Department of Computer Science

Master's Thesis

The Holumbus Framework

Creating fast, flexible and highly customizable search engines with Haskell

Submitted on:
8. April 2008

Submitted by:
Timo B. Hübel
Dipl.-Medieninform. (FH)
Goldbekweg 1
22303 Hamburg, Germany
Phone: +494022629134
E-Mail: t.h@gmx.info

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

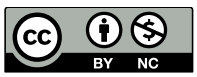
The Holumbus Framework

Creating fast, flexible and highly customizable search engines with Haskell

Master's Thesis by Timo B. Hübel

Abstract

The growing amount of information in today's knowledge society makes effective information retrieval an increasingly challenging task. While effective methods for querying large collections of documents are already established for highly unstructured data (Google Websearch, for example), these will also be necessary for very specific and structured types of data. A framework for the creation of customized search engines, specifically designed for structured data and able to handle large collections of documents by distributing the index for query processing over several machines, is presented in this work. The implementation of the framework in a relatively short time was made possible by using the purely functional language Haskell.



Revision 42 as of April 5, 2008

Layout done with the help of LaTeX 2ε, KOMA-Script and BibTeX.

Contents

List of Figures

List of Tables

Listings

1 Introduction

In today's information society, the creation, distribution and use of information has become a critical factor in economics, politics and even cultural activities. The economy in highly developed countries is often already based on knowledge and information. Therefore, efficient methods for information access are of great importance and the easy access to information becomes even more crucial for developing countries to keep up.

A study by [LV03] on the worldwide growth of information per year showed a total of five exabyte[1] of newly generated information for 2003. They have estimated that the amount of new information has doubled in between 1999 and 2002. According to the study, 92% of this new information is stored on magnetic media, primarily hard disk. This means, that most of the information is available in some arbitrary digital format and therefore can be accessed by using computers. A more recent study by [Gan08] estimates a total amount of digital information of 281 exabytes available worldwide in the year 2007 and predicts a total of about 1,800 exabytes of electronic data for the year 2011. This means, the digital universe is growing by a factor of ten every five years.

[1] This equals five million terabytes or 5,000,000,000,000,000,000 bytes and is equivalent to 800 megabytes per person.

Unfortunately, most of the digitally stored data is organized in some highly chaotic way, often just as an unstructured collection of documents. With a document being the smallest unit of uniquely addressable information, one of the basic access methods to this kind of data is the classical keyword search. It brings up all documents containing a specific keyword entered by the user, often ordered by some kind of relevance automatically determined by the system used for searching.

The importance of search has increased over the last years (compared to other access methods), as the following example will show: During the beginning of the internet era, a short and concise domain name was most important for commercial websites to make remembering and typing the name as easy as possible for the user. Today, a high ranking at Google is much more important than the domain name, because most people explore the web by using Google as starting point and the primary way of information discovery on the web. Therefore many people are already used to search functions as generic methods for information access.

1.1. Motivation

While Google is doing a fairly good job at providing easy access to highly unstructured data on the internet, it seems that there is a need for more specialized search mechanisms, which make data on specific topics more easily accessible. Such a specific search mechanism could take the inherent structure of the data into account, which perhaps would lead to better search results. This data, for example, could include all technical specification documents generated by the engineering department of a company where all documents exhibit the same structure. Another example would be a collection recipes for cooking, which always show the same structure, too. In particular, recipes can be divided into a list of ingredients and some cooking instructions. Very different results would be generated by searching only the ingredients or just the instructions.

As the amount of data on a specific topic is not necessarily small and is likely to grow in the future, such a specialized search function should be able to handle large collections of documents, just as a classical search engine for the world wide web. A solution for handling increasing amounts of data (and the resulting demand of processing power and storage capacity) seems to be parallelization and distribution, also shown by the current trend towards multicore CPU architectures. This allows partitioning of the data and distribution of the workload onto several cores and even different machines. Thus it seems that even a highly specialized search engine should be able to deal with a parallel and distributed system architecture in order to handle huge amounts of data and to meet reasonable performance requirements.

1.2. Scope

The aim of this work is to develop and implement a framework to support the creation of full text search engines in Haskell. This framework, called *Holumbus*, should be able to deal with large document collections, which will be achieved by distributing the index data and processing queries in parallel. Additionally, the framework should be capable of making the advantages of structured data (and the resulting structured index) available to the user of a search engine. This means, the known structure of the documents will be reflected in the index data and can therefore be exploited to improve search results.

Search engines created with the Holumbus framework should be able to support advanced features like *find as you type* (display of search results while typing a query), *suggestions* (retrieval of additional query terms which might be used to refine the query) or *fuzzy queries* (retrieval of sensible results despite small spelling errors). Hence, the framework and the underlying data structures need to include mechanisms which can be used to implement this functionality.

The ranking of results itself is not emphasized in this work and thus will not be discussed in high detail. Likewise, the construction of the underlying index data is not part of this work (but is described in [Sch08]). However, the design of the internal index data structures and appropriate formats for persistent storage and distribution of indices are taken into account. These are especially influenced by the advanced query features mentioned before.

An example application of the framework to a very special use case is demonstrated by providing a search engine for the APIs of Haskell libraries, just as the existing Haskell API search engine *Hoogle* [Mit] does. Of course, all advanced features like suggestions, find as you type and structural indexing will be implemented to demonstrate the capability and power of the Holumbus framework.

1.3. Related Work

Because information retrieval in general receives much attention in research, there are several recent developments regarding features like auto completion and suggestions, for example [BW06; BW07]. On the practical side, the open source framework *Lucene* [Luc], developed by the Apache Software Foundation, provides very sophisticated indexing and search technology and lately has received a lot of attention.

The advanced query features mentioned are already implemented by various kinds of software. *Google Suggest* for example [Goo], provides query suggestions based on

queries issued by other users. Find as you type is implemented by the *Firefox* browser for the full text search inside a loaded web page.

1.4. Outline

At first, some insights into the basic concepts and technology of search engines are given in chapter 2. This includes a short introduction to the field of information retrieval (IR) in general. Afterwards, the core features of a framework for the creation of search engines are analyzed in chapter 3. Some fundamental requirements are derived from these features and possible solutions are roughly sketched. The following detailed description of the implementation of the Holumbus framework with accompanying explanations of important design decisions in chapter 4 makes up the substantial part of this work. Finally, the results and lessons learned are discussed along with possible future work in chapter 5.

2
Fundamentals

This chapter provides the fundamentals on most of the techniques used in the Holumbus library. The first section gives a short introduction to the field of information retrieval and its applications. The second section focuses on the processing and optimization of the user input while the third section examines the basic data structures employed to speed up searches in large document collections. The last section introduces distribution and parallelism and outlines their use for information retrieval systems.

2.1. Information Retrieval

"Computers can compute, but that's not what people use them for, mostly. Mostly, computers store and retrieve information." This quote from [Bra07] demonstrates that efficient methods for handling information with computers are quite important. And that is exactly what information retrieval (IR) is about, it "deals with the representation, storage, organization of and access to information items" [BYRN99].

While the general properties of IR systems employing the well-known Boolean model are examined in section 2.1.1, a whole section is dedicated to search engines for large document collections. Due to the popularity of web sites like Google or Yahoo and be-

cause "search has become a dominant application for people using computers" [Bra07], these are probably the most common and well-known types of IR systems.

2.1.1. The Boolean Model

According to [BYRN99], a general model of any IR system consists of the following four parts:

- A set of representations for the documents.
- A set of representations for the user information needs (also called query).
- A framework for modeling document representations, queries, and their relationships.
- A ranking function which defines an ordering among the documents with regard to a specific query.

In terms of the Boolean model, it is assumed that every document is described by a set of representative keywords called *index terms*. These are used to summarize the document's contents. The Boolean model is based on the set theory and Boolean algebra, therefore a query is described as a conventional Boolean expression. A query for a single index term defines a resulting set, which consists of all documents relevant with respect to the index term, whereas the Boolean model only considers that index terms are present or absent in a document. The resulting sets are combined using the corresponding set operation for every Boolean operator, i.e. *union*, *intersection* and *difference*.

The concise formalism makes the Boolean model quite powerful, despite its simplicity. This has led to great attention and common use of the Boolean model in many commercial IR applications [Fer02]. While being an advantage, the unambiguousness of queries also is a major drawback of the Boolean model. The binary decision criterion for the relevance of a document leaves no room for any grading scale, which "prevents good retrieval performance" [BYRN99]. One way to deal with this problem is the so called fuzzy set model. It is based upon the fuzzy set theory, which is described in the next section in more detail.

2.1.2. Fuzzy Set Theory

Generally, "*fuzzy set theory* deals with the representation of classes whose boundaries are not well defined" [BYRN99]. This is usually done by associating a membership

function with the elements of a class, indicating the *degree of membership* of an element in a class. The membership function often takes values in the interval $[0, 1]$ with 0 indicating no membership in the class and 1 corresponding to full membership in the class. A fuzzy set is defined in [BYRN99] as follows:

> *A fuzzy subset A of a universe of discourse U is characterized by a membership function $\mu_A : U \rightarrow [0, 1]$ which associates with each element u of U a number $\mu_A(u)$ in the interval $[0, 1]$.*

The typical operators from the boolean model, *union*, *intersection* and *complement*, can be defined as follows:

> *Let U be the universe of discourse, A and B two fuzzy subsets of U and $\overline{A}$ be the complement of A relative to U. Also, let u be an element of U. Then,*

$$\begin{aligned} \mu_{\overline{A}}(u) &= 1 - \mu_A(u) \\ \mu_{A \cup B}(u) &= max(\mu_A(u), \mu_B(u)) \\ \mu_{A \cap B}(u) &= min(\mu_A(u), \mu_B(u)) \end{aligned}$$

In information retrieval, a fuzzy model can be deduced from this theory by considering that each query term defines a fuzzy set and that each document has a degree of membership in this set. This allows for the scale of document relevance to be gradual, instead of abrupt as in the Boolean model.

One approach to generate a fuzzy set from a query term is to adopt a thesaurus. This thesaurus defines term relationships and therefore allows the construction of a set of related terms to a given query term. This way, the user query can be expanded by additional terms from the thesaurus. Hence the expanded query allows for the retrieval of additional documents, which may be relevant for the user and otherwise would not have been retrieved.

2.1.3. Search Engines

As stated before, search engines are probably the most known type of IR systems. Basically, they are specialized IR systems which model the World Wide Web as full-text database [BYRN99]. The main difference is, that all queries must be answered without accessing the original text, only based on the indices or heavily reduced cached documents. Most search engines are based on a crawler-indexer architecture, which is shown in figure 2.1.

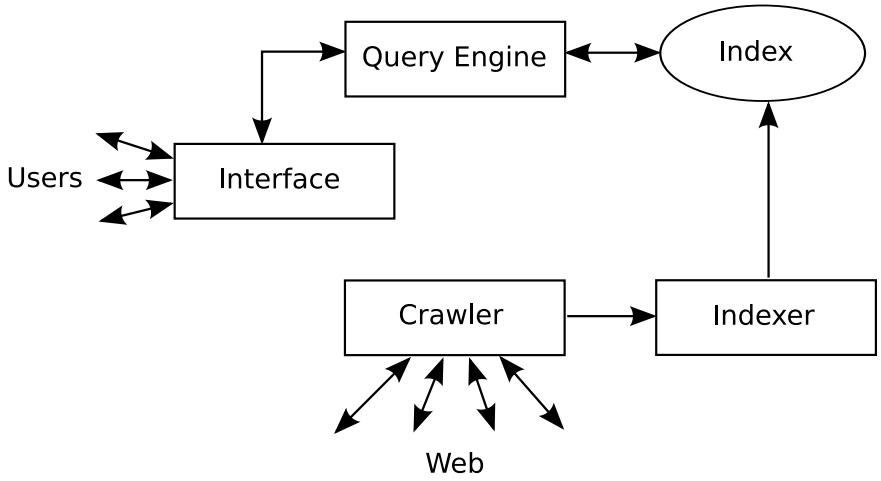


Figure 2.1.: *A typical crawler-indexer architecture (from [BYRN99]).*

The architecture has two basic components: One that deals with the users, made up from the user interface and the query engine and another resembling the backend, consisting of the crawler and indexer. Their common point of interaction is the index, which is created by the backend part and is used by the user part to answer queries.

The crawler (also called spider or robot) traverses the Web and collects the documents. Each document is then processed by the indexer, which inserts the index terms into the index. The user enters a query through the interface, which is then processed by the query engine. The engine uses the index to retrieve the documents and applies a ranking mechanism. The (ordered) list of results is then presented to the user through the interface. The single steps of processing a query are discussed in detail in the following section.

2.2. Query Processing

Processing a query involves four basic steps: Preprocessing the user-input into some internal query structure, optimizing this query structure to speed up the retrieval process, the query evaluation itself and finally ranking the results by relevance. As the query evaluation heavily depends on the index structure used, it will not be discussed any further in this section.

2.2.1. Preprocessing

Most IR systems (especially search engines) allow the user to formulate a query in a somewhat arbitrary syntax understood by the system. This query string has to be

parsed afterwards to extract tokens of the query language. These tokens are either query terms or operators. A term consists of just a single word, of which the user wishes that it is in some relationship with the documents to be retrieved. Operators are keywords which define how the retrieved documents are processed. Widely known operators are:

- Phrases ("what is the question")
- Boolean operators (AND, OR, NOT)

Several other types of operators are available, such as operators influencing the case-sensitivity of terms or specifying a term as prefix term. The potential set of operators depends on the implementation of a search engine as well as the index data available.

During the preprocessing of a query, several other techniques to improve retrieval quality can be employed. For example, so-called *stop words* can be removed. These are words which are very common among the document contents but do not contribute much to the meaning of the text (for natural language these would include words like *the*, *and* or *a*). They usually cause high retrieval costs, because the number of associated documents tends to be very large. Therefore, omitting them increases retrieval efficiency and possibly retrieval quality, too.

Another example would be a process called *stemming*. This means the reduction of every query term to its stem by removing its prefixes and suffixes (originating in plurals or past tense). For example, the words *connected*, *connection*, *connecting* and *connection* would all be reduced to *connect*. This process also has to be performed during the indexing of documents which has the secondary effect of reducing the size of the index structure. Although stemming is thought to be useful for improving retrieval performance, there is controversy about the real benefits (see [BYRN99]). Because of these doubts and the lack of efficient stemming algorithms for languages other than English, it is seldomly used in practice.

2.2.2. Optimization

After creating the internal structure representing the query, it is possible to transform the terms and operators to reduce the time needed for query evaluation while the semantic meaning of the query remains the same. This process is called *query optimization*. A very simple example of an optimization is the removal of any duplicate terms. Good algorithms for query optimization are capable of algebraic transformations (in terms of Boolean algebra) and can predict if a query will yield any results. This way, senseless queries like "foo AND NOT foo" can be eliminated.

No matter what kind of optimization is performed on the query, its cost must not be higher than the expected savings during query evaluation. Otherwise it would be more efficient not to optimize the query at all.

2.2.3. Ranking

If a query returns a really large set of documents, the user will not be able to recognize the results relevant for his information needs. Most IR systems therefore employ some kind of ranking mechanism. This provides the user with a preselection of results by automatically calculating the relevance of each document. Afterwards it is possible to select just the best results. Sometimes relevance information is already available from the query evaluation, for example through the membership function from the fuzzy set model (see section 2.1.2). However, in most cases this information has to be calculated explicitly.

Several ranking methods are known throughout the literature, with the *word frequency* being the most simplest one: For every document, occurrences of every query term in its content are counted and summed up. This number then is used to determine the document's relevance. Unfortunately, this method is heavily influenced by the length of a document, because the number of occurrences of a given word potentially increases with the length of the document.

Two other ranking methods have there origin in the vector space model for IR systems [BYRN99; WMB99]. This model represents documents and queries as vectors in n-dimensional space. The relevance of a document can then be determined by calculating either the cosine of the angle between the two vectors (called *cosine measure*) or the reciprocal of the familiar Euclidean distance between vectors.

Another method lately receiving great attention is the *PageRank* algorithm employed by Google [BP98]. It takes the hyperlink graph between websites into account and achieves very good retrieval quality. Unfortunately, this method is restricted to hypertext documents or documents with an equivalent linking structure.

2.3. Indexing Strategies

Several methods and data structures for organizing and storing indices of text collections have been developed over time. They share the aim to speed up the processing of a specific type of query compared to a simple sequential search. The most important one, the inverted file (or inverted index), is presented in section 2.3.2. According to [BYRN99], it is "currently the best choice for most applications".

Most indexing techniques can be built upon different basic data structures, from which the trie (pronounced "try") is "heavily used" [BYRN99] and probably the most interesting one. It is described in detail, along with another structure closely related to tries, in section 2.3.1. For the sake of completeness, two other important structures should be mentioned here, but will not be described any further: The suffix-tree and its more space efficient counterpart, the suffix-array. The former was introduced by [MM93] and both are quite common, too.

As the document collections to be indexed potentially can be infinitely large, several techniques to optimize the space efficiency of inverted files exist. A quite recent compression scheme employs so-called word-aligned binary codes, of which one is explained in section 2.3.3.

2.3.1. Trie and Patricia

According to [Knu98], a trie is "an M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters". The interesting thing about tries is the fact, that they use every single bit of information to speed up the search tremendously, precisely to $O(l)$ with l being the length of the key to search for. Unfortunately, space efficiency is traded in to achieve these speeds, which gives the trie a worst case space complexity of $O(n^2)$.

In a very simple trie of strings, each node corresponds to a single character of a key. Starting at the root node, a node is inserted for the first character of the key. If there is a node for this character already present, it will be selected and the next character of the key will be considered. The next node will be appended to the current node, again any existing node for the character will be reused. This process continues, until there are no characters left. The node for the last character will be marked as a special node, denoting the end of a key (when the trie is used as associative storage, a value for this key can be stored inside this node). The key of an arbitrary node corresponds to the sequence of characters on the path from the root to the node itself. Unfortunately, in the worst case a single node is required for every character in the whole trie. Even the simple example in figure 2.2 (left) illustrates the large number of nodes necessary to store just five keys.

The search on a trie is performed by following the branches in the tree for each character in the query string one by one. If no branch matches the current character or the last character led to a node not denoting the end of a key, the search did not succeed. Otherwise, if the process ends with the last character leading to a node denoting the end of a key, exactly this key is found.

A special case of the trie is the "Patricia" data structure, introduced by [Mor68]. The name stands for "Practical Algorithm To Retrieve Information Coded In Alphanumeric". It was originally based on a binary trie which avoids one-way branching by including positional data for each node. This additional information specifies the next significant bit which determines the branching for subsequent nodes.

The Patricia structure can be generalized for M-ary tries by combining consecutive one-way branches of nodes, which do not denote the end of a word, into one single node. This means, each node does not just store one character anymore but instead a sequence of successive characters. The example in figure 2.2 (right) shows how five nodes can be saved by combining the one-way branches. The tradeoff in the Patricia structure lies in a slightly higher time complexity of the insert operation, because the nodes have to be splitted correctly instead of just appending a node for every character.

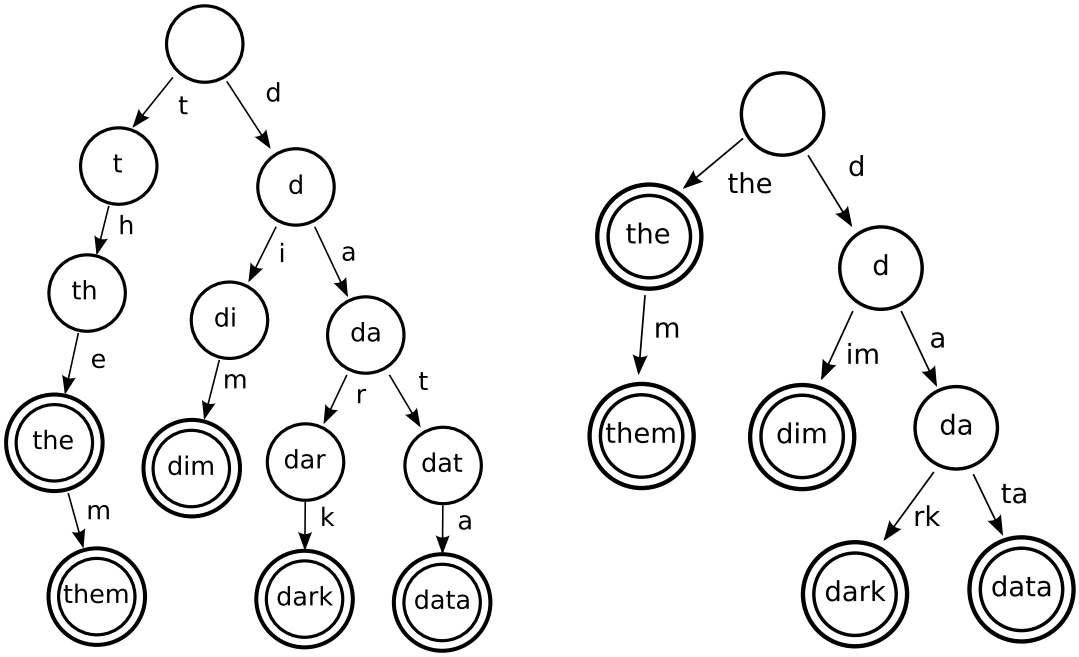


Figure 2.2.: *A trie (left) and a Patricia structure (right) containing the words* dark, data, dim, the *and* them.

One advantage in both the normal trie and the Patricia data structure lies in their ability to provide a prefix search very efficiently. All keys with the same given prefix can be found by just searching the prefix and recursively returning all successors of the node where the last character of the prefix was matched.

2.3.2. Inverted File

The inverted file (also called inverted index) is a mechanism to speed up the search for words inside a collection of text documents. The inverted file stores a set of all different words in the collection and for each of these words a list of positional information where the word appears inside the collection. The set of words is often referred to as *vocabulary* while the set of all lists with positional information is called the *occurrences*.

The positional information can refer to documents, words or even single characters. Often, the words in the index are referencing lists of documents (each represented by a unique identifier), where every element of these lists references another list of positions (the positions of the word within the document). This specific type of index is often referred to as *full inverted index*. An example of this kind of inverted file is shown in figure 2.3.

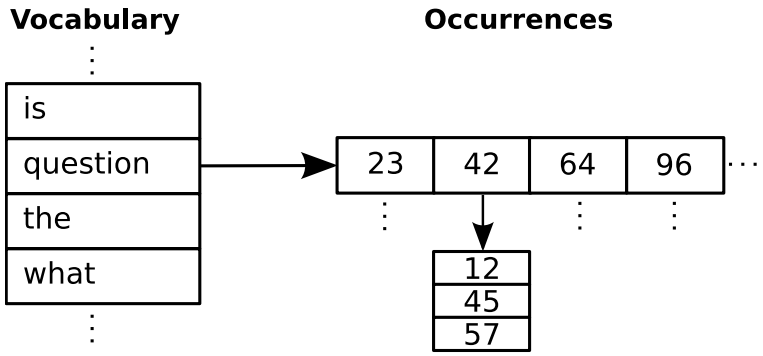


Figure 2.3.: *A sample application of the inverted file, where a word references a list of documents (represented by unique identifiers) which itself reference a list of positions.*

According to [BYRN99], the space required for the vocabulary is rather small, about $O(n^{\beta})$, where n is the size of the document collection and β is a constant between 0 and 1 dependent on the characteristics of the text, ranging between 0.4 and 0.6 in practice. In contrast, the occurrences require much more space. Because the position of every word appearing in the text is referenced once in the occurrences, the space requirement is $O(n)$. In practice, a space overhead between 30% and 40% of the original text size is estimated in [BYRN99].

The specific data structure used for the vocabulary and the occurrences can be any arbitrary list structure. Usually a structure providing fast lookup is used for the vocabulary, like a binary tree or a trie (see section 2.3.1). According to [BYRN99], even storing the words as plain list in lexicographical order is very competitive in terms

of performance when using binary search for lookup. As the occurrences contribute most to space consumption, a structure focusing on compact representation of data is more likely to be used.

The algorithm to search an inverted file is described in [BYRN99] through the following three general steps:

1. *Vocabulary search* Every word in the query is extracted and searched in the vocabulary. More complex queries like phrases are split into single words.

2. *Retrieval of occurrences* For every word found, the corresponding list of occurrences is retrieved.

3. *Manipulation of occurrences* The occurrences are processed to solve phrases or Boolean operations.

Thus a single word query can be answered at costs independent from the text size (depending on the data structure used for the vocabulary), which is the reason for the popularity and common use of the inverted file. Still, the inverted index has some drawbacks, most importantly, it consumes quite some space as the document collection grows larger. Some techniques for index compression exist to alleviate this problem, of which one is explained in more detail in the next section.

2.3.3. Compression

Several compression schemes for inverted indices have been developed over time, most of them being quite complex. Generally, "for the majority of practical purposes, the most suitable index compression technique is the local Bernoulli method, implemented using a technique called Golomb coding" [WMB99]. This is still true regarding compression efficiency, although one of the authors of the aforementioned book created a new set of techniques, the so-called word-aligned binary codes. These are almost as good in compression as the Golomb coding but run much faster and are easier to implement [AM05].

All of the codes are based upon the idea to put several values into one machine word. This alignment allows for considerable speed improvements compared to bit-stream oriented compression schemes. The simplest one is called *Simple-9* and works as follows: Different words store different amounts of values, but within each word each value is represented using exactly the same number of bits.

Within a single 32-bit word, the number of data bits is signalled through four selector bits (usually the first four bits), while the remaining 28 bits contain the values. This

makes up for twenty-eight 1-bit values, or fourteen 2-bit values, or nine 3-bit values and so on, up to one single 28-bit value. All possible combinations (including the number of possibly wasted bits) are listed in table 2.1. While this limits the upper bound to $2^{28} - 1$, the scheme can be easily extended to support 64-bit words (storing values up to $2^{60} - 1$).

It should be obvious, that compression is most effective for small values (the smaller the values, the more values fit into one word). Therefore efficiency can be improved substantially, if the original values are encoded through their differences. This means, a sequence of values will be sorted in ascending order and for every value only the difference to its predecessor is stored. The first value is stored unchanged (its predecessor is zero). Of course, this is only possible if there is no meaning in the ordering of values. For example, the list

$$\langle 4, 10, 11, 12, 15, 20, 21, 28, 29, 42 \rangle$$

will be transformed to

$$\langle 4, 6, 1, 1, 3, 5, 1, 7, 1, 13 \rangle.$$

Still, it is possible to further improve the Simple-9 compression scheme: As mentioned in table 2.1, some of the combinations do not use all available bits, and even the four selector bits could represent sixteen different values but only nine of them are used. In [AM05], some modifications to the Simple-9 scheme, called *Relative-10* and *Carryover-12*, are presented, which aim to work around these issues. While the compression efficiency is only slightly improved, they are quite more complex than the Simple-9 scheme.

Selector	*Number of codes*	*Length of each code (bits)*	*Number of unused bits*
a	28	1	0
b	14	2	0
c	9	3	1
d	7	4	0
e	5	5	3
f	4	7	0
g	3	9	1
h	2	14	0
i	1	28	0

Table 2.1.: *Nine different ways of using 28 bits for flat binary codes (from [AM05]).*

2.3.4. Distribution

The staggering volume of electronic text available online today makes the management of large document collections with IR systems increasingly expensive. One way to support these demanding requirements are parallel and distributed retrieval techniques which enable concurrent query evaluation and distributed storage of index data.

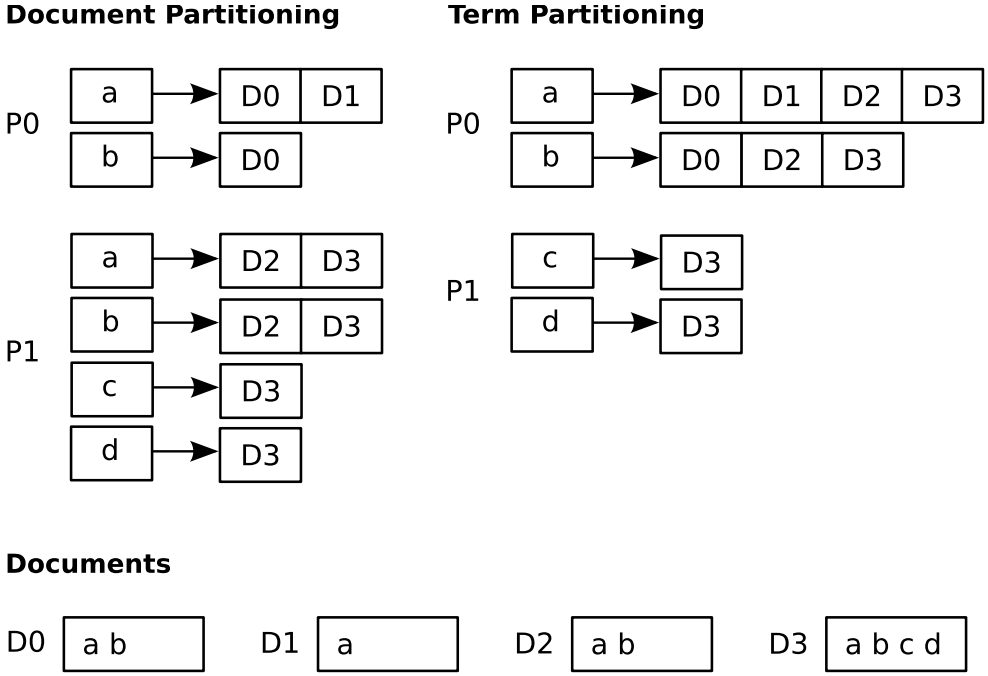


Figure 2.4.: *Different partitioning schemes for an inverted file index containing four documents (D0, D1, D2, D3) and four terms (a, b, c, d). The index is splitted into two parts, notably P0 and P1.*

The main prerequisite for concurrent query evaluation to search very large document collections is the partitioning of the index data. Two main partitioning schemes are possible for inverted files in distributed environments [BYRN99]: *document partitioning* and *term partitioning* (or, as they are called in [RNB98], *local inverted files* and *global inverted files*). The former one partitions the index data into disjoint subsets of documents while the latter one partitions the index data into disjoint subsets of index terms. An example of both strategies for the same set of documents is shown in figure 2.4.

Splitting the index data by the documents seems to be straightforward, as each part can be treated as a completely independent index. But it is stated in [RNB98] that term partitioning outperforms document partitioning due to the greater concur-

rency among queries. Additionally, different physical partitionings of the inverted lists are possible within the whole system, but the results in [TGM93] seem to suggest that a distinct index containing a disjoint subset of the document collection for every computing host (for every CPU or core) delivers the best results.

One problem arising from splitting the index data by the documents is poor ranking quality, if no additional measures are taken. This is due to the fact that a combined vocabulary exists in no place. Despite this disadvantage, [dMSZ98] found that distribution does not only help scalability but can also lead to faster query response times, with network delay becoming the main performance bottleneck.

3 Analysis

At the beginning of the development, there had just been a vague guess about the possible need for highly customizable search engines to satisfy society's increasing information needs. This quickly turned into the idea of the implementation of a flexible framework for the creation of search engines. To ensure high adaptability of the framework, some basic requirements were conducted from this idea. They are presented in this chapter, together with references to possible implementations.

One basic constraint, which was fix from the beginning, was the use of the pure functional language Haskell. On the one hand, this decision was made to be able to implement a fairly complex system in a very limited amount of time, because functional programming can boost the productivity of the programmer by an order of magnitude [Hug89]. On the other hand, it was just chosen to show the use of Haskell "in anger" [Wad98] and to provide another example of real-world problems solved in Haskell.

Another decision, which was made quite early, was the restriction to one type of index structure. Because the *inverted file* (as described in section 2.3.2) has proven itself for high scalability, good performance and ease of implementation, it was selected as the basic index structure for the framework. This has also been confirmed by a recent comparison of the inverted file with other index structures in [Sch07]. Although the inverted file has lots of advantages, the framework was still designed to provide extensibility for other types of indexes.

3.1. System Requirements

The system requirements define the general properties of the whole system. They should be met by almost every module of the framework. These are either the general constraints, like *flexibility* and *customizability* or the more technical aspects, which influence the design of the whole system, like *distribution*.

