Technische Universität Wien

DIPLOMARBEIT

Semantic Information in Document Retrieval Systems

Ausgeführt am

Institut für Informationssysteme
Arbeitsgruppe für Datenbanken und Artificial Intelligence
der Technischen Universität Wien

Unter der Anleitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Georg Gottlob und
Mag.rer.nat. Dr.rer.nat. Robert Baumgartner
als betreuender Assistent

durch

Martin Domig
Hippgasse 25/20

A-1160 Wien

Wien, 12. Dez. 2005

Martin Domig

Abstract

The constant and fast growth of the internet and the world wide web results in a problem known as information overflow. For a user who searches for information regarding a specific topic, it is very hard to find the few web sites that cover his scope of interest. To be able to navigate through the large amount of data, users need the help of automated content categorization mechanisms, also known as internet search engines. The task of those search engines is to process user queries and search through all sites on the web, fetching only those sites carrying content that is related to the users topic of interest.

However, understanding the semantic content of any document written in a human natural language is very difficult for a machine. Very often, search engines are limited to string or word matching only, possibly also exploiting the very little additional meta information that is given by the linked structure of the web itself. Truly understanding the meaning or sense of a text is beyond the scope of common search engines.

To overcome this problem, machines need additional information that helps in understanding and categorizing web sites. This thesis presents the implementation of a document retrieval system called *Monkville*, that introduces the concept of context information to documents. Context is defined as semantically relevant text, which is not part of a document itself, but closely related to the documents semantic content.

Context was extracted from web sites using *Lixto*, a very powerful, semi-automated information extraction utility. Text and context information was then processed in the document retrieval system, enabling it to perform queries based on context as well as text of the documents. This functionality provides new ways to perform full text queries in search engines, resulting in a document retrieval system capable of processing semantic searches.

Experiments conducted on several thousand texts that were extracted from various online newspapers showed that this approach of context processing gives good results. Context information is available on many web sites, unfortunately it is, most of the time, given only implicitly, embedded in the web site. If such information can be made explicitly known to document retrieval systems, it is possible to conduct new and very powerful semantical searches on the internet.

Contents

1 Introduction

The internet has become the fastest growing virtual database, but no one knows exactly how large it really is. Unfortunately, the size is not the only unknown dimension of the "world wide web". Our knowledge about the content, the quality of the content and the location of any specific information is even less. In fact, the world's largest virtual database turns out to be the one that is the most difficult to use efficiently.

This is due to the original nature of the web itself. Initially, the construction of what should later become "the internet" began during the cold war in the 60ies, as a communication framework that could maintain operability even after the failure of any node. This led to a decentralized network structure, without a controlling or monitoring entity. At this time, nobody could have foreseen the explosive growth that happened in the subsequent years. The decentralized design allows the network to expand into any direction, whenever needed - and it has been expanding ever since.

The same applies to the websites on the internet, and their content. Everybody can publish any information at little or no cost, without anyone validating the information, and without any need to register the website in some form of central registry. The result is a peculiar situation: the decentralized nature of the internet, while being the most important reason for its success, is at the same time the biggest problem when it comes to using the network as a source of information. There is an enormous amount of data in the network, but the exact amount and content is unknown. Without a global index, it is extremely difficult to find any specific information, and even if the information can be found, there is no guarantee whatsoever about its quality.

The only way to make the internet usable as a source of information is by creating a global registry of as many websites as possible. Due to the amount of data, this can only be done by an automated process. This registry can then be used by humans to quickly retrieve a list of websites for all possible topics of interest. Search results returned by such registries should be short because they are going to be processed manually, thus they should contain the most relevant websites only. This is what internet search engines are trying do: be of help in navigating through the world wide web.

However, there are some fundamental problems involved with the automated processing of websites. First of all, no registry can ever claim to cover all websites, because a complete list of sites does not exist. Next, it is very difficult to process the contents of a website to begin with. They are only weakly structured and content often is not organized at all, which makes it very unsuitable for automated processing. Website content is intended

to be viewed by humans, not by computers. Not even humans are able to understand all websites, because they could be written in any language and script.

1.1 Search Engines

The information explosion in the internet leads to an issue commonly known as *"resource discovery problem"*: due to the large amount of data, it is very hard to find one specific piece of information. This calls for methods to relieve the user from this problem. [50]

The perfect internet search engine would understand all human languages and have humanistic background knowledge, it would contain all web sites and it would always be up to date. Unfortunately, this is impossible. The efforts that are required to create a search engine that is just reasonably usable is enormous, and there exist only a hand full of such engines that could come close to a somewhat complete internet index. Computer systems are not suited to process unstructured data such as text in a natural language. [25] "Understanding" web sites is, even for humans, a very complex task.

The duty of an internet search engine is to support a human user in the process of finding web sites that are relevant to his topic of interest. The goal is not to replace browsing in the internet with a keyword search, but to supplement it and facilitate the navigation on the world wide web. Search engines do that by adding as much websites as possible to a document retrieval system, create an index of keywords and then present a user interface for human users, which can be used to perform search queries. Document retrieval systems try to determine, with all means possible, the semantic information contained within a document, and what the key topics are, and estimate the semantic relevance of the document to the users search terms.

To estimate the relevance of a document, a search engine needs to "understand" the content of a website, at least to a certain degree. Based on the content of all documents, the search engine creates a list of keywords - the index. More advanced systems try to somehow interpret the semantic meaning of the text, and try to group similar documents together in a query result to improve the usability for the human user. Based on semantic information, it is possible to create such content grouping. However, the degree to which a computer system is able to understand the semantic meaning of any text written in a natural language is very limited. [25]

There are two basic ways for thematic grouping of documents: mapping documents onto existing ontologies [33] and fuzzy content clustering. The first method can achieve a good performance in recognizing the best (existing) topic for a given document, while the second approach is not limited to the

already existing categorization – but has the problem of automated topic classification. Both approaches have in common that the computer has to somehow "understand" the content of the text, at least to a degree exceeding the plain creation of a key word index.

The prevalent techniques to interpret the semantic content of natural language text are based on purely statistical and mathematical methods. They calculate key values by statistically evaluating word distributions in single documents (local evaluation), and compare it to the average of word distributions over the global text collection (global evaluation). Based on the differences between those two values, they try to find a good compromise for a document categorization.

Local evaluations only respect the currently processed document, without knowledge about the characteristics of an entire text collection. Global values are calculated over the total text collection, and need to be updated as the text collection changes. In document retrieval systems, both play an important role. Local evaluations are generally more precise and accurate when being compared to a given search term, but they often are inefficient. Global values lack the fine granularity of local evaluation, but show a good overall performance. A good example for a global evaluation system is the *PageRank* [6] algorithm, with all the side effects described below. *PageRank* will be explained in detail in chapter 4.5.1.

The proper application of purely statistical evaluation methods to large, heterogeneous and uncontrolled text collections – such as web sites – has not been a subject of research for a very long time. Such text collections have been available for a few years only, and first experiments with that kind of data, while confirming a certain degree of efficiency for standard techniques on full text databases, have revealed many surprises and problems. [25]

Purely statistical ranking algorithms are limited in the quality of the results, and usually are vulnerable to search engine spammers. It is possible to create artificial documents, specifically crafted to achieve a high ranking in statistical evaluation algorithms. For *PageRank*, the most common and obvious attack is *link farming*[1]. For *TFxIDF*[2], a very prominent ranking algorithm (see chapter 4), a document that contains exactly (and only) the users search terms will achieve the best result. Because the quality of web sites is often very limited and cannot be assured, this can happen on a regular basis, especially with search engines that contain a very large text collection.

The structure of web documents also offers to exploit some additional

[1] A large collection of small web sites that contain links to each other and to the favored web site, see also `http://en.wikipedia.org/wiki/Link_farm`

[2] *TFxIDF*: Term Frequency times Inverse Document Frequency, a very popular ranking algorithm in document retrieval systems. See also chapter 4.2

meta-information, like hyper links. However, such intuitive approaches are often insufficient. [50] Once the ranking algorithm in use is known, there are ways to exploit the weak spots of the statistical evaluation – which is one of the reasons why search engine operators tend to keep their ranking functions a closely guarded secret.

1.1.1 Search Engine Requirements

An internet web search engine has requirements that are different to classical database systems. First of all, it operates mostly on unstructured, heterogeneous textual data. It will, over time, become very large and thus has to be scalable to support the ever growing text collection. Most importantly, the requirements of a document retrieval system that is to be used as a internet search engine is fundamentally different to the requirements of a standard (relational) database system: the content does not necessarily need to be consistent at all times, and changes to the index usually are not time critical. Furthermore, the results for search queries are not clearly defined, but rather fuzzy: "relevance" is a subjective property that is evaluated differently by different users.

The critical factors for a web search engine are different: it has to provide a high availability, and only little response times. [23] Therefore, a search engine has to be able to add documents to the index at run-time without affecting the search performance, a feature referred to as *"hot updates"* (ref. to chapter 3.2.4). However, updates can happen in batches in order to improve the overall performance.

It is not mission critical to add new sites to the database immediately, neither is it necessary to remove them instantly. Compared to a relational database, these requirements seem to be very comforting – but in fact, they are vital to various implementation details that can improve the overall system performance and reduce computing costs. Processing of natural language texts on a large scale is difficult enough, adding the unnecessary burden of absolute data integrity at all times is not necessary.

1.2 Motivation

Purely mathematical ranking and grouping algorithms are inherently limited in the quality of their results. They can only consider statistical distributions, and maybe apply some plausibility checks based on secondary knowledge databases (such as dictionaries), but computers are unable to understand the semantic meaning of a written text. Thus, purely statistical ranking and grouping algorithms will always produce results that, while of acceptable

quality, are poorer than what a human could do. Unfortunately, there is currently no viable alternative to mathematical methods.

However, it is possible to increase the quality of the results by increasing the amount and granularity of input data, and by adding additional semantic information to the text. This requires additional knowledge about the processed documents, which often is not available or not obviously present for a computer – but it can be made know explicitly by annotating it before documents are processed further.

The idea is: if a computer is unable to understand on its own what a text is about, it needs to be told in the form of a small semantically relevant context - for example, a summary. Computers won't understand that either, but the additional contextual data could increase the precision of statistical ranking and grouping algorithms, by providing – for each document – additional semantic information, as shown in Figure 1. This contextual data is not part of the text itself, but will be processed together with it. If the context information is well chosen, of good quality and semantically coherent to the document itself, it should be possible to improve the quality of ranking results and content clustering.

In the light of the ever growing internet and the growing amount of textual data on web sites, such capabilities will be of constantly increasing importance in the future.

Currently, one of the main problems for this idea is the availability of semantic information in the internet, or better – the lack thereof. With the *Semantic Web* initiative[3] this could change in the future. But currently, the amount of sites that annotate content with semantic information did not reach the critical mass yet that is necessary to make it interesting for internet search engines. Until this happens, the extraction of semantic information will be a semi-automated process. With the right tools [49], the amount of manual work is minimal, but the time being, it is next to impossible to exclude it entirely.

Existing algorithms like *TFxIDF* (see 4.2) or *Most-cited* (or the better known variation *PageRank*, see 4.5) use meta-data that is present in the structure of web sites – such as hyper links or text markup. The semantic approach presented here is different: it is based on data that needs to be extracted by semi-automated content filtering utilities, which, while having the disadvantage of not being fully automated, leads to an unexpected advantage: the content of the textual database itself as well as the additional semantic context information can be controlled, and is of high quality.

[3]Semantic Web: `http://www.w3.org/2001/sw`

Figure 1: Extracted BBC article. The portions to the right marked with red rectangles are extracted as contextual information, the remaining article in its entirety as document content

1.3 Use Cases

Some of the algorithms and concepts presented in the subsequent chapters will be explained by fictionally applying them to small use cases. The goal is to demonstrate how different algorithms and ideas can be applied in a practical implementation, and to point out some problems, difficulties and fundamental errors that may exist.

Most use cases will be related to the actual project environment itself, which is described below. Sometimes, fictional set ups without any practical relevance will be used to hilight weaknesses, that might not be obvious when applied to a generic problem.

1.3.1 The Project Environment

The assumption is a textual database that stores news articles about all kind of events that happened in the past. The application will be able to perform full text searches on those articles, and will – to a certain degree – be able to derive semantic and causal relations between them. Based on these

relations, the user can derive his own conclusions, and can use the system for investigations and in-depth topic related researches of historical facts (or better, the news coverage thereof).

To do this, the user wants to search for semantical concepts rather than specific keywords. For example, a query regarding the financial situation of the united states could be formulated like "national finances USA", instead of "health budget deficit Louisiana". A query for people's responses to high taxes might be formulated like "unfair taxes", instead of "tax rebel goes to jail".

Textual data is being acquired from a list of well regarded news sites, filtered with *Lixto*[4] and then stored in a *MySQL* database for later processing and performance evaluation. The content extraction and filtering for this project was done by Lixto, a tool for semi-automated data extraction from semi-structured data sources like web sites. [2, 3]

News articles were retrieved from the web sites of the following online media: *"The New York Times"* (`www.nyt.com`), *"The British Broadcasting Corporation"* (`www.bbc.co.uk`), *"MSNBC online"* (`www.msnbc.msn.com`) and *"The Guardian"* (`www.guardian.co.uk`). These sites were chosen because they provide additional editorial information to most of the published articles, like headlines of related stories, links to websites with related content and other additional information.

This test environment is in some points different from a situation of a web search engine that has to operate on unfiltered web sites. The difference is the quality of the content: textual data is of a high consistency, can be assumed to have a reasonable minimum length and does, due to the filtering, not contain all the "noise elements" that is present in HTML pages (like site navigation links, forms or advertisement). While there are projects towards the automated processing of web site components (see also Related Work, section 8.2 on page 71), this technology is not yet generally available. Thus, the results of this project are not directly applicable to the domain of internet search engines.

[4] Lixto: `http://www.lixto.com`

2 Document Retrieval

"Document retrieval systems are for the user who wants to learn about something by reading about it" [25], as opposed to expert systems that answer questions by building a logical knowledge database, and interact with the user in a dialogue to solve a particular problem. Instead of a knowledge database, a document retrieval system consists of a database of documents, a classification algorithm to build an index from the documents, and an online user interface to access the database. [8]

Internet web search engines are classical examples of document retrieval systems, on a very large scale. There are many smaller information retrieval applications, like indexing software for corporate office documents and correspondence, none of them reaching the dimensions of a web search engine.

There are two main classes of indexing schemata for DR systems: *form based* (or *word based*), and *content based* indexing. Form based DR addresses the exact syntactic properties of a text, comparable to substring matching in string searches. The text is generally unstructured and not necessarily in a natural language, the system could for example be used to process a large set of chemical representations in molecular biology. The content based approach exploits semantic connections between documents and parts thereof, and semantic connections between queries and documents. The document classification scheme (or *indexing algorithm*) used determines the basic nature of a document retrieval system.

All DR systems have a basic principle in common: they process text content and user queries in a natural language, and *"have to find relationships between the information needs of users and the information held within documents, both considered in a very general sense, and neither directly available to the computing system."* [25] As elaborated above, natural language processing (NLP) is not a trivial task for any computer system, but it is crucial for good results in document retrieval.

2.1 Definition of Terms

The terms *document retrieval* and *text retrieval* will be used as synonyms here. Both are focused on the same problems and face the same technical requirements. In the past, *document retrieval* (DR) used to be concerned with pointing the reader to offline text resources, typically a book or a journal in a library. With today's amount of storage space available, *text retrieval* (TR) became more practical: index the full document, and present it online to the user to read. [25]

Also, the terms *document* and *text* will be used as synonyms to describe

uninterpreted prose in a natural language, without any specific formatting (like HTML markups), and in arbitrary length – anything between paragraph and book size. The *medium* in which the documents are contained is only secondary, it could be a raw text file, a web site or a proprietary binary text source.

The term *information retrieval* will be used as a collective term for *document retrieval* and *text retrieval*, with the goal to interpret or evaluate semantic content of documents. However, *information retrieval* is not to be confused with *knowledge retrieval* or *information extraction*, also referred to as "question answering". Those are focused on extracting facts from natural language text.

When the difference between any of these terms is of relevance, it will be explicitly mentioned.

2.2 Indexing Algorithms

The indexing algorithm determines the basic nature of the document retrieval system (*content* or *form based*), and has a very large impact on the functionality. Content based algorithms often choose to ignore certain "noise" like *stop words*[5] or interpunctation symbols in a text, which makes them unsuitable for certain classes of documents like program source code. By ignoring this "noise", it becomes impossible to use certain terms or substrings in search queries, or to perform exact syntactic searches (like string matching). Form based approaches, while providing this functionality, have other shortcomings when it comes to evaluating the semantic relevance of a document to any given search query.

The indexing algorithms need to provide some basic functionalities for the document retrieval system: *updating*, *querying* and *result ranking*. *Updating* usually means insertion of new documents. Due to the archival nature of most full text systems deletions are of less importance but still necessary (for example when existing documents need to be updated, which happens regularly in internet search engines). *Querying* is the process of retrieving all records that satisfy a user query, and *result ranking* is the sorting of the resulting records based on their relevance to the search query. [31]

There are some indexing algorithms that do not operate on purely textual data, but are able to process more structured data sources like XML databases, or specialized NLP systems that use a proprietary encoding schema

[5] *Stop words* are words that appear very often in a text and therefore have little or no significance. Many text database systems do not process stop words to save storage space and processing time.

for text. Those algorithms often allow a high degree of additional functionality and flexibility, but are usually not designed with search performance as the primary goal. As such they are not of interest here, the focus will be on performant methods that sacrifice a degree of flexibility to gain speed.

The list of indexing algorithms presented here is not complete, but covers the most commonly used indexing approaches in document retrieval systems. Each of these algorithms has specific advantages and shortcomings.

2.2.1 Suffix Tree

A *suffix tree* or *suffix array* is a data structure that allows to perform exact string searches very quickly. It is flexible enough to be used for a variety of string search problems, for example it allows to operate on regular expressions, string prefixes and phrases, and is specifically suitable for substring searches. Being a *form based* indexing algorithm, *suffix trees* provide more functionality in terms of search details than *signature files* or *inverted indexes*. [15]

A suffix tree is constructed by sorting all *suffixes*[6] in the text in lexigraphical order, and then store them into an array. On this array, it is possible to perform a binary search, with logarithmic costs. [30]

Suffix trees are not only useful to perform string searches in natural language text, or in document retrieval, they are very prominent in general substring matching. The range of usages is very wide and includes applications in molecular biology and data compression. [15]

However, the disadvantage of suffix trees is the very expensive construction of the data structure and the index size. Index construction is either very slow or consumes a lot of memory. The size of the resulting index usually exceeds the size of the text itself, and with the data in the index being of high entropy, it is very difficult to compress the index after it has been created. Changes in the indexed text or updates to the index file itself are very costly.

The features of a suffix tree index come at a cost: no data structure that offers this kind of functionality and supports fast queries even in the worst case is reasonably small. [15] Although there has been some progress with this index form, it is effectively not suitable for document retrieval systems on a large scale. Especially the index size itself presents a big problem for large amounts of text in a database. It is possible to reduce the index size by applying some compression algorithms in the process of index creation as shown by Mäkinen [30] and Grossi et al. [15], but those reductions of index

[6] Formally: S is suffix of $U \Leftrightarrow U = uS$

size come at the cost of a sub-optimal search performance. Also, they are not suitable to achieve the same data compression rate than other indexes.

2.2.2 Signature File

In a *signature file* index, each document is represented by a signature of a fixed size. Words that appear in a document are hashed multiple times to determine which bits in the signature should be set. It is possible (and in fact, happens regularly) that two distinct words in a document set the same bit in the hash, which introduces a degree of ambiguousness into the system. As the database grows, this will happen more often, especially when the hash size is chosen too small. Also, adding stop words to the index increases this problem.

User queries are hashed in a similar way, and then evaluated by comparing the signature of the query to all document signatures. All documents whose signature have a 1-bit corresponding to every 1-bit in the query signature are *potential* answers. Each such document must then be fetched and checked directly against the query to determine whether it is a false match[7] or a true match. [51]

Compared to *suffix trees*, costs for index creation and updates are relatively small, similar to *inverted index* files. Also, the functionality provided by a signature file index matches the one as offered in a *record level inverted index*. However, Zobel et al. demonstrate at great length in [51] why a signature file index is inferior to the *inverted index*, mentioning the following main reasons as sources for performance losses.

A basic problem with signature file indexes is the fixed size of the hash values. It has been suggested to remedy this problem by clustering the database index into several parts, grouped by document length. When processing large amounts of text, the length of the document signature has to increase to keep the number of false positives to a manageable level, which increases query costs – because each signature has to be fetched, irregardless of the actual query – leading to a linear raise of query processing costs with growing index sizes.

Records of varying length, unfortunately very common in document retrieval systems, present a big challenge for signature file indexes. [22] If the hash is too small, the number of false positives grows beyond reasonable levels, if it is too big, the index grows too fast. There are other parameters

[7] False matches happen regularly due to overlapping 1-bits in document signatures. A false match is a database entry which the signature indicates may be an answer, but in fact is not.

that need to be well chosen before starting to create the index. Unfortunately, those parameters depend on the properties of the actual text data being indexed, which might be unknown in advance.

2.2.3 Inverted Index

The *inverted index*, also referred to as *postings file* or simply *inverted file* [18, 52] is a list of distinct terms (the *vocabulary*, or *dictionary*) and, for each single term, an *inverted list* of pointers to documents containing the term. Depending on the implementation, the inverted list might also contain a list of positions of terms, allowing for *phrase queries*[8] in the inverted index. With the smallest processed entity being a single word, this index form does not support substring searches and implements a content based document retrieval system.

Search queries are evaluated by fetching the inverted lists for all query terms, and then merge or intersect the resulting lists of documents depending on the query type.

With the usage of index compression, inverted *record level* index sizes have been reported to be as little as 6% – 10% of the original text size. [51] Additionally to the index itself, it is necessary to store the vocabulary, which usually is small enough to be kept in memory[9].

Almost all document retrieval systems are using an *inverted index* [40] — one of the most prominent being the internet search engine *Google* [6]. The index architecture is intuitive, open for extensions and flexible in functionality, it has been shown to work in many existing applications, and there currently are different research groups working on further improvements of the inverted index (ref. to chapter 8).

Inverted list indexes have been shown to be *"distinctly superior to signature files"* [51] and are better suited for index updates than *suffix trees*. Index sizes are small, and index creation and update costs can be greatly improved by applying data compression and other improvements as shown in section 8. Query times are guaranteed to be optimal, without the excessive amount of false positives as given in a *signature file* index. Being a content based algorithm, inverted indexes are also suitable for adding semantic processing properties to the document retrieval system.

[8] A *phrase query* is a search query over multiple distinct words, where the relative word positions are relevant. A phrase query like "*bird seed*" returns different results than a conjunctive query for "*bird*" and "*seed*".

[9] The amount of distinct words in a text collection grows constantly, even after processing extreme large amounts of documents [47] – but the growth of the dictionary is proportional to the square root of the total text size. [38]

2.2.4 Comparison

Due to the very big costs of index modifications in *suffix trees*, they are not suitable at all for document retrieval systems that contain a changing set of textual data. Each major update would basically require the index to be rebuilt completely, which definitely is not a desirable quality. Furthermore, index size and creation costs are inferior to both, inverted indexes and signature files. The additional functionality of the suffix tree index – form based substring matching in query evaluation – is not necessary for document retrieval systems. In a summary, it can be said that the suffix tree is a very powerful and feature-rich index, but while most of these features are not needed, they have a heavy impact on index creation costs and storage size. Suffix trees are not suited for large scale document retrieval systems.

This leaves the inverted file and signature file indexes as possibilities. The evaluation by Zobel et al. [51] suggests the final conclusion that *inverted indexes* are superior. The main reason for this conclusion is the characteristic behavior of *signature files* to produce *false positives*, which have a high impact on the overall performance. To exclude false positives from answer sets, every possible answer has to be examined against the query, which is very expensive, especially with large answer sets. Unfortunately, with a growing text collection, the answer sets usually grow proportionally, as well as the probability of false positives in *signature file* indexes.

This leaves the *inverted index* as the only viable indexing algorithm for growing text collections: it is reasonably small (and can be compressed further), without the bad characteristics of a signature file index – and it scales to growing text collections. This is why it is the most popular indexing algorithm for full text indexes.

2.3 Precision and Recall

The quality of search results returned by a document retrieval system usually is gauged by two characteristic values: *precision* and *recall*. Sometimes, additional quality measurements are introduced in the literature. [14]

Such performance measures are arguable, because they require extensive a-priori knowledge above the evaluated text collection, and because "quality" of search results is a subjective impression, and as such a property that is very hard to measure with mathematical and statistical methods. In order to make the evaluations of precision and recall in document retrieval systems comparable, there are standard text collections and tests that can be

performed with full text systems, like TREC[10] or CLEF[11]. To estimate precision and recall, the database system needs to contain the respective text collections only. On the indexed test data, a list of defined queries are performed. The results returned from these queries are then compared to a predefined result set, and based on the differences precision and recall values can be computed.

This works because the contents of the test data are known, but has the disadvantage that such text collections usually contain high quality textual data only – a situation that cannot be assumed to be present in internet search engines. Ranking functions that perform well on test data sets do not necessarily perform equally on "live web" data, and vice versa. Therefore, such test results need to be reviewed very carefully and evaluated depending on the actual situation. Such test results never tell the whole story.

The exact definition of precision and recall in the literature is varying. Shafi et al. define precision as *"the fraction of a search output that is relevant for a particular query"*, and recall as the *"ability of a retrieval system to obtain all or most of the relevant documents in the collection."* [41] They can be calculated as fractions based on the amount of *true positive* (TP), *true negative* (TN) and *false negative* (FN) records in a query result: [14]

$$p = \frac{TP}{TP + FP} \qquad r = \frac{TP}{TP + FN} \tag{1}$$

A *true positive* answer is one valid record for a query that has been returned by the text database. A *false positive* is a record that has been returned for a query, but is not a valid response. A *false negative* is a valid response to a query that is not included in the query result. There also are *true negatives*, documents that are not a valid response to the query and have not been returned.

This definition differs slightly to the one used by Kobayashi et al. [23] who use a different calculation:

$$p = \frac{relevant\ documents}{retrieved\ documents} \qquad r = \frac{relevant,\ retrieved\ documents}{relevant\ documents} \tag{2}$$

There are also probabilistic definitions for precision and recall. Precision can be seen as the probability that a document is valid, given that it is returned in a query response, whereas recall is the probability that a relevant record is being returned: [14]

[10] TREC: Text REtrieval Conference, `http://trec.nist.gov`

[11] CLEF: Cross Language Evaluation Forum, `http://www.clef-campaign.org`

$$p = P(relevant \mid returned) \quad r = P(returned \mid relevant) \tag{3}$$

There is a conflict between precision and recall, and the need to probably remove large parts of a query result to avoid information overflow for the user. [25] Also, Kobayashi et al. identified a conflict between precision, recall and the system performance in terms of speed: good precision and recall values are difficult to achieve in real time. [23] The goal of result ranking is to list only the most relevant documents, and exclude irrelevant responses. This of course influences the system performance as measured by precision and recall: if too many responses are determined as being "irrelevant", precision and recall values will decrease. The critical point is how to specify a good threshold that can meed optimal results, [43] and to find a good trade off between precision, recall and a satisfied user.

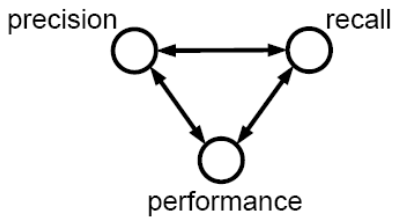


Figure 2: Trade off precision - recall - performance

2.4 Processing Large Amounts of Data

One obvious result of processing large amounts of data is the amount of time that required to do it. This goes for all data sets, but processing cost requirements depend on the application. For example, the streaming of media files has no relevant processing costs attached to locating and retrieving the actual file, it just needs to be fetched via the file system. The subsequent processing itself has linear costs, which is acceptable for this type of application. While there are possibly very large amounts of streamed data, the streaming itself is not time critical.

For text databases, the opposite is true. The total collection of accumulated text data needs to be processed as a whole at least once, during index creation. Subsequent queries in the database usually are time critical, thus need to be performed in sub-linear time, especially for large data bases.

With growing index sizes, processing of the complete data base on one single host computer will become impossible at one point, at which the index needs to be splitted and processed in smaller parts, on a number of nodes in a

distributed database computing system. Planning ahead for this event while creating a large scale document retrieval system is imperative to provide scalability.

Other considerations for processing of large data amounts on multiple computer systems is the occasional occurrence of transient hardware or software failures. For example, manufacturers of hard disk drives or memory components specify a failure probability for reading or writing operations on the medium. Usually, this probability is small enough to be neglected in conventional applications, but when operating a large amount of computers, and when performing read and write operations on a large scale on every single one of them, these failures will occur often enough to present a real problem for the application. Thus, transient hardware failures have to be expected and possibly corrected on the application level.

This leads to another requirement for applications operating on large amounts of data: fault tolerance, error detection and recovery – which is notoriously hard to implement.

Usually it is not uncommon in distributed systems to move the data to the best available CPU for a computational task. Especially in grid computing systems this is a widely used method. However, when doing something as data intensive as creating a full text index, there will be no nodes in the system that have to process significantly less data than others, so this is obviously not a viable solution. Therefore, each node in the distributed database cluster needs to be equipped with sufficient hardware to process that amount of data. In order to allow the document retrieval system to scale up with the ever increasing amount of indexed text, either the data amount per node has to be reduced to scale with the available hardware (resulting in more nodes on the same database system[12]), or the hardware has to scale up with the data amount (which can only be achieved by upgrading the hardware of all nodes in the database). Usually, the latter choice is the more expensive one, and the former the one that is harder to implement, because the effective total number of nodes might be growing as the database size increases during the operational use.

[12]Googles full text index for example runs on a very large number of standard PC hardware. It is more cost effective to operate a large number of cheap computers than using a few high end servers.

3 The Inverted Index

An *inverted index* is composed from two main components: a word list (the *dictionary*), and an *inverted file*. Each distinct word in the text collection is represented by one record in the *dictionary*, which also contains a pointer into the *inverted file*. Every record in the *inverted file* contains a series of references to all the documents in which this word occurred. These references are used to perform the full text search. As a content based approach, the *inverted index* is motivated by semantics-based search schemes that represent documents as term vectors.

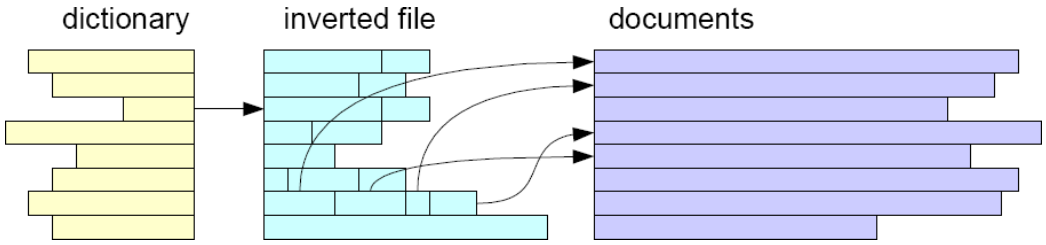


Figure 3: Structure of the inverted index

The name *"inverted index"* stems from the inverted order of the index file: when a document is a *"forward list"*, with each document containing a vector of words, the *"inverted list"* is a single word that points to a vector of documents.

By splitting the index into two parts it becomes possible to maintain the dictionary in memory. Usually it is small enough to be kept in memory, even for large text collections.

This data structure is a very common full text indexing principle, which is being used in almost all working full text database systems. [38] In the literature, it is often referred to as *"inverted list"* or *"inverted file"*. Here, the term *"inverted index"* will be used to describe the indexing structure in its entity, and the terms *"inverted file"* or *"inverted list"* are used to refer to the list of references into the documents (as described in figure 3).

The basic idea of this index is so intuitive that it has in fact been re-invented many times, and is described in different program documentations, without explicitly calling it by the name of *"inverted index"*. For example, parts of the *MySQL* documentation refer to MySQLs full text index as something that *"is much like other indexes: a sorted list of keys with point to records in the data file."* [16] While the inverted index implementation used by MySQL differs only slightly from the classical inverted index definition (it is not using a separated integer dictionary), the terms *"inverted index"*,

"inverted list" or *"inverted file"* are completely absent from the MySQL documentation[13].

Its popularity has made the inverted index a subject of research for many experts, [52, 40, 44, 7] who are continuing the development in order to optimize it for different requirements, add new features and illustrate the different basic variations of the algorithm. Some of the proposals are mentioned below.

While the inverted index has a few shortcomings and difficulties, it is the index form that is best suited for an internet search engine: It performs well on records with variable length, scales up in size and can scale further by using distribution techniques. It also offers many possibility to calculate parameters and statistical values for later result ranking on a per-word basis, if needed. None of the other indexing algorithms presented above can offer the same functionality while maintaining reasonable index size, acceptable indexing costs and sub-linear query costs.

The intuitive and simplistic design makes the basic principles relatively easy to understand, as opposed to other indexing algorithms. This is a big help for implementation and deployment of the algorithm, in finding possible optimizations and performance bottlenecks, as well as in adding new features. However, the actual implementation of an inverted index can become very complex, dependent on the optimizations and extra features. A simplistic indexing algorithm does not necessarily result in a simple database implementation.

Chapter 3.1 explains the indexing process, the components of the index structure and the data contained therein. Chapter 3.1.4 illustrates the search process and possibilities for result ranking, chapter 3.2 lists some of the main variations that exist for the inverted index. Common problems and pitfalls are discussed in chapter 3.3, and the concluding chapter 3.4 lists a few inverted index implementations in software that is widely used.

3.1 Index Creation

The indexing process can be divided into the following steps:

1. Splitting text into words
2. Translation of words to word IDs
3. Assigning an unique ID to each document
4. Sorting

[13] As determined by a full text site search on the MySQL homepage

5. Query evaluation

6. Result ranking

3.1.1 Splitting Text into Words

The first step in text indexing always is the parsing of the input data. If the raw data is not in plain text format but contains formatting markups (such as HTML or PDF), this meta-data needs to be removed from the text source[14]. This could already present the first problem, as the exact input format might be unknown, or the decoded text could contain a lot of additional "noise" (ref. chapter 3.3).

After that, the text needs to be split into a list of separated words. Identifying words and word boundaries might look trivial at a first glance, but it is in fact a challenging and fundamental problem: Word boundaries are not always obvious (especially in non-latin language encodings, for example in japanese or chinese text) and the definition of one *"word"* greatly affects the later functionality and performance of the whole system. If the definition is too strict, many legal words (or better: substrings that might be interesting for searches) will be ignored and not be available for search queries, resulting in a loss of functionality. If the definition is too wide, too many words will be added to the dictionary and the overall index size will increase, resulting in a loss of performance.

In this step, the size of the database can already be reduced by ignoring the most common words (*"stop words"*), or by transposing words into word stems – techniques used by many implementations to keep the index size to a minimum.

3.1.2 Translation

Operations on textual data can be optimized a lot by transposing the words into something more comfortable for computer systems: integer numbers.

In classical inverted index implementations, each distinct word is mapped to its own unique integer identifier – the *"word ID"*. This assignment has to be global and deterministic, meaning that the mapping between words and word IDs has to be the same for all occurrences, and that it must never change.

As shown in Figure 4, the dictionary is created during the translation process (the translation is the only step that writes to the dictionary). The

[14] The system might want to exploit formatting information for advanced result ranking. Google for example weights headlines and otherwise highlighted text differently.

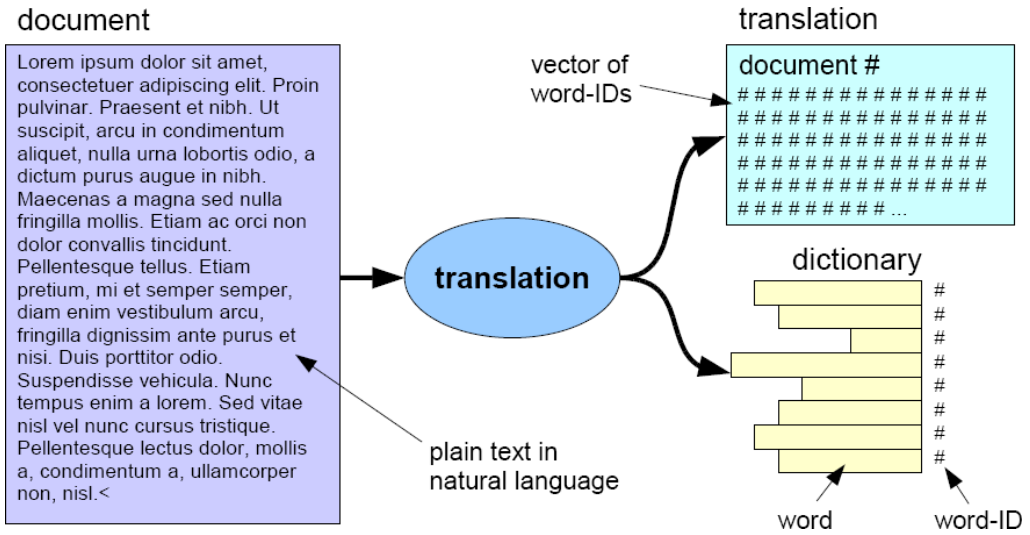


Figure 4: Translation of a document into a vector of word IDs. Only during this process there are additions to the dictionary

dictionary is a list of the global vocabulary and contains the mappings from words to word IDs. Each time when a new word appears that was never seen before, it needs to be added to the dictionary. This will of course happen very often in the beginning of the indexing process, but it also has to be expected occasionally even after very large amounts of text have been indexed. [47] Thus, the dictionary needs to be stored using a data structure that allows easy additions at random positions. As we will see later, the data structure also needs to provide little reading costs for faster translation and searching. For performance reasons, it is wise to hold the total dictionary in memory. As it will contain many entries, the storage costs for one single entry in the dictionary need to be minimal.

For later performance optimizations, the document itself should also be assigned an integer identifier – the *"document ID"*. The result of the translation step is the *"translated document"* (sometimes referred to as "forward index" in the literature): a vector of word IDs, ready for further processing, without the need to perform any more string operations.

3.1.3 Sorting

Sorting is the most expensive phase during the creation of an inverted index. In this step, the *inverted file* is created. For each word in the translated documents, a reference to the document is added in the inverted list, effectively resorting the *translation* into the *inverted file*. This process often requires up-

dates to many entries in the inverted list: the more distinct words are present in the document, the more index entries need to be created or changed. This happens especially when very long texts are added to the index.

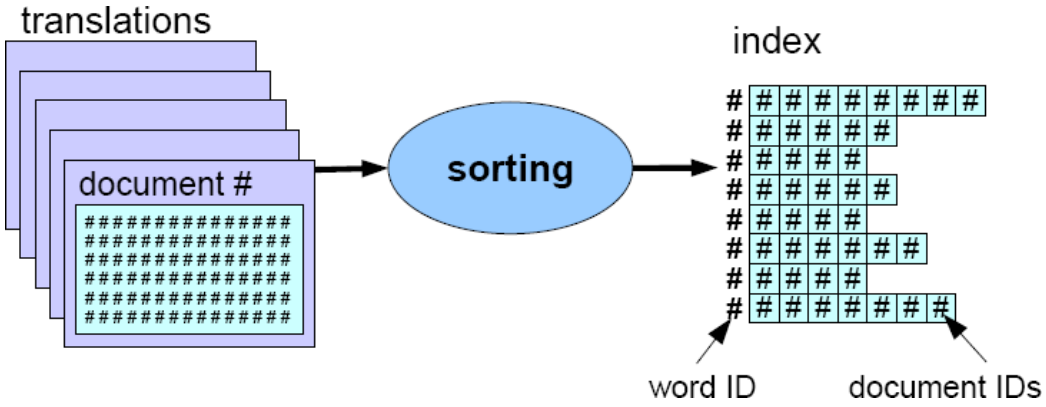


Figure 5: Sorting of translated documents into the inverted index

As the inverted index grows proportionally to the amount of text, it is impossible to hold it in memory. Therefore, each alteration or creation of a list entry requires one disk access. For optimization, the most frequently used entries in the inverted list should be cached in memory, to avoid a bottleneck of too many disk operations. If *stop words* are added to the index, the most frequently used inverted list entries will be stop words. Also, being the most frequent words in the text collection, inverted list entries for stop words will become very long, and thus eventually impossible to hold in memory.

Another problem is the increasing growth of the inverted lists during indexing. Assuming that the lists are stored within blocks in data files, the constant growth can lead to data file fragmentation: when a record grows beyond the size of its assigned data block in the file, it either needs to be split, or it needs to be relocated into a larger block. The first option will lead to data file fragmentation, resulting in many seek operations when the inverted list needs to be loaded. Usually this is a major cause of performance bottlenecks during indexing. The second option will lead to many unused blocks in the data file, and thus to a waste of disk space.

3.1.4 Query Evaluation and Result Ranking

Queries are evaluated by first translating the query terms into the respective word IDs, using the dictionary. If a search term is not in the dictionary, an empty response set is returned – the missing term is not added to the dictionary.

For each word ID, the inverted lists are fetched from the index, and merged or intersected as needed: the user might perform a boolean query and specify a list of terms that have to be present, and an other list of terms that must not be in all results (for example a query like "cats dogs -horses", returning all documents that contain "cats" and "dogs", but do not contain "horses"). If the inverted index is a record level index (see chapter 3.2), it is also possible to perform phrase queries[15].

Despite the fact that text-based retrieval is the primary method for identifying the relevant records for a search query, it is required to sort the resulting documents according to their relevance. Especially the (many) query results in very large full text databases would be useless without a relevance sorting function, because otherwise the user would have to scan hundreds of documents manually.

Such relevance functions try to estimate a documents relevance by applying different statistical methods, based on numbers that need to be stored in the index. Depending on the required granularity of the ranking and the type of ranking values, it can be difficult to store them and keep them up to date. As the text collection is being changed, they might need to be updated. [31] Ranking functions and their required input values are evaluated in more detail in chapter 4.

3.2 Variants

Besides the basic structure described above, there are some possible enhancements to the inverted index. Usually, they come at the cost of increased storage space requirements. Dependent on the course of implementation and the actually needed features, additional functionality can be provided by adding more data to the inverted file.

The most basic distinction can be made based on the type and amount of data present in the inverted list itself.

3.2.1 Record Level vs. Word Level Index

The system as described above provides the functionality of a *record level index*, that is, a index that is able to return all records for any given word, but does not tell anything about the exact position of the word in a document. This is the most common query type for document retrieval systems, but a record level index does not support phrase queries. Record level inverted

[15] Search queries with multiple different words, where the relative word positions in the query have to match the relative positions in the document

indexes provide functionality that is comparable to *signature files* [52], but offer the additional possibility of storing ranking information in the index.

An index that also stores the word positions is called a *word level index*. The additional information results in a considerable growth of index size, thus it is not as popular as record level indexes. With compression, a record level index can shrink to as little as 10% of the size of the source text, whereas a word level index with compression uses 25%. Also, the amount of memory required for sorting can grow to a non-trivial amount. [52]

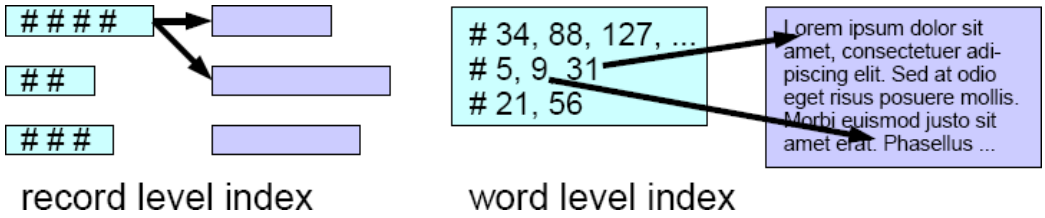


Figure 6: Record vs. Word Level Index

3.2.2 Archival of Originals

In modern systems, disk storage space is relatively cheap and therefore it is often feasible to store the original documents along with the index, for online text retrieval. The advantage is having the documents readily available upon request, instead of pointing the user to an offline source. Text in a natural language can be compressed very well, and storage requirements for the compressed source text and the record level index thereof have been reported to be 40% of the original text size. [51, 52]

Especially for an internet search engine, this is an interesting feature – given that a web site can go offline any moment or might be unavailable to the user who performs a query. Also, this offers the functionality to present short text excerpts along with the query results, short summaries of the text passages that contain the query terms.

3.2.3 Distributed Indexing

In order to scale with the constant growth of a text database, it is necessary to provide the functionality of a distributed index. With a constantly growing text collection (as present in a web search engine), it sooner or later will be impossible to process all data on one single node. Also, in order to maintain operability in case of a node failure, a redundant system is required.

To distribute an index to multiple nodes, it needs to be partitioned. There are two basic strategies to partition an inverted index: partition of the *document collection* (creating multiple *local inverted indexes*) or partition based on the *index terms* (resulting in *global inverted indexes*).

Performance studies indicated that local inverted indexes are very effective and can provide good query throughput [31], while global inverted indexes have the disadvantage of lacking fault tolerance: if the node serving the queries for a certain word ID is not available, it is impossible to perform the query and an empty result has to be returned. If a node serving one of many local indexes is unavailable, the query will possibly not contain all matching records, but it can be performed, which increases the total availability of the system.

The Google index uses both techniques. The global full text index is partitioned by document collections, and each partition is further divided by query terms, each group of terms being served by multiple nodes. This ensures availability if a few nodes are down, and provides very good query throughput on a global scale.

3.2.4 Hot Updates

The term *"hot updating"* refers to the ability to modify the index at query time. [31] This feature requires very good concurrency control mechanisms, because the index file must not be modified while it is being read to answer a query. One approach is to lock certain parts of the index for updating, while other parts can still be used for query answering. This is an obvious solution for distributed indexes, but can be tricky to perform on one single node. The second possibility is to lock the complete index on one node for reading, then perform the update and unlock the index on that node again. Especially during peak loads, such updates on a "hot" index can introduce bottlenecks and high query response times, which is of course not desirable. A possible solution is to schedule updates in batches off peak load times.

3.3 Problems and Difficulties

The very first and most fundamental problem when creating a full text search engine for web sites starts with the exact *definition of text*.

Especially with HTML documents from web portals, the problem is that the raw text contains too much data. A lot of *"non-content"* is being added to full text search engines, because it is impossible to identify unnecessary web site components, like advertisement or navigation links. While there has

been progress with identifying such components [49], this technique is not yet generally available.

The second problem is textual content on web sites that is invisible to the user, like meta information in HTML headers or embedded image descriptors. This information was originally intended to help machines to digest essential information automatically, but is nowadays often abused to spam web search engines. Therefore, it is questionable if content, that is invisible to a human reader, should be added to the full text index of a web search engine. However, identifying invisible text is not a trivial task. For example, the effort to recognize white text on a white background image as invisible would include the usage of a HTML renderer and pattern recognition techniques – not something one wants to do in a performance critical operation.

On the other side, one might encounter textual information where it is unexpected, like in file names, HTML hyper links or in proprietary binary data files. Identifying and parsing textual information from binary sources requires extensive knowledge about proprietary file formats, which often is not publicly available.

When a document changes and needs to be updated in the index, it is in most cases necessary to first remove the document from the index, and then add the updated version. Removing one complete document from the inverted list either requires a backup copy of the original document translation, or a very expensive[16] traversal of the complete inverted list. All enttries in the inverted list that contain pointers to the removed document need to be deleted. Locating those entries is easier if the original document translation can be used to find all words in the document.

An inverted index has some inherent limitations of functionality, as a direct result from the way it is constructed: with the smallest possible data unit inside the database being one single word, it is impossible to search for word substrings, or to include non-words (like a string of interpunctation symbols) in a query.

Closely related to the problem of defining *"text"* is the exact definition of a *"word"*. This definition has a huge impact on the whole system: with the single word being the smallest unit in an inverted index, finding a good definition for *"word"* becomes a key issue. The functionality of the overall search engine, the size of the index file, the size of the dictionary – and thus the performance of the whole system – depend on this definition. [47] If the definition is too strict, many words that could be interesting for queries will not match it and will therefore not be added to the dictionary, which makes it impossible to search for them. If the definition is too loose, many

[16] Index traversal is expensive because it usually is very large

unnecessary words will be added to the dictionary, which might become too large to be held in memory. In addition to that, the index file itself will grow very fast.

Independent from the actual definition, the amount of distinct words in the index grows proportionally to the square root of the total amount of text in the database. [31] One billion english documents can contain roughly 310 million distinct words, [47] so finding data structures and algorithms that can handle that amount of data on the available hardware is, especially in the long run, very difficult.

In documents that have been acquired from web sites, especially in HTML files, it has to be expected that a considerable part of strings do not form a valid word. For example, it would be pointless to add a complete URL into the vocabulary. On the other hand, some uncommon words that do not appear in any english dictionary might be interesting for the user. Nobody will search for `2005/TECH/28/cns_itrnt_ap.html`, but query terms like XEROX, KLEENEX or HAL9000 are not so far-fetched.

Many implementations are using *stop word lists* to reduce the index size. However, such lists are dependent on the language in which the text is written, and reduce the functionality of the resulting index. Many english phrases contain stop words like *"I"*, *"am"* and *"the"*. So, a fictional query for *"I am the king of pop"* would effectively result in a query for *"king pop"*, which is not quite the same. Even worse, some systems reject words that are less than four characters in length (like MySQL), rendering the query into *"king"* – and probably completely useless.

A more advanced but also more complicated approach is to use word stemming before adding words to the dictionary. [47] This requires word stemming algorithms that are, just like stop word lists, dependent on the actual language, and might produce incorrect results. Finding good stemming algorithms is not trivial, especially if they should take care for all the different grammatical anomalies, like irregular verb declinations. When operating in an international environment (like the internet), both word stemming and stop word lists would require some heuristic algorithm to determine the language of a document before they can be applied. If a internet search engine wants to apply correct word stemming and stop word lists, the efforts to do so must not be under-estimated.

The next problem is the index size: the less data there is in the index, the better is the system performance. As mentioned above, there are several approaches to reduce the text size before adding it to the full text index. The number of words could be reduced even further by checking the words against a dictionary and use heuristics to determine if a new word is valid or not. But irregardless of the steps taken to keep the index size at a manageable

level, it will be constantly growing. As already explained in chapter 3.1.3, the constant growth of the inverted lists result in data fragmentation or disk space wastage. However, this problem can be overcome during runtime, by running defragmentation routines at periodic intervals.

3.4 Existing Implementations

Most document retrieval systems use the inverted index, or a variant thereof. While there do exist systems based on other indexes[17], they do not fulfill the specific requirements for an operational internet search engine.

The inverted index is the algorithm of choice for document retrieval systems and has been shown to be superior to all other index forms, when evaluated in the aspect of search engine requirements. [51, 31]

The following sections contain a short description of two well known systems that implement full text search functionality based on an inverted index: *Google*, as a representation for document retrieval systems, and *MySQL*, a relational database with full text search capabilities.

3.4.1 Google

Google currently is the largest internet search engine available, serving more than 200 million search queries a day. In June 2005, it contained an estimate of 8 billion web pages[18]. The name *Google* was chosen because it is a common spelling of the number *"googol"*, or 10^{100}.

The Google index exploits the structures found on web pages and in HTML to a large degree, evaluating hyper links in web pages, the text of those hyper links and other HTML markup on web sites. The text of links for example (the *anchor text*), is threated as content of the document being referred to, and as content of the document in the link appears. The effect is that web sites that contain no textual information at all (front pages with only a logo for example) can be found by searching for text that was placed in links to that page.

Additionally, the system evaluates HTML markup of text, and weights document contents differently. Terms in big or bold fonts for example are

[17] Most systems that are not based on the inverted index are using a signature file index. [51] For example, ATLAS as presented in [39]

[18] According to a presentation of Monika Henzinger (Research Director at Google Inc.) at the Technical University Vienna, in June 2005. However, in September 2005 Google stopped to publish the number of indexed documents, because *"metrics for index size measurement vary greatly and are no longer easily comparable"*. (`http://www.google.com/help/indexsize.html`)

evaluated as being more important than others, that are written in smaller fonts.

The main component of the Google system however is the *PageRank* algorithm, [6] introduced by S. Brin and L. Page, and illustrated in more detail in chapter 4.5.1. The algorithm tries to find an *"objective measure of citation importance of a web site that corresponds well with peoples subjective idea of importance"*. [6] According to Googles own spokesmen, *PageRank* has been a main reason for the success of the search engine, as it is producing satisfactory ranking results.

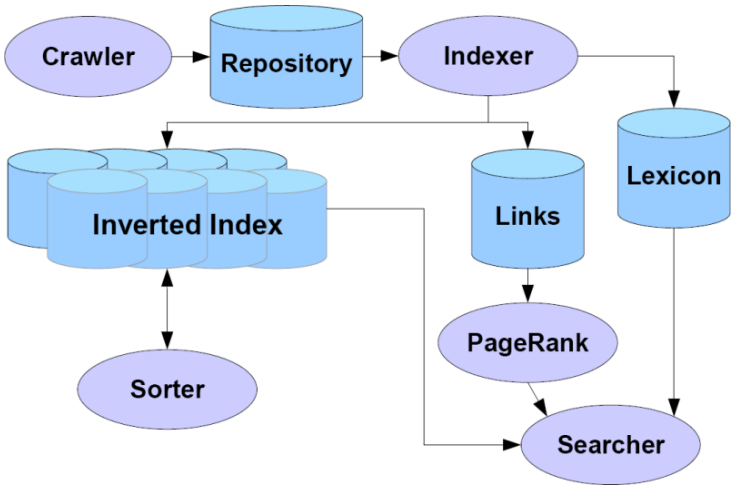


Figure 7: Simplified data flowchart in the Google full text index. This graph is reduced to the components essential to data flow

As shown in figure 7, the main components of the Google indexing system are the *crawler*, *indexer*, *sorter* and *searcher*. There are some additional subsystems to manage the process of web crawling and document indexing, but they are not essential to the basic data processing itself, and therefore not shown in the figure. A more detailed diagram of the Google indexing process can be found in [6].

The crawler retrieves web sites and documents from the internet and stores them in compressed raw form into the repository. From there, they are fetched by the indexer, which parses the documents (most of them are HTML), converts the document content into a vector of *hits* (containing word identifier, position and weight for each word) and stores these *hits* in a number of *barrels*, in the form of a *forward index* (as opposed to *inverted index*). In this process, the indexer also creates the *lexicon* (a synonym for the *dictionary*), and identifies hyper links contained in web pages. Links are stored to a own repository for later *PageRank* calculation.

The creation of the inverted index itself is done by the *sorter*: it takes the translated documents from the *barrels*, and resorts them by word IDs. The actual searching is done by the *searcher*, which is being run on a web server to present a user interface to the end user. It uses the *lexicon*, the *inverted index* and the calculated *PageRank* to answer search queries and do the result ranking.

3.4.2 MySQL

MySQL is a relational database and optimized for fast transactions of SQL statements. As such, it usually operations on highly structured data tables. It is not a full text indexing system, but offers this functionality for tables that contain textual data. As a result, the full text search capabilities of MySQL perform worse than specialized full text indexes. Especially for large amounts of textual data, MySQL is not capable of maintaining an acceptable performance during indexing and searching, and is therefore not suitable to serve as the core index for larger text search engines.

This is mainly related to the relational nature of the database, which requires it to store each document in a separate record – even when using the full text index. However, the index implementation itself is closely related to the inverted index, and worth looking at. Technically speaking, it is not a classical inverted index, because it lacks the dictionary and the translation of words into word IDs. But besides that, all other aspects of the implementation are identical.

MySQL treats anything that is alphabetic or numeric as part of a word, (almost) everything else is used as a separator. The word parser is quite naive: one *"word"* is a string of alphabetical or numerical characters, optionally separated by no more than one sequential apostrophe. Words that do not exceed a minimum length (default is four characters) are being ignored, as well as words in the stop word list. Furthermore, words that are present in at least 50% of the documents are automatically classified as *stop words*. [16]

Words in the text collection are weighted according to their estimated significance in the collection. This way, common words that appear many times, and in many documents, have a lower weight, because the estimated semantic value in the text collection is little. Rare words receive a high weight. The weights of the words in the text and query are then combined to compute a relevance factor for one record. The records returned are automatically sorted with the highest relevance first. [1]

The full-text index itself consists of a sorted list of keys that point to records in the database, as shown in table 1.

The basic concept of this index structure is very similar to the inverted

Word	The word itself (string)
Count	Number of occurrences of this word
Weight	Evaluation of the word importance
Rowid	Pointer to the records in the table

Table 1: MySQL full text key structure, as used in MySQL 4.1

index. The difference is only that the string representation of word itself is being used instead of an integer word ID. This can be explained by the different requirements on MySQL: being a light weight relational database, it cannot afford to load excessive amounts of table indexes into memory. The way of relevance calculation is basically the same as presented in the *TFxIDF* algorithm (chapter 4.2). The whole indexing process itself is based on distinct words, represents a content based indexing approach and a *word level* inverted index structure.

3.4.3 Other Variations

There are many scientifically developed and explored variations of the inverted index, most of them deal with different kind of optimizations. Some were implemented for performance analysis only [51], others as a demonstration of a concept [8, 31] or as a business application [6].

Most variations deal with performance optimization by applying different techniques for index and data compression. The idea is that the extra computational overhead that is necessary to compress and decompress data is remedied by the time gained when reading and writing the compressed data to the hard disk. Disk operations are the bottleneck for inverted index creation, and everything that reduces the data amount that needs to be read and written on disk increases the overall performance, even when it comes at a computational overhead. This compression can already start with the representation of natural text [12], but more often is done at index level. [40, 6]

The focus is also on scalable database implementations, which usually means building a distributed database system [31] or data distribution in general.

A different but interesting aspect of the development of the inverted index is the "historical" time line. It can roughly be divided into pre-Google and post-Google. Google was first published in 1996 as a research project at the Stanford University. At this time, it was not very popular, but that changed in 1998 when *Google, Inc.* was founded. The development of full text search engines until then focused on the basic techniques, how to create

the index and how to store data. The research was done on a broad band of topics, including ranking algorithms [50], index algorithms, and fundamental problems when dealing with natural language text [25] and the creation of "robot-based" search engines, as opposed to manually built internet indexes.

After the overwhelming success of the Google search engine, which is based on an inverted index, the focus changed a bit. Around the same time, Zobel et al. published a very sophisticated paper, comparing the inverted file index to signature files [51], with the conclusion that inverted indexes are, in almost all aspects, technically superior to signature files. After this time, publications on signature file indexes seem to be sparse, and the focus switched to different semantic evaluation schemes of natural language text [4] – which has been a very popular topic until today.

4 Result Ranking

Ranking is the process of sorting documents in a query result according to their relevance, which is being calculated by a *"relevance function"*. This function is not used to determine if a document is a match for a search query or not, that is done by the query evaluation in the indexing algorithm.

Creating a full text index is only the first step towards an usable search engine. [45] When document retrieval systems perform a query, they first do a boolean evaluation of the search terms and the contents of the full text index. This evaluation yields all candidate documents (that is, documents that match the search criteria), which, especially for large database systems, can be a very large set of responses. Processing these response manually is not feasible because it is too large. The result list needs to be re-ordered in such a way that only the most relevant documents are shown at the beginning of the result list. Studies showed that most users do not look at more than the first two pages of search results on internet search engines. [45]

In document retrieval systems, most ranking functions try to determine the semantic relevance of all returned documents to the query. For large response sets, this can be a very costly operation, but it is possible to increase the ranking speed by using approximation. [11]

As mentioned in section 1.2 and elaborated in [25], the semantic content or meaning of documents in any natural language is not explicitly available to a computer system, thus the semantic relevance needs to be determined by using statistical functions. Most document retrieval systems use a combination of multiple relevance functions, but details about the used ranking algorithms and the exact combination of their results usually is a well kept secret. This information is considered to be valuable and can be exploited by search engine spammers. Detailed descriptions of ranking algorithms are notoriously hard to find.

To perform a relevance evaluation of a document compared to a search query, there are two possible basic methodologies. We will refer to *dynamic relevance functions* when the result of the function for a single document depends on the documents content as it is being compared to the particular query (like in the *TFxIDF* algorithm), and *static relevance functions* if the ranking result is global and does not depend on the query itself (like *PageRank*). Usually, dynamic ranking functions require a *word level index* (see 3.2.1).

Static relevance functions can have a big advantage of computational costs over dynamic functions, because they need to be calculated only once, at indexing time. They do, however, not have the fine granularity of dynamic relevance functions, and should not be used as the only ranking criterion.

Also, their calculation can be expensive due to the large amount of documents in the collection.

The overall quality of a document retrieval system depends basically on two factors: the completeness of the text collection, and the quality of the query results. As elaborated in chapter 1.1, an internet search engine can never be truly complete. Due to the unknown size of the internet and the fast rate of change on the web[19] it is impossible to get a complete copy of the web. Even if it would be possible, it would become obsolete very quickly. The only possibility is to constantly add and update as many web sites as possible in the collection, and hope that the data base is large enough to provide satisfactory results to the users.

This leaves only the quality of the query results as a real opportunity for improvements. As shown above, the answer sets for any query need to be evaluated and resorted before they can be presented to the user. This evaluation can only be done on records that were already identified as valid matches by the indexing structure, it is impossible to add additional documents based on this evaluation. However, it is possible to determine matches as unsuitable and therefore exclude valid (but irrelevant) results from the answer.

4.1 The Vector Space Model

Statistical ranking functions need input data on which they can base their calculations. Such values can be stored in the full text index directly, but this increases the overall index size, and those values might be hard to calculate and to maintained up to date.

Basic ranking techniques do not require much more than the number of documents that contain each word to determine the relevance, but to improve the performance, more information is needed. For example, if the frequency of each term in the global text collection and in each document is known, the terms semantic value can be estimated: rare words are more important than common words. However, these parameters might affect index updating costs. If a new document is added, the relative frequency of word occurrence may change, requiring an update of the total index. [52]

The *vector space model* is a useful tool to effectively assign such values to documents and terms. In the general vector space model, a text is represented as a vector of attributes. Usually, some form of the *term-weighting* vector space model is used in search engines: the attributes in the vector are the

[19] Some studies assume that an estimate of 40% of the web changes at least once per month. [23]

words in the text, the position in the text defines the position of the attribute in the vector. [23]

Each word is assigned a weight that defines the semantic significance of that word, which is relatively simple to calculate if the global word frequency is known. [45]

Note that this evaluation is based on the number of global occurrences of the search term in the total text collection. Obviously, this number changes with each update to the index, thus it would be futile to store it in the index structure: it would require a change of the complete index for each update.

$$W_i = \frac{O_i}{G_i} \tag{4}$$

Where

W_i = semantic weight of the i-th term in a vector
O_i = number of occurrences of this term in a vector
G_i = number of global occurrences of this term in the collection

Uncommon terms, like names, will receive a higher score than common terms because they appear much less frequently in the text collection, and they are not overruled by common terms that appear frequently in one document. [45]

However, herein lies a major weakness: Documents that contain many rare terms, maybe as hidden text on a web page so that they are invisible to the user, will always achieve a high semantic evaluation and thus a high ranking – a fact that is being exploited by web search engine spammers. Unfortunately, they are mostly successful, because many search engines are using a variation of the term-weighting vector space model. [23]

4.1.1 Latent Semantic Indexing

Latent semantic indexing (LSI) is *"one of the more widely used vector space model-based algorithms"* [23] and one of the few that consider *synonymy* and *polysemy*. [23] The observation is that often terms are used as *synonyms* for the same idea, like "aircraft", "aeroplane" or "plane" all represent a flying machine with fixed wings. At the same time, the term "plane" is *polysemic* because it refers to multiple, unrelated things: a flying machine, a flat surface or a planing tool used by carpenters.

LSI ranks texts by semantic content, but it does more than the usual boolean matching. It discovers synonyms and polysemic terms by using statistical evaluations over terms that often appear together in the same text. [45]

4.2 TFxIDF

Text Frequency times Inverse Document Frequency is a relevance function used in basically all existing inverted index systems, sometimes with varying implementation details. Like the other ranking algorithms presented here, TFxIDF is a dynamic relevance function that tries to determine the document relevance to a query by calculating a distance value between retrieved documents queries using the vector space model.

Search queries are mapped onto a pseudo-document [45] that contains only the query terms, as formulated by the user. Boolean queries that use logical operators for advanced searches (like "*king AND pop NOT Jackson*") are split into multiple pseudo-documents, and the respective results are later merged or intersected as needed. All retrieved documents are then compared to the pseudo-document, typically by calculating a value that is proportional to the cosine of the angle between the document and query vectors in the multi-dimensional vector space [50], or by computing the dot product [23] (which effectively makes no difference). To compensate for the varying query and document lengths, vector-length normalization can be applied [50, 23] – but Yuwono et al. showed empirically that this normalization is expensive and, in fact, decreases precision and recall. [50]

The vector distance $S_{i,q}$ between a document P_i and a query vector q, with vector-length normalization, can be calculated according to the following equation. Table 2 contains explanations to the terms used in the equation: [50]

$$S_{i,q} = \frac{\sum_{term_j \in q} \left(0.5 + 0.5 \frac{TF_{i,j}}{TF_{max_i}}\right) IDF_j}{\sqrt{\sum_{term_j \in P_i} \left(0.5 + 0.5 \frac{TF_{i,j}}{TF_{max_i}}\right)^2 (IDF_j)^2}} \tag{5}$$

As mentioned, vector-length normalization can (and in fact, should) be omitted. This reduces the costs of the ranking calculation and at the same time increases the quality of the ranking results, as measured by precision and recall. By ignoring the vector lengths, the angle between document and query vectors becomes mathematically incorrect, but in the aspect of the gained performance, this is only a small price to pay. After all, the vector space model and distance calculations therein can only be seen as an estimate of semantic relevance, which is a subjective value anyway.

Without vector-length normalization, TFxIDF can be calculated according to the following equation. [50] Again, refer to table 2 for a legend of terms:

M	=	the number of words in the query
Q_j	=	the j-th query word in a query with M words
N	=	the number of documents in the database
P_i	=	the i-th document (or its ID number) in the database
$TF_{i,j}$	=	the term frequency of Q_j in P_i
$TF_{i,max}$	=	the maximum term frequency of a word in P_i
IDF_j	=	$log\left(N/\sum_{i=1}^{N} C_{i,j}\right)$
$C_{i,j}$	=	weight of Q_j in P_i: $\begin{cases} W_j \Leftrightarrow Q_j \in P_i \text{ see eq. (4)} \\ 0 \quad \text{otherwise} \end{cases}$

Table 2: Legend of terms used in ranking equations

$$S_{i,q} = \sum_{term_j \in q} \left(0.5 + 0.5 \frac{TF_{i,j}}{TF_{i,max}}\right) IDF_j \tag{6}$$

While being very prominent and providing very good results in environments operating on data sets of good quality (data bases with controlled content, like journal articles, papers and publications), TFxIDF can yield poor results on uncontrolled text collections, like web pages on the internet.

The smaller the difference between a query term and a document is, the higher is the result of the relevance computation returned by TFxIDF. The algorithm does not take into account the general quality of a document. The highest possible ranking will be achieved by documents that contain the exact query term, and only the query term – which can happen regularly when operating on very large sets of web documents. But this is of course not a desirable result. For example, evaluating the query "*I am the king of pop*" with a document that contains only "*MJJ: I am the king of pop*" will result in a very high relevance value, while the document itself only provides little semantic content and is likely to be irrelevant for the user.

4.3 Boolean Spread Activation

Boolean text retrieval algorithms are solely based on the presence or absence of query terms in documents, without applying different weights or frequencies to the terms. Ranking is done on the amount of query terms that are contained within a document, without looking at the exact term positions or other semantic context.

More formally, the relevance $R_{i,q}$ of a document P_i and the query vector q is calculated according to the following equation: [50]

$$R_{i,q} = \sum_{j=1}^{M} H_{i,j} \tag{7}$$

$H_{i,j}$ is 1 if the query word Q_j is contained in document P_i, 0 otherwise. This algorithm does not require a word-level index and therefore is a viable option for very light-weight text indexes. However, the relevance estimation is quite naive and favors long documents.

Yuwono et al. extend this evaluation in the *Boolean Spread Activation* algorithm by "propagating the occurrence of a query word in a document to its neighboring documents". [50] Documents are neighbors if they contain hyper links to each other: document A is a neighbor of document B if B contains a link to A. The assumption is, that if documents link to each other, there is a semantic relationship – which is questionable, as we will see later.

4.4 Vector Spread Activation

Vector spread activation is a combination of TFxIDF and Boolean Spread Activation. First, the score of a document is calculated based on the vector space-model as used by TFxIDF, then this score is propagated to all documents it contains links to. Formally, the relevance assignment is calculated by the following equation. [50]

$$R_{i,q} = S_{i,q} + \alpha \cdot \sum_{j=1, j\neq i}^{N} Li_{i,j} \cdot S_{j,q} \tag{8}$$

$S_{i,q}$	=	the TFxIDF score of P_i respective to query q
α	=	a constant link weight $(0 < \alpha < 1)$
$Li_{i,j}$	=	the occurrence of an incoming hyper link from P_i to P_j, where $Li_{i,j} = 1$ if such a link exists, 0 otherwise

The idea of propagating site weights has been further developed by Brin and Page [6] in the *PageRank* algorithm as used by Google (chapter 4.5.1), but without using the semantic site content in the relevance calculation. Yuwono et al. concluded that the combination of link analysis in web pages and the evaluation of page content as done in the *vector spread activation* calculation achieved the best results in their experiments, and that it was more accurate than other ranking functions that exclusively exploited the link structure of the web. This is worth noting, because Googles tremendous success is always, to a large degree, being attributed to PageRank.

4.5 Most-cited

When confronted with the task of evaluating the content of web sites, one natural approach seems to be to exploit the hyper link structure of the web. This technique is called *link analysis* and has been proposed numerous times in the literature. [50, 6, 23, 28, 41] The presented boolean and vector spread algorithms combine link analysis with content evaluation, however there are algorithms that do not use content evaluation for result ranking.

The assumption is, that if the author of a web page sets to a different web site, there exists a semantic relationship. Also, with link analysis it is possible to exploit non-textual meta information that is present in websites, but cannot be evaluated by basic indexing algorithms.

The *most-cited* algorithm as presented by Yuwono et al. is a naive interpretation of the linked structure of web sites. The relevance score for one single page is the sum of the number of query words contained in other pages that link to it, or are being linked to: [50]

$$R_{i,q} = \sum_{k=1,k\neq i}^{N} \left[Li_{i,k} \cdot \sum_{j=1}^{M} H_{k,j} \right] \tag{9}$$

Where

$$H_{i,j} = \begin{cases} 1 \Leftrightarrow Q_j \in P_i \\ 0 \quad \text{otherwise} \end{cases}$$

$Li_{i,j}$ = the occurrence of an incoming hyper link from P_i to P_j, where $Li_{i,j} = 1$ if such a link exists, 0 otherwise

For the remaining terms refer to table 2.

4.5.1 PageRank

PageRank is a static ranking function implemented in the Google search engine. Like most-cited, it exploits the meta information given by the linked nature of the web, but unlike it, PageRank does not count all links equally. It determines the "global importance" – or better, the "prominence" – of a website by evaluating the links between them. However, the problem still stands that hyper links do not necessarily imply a semantic relation between sites [50] (see below).

PageRank can be explained as a Markov-chain, that simulates the behavior of a user browsing web pages. [28] It extends the idea of link-counting as presented in most-cited by *"not counting links from all pages equally, and by normalizing by the number of links on a page."* [6] The ranking value determines the likelihood for all sites that the user will end up visiting this site,

thus giving a higher rating to popular web sites that have many other sites placing hyper links to it.

PageRank is an iterative, recursive algorithm. With increasing amounts of web sites, the calculation can be very expensive. [28] Assuming that there are web pages P_1, ..., P_n with links pointing to web page A, the PageRank rating of A is calculated by the following recursive equation: [6]

$$Pr(A) = (1 - \delta) + \delta \cdot \sum_{j=1}^{n} \frac{Pr(P_j)}{C(P_j)} \tag{10}$$

Where

δ	=	damping factor, $0 < \delta < 1$
$Pr(P_x)$	=	PageRank rating of web page P_x
$C(P_x)$	=	number of links from web page P_x

There are concerns that the lack of content evaluation in the major ranking technique combined with Googles leading position on the market[20] could lead to a viscious circle: popular sites get a higher PageRank value in Google, and with Google as the current market leader this will lead to even higher awareness of the web page, resulting in even more people placing links to it, further pushing the PageRank rating. Ironically, Googles success could become a problem for PageRank, which is a major reason for the success to begin with.

> "Occasionally, when a particular website is the subject of public attention, other sites begin linking to it. This may elevate its importance as gauged by our ranking software. [...] Higher ranking in Google results may lead to more awareness, which may lead to more links and so on."
>
> "Anomalies occur from time to time. They show the weak spots in the result ranking and show how the used algorithms need to be improved in the future."
>
> Quoting the FAQ on `www.google.com`

As an interesting side note, Yuwono and Lee [50] explored some ranking algorithms in 1996, including the most-cited algorithm as described above. PageRank can be seen as a variation of most-cited, which was evaluated as an inferior ranking algorithm by Yuwono et al.:

[20] According to `searchenginewatch.com`, Google served 46.2% of all search queries in 2005, the rest was divided on over 10 other search engines. [23] includes a (slightly outdated) evaluation of search engines that shows the same trend.

> "The relatively superior retrieval effectiveness of TFxIDF and Vector Spread Activation search algorithms shows that the concentration or distribution of words in a WWW page and across WWW pages is a good indicator of the page's contents or portions thereof. Algorithms which rely on meta-information such as hyper links information, while intuitive, did not perform as well. This indicates that the interconnectivity between WWW pages does not always indicate semantic relationships between the contents of the linked pages."
>
> Budi Yuwono and Dik L. Lee,
> "Search and Ranking Algorithms for Locating Resources on the World Wide Web", 1996

The discrepancy between this conclusion and the reality as shown by Google could be explained by the size of the text collections. The word level evaluation of the semantic importance of one single word based on its frequency of occurrences works only well for large text collections [16], the same could hold true on a document level evaluation of hyper links. The 2393 WWW pages evaluated by Yuwono could have been too little to draw final conclusions. Also, the PageRank algorithm works a bit differently to most-cited: it does not weight all hyperlinks equally.

5 Palaver: Full Text Search

The implementation of this project was done in two stages: *Palaver*, and *Monkville*. The task during the first stage was to find a suitable way to implement text retrieval capabilities based on a basic database system, which was nicknamed *Palaver*[21].

Palaver is a basic document retrieval system using an inverted word level index and a customized storage engine for binary data, like the index. It has no sophisticated ranking functionality and does not implement index compression. Its sole purpose is to show how an inverted index can be used, how an implementation should be planned, and to point out bottlenecks and problems that appear only during the practical implementation – but might not be obvious at a first thought. However, it implements the full functionality of a document retrieval system, but it would require some additional work before it can be used in a productive environment.

In the second stage of the project, Monkville, this implementation has been enhanced with semantic full text search functionality. This extension will be discussed in chapter 6, the focus here will remain on the basic index implementation.

This chapter first gives a brief overview over the project history, then it describes system architecture and design principles, as well as data structures, data formats and algorithms used to solve different problems.

5.1 Project Overview

This project started as a replacement for an already existing, proprietary full text search database that has been in use at a medium sized[22] internet search engine. The existing solution showed a series of undesirable characteristics, such as bad indexing and search performance, non transparent and poor result ranking, and system unavailability during indexing and modifications to the index. This proprietary database had to be replaced.

The new implementation had to fulfill a series of requirements, in addition to the general search engine demands as listed in chapter 1.1.1. Due to the limited amount of available high-end hardware, the engine had to operate

[21] *Palaver* stems from the latin verb parabola. A palaver is a usually (very) lengthy discussion or conference without a particular topic, common in large parts of Africa. Its purpose is to become acquainted with somebody or to cultivate social contacts at meetings. Palavers are celebrated before the actual reason for a meeting is being discussed. The more important the attending personalities or the topic of the meeting is, the lengthier the palaver.

[22] At this time, the search engine indexed a rough estimate of 2.5 million web sites

on one server system only, providing search functionality even during index updates (thus, featuring *hot updating*, ref. chapter 3.2.4). Additionally, the solution had to be scalable. In order to accommodate the constantly growing amount of textual data in the index on a long-term time scale, the implementation needed to be done in a way that allowed easy addition of distributed indexing and searching. To accommodate future needs of additional features, it also needed to be designed and implemented in a flexible, transparent and maintainable way. This should allow easy additions of features in the future – a property that proved to be of very great importance during the second project stage.

The first implementation was a mere proof-of-concept, a collection of test programs that could not run in parallel. This was useful to determine if the inverted index could be implemented in a way that showed sufficient performance – as it turned out, it did. The basic duties of a document retrieval system (translating, sorting and searching) were done by separate programs each, that were run in sequence and operated on a series of binary data files. One of the main tasks for the Palaver implementation was to implement the same functionality in one monolithic program, that could run all tasks in parallel. It had to be implemented in such a way that the resulting source code was flexible and easily understandable, to facilitate further improvements.

5.2 System Architecture

The first architecture drafts were oriented on the Google implementation as shown in figure 7, and based on the same terminology and components as presented by S. Brin and L. Page [6]. However, this particular architecture appeared not to be optimal for the specific requirements. For example, many tasks in Google are done by separate programs that operate on a set of data files and create new ones, which seemed to be contrary to the idea of hot updates, as it is difficult to synchronize independent processes.

Furthermore, Google contains some components that were not desired for various reasons. The *PageRank* calculation for example was not included because of the following reason: due to the highly diverse linked nature of the web [23], *PageRank* requires a very large set of web pages to achieve reasonable results, more than the two million pages that were present in the full text index at the time. The indexed sites contained many links to web sites which were not in the database. The assumption was that the costs of implementing and calculating *PageRank* would not be justified by the quality of the result, due to the rather small size of the text collection.

The evaluation of the actual *PageRank* performance on a incomplete col-

lection of web sites would be an interesting topic, but is not within the scope of this project.

5.2.1 Architecture Overview

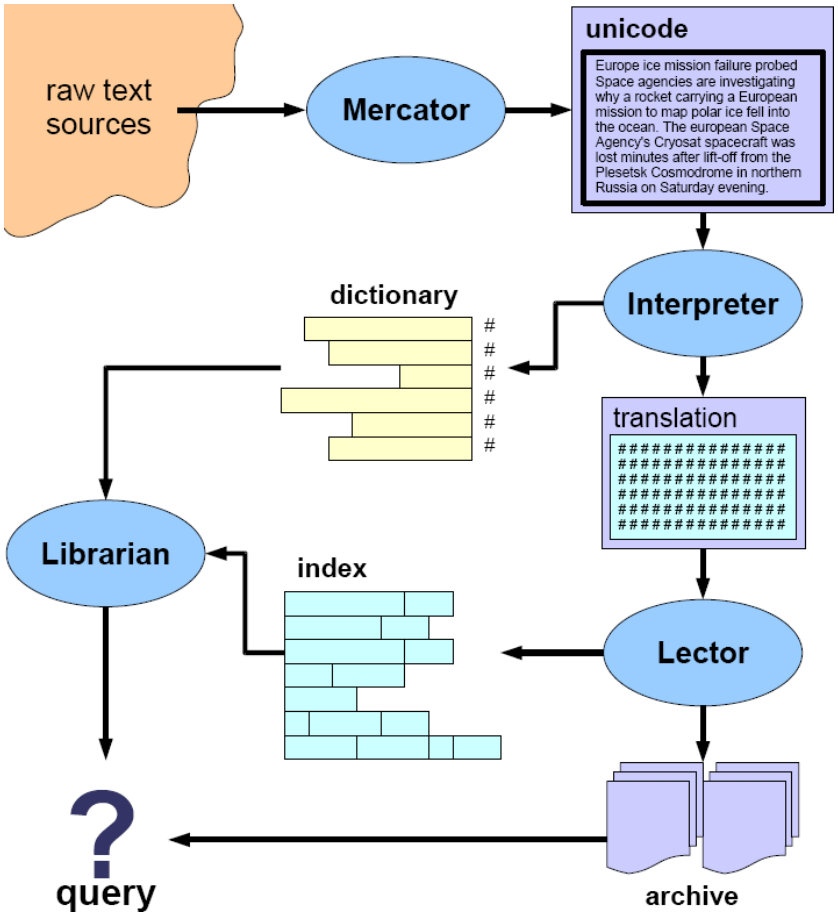


Figure 8: Structogramm of *Palaver*

To gain the highest possible degree of flexibility, the basic design principle was to use small modules that are specialized on one particular task only, and that do not need to exchange control information between each others – except for the processed textual data. No global control variables were allowed.

The application was split into a series of components, which can be grouped into two types: *task objects* and *storage objects*. Tasks read the input data from a storage object, process the data and then store the result in a different storage object, for the next task to process.

The processed textual data is represented as *Document* or *Translation* objects. The full text index is represented as a collection of *Hit*, *HitList*, *HitMap* and other data objects. Unique words are not represented in an object oriented fashion.

A *task object* could be seen as the implementation of a routine. For example, translating a natural language document into a vector of word IDs is one *task*, called *"Interpreter"*. *Storages* could be regarded as little databases of their own, which allow concurrent data access for multiple tasks. Concurrency has only to be observed inside storage objects which reduces the program complexity and thus facilitates development. Also, storages are very generic, leading to source code that is being reused in multiple objects.

This software development approach is known as *"divide and conquer"*: by breaking down the very complex task of a monolithic and multi-threaded full text database engine into small, independent and compact software modules, the complexity that needs to be observed while programming is reduced a lot. However, finding a global architecture that allowed this kind of breakdown was not trivial. This process took several months until it was present in the current form, as shown in figure 8 (page 44). During the early days of the applications life time, the global design was changed and refactored numerous times.

Breaking down the application into small modules facilitated the process of development and debugging, it also led to very flexible software. With the modules being specifically adjusted, very specialized to perform one single task only and thus very compact, it was easy to extend them with new features, to reimplement them if needed and to replicate them on other, remote machines. For example, implementing a distributed database system could be done by modifying the storage objects only: instead of storing data locally, it could be transferred to a remote node on the network. Task objects or other storages would not require any changes.

The simplistic data exchange between the modules makes it very easy to extend the database system to a network of multiple machines, thereby creating a distributed document retrieval system that can scale up with the amount of data.

5.3 Implementation Details

For performance reasons, the implementation itself was done in C++, using a lot of templated classes to facilitate code reusing[23]. Where possible, tex-

[23] A detailed documentation of the object oriented design and the actual source code can be found on `http://www.monkville.org`

tual data is stored using a *unicode* representation, which makes the system compatible to a wide variety of human languages and scripts. The usage of platform specific operating system calls has been avoided in order to create platform independent code. Wherever necessary, a software library was used as a wrapper for platform specific operations. For example, the handling of networking and threads is done by *PTypes*[24], to ensure platform spanning binary compatibility of unicode text, *ICU*[25] was used. Binary data is always stored in big endian format, data files created on a little endian machine can be used on a big endian CPU without further modifications.

To maintain high flexibility with all languages and scripts, the identification of single words in natural language text is done by ICU. As mentioned in chapter 3.3, the definition of a *"word"* also depends on the language and the script, therefore it would be futile to hardcode that definition into the implementation. ICU provides this functionality and is also flexible enough to accommodate different word boundaries for non-latin character sets.

5.3.1 Storages

All data and storage objects implement the possibility to store and load the object contents to and from a binary data repository. Such a repository can be a memory buffer or a binary data file.

Furthermore, data objects (*document*, *translation* and a series of smaller structures for the inverted index) implement the `bufSize()` method, which calculates the amount of bytes used to store all data in the object (or, to *serialize* the object contents). This is used for memory and data file size management, and a series of run-time performance optimizations.

Full text index data is stored in a series of binary files. The basic implementation is the one of a file system (figure 9): binary data is split into multiple *data blocks* (table 5, p. 47). The assignment, size and relative position of those blocks is stored in a *inode* file, which contains a series of linked lists of inode records as shown in table 4 (p. 47). The managed content itself, that is, the list of objects, is stored in a *inventory* file. It contains all object identifiers, the first data inode for each object and other global management information like inodes that point to unused data blocks (see table 3, p. 47).

The logical implementation of storages can be of two basic natures: a *FIFO queue*, or a *random access storage*. FIFO queues (called *"FifoStore"* in the implementation) are used whenever objects need to be stored, that cannot be unambiguously identified in the application. For example, when a raw text file is being read and cached for later processing, it is first loaded

[24] `http://www.melikyan.com/ptypes` – C++ Portable Types, by Hovik Melikyan
[25] `http://icu.sourceforge.net` – International Components for Unicode, by IBM

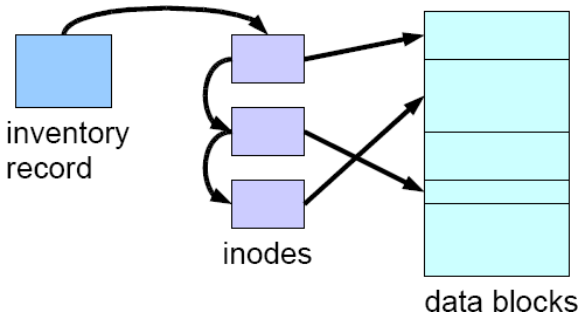


Figure 9: Structure of binary data file storages. For information about more detailed binary layout of the components, refer to table 3 (inventory record), table 4 (inodes) and table 5 (data blocks)

field	**length**	**description**
# free inodes	32bit	number of unused inodes
<free inode offset>	32bit	file offsets to free inodes
# number objects	32bit	number of stored objects
<object ID	32bit	unique object identification number
inode offset	32bit	pointer to the first inode
object size>	32bit	total size of the object in bytes

Table 3: Structure of the binary inventory file. Entries enclosed in *<brackets>* represent repeated data records.

field	**length**	**description**
# inodes	32bit	number of inodes in this block
<block offset	32bit	offset of the data block in the data file
block size>	32bit	total size of the data block
next inode	32bit	offset of the next inode block

Table 4: Structure of one single binary inode record. Entries enclosed in *<brackets>* represent repeated data records.

field	**length**	**description**
object ID	32bit	object identifier
block length	32bit	available bytes in this data block
data length	32bit	used size of the data block
data	arbitrary	used data
unused	arbitrary	optional unused padding to fill the block

Table 5: Structure of one single binary data record. The object ID is used for error detection.

into a cache. At this point, the document ID for this file is still unknown, therefore it cannot be unambiguously identified yet.

Random access storages (called *"OrganizedStore"*) allow to read and write any object that has an identifier. Therefore, their implementation is more complex than the one of FIFOs. These stores are also the ones that require the file system implementation as shown in figure 9.

To increase performance and reduce the amount of disk operations, the implementation makes use of memory caches, using the *"least recently used"* (LRU) caching strategy. This way, objects that are used regularly do not have to be fetched from disk every time.

The data storage implementation is not yet ideal and leaves much opportunity for optimizations. For example, no data compression is used, and the implementation shows a tendency towards file fragmentation when operating on very large amounts of data, where only a small part can be held in the memory cache. Some of the improvements listed in chapter 8 could also be applied to the *Palaver* implementation, but are beyond the scope of the current project state.

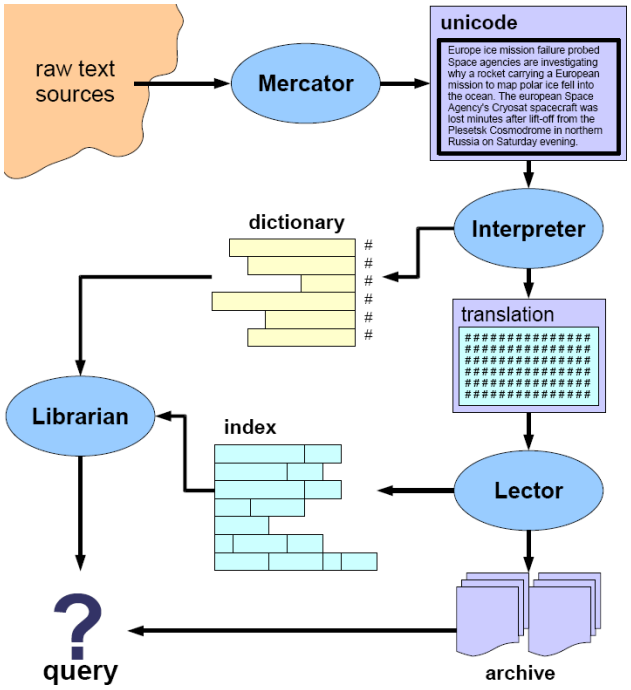
5.3.2 Description of Modules

To add a document to the index, the first step is to acquire it from whatever source that is available. This is done by the *mercator* task: it reads textual input from any form of available medium like web pages or plain text files on disk. One could also think of the possibility to acquire textual data from proprietary binary formats such as *Adobe PDF* files, or *Microsoft Word* documents.

Mercator

The *mercator* transforms the document from its raw encoding into plain unicode text and creates a *document* object from the text source. Table 6 contains a list of the important values stored in this object. However, the document identifier (ID) cannot be set at this point because it is known only after the document was added to the *archive*. But as the ID is not needed at this point, this is not a problem.

The *mercator* stores all acquired documents in a local cache, the *mercator storage*. This cache is, like all other caches in this implementation, a *FIFO queue* – meaning that it is impossible to fetch one specific document from the cache. As they do not have a document ID at this point, it would be impossible to identify one specific object anyway.



field	type	description
URI	unicode	unique resource identifier (or URL)
ID	int 32bit	numerical document identifier
body	unicode	documents text content

Table 6: Relevant variables stored in a *document* object (the binary representation in the data file contains additional values)

Interpreter

The *interpreter* reads documents from the mercator storage, creates the *document translation* (see table 8) and stores them in the *interpreter storage*.

During the translation process, it identifies all words in the document and transposes them into word IDs. Distinct words are identified by using the *word iterator* from the ICU library, word IDs are generated by reading and storing words into the dictionary. Before they are processed further, words are converted to uppercase strings, again using an ICU function in order to support uppercase conversion for all encodings. For each word, the interpreter first calculates the word ID, thereby adding new words to the dictionary. Then it appends that word ID to the vector of word IDs in the translation object as shown in table 8. The layout and content of the dictionary can be seen in table 7.

The document translations are stored in the *interpreter storage* (again a FIFO queue, like the *mercator storage*).

field	type	description
wordstring	unicode	concatenated unique words in the dictionary
# words	int 32bit	number of unique words
<*wordID*	int 32bit	unique word ID
offset	int 32bit	offset of the words first character in wordstring
length>	int 16bit	length of the word

Table 7: Structure of the dictionary, entries enclosed in <*brackets*> represent repeated data records. Instead of storing separate unicode string objects for each distinct word, only one string object is used to save memory. For each unique word, only the offset of their first character in that wordstring is stored.

field	type	description
URI	unicode	unique resource identifier (or URL)
ID	int 32bit	numerical document identifier
body	unicode	documents text content
# words	int 32bit	number of words in the document
<*word ID*>	int 32bit	vector of numerical word identifiers

Table 8: Contents of a translation object – The document (table 6) is being extended by a vector of word IDs.

Word IDs are 32 bit values, created by hashing the unicode string of the word using a hash function. Unfortunately, finding a good hash function that guarantees an unambiguous mapping of two distinct words to two distinct word IDs is a challenge. *Palaver* uses a subset of the common MD5 hashing function to generate word identifiers. MD5 maps arbitrary data to a 128 bit "fingerprint", but only 32 bits are used for the word ID.

By using this hashing function, it is possible that two distinct words are being mapped to the same word ID. Therefore, to guarantee unique word IDs, the word mapping has to be checked against the dictionary, using the algorithm *"Adding words to the dictionary"*.

This results in the implication that, whenever a word in its textual form needs to be translated into the word ID, this cannot be done without reading the dictionary. It has the further implication that words must never be removed from the dictionary, as this would jeopardize this algorithm.

Consider the following example: the strings of words A and B both map to the same word ID X. When word A is being added to the dictionary, it

Algorithm 1 Adding words to the dictionary

```
1:  procedure addWord(word)
2:      wordID ←hash32(word)
3:      if wordID ∉ dictionary then
4:          addToDictionary(word, wordID)
5:          return wordID
6:      end if

7:      w ← readFromDictionary(wordID)
8:      while w ≠ word do
9:          wordID ← wordID + 1
10:         if wordID ∉ dictionary then
11:             addToDictionary(word, wordID)
12:             return wordID
13:         end if
14:         w ← readFromDictionary(wordID)
15:     end while

16:     return wordID
17: end procedure
```

will be inserted using ID X. When word B is added at a later time, it will be inserted using ID $X + 1$. Now, if word A is removed from the dictionary, and word B re-appears in a new document at a later point, it will be added using ID X, leading to the same word being stored twice in the dictionary, and to ambiguous word IDs.

Lector

The *lector* is responsible for the most expensive task: the *indexing*, or sorting document translation into the *inverted index*. It reads the objects stored in the interpreter storage, and inserts the translation into the inverted index, as described in chapter 3.1.3. The indexed documents are then stored in the *archive*.

Actually, the implementation *first* stores the document in the archive, *before* indexing it. By doing so, the system assigns a unique document ID to the translation object which is needed in the inverted index. This ID is calculated as a hash function over the documents *unique resource identifier* (URI), very similar to algorithm 1. The archive is of the *"OrganizedStore"* storage type (refer to the logical storage classes on page 48), which allows to

read and write arbitrary objects. Documents are relatively seldom fetched from the archive, therefore they should be stored in a compressed form.

Librarian

The librarian is the task responsible for handling user search queries. All query terms are translated to the respective word IDs using the dictionary. Then, all documents that contain these terms are fetched from the inverted index. The user may add boolean operators to query terms, thereby searching documents that contain some specific terms, but do not contain others – like *"king AND pop NOT Jackson"*. If such operators are being used, the resulting document lists need to be merged or intersected accordingly.

Once the query result is calculated, it is being ranked (ref. chapter 4), and returned to the user in descending order of relevance. The responses include the document URIs, and references to the archive. The actual ranking algorithm used in Palaver is very basic, it does not determine any actual measurement of relevance. Ranking is done globally, based on a naive estimation of document quality.

5.4 Runtime Considerations

The implementation itself is, despite its modular software design, one monolithic process that spawns a number of threads. During index creation, two threads work in parallel: one for the *interpreter*, identifying words and creating the dictionary. The other one is running the *sorter*, which requires a lot of computation and disk operations.

Test results have shown that this parallelization increases the throughput during index creation, an observation also made by Melnik et al. [31] This gain could probably be further increased by having multiple sorter threads running in parallel, and using a pipelining system: while one thread is resorting the index, the other one performs disk operations.

By using a cache for the index, the amount of necessary disk operations can be reduced. The cache should be large to reduce the amount of disk operations to a minimum. Tests have shown that the usage of a limited amount of memory for the index does not significantly increase the time needed to index documents. Compared to an implementation with unlimited memory usage, indexing time increased only by an estimate [34] of 12%.

6 Monkville: Introducing Semantics

So far, the project covered basic full text search functionality, which is sometimes also referred to as boolean search. [23] Boolean search only retrieves documents which meet an exact matching criterium, thus documents that contain all required search terms, and do not include terms that have been excluded from the result. This functionality was implemented in *Palaver*.

However, the scope of boolean full text search is limited. To get useful results for any query, the user needs to predict – to a degree – the content of the documents he is interested in. Common boolean text search does not support any abstraction of content. For example, to search for web sites related to *"outdoor activities"*, a user would likely have to use query terms that contain more specific keywords, like *"fishing"*, *"hiking"*, *"camping"* etc. But by using these search terms, the query result will of course be restricted to sites containing those specific keywords. The search engine will never return sites related to *"sailing"*, *"para gliding"* or *"diving"* if those keywords were not given, despite the fact that they are outdoor activities too. This additional level of meta-search requires some form of inference over the text collection.

Monkville aims to add this additional level of content abstraction, allowing semantic meta-searches in a full text index. This is achieved by inference over additional content in the full text index, like context information for web sites as shown in figure 1. Also, it aims to increase the quality of the search results in terms of semantic relevance, and to create a "smart" full text search engine that helps users to navigate through textual content.

With the concept of semantics added, it is possible to perform topic based "meta-searches" over the full text index: fetch all documents that are related to a given topic of interest, and on that subset of documents, perform a boolean keyword search. For example, instead of searching for *"Google"*, *"Yahoo!"* or *"MSN search"* to retrieve all sites about web search engines, the user can enter the query term *"search engine"* to get a comparable result. On that subset of documents, he then can further restrict the results to documents containing additional specific terms, like *"architecture"* or *"size"*.

A second and very interesting possibility is to find web sites that are semantically related to each other. For example, if the user finds a document that focuses on his topic of interest, he can search for similar sites based on the semantic content, instead of repeating his queries all over again, using slightly different keywords.

6.1 Sources of Semantic Information

In order to increase the awareness of semantic relationships between documents, the semantic content of a text needs to be assessed and "understood" – at least to a certain degree. Facts and topics that are given in the text need to be identified and extracted to be able to infer semantical relations. As this type of information usually is given only *implicitly* in the unstructured natural language text, it is unknown to the computing system [25] and needs to be extracted by using heuristic algorithms, like LSI (see p. 35) or *"document fingerprinting"* as presented in chapter 6.3.1.

But semantic information can also be given *explicitly*. If the text is structured, and if that structure is known to the computing system (or can be learned automatically [49]), it is possible to extract explicit context information from the natural language text, and use it directly – without running through heuristics. For example, news articles on web sites often contain informational elements that are not part of the textual content itself, but are meant as an aide for human readers who want to know more about the story. Such context information is shown in figure 1 on page 7, and usually contains links to related stories and web sites – precious information for a computing system.

Explicit semantic information can also be given in other ways. For example, HTML documents often contain content descriptions in meta-tags. Facts and relations between them could also be given in *RDF*[26] annotation [37] as defined by the *semantic web*.

Explicit semantic information can be hard to come by, because it is not necessarily given with all documents. Implicit semantic data is always given with the text itself, but it is hard to extract.

6.2 Explicit Semantic Information

Explicit semantic information has a number of drawbacks: either it is not generally available, or its quality cannot be guaranteed. HTML meta-tags for example, which were intended to contain a descriptive summary of a web sites, were originally introduced to help in the process of navigating through the web. Automatically created index sites contained lists of web sites, with the short summary directly from the HTML meta-tags. As the web became more popular, those meta-tags were being abused to artificially increase a web sites popularity in search engines. Also, it never was very clear what those meta-tags were supposed to be used for – or better, who would be

[26]RDF: *Resource Description Framework*, easily parseable data files containing specifications of meta data models, see also [5] and `http://www.w3.org/RDF`

using them, automated index systems or human readers. They often contain excerpts in natural language that are useful for human readers, but hard to comprehend for a computing system. Others only list some keywords, which is mostly useless for a human reader.

The semantic web annotates explicit logical information in a form that is specifically tailored to be read by computers. Facts (or objects) and the relations between them are represented in the RDF notation, which can come in different formats but is well defined, and as such easy to parse for a computing system. However, the semantic web is not yet widely applied on common web sites, and thus the concept is not interesting for search engines – the vast majority of web sites comes without RDF annotations. The semantic web could suffer from a similar fate as HTML meta-information. As soon as it is being used as a ranking criterium in web search engines, it will become subject to spamming.

6.2.1 Lixto: Semi-Automated Extraction

A common and widely used format for semantic information on the web is not yet available. However, it is possible to exploit the structure of web sites that contain rich information with textual data. For example, many news sites provide additional, redactionally edited content on the web where they publish their news stories and reports online. Such additional content can be links to related stories, links to web sites that cover specific topics mentioned in the report itself, and other similar information. For a computing system, this information is not explicitly known, while it is obvious to a human reader. With Lixto, it is possible to make this implicit information explicit, and to ease its further processing.

While the presence of context information is obvious to a human reader, it is hard to identify for computer systems without the help of an operator. With Lixto, it is possible to define the structure of a web site using a visual user interface, and then generate a wrapper program to extract the required context information from the news sites. Lixto creates a wrapper program, which can be used for all web sites with the same or with a similar structure. Using this technique, it is necessary to define the site structure (the wrapper program) only once, and then reuse the program in an automated process. Changes to the web site layout usually do not affect the extraction result of Lixto, as the wrapper program is very robust and allows for a certain degree of structure variations.

The output of the extraction process is an XML file (see figure 10), which can easily be processed further as needed.

During the course of the project, Lixto extracted over 7000 articles from

Figure 10: Lixto identifies text and context components from web sites, and makes the data available in a XML format

four major news sites within 4 months, without requiring corrections to the extraction programs – despite the fact that two of the four sites gradually changed the site layout during the time.

With this wrapper technique, it is also possible to use only the required parts of news articles. Irrelevant components on the web sites, like menu structures, navigation links and advertisement are not processed into the full text index, which greatly increases the quality of the entire text collection.

Articles are basically divided into two components: the article *text*, and *context* information. The text contains all components of the entire article, like headline, subtitle and the subsequent article body. Context information is everything on the web site that somehow relates to the article itself, but is not an integral part of it. For example, a *BBC* report about the british pension system contained numerous links to related stories about the british public sector, life expectancy and state pension facts. It also contained links to *The Financial Times* and the british *Pension Commission*. While this information is not actually part of the article itself, it gives very valuable hints about the semantic content of the article, and the covered topics. For a computing system, this is a great help to infer the actual semantic content of the article.

The disadvantage of using this approach for context extraction is the lack of full automatization (the wrapper program has to be created manually). However, the amount of work required to do so is minimal, and the result is of a very high quality. There is a second advantage: this method is very resistant to spamming. As all textual source of information are clearly visible

for human readers (unlike HTML meta-tags), it is very unlikely to be abused for search engine spamming.

6.3 Implicit Semantic Information

While implicit semantic information cannot be subject to spamming, it has the big disadvantage of not being explicitly known. It needs to be assessed or extracted from natural language text, which is notoriously hard [25] and requires sophisticated heuristic algorithms. There are some techniques based on the vector space model (ref. to chapter 4.1) like *TFxIDF* or *latent semantic indexing* (LSI). Those techniques are able to identify – to a limited degree – the most important keywords in a text, without "understanding" their meaning or context. Thus, while they are able to identify the keywords of a document, they are not suited to group documents into topics. However, they are able to evaluate a form or "semantic relevance" between two given documents, thus to perform fuzzy topic grouping as described in chapter 6.6.1.

6.3.1 Document Fingerprinting

There is a technique that can be used in vector space models to help or improve the identification of significant parts in a text: Extracting *n-grams* [4, 32] or word sequences from documents. The idea is based on the assumption that, if a sequence of words appears numerous times in a text, it bears some semantic significance. A repeated sequence of words (which is one *n-gram*) has a different, more detailed semantic weight in the text, which makes them predestined to be used as a form of implicit semantic information, or meta-information of the document. In fact, it has been shown that by using this technique it is possible to achieve good document topicing results, that is, mapping documents to existing topic categories [33, 32].

The exact way to extract those *n-grams* differs depending on the needs, but the basic algorithm is always the same. Extraction of word sequences is done in n passes, where sequences of length n are extracted in the n-th pass, as shown in algorithm 2. Mladenic et al. show in [33] that word sequences longer than 5 words do not appear often enough or contain much semantic relevance, thus the length of word sequences used in the *monkville* implementation is limited to 5.

These *n-grams* also can serve a second purpose: allowing phrase queries including stop words, without processing stop words in the full text index. For example, if a phrase like *"I am the king of pop"* appears more than once in a document, the complete phrase could be added as one "word", without

Algorithm 2 Generating the document signature

```
1:  procedure CREATESIGNATURE(V)                    ▷ V is a vector of words
2:      nGrams ← ∅                                  ▷ nGrams is a set of all word sequences
3:      pass ← 2

4:      while pass ≤ 5 do                           ▷ add all word sequences from 2 to 5 words
5:          i ← pass
6:          while i ≤ |V| do
7:              seq ← {V_{i−pass}, ..., V_i}        ▷ seq is the current word sequence
8:              if seq ∉ nGrams then
9:                  C_seq ← 1                       ▷ C counts the sequence occurrences
10:                 nGrams ← nGrams ∪ seq
11:             else
12:                 C_seq ← C_seq + 1
13:             end if
14:             i ← i + 1
15:         end while
16:         pass ← pass + 1
17:     end while

18:     for all seq ∈ nGrams do                     ▷ remove unique sequences
19:         if C_seq < 2 then
20:             nGrams ← nGrams\seq
21:         end if
22:     end for

23:     return nGrams                               ▷ sequences that appeared more than once
24: end procedure
```

indexing the distinct stop words themselves. Like regular words, the phrase itself gets its own "word" ID and is being added to the dictionary as one word string, consisting of a series of distinct words, separated by blanks. This way, a lot of stop word sequences can be used in full text search, without increasing the index size by adding each single distinct stop word on its own.

Systems that use *n-grams* for semantic evaluations only can also choose to remove stop words before calculating them [32], thus focusing on semantic evaluation only. Short phrases containing stop words are likely to be very common, like *"I am"* – and thus are semantically relatively irrelevant.

An other possibility is to take the best of both worlds: *n-grams* could be generated in two passes. First, including stop words, to add stop word phrases to the index for phrase queries. In the second pass, stop words could be ignored to increase the quality of the semantic evaluation. *Monkville* currently uses stop words in the index, as well as stop word phrases in document signatures. As it is not using stop word lists and indexes all words irregardless of their relevance and frequency, this option is a further possibility for performance improvements.

6.4 Exploiting Context

Based on the architecture used in *palaver* and the ideas presented above, the next step of the project was to add the concept of semantics to the full text index implementation. As shown in figure 11, the basic architecture was preserved, but extended by one very important component: the *semantic index*.

The *monkville* implementation uses three different types of content for each document, and two different full text indexes to store this content.

Each document is represented by three components: the usual document *text*, the document *context* as extracted by Lixto, and the document *signature* as generated using algorithm 2. These components are stored in two different full text index structures: the *search index*, containing the full text collection but excluding context information. Both indexes use the same dictionary to keep the amount of information redundancy to a minimum. The *search index* is used to perform the usual boolean full text queries, as in *palaver* and any other full text database.

Context information is stored in the *semantic index*, together with the documents signature, as shown in figure 12. This index is exclusively used for topic related meta-searches, not for the usual full text queries. As the signature contains semantically relevant entries (the longer the phrase, the higher the relevancy), it is being used in both indexes: In the semantic index as contextual information for documents, and in the search index to allow

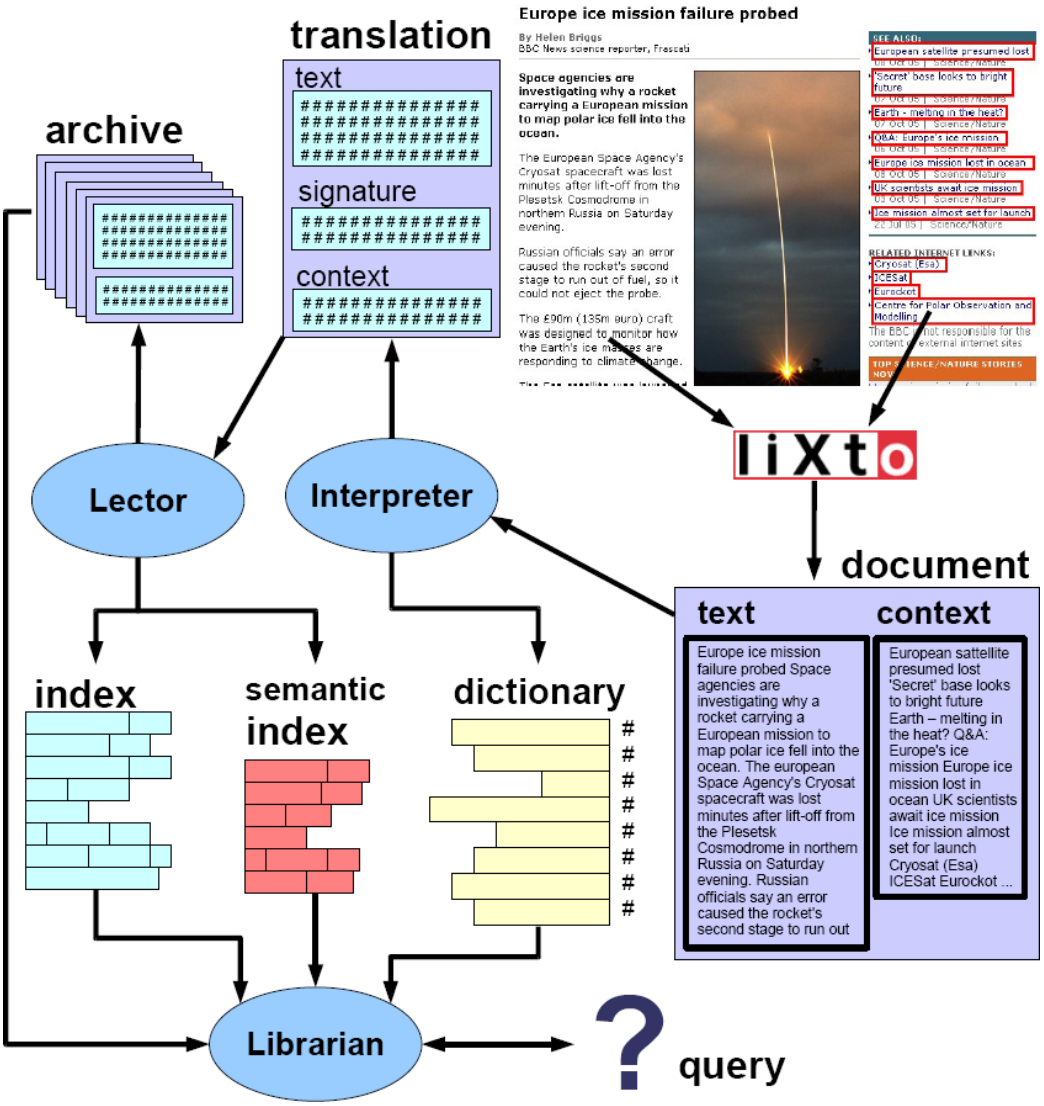


Figure 11: Structogramm of the semantic index implementation. Based on Palavers architecture, Monkville uses an additional full text index for semantic data. Context information is extracted from web sites, using the ideas and tools presented in 6.2.1.
Documents processed by Lixto are stored in XML and then converted into two text files: one for content, one for context data. Those text files are being read by the mercator which is not shown in this figure, but described in 5.3.2 (p. 48)

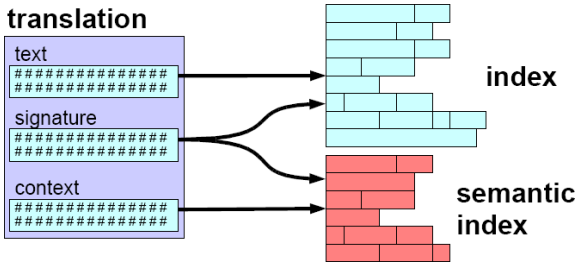


Figure 12: Mapping of content forms to the two indexes

phrase queries on documents – even when using a *record level index*[27] (see 3.2.1, p. 24).

6.4.1 Context Search

By separating the search index from the semantic index, it becomes possible to perform specific context searches. A context search returns different results than a usual boolean query on the full text index, because the search is not performed within the document contents itself, but in the semantically related context of the documents. This is an important distinction: if document contexts would be added to the full text index, search results would become incorrect. Context is not a part of the document itself, it merely is a list of keywords that are semantically related to the document. Thus, context information must not be added to the search index.

Context search can also be seen as *semantic search*, meaning a meta-query for semantic content, not keywords. For example, it is possible to search for documents related to *"fund raising"* in the context of *"tax cuts"* – a level of functionality that cannot be offered by usual document retrieval systems. The context search depends on the quality of the extracted document context, as it is impossible to determine such information from a text in natural language only.

The combination of semantic searches with the functionality of standard full text search provides a very high degree of flexibility. There are many possible search combinations: the usual boolean full text search, the same boolean full text search on the semantic index only, and any possible combination thereof. For example, it is possible to retrieve all documents in the context of *"weather phenomenon"* while excluding all documents that contain the keywords *"hurricane"* and include all documents containing the phrase

[27] The implementation uses word level indexes, but does not use the additional information for querying and result ranking yet. Possible performance and/or quality improvements are discussed in chapter 6.6

"global warming" in the signature. The result of this query could further be reduced to documents in the context of *"oil price"* – there is no limit to the possibilities.

6.4.2 Discovery of Related Documents

The document context is not essentially a part of the document content itself, but very useful to perform a series of statistical content evaluations. With a given document d, it is possible to find semantically related records by using the queries as shown in figure 13.

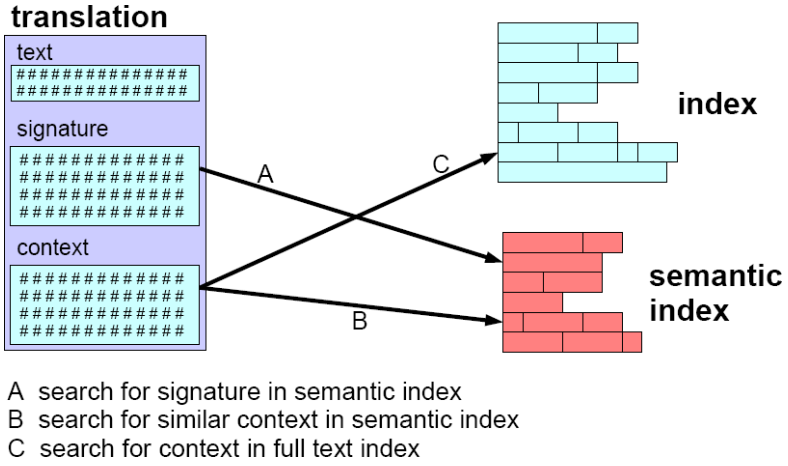


Figure 13: Discovery of related documents based on document signature and context

To discover related documents, three queries are performed on the semantic and full text index: A, B and C.

Query A is an evaluation of the document signature. The n most frequent signature entries are being fetched from the semantic index, resulting in a list of documents that contain similar signatures. Formally:

$$A = \bigcup_{i=1}^{n} Q_{sem}\left(S_{d,i}\right) \tag{11}$$

d	=	the document being evaluated
$S_{d,i}$	=	i-th word in the signature S of document d
$Q_{sem}(w)$	=	search result for term w in the semantic index
n	=	upper bound for evaluated signature entries

Query B is a similar evaluation of the context. It returns all documents within a similar context group and is the strongest indicator for semantic

relationship. Unfortunately, not all documents necessarily have context information, thus it cannot be used alone. Formally, B is:

$$B = \bigcup_{i=1}^{c} Q_{sem}\left(C_{d,i}\right) \tag{12}$$

$C_{d,i}$ = i-th word in the context C of document d
c = the number of words in the context of d
Remaining terms as above.

Query C could be seen as the common semantic denominator. It includes all records that contain query terms from the evaluated documents context. C usually is the largest set in the evaluation process and is intersected with the union of A and B, therefore it should be evaluated last. Like Query B, it depends on the document context. Formal description of C:

$$C = \bigcup_{i=1}^{c} Q_{tex}\left(C_{d,i}\right) \tag{13}$$

$Q_{tex}(w)$ = search result for term w in the full text index
Remaining terms as above.

The final combination of these evaluations gives a set of responses that contains all records with a semantical relationship to the evaluated document d. The "strength" of the semantical relationship can be related to the number of occurrences of the same document within the different sets.

Some documents might have no associated context information, therefore it is necessary to make a distinction based on the presence of context. The best way to combine the sub-queries remains to be determined, but the following combination of results showed good results:

$$R_d = \begin{cases} (A \cup B) \cap C & \Leftrightarrow C_d \neq \emptyset \\ A & \Leftrightarrow C_d = \emptyset \end{cases} \tag{14}$$

Where R_d is the set of documents with a semantical relationship to d. The "significance" of the relationship can be estimated by evaluating the number of occurrences of the documents in the multisets returned by the different sub-queries:

$$S_{r,d} = \begin{cases} \left(\frac{\sigma_A(r)}{|A|} + \frac{\sigma_B(r)}{|B|}\right) \cdot \frac{\sigma_C(r)}{|C|} & \Leftrightarrow C_d \neq \emptyset \\ \frac{\sigma_A(r)}{|A|} & \Leftrightarrow C_d = \emptyset \end{cases} \tag{15}$$

$S_{r,d}$ = significance of relationship between documents d and r
$\sigma_M(x)$ = number of occurrences of element x in multiset M

field	type	description
URI	unicode	unique resource identifier (or URL)
ID	int 32bit	numerical document identifier
body	unicode	documents text content
context	**unicode**	**document context as extracted by Lixto**

Table 9: Modified document object. The only addition (in **bold**) is the context field. The signature is calculated during translation, and not included in the raw document object

Obviously, this approach delivers best results only if it is being used with documents that have semantic context assigned to them. Also, note that neither the calculation in equation 14 nor the one in equation 15 are commutative. Thus, if any given document A is determined to be related to a document B, the same evaluation of document B might not indicate any relationship to A.

These evaluations currently are experimental and have not yet been compared in detail to other possible combinations. Such a comparison would require some form of standard, like TREC or CLEF which are used to estimate precision and recall values for document retrieval systems (ref. chapter 2.3, p. 15). But similar to those evaluations, the concept of "semantical relationship" is subjective and depends on the actual needs of the user. It may be difficult to find good methods to objectively measure the quality of these results.

6.5 Modifications to the implementation

Due to the modular design of *Palaver*, the addition of the semantic index and the extra data fields to documents could be done with a minimal amount of changes of the source code.

Table 9 describes the new data structure for the *document* object, as it is being used in *monkville*. The only necessary addition was a new data field for the documents context, with some obvious modifications to the object implementation itself: loading and saving routines needed to be adapted. The *mercator* (see figure 8, p. 44) was extended to include context information loading when acquiring new documents. Other than that, no significant changes needed to be done for the process of text acquirement.

During the *translation*, the document body and context are transformed into vectors of word IDs. Also, the document signature is calculated from the document body, using algorithm 2. Context information is not included in signature calculation. Table 10 shows all modifications to the *translated*

field	type	description
URI	unicode	unique resource identifier (or URL)
ID	int 32bit	numerical document identifier
body	unicode	documents text content
# words	int 32bit	number of words in the document
<word ID>	int 32bit	vector of numerical word identifiers
# context	**int 32bit**	**number of words in the context**
<word ID>	**int 32bit**	**numerical context word identifiers**
# signature	**int 32bit**	**number of words in the signature**
<word ID>	**int 32bit**	**vector of signature word identifiers**

Table 10: Modified Translation object. The additions (in **bold**): translated context and signature

document object.

For the semantic index itself, the existing full text index implementation could be reused without modifications. This resulted in a word level semantic index, which presents an extra overhead that is not necessary – a record level index is sufficient for semantic search. The additional data in the word level index (word positions inside the context) results in extra processing costs, which have of course an effect on the test results. Nevertheless, the overall system performance is satisfying.

The only major modification of the *Palaver* implementation was related to query evaluation as discussed in chapter 6.4.2. Queries are being evaluated using the search index and the semantic index at the same time. The command interface for semantic searches needed to be implemented and extended, as explained in the database manual (see appendix A, page 79).

6.6 Further Work

The implementation of *monkville* leaves a lot of possibilities for improvements. While the system can demonstrate that the general functionality is working and the concept shows good results, some further work is necessary to include additional features, and to remove some bottlenecks and unnecessary overhead.

One basic improvement would be the usage of stop word lists to reduce the amount of processed data, and thus increase performance. Unlike usual document retrieval systems, the usage of stop word lists in *monkville* would not have a serious impact on the functionality of the search engine. While it would become impossible to search for distinct stop words (such as a query for *"the"*), such searches would be useless anyway, because the amount of

documents matching such a query is beyond any reason. However, it would still be perfectly viable to search for phrases that contain stop words (for example, a phrase like *"to be or not to be"*) – based on the document signatures. Only documents that contain the queried phrase more than once would be returned, which is a desirable behavior: especially short stop word phrases (like *"I am"* or *"this is"*) appear very often in the text collection. Reducing the size of the query result to include significant documents only helps improving the global performance and the quality of search results.

As stop word phrases have a relatively small semantical relevance, the document signature could be generated in two passes. In the first pass, phrases including stop words could be added to the signature, thereby adding all repeated phrases to the full text index. For the second pass, all stop words could be removed from the translation vectors, and the signature could be recalculated to be used for the semantic index. A similar approach was proposed by Mladenic et al. in [32].

Other improvements could be made by reducing the amount of unnecessary data granularity. With the support of phrase queries being provided by the document signatures, the necessity of a word level index for full text search could be discussed. But in any case, using a word level index for semantic searches is not necessary. A word level index was used in *monkville* to reduce the amount of source code changes. By reducing the indexes to record level, the index size on disk could be reduced by over 50% [40], at the same time reducing the indexing costs [52] – without reducing the functionality of the database.

Another obvious improvement of the implementation would be the usage of data compression for the indexes, as well as for the archived documents. Using data compression for an inverted index has been proposed numerous times [51, 40, 44, 12, 34]. The computational overhead for the compression can be more than compensated by the reduced amount of data that needs to be transferred to and from a disk storage.

6.6.1 Topic Grouping

With the methods described above, it is possible to find semantical relationships between texts. One document can be related to many other documents, and to many different topics. Those relations could be used to map documents on "topic groups". Unlike traditional content clustering where documents are mapped on existing topic categories [33] or where the intention is to find good topic definitions for existing text collections [27, 10], the goal here is not to identify good topics for documents, but to find fuzzy "topic clouds" that have a semantical relation to any document.

Documents with similar content are grouped together in similar topic clouds, whereby one document can be assigned to numerous topics, and one topic can be related to numerous documents. The result would be a multi-dimensional "topic space", where one document is related to a number of points in this space. Areas with many points in close proximity could be grouped heuristically into "topic clouds", without the need to actually find good descriptions for each specific topic.

With this fuzzy definition of "topic", it would be possible to find a high-dimensional linked structure for a text collection, where links are based on semantic similarities, rather than on existing hyper links on the web. Navigating in this high-dimensional semantic space would mean to travel on the net using meta-paths that show the way to semantic content, rather than to specific web sites. The links between web sites would be based on semantic relevance, leading to a semantical web structure.

6.6.2 Semantic Result Ranking

With the discovery of these relationships between documents, it is possible to create a new ranking algorithm which is based on semantic relations.

Documents that belong to the same topic group (see above) can be grouped together. To do this for a search result, one first should find m topic groups that accumulate the most amount of "significance" into them, then assign exactly one of those groups to each document in the result set. To avoid reappearing results, one document may only be assigned to one single topic group, ideally the one with the strongest semantic relation. After forming these groups, take the n most significant documents for each topic group, and exclude the remaining documents from the result list.

By using such a content-based ranking algorithm, it should be possible to create a search engine that provides the possibility to navigate through the textual content, based on semantic concepts rather than specific keywords or links. Also, such a ranking algorithm would not prefer popular web sites to unknown ones, because it is not based on link analysis (see chapter 4.5). Instