that, subindexes are computed over all subsets using either in-memory or sort-based inversion. When one subindex has been computed, it is flushed to the hard disk. Since this intermediary data structure it is not designed and used for the processing of queries, the vocabulary can also be written to disk. For example the in-memory file structure can be modified so that terms and occurrences are stored as a pair in the file. Once all subindexes have been build, one large file exists where for one word multiple entries with occurrence lists exist. To build the final index, the occurrence lists have to be merged per word and then written to the index file in a term-sorted

order. Of course it is necessary to build a dedicated dictionary again to be able to provide fast access to the hitlists of a queried terms.

Since the used subindexes can be of arbitrary size, merge-based inversion can be processed with little memory usage by choosing only few documents for the subindexes. The smaller the subindexes get, the more expensive the merging will be. Merge-Based inversion can be parallelized when multiple subindexes are processed at the same time using different intermediary file areas or even single intermediary files that are then merged together when the final index is built. By the same way, merge-based inversion can be performed on multiple computers. The intermediate results would have then to be transferred to a single machine before the index is built.

2.4.4 Index Maintenance

While building an index offline can be done straightforwardly using the described algorithms, index maintenance is a problematical task. There are three main strategies to cope with index updates: in-place updating, index merging and re-building.

In-place Updates

Using in-place updates, words from newly indexed documents are simply added to the existing index. Files that store occurrences will fragment when new hits are inserted since new values have to be added to hitlists that are stored right inbetween other hitlists with no place to grow. Thus, the concerned hitlists have to be removed from the file, the new hits have to be added and the merged occurrence-list has to be written to free disk space. In many cases this will be exactly the end of the file where the changed list has to be appended. So a inverted file is likely to grow when new documents are added, not only in the necessary way but even more due to fragmentation. To solve this problem, the index file can be defragmented which causes additional administration costs and causes problems when query engines need to access the index files at that time.

To avoid problems with fragmented files, an embedded database like BerkeleyDB [Ber] or SQLite [SQL] can be used as proposed in [KNNS05]. This liberates the developer from implementing his own file storage solutions at the cost of a worse performance.

Merge-based Updates

The merging strategy schedules documents, that have to be added to an existing index are processed and a new subindex is computed in the memory. After that, a complete processing of the old existing index is necessary. The index data is read from hard disk sequentially, merged with the data from the newly indexed document in memory and then written to a new area on the hard disk. A great care has to be taken, that the unavailability of index parts is kept short enough to avoid interferences with the query engines.

Re-building

The third possibility is to completely rebuild the index. Assuming the availability of resources⁵, a new index is built and the old index is replaced by the new and complete index. This can cause unavailability for a certain time when central data like the vocabulary has to be loaded into the memory. Detailed information on index maintenance is given in [LZW04; LMZ05].

2.5 Parallel and Distributed Crawling and Indexing

Google uses MapReduce[DG04] for extensive distribution of crawling and indexing processes. Frank Lämmel has written a nifty essay[Läm06] about MapReduce in which he describes how an implementation could look in the strictly typed language Haskell. The basic MapReduce interface can be seen in listing 2.2

```
mapReduce ::
-> (k1 -> v1 -> [(k2, v2)]) -- ^ Map
-> (k2 -> [v2] -> Maybe v3)  -- ^ Reduce
-> [(k1, v1)]                -- ^ Input
-> Map k2 v3
```

Listing 2.2: Basic MapReduce interface

Despite the simplicity of the interface which only offers a MAP function to be applied to a list of (key, value) - input pairs and a REDUCE function that computes the final result from the data created by the MAP function, a lot of tasks can be solved using the MapReduce programming model. The developer only has to implement these two functions - which can be nicely short - while the MapReduce implementation

⁵Which means that the query engine can be run although the complete index is being rebuilt

takes care of available resources. This means that the job can run on only one or on thousands of CPUs without causing any constraints for the developer⁶. To deal with the storage of large data sets, MapReduce works with the so-called Google File System which allows the user to write large files into a storage that behaves like a single big hard disk even though it is spread over several different machines. The implementation takes care that tasks are assigned to worker machines that have the data sets which are to be processed stored mainly on their local hard drives to reduce network traffic and data accessing time.

The word count example explains the programming model in a graphic way:

Word count: To count the frequency of words in a set of documents, the k1s can be chosen to be document identifiers while the v1s contain the documents' contents like shown in table 2.5:

k1 (document-id)	v1 (content)
A	a rose is a rose is a rose
B	to be or not to be
C	what is in a name

Table 2.2: Input data for MapReduce job "word count"

The map phase would then extract the words from the input-v1s and create pairs of the kind (word, occurrence in document)⁷. The resulting pairs would look like shown in table 2.5

The pairs from the map phase are then sorted, which is done by the MapReduce implementation. After that the values are used keywise as input for the reduce phase with all values for identical keys concatenated to a list. The reduce function for the word count example would then have to sum up all v2s that exist for a pair which would lead to the result of the MapReduce job like shown in table 2.5.

The types of all keys and values are not specified and can be chosen arbitrarily, as long as the the types of the different phases and functions fit together. For example, the v1s can be altered to denote a file name and the map function could then load the contents for the job from the files⁹. In haskell it is also possible to use MapReduce with functions that have only one parameter by choosing the keys with the simple datatype ().

⁶Of course it affects the amount of input data, the developer can reasonably use.

⁷It would also be possible to compute (Word, 1)-pairs but that would make the intermediate data table too larger than necessary in this example

⁹Which would need a modified interface with monadic functions to allow IO operations.

k2 (Word)	v2 (frequency in document)	Document ⁸
a	3	A
rose	3	A
is	2	A
there	1	B
to	2	B
be	2	B
or	1	B
not	1	B
what	1	C
is	1	C
in	1	C
a	1	C
name	1	C

Table 2.3: Intermediary data for MapReduce job “word count”

Both, map and reduce phase can easily be parallelized. In the map phase, the list of input data can be split into a set of sublists which can be processed on multiple computers. The intermediary consisting of a list of (k2, v2)- pairs then has to be sorted by k2s implicitly by the MapReduce implementation. The resulting map - consisting of k2s and lists of v2s - could then again be computed on multiple computers.

Since the reduce phase can possibly fail, the result type of the reduce function is chosen to be a Maybe value, meaning that either a Just result is returned or, on failure, Nothing. To compute the final result, the results of the reduce phase would have to be collected by a single computer and merged together to the final result Map. If and how map and reduce phase are being parallelized is implementation-dependant. Lämmel also proposes the introduction of a third combiner-function which can help to compute intermediary results in the reduce phase which can offer better opportunities for work load balancing [Läm06].

Of course, real implementation have an extended interface adding configuration possibilities and possibly more functions to affect the MapReduce job like for example a hash function to influence which k2s are processed together on one single machine.

2.6 Case Folding, Stemming and Stop Words

Besides compression of index data, other techniques can be used to lower the memory usage of a search engine. The popular and efficient techniques case folding, stemming

k2 (word)	v3 (frequency in all document)
a	4
be	4
is	3
in	1
name	1
not	1
or	1
there	1
to	2
what	1

Table 2.4: Result for MapReduce job “word count”

and stop words are explained in this section. More details on these techniques can be found in [WMB99; Cha02].

Case folding

Depending on the position in a sentence, the first letter in a word can be a capital or not. If words are indexed “as-is”, they can possibly occur twice with different spelling in the index. This may cause increased memory consumption and can lead to incomplete search results if the search implementation does not consider this fact. To avoid all these problems, words can be case folded before they are inserted into the index. This means, that every upper-case letter is replaced by the corresponding lower-case one.

When case folding is used, the users of the search engine can not intentionally search for certain words that start with capitals. In some cases, this might be a desirable option like when a user wants to search for certain names, that are always written with upper-case letters. As a workaround, a postretrieval scan can be used where the original documents are scanned for upper-case word after retrieving the possible search results. Of course, this is only possible if the original documents are available while the search is performed and it leads to a much longer processing time of a search query. In the end it depends on the search engine purpose if it is reasonable to use case folding or not.

Stemming

Stemming means to strip one or more suffixes off a word and reduce it to root form. Words are indexed and later searched for without regarding tense and plurality of the concrete occurrence of the word. Since languages differ a lot in their grammar, it is necessary to implement different stemming algorithms for different languages. For english, the Porter-stemming-algorithm can be used, for other languages with more complicated grammars - german for example - the implementation of a stemming algorithm is a non-trivial task.

Stemming can cause a dramatic loss in the precision of search results and so it again depends on the purpose of the search engine if it makes sense to use stemming. A much easier and even less precise mechanism to lower the memory consumption is to truncate words at a certain position. Since usually the ends of words are truncated, this leads to a prefix search for the users.

Stop Words

Another possibility to shrink the index data is to define stop words, that are not inserted into the index. Usually these will be words, that occur very often in the indexed documents, because these words do not add much information to a document. For the english language “the” and “a” are very good candidates to be chosen as stop words. Another possibility can be to exclude all words that are shorter than a certain length.

The exclusion of stop words leads to problems when searches for phrases are performed. If for example “the” has been defined as a stop word, a phrase search for “the catcher in the rye” would not lead to any results. To solve this problem, either a postretrieval scan on the original documents that are candidates for the search results could be performed, which would again lead to much longer processing times. Another possibility would be to search with less accuracy and transform the query “the catcher in the rye” into something like “_ catcher _ _ rye” and search only for the really included words. But as the example shows, this can make the search fuzzy when the user maybe does not want to search fuzzy at all.

Another problem with stop words can be that some very common words - for example may, can and will - are homonyms for rarer nouns. Removing “can” from the index would prevent the search for “beer can” from working correctly and so the definition of stop words has to be done with care.

2.7 Duplicate Document Detection

Web pages are often available under several different URIs at the same time. For example, `http://www.google.com/index.html` and `http://www.google.com` lead to the same page because Google seems to have `index.html` defined as the standard file when a directory is being accessed. Although this is the standard configuration for many webservers, it is possible to have different configurations, for example when a web project is php-based. This leads to the conclusion that the evaluation of only URIs is not sufficient to identify duplicate documents.

Duplicate documents can be identified by the calculation of hash values as proposed in [GF04]. Even though it is in the nature of hashes that different input values can lead to the same hashes, using for example the domain of md5 with 32 digit hexadecimal numbers can be sufficient to avoid collisions in even midsize datasets.

3

Analysis

3.1 Requirements

3.1.1 Fast Search Operations

To be able to create fast search engines the decision was made to keep as much of the index data as possible in the main memory. Since it was predictable that with growing document sets more and more memory would have to be used, simple yet fast possibilities to move some of the index data into the secondary storage were to be tested. Users measure the quality of search results by the accuracy and speed of query processing and thus this had to be one of the primary design goals.

3.1.2 Prefix Search

Prefix searching adds the possibility to offer possible query completions to search engines. Combined with a find-as-you-type interface, prefix search can be an important tool for the user because it can help him to identify the documents that can satisfy his information faster and easier. Query completion hints can be generated based on different data sources: Google Suggest [Goo] uses already-processed queries to guess which completions could be interesting users. Buch.de [buc], a german online

bookstore, searches the product list and offers a list of 10 products, that match the user's query.

For prefix query processing, the index dictionary needs to be prefix-queryable in a time as short as possible. Although prefix searches could be performed on several datastructures, most of them would lead to long processing times for larger data sets. For example, binary search can look up values in $O(n)$ time, where n is the length of the input list. For large document sets and large resulting dictionaries, a faster data structure, that is as dictionary-length-independant as possible, are needed. Tree-based data structures qualify best to be used for dictionaries since they run in $O(n)$ time where n is the length of the query term. The main problem with these data structures is that they add overhead to the stored data and therefore less data can be stored in tree-based indexes than in list-based ones.

3.1.3 Scalability

The system has to be able to scale with the processed document sets. The construction time for indexes grows at least linear with the size of the underlying data. The processing of documents can only be slightly accelerated by algorithmic optimizations since bottlenecks exist when the data has to be retrieved either from local storages or - even worse - over slow network connections. The best way to speed up the index construction is to parallelize and/or distribute the tasks.

3.1.4 Configurability

Holumbus is designed to help developers build their own customized search applications. For the customization of search applications, the following configuration possibilities should be given:

Identification of Relevant and “Interesting” Document Parts

Document parts, that are of interest for a search application can be identified by the surrounding html tags and the corresponding attributes. For web pages that use HTML (as intended) to logically structure the information, this can be a relatively simple task. Nowadays most pages are administrated using web content management systems and therefore all pages are generated with the same template.

Pages, that use HTML wrongly for the manipulation of the pages' appereance, the definition of relevant document parts is an unlikely harder task. A common way to manipulate the layout in times when CSS was not yet established were HTML tables.

Even today many pages are built this way because it is considered to be easier than using CSS¹. In these layout tables, entities that logically belong to or depend on each other can be wrapped in table cells so that the original relation is not represented by the page structure. Restoring the underlying structure can optionally be done by manipulating the XmlTrees of the pages before the indexing is carried out.

A basic idea for the development of *Holumbus* is to make extensive use of the documents' structure, identify the interesting parts and to help improve the quality of search results and like proposed for example in [SHB06].

Flexible Identification of References to Other Documents

There are several possibilities in HTML how references to other documents can be expressed. The most common way to link to other pages are a-elements with href attributes. But also framesets and iframes can contain references to documents that are of interest for a search engine. Because there are some more possibilities in HTML and because in user-defined XML files arbitrary elements can be used to indicate references, the reference-extraction has to be easily configurable by the framework-user.

In- and Exclusion of Documents

In many cases not every page of a web site has to be added to a search engine's index. For example in a web shop the search can be designed to cover only the product pages and not pages that provide general information about the company. Whether a document has to be indexed or not can be decided either by looking at the documents' contents or - much more easily - by looking at the documents' URIs. To stay in the web shop example, the product pages could be found in a products-subdirectory while all general information is stored in a aboutus-subdirectory. Of course, the inclusion decision based on the documents' contents offers finer configuration options for the user, but since configuration can be also done on other levels - for example identification of relevant document parts - the URI-based solution should be sufficient for nearly all *Holumbus* use cases.

Definition and Exclusion of Stop Words

Stop words are commonly used by search engines to limit the index size and remove "uninteresting" words. Uninteresting in this case means, that very common words

¹Disclaimer: the author strongly disagrees

like “and” appear in too many documents and thus users would barely search for these words. For specialized search engines stop words can be very important. For example in a web shop the company’s name could be defined as stop word as it would probably appear in every document and thus add no valuable information for the search engine users.

Creation of a Preview Cache

Search engines usually provide previews of the original document which consist of a few sentences that help the user to decide whether a document is interesting for him or not. It should be possible to use this feature also in custom search engines built with the Holumbus framework. It is not reasonable to store the complete original documents because during the information extraction, it is possible that (even big) parts of the pages are not added to the index and therefore which “interesting” document parts are added to the cache should be configurable.

Flexible Design of Crawler/Indexer- Applications

Purpose and scope of a search engine heavily influence the index maintenance strategy to choose. This again determines in which way the crawler and indexer for the index creation have to be implemented. It is for example possible that a crawler is started periodically and only if new documents have been found, the corresponding indexer is run. Although crawling and indexing will be done sequentially in most use cases, it is also possible to use only one of the libraries. A crawler library for example could be used to implement a tool that generates sitemaps [sit] that can be used to improve the pages’ indexing in large-scale web search engines.

Definition of Document-specific Custom Information

Ranking of search results can be either done only based on the index contents or helped by precompiled information. Many web search engines adopted Google’s PageRank [PBMW98] to improve their ranking functions. The storage of such information or other information that can be used for the presentation of search results should be possible for framework users.

Evaluation of Different Index Types

While most search engines nowadays use some kind of an inverted index, it should be possible to implement and test other types like the Hybrid Index [BW06; BW07;

[BCSW07](#)] or even newer approaches in the future without having to rewrite the already-available modules. It should be tried to provide indexing mechanisms that can be easily used for other index types and at the same time leave enough possibilities to use the properties of the index to accelerate the index construction.

Usage and Evaluation of Haskell Packages

The *Haskell Xml Toolbox* [\[Sch\]](#) offers multiple nifty functions for XML processing with `haskell.HXT` was chosen for XML processing to add some advanced testcases for the library., The included support of XML pickler combinators (see [\[Ken04\]](#)) offers convenient possibilities for data persistence and debugging.

Janus [\[Uhl\]](#) is a web- and application server entirely written in Haskell. Since it is a young project, it was intended to also provide testcases and see how *Janus* would perform in real-life applications.

3.2 Basic functionality

3.2.1 Crawler State

To avoid the repeated processing of a single document, a crawler has to be aware which pages were already processed. Additionally a list of unprocessed documents is needed which is initialized with user-defined starting pages. In both cases sets are a reasonable choice to store the documents. After a document was processed, the internal state of the crawler has to be changed. The set of unprocessed documents is unioned with the set of documents that has been extracted as links and is not contained in the set of already-processed documents and the just-processed document is inserted into the set of processed documents.

In a distributed or parallel setup it is necessary to have a central crawler repository. MapReduce can be used to implement the crawler functionality since it can make use of multiple threads or computers and returns the results in a single process. To enable recursion, multiple MapReduce jobs have to be performed.

3.2.2 Caching Temporary Data

Building an index from scratch usually consists of at least two main tasks: crawling to find documents that are supposed to be included in the index and the index construction over the documents identified by the crawler. When data over which the index is to be built is not available on the machine(s) where the index is constructed

but has to be accessed over a possibly slow network connection, the documents accessed by the crawler can be saved as temporary data and so be made available for the indexer on local storages.

It is possible to compress the temporary data, but as this leads to longer processing times because the data has to be compressed during the crawl phase and decompressed during the indexing phase, compression should only be used for large datasets, where hard disk space may become a critical factor.

4

Implementation

4.1 Environment

The crawler and indexer libraries were performed under linux in a 2-core virtual machine with 2,5 GB memory on a 32-bit machine. To improve the testing, two other computers were used for index construction tasks so that up to three programs could be run at a time.

Haskell, a pure functional programming language, was used for the implementation. Besides the increased expressivity that is a common advantage of functional languages, Haskell has an advanced type system that forces developers to develop their programs with care. In many cases only little adjustments have to be made to a program once it has passed the *Haskell* typechecker. This helps to improve the productivity a lot since the usual write-compile-debug cycle is usually not necessary.

4.2 Framework

The crawler/indexer part and the search engine part of the *Holumbus* framework were developed at the same time, but it was predictable that the search part would be finished first. Thus the crawler and indexer libraries had to be used in beta-stages

to create test data for the testing of data structures and search algorithms. Figure 4.1 shows the overall structure of the *Holumbus* framework.

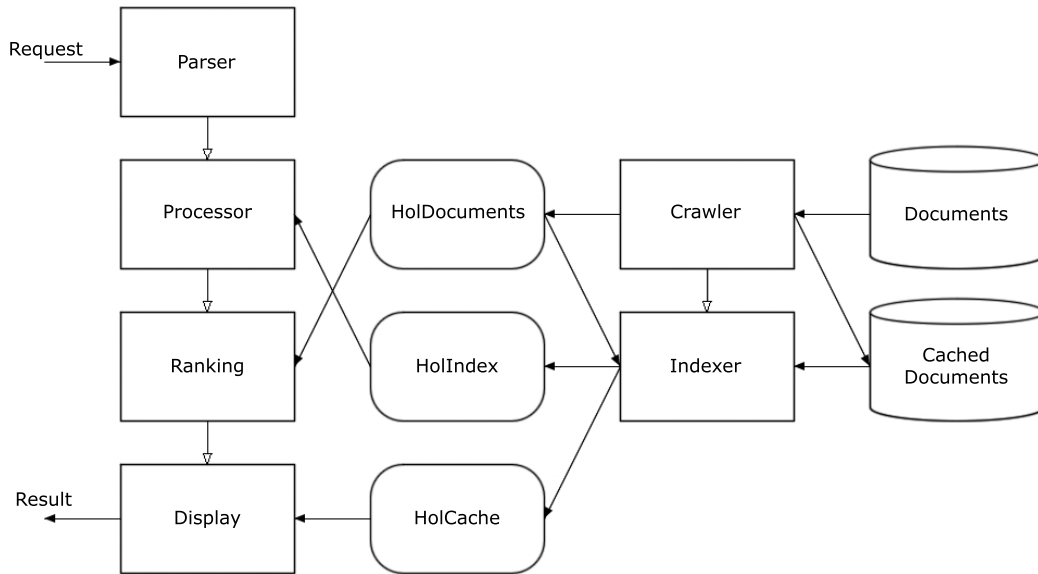


Figure 4.1: *Holumbus* framework architecture

The interfaces between query processing are defined in three Haskell typeclasses:

- **HoDocuments:** Stores the document set over which an index is built. The main purpose is that the URIs that point to the documents can be identified by key values of the type `Int`. Since documents usually contain more than one word and thus every document occurs in multiple occurrence lists, a lot of memory can be saved by just storing the `Int` key instead of the `URI String`.
- **HoIndex:** The index is the data structure that will be traversed when a search is performed. The class provides methods for the insertion of data, set-like functions and functions for query processing. To be able to test different index types, there are commonly defined functions for the creation of the index. Even though the creation process can be done in slightly shorter time if the inner structure

of an index is taken into account, it seemed reasonable to restrict the index creation to the common interfaces. So the crawler and indexer implementations can work with the common interface and when a new index implementation is available, a search application can be developed in short time because the existing library modules can be used.

- **HolCache:** A cache can optionally be used to store the plain text contents of the documents. This can be useful, when a preview of the documents is displayed on a search result page. Even though the plain text of a document could be retrieved from an index, every index implementation that is designed to allow fast query processing will not allow to be transformed back into the original documents within a time that would be acceptable for search engine users to wait for the results.

Since no distributed MapReduce implementation was available by the time the crawler and indexer libraries were developed, they were implemented to run on a single machine with regard to a later portation to run with such a distributed implementation. To be able to work with the MapReduce abstraction, a local library, that works with parallelization by spawning a configurable number of threads was implemented. The development of and experiences with this MapReduce library will be described in [4.5](#).

The index construction part basically consists of 3 modules: *Holumbus.Build.Crawl* provides functions to crawl file systems or web sites recursively. *Holumbus.Build.Index* offers functions for the construction of indexes over document sets contained in a `HolDocuments` type that has been generated by a crawler. Both modules can either be used in a combined indexer and crawler program or can be used to implement dedicated crawlers and dedicated indexers. The application structure to use depends on the specific setup for which a searchengine is built and on the strategy that is chosen to maintain the document table and index. The third module, *Holumbus.Build.Config*, contains the data structures that are used to store the configuration and program state during the document table & index construction. Additionally, small utility functions are provided that help the user with the configuration of crawlers and indexers.

In the beginning, the implementation of a dedicated library for the processing of xml configuration files was also considered. The definition of even a simple dtd to configure crawlers, that only included a set of pages to start with, a path in which to store temporary data and a path in which to store the results, showed that these configuration files would have to be much more verbose than corresponding configuration instructions in Haskell Syntax. It was obvious, that more complicated configuration

parts like arrows for XmlTree transformations would add much overhead and lead to a much more complicated syntax than Haskell's. So the idea of xml configuration was discarded and instead it was decided to show the configuration possibilities by providing simple examples that can be easily changed to implement simple crawler and indexer prototypes.

4.3 Crawler

4.3.1 Configuration and Internal State

The configuration and internal state of crawlers are saved in the CrawlerState type. This type is parameterized with a HoldDocuments type which determines the result type (12) of a crawler implementation.

```

data CrawlerState d a
  = CrawlerState
  { cs_toBeProcessed    :: Set URI                -- (1)
  , cs_wereProcessed    :: Set URI                -- (2)
  , cs_docHashes       :: Maybe (Map MD5Hash URI) -- (3)
  , cs_nextDocId       :: DocId                  -- (4)
  , cs_readAttributes  :: Attributes             -- (5)
  , cs_tempPath        :: Maybe String          -- (6)
  , cs_crawlerTimeOut  :: Int                    -- (7)
  , cs_fPreFilter      :: ArrowXml a' => a' XmlTree XmlTree -- (8)
  , cs_fGetReferences  :: ArrowXml a' => a' XmlTree [URI]  -- (9)
  , cs_fCrawlFilter    :: URI -> Bool            -- (10)
  , cs_fGetCustom      :: IOSArrow XmlTree (Maybe a)     -- (11)
  , cs_docs            :: HoldDocuments d a => d a         -- (12)
  }

```

Listing 4.1: Crawler configuration and state

The CrawlerState contains two Sets of documents (represented by URIs); one contains documents that still have to be processed (1) and one those already processed(2). Newly retrieved documents, that have passed the crawl filter (10) and thus are candidates to be processed, are first looked up in the wereProcessed-Set and if they are not included, they are added to the toBeProcessed-Set. Since usually at least a few documents are processed in a row before the new CrawlerState is calculated, the links extracted from these documents are added to a Set and the described lookup and insertion is carried out by calculating the difference of the Set of new links and

the Set of already processed documents and unioning this difference Set with the Set of unprocessed documents.

The `docHashes` (3) Map is used to avoid the duplicate insertion of identical documents that are available under different URIs into the result type. It is implemented as a `Maybe` type so duplicate document detection can be switched on by initializing the `CrawlerState` with `Just` an empty Map and switched off by setting the value to `Nothing`. The document hashes are calculated using `md5`, which provides 32-bit hashes and thus collisions are very unlikely to happen for document sets that can be processed with the *Holumbus* framework. To avoid the accidental exclusion of documents because of hash collisions, the implementation could be enhanced so that original documents are compared to verify the documents' identity. If two documents are detected that lead to the same hashes, one of the documents has to be removed. The implementation of the crawler library always keeps the document that is represented by the shorter URI because this is reasonable for most cases. For example, `www.haskell.org` would be selected instead of `www.haskell.org/index.html` and `haskell.org` would be selected instead of `www.haskell.org`. To allow more user control, the crawler configuration could be enhanced so that a function can be supplied that decides which URI to choose in case of identical document hashes.

In the beginning of the development, when the interfaces of the typeclasses were still subject to changes, the crawler library did not work with the interfaces of the `HoldDocuments` class but built the document table by constructing an `IntMap` and later converting it into a `HoldDocuments` type. Therefore the ids representing the documents had to be managed by the crawler. This was done using a list of unused ids (which was initialized with the infinite list `[1..]`). Later this was changed and only the next free id was kept in the state and thus no ids were recycled when documents were removed from the documents table. When the crawler library was changed to work only on the `HoldDocuments` interface, the `nextDocId` (4) field of the `CrawlerState` was not removed because it can be used to give the user feedback about the currently processed document when tracing is switched on.

The `readAttributes` (5) are passed to *HXT's* `readDocument` function and can be initialized with standard values that work with many tested pages offered by the *Config*-module. The `tempPath` (6) can be used to switch on document caching by the crawler so that the indexer can work with local copies of the files instead of having to access the remote sites again. `Just`-values enable caching while `Nothing` switches it off. The `tempPath` is also used to write dumps of the `CrawlerState` to the hard disk. This enables resuming a crawler application that has been interrupted. When no `tempPath` is configured, the path for the additionally written files is defaulted to

the `/tmp` directory. If temporary files have been written, the user has to take care that the indexer works with the temporary files (supposing that crawler and indexer run sequentially). This can be easily done by mapping an auxiliary function over the documents that replaces remote URIs by the local copies. Since the index works with document identifiers, the index created can be used together with the documents table that contains the remote URIs so that search results will point to the original documents instead of the local copies (which would lead to problems, since many local paths should not be accessible for webservers).

The `crawlerTimeout` can be set (0 means no timeout) to interrupt the processing of pages that takes a long time. As experienced during the implementation, certain pages seem to cause trouble for *HXT*. These are often pages that contain long lists or huge tables and are not interesting for search engine users anyways.

Sometimes it is necessary to manipulate the `XmlTree` of a document before the links are extracted. For example, certain parts of a page could be removed to prevent the crawler from finding links in these parts. The `XmlTree` manipulation can be configured with the `preFilter` (8). The `preFilter`-arrow can be set to `this`, which leaves the `XmlTree` unchanged.

To make the extraction of references configurable, the `getReferences`-arrow (9) was added to the `CrawlerState`. The arrow can be generated with an utility function from the `Holumbus.Build.Config` module, a standard function that extracts anchors and pages from frameset and iframe-definitions, is also available. The utility function is based on XPath expressions because this eases the configuration for unexperienced users. To provide more flexibility for advanced users and enabler faster arrow implementations, it was decided to use an arrow instead of XPath expressions for the configuration inside the `CrawlerState`.

The last and one of the most important configuration options is the `getCustom`-arrow (11). This adds a flexible possibility to extract this custom information from xml documents. This is useful since the `HolDocuments` type can store arbitrary data for each document which can be used for ranking or preview purposes. Unexperienced users can set this arrow to something like `constA Nothing :: Maybe` which will skip the extraction of custom information.

4.3.2 Algorithm

From the beginning, the crawler library was implemented to work with parallelization or distribution. Of course this added some constraints to the implementation because the single crawler processes have to work independently but not anarchically. It has

to be avoided that pages are processed multiple times by different processes and the results have to be collected and merged together on a single machine. The MapReduce abstraction can be used to easily implement parallel or distributed crawlers.

The list of unprocessed documents can be passed to the *MapReduce* implementation and the *CrawlerState* can be computed as result of the job. This has to be repeated until no new references are found by one job and the *HoldDocuments* have completely been processed.

4.4 Indexer

4.4.1 Configuration

Indexer applications have to be configured using the *IndexerConfig* type. Besides the definition how information has to be extracted from the XML pages, it also contains some configuration options that are also or only needed for crawlers. The options were implemented in this way because unexperienced users can only configure the *IndexerConfig*-type and then create a corresponding crawler configuration with an utility function from the *Config*-module which sets the remaining options to commonly useful default values.

```

data IndexerConfig
  = IndexerConfig
  { ic_startPages      :: [URI]           -- (1)
  , ic_tempPath       :: Maybe String   -- (2)
  , ic_indexPath      :: String         -- (3)
  , ic_contextConfigs :: [ContextConfig] -- (4)
  , ic_fCrawlFilter   :: URI -> Bool    -- (5)
  , ic_readAttributes :: Attributes      -- (6)
  , ic_indexerTimeout :: Int           -- (7)
  }

```

Listing 4.2: Indexer configuration and state

The *startPages* (1) consist of a list of URIs with which a crawler is supposed to start and are not needed by the indexer module. Other options that are only useful for the crawler initialization are the *readAttributes* (6) that are passed to *HXT*'s *readDocument* function and the *crawlFilter* (5) that decides whether a document has to be processed. The *tempPath* (2) field is used to store temporary data and is mainly passed to the corresponding crawler where it is used as described before.

The `indexPath` (3) defines, where the resulting files (document table and index) have to be stored. Since the data structures have to be written to the hard disk by the framework user, this can be easily overwritten if needed but it is helpful to configure different `indexPaths` if different `IndexerConfigs` are tested inside a single program. A configuration switch can then easily be performed without having to adjust the paths and filenames under which the files have to be saved.

To interrupt the processing of documents that are processed very slowly, a timeout can be set. As for the crawler timeout, setting the value to 0 means no timeout. For testing purposes this can be set to a relatively short time and when the configuration has been tested and produces an index as intended, the period can be expanded or the timeout can be removed.

The `contextConfigs` (4) is - as the name says - a list of configurations for the different contexts of the index. A config represents information that is found in specific parts of the documents. For example, the headline-tags in HTML (`h1 . . . h6`) could be added to a context "headlines". The name (*a*) is used as identifier for the context and is needed for query processing. Different weights can be assigned to contexts for ranking purposes or the query syntax can allow the search engine user to only search in specified contexts.

```

data ContextConfig
  = ContextConfig
    { cc_name           :: String                -- (a)
    , cc_preFilter     :: ArrowXml a => a XmlTree XmlTree -- (b)
    , cc_fExtract      :: ArrowXml a => a XmlTree XmlTree -- (c)
    , cc_fTokenize     :: String -> [String]         -- (d)
    , cc_flsStopWord   :: String -> Bool             -- (e)
    , cc_addToCache    :: Bool                     -- (f)
    }

```

Listing 4.3: Context configuration

When the structure of a document is not properly represented in the documents's markup or when `XmlTree` transformations have to be made so that certain information can be correctly identified, a `preFilter` (*b*) can be defined. The easiest filter again would be the `this-arrow`, that leaves the `XmlTree` unchanged.

To extract the interesting information from a `XmlTree`, again an arrow can be used (*c*). The easiest way to define an extraction arrow would be to use HXT's `getXPathTrees`-function. For example, if the contents of a `div`-element that has a `class`-attribute with the value `interesting` the following function could be used:

`getXPathTrees ‘‘//div[@class=’interesting’]’’`. Care has to be taken, that nodes, that contain text nodes are selected instead of the text nodes itself. Using the XPath expression `‘‘//div[@class=’interesting’]//text()’’` would extract all text inside the div-element, but if the text was further formatted, for example if it contained links, the text would be extracted not in the right order. This would lead to failing phrase searches and flawed document previews.

The `tokenize`-function (*d*) defines how an extracted String has to be split up into single words. In many cases this can be done by considering whitespaces to be word delimitiers, but in certain cases (for example *Haskell* function signatures), word seperators have to be defined different or extracted text has to be inserted completely as one term. The `tokenize` function can be used to implement more features than obvious at first sight. By stripping certain characters from the beginning or ending of words, the vocabulary can be heavily influenced and simplified. Words that are followed by a punctuation mark would be indexed with this special character if not stripped by the tokenizer. So a single term could occur in mutiple forms in the index, which would lead to more memory consumption and irritating query completion hints.

The `tokenize`-function is also the point where stemming and case folding can be configured. Since implementing stemmers is not a trivial task for many languages, the simplified truncation could be used. This would be very easy to implement in Haskell by using the `take`-function. Since *Holumbus* is prefix-search-capable, truncation could be interesting to test and use with large data sets. In the same way, case folding could be implemented by extending the `tokenize`-function with a function that converts uppercase to lowercase characters.

Stop words can be defined by implementing the `isStopWord` function (*e*). `const False` can be used to disable stop word exclusion. Because the stop words are implemented using a function instead of a list of words, it is very easy to exclude words that are too short and are thus considered to be uninformative. Of course this can be combined with a list of stop words to refine the indexer configuration.

The `addToCache`-flag (*f*) defines if the contents of a certain index should be added to the cache. This is useful when not all contexts are used for the preview of document. If a cache is created the concrete `HolCache` type can be defined with the parameters of the `buildIndex`-function, that is imported by the `indexer`-modules.

It is also possible to merge different page areas into one context by simply using the same name for different `ContextConfigs`. This can help to avoid `XmlTree`-transformations before the interesting information is extracted and is probably more difficult to configure and test.

4.4.2 Index Construction

To understand the difficulties of the index construction, a review of the index implementation is helpful. Since *Holumbus* is designed for prefix searching and trie-data structures qualify best for the implementation of prefix searching since the lookup of values is mainly depending on the length of the query term, this section will deal with trie-based indexes. The first implementation of the `HolIndex` interface was a memory-based inverted index in *Holumbus.Index.Inverted.Memory*. Since all developments thereafter, that were designed to reduce the memory usage, are also based on the general structure of this implementation and only store the hitlists in hard disk files, this first implementation is used as a role model. The definition of the index was simplified to ease the understanding. For example, index compression is not taken into account.

```

import qualified Data.Map           as M
import qualified Holumbus.Data.StrMap as SM
import qualified Data.IntMap        as IM
import qualified Data.IntSet        as IS

type Inverted    = M.Map Context Part
type Part        = SM.StrMap Occurrences
type Occurrences = IM.IntMap IS.IntSet
type Context     = String

```

Listing 4.4: *Memory-based inverted index*

The simplest way to insert new words into the index would be to use a function like `insertPosition`, as shown in listing 4.5. The problem with this kind of insertion is, that for every new word, a lot of datastructures have to be traversed. First, the outermost `Map`, that maps the contexts to dictionary tries of the type `StrMap` has to be searched for the context in which the new word occurs. This runs in $O(n_1)$ time where n_1 is the number of contexts. When the context is already included in the index, which it will be in most cases as the number of contexts is likely to be very little, the corresponding `StrMap` has to be searched for the word. This adds $O(n_2)$ to the execution time where n_2 is the length of the word. Again it is important whether the word is already included in the `StrMap`. Because most words occur in multiple documents, the word will already be included in the majority of cases. So the insertion algorithm often has to go down another level. The `IntMap` is searched for the document id which adds another $O(n_3)$ to the execution time. Since many words occur multiple times in a

document, in many cases also the lowest level has to be processed and $O(n_4)$ is added to the execution time where n_4 is the number of positions in the `InteSet`.

```
insertPosition :: Context -> Word -> DocId -> Position
              -> Inverted -> Inverted
insertPosition c w d p i =
  M.unionWith (SM.unionWith $ IM.unionWith IS.union)
    i
    (M.singleton c $
     SM.singleton w $
     IM.singleton d $
     IS.singleton p )
```

Listing 4.5: *Inserting words into an inverted index*

It is obvious that this way of index construction is not usable since it leads to exponential execution times. The first naive indexer implementation worked in the described way but soon, as larger document sets were processed, it became clear that the index construction had to be done more efficiently. A first optimization is to avoid the necessity to go down all index levels when new hits are inserted into the index. When a document is processed, the `IntSet` of positions in which a word occurs in the document can already be computed. The processing time can be reduced even more if parallelization is used because the process that merges all precomputed values now can work with partial results. To insert the precomputed position-sets into an index, the `insertOccurrences`-function from listing 4.6 can be used.

```
insertOccurrences :: Context -> Word -> Occurrences
                 -> Inverted -> Inverted
insertOccurrences c w o i =
  M.unionWith (SM.unionWith $ IM.unionWith IS.union)
    i
    (M.singleton c $
     SM.singleton w o)
```

Listing 4.6: *Inserting occurrences into an inverted index*

But there is more to optimize to speed up index construction. When the lower levels of the index are precomputed, the merging of the higher levels can be done in a relatively short time. The fastest way would be to first compute the set of positions of words in documents and create the resulting hitlists of the type `IntMap IntSet`. After that the hitlists of identical words in the same contexts in all documents

should be merged, which would be done by a call like `SM.unionWith $ IM.unionWith IS.union`. This would be a conventional inverted index without contextual information like it is used by many search engines. The last level of merging to construct a *Holumbus* index then would be to merge the single contexts to build the outermost Map from contexts to dictionaries.

The *MapReduce* abstraction can help to get very close to this described fast way of *Holumbus* index construction. This is no surprise since *MapReduce* was mainly introduced to improve index construction, but even the additional context level can be processed using the abstraction. In [BP98], the index construction as performed by the google prototype is described. The first step - disregarding crawling etc. - is to compute a forward index that contains the per-document hitlists for every word. This is similar to the described per-document `IntMap IntSet`- calculation in *Holumbus*. The second step is then to sort the hitlists per word and compute. Because the result of *Google's* index construction was an inverted file and no contextual information was needed, the index construction was then finished. The in-place sorting performed by the *Google* indexer is similar to the merging of word-wise hitlists per context as described for *Holumbus*.

To illustrate how *MapReduce* helps with efficient index construction, the basic interface is relisted:

```
mapReduce ::
-> (k1 -> v1 -> [(k2, v2)]) -- ^ Map
-> (k2 -> [v2] -> Maybe v3) -- ^ Reduce
-> [(k1, v1)] -- ^ Input
-> Map k2 v3
```

Listing 4.7: Basic *MapReduce* interface

When no contextual information is taken into account, the keys and values could be used as listed in table 4.4.2.

k1	DocId	Integer value that represents the document
v1	URI	The documents' URI as a String value
k2	Word	A unique word (term), String value
v2	Occurrences	Word-per-document hitlists

Table 4.1: *MapReduce* types for index construction

The index construction would then be performed as follows: During the map-phase, the hitlists for every word in every document would be computed. The *MapReduce* implementation would then sort these partial hitlists by words. In the reduce phase, the hitlists for every word would be merged and the final result would be put together by the *MapReduce* implementation. Listing 4.8 shows two pseudo-code-functions in Haskell notation that could be used for the index construction.

```

-- map-function
computeOccurrences :: DocId -> FilePath -> Occurrences
computeOccurrences d f =
  mergePositions $ zip [1..] (words . readFile f)
  where
    mergePositions :: [(Word, Position)] -> Occurrences
    mergePositions = ... -- folds the list to per-word IntSets, adds
                        -- the document id as mid-level IntMap key
    readFile :: FilePath -> String
    readFile = ... -- reads file contents into markup-free String

-- reduce-function
mergeHitlists :: Word -> [Occurrences] -> Maybe Occurrences
mergeHitlists _ os = Just $ foldl (unionWith IS.union) IM.empty os

```

Listing 4.8: Basic MapReduce interface

To extend the index construction so that the context is taken into account is then an easy task: The *k*2s have to be chosen to be (Context, Word)-pairs. The result of the MapReduce job would therefore be a map from these pairs to the hitlists and this intermediary Map then has to be transformed into the Inverted type. The MapReducible-typeclass that will be introduced in 4.5 helps to simplify the index construction and avoid this final transformation.

Document Processing

Of course, the information extraction is much more complex than showed in listing 4.8. To illustrate how the indexer works with the supplied configuration, the basic algorithm for index extraction is shown in table 4.4.2. It is important to process the single steps in the right order. For example if stopwords were removed before the words are numbered, it would be impossible to search for phrases that include a stop word. If the query parser is also aware of stopwords, it can construct a query that

takes missing words in between other words into account. The algorithm is described for a single document as parallelization or distribution are managed by *MapReduce*.

- (A) Read the source document with HXT's `readDocument` function
 - (B) For all configured contexts
 - (1) Apply the `preFilter` to (optionally) manipulate the `XmlTree`
 - (2) Extract information that has to be inserted into the context
 - (3) Concat results, if multiple nodes contained interesting information
 - (4) Tokenize the `String` into single words
 - (5) Write document cache if configured
 - (6) Number the words in the context
 - (7) Filter stopwords to exclude them from the index
 - (8) Compute sets of in-document positions per word
-

Table 4.2: Basic *Holumbus* index construction algorithm

4.5 Parallelization and Distribution with MapReduce

4.5.1 Memory-based Parallel Implementation

Even though no *MapReduce* implementation was available by the time, the *Holumbus* development started, it seemed reasonable to prepare at least the crawling and indexer modules of the framework for the use with such an implementation in the future. To be able to work with the *MapReduce* abstraction it was necessary to implement a basic *MapReduce* library with multiple threads on a single machine to simulate the later-planned distribution with a *MapReduce* framework.

In the already-shown *MapReduce* definition, the result of a *MapReduce* job is always a `Map`. Even though an inverted index is always some kind of `Map`, this did not provide enough flexibility for different implementations. For example, when the memory-based inverted `HoIIndex` is altered to have no contextual information, the highest-level datatype in the chosen implementation would be a `StrMap` and thus not be computable with the *MapReduce* function.

To add more flexibility, the type class `MapReducible` was introduced which defines two basic methods that are needed to construct data types with *MapReduce*. The first function is needed to insert data into the result type, which is done in the reduce phase and the second is a function that merges two result types as it is necessary when intermediate results are computed by different processes and have to be put

together to form the overall result. Instances of the `MapReducible` class depend on both types that are used in the reduce phase and the result type and so an instance declaration could look like `instance MapReducible InvertedIndex String DocId` for an index, that the documents in which a word occurs for every word.

To avoid name collisions with other modules, the functions were named `reduceMR` and `mergeMR`. Because IO operations might be necessary during the construction phase or when data is merged, both methods were defined as monadic IO operations.

```

module Holumbus.Control.MapReduce.MapReducible where

class (Ord k2) => MapReducible mr k2 v2 | mr -> k2, mr -> v2 where
  reduceMR :: mr -> k2 -> [v2] -> IO (Maybe (mr))
  mergeMR  :: mr -> mr -> IO mr

mapReduce :: (Ord k2 , MapReducible mr k2 v2, Show k2) =>
  Int                -- ^ No. of Threads
-> mr                -- ^ initial value for the result
-> (k1 -> v1 -> IO [(k2, v2)]) -- ^ Map function
-> [(k1, v1)]        -- ^ input data
-> IO (mr)

```

Listing 4.9: Basic *MapReduce* interface

The basic interface of *MapReduce* was extended as shown in listing 4.9. It is possible, to define the maximum number of concurrent threads. The second parameter is only used to determine the type of the concrete `MapReducible` that is being built¹. The map function has also been extended to be a monadic IO function since network and hard disk access is needed in many applications. Building indexes - the reason why the interface was set up - would be more or less impossible without IO operations.

MapReduce runs in four phases:

- computing intermediary data (map phase)
- grouping intermediary data
- computing partial results (reduce phase)
- merging the partial results to become the final result

The map phase was implemented different than in the original sense. In a distributed setup, the input data list is partitioned and the single partitions are handed

¹the first parameter of the `reduceMR` method serves the same purpose

to the worker machines that compute the intermediary data and notify the master computer when they are done. This could be changed because the communication barriers over slow network connections was no bottleneck in the single-machine-setup.

More problems were caused by very large documents. The processing time with *HXT* and multiple contexts can sometimes grow to several minutes. If many of these slowly-processed documents would be in the same partition - what is likely to happen because big files are likely to be located in the same directories - and the partitions would be handed to threads, this could cause that one thread has to continue computing for minutes or hours even when all other threads are already finished. To get a better load of potentially hard-to-process input data, the input data list is handed one element by one to the worker threads. If one thread has reported his results, it dies and a new thread is started by the main thread. Of course this adds additional costs but tests with different document sets showed that this overhead is worth the advantages gained by the better work load. Another possibility would have been to not let the workers die but signal to the main/dispatcher thread that they are finished. The dispatcher could then have sent new data to be processed to the idle threads. This might have led to slight improvements but since the parallel library was an interim solution, no further changes were made to the library.

The grouping phase runs in a single thread and inserts all intermediary data into a `Map k2 [v2]` which is then passed to the reduce phase. In the beginning, the reduce phase was not parallelized. Performance measurements showed, that the reduce phase would take more and more time of the total processing time the bigger the input data sets are. So it was changed to also run with multiple threads.

In contrast to the map phase, the input data of the reduce phase is partitioned and then passed to the working threads. This decision was made because the processing times for the $(k2, [v])$ pairs are more likely to be alike than the document processing times. Of course, when the `[v2]` lists differ in length, there will be different processing times but because every thread has to process approximately the same number of pairs, it is probable that threads will be done at approximately the same time. Of course it would be possible to measure the lengths of the `[v2]` lists and partition the data more sophisticated but again this was discarded because the parallel library was only intended to be an interim solution.

To build the final result, the intermediate results of the single threads are collected by the main thread and merged together.

4.5.2 Parallel Implementation with Hard-disk-stored Intermediary Data

The *MapReduce* implementation with multiple threads worked fine for crawlers with a limited set of input documents and indexers that were not to index too many word occurrences. If too much input data was passed to the *MapReduce* function, this resulted in massive memory consumption until finally the processes were killed by the operating system.

So there was a certain limit to the size of processable datasets. The first solution, to process subresults over parts of the input data list did not work efficiently because the merging of inhomogenous is expensive as described in the indexer section. Several tests to overcome this problems have been done, unfortunately, to say it in advance, no satisfying implementation could be found. The tests were stopped with regard to the development of the Holumbus MapReduce framework that was already developed in parallel and will have to solve these problems even for single-machine computations.

The first try was to have a single thread that collects all data from the threads processing the map phase and stores the $(k2, v2)$ -pairs in a SQLite database was introduced. The problem was that this lead to enormous increasing processing times. Simple performance tests led to the conclusion that these were mainly caused by the new bottleneck, the result-collecting thread.

So this approach was refined. For every worker thread, an extra database was created and every worker wrote his results into his own database. Before the reduce phase, it was now necessary to sort the databases. This was very expensive as it turned out, even when the sorting was optimized with multiple threads. In performance tests, the sorting of the databases accounted for at least $3/4$ of the whole processing time. This indicated that the SQLite database was simply not fast enough for this task. The same conclusion could be drawn from the long conversion times from a memory-based into an database-driven one.

The next try was to store one file per $k2$ on the hard disk and append new values to these files whenever they were computed from the worker threads. Again the IO operations accounted for most of the processing time so that this way had to be discarded, too. The best way for the storage of the intermediate data will probably be to use one big file per worker process, which will be computers or multiple applications on one computer in a distributed setup. This solution will be discussed more in [7.2](#)

4.6 Monadic Indexes

As the crawler and indexer library were stepwise improved and the possible scope of processable documents was enlarged, it became obvious that some parts of the index had to be moved to the hard disk to reduce the memory consumption. The average growth of an inverted index and its dictionary can be seen in figure 4.2. The growth of the number of unique words becomes less and less the more documents are added to an index, because the vocabulary of natural languages is limited. At some point there are no more words that can be indexed. Of course this is highly theoretical because names, words that are composed of other words and misspelled words can always appear as new unique words and it is unlikely that even the biggest web search engines have a dictionary that includes “every” word of a natural language.

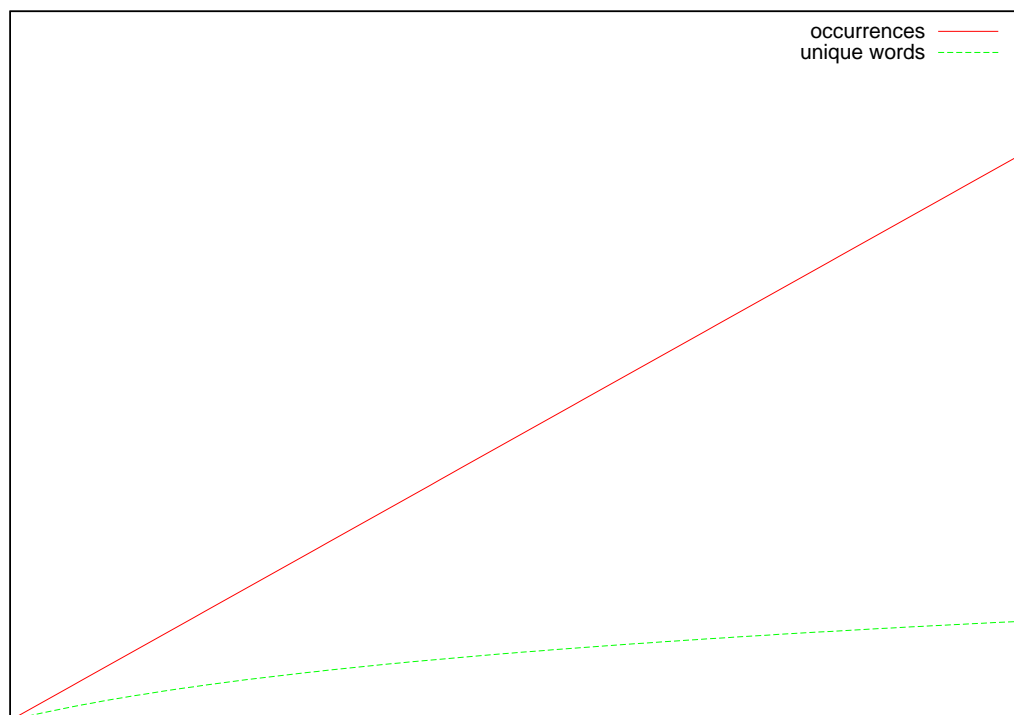


Figure 4.2: *Holumbus.Index.Inverted.Memory*

The first thing to do when an index has to be moved to the hard disk partially is to remove the hitlists from the main memory, store them on the hard disk and read them during query processing when they are needed. As the dictionary is likely to grow much less than the data stored in the hitlists, the effect of a hard disk based dictionary can be considered less than hard-disk-stored occurrence lists. Since *Holumbus* uses

trie-based data structures that consume more memory than other searchable data structure to improve the retrieval efficiency, the dictionary also consumes a significant amount of memory. To be able to move occurrence lists and optionally the dictionary or at least parts of the dictionary onto the hard disk, the `HolIndex` interface had to be extended to work with monadic functions. The introduced typeclass `HolIndexM` can be seen in listing 4.10.

```

class (Monad m, MapReducible i (Context, Word) Occurrences )
  => HolIndexM m i where

  sizeWordsM      :: i -> m Int
  contextsM       :: i -> m [Context]

  allWordsM       :: i -> Context -> m RawResult
  prefixCaseM    :: i -> Context -> String -> m RawResult
  prefixNoCaseM  :: i -> Context -> String -> m RawResult
  lookupCaseM    :: i -> Context -> String -> m RawResult
  lookupNoCaseM  :: i -> Context -> String -> m RawResult

  insertOccurrencesM :: Context -> Word -> Occurrences -> i -> m i
  deleteOccurrencesM :: Context -> Word -> Occurrences -> i -> m i

  insertPositionM  :: Context -> Word -> DocId -> Position -> i -> m i
  deletePositionM  :: Context -> Word -> DocId -> Position -> i -> m i

  mergeIndexesM    :: i -> i -> m i

  updateDocIdsM :: (Context -> Word -> DocId -> DocId) -> i -> m i

  toListM      :: i -> m [(Context, Word, Occurrences)]
  fromListM    :: i -> [(Context, Word, Occurrences)] -> m i

```

Listing 4.10: *Monadic index interface*

Three different approaches to store the occurrence lists on hard disk have been examined so far. All of them have specific advantages but also disadvantages that will be discussed in the next subsections. Emphasis is placed on the SQLite-based approach since it seems to offer the best balance between benefits and handicaps.

While index construction could easily be done using the `HolIndexM`-Interface, the query processor was not yet completely able to deal with the new interface. So the `HolIndex`-interface was also implemented by using `unsafePerformIO`.

4.6.1 `Holubus.Index.Inverted.OneFile`

The first try to store occurrences on the hard disk used a single file. Higher levels of the memory-based index were left unchanged, but instead of the Occurrences-lists, pointers into the file were stored. The main problem with this solution was that inserting new occurrences into the index could not be done easily. All occurrence lists were stored sequentially in the file and the new hits could not be inserted at the end of an existing list. So the list would have had to be removed, merged with the new hitlist and stored at the end of the file. This is problematic because the occurrences-file would fragment over time and either occupy much more space than needed or be subject to defragmentation.

So this approach could only be used for newly generated indexes or be constructed from an existing index. But this would mean that enough memory to keep the index in it was available and the hard disk storage was not necessary any more. Implementing sophisticated disk-storage solutions was neither a primary goal of [Hüb08] nor of this work and so the implementation was discontinued at a certain point. The development of for example a b-tree based storage solution could be an interesting and important task for the future of the Holubus framework.

4.6.2 `Holubus.Index.Inverted.Persistent`

The second approach tried to shift the fragmentation problem to the operating system. Instead of in-file pointers, the Dictionary stored Integer values from which filenames could be computed. The occurrences of one word in one index were stored in a single file. To insert new occurrences for an already existing word, this file only had to be read, the occurrences had to be merged with the new occurrences and the combined hits had to be rewritten to the file.

The main problems of this solution were IO exceptions due to lazy IO operations and that hundreds of thousands of files in a single directory are not very elegant and cause problems to the file system, when for example the contents of a directory have to be listed. To deal with the lazy IO, a strict version of `decodeFile` from `Data.Binary` was used. But this caused a dependency conflict because the *Janus* webserver needed an earlier version of the `ByteString` library which carried a bug that prevented the strict `decodeFile` function from working correctly. The third main problem was that the construction of the index, even when it was These two disadvantages led to discontinuation of the development.

4.6.3 *Holumbus.Index.Inverted.Database*

In order to reduce the number of files, a database-driven solution was tried. Again the Dictionary trie was to be kept in the memory and this time it stored Integer values that represent database keys that can be used to retrieve the occurrences from the db. The goal was to create an index type that would be capable of online updates. Once again the problem was, that the index construction was very complex. The construction by mapping a function that stores the occurrences in the db over a memory-based index would take hours even for indexes over less than 5000 words.

4.6.4 Synopsis

The index that stores the occurrences in one single file was implemented first to reduce the memory usage of search engine applications. Because of the problems with index maintenance, the development was stopped and other index types were tested. Both shifted the problem with fragmentation to either the filesystem or a SQLite database and both suffered from very long processing times. The disk access was not fast enough to be used for larger data sets but as the amount of information was the reason to implement disk-based indexes, both index types had to be discarded.

As the index construction algorithm had been improved in the meantime and occurrences for one word only had to be written once per index construction, the index from *Holumbus.Index.Inverted.OneFile* was improved so it can be used for index construction. The best way to construct this semi-persistent index is still to build a memory-based index and then convert it into the Persistent type. In this case, the occurrences are written in order and accesses to occurrence list of similar words can be performed faster.

To be able to process even more data in the future, the Persistent index has to be further refined. The first step would be to add a function that defragments an occurrence file in place or converts the old fragmented file to a new, defragmented one. This should also be practicable for datasets within the scope of the framework. The performance and memory usage reduction are examined in chapter 5.

5

Evaluation

In this section, the existing index implementations are examined concerning their memory usage. The evaluation was based on a set of approximately 80.000¹ German wikipedia pages. The document set contains articles about words starting with “a”, but since the explanation of the words contributes most to the index, this should have no significant impact on the index statistics.

To be able to monitor the memory usage for growing indexes, several smaller indexes were built, each containing 500 documents. The documents were sorted alphabetically before being partitioned for the creation of the partial indexes. Naturally the constructed indexes vary in size but the sizes of the first indexes do not differ abnormally from the sizes of the last indexes. To get the test data, the indexes were merged one-by-one and every intermediate index was saved to disk.

The indexes contain three contexts:

- `title` includes all words from the `title`-tag
- `categories` includes the wikipedia categories in which the explained word is listed
- `content` includes the description of the current word, but not menus and other information like updates.

¹Disclaimer: of which only about 22,500 documents were used

5.1 Memory Usage

The memory usage was examined for the two index types that have performed best and qualify for usage in search engine applications: The memory-based Inverted index from *Holumbus.Index.Inverted.Memory* and the Persistent index from *Holumbus.Index.Inverted.OneFile* that stores the occurrence lists in a single binary file.

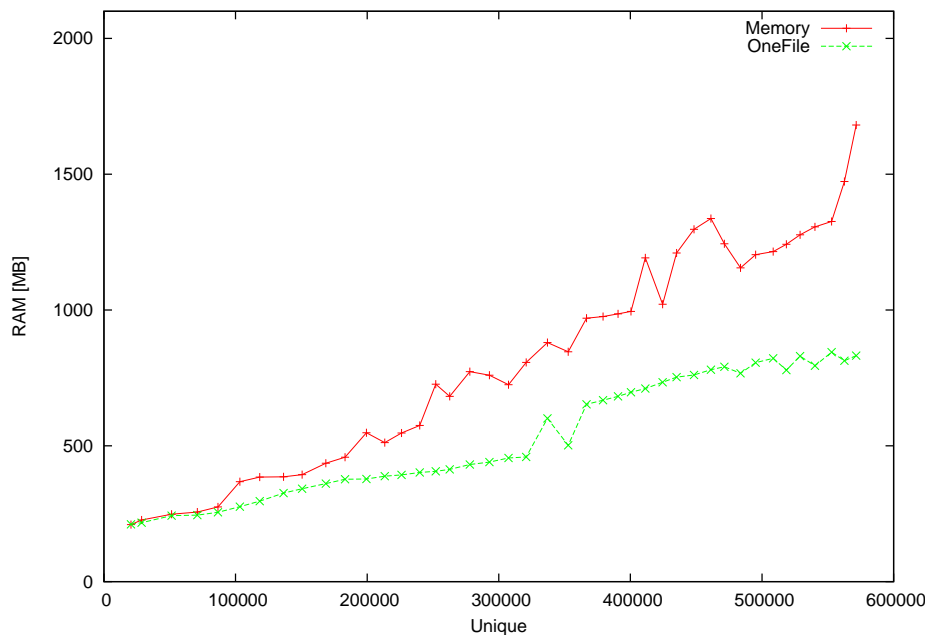


Figure 5.1: Memory usage of inverted indexes with different implementations II

The memory usage is shown in relation to the number of unique words contained in the index in figure 5.1. When the usage is compared to the number of documents in the index, the figure looks very similar. One remarkable fact about the graph is that at some points adding new documents and new unique words to the index leads to a size reduction. This is mainly due to the *StrMap* implementation that is used for both indexes.

The Persistent index's memory need grows significantly slower than the memory-based Inverted index's and the gap between both graphs becomes larger the more documents / unique words are added to the indexes. The strong growth at the end of the graph should be considered incidental. Because of the limited indexer scope due to the memory-based MapReduce implementation, it was not yet possible to compare larger indexes. This has to be caught up once the *Holumbus MapReduce* framework is ready to compute larger indexes.

5.2 Query Processing

The quality of a search engine is mainly determined by the accuracy of the search results, but the processing time is also a very important factor. Queries have to be performed in a time as short as possible, even more when a find-as-you-type interface is used. To determine whether the Persistent index from *Holumbus.Index.Inverted.OneFile* could be used in a productive environment, it was compared to the memory-based inverted index from *Holumbus.Index.Inverted.Memory*.

The tests used queries which all contained only one or two lowercase characters in arbitrary combinations. Queries of this kind are hard to process for the persistent index, because many occurrences lists have to be read from disk to compute the result. The results of the test are shown in table 5.2.

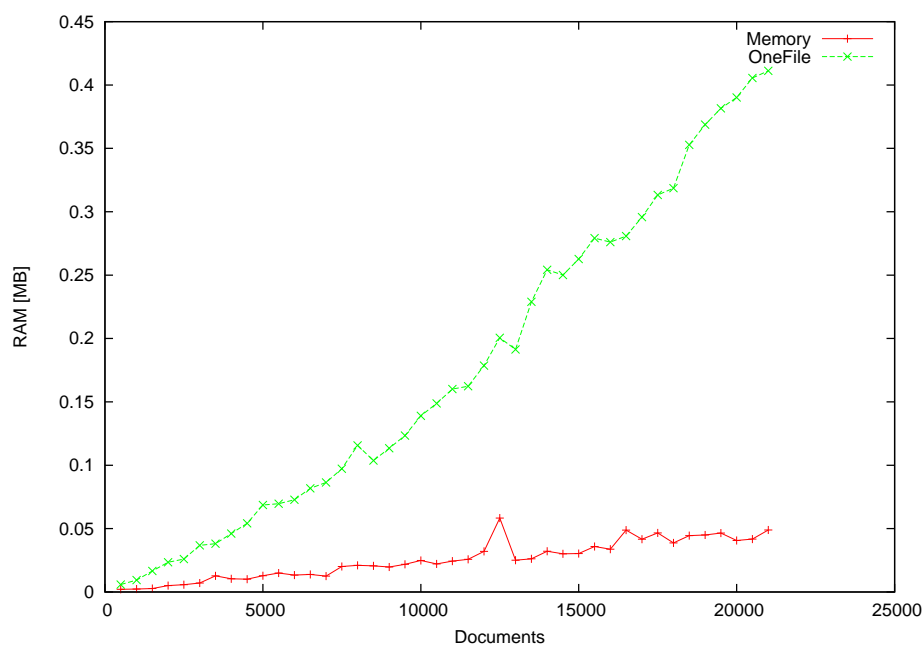


Figure 5.2: Index construction time with different thread count in seconds

As expectable the query processing takes more time for the IO-using index. The results listed in table 5.1 show that even for the biggest tested index with 571535 unique words, even the Persistent was able to answer queries within 0.4 seconds averagely. When more complex queries are performed or even bigger datasets are being processed, optimization to keep the query processing time within acceptable ranges is needed. For example, short query terms that consist of only one letter could not be processed to avoid extremely expensive retrieval tasks.

	Memory-based	Persistent
total processing time [seconds]	30	290
seconds per query	0.0429	0.4127
queries per second	23.33	2.42

Table 5.1: Query processing benchmarks (702 queries, 571535 unique words)

5.3 Index Construction

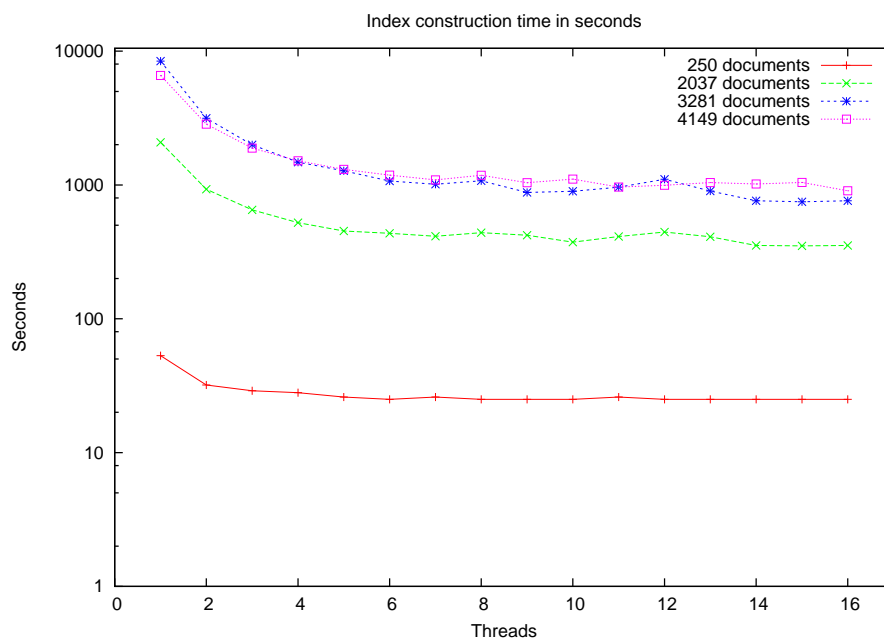


Figure 5.3: Index construction time with different thread count in seconds

After the development of the libraries, the time that was needed for the construction of an index was also evaluated. The most interesting question was if and to which degree the parallelization could help to speed up the index construction processes. 4 different-sized document sets were used to evaluate the construction time.

Originally it was planned to also compare the memory-based MapReduce module to ones that used persistent intermediary data but as described before, the performance of all alternatives that were tried could in no way compete with the memory-based solution. The index construction was tested with 1 to 16 threads, the results are shown in table 5.3 and in figure 5.3, which uses a logarithmic scale.

The result of the evaluation is that the parallelization works fine and helps to

significantly reduce the index construction time. The performance was especially boosted when the MapReduce-module was changed to work with a parallel reduce phase.

Documents	1	2	3	4	5	6	7	8
250	53	32	29	28	26	25	26	25
2037	2081	929	651	523	453	435	414	440
3281	6576	2846	1885	1515	1307	1183	1092	1182
4149	8431	3148	1996	1479	1276	1070	1012	1074

Documents	9	10	11	12	13	14	15	16
250	25	25	26	25	25	25	25	25
2037	421	374	412	445	410	353	351	353
3281	1040	1105	964	995	1042	1018	1045	904
4149	880	897	963	1104	900	761	749	762

Table 5.2: Time (in seconds) to construct an index over precomputed *HolDocuments* on a local hard disk with different thread counts

5.4 Janus Webserver

Janus [Uhl] is used as webserver for the *Hayoo* search engine. The logs show that on the days of the announcements of new *Hayoo* over 1,250 unique visitors used the search engine and generated more than 15,000 single requests per requests.

There are no major problems with the performance of *Janus*. After some days of running, *Hayoo* tends to become slightly slower which is probably due to the built-in memory limit on the web server *Hayoo* is being run on. To overcome this shortcoming, it is planned to replace the memory-based *Inverted* index with the *Persistent* index which will lower the memory usage significantly and therefore hopefully lead to shorter processing times, even when *Hayoo* is running for weeks and months without restart.

6

Example Applications

6.1 Hayoo! Haskell API Search

Hayoo! was the most important example application that was created during the development of the Holubmus search engine framework. It helped to clarify the needs for the configuration of crawlers and indexers and shows the enormous flexibility that is offered by *Holumbus*.

Haskell API documentation can be generated using *Haddock* [Mar], which is a virtual standard that is used for example on *Hackage* [Hac], the most common place for *Haskell* library downloads. *Hoogle* [Mit], a search engine explicitly specialised for *Haskell* API search, served as a role model for *Hayoo!*. In contrast to *Hoogle*, which - at least partially - works with files that are explicitly generated for the *Hoogle* index by *Haddock*, *Hayoo!* completely works on the generated HTML files.

Haddock-generated pages list all datatypes and functions defined in a module on one HTML page. *Hayoo!* was designed as a function-based search since any element in a *Haddock*-generated page can be directly accessed by in-page-anchors. The basic idea was to split up the original HTML documents into separate virtual documents. The URIs of those documents can exactly point to the datatype or function by adding the in-page-anchors at the end of the URIs.

The HTML generated by *Haddock* is difficult to handle for structure-aware indexers. There is little usage of CSS for the page formatting and a page consists of a set of nested tables. The tables are mainly used for formatting and they do not represent the logical structure of the page necessarily. For example the declaration of a function can be wrapped in a table row and the corresponding function explanation can be included in a table row on the same hierarchy level. Besides the need to split the documents into virtual documents to create a function- and datatype-based search engine, comprehensive transformations have to be applied to the original documents to be able to index them properly.

The first transformation that is applied is that the table rows describing type classes is transformed. The methods of every class are wrapped in a nested table and thus would not be recognized by the indexer. The table is lifted up one hierarchy level and so the methods' HTML looks identical to the HTML of normal functions.

There is an option for *Haddock* that allows the generation of links to syntax-highlighted *Haskell* source files. When this option is turned on, the table cell that contains the function declaration is filled with another table that separates the declaration from the source link. To help the indexer, this inner table is removed and declaration and source-link are put in only one table cell.

After that, the documentation of datas is removed and the definitions of datas, types and newtypes are transformed. In the current version, these elements are only indexed with their names and any description that is contained is discarded. This solution was chosen because - especially for data declarations - it is not clear which information should be put into the index. For a data struct, the documentation of the fields could be more interesting than a short sentence that describes the struct as a hole. The inclusion of data, type and newtype-documentation should be included later to further improve Hayoo.

The next transformation step deals with function declarations where the signatures are split into multiple rows. When the *Haskell* source code contains signatures where the parameter types are listed in single rows and commented with *Haddock*-comments of the type `-- ^`, the *Haddock* HTML page looks similar and lists the parameters in a nested table where one cell in a row contains the parameter type and another cell contains the description of the parameter. Because this parameter-wise description is not included in the index, the nested table is once again transformed into a single table cell on the same hierarchy level as other functions.

The last and most difficult step prepares the *Haddock* pages for easy access by the indexer and transforms the HTML into a new page that represents the logical structure of the document. Every function is wrapped in a `tr`-element with an `id`-attribute that

contains the function name. It includes table cells, one of which contains the function declaration and the other one contains the corresponding documentation. The pages are then split into smaller documents where each document contains exactly one function. During this step, each table row is wrapped in a table-element and another row containing cells with module and package name is added.

After all transformations have been performed, the split documents have been saved to disk and a new `Documents`-instance has been processed, the standard `buildIndex`-function is used to build the index over all files.

To avoid the multiple inclusion of a package with different versions in the index, the split `Documents` are processed with a simple filter. The package names are extracted from the URIs for the *Hackage* documents and set by hand for all packages that are included additionally¹.

To enable efficient information retrieval, the *Hayoo* index contains several contexts, which are listed in table 6.1²

Name	Description
name	Full function names, e.g. <code>getTime</code>
partial	Split function names, e.g. <code>get</code> and <code>Time</code>
module	Full module names, e.g. <code>Data.Map</code>
hierarchy	Split module names, e.g. <code>Data</code> and <code>Map</code>
description	All words from function descriptions
signature	Function signature with unnecessary whitespaces removed, e.g. <code>Int->IO Bool</code>
normalized	Generalized function signatures, e.g. <code>a->b</code>
package	The name of the package without version, e.g. <code>bytestring</code>
category	Category in which the package is listed on <i>hackage</i> , e.g. <code>Database</code>

Table 6.1: *FH Wedel contexts*

Prefix search helps to even search for *Haskell* function signatures because these are contained in the signature context. The search for e.g. `Int` will return all functions that take an `Int` value as first parameter. Unnecessary whitespace is stripped by the query parser so that a search for `Int -> Int` and `Int->Int` do return the same results. The partial context enables some kind of specialized infix search because function names with camelcase notation are split up into single words and so the search for `time` would return results like `setTime` or `getTime`.

¹Currently this is only the `gtk2hs[gtk]` package

²The table from [Hüb08] was taken and adjusted to the current *Hayoo* version

For the future, it is planned to improve the quality of Hayoo search results by computing a specialized PageRank[PBMW98], that will help to measure the importance of packages and help to order the results in a way so that often-used packages will be listed first.

6.2 FH Wedel

As a second example application, a full text search for *FH Wedel - university of applied sciences* was developed. The search has a scope of over 10.000 documents and thus could be used to test the *Holumbus* behaviour with slightly larger data sets.

The *FH Wedel* has a common layout for the pages that are published on the web site, where the content can be easily identified by a `div`-element with a certain `id`-attribute. So the contents of the pages using the layout can be indexed without information that is only layout-related, like for example menus, headers and footers. Nevertheless there are some pages that do not use this layout because it is either not reasonable³ or the pages have not yet been converted to the new layout. The main context, *content* does not include the contents of these pages which will mean that they can not be retrieved. To deal with this shortcoming, an additional *raw* context was introduced that contains the raw text of all pages. This context is ranked with very low importance and therefore the results from this context will only be displayed on the first page(s), when the search in the other contexts did not return (enough) results. An overview over all contexts defined is given in table 6.2.

Name	Description
title	All words from the <code>title</code> -tag
partial	All meta keywords and all words from meta description
content	The page content without menus etc.
raw	Raw contents of the page without unnecessary information removed

Table 6.2: *Fh Wedel contexts*

Since the processing of non-xml filetypes has not been implemented yet, *Holumbus* can not be used as a search engine for the *FH Wedel*. There is much important information given in pdf-documents and this information would not be available any more over the search interface. The currently-used search engine supports pdf files and therefore *Holumbus'* ability to also do so is a critical factor before it would be possible to use it as *FH Wedel* site search engine.

³For example for pages that are used for presentations

In contrast to the *Hayoo* interface, that displays possible word completions in a tag cloud, the *FH Wedel* search interface will use a normal text field. To provide a find-as-you-type interface, a dropdown box⁴ will be opened when characters are entered into the interface providing possible completions that will return search result that seem to be interesting for the user. This kind of interface allows the use of the search box on every page, which is a common feature on many webpages and will thus be also usable for other *Holumbus* applications.

6.3 CGI-based Search Engine

The *Janus* webserver [Uhl] has been built using *hs-plugins* [Steb] to load user modules at runtime. *Hs-plugins* is vulnerable to changes in the version of the used bytestring-module and compiler and hands this vulnerability down to *Janus*. To become more independent of this software and to be able to provide an example application that can be easily changed by an user to build searchengine prototypes, the **cgicomplete** example was introduced. It consists of 3 modules:

- server application, that loads document table and index data into the main memory and listens to queries on a configurable port.
- cgi-client, that can be run in a webserver's cgi-bin directory and processes the users' queries inserted in a web interface
- simple crawler/indexer application, which can be easily changed to implement custom search engines.

Additionally, some template files are provided which can be adjusted (css-file or template xml-file) or replaced (logo-image) to implement a custom web interface. The **cgicomplete** example only requires the *Holumbus* framework to be installed and then can be run on any webserver that is cgi-capable.

Currently the *cgicomplete*-example can be mainly used for testing indexes. It will be expanded to offer more complex possibilities like type-as-you-find and query completion hints in the future.

6.4 Sitemap Tool

Using only the crawler library, it is simple to implement a tool for the construction of sitemaps that can be used to help search engines to index the pages of a web site [sit].

⁴Like Google Suggest[Goo]

The sitemap example was implemented to build such a sitemap over the haskellwiki at <http://www.haskell.org>.

It uses the crawler module to recursively discover all pages that have to be added to the sitemap and creates a document table (`Documents`). The data is then written to a file using a `XmlPickler`. The `XmlPickler` that is already implemented in `Holumbus.Index.Documents` can not be used because it builds the xml file with a different structure and the resulting file would have to be manipulated to form a proper sitemap. Instead, a new pickler was implemented for the `Documents` type and this is explicitly used with the `xpickleDocument`-function instead of the `Holumbus`-integrated `writeToXmlFile`-Function.

It takes little imagination to expand this little example to the implementation of more sophisticated content syndication / mashup tools. The crawler and optionally the indexer module provide good possibilities to extract information from webpages. `HXT`'s pickler functions can help to convert the resulting data into XML/HTML. Because of the nature of picklers it is also simple to reload the constructed files again and use it in other haskell applications or - when a mashup tool is run periodically as initializing data.

7

Conclusion

7.1 Synopsis

The crawler and indexer modules that were developed in this work provide a good base for the implementation of crawler and indexer applications. The main development goal - to be able to implement highly configurable crawlers and indexers - was reached. The most important use case is the *Hayoo!* search engine[HS], that extracts data from HTML pages that are not indexer-friendly and converts it into an index structure that allows sophisticated queries and efficient information retrieval.

The most important shortcoming of the *Holumbus* framework so far is that the size of processable document sets is limited. This is mainly due to the concentration on MapReduce-based index construction, which helps to parallelize and distribute the computation but also leads to restrictions when all intermediate results are kept in the main memory. During this work several tests to store the intermediate data on the hard disk have been carried out, but a final solution was not yet implemented. Nevertheless, important implications for the implementation of a distributed MapReduce framework could be drawn from these tests and will be summarized in section 7.2

By using the index from *Holumbus.Index.Inverted.OneFile*, the memory usage of on-line indexes can be significantly reduced, so that this is no longer the main limiting factor. Once the size limits caused by the parallel memory-based MapReduce module

are removed by using the distributed MapReduce framework ¹, it will be interesting to measure the memory usage of indexes over larger document sets using the Persistent index.

Much of the configurability of the crawler and indexer has been gained by using functions in the configuration data types. This is equivalent to the strategy pattern [GHJV95] in object-oriented languages but adds no implementation costs when a pure functional language like *Haskell* is used. The configuration with functions offers experienced users the possibility to optimize their functions which can help to improve the performance significantly when the *HXT* is used, because there are often different ways how information can be extracted.

The usage of *Haskell* helped to implement the *Holumbus* framework in a shorter time and with less lines of code than a comparable work that would be developed using java. The strictly typed nature of *Haskell* improved the collaboration because when changes were made to certain modules those changes were exactly indicated by compiler errors and adjustments to depending modules could be made quickly. Because the crawler and indexer modules are mainly based on XML processing using *HXT*, it was necessary to deal with the arrow notation right from the beginning. This made it a little more difficult to gain experience with *Haskell*.

The benefits of a strictly typed language became obvious whenever the typed environment had to be left, for example when binary data was written to a disk and then re-read again or when `XmlTrees` had to be manipulated in order to enable an indexer to extract the data. This was mainly done during the development of the *Hayoo* indexer, where the source pages had to be passed through a number of filters to get the data base for the indexer. The main problem was, that any conversion from one `XmlTree` to another will pass the *Haskell* type checker without problems. But if one filter is erroneous, it may damage the `XmlTree` in a way that the later-applied filters will not work any more and that the information can not be extracted. For example, if all nodes are removed from an `XmlTree`, this can still be passed into another filter that tries to manipulate the tree, but the type checker can not notice this. Most of the debugging that was necessary during the development was due to the impossibility to benefit from *Haskell's* strict type system.

¹Which will be set up so it is also usable on only one computer

7.2 Implications for distributed MapReduce

7.2.1 Intermediary Data

To be able to build indexes in a parallel and distributed manner, the crawler- and indexer-libraries have been developed using the *MapReduce* abstraction. The first naive implementation stored all data - input, intermediary results and output - in the main memory. This led to performance problems because for large input data - many documents or big documents - the memory would run over and force the kernel to stop the Crawler- or indexer processes. To be able to process the data, the input data was processed partitionwise. For example, the crawler recomputed the next state before all documents were processed to reduce the intermediary data and the indexer module was designed to be able to build subindexes for large datasets. The problem with this solution was, that on the one hand it contradicted the idea of *MapReduce* where tasks are assigned to worker clients who then work independently and on the other hand the built sub indexes had to be built subsets of the whole document set. The merging of all sub-indexes to one final big index was unaffordable expensive because the indexes were not disjoint in matters of words but disjoint in matters of documents and so on a lower level of an inverted data structure.

To deal with the big memory needs, another parallel *MapReduce* abstraction was implemented that stores the intermediary data in an embedded sqlite-database file on the hard disk. This led to an enormous drop in the memory usage during the map phase. Only in the reduce phase, when the final result was built, the memory usage grew continuously for memory-based data structures like *Holumbus.Index.Inverted.Memory*. The tests that were carried out with the SQLite-based MapReduce module and also with a module that used one file per k2 did not lead to satisfying results.

The probably best solution will be to open binary files and append the results from the map-phase to these files. After all data has been processed in the map-phase, it will be necessary to presort the list of keys and values for the reduce-phase per worker and then do a cluster-wide sorting. Since the *bytestring*-package offers append-functions, this should be implementable with affordable costs. As the test with different index types with occurrence lists stored on a hard disk showed, big binary files that can be accessed with file pointers perform much better than SQLite or many small files.

7.2.2 Shared Dictionary

Another problem that has to be solved is the memory-efficient storage of the k2-keys. In the original *Google*-model, the k2-keys in a index-building MapReduce Job were words for which the occurrences had to be stored as v2-values. For *Holumbus* this has been extended to a pair (Context, Word). Since words usually occur in different documents, there can be many intermediary items with identical k2-keys. To be space-efficient, they should be replaced by an Integer and stored in an extra data structure like a Map. The parallel implementation uses this way to reduce the amount of data that has to be written to disk. But it takes advantage of the fact, that it runs on a single machine and so the k2-keys do not have to be shared across the network. The Map from keys to int-values is managed by the main thread, which collects all results of the worker threads and is so able to have a globally valid Dictionary. In a distributed implementation, the dictionary has to be managed in a different way. *Google's* approach [BP98] was to start with a large default dictionary with about 14 millions entries and give copies of these entries to the worker clients. All new words, that could not be found in this dictionary were then processed by a single indexer at the end of the MapReduce Job so that this client was able to insert the missing words into the dictionary and assign Integer values representing the original k2-keys. For the *Holumbus* framework, which is designed to implement search engines based on significantly less documents, this might not be an appropriate way. The underlying vocabulary will be unknown before the index construction in many cases² and the implementation will probably not run on hundreds or thousands of clients within the next years. A possibility to solve the problem might be the introduction of a dictionary server that manages the global dictionary and sends changes to the clients. The clients could query the server with a list of unknown keys and the result would be a Map from those Words³ to Int values. Of course this would lead to much network traffic in the beginning of a *MapReduce* job and thus has to be tested for practicability. To reduce the network traffic, the *MapReduce* jobs could be initialized with default dictionaries of very common words in the Contexts where it is considered helpful.

7.2.3 Distributed Reduce-phase

In [Läm06], Lämmel suggests the introduction of a combiner function that can be applied to the intermediary data on the worker computers to compute second-stage intermediary results. Those newly generated values would then have to be merged

²Unless a re-indexing is performed

³and other words that were discovered by other workers first and are unknown to the client

on a single computer to compute the job result. When a data-structure like the `MapReducible`-class presented in 4.5 is used, the explicit definition of a reduce function should not be necessary. Workers could compute intermediary results and the master could collect these and build the final result. Probably more important is the sorting of the input data between the map- and the reduce-phase. As already stated, the merging of indexes that are build on different document subsets is extremely expensive and since the final result would be computed by the hierarchical merging of subindexes, the last merging step would use an unaffordable amount of time. The merging can be performed better, when the lower levels - in terms of an inverted index the occurrences - are computed first and then the merging is done only for the higher levels - which would be the dictionary for the index. To achieve this kind of computations, the intermediary data after the map-phase has to be sorted by k2-key so that - again taking the index as an examples - all occurrence-lists for one word are processed on a single worker. The merging of the indexes would then be performed only on the dictionary since the complete occurrence lists for every word would already be computed and thus the index merging should be performed in significantly less time than with the naive approach. A hash-function for the partitioning of data would offer more flexibility for the *MapReduce*-user, but is probably not necessary in the first place.

7.3 Future Work

7.3.1 Current Projects

There are two ongoing projects by the time this work is finished. Stefan Schmidt is developing a *MapReduce* framework in Haskell[Sch08] which will be designed to support any MapReduce computations. Because of the experiences made during the crawler and indexer-implementation, some optimizations towards search engine applications will be made. Ludger Steens is working on an efficient medium-scale *PageRank* calculation library which will be using the common *Holumbus* datatypes and will be integratable into *Holumbus*-driven search engine applications.

7.3.2 Support Different Filetypes

Currently *Holumbus* is set up to work with XML files. This is a reasonable choice since one of the key design goals was to be able to build context-aware search engines. Nevertheless, to be able to use *Holumbus* for several applications, it will be necessary

to support other filtypes. For example, pdf files can be processed using the command-line tool `pdftotext` and pure text files can be indexed with their full text content. The lack of structural information has to be tackled in the configuration of concrete search engine implementations. For example the contents of pdf files could be added to a special “pdf”-context which is given little priority for the result ranking. So, pdf files are not likely to be found when common words are searched, but when they contain words or word groups, that are not available in other contexts, they would be presented as search results.

7.3.3 Evaluate More Index Types

The implementation of the first memory-based inverted index was nice to define the basic operation for the Holumbus data types. To be able to build indexes over larger datasets, IO was made available in the new monadic interface `HolumbusIndexM` and a first implementation set up. More efficient data backends should be implemented and evaluated and it could be tried to find efficient implementations to store not only the hitlists on hard disk but also the vocabulary or at least parts of it.

In [BW06; BW07; BCSW07] a hybrid index structure is introduced where the index data is not completely inverted. The vocabulary points to data blocks in which sequential searching has to be done. Although the hybrid index structure was evaluated with only medium success in [Sch07], it might be worth considering the implementation of such an index as part of Holumbus or at least try to benefit from the basic ideas to find efficient ways to store occurrences and parts on disk and so reduce the memory usage.

7.3.4 Index Maintenance

In the example application built so far, index updates are always performed by building a new complete index and then replacing the old one. For smaller indexes like the ones that were developed this is a passable opportunity. The *Hayoo!* index for example can be loaded within seconds on the *Hayoo!* virtual machine and so index updates lead to very short downtimes. For larger index and different applications, other possibilities for index updates should be tested. For examples web shops depend on on-the-fly-availability of new products in their search interface.

7.3.5 Stemming

Stemming can be an important tool to lower the memory needs of an index. There is now a new Stemmer library[[stea](#)] available on *Hackage*, that provides a haskell binding to the *Snowball* stemming library[[sno](#)]. *Snowball* offers stemmers for various languages and it will be interesting to examine how stemming can enrich the functionality of the Holumbus framework.

7.3.6 Sophisticated Index Merging

The main limiting factor for building larger indexes was the memory usage of the naive *MapReduce* implementation. Splitting the set of documents over which an index has to be built into subsets and then building subindexes can help to limit the memory wastage. With the currently available naive merging algorithms it is virtually impossible to merge the subindexes to generate a final index over all documents. This could be overcome by the implementation of sophisticated merging algorithms that take advantage of the underlying data structure and optionally merge different parts of the indexes in different threads to accelerate the merging.

Bibliography

- BCSW07** BAST, Holger ; CHITEA, Alexandru ; SUCHANEK, Fabian ; WEBER, Ingmar: ESTER: Efficient Search in Text, Entities, and Relations. In: CLARKE, Charlie (Hrsg.) ; FUHR, Norbert (Hrsg.) ; KANDO, Noriko (Hrsg.): *30th International Conference on Research and Development in Information Retrieval (SIGIR'07)*. Amsterdam, Netherlands : ACM, 2007, pages 671–678
- Ber** BerkeleyDB. <http://sleepycat.com>
- BP98** BRIN, Sergey ; PAGE, Lawrence: The Anatomy of a Large-Scale Hypertextual Web Search Engine. In: *Computer Networks* 30 (1998), Nr. 1–7, pages 107–117
- buc** Buch.de web shop with find-as-you-type interface. <http://www.buch.de>
- BW06** BAST, Holger ; WEBER, Ingmar: Type less, find more: fast autocompletion search with a succinct index. In: *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. New York : ACM Press, 2006, pages 364–371
- BW07** BAST, Holger ; WEBER, Ingmar: The CompleteSearch Engine: Interactive, Efficient, and Towards IR&DB Integration. In: *CIDR '07: Proceedings of the 3rd biennial conference on Innovative Data Systems Research*, www.cidrdb.org, 2007, pages 88–95
- BYRN99** BAEZA-YATES, R. ; RIBEIRO-NETO, B.: *Modern Information Retrieval*. Addison-Wesley, 1999
- Cha02** CHAKRABARTI, S.: *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann, 2002
- DG04** DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *OSDI '04: Proceedings of the 6th symposium on Operating System Design and Implementation*. Berkeley : USENIX Association, 2004, 137–150
- Gan08** GANTZ, John F.: *The Diverse and Exploding Digital Universe*. Version: March 2008. <http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf>, last checked: 12.03.2008. – IDC white paper

- GF04** GROSSMAN, David A. ; FRIEDER, Ophir: *Information Retrieval – Algorithms and Heuristics*. 2nd edition. Dordrecht : Springer, 2004
- GHJV95** GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns*. Boston : Addison-Wesley, 1995
- Goo** *Google Suggest*. <http://labs.google.com/suggest>, last checked: 01.03.2008
- gtk** *Gtk2Hs - A GUI Library for Haskell based on Gtk*.
<http://www.haskell.org/gtk2hs/>, last checked: 15.07.2008
- Hac** *Hackage*. <http://hackage.haskell.org/>, last checked: 01.08.2008
- HS** HÜBEL, Timo B. ; SCHLATT, Sebastian M.: *Hayoo! Haskell API Search*.
<http://www.holumbus.fh-wedel.de/hayoo>, last checked: 01.08.2008
- Hüb08** HÜBEL, Timo: *The Holumbus Framework: Creating fast, flexible and highly customizable search engines with Haskell*. 2008. – Master’s Thesis
- Ken04** KENNEDY, Andrew J.: Functional Pearls: Pickler Combinators. In: *Journal of Functional Programming* 6 (2004), Nr. 14, pages 727–739
- KNNS05** KAAE, Rasmus ; NGUYEN, Thanh-Duy ; NØRGAARD, Dennis ; SCHMIDT, Albrecht: Kalchas: a dynamic XML search engine. In: *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–140–6, pages 541–548
- Läm06** LÄMMEL, Ralf: *Google’s MapReduce Programming Model – Revisited*. 2006. – Accepted for publication in the *Science of Computer Programming Journal*
- LMZ05** LESTER, Nicholas ; MOFFAT, Alistair ; ZOBEL, Justin: Fast on-line index construction by geometric partitioning. In: *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–140–6, pages 776–783
- LZW04** LESTER, Nicholas ; ZOBEL, Justin ; WILLIAMS, Hugh E.: In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In: *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2004, pages 15–23
- Mar** MARLOW, Simon: *Haddock*. <http://www.haskell.org/haddock/>, last checked: 15.07.2008
- Mit** MITCHELL, Neil: *Hoogle*. <http://haskell.org/hoogle>, last checked: 15.07.2008

- PBMW98** PAGE, Lawrence ; BRIN, Sergey ; MOTWANI, Rajeev ; WINOGRAD, Terry: The PageRank Citation Ranking: Bringing Order to the Web. / Stanford Digital Library Technologies Project. Version: 1998.
<http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1999-66&format=pdf&compression=\&name=1999-66.pdf>. 1998.
– Forschungsbericht
- Sch** SCHMIDT, Uwe: *Haskell XML Toolbox*.
<http://www.fh-wedel.de/~si/HXmlToolbox>, last checked: 04.03.2008
- Sch07** SCHMIDT, Stefan: *Typhoon 2 - Volltextsuche*, FH Wedel, Diplomarbeit, 2007
- Sch08** SCHMIDT, Stefan: *The Holumbus Framework: Distributed computing with MapReduce in Haskell*. 2008. – Master’s Thesis
- SHB06** SAUVAGNAT, Karen ; HLAOUA, Lobna ; BOUGHANEM, Mohand: XML retrieval: what about using contextual relevance? In: *SAC ’06: Proceedings of the 2006 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2006. – ISBN 1-59593-108-2, pages 1114–1115
- sit** *sitemaps.org*. <http://sitemaps.org>, last checked: 21.07.2008
- sno** *Snowball - Stemming library*. <http://snowball.tartarus.org/>, last checked: 01.08.2008
- SQL** *SQLite*. <http://sqlite.org>
- stea** *Stemmer - Haskell binding to the Snowball stemming library*. <http://hackage.haskell.org/cgi-bin/hackage-scripts/package/stemmer>, last checked: 01.08.2008
- Steb** STEWART, Don: *hs-plugins - Dynamically Loaded Haskell Modules*.
<http://www.cse.unsw.edu.au/~dons/hs-plugins/index.html>, last checked: 25.06.2008
- Uhl** UHLIG, Christian: *Janus Application Server*.
<http://darcs.fh-wedel.de/janus>, last checked: 24.07.2008
- WMB99** WITTEN, Ian H. ; MOFFAT, Alistair ; BELL, Timothy C.: *Managing Gigabytes – Compressing and Indexing Documents and Images*. 2nd edition. San Francisco : Morgan Kaufmann, 1999
- yah** *Yahoo! Search builder*. <http://builder.search.yahoo.com>, last checked: 02.08.2008
- ZM06** ZOBEL, Justin ; MOFFAT, Alistair: Inverted files for text search engines. In: *ACM Comput. Surv.* 38 (2006), Nr. 2.
<http://dx.doi.org/10.1145/1132956.1132959>. – DOI 10.1145/1132956.1132959. – ISSN 0360-0300

Affidavit

I hereby declare that this thesis has been written independently by me, solely based on the specified literature and resources. All ideas that have been adopted directly or indirectly from other works are denoted appropriately. The thesis has not been submitted to any other board of examiners in its present or a similar form and was not yet published in any other way.