3.1.1. Flexibility

Building a search engine involves several technical aspects, which can be solved in many different ways. As the "best way" to take on a problem heavily depends on the characteristics of the application, the decision on how to implement any specific aspect should therefore be left to the user of the framework. This requires a very open API design, which enables the user to employ only those parts of the framework he really needs. Or, to put it the other way round, it has to be possible for the user to replace almost every part of the framework with a customized implementation.

In general, it should not be a problem if a user only wants to utilize the indexing data structure and supplies a custom search engine or if another user wants to provide a custom parser for the query grammar to exclude specific operators. Almost every part of the framework should be usable independently from the others.

But in general, this coarse kind of flexibility, which involves custom implementations of whole parts of the framework, can only be the last resort for the user. There should be the possibility of much more fine granular customizability, which is discussed in the next section.

3.1.2. Customization

Often, users only need to adjust very small and specific parts of some functionality provided by a given framework. This does not involve custom implementations of complete modules by the user but just tweaking the behavior of the framework at very limited spots. In object-oriented programming, this is traditionally achieved by deriving a class, inheriting all the default behavior from the framework and only overwriting the parts which have to be customized.

In functional programming, functions are treated as first-class values. This means, they can be passed around like regular values, either as a function argument or as a return value. Therefore it is possible to create functions which can be parameterized with a user-supplied function, which is responsible for a specific part of the computation. This concept allows for highly customizable but easy to use functionality.

Another technique to increase customizability employed by Haskell are polymorphic types and type classes (see [Pey03] or [Hut07]). Polymorphic types can be used to customize the signature of a function and will make it available for different types while preserving type-safety. This mechanism can be used to enable the user to provide his own customized types (and therefore customized data) for interaction with the framework.

A type class provides a set of operations which must be supported by any type that is an instance of that class. They could be seen as equivalent to interfaces in Java. A class enables the user to make his own type an instance of the class and therefore provide customized implementations for customized types of data to the framework. This requires several parts of the framework to be designed to work with type classes instead of distinct types.

In total, all of these features can be used to provide a lot of customizability for crucial parts of the framework without requiring the user to implement complete modules on his own. The granularity can thereby range from small specifically tailored functions to custom instances of type classes.

3.1.3. Distribution

As already mentioned in the introduction, the amount of information available grows at tremendous speeds. This is why scalability with the size of the underlying document collection of a search engine using the framework is a very important requirement. One way to achieve scalability (as in *just add hardware*) is distribution of the information retrieval system over multiple machines. These can perform parts of the necessary calculations in parallel and could provide distributed storage for large amounts of data as well.

To retain these possibilities for scalability, the creation of facilities for index distribution became an important design goal (because this is the basic requirement for distributed information retrieval, as explained in section 2.3.4). Due to the fundamental nature of the index data structure for the entire system, the design of almost every aspect of the framework is influenced by this decision. To restrict possible distribution schemes in no way, there has to be at least support for *document partitioning* and *term partitioning*.

Because just being able to distribute the index makes no distributed information retrieval system, facilities for distributed query evaluation are necessary, too. These imply parallel computation of query results which is supported through Haskell's easy to use abstractions for parallelism (as described in [JH93], for example). At the same

time, tools and utilities are required for the maintenance of distributed indices. For example, it has to be possible to add or remove documents from an index which is distributed over several machines.

3.2. Search Requirements

In addition to the general system requirements, there were very specific requirements for the search functionality itself. These are mainly related to the types of queries to be supported by the system. As most users are already familiar with search engines using the Boolean model (as explained in section 2.1.1), the framework should at least provide support for this model and its operators *AND*, *OR* and *NOT* (which means support for *intersection*, *union* and *complement* operations on result sets). Additionally, some advanced query techniques should be supported, too. These are described in the following sections.

3.2.1. Prefix Queries

The basic type of a query in traditional search engines is a single full word. Searching for "foobar" returns all documents containing the single word "foobar". Quite recently, several search engines adopted a technique called *suggestions* (for example, there is the already mentioned *Google Suggest* [Goo]). This means a way to provide the user with hints on possible completions for his query which lead to good search results. Another relatively new technique is called *find as you type*, which is, for example, employed by the *Firefox* browser for searching inside a web page. It already shows possible results to the user while the query is still being typed. This is often done by regarding the already typed letters as partial query and evaluating it upon every keystroke. Both techniques can help the user to formulate his queries more efficiently and can greatly improve the interactivity of the retrieval process. Therefore, the framework should provide mechanisms to support these features.

A simple way to support both techniques is through regarding a single word entered by the user as *prefix query*. This means, the word is substituted by all its possible completions and therefore virtually resembles a set of words. The result of the query contains all documents containing any of the words from this set, while the set itself can be used to generate suggestions. As this evaluation can take place on a per-character basis, it also allows implementation of the *find as you type* technique.

Of course, the search for prefix terms has to be supported by the underlying data structures. The trie structure explained in section 2.3.1 can be used for the dictionary

of an inverted index to enable the retrieval of possible completions for a prefix term and respective documents. The *find as you type* technique requires queries to be evaluated fast enough to show the result to the user before the next letter is typed. Luckily, the trie structure provides very fast lookup in just $O(l)$ time (with l being the length of the prefix term). Hence the time complexity of query evaluation becomes independent of the size of the document collection (and probably fast enough for *find as you type* being very responsive, even with large document collections).

3.2.2. Phrase Queries

Another basic type of query known to most users of search engines is the *phrase query*. This denotes a query for an exact sequence of words, for example "where is foobar". The query enables the user to provide proximity information about the words to further restrict the result of the query (as opposed to a query just concatenating the terms through the *AND* operator). This query requires every document of the result to contain all of the words entered by the user in that exact order. Because phrase queries offer the user a way to specify his information needs far more precisely, the framework should contain appropriate support for this functionality.

To support phrase queries, the index structure has to contain positional information about the words. Otherwise, the system would not be able to detect that a word is located directly after another. As described in section 2.3.2, the inverted file can easily be extended to store the positions of words. The resulting index is then sometimes referred to as *full inverted index*.

A notable side effect of providing phrase queries is the possibility for the user to intentionally prevent a word from being regarded as a prefix query. If just one word is specified as a phrase, this returns the identical results as if a search for this exact word without its possible completions was issued.

3.2.3. Structural Queries

Often the searched documents are inherent to some kind of structure consisting of well defined parts, like a title, some headlines and plain content. This additional information about a document could be used to improve retrieval quality, if the user is able to specify his information needs more precisely by including structural information in the query. This is usually done by providing the user a possibility to restrict the query to specific parts of the document (called *contexts*). In addition, the ranking algorithm could benefit from this extra information by weighting the retrieved documents by

the different contexts containing the query terms. Therefore, the framework should support structural indexing as well as structural queries.

A very simple way to achieve this is the annotation of every index term by prepending the word with a symbolic name of the originating context, separated by a special character. If the user prepends a word in his query in the same way (e.g. "title:foobar"), the word is only searched in that specific context. This method imposes some major drawbacks, for example, many unnecessary characters are added to the index and it is not possible to search several context at once.

One approach to overcome these problems is to create several different indexes, preferably one for each context. Although the solution imposes some additional space overhead, this disadvantage is outweighed by increased flexibility and even another option for distribution index data. Instead of splitting by words or documents, it is possible to split by contexts. Both techniques require the underlying mechanism for index generation to be able to extract the structural information from the documents in the collection.

3.2.4. Fuzzy Queries

As explained in section 2.1.1, the most important drawback of the Boolean model is caused by its binary decision criteria. Unfortunately, it prevents the retrieval of documents which could possibly satisfy the user's information needs but do not match the query terms exactly. In particular, it makes the system return empty result set if the user just incorporates small typing errors in the query (like duplication of a character or swapping two adjacent characters). Likewise, the user is forced to reformulate and execute a query several times if there is some uncertainty about the exact spelling of a word.

One way to leverage this problem is through query expansion. This means, each query term is replaced by a number of terms similar to the original term. These replacement terms are concatenated by the union operator (typically OR). For example, the query

$$what\ AND\ where$$

could be expanded to

$$(what\ OR\ waht\ OR\ wat)\ AND\ (where\ OR\ wehre\ OR\ were).$$

The main difficulty of this solution is the generation of sensible replacement terms, because retrieval quality heavily depends on the replacements. For instance, if a

replacement “where” is defined for the term “were”, but another sensible replacement like “weren’t” is omitted, the resulting set of documents is very different from the one retrieved as if “weren’t” was included in the replacement definitions.

The easiest but also most inflexible solution is the use of a static thesaurus, which contains predefined replacement terms for specific words. If the thesaurus should contain sensible replacements for almost every word including common typing errors, it would grow very large and the process of query expansion becomes quite inefficient.

A similarity function, as explained in section 2.1.2, can be used to measure the variation between the original term and a replacement term. If the set operations are defined in terms of the *fuzzy set theory*, this allows for a gradual decision criteria as compensation for the binary decision criteria from the Boolean model.

4 Implementation

This chapter describes the implementation of the Holumbus framework. For every requirement listed in the previous chapter, the technical realization is explained in detail. This includes descriptions of problems encountered and how these have affected and influenced design decisions. The last section describes the exemplary use of Holumbus for the implementation of a highly specialized search engine, a full-featured Haskell API search.

4.1. Environment

Development was performed under 64-bit GNU/Linux 2.6.23 using the Glasgow Haskell Compiler (GHC) 6.8.2 (available from [Ghc]). The GHC seemed to be the most widely adopted Haskell implementation and is sometimes even regarded as the standard implementation.

4.1.1. Advantages of Haskell

Despite the general advantages of functional programming, like boosting the productivity through higher-order functions and function composition or the eligibility for parallel programming through the lack of side effects, there are some benefits which are pertinent to Haskell.

First of all, Haskell is a strong and static typed language that comes with a sophisticated type system. Every compilation of Haskell code is preceded by a step called *type inference*. During this phase, the types of all values derived from the eventual evaluation of expressions are automatically, either partially or fully, deduced. This allows for very extensive type checking at compile-time, eliminating all possibilities of respective run-time type errors.

Another unique feature of Haskell is its *lazy evaluation* of expressions. This delays the computation of the result of an expression until such moment as the result of the computation is known to be needed. There are several advantages resulting from this evaluation strategy. First of all, a significant performance increase is possible by avoiding unnecessary calculations. Secondly, it is possible to construct infinite data structures, like a list containing all positive integers, which allows very elegant solutions for some problems. Other benefits are the ability to define control structures as regular functions rather than built-in primitives and avoiding error conditions when evaluating compound expressions.

A well-known problem of pure functional languages is interaction with the rest of the world through I/O operations, because these almost always depend on side effects. Haskell solves this through a mathematical concept called *monad*, which allows the integration of I/O operations into a purely functional context (again, see [Pey03; Hut07]).

4.1.2. Tools and Libraries

The Holumbus framework makes extensive use of several libraries and tools to facilitate the development process. Some of these provide unique features or outstanding value and are presented in this section.

First of all, the *QuickCheck* [CH00] tool was used throughout the entire development process for automatic creation of test cases. The tool provides easy to use combinators for the definition of invariants for any function, which must hold for arbitrary test data. QuickCheck automatically creates testcases from these invariants by checking them with randomly generated input data. This easy way of testing greatly improved test coverage and therefore helped (together with some specialized unit tests) to gain confidence in the code.

Another library extensively used throughout the framework (especially for the processing of input documents and persistent storage of index data) is the *Haskell XML Toolbox* [Sch], a collection of tools for processing XML with Haskell. It provides a generic data model for representing XML documents and includes filters and com-

binators for processing this kind of data. The library provides *pickler combinators* [Ken04] for easy-to-use de-/serialization of arbitrary data types from/to XML. This mechanism is used to provide a human readable format for persistent indices, mostly used for debugging.

As XML for persistent storage of large data quantaties notoriously tends to be very inefficient, a binary format for persistent indices is provided, too. This is based on the famous *ByteString* library [CSL07], which provides a time and space efficient implementation of byte vectors. De-/serialization from/to ByteStrings through easy to use Put and Get monads is provided by another library (Data.Binary in particular) and used for encoding and decoding of binary data. As several compression codecs are provided on the basis of ByteStrings, this transformation is also used for the compressed transmission of query and result data over the network for distributed query processing.

The framework includes a parser for the default syntax of the employed query language. This parser is implemented using the *Parsec* library [LM01], which provides monadic parser combinators. These allow easy construction of parsers using the same language as the remaining parts of the program (as opposed to parser generators like *Yacc*). The main advantage of this is the fact that parsers are treated as first-class values within Haskell, allowing them to be passed around as arguments or return values.

The build system used for packaging of the core library is *The Haskell Cabal* [Cab]. It provides a unified infrastructure and common interface for package authors and distributors to build Haskell applications and libraries in a portable way. The remaining parts of the framework are built using the make mechanism of GHC and makefiles for standard GNU make.

4.2. Framework Structure

The Holumbus framework established is divided into three main parts with the core library providing most of the functionality. The examples contain some small utilities as well as more widely usable example applications like a simple command line search and a query server for distributed search engines. The third part contains all test modules, specialized unit tests as well as properties for the QuickCheck tool. This section presents the basic structure of the framework along with respective interfaces and modules from the core library together with the most important utilities. A basic introduction to the use of the Holumbus library is given in appendix A.

4.2.1. Basic Structure

The core functionality of the framework is composed from components covering two different areas: The components for creating the core data structures from a document collection (described in [Sch08]) and the components for building a search engine on top of the resulting index (described in this work).

The points of interaction between these two areas are the core data structures, explained along with the components of the query part of the framework in this section. An overview of the structure of Haskell modules can be found in table 4.1.

Modules	*Description*
Holumbus.Build.*	Modules for creating an index from a document collection.
Holumbus.Control.*	Generalized control structures (mainly for distribution).
Holumbus.Data.*	Basic data structures commonly used throughout the framework.
Holumbus.Index.*	Definitions of index interfaces and types.
Holumbus.Query.*	Modules for creating a search engine on top of an index.

Table 4.1.: *The hierarchical structure of Haskell modules in the Holumbus framework.*

The structure of components in the query part of the framework is derived from the basic steps of the process of answering a search request. These steps are:

1. Parsing of the input string to build the query.
2. Evaluation of the query using a provided index.
3. Ranking and transformation into the final result.
4. Display to the user.

These four steps make up the components of the query part of the Holumbus framework, with the query processor for evaluating a query being at the core. It is accompanied by the parser, which produces the internal query structure, the ranking mechanism, which evaluates the relevance of results and a display function, which presents the results to the user.

These components (except the parser) are built upon three core interfaces to the respective data structures. The most important one is *HolIndex*, representing the index which contains the associations between words and documents. To increase space efficiency, a document is represented by an artificial identifier inside the index.

The *HolDocuments* structure maps these identifiers back to the original informations about the documents. The third one, *HolCache*, enables retrieval of the (simplified) full text for any document, which can be used for displaying previews of the documents in the search result.

The data structures are created from a document collection by the indexing part of the framework, which is not part of this work, but is described in [Sch08]. The overall structure is illustrated in figure 4.1. From these components, only the processor is mandatory while the others are optional. This enables the user of the framework to pick just the functionality required to implement a search engine satisfying his specific information needs.

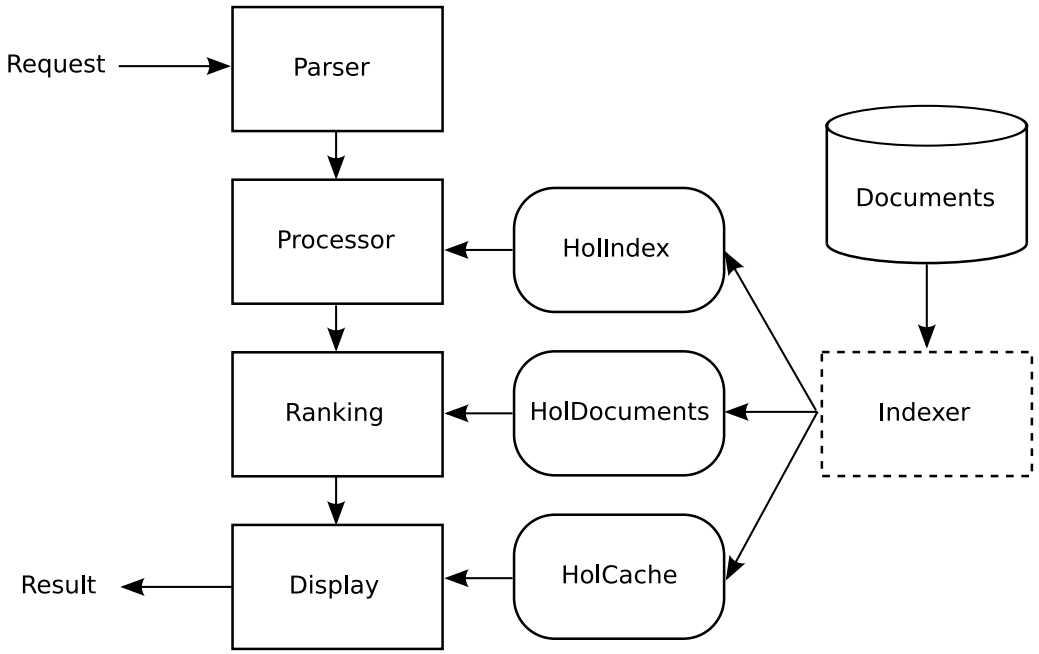


Figure 4.1.: *The overall structure of the Holumbus framework.*

The three core data structures are implemented using type classes to provide a unified interface to different implementations. This also allows the user of the framework to implement custom variations of these structures. The interfaces are presented in the following section.

4.2.2. Interfaces

The type classes for the three core data structures are built upon a number of basic types, which are shown in listing 4.1. While most of these are just aliases for primitive types, some are worth a more elaborate explanation.

The Document type represents a single document and is implemented as a polymorphic type. Apart from the mandatory information about a document (particularly the title and URI), it can contain some optional customized information where the specific type is provided by the user of the framework. Originally the Document type was implemented by just using a tuple (Title, URI), but as this does not allow for custom information about a document, the implementation was changed.

The occurrences of a word are represented by a mapping from document identifiers to sets of word positions. To provide phrase queries as described in section 3.2.2, the word positions have to be included in the occurrences (and therefore in the index). The Occurrences type is implemented using predefined containers provided by the Haskell hierarchical libraries. Because these are limited to integers, they provide very fast set operations like union, intersection and difference, which is important for the implementation of the query processor.

The RawResult type is used as return value for the query functions of the index interface. Because the index has to support prefix queries (as described in section 3.2.1), these functions do have to return possible completions together with the respective occurrences. To avoid the overhead of a more sophisticated map structure and to achieve maximum flexibility, RawResult is defined using the primitive structures provided by Haskell, particularly as a plain list of tuples.

```
data Document a = Document
  { title   :: Title
  , uri     :: URI
  , custom  :: Maybe a
  }

type DocId          = Int
type URI            = String
type Title          = String
type Content        = String
type Position       = Int
type Context        = String
type Word           = String

type Occurrences    = IntMap Positions
type Positions      = IntSet

type RawResult      = [(Word, Occurrences)]
```

Listing 4.1: *The definition of basic types used throughout the framework.*

The most important and at the same time most extensive interface is the HolIndex class, as shown in listing 4.2. It defines several functions, which can be roughly grouped

into three areas: Functions which return information about the index, e.g. the size in terms of the number of words, functions used to query the index which return the occurrences of a word, and functions to manipulate the index, for example by inserting or removing documents and their words. Most of the functions employ a Context argument, which identifies the originating document part for a word. These context descriptions are used to implement structural queries as explained in section 3.2.3. As a consequence, occurrences of a word can only be inserted with respect to a specific context and at the same time the index can only be queried for a word with respect to a specific context.

```
class Binary i => HolIndex i where
  sizeWords      :: i -> Int
  contexts       :: i -> [Context]

  allWords       :: i -> Context -> RawResult
  prefixCase     :: i -> Context -> String -> RawResult
  prefixNoCase   :: i -> Context -> String -> RawResult
  lookupCase     :: i -> Context -> String -> RawResult
  lookupNoCase   :: i -> Context -> String -> RawResult

  insertOccurrences :: Context -> Word -> Occurrences -> i -> i
  deleteOccurrences :: Context -> Word -> Occurrences -> i -> i

  insertPosition :: Context -> Word -> DocId -> Position -> i -> i
  deletePosition :: Context -> Word -> DocId -> Position -> i -> i

  mergeIndexes      :: i -> i -> i
  substractIndexes  :: i -> i -> i

  splitByContexts   :: i -> Int -> [i]
  splitByDocuments  :: i -> Int -> [i]
  splitByWords      :: i -> Int -> [i]

  updateDocIds :: (Context -> Word -> DocId -> DocId) -> i -> i
```

Listing 4.2: *The interface definition for Holumbus indices.*

The functions for querying an index are available in several variations. These allow for prefix queries as well as exact word queries, both in case-sensitive and case-insensitive fashion. The allWords function, returning all words and their occurrences for a specific context, is required for the support of single negation queries like "NOT foobar" (all documents except those containing the word "foobar").

Of the functions for the manipulation of indices, mergeIndexes and substractIndexes are most notable, because in most cases the implementation of the insert/delete operations should be possible in terms of merging/substracting indexes. In fact, the functions

for inserting/deleting a single position are already defined in the class in terms of insertOccurrences / deleteOccurrences. As merging several indices into one large index incurs possibly conflicting documents (different documents with the same identifier), a generalized function for the modification of document identifiers is provided. This allows to sort out all conflicting identifiers by re-assignment before merging the indices.

To support the distribution of index data, any index has to support some split functions. These can be used to create a number of smaller indices from one large index. As splitting is possible in several ways, there are different functions for splitting by documents, index terms or contexts (see also section 3.1.3).

Functions for the creation of an index are intentionally excluded from the interface. The reason is that the creation of an index determines its specific type, but the available types are not known to functions from the interface. Therefore any creation function from the interface would return something of type HolIndex and the user of the framework would have to specify the type manually, e.g. **let** idx = emptyIndex :: InvIndex. Additionally, there are several possibilities for the creation of indices, for example, one index type is created by inserting elements into an empty index while another type is always constructed from some intermediate data structure. Enforcing a unified creation function for all index types would hamper this flexibility to create indices in different ways.

```
class Binary (d a) => HolDocuments d a where
  sizeDocs          :: d a -> Int

  lookupById        :: Monad m => d a -> DocId -> m (Document a)
  lookupByURI       :: Monad m => d a -> URI -> m DocId

  mergeDocs         :: d a -> d a -> ([(DocId, DocId)], d a)

  insertDoc         :: d a -> (Document a) -> (DocId, d a)

  removeById        :: d a -> DocId -> d a
  removeByURI       :: d a -> URI -> d a

  updateDocuments   :: (Document a -> Document b) -> d a -> d b
```

Listing 4.3: *The interface definition for Holumbus document tables.*

The interface for the document table is less bulky, although its complexity is increased by the additional type parameter for the Document type. Because a document can be customized with additional information of arbitrary type and the HolDocuments interface refers to the Document type, it has to be parameterized with a type variable, too.

HolDocuments represents a mapping between artificial document identifiers and the real documents. A document is unambiguously identified by its URI, which should be unique by definition. Functions providing lookup either by URI or by identifier are defined by the interface, as well as functions for the manipulation of the document table. Again, a merge function for the union of several document tables is defined. Noteworthy about this function is its return value, which consists of the merged table and a list of conflicting document identifiers (identifiers that have been changed during the merge).

Because the document table ist able to sort out conflicting documents with the help of document identification based on URIs, the merge function is required to resolve conflicts and to return a list of identifiers from the document table in the second argument which have been reassigned due to a conflict. In other words, if conflicting documents are encountered, the identifier of the document from the second document table is changed and the old identifier is returned together with the new identifier. This allows for the update of conflicting document identifiers in other data structures, e.g. by using updateDocIds from the HolIndex interface.

Instead of providing a generalized substract function like the HolIndex interface does, only two remove functions are defined by the interface for reasons of simplicity. These offer removal of a document by either its identifier or URI. Functions for the creation of a document table are again omitted intentionally, for the same reasons as with the HolIndex interface. A generalized function for the modification of documents is provided, which applies a custom function to every document. It is a bit less general than the **map** function on lists but still comparable to it.

```
class HolCache c where
  getDocText  :: c -> Context -> DocId -> IO (Maybe Content)
  putDocText  :: c -> Context -> DocId -> Content -> IO ()
```

Listing 4.4: *The interface definition for Holumbus document caches.*

To provide a full text preview when displaying results from a query, the contents of every document from the result set have to be retrieved. To speed up this process and to avoid problems with vanished resources on the Internet, a cache structure can be built during the indexing process. This cache is able to store simplified copies of the original content, resembling the full text of all documents. Because this amount of data tends to grow very large, it will probably be stored on hard disk by implementations of this interface. Hence the two functions provided are using the I/O monad to allow the usage of I/O operations in implementations of the interface.

4.2.3. Core Modules

The basic data types and the interfaces mentioned in the previous section are defined in the module Holumbus.Index.Common. This module and the respective implementations in Holumbus.Index.Inverted, Holumbus.Index.Documents and Holumbus.Index.Cache make up the core data structures of the Holumbus framework.

The module Holumbus.Query.Language.Grammar defines the query language while the parser for the default syntax is located in Holumbus.Query.Language.Parser. Together with the query processor in Holumbus.Query.Processor, the ranking mechanism in Holumbus.Query.Ranking and the result data structure in Holumbus.Query.Result, these modules are the core of the query part of the Holumbus framework.

All of these modules are bundled in the core library. The library itself is used by the utilities described in the next section, which are provided together with the examples for the framework.

4.2.4. Utilities

The main utility provided by the examples is a command line application offering a simple search interface to an index. It is called *SimpleSearch*. The application can handle local indices as well as indices distributed over several machines. The default syntax provided by the framework is supported and can be used to formulate queries, which exposes all possible types of queries to the user.

To measure query performance on a particular system, there is a program called *Benchmark*. It supports both local and distributed queries, too. During startup, about 4,000 queries are generated automatically and executed on a given index and its associated document table. The total execution time is measured and some statistics are printed upon program termination. Only query evaluation itself is measured, time needed for parsing an input string is omitted.

Another important application provided is the *QueryServer*. It is a simple example implementation of the server interface for query distribution. The program just loads a (partial) index file and waits for query requests coming over the network. As mentioned before, the command line search as well as the benchmark program include support for distributed queries and can be used to send requests to a number of query servers. The server does not only support query requests but can also be updated over the network by sending an index and instructions on how to combine the new index with the existing one. Operations available include a full replacement of the existing index, substraction from the existing index and merging with the existing index.

Utility	*Description*
Convert	Converts an index or documents file from XML to binary format and vice versa.
Merge	Merges an index and its documents file with another index and documents file.
Split	Splits an index either by documents, words or context into a number of smaller indices.
Stats	Prints some statistics about an index.
Update	Updates the indices on a number of query servers by merging, substracting or replacing with another index.
Words	Prints a list of all words contained in an index.

Table 4.2.: *The collection of small utilities provided by the Holumbus framework.*

Some more programs are available, most of them being just small helper applications, e.g. for splitting indices into several parts or converting from XML to binary format. These are listed in table 4.2 together with a short description. Currently, all programs operating on a document table assume just a single **Int** as custom information for documents. This restriction is necessary, as the specific type of a document table has to be specified upon loading from a file.

4.3. Basic Functionality

The basic functionality, which is used by almost every other part of the framework, is provided through the HolIndex interface. An implementation of this interface based on the inverted file (see also section 2.3.2) is included in the framework. To meet the requirements for prefix queries and structural queries (see sections 3.2.1 and 3.2.3), this implementation uses the techniques and data structures further explained in this section.

4.3.1. Trie

The Holumbus framework includes a very flexible trie implementation, based on M-ary trees. The main difference to arbitrary M-ary trees is in the two different types of nodes, as shown in listing 4.5. Both can be used anywhere in the tree and are used to tell apart nodes representing the end of a key from intermediate nodes. Every node contains a part of a key, but only nodes containing the last part of a key can store an associated value. Therefore a Seq node always contains at least two successors (see section 2.3.1) and a leaf is represented by an End node containing the empty list.

```
data Trie a = End Key a [Trie a]
            | Seq Key   [Trie a]

type Key = [Word8]
```

Listing 4.5: *The internal data structure of the trie.*

To make the trie implementation more widely usable, keys are internally represented by lists of single bytes. Hence every type of data which can be transformed into a list of bytes can be used as key. The framework includes a module providing a mapping from strings to arbitrary values on top of the trie implementation, which uses UTF-8 encoding for the transformation of strings into lists of bytes.

The interface of the trie module is loosely modeled after the interfaces of the Data.Map and Data.IntMap modules. Traversing the trie is possible using the Functor and Foldable interfaces. Some additional functions are provided, mainly to support prefix search for keys. These are also available in generalized versions which take an additional function argument to preprocess keys before comparison. This mechanism is used by the string map to provide case-insensitive search functions.

Initially, there was an attempt to employ compression for the keys in the trie, by using the compression scheme explained in the next section. Unfortunately, there was a major impact on the performance of several functions of the trie. Some statistics about a typical trie when used as a dictionary for an inverted index have shown that more than the half of all nodes just contain one single character as a key. As a single character can not be compressed anymore, the effect of any compression scheme is probably negligible. Additionally, the efficiency of the particular compression scheme used is further reduced as the order of characters may not be changed (see next section). Thus the compression was removed from the trie module.

4.3.2. Compression

In general, the occurrences of an index are contributing most to its total space requirements. Therefore almost any compression scheme employed by an index structure tries to compact the occurrences. For performance reasons, the HolIndex interface requires occurrences to be represented as IntMap of IntSet, which is very space consuming. To overcome these limitations, an alternative representation of occurrences is provided by the module Holumbus.Index.Compression. A new type CompressedOccurrences is defined together with appropriate encoding/decoding functions, which can be freely used by any index implementation.

Compression ratios for three small indices are listed in table 4.3. These numbers demonstrate that compression is more efficient for larger indices. This is caused by the initial overhead for every unique word in the dictionary. If a word already exists in the dictionary, its position is just added to the occurrences and will be compressed. The probability for a word being already present in the dictionary grows as the index size grows. Therefore the ratio of unique words to the total number of words heavily influences compression efficiency.

Index	*Documents*	*Total words*	*Uncompressed*	*Compressed*	*Ratio*
java-index	86	19,707	66 MB	61 MB	7,6%
sd-index	100	23,421	71 MB	63 MB	11,2%
vl-index	449	96,249	117 MB	93 MB	20,5%

Table 4.3.: *Memory footprint and compression ratio of three small example indices.*

The compressed occurrences are still based on the IntMap structure, only the IntSet storing word positions is replaced by a more space efficient alternative. This is derived from lists of differences which are encoded by the Simple-9 compression scheme, as explained in section 2.3.3. Using this scheme, the compression of a set of integers involves the following steps:

1. Transforming the IntSet into a list of sorted integers.
2. Replacing each integer by the difference to its predecessor.
3. Encoding of this list of differences into a list of words using the Simple-9 scheme.

The inverse operation of decompressing the set of integers thus exhibits the steps:

1. Decoding the list of words into a list of differences using the Simple-9 scheme.
2. Replacing each integer by the sum to its predecessor.
3. Transforming the list of sorted integers into an IntSet.

The transformation into and from a sorted list of integers is already included in the Data.IntSet module, the calculation of differences and sums is done in the Holumbus.Data.DiffList module. The encoding and decoding functions for the Simple-9 compression scheme are available through the Holumbus.Data.Crunch module. These modules are completely exposed by the library in order to allow their use for customized index implementations (or anything else requiring space efficient storage of integers).

For the use of the Simple-9 compression scheme in the Holumbus library, it has been extended to work with 64-bit words, because development took place under a 64-bit system. Additionally, 64-bit architectures become increasingly popular among ordinary computers and therefore 64-bit wide words will be the forthcoming standard. The possible combinations of word lengths and number of values are listed in table 4.4.

Selector	*Number of codes*	*Length of each code (bits)*	*Number of unused bits*
a	60	1	0
b	30	2	0
c	20	3	0
d	15	4	0
e	12	5	0
f	10	6	0
g	8	7	4
h	7	8	4
i	6	10	0
j	5	12	0
k	4	15	0
l	3	20	0
m	2	30	0
n	1	60	0

Table 4.4.: *Fourteen different ways of using 60 bits for flat binary codes (Simple-9 extended for 64-bit words).*

4.3.3. Inverted File

The implementation of the HolIndex interface using an inverted file structure is based upon the trie and the compressed occurrences described in the previous sections. This leaves a quite simple structure consisting mainly of a mapping from strings to compressed occurrences, as shown in listing 4.6.

To meet the requirements of structured queries, the index data structure in fact consists of several independent indices. These are stored using a mapping from context identifiers to actual indices. To answer a query, the index for the requested context is retrieved and the search functions for the trie are used to retrieve the occurrences.

Most of the functions for the manipulation of the index are based on the mergeIndexes and substractIndexes functions required by the interface. These are implemented using

the union and difference functions provided by Data.Map, Data.IntMap, Data.IntSet and Holumbus.Data.StrMap (which is in fact based on Holumbus.Data.Trie), respectively.

```
newtype Inverted = Inverted
  { indexParts :: Parts }

type Parts       = Map Context Part
type Part        = StrMap CompressedOccurrences
```

Listing 4.6: *The internal data structure of the index implementation based on the inverted file.*

The implementations of the split functions defined by the interface are more tricky. They are derived from a general allocation scheme based on so-called "buckets". For every split requested, an empty bucket is generated by creating an empty index. For example, if the index has to be split into three parts, three empty buckets are created. Afterwards, depending on the type of the split function, occurrences are extracted from the original index and inserted into the least filled bucket. This process continues until all occurrences are allocated to one of the buckets.

The allocation scheme is implemented by a generalized allocation function, usable by all split functions. The implementation is shown in listing 4.7. The different split functions only need to extract the occurrences from the original index in a specific manner. For example, the splitByWords function has to extract the occurrences on a per-word basis.

The allocate function has three arguments, a custom function for the combination of two values, a list of all available values and a list containing the buckets. For performance reasons, these two lists are already annotated with the size of the values to avoid recalculations. The function returns a list of merged values, with the number of elements being equal to the number of initial buckets.

```
allocate :: (a -> a -> a) -> [(Int, a)] -> [(Int, a)] -> [a]
allocate _ _ [] = []
allocate _ [] ys = map snd ys
allocate f (x:xs) (y:ys) =
  allocate f xs (sortBy (compare `on` fst) ((combine x y):ys))
  where
    combine (s1, v1) (s2, v2) = (s1 + s2, f v1 v2)
```

Listing 4.7: *The generalized allocate function, using a customizable function for the combination of arbitrary values.*

Persistent storage of the inverted index is possible in either XML format or binary format. To provide this functionality, the XmlPickler interface and the Binary interface are both implemented (the latter is already required by the HolIndex interface), as mentioned in section 4.1.2. A generalized load function with automatic detection of the persistent format for data structures implementing both interfaces is provided by the module Holumbus.Index.Common.

4.3.4. Document Structures

The default implementation of the HolDocuments interface is based on two map structures (Data.Map and Data.IntMap in particular), as shown in listing 4.8. One of them is mapping the artificial document identifiers to the actual documents, the other one maps document URIs back to document identifiers. Although the second map only contains redundant information, it is needed to provide fast lookup of documents by their URI's (mostly needed by the indexer and for collision detection when merging indices and their respective document tables).

As required by the interface, the implementation automatically assigns unique identifiers to newly inserted documents. This is done by always keeping the last number used as identifier together with the two maps. The implementation provides a function for the creation of the whole document table structure from just the first of the two maps, which makes persistent storage more efficient. Due to the customizability of a single document with custom information, the document table has to employ a type variable, too.

```
Documents a = Documents
  { idToDoc    :: IntMap (Document a)
  , docToId    :: Map URI DocId
  , lastDocId  :: DocId
  }
```

Listing 4.8: *The internal structure of the document table implementation.*

The HolCache interface is implemented using a simple file-based database. The HDBC package provides a uniform and independent interface for database systems. It is used to access a SQLite3 database, which contains just one table with three columns. These store a document identifier, a context name and the associated text, which enables retrieval of document contents with respect to a specific context. The cache data type itself just contains the database connection and is created from a filename containing the path to the database file. Currently, the HolCache interface does not

require any merge or update support for cache structures, but these will probably be added in future versions of the framework.

4.4. Query Processor

The main component for query evaluation of the Holumbus framework is the query processor. It is responsible for parsing the input string, interpreting and evaluating the resulting query structure and generating a result set.

4.4.1. Language

The query language of the query processor supports several kinds of primitive queries and a number of operators to construct more complex queries. The definition of the grammar is shown in listing 4.9. It provides constructors for case-sensitive/case-insensitive words and phrases as well as fuzzy words, whereas a single word query is always interpreted as prefix.

A constructor for an operator restricting a query to a number of contexts is supported as well as constructors for the usual set operators like complement, intersection, union and difference (called Negation, And, Or and But). The difference operator is mainly used to optimize queries containing combinations of the intersection and complement operators, because the complement operation could involve the retrieval of all documents in the index and is therefore potentially expensive.

The framework itself does not specify any operator precedence to allow a maximum of flexibility for custom parser implementations. Operator priorities are thus defined by the specific syntax used and implemented by the parser respectively.

```
data Query = Word       String
           | Phrase     String
           | CaseWord   String
           | CasePhrase String
           | FuzzyWord  String
           | Specifier  [Context] Query
           | Negation   Query
           | BinQuery   BinOp Query Query

data BinOp = And | Or | But
```

Listing 4.9: *The grammar of the Holumbus query language.*

Although most users of the framework will have to implement custom parsers for the query language, the framework provides a parser implementing a default syntax (see appendix B) for query formulation. This parser is implemented using the monadic

parser combinators from the Parsec library mentioned in section 4.1.2. The syntax exposes all operators except the *But* operator and allows to create almost infinite complex queries using parentheses. An example of a resulting query structure is shown in figure 4.2.

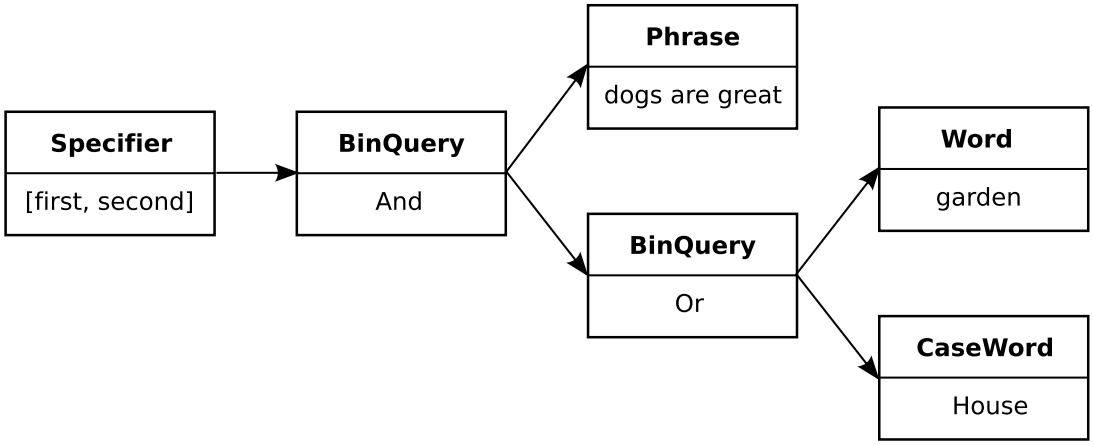


Figure 4.2.: *An example of a query structure resulting from the input string first,second:("dogs are great" AND (garden OR !House)) when using the default syntax.*

The module providing the grammar of the query language also contains two functions for processing a query structure. The optimize function implements some very primitive optimizations through algebraic transformation. The first optimization exploited by the function is the already mentioned replacement of the complement operator by the difference operator when used in conjunction with an intersection. For example, $q1\ And\ (Not\ q2)$ can be transformed to $q1\ But\ q2$ with $q1, q2$ being arbitrary queries.

The second optimization is the removal of unnecessary query terms containing shared prefixes. For example, it is not necessary to evaluate the term $Word\ foo$ in the query $(Word\ foo)\ And\ (Word\ foobar)$, because all documents containing a word beginning with "foobar" are already known to contain a word beginning with "foo". Therefore the term with the shorter prefix does not influence the result set and can be omitted. The same applies for the union operation vice-versa, where the longer prefix can be omitted.

The second function, checkWith, can be used to check the string arguments of the primitive query terms with a custom predicate. This is especially useful to guarantee a minimum amount of characters for the prefixes queries to avoid very large result sets. The function lazily traverses the whole query expression and evaluates the predicate for every primitive found.

4.4.2. Result Types

The query processor employs three different structures for the representation of query results. Two of these are only used internally and one represents the final result returned to the calling function. The first type, RawResult (see section 4.2.2), is just a plain list containing tuples of a word and its occurrences. These lists are returned by the querying functions of the HolIndex interface. They have to be transformed into a structure based on documents, because otherwise the query processor would not be able to perform set operations efficiently as all operators of the query language are defined in terms of documents. This second type of result, called Intermediate, is shown in listing 4.10. It is only used internally during processing of the operators and for distributed query evaluation (by serializing the intermediate result using an instance of Binary).

```
type Intermediate = IntMap IntermediateContexts
type IntermediateContexts = Map Context IntermediateWords
type IntermediateWords = Map Word (WordInfo, Positions)
```

Listing 4.10: *The intermediate result structure used for query evaluation.*

The structure is organized in a tree-like fashion, with each document identifier being at the root of a tree containing the related contexts, words and positions. Basically, there is a set of contexts for each document found. Every context contains completions for prefixes from the query. Then again for every such completion there is a set of positions and a WordInfo structure (included in listing 4.11). This structure contains a list of original prefixes from the query string which led to the inclusion of the actual word in the result. The main advantage of the intermediate result type is the usage of an IntMap for the representation of documents at the topmost level because its implementation provides very fast set operations.

After full evaluation of the query, the intermediate structure is transformed into the final result type shown in listing 4.11. A result consists of two main parts (illustrated in figure 4.3), the documents matching the query and the possible completions of prefixes from the query which are occurring somewhere in these documents. The part containing the documents is almost equivalent to the intermediate result, it is just extended by a DocInfo structure containing information about the actual document.

The other part representing the words is a similar tree-like structure, with every word being at the root of a tree containing the related contexts, documents and positions. Basically, there is a set of contexts and a WordInfo structure for each word. Every context contains documents in which the word occurs. Then again for every document there is a set of positions of the word in this document.

Instances of XmlPickler and Binary are provided for the final result type to allow easy serialization and persistent storage. The additional information structures for words and documents also contain score values. These are used by a ranking function which can be applied to determine the quality of search results (see section 4.4.6).

```
data Result a = Result
  { docHits  :: (DocHits a)
  , wordHits :: WordHits
  }

data DocInfo a = DocInfo
  { document :: (Document a)
  , docScore :: Score
  }

data WordInfo = WordInfo
  { terms     :: Terms
  , wordScore :: Score
  }

type DocHits a = IntMap (DocInfo a, DocContextHits)
type DocContextHits = Map Context DocWordHits
type DocWordHits = Map Word Positions

type WordHits = Map Word (WordInfo, WordContextHits)
type WordContextHits = Map Context WordDocHits
type WordDocHits = Occurrences

type Score = Float
type Terms = [String]
```

Listing 4.11: *The structure of a final result returned by the query processor.*

Originally, there was no intermediate result structure and the final result was used during query processing. Some major drawbacks resulting from its rather complex structure have caused the introduction of the intermediate result type. First of all, the set operations were performed independently on both the documents and the words. This may result in completions to be included in the words part which do not appear in any of the documents from the documents part. Therefore the possible completions actually leading to a document had to be extracted out of the documents part after query evaluation was finished, which rendered the words part obsolete. Additionally, there have been severe performance implications due to the need to create the words part from the RawResult. Quite some time was spent optimizing the transformation process for the words part, but just omitting it resulted in the best performance improvement.

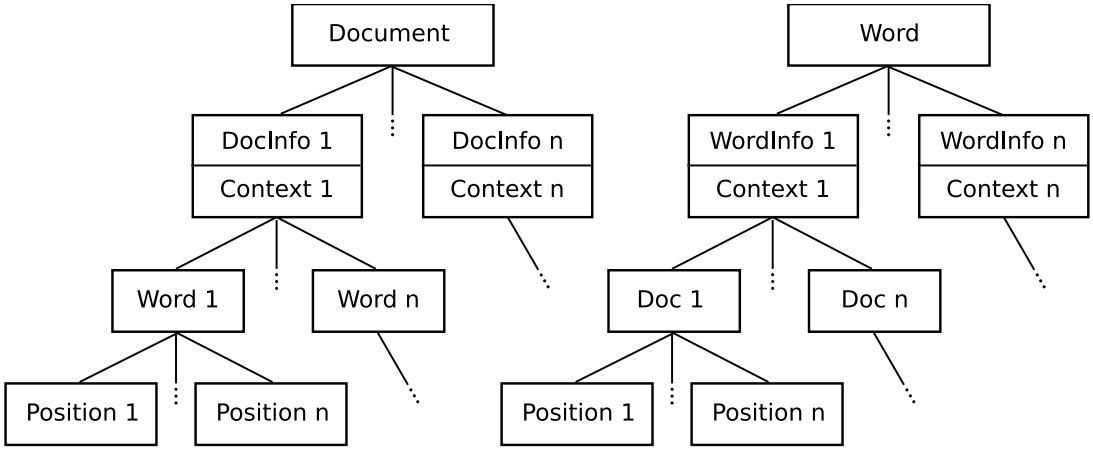


Figure 4.3.: *The structure of a document and a word in a final result.*

4.4.3. Query Evaluation

In general, every query expression is evaluated with respect to a number of contexts. This defaults to a list of all contexts when the evaluation starts. This list is then further restricted or extended by the Specifier operator. The evaluation of a query starts with a recursive descend of the structure representing the query expression and calling appropriate functions for processing the different language elements. A processing state containing the index, the current list of contexts and some configuration values is weaved through all of these functions.

At first, the language primitives are evaluated by retrieving words and occurrences as RawResult from the index using the querying functions provided by the HolIndex interface. To enable further processing, the result is transformed into the Intermediate structure (see figure 4.4). This is done for every context in the list of current contexts. The intermediate results obtained are merged using the union operation. Primitives for fuzzy and phrase queries are evaluated differently, as explained in sections 4.4.4 and 4.4.5.

Binary query expressions are evaluated by just executing the appropriate set operation on the intermediate result structures returned by evaluation of the two sub-expressions. A negation operation is evaluated by retrieving all documents for the current contexts and substracting the intermediate result returned by evaluation of the negated expression. A query expression starting with a specifier operation is evaluated by replacing the list of current contexts in the processing state by the list given by the specifier operator. Afterwards the sub-expression is evaluated using the modified processing state.

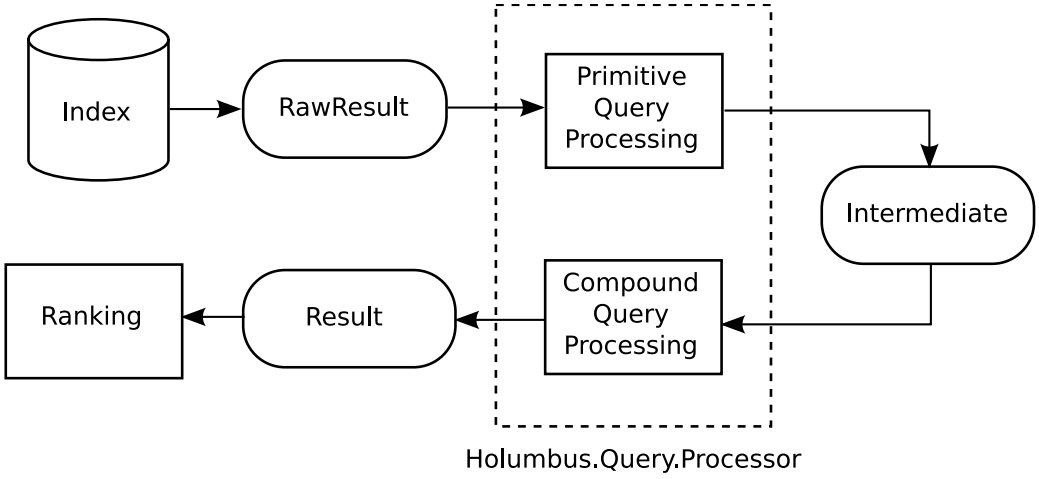


Figure 4.4.: *The different result types used by the query processor.*

Because the transformation of a RawResult into an Intermediate result is potentially expensive if a large number of completions is retrieved for a prefix from the query, the RawResult list can be limited to a fixed amount of completions. This configuration option is contained in the processing state to provide access for every processing function. Only the best completions for a prefix (and their respective occurrences) should be used for further query processing. Therefore some heuristic for estimating the quality of a completion has to be used. This should not be confused with a ranking function evaluating the quality of the retrieved documents. The heuristic used is the so-called *inverse document frequency*. It is defined as follows [BYRN99]:

> *Let N be the total number of documents in the system and n_i be the number of documents in which the index term k_i appears. Then, let idf_i, inverse document frequency for k_i, be given by*
>
> $$idf_i = \log \frac{N}{n_i}$$

As the heuristic has to be calculated for every completion, it needs to be reasonable fast. This requirement is met by the inverse document frequency, because the total number of documents is included in the processing state and therefore just requires a simple lookup which is done in constant time. The number of documents is retrieved by calculating the size of the map containing the document identifiers. The time required for this is linear with the number of elements for the current implementation of IntMap. Hence the total time required for calculating the inverse document frequency is just linear with the number of documents containing the word.

4.4.4. Phrase Queries

Phrase queries are not natively supported by the HolIndex interface. Instead, phrases are decomposed into a number of single word queries. As the occurrences include word positions, the query processor is able to identify juxtaposed words. At first, the phrase is split into a list of single words using whitespace characters as delimiters. Afterwards, a single word query is issued using lookupCase or lookupNoCase and the first word from the list to retrieve an initial result.

The algorithm continues by retrieving another result for the second word in the list and the initial result is compared to the current result. Documents which do not appear in the current result or have no position in their occurrences, which is incremented by one compared to the respective position from the previous result, are removed from the initial result. This process continues until either no documents are left or every word in the list has been processed. Because the comparison of positions only requires the plain occurrences structure for a word, the whole algorithm can be implemented using the RawResult type. This avoids the computational expensive transformation operation for every single query, as it is only needed once for the final result when the algorithm has finished.

If the initial result is not empty after the last word has been processed, it contains the documents matching the query. The exact sequence of words as stated by the phrase query occurs at least once in one of the contexts. Currently, the Holumbus framework does not provide suggestions for phrase queries, because these would require very different index data structures and/or a lot of processing power. At the same time, most users issuing a phrase query want to search for that exact sequence of words and do not require any hints on how to formulate their query.

4.4.5. Fuzzy Queries

Fuzzy queries are also evaluated differently from other language primitives. As the HolIndex interface itself does not provide any query functions supporting fuzzy queries, these have to be evaluated entirely by the query processor using the existing query functions. The basic strategy for the evaluation of fuzzy queries is to execute a normal word query for the current query term and to continuously apply more fuzziness to the term as long as the query does not yield any results.

This strategy is implemented by generating a list of fuzzy terms using the original term as starting point. These are rated by a “fuzzy factor” representing the fuzziness with respect to the original term. The list containing the generated fuzzy terms is sorted accordingly. The algorithm now starts by executing a usual word query using

the original term. If this query yields any results, the algorithm terminates. If the result set is empty, the algorithm starts over using the first fuzzy term from the list. This process continues until something has been found or every fuzzy term has been tried. As the fuzziness of terms increases, possible results become more fuzzy, too.

The main difficulty of this algorithm lies in the generation of reasonable fuzzy terms together with a sensible rating of fuzziness. The Holumbus framework employs its own technique for generating sets of fuzzy terms, based upon replacement rules and permutation of characters.

A set of replacement rules can be provided by the user of the framework as list of tuples, while permutation of characters can only be enabled or disabled. These configuration options are part of the processing state and therefore are available to the function evaluating fuzzy queries. As the importance of a character for the meaning of a term decreases as its position in the string increases, the fuzzy ratings are always based on character positions for both replacements and permutations of characters.

Each rule contains a tuple of two strings and a weight value used for calculating the fuzziness when applying the rule. The values used for weights and fuzziness ratings are floating-point numbers, with 1.0 being defined as the fuzziness when just the first character of the original term is altered. The two strings define a replacement, where one string is searched and replaced by the other, if it was found. Every replacement is applied in both directions. The weights of a set of replacements are automatically scaled to the interval 0.0 to 1.0 and are only regarded in relation to the weights of other rules in the same set. Therefore a single rule is always scaled to 1.0, independent of its actual weight defined by the user. The fuzziness resulting from the application of a single replacement rule is calculated as follows:

$$f_{rt} = w_r * \frac{l_t - p_{rt}}{l_t}$$

With f_{rt} being the fuzziness resulting from successful application of replacement r to term t, w_r being the weight of the replacement rule r, l_t being the length of the term t and p_{rt} being the position where r was applied to t (counting from zero). Hence, the maximum fuzziness resulting from the application of just one replacement rule is 1.0. For example, the rule *(“ck”, “g”)* with an effective weight of 0.5 applied to the term *“pick”* results in the term *“pig”* and a fuzzy rating of 0.25.

Calculating the fuzziness resulting from permutation of characters is less complex. The algorithm only swaps adjacent characters and calculates the fuzzy rating based on the position of the swapped pair. The fuzziness resulting from swapping a single pair is calculated the same way as for a replacement rule, just omitting the replacement-

specific weight. For example, three different fuzzy terms result from permuting the characters in *"pick"*: *"ipck"* with a fuzzy rating of 1.0, *"pcik"* with 0.75 and *"pikc"* with 0.5.

To be able to generate almost infinite amounts of fuzzy terms, both algorithms are combined and applied recursively to already generated fuzzy terms. The total fuzziness resulting from several applications of replacement rules or permutations is calculated by adding the individual fuzzy ratings. To avoid infinite recursion, the configuration for fuzzy set generation employs an upper limit of fuzziness, specified by the user of the framework together with a set of replacement rules. The module Holumbus.Query.Fuzzy provides predefined sets of replacement rules suitable for English and German natural language.

This method for generating fuzzy terms can be seen as refinement of the general use of a thesaurus, mentioned in section 2.1.2. Instead of a generalized membership function the fuzzy rating is used to determine the similarity between the original term and its fuzzy variations.

4.4.6. Ranking

The Holumbus framework provides a highly customizable ranking mechanism, partly shown in listing 4.12. Besides ranking of retrieved documents, a ranking of completions for prefixes from the query is also possible. When using the framework, customized ranking functions can be passed to the ranking mechanism for both documents and words. The application of the ranking mechanism is not mandatory, although it is fairly indispensable when the underlying document collection is quite big and therefore large result sets are retrieved from the index (see section 2.2.3).

```
data RankConfig = RankConfig
  { docRanking  :: DocRanking
  , wordRanking :: WordRanking
  }

type DocRanking = DocId -> DocContextHits -> Score
type WordRanking = Word -> WordContextHits -> Score
```

Listing 4.12: *The configuration of the ranking mechanism.*

The framework includes some predefined ranking functions which can be used instead of custom implementations. These are using very simple ranking schemes, all based on counting the occurrences. For ranking of documents, the simplest scheme just counts the number of matching words occurring in the document. The same is done

for ranking of words, where the number of documents from the result set containing the word is counted.

The second scheme is also based on counting occurrences, but counts are weighted by their original context. These ranking functions take lists containing weights for different contexts and calculate the score accordingly. The scores are scaled to the interval 0.0 to 1.0 automatically, as with the weights of replacement rules for fuzzy queries. Both schemes are implemented using a generalized ranking function which takes a configuration containing the custom ranking functions and applies these to all elements in the result set.

As mentioned before, ranking of documents is not emphasized in this work. Therefore only quite simple predefined ranking functions are provided by the framework. It is nevertheless possible for users of the framework to implement arbitrary ranking schemes, due to the highly customizable interface of the ranking mechanism.

4.5. Distribution

The Holumbus framework supports distributed queries to provide scalability for large indices. As index data could easily exceed available system memory, distribution of a query over several machines allows to create a large virtual index from a number of small indices running on these machines.

4.5.1. Distributed Queries

The distributed evaluation of queries is mainly based on the split functions required by the HolIndex interfaces. Right now, only indices splitted by documents are supported by the mechanism for distributed query evaluation (see section 4.5.3). The framework defines a simple protocol for the communication between clients and query servers and provides implementations for both sides. The general architecture of a distributed system using the Holumbus framework is illustrated in figure 4.5.

Every query is sent to a number of query servers. Each of these is going to process the query independently using its local index. The results are sent back to the client, where they are merged into one final result. This requires the original index to be split into several parts, one for each query server. Hence, the distributed evaluation of a query is based on the following steps:

1. Sending the query to the query servers.

2. Independent evaluation of the query on the query servers.

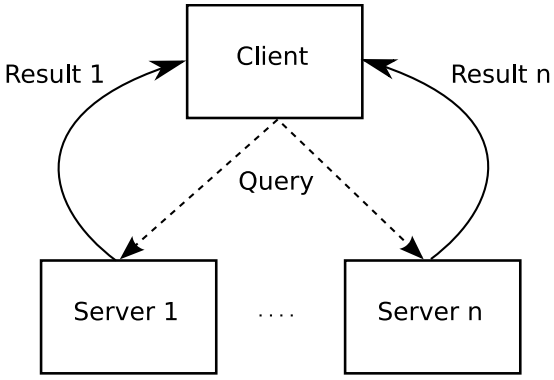


Figure 4.5.: *The general architecture of a distributed search engine using the Holumbus framework.*

3. Transmitting the individual results back to the client.

4. Merging the partial results into one final result.

The functionality for merging several results is already available, because it is nothing else than the union operation used to implement the Or operator. Therefore, the implementation of distributed query evaluation is quite straightforward. The query processor includes an additional function, which returns an `Intermediate` result instead of the final result structure. These are merged using the union function and transformed into the final result afterwards.

The missing part is only about communication between client and server. Originally, the idea was to use HTTP and binary serialization via ByteStrings for client server communication. The advantage in this approach is the general availability of HTTP libraries and servers, which could be used as a starting point for individual client and server implementations. In practice, it turned out that even the simple HTTP seemed to be way too complex for this task. In consequence, a custom protocol was created, using simple one-line plain text headers and ByteStrings for binary data (see section 4.5.2). In addition to distributed query evaluation, the protocol supports commands for remote updates of a query server's index by sending index data, which is combined with the existing index. Avaliable combination options include merging the new index with the existing index, substracting the new index from the existing index and replacing the existing index by the new index. The general flow of communication between client and server is illustrated in figure 4.6.

On the client side, a function `processDistributed` is provided for query processing. The function takes some configuration options (shown in listing 4.13), the document table and the query itself as arguments. The configuration contains the names of

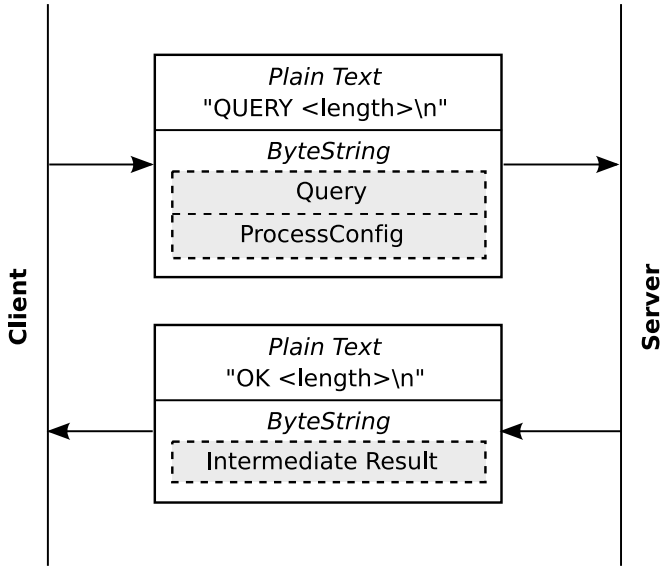


Figure 4.6.: *The flow of communication between client and server when processing a distributed query.*

the query servers, an option whether results from the query servers are compressed before transmission and the configuration for the query processor. The latter is sent to the query servers together with the query and the compression option. The compression of result data is optional, because it only pays off if large result sets are to be expected. For small results, the compression just adds computational overhead on both the client side and the server side and should be turned off (as shown by the performance figures in appendix C). Index data for the update commands is always compressed before transmission because it is expected to be almost always quite large. The update commands are available through three functions, updateAdd, updateRemove and updateReplace.

```
data DistributedConfig = DistributedConfig
  { queryServers    :: [Server]
  , compressResult  :: Bool
  , processConfig   :: ProcessConfig
  }

type Server = String
```

Listing 4.13: *The configuration structure for distributed query evaluation.*

When processDistributed is called, a global MVar is allocated and initialized with an empty Intermediate result. This allows synchronized access to the result structure for

several concurrent threads. Afterwards, a lightweight Haskell thread is spawned for every server specified by the configuration structure using forkIO. Every thread opens a socket, encodes the query, the process configuration and the compression option into a ByteString and creates the message header. The header is written to the socket, followed by the encoded data. Now the thread listens on the socket for a response from the server.

As soon as the response arrives, the header is read and the appropriate number of bytes is read from the socket into a ByteString. After decoding the binary data, the thread merges the partial result with the global result and terminates. If compression of result data was requested, it is decompressed before decoding. This is done using the popular *bzip2* algorithm. When all threads have finished their work, the global MVar contains the complete result set. It is transformed into the final result structure and returned to the calling function.

The same scheme applies for the update functions, which carry out one of the modification commands available. The only difference is the absence of any result data. Instead, the server just transmits a single header indicating success or failure of the operation.

For the server side, a function called listenForRequests is provided. It takes the index, a port number and a custom function used for logging as arguments. The function also allocates a global MVar which is initialized using the index. Afterwards, the server starts listening on the provided port number and waits for requests. As soon as a client connects, a lightweight Haskell thread is spawned to process the request. At first, the type of the command is extracted from the header. Afterwards, the binary data is read according to the command and the length specified in the header. The local index is read from the global variable and used to carry out the operation requested by either processing a query using the provided data or modifying the index. The result of the operation is reported back to the client, possibly including the result of the query.

The framework includes an example for a query server based on the listenForRequests function as well as a small program enabling the remote update of query servers, based on the update functions mentioned before.

4.5.2. Protocol

The communication between a client and a query server is based on messages. Each message contains a header and the actual data. While the header is transmitted as plain text, the data is encoded as ByteString. A message sent by the client to the server is always some kind of request. The server just sends a single response for every

request received. While requests from a client always contain some data, it is possible for the server to send a single header without any data as response. Headers sent by the client contain a command and the length of the data sent together with the request. Headers from the server contain a status code, the length of the data (if any) and, in case of an error, an appropriate message. The header fields are separated by whitespace and the end of the header is indicated by a newline character. Therefore a header always consist of just one single line of plain text. The available commands are described in table 4.5.

Command	*Description*
`QUERY`	The client sends a query and the server answers with the partial result.
`ADD`	The client sends an index which should be merged with the server's index.
`REMOVE`	The client sends an index which should be substracted from the server's index.
`REPLACE`	The client sends an index which should replace the server's index.

Table 4.5.: *The available commands for client requests sent to a query server.*

The response from a query server contains either `OK` or `FAIL` in the first field of the header. In case of failure, an error message is required to be sent in the second field of the header. The error message may contain spaces, as nothing else has to be transmitted in the header. If the server has processed a request successfully, this is acknowledged by an appropriate response. In case of a `QUERY` request, the length of the result data is transmitted in the second field of the header and the header is followed by the binary data.

Through the transmission of the length of the binary data, the opposite side is able to safely read the required amount of incoming data. The Data.ByteString module provides support for reading and writing ByteStrings directly from/to handles (sockets), as does the System.IO module for plaintext. These functions are used for the transmission of binary and header data.

4.5.3. Combining Results

The most difficult part on the client side is, despite network communication, the correct merging of partial results. This is the main reason why the current implementation only supports indices splitted by documents. While merging the results is very simple for indices splitted by documents, it is much more complex for the other types.

The partial indices resulting from splitting an index by documents can be regarded as completely independent indices, because every document and the related occurrences are guaranteed to be contained in only one part. This allows for evaluation of a query on a partial index in the very same way as it is done on a single index and just summing up the results. This is different for indices splitted by words or contexts, where either the different contexts or the words originating from one document are spread over several partial indices.

The easiest way to overcome this problem for an index splitted by words or contexts would be to drop the distributed query evaluation. Instead, the query is evaluated by the client and only RawResult structures are retrieved from the query servers. Of course, there is a major drawback: Any possible performance gains from the parallel evaluation of the query are lost. If parallel evaluation of the query is employed, merging the partial results becomes quite complex.

For example, a large index is splitted by words into two small indices and the query $foo\ AND\ NOT\ bar$ is evaluated independently for every partial index. If the words foo and bar happen to be located in different indices, merging the partial results is almost impossible without retrieving additional information from the query servers. The result returned from the index containing foo consists of all documents containing foo, because obviously bar is located in the other index and therefore documents containing foo and bar were not removed from the result.

At the same time, the result from the index containing bar is empty, because foo is not known to this index and removing something from nothing is still nothing. When merging the results, how should the client know that there are still documents in the first result, which actually contain bar? The same problems arise when splitting by context, because queries can be arbitrarily limited by contexts using the Specifier operator.

In general, splitting an index by documents seems to be the best solution for distributed query evaluation, although splitting by words promises to result in some performance advantages (see section 2.3.4). Despite being able to evaluate queries in parallel, the main advantage of splitting by documents is the easy implementation by using functionality which is already available because it is required for normal query evaluation. Some figures on distributed query performance are listed in appendix C.

4.6. Hayoo!

To evaluate the flexibility and customizability of the Holumbus framework, it has been used to implement a very special kind of search engine. A Haskell API search

engine was developed based on the model of the existing Haskell API search engine *Hoogle* [Mit]. The resulting search engine is called *Hayoo!* and is equipped with a web interface which can be reached at `http://www.holumbus.org/hayoo`.

4.6.1. Index Layout

The search engine generally searches for Haskell functions. Instead of just searching for function names, searching for the signature of a function is supported, too. Additionally, searching for module names and function descriptions is possible. In every case, the search engine returns a list of functions matching the query.

To support these different types of information about a function, the underlying index contains several different contexts, shown in table 4.6. These store basic informations, like function name, module name and description. Because the Holumbus framework always performs prefix queries, single words from a function name will not be found if preceded by another word. For example, searching for "time" won't return the function "getTime", as the user might actually expect. To overcome this limitation, function names are splitted at uppercase letters during index creation and the resulting words are added to an additional context. The same problem applies to hierarchical module names, hence these are splitted at separating dots and the single words are inserted into an additional context.

Context	*Contents*
name	Full function names, e.g. `getTime`
partial	Splitted function names, e.g. `get` and `Time`
module	Full module names, e.g. `Data.Map`
hierarchy	Splitted module names, e.g. `Data` and `Map`
description	All words from function descriptions.
signature	Function signature with whitespace removed, e.g. `Int->Bool`
normalized	Generalized function signatures, e.g. `a->b`

Table 4.6.: *Contexts available in the index of the Hayoo! search engine.*

Searching for signatures is even more complicated. These can be searched using either explicit type names or type variables and users could enter the same signatures differently, for example including or excluding whitespace around `->` or using arbitrary type variable names. Searching for `a->b->c` should yield the same results as searching for `d->e->f`. Therefore function signatures are stored in two different contexts and will be preprocessed during index creation as well as during query evaluation. One context stores function signatures using the original type names while the other context uses generalized signatures with type variables. Whitespace around `->` is stripped for both

types of signatures. The same algorithm for replacing explicit type names by type variables is used when searching for signatures, to make `a->b->c` literally the same as `d->e->f`. Unfortunately, this scheme limits the combined use of explicit types and type variables in the same signature. Those signatures will only be found if the same name as in the original signature is used for type variables.

To provide extensive information about a function when displaying results, the document table is used to store a lot of custom information. The document structure contains the function name as title and its URI. The custom function info structure (shown in listing 4.14) contains the full module name, the original signature and an URI pointing to the source code of the function (if available).

```
data FunctionInfo = FunctionInfo
  { moduleName :: String
  , signature :: String
  , sourceURI :: Maybe String
  }
```

Listing 4.14: *Custom document information stored in the document table.*

To provide a preview of function descriptions, these are stored using the full text cache of the Holumbus framework. It only stores the full document contents for the description context, the other contexts are omitted during index creation. In total, this vast amount of information is used to create a highly interactive web-based search interface, which is described in the next section.

4.6.2. Search Function

The web interface of Hayoo! is organized like almost every other web-based search engine (see figure 4.7). The upper part contains a single text box, where users can enter search terms. Even though Hayoo! starts searching as soon as the user starts typing, the text box is still accompanied by a search button. Frequent users of search engines might expect this and would be irritated otherwise, because they are used to it. Clicking the button simply executes the current query, just as hitting the return key in the text box does.

The part below the text box, separated by a status message indicating the number of results or possibly showing error messages, contains suggestions matching the query terms and a list of results. The suggestions are displayed as a so-called *tag cloud*, usually used for displaying a number of tags associated with blog articles, pictures or anything else. The main advantage of this display format is the indication of some kind of score value while retaining the original alphabetical order. Specifically,

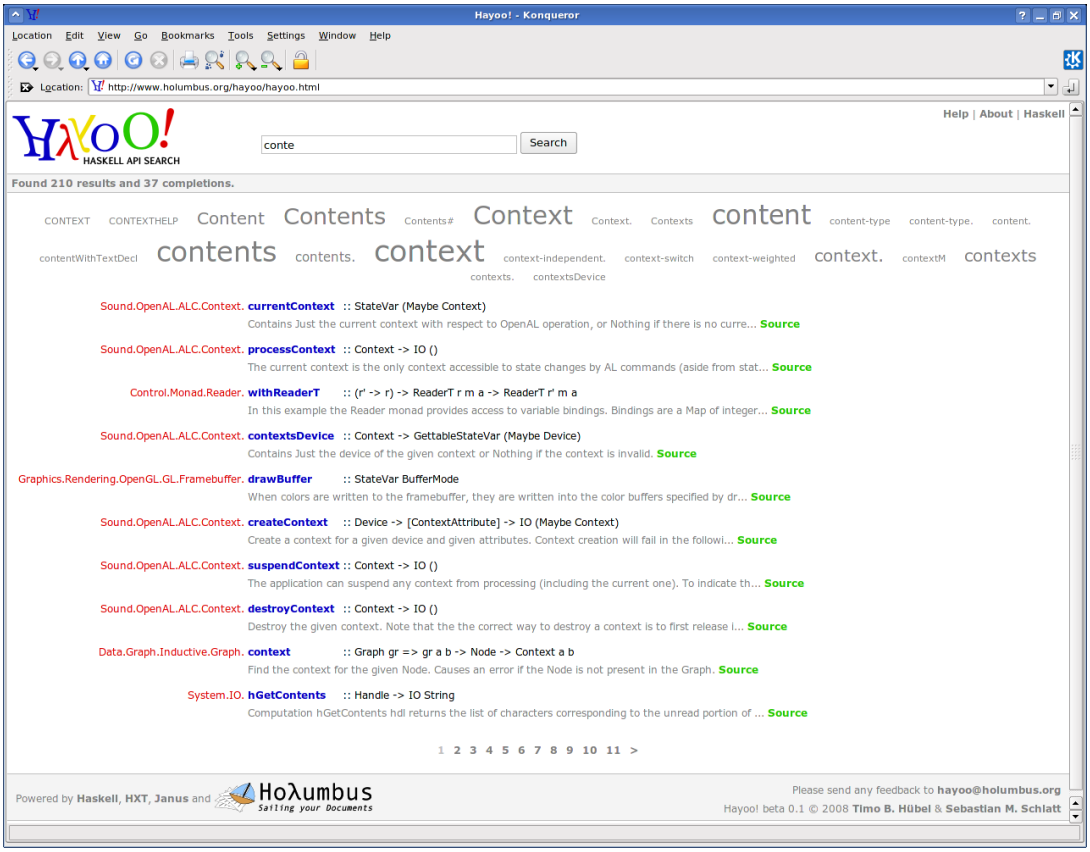


Figure 4.7.: *The web interface of the Hayoo! Haskell API search engine.*

possible completions for query terms are scaled in size according to their rating while the original order is preserved. This allows users to find a specific word quickly because of the familiar ordering while catching the best suggestions at a glimpse because of their size.

Results are displayed as a list of functions, including the respective module name, function name, signature and description. The module name links to the Haddock documentation of the corresponding module and the function name links to the documentation of the function directly. Additionally, a source link is provided, which links to the source code (if available online). A paging mechanism is employed to limit the display to ten results at a time. This requires the user to actively change the current page to view another ten results.

The Hayoo! web interface makes heavy use of AJAX technology to provide search results and suggestions while the user is still typing a query. The JavaScript `onkeyup` event is used to get hold of every keystroke. After a letter is typed, the script waits for 300 milliseconds and then checks whether the query has changed in the meantime. If

this is not the case, the query is executed, otherwise nothing happens. This mechanism prevents unnecessary queries from being executed if the user is typing very fast.

When a query is executed, it is sent together with an offset indicating the current page to the server in asynchronous fashion. The server generates the HTML code for the status message, the suggestion cloud, the result list and the paging mechanism. These are enclosed in a single `<div>` tag, which is used to replace the according tag in the DOM of the static Hayoo! HTML page. If the user changes the page by clicking one of the navigation links, the current query is just executed again with a different offset indicating the respective page.

The words in the suggestion cloud are linked to a small JavaScript function. The function is used to replace the according prefix term in the text box by the suggested completion and to re-execute the query afterwards. This enables the user to quickly restrict the query by just clicking on the respective suggestion. Usually, suggestions are displayed in a drop-down menu just below the text box used for entering the query. Because this only allows one-dimensional scaling by ordering the elements in the list and restricts the number of suggestions to just a few items, the cloud representation was chosen for Hayoo!.

The current version of Hayoo! uses the Janus application server (available from [Uhl]) as servlet container and web server. It is entirely written in Haskell and allows a seamless integration of Haskell code in web applications. In total, Hayoo! provides a fast, interactive and easy to use search interface for Haskell documentation.

5 Conclusion

This chapter provides a summary of the previous chapters which includes a review of what has been achieved. The original requirements are compared to the functionality implemented by the Holumbus framework. An outlook on future work and development is derived from this comparison. Additionally, the experience gained from the use of Haskell in this project is outlined.

5.1. Summary and Results

The original goal of this work has been the development of a full-text search engine, which is able to deal with large amounts of data and could be easily adapted to particular use-cases. This goal has been almost completely reached, as shown by the implementation of a specialized search engine like *Hayoo!*. The only thing remaining open is the scalability of the index data structure with the size of the document collection. Because the current implementation holds all index data in system memory, there is a clear upper limit on the index size. Right now, this restriction is only leveraged by distribution of index data onto several machines. As this solution imposes additional hardware requirements, it might not be acceptable for every user of the framework.

High flexibility and customizability of the framework has been a very basic requirement. The framework offers several mechanisms for the integration of custom functionality. At first, a lot of modules from the core library can be used quite independently from each other. Users could just pick the index data structures or the parser for the query language. Secondly, clear interfaces for the core data structures enable users of the framework to supply and integrate their own implementations for these critical components. More fine granular customizability is available through the use of functions as configuration parameters. For example, users can provide customized functions for the ranking of results.

Scalability through distribution of index data has been another basic requirement. This is mainly reached by providing several split functions for index data. These can be used to partition a large index into several smaller ones by different splitting schemes. The query processor is already prepared for distributed query evaluation. A function for partial evaluation is provided, which returns an intermediate result structure usable for further processing. Based on this core functionality, a simple client/server architecture is included in the framework. It supports distributed evaluation of queries for indices splitted by documents and provides a convenient update mechanism for distributed indices (also see the performance figures in appendix C).

The Holumbus framework requires any index structure to support prefix queries. Hence the requirements for interactive query evaluation using suggestions and *find as you type* techniques are met. The current index implementation is based on the inverted file and uses a trie structure for the dictionary to enable prefix queries. This combination has proven its eligibility through very fast lookup times and moderate space requirements. The transformation of result data requires far more time than the lookup itself. If large result sets are efficiently limited to reasonable size, queries can be evaluated in tenths of a second. The result structures provided by the Holumbus framework enables users to create highly interactive search interfaces.

As phrase searches are expected by most users to be supported by a search engines, these have also been a basic requirement. They are not directly supported by the index data structures. Instead, these are only required to support lookup functions for exact words. The provided functions are used to implement phrase searches in the query processor by issueing several queries and comparing the positions returned for every word.

Support for structural queries has been another requirement for the Holumbus framework. Because a specialized search engine is possibly used for collections of uniform and structured documents, there should be a way to exploit this fact for the generation of better search results. The current index implementation in fact uses

several internal indices which can be searched independently. These are used to model the inherent structure of documents and to enable structural queries.

The Holumbus framework includes a unique technique for fuzzy queries, based on simple replacement rules and character permutations. Hence the remaining requirement for fuzzy queries is met by the framework, too. Despite its simplicity, the generation of a fuzzy set from a single query term through these mechanisms seems to be able to improve retrieval quality at least for common spelling and typing errors.

5.2. Lessons Learned

Due to the slightly exotic nature of Haskell, several interesting experiences made during the development of the Holumbus framework are consequences of the use of Haskell and will be elaborated in this section.

The current version of the framework was developed in about three man-months and consists of roughly 3,900 lines of code (excluding the indexer components). Additionally, about 1,400 lines of code for tests and 2,900 lines of example code have been written. Estimates for implementing the same functionality in Java vary between four to six months, which is about two times longer. Given the fact that there was almost no experience in functional programming and Haskell in particular at the beginning of development, these figures lead to the conclusion that programmer productivity is clearly increased by the use of Haskell.

The overall increase in productivity seems to be caused by several advantages resulting from the use of Haskell. First of all, Haskell provides a lot of flexibility through the treatment of functions as first-class values and the resulting possibility to use and create higher-order functions. For example, almost every time a function is used as a parameter for some abstract processing scheme, an object-oriented language would have required the implementation of the (nontrivial) *strategy* pattern [GHJV95].

Another advantage of Haskell contributing to the increase in productivity is the possibility of easy and fast creation of small prototypes and proof-of-concepts, which help finding the right way of implementing a specific feature. This style of development is very much encouraged by the language itself, because every rough sketch in Haskell code almost always leads to an executable specifications of the required functionality. After pointing into the right direction, these specifications can be either used as a guideline for a clean implementation from scratch or be further refined until the original requirement is met.

As mentioned before, Haskell's strict type system helps achieving code robustness a lot. But the real advantage (and resulting increase in productivity) of a strong

and static type system is revealed during collaborative development. Because Haskell requires a reasonable amount of thinking about data types and interfaces, these are often designed in a much cleaner way. This fact in combination with the absence of any unambiguousness in Haskell's type system is really helpful in a collaborative development environment.

Despite all of these benefits resulting from the use of Haskell, some problems have been encountered, too. The main issue during development was the lazy evaluation strategy employed by Haskell. Because search engines usually process large volumes of data, space efficiency of the internal data structures is a critical factor for practicability. Unfortunately, lazy evaluation does not help in measuring the space efficiency because one never really knows if the data structures are fully evaluated or if there are still some computations pending (and therefore keeping eventually larger structures in memory). Hence, it is hard to tell whether there is too much laziness in place or the data structures are just badly designed if the measured memory usage is higher than expected. This issue is hard to overcome, as avoiding such space leaks often requires a carefully chosen amount of strictness in a program. But the more complex a program gets, the more difficult is the correct placement of strictness annotations.

5.3. Future Work

The current version of the Holumbus framework provides a good starting point for the implementation of search engines. Nevertheless, there are a lot of things that can be improved. An overview about these issues is given in this section.

The index data structures probably holds the most potential regarding scalability and further improvements on space efficiency. As mentioned in section 2.3.3, there are more efficient variants of the Simple-9 encoding scheme, which could be employed for the compression of occurrences. At the same time, it may be possible to use ByteStrings for the representation of keys in the trie structure, which have the ability to share substrings and could therefore avoid some of their initial overhead. Additionally, at least the occurrences of the index structure could be externalized onto secondary storage media. This could be done using *B-Trees*, which seem to be suitable for keeping large data quantities on disk storage [CLRS04].

Concerning retrieval speed, caching of intermediate results probably pays off and is worth the effort for implementation, especially in distributed environments. Likewise, the query optimizer is currently quite primitive and could be extended to employ more sophisticated techniques for algorithmic transformation of queries. Additionally, distributed query evaluation could be adapted to the *MapReduce* programming model

[DG04; Lä06]. This simple abstraction for parallel programming could be a great help towards scalability of concurrent (and distributed) query processing.

Although the inverted file structure has widely proven its eligibility for full-text search engines, there are some interesting new index data structures emerging from current research developments. For example, the *Hybrid Index* promises to combine the space efficiency of the inverted file with direct support for suggestions and thus avoids costly transformations of result data structures [BW06; BW07].

As ranking of retrieval results is an underemphasized topic in this work, this area holds a lot of potential for further improvements. As a first step, several more sophisticated ranking methods could be included in the framework, like the cosine-measure mentioned in section 2.2.3.

Finally, some research is needed on the fuzzy query mechanism used by the Holumbus framework. A scientific study on the consequences for retrieval quality could give further insights into the effect on the user experience.

5.4. Outlook

The Holumbus framework is available at `http://www.holumbus.org` under the MIT open-source software license and will therefore hopefully continue to evolve. As mentioned in the previous section, there is still a lot of work to do. However, the framework has reached a usable state and is expected to be used for custom search engines in the near future. Installation instructions and a brief introduction to the API are given in appendix A.

The Hayoo! Haskell API search is publicly available at `http://www.holmbus.org/hayoo` and will be continuously improved. The next major step is the inclusion of all Haskell documentation available on *Hackage* (a central repository for Haskell packages) and the establishment of Hayoo! as a general purpose interface to Haskell API documentation.

The *MapReduce* programming model is already scheduled for inclusion in the Holumbus framework and succeeding work on this has already been started. MapReduce will mainly be used to facilitate parallelization and distribution of the crawling and indexing process, but as mentioned before, could also be used for distributed query evaluation.

A Manual

The Holumbus framework is available at http://www.holumbus.org. The project homepage also contains some additional documentation and information about the framework. This manual only covers the query part of the Holumbus framework.

A.1. Requirements and Installation

The Holumbus framework has been tested in 32-bit and 64-bit GNU/Linux environments. Building the framework requires a working Haskell environment with at least GHC 6.8 and the according Haskell hierarchical libraries. Cabal 1.2 or later is required for building and packaging and should be shipped together with GHC 6.8. In addition to the standard libraries, the following packages are needed (available from http://hackage.haskell.org):

- binary 0.4.1 or later – Data.Binary
- bzlib 0.4.0.1 or later – Codec.Compression.BZip
- HDBC 1.1.4 or later – Database.HDBC
- HDBC-sqlite3 1.1.4.0 or later – Database.HDBC.Sqlite3
- hxt 7.5 or later – Text.XML.HXT

- regex-compat 0.71.0.1 or later – Text.Regex
- utf8-string 0.2 or later – Codec.Binary.UTF8.String

Running the test suite requires HUnit 1.2.0.0 or later as well as QuickCheck 1.1.0.0 or later. The simple examples do not require any additional packages, the more complex examples require the Janus application server 1.0 or later for the web interface and hslogger 1.0.5 or later for logging purposes.

The core library is built and installed using Cabal. For a simple installation, the following commands have to be executed in the Holumbus root directory:

```
$ runhaskell Setup.hs configure
$ runhaskell Setup.hs build
$ runhaskell Setup.hs install # with root privileges
```

The API documentation for the core library can also be generated using Cabal. To create the documentation, the following command has to be executed in the Holumbus root directory:

```
$ runhaskell Setup.hs haddock
```

The test suite is built and executed using `make`. To run all tests, the following command has to be executed in the Holumbus root directory:

```
$ make alltests
```

Building the simple examples requires the core library to be installed (see above). To build the examples, the following command has to be executed in the Holumbus root directory:

```
$ make allexamples
```

The more complex examples have to be built in combination with the Janus application server. The exact procedure is described in the `README` file available in the root directory of every example.

A.2. Basic Usage

Building a simple query interface using the Holumbus framework is probably best illustrated by taking a look at the source of the *SimpleSearch* example. Basically, querying an index involves the following steps:

1. Loading an index and its document table using Holumbus.Index.Common.loadFromFile. As this is a generic function, the specific type of the structure to load has to be specified, e.g. **do** idx <− (loadFromFile "path/file") :: **IO** Inverted.

2. Creating a query either by using a custom parser or the parser for the default syntax. The latter is available by calling Holumbus.Query.Language.Parser.parseQuery.

3. Passing the index, the document table and the query together with a custom configuration to Holumbus.Query.Processor.processQuery. The documents matching the query are returned together with possible completions.

4. Ranking the documents and words by passing the result together with appropriate ranking functions to Holumbus.Query.Ranking.rank.

The result structure can now be used to display the respective documents to the user or to provide suggestions based on the completions included in the result. The function Holumbus.Index.Cache.getDocText can be used to retrieve the full text of a document for preview purposes.

More detailed information about the available functions and modules is provided by the API documentation of the core library. A rather complete example of what is possible with the Holumbus framework is available through the Hayoo! web search.

B
Query Syntax

The default syntax for the query language allows complex queries to be created from basic primitives and several operators. The exact syntax is denoted here, splitted in its nonterminal symbols and its terminal symbols.

B.1. Nonterminal Symbols

A query may be built from the following basic operators: The implicit AND operator, denoted by space and the explicit AND, OR and NOT operators, denoted by the terms "AND", "OR" and "NOT". Queries may be enclosed in brackets to influence the operator precedence (unary operators have highest priority, followed by the OR operator and leaving the AND operator with lowest priority). The context operator : is used to restrict the query to a number of contexts, the case-operator ! will make a query case-sensitive and the fuzzy-operator $\sim$ will perform a fuzzy search.

$$
\begin{array}{lll}
query & ::= & andQuery \\
andQuery & ::= & orQuery\ and\ andQuery \mid orQuery \\
orQuery & ::= & notQuery\ or\ orQuery \mid notQuery \\
notQuery & ::= & not\ contextQuery \mid contextQuery \\
contextQuery & ::= & contexts\ spec\ parQuery \mid parQuery
\end{array}
$$

$parQuery$ ::= $leftPar\ query\ rightPar \mid caseQuery \mid fuzzyQuery$
$caseQuery$ ::= $case\ simpleQuery \mid simpleQuery$
$fuzzyQuery$ ::= $fuzzy\ wordQuery$
$simpleQuery$ ::= $wordQuery \mid phraseQuery$
$contexts$ ::= $context \mid context\ sep\ contexts$

B.2. Terminal Symbols

The basic query elements consist of a single word or a phrase enclosed in quotation marks. A context may be specified with the name of the context and a following colon. The terminal symbols are denoted in regular expression syntax.

$wordQuery$ ::= `[^"\(\)\s]+`
$phraseQuery$ ::= `"[^"]+"`
$context$ ::= `[a-zA-Z0-9]+`
$leftPar$ ::= `\(`
$rightPar$ ::= `\)`
$fuzzy$ ::= `~`
$case$ ::= `!`
$spec$ ::= `:`
sep ::= `,`
and ::= `AND|\s*`
or ::= `OR`
not ::= `NOT`

C
Performance

In the following, some figures on query performance in both distributed and single machine environments are given. They are all based on the *vl-index* which contains 449 documents and about 96.000 words. The index is either used as a whole or splitted into two parts. The *Benchmark* program provided by the Holumbus framework was used to execute 4.134 automatically generated queries. Three systems were used in different combinations in a switched GBit ethernet environment:

- **A**: Athlon 64 3200+, 2.0 GHz, 2.5 GB RAM
- **B**: Core2Duo T7500, 2.2 GHz, 3.0 GB RAM
- **C**: Athlon XP 1200, 1.2 GHz, 768 MB RAM

Machine	*Running time*	*Seconds per query*	*Queries per second*
A	77.87 s	0.019	53.09
B	78.50 s	0.019	52.66
C	119.06 s	0.029	34.72

Table C.1.: *Performance figures for each single machine.*

The results of the benchmark program for each of these machines are given in the above table. Currently, the benchmark program is single threaded and makes no use of

the additional cores available in multi-core CPUs. These performance figures provide some insight into the efficiency of query evaluation on very different systems.

Compression	*Running time*	*Seconds per query*	*Queries per second*
enabled	480.43 s	0.116	8.60
disabled	420.27 s	0.101	9.84

Table C.2.: *Comparison of distributed query performance regarding compression.*

The impact of compressed transmission of query results on the overall query performance is measured by using all three systems in a distributed configuration. Machine A is used to run the benchmark program while machines B and C are running the query servers.

Configuration	*Running time*	*Seconds per query*	*Queries per second*
local	273.37 s	0.066	15.12
remote	453.39 s	0.110	9.12

Table C.3.: *Comparison of distributed query performance regarding network latency.*

The influence of network latency on distributed query evaluation is measured by using only two systems. Machine B is used to run two query servers because of its dual-core CPU. The benchmark program runs either locally on machine B or via network on machine A.

Acknowledgments

This work could not have been done without the help of several other people. First of all, I would like to thank my supervisor at FH Wedel University of Applied Sciences, Prof. Dr. Uwe Schmidt for his overall support, many fruitful discussions and the encouragement for writing a paper for ICFP'08.

The same thanks are due to my fellow student Sebastian M. Schlatt, who always managed to provide me with lots of example index data and had an open ear for many technical questions. Additionally, I have to thank Stefan Schmidt for a lot of information about search engines and index data structures.

Special thanks go to Axel Tetzlaff, who helped out on problems regarding the Hayoo! web interface and several other HTML, CSS and JavaScript related questions. I also have to thank the guys from `#haskell` on `freenode.net` as well as on the *Haskell-Cafe* mailing list, who have answered every single question, no matter how simple.

Many thanks go to my father, Wolfgang Hübel as well as Hanna Kranz, who have proof-read this work and the resulting submission for ICFP'08 and helped to make all possible errors and mistakes vanish.

Bibliography

AM05 Anh, Vo Ngoc ; Moffat, Alistair: Inverted Index Compression Using Word-Aligned Binary Codes. In: *Information Retrieval* 8 (2005), Nr. 1, pages 151–166

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

Bra07 Bray, Tim: Finding Things. In: Oram, Andy (Hrsg.) ; Wilson, Greg (Hrsg.): *Beautiful Code*. Sebastopol : O'Reilly, 2007, pages 41–57

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, Ricardo ; Ribeiro-Neto, Berthier A.: *Modern Information Retrieval*. New York : ACM Press, 1999

Cab *The Haskell Cabal*. http://haskell.org/cabal, last checked: 04.03.2008

CH00 Claessen, Koen ; Hughes, John: QuickCheck: a lightweight tool for random testing of Haskell programs. In: *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. New York : ACM Press, 2000, pages 268–279

CLRS04 Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald ; Stein, Clifford: *Algorithmen – Eine Einführung*. München : Oldenbourg, 2004

CSL07 Coutts, Duncan ; Stewart, Don ; Leshchinskiy, Roman: Rewriting Haskell Strings. In: *PADL' 07: Practical Aspects of Declarative Languages 8th International Symposium*. Heidelberg : Springer, 2007, pages 50–64

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

dMSZ98 de Kretser, Owen ; Moffat, Alistair ; Shimmin, Tim ; Zobel, Justin: Methodologies for Distributed Information Retrieval. In: *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*. Washington D. C. : IEEE Computer Society, 1998, pages 66–73

Fer02 Ferber, Reginald: Dokumentsuche und Dokumenterschließung. In: Rechenberg, Peter (Hrsg.) ; Pomberger, Gustav (Hrsg.): *Informatik-Handbuch*. 3rd edition. München : Hanser, 2002, pages 913–934

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

Ghc *The Glasgow Haskell Compiler*. http://haskell.org/ghc, last checked: 03.03.2008

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

Hug89 Hughes, John: Why Functional Programming Matters. In: *Computer Journal* 32 (1989), Nr. 2, pages 98–107

Hut07 Hutton, Graham: *Programming in Haskell*. Cambridge : Cambridge University Press, 2007

JH93 Jones, Mark P. ; Hudak, Paul: Implicit and explicit parallel programming in Haskell / Yale University. Version: 1993. http://web.cecs.pdx.edu/~mpj/pubs/RR-982.pdf. New Haven, 1993 (YALEU/DCS/RR-982). – Research Report

Ken04 Kennedy, Andrew J.: Functional Pearls: Pickler Combinators. In: *Journal of Functional Programming* 6 (2004), Nr. 14, pages 727–739

Knu98 *Chapter* 6.3 Digital Searching. In: Knuth, Donald E.: *The Art of Computer Programming*. Vol. 3, Sorting and Searching. 2nd edition. Reading : Addison-Wesley, 1998, pages 492–512

LM01 Leijen, Daan ; Meijer, Erik: Parsec: Direct Style Monadic Parser Combinators for the Real World / Department of Computer Science, Universiteit Utrecht. Version: 2001. http://research.microsoft.com/users/daan/download/papers/parsec-paper.pdf. Utrecht, 2001 (UU-CS-2001-27). – Research Report

Luc *Lucene*. http://lucene.apache.org, last checked: 03.03.2008

LV03 Lyman, Peter ; Varian, Hal R.: *How Much Information?* Version: 2003. http://www.sims.berkeley.edu/how-much-info-2003, last checked: 07.11.2007

Lä06 Lämmel, Ralf: *Google's MapReduce Programming Model – Revisited*. 2006. – Accepted for publication in the Science of Computer Programming Journal

Mit Mitchell, Neil: *Hoogle*. http://haskell.org/hoogle, last checked: 10.12.2007

MM93 Manber, Udi ; Myers, Eugene W.: Suffix Arrays: A New Method for On-Line String Searches. In: *SIAM Journal of Computing* 22 (1993), Nr. 5, pages 935–948

Mor68 Morrison, Donald R.: PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. In: *Journal of the Association for Computing Machinery* 15 (1968), Nr. 4, pages 514–534

Pey03 Peyton Jones, Simon: *Haskell 98 Language and Libraries: The Revised Report*. Cambridge : Cambridge University Press, 2003

RNB98 Ribeiro-Neto, Berthier A. ; Barbosa, Ramurti A.: Query Performance for Tightly Coupled Distributed Digital Libraries. In: *DL '98: Proceedings of the 3rd ACM conference on Digital Libraries*. New York : ACM Press, 1998, pages 182–190

Sch Schmidt, Uwe: *Haskell XML Toolbox*. http://www.fh-wedel.de/~si/HXmlToolbox, last checked: 04.03.2008

Sch07 Schmidt, Stefan: *Typhoon 2 - Volltextsuche*. 2007. – Diploma Thesis

Sch08 Schlatt, Sebastian M.: *The Holumbus Framework: Creating scalable and customizable crawlers and indexers*. 2008. – Master's Thesis

TGM93 Tomasic, Anthony ; Garcia-Molina, Hector: Performance of Inverted Indices in Shared-Nothing Distributed Text Document Information Retrieval Systems. In: *PDIS '93: Proceedings of the 2nd international conference on Parallel and Distributed Information Systems*. Los Alamitos : IEEE, 1993, pages 8–17

Uhl Uhlig, Christian: *Janus Application Server*. http://darcs.fh-wedel.de/janus, last checked: 25.03.2008

Wad98 Wadler, Philip: An Angry Half-Dozen. In: *SIGPLAN Notices* 33 (1998), Nr. 2, pages 25–30

WMB99 Witten, Ian H. ; Moffat, Alistair ; Bell, Timothy C.: *Managing Gigabytes – Compressing and Indexing Documents and Images*. 2nd edition. San Francisco : Morgan Kaufmann, 1999

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 appropriatly. 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.

Place, Date *Timo B. Hübel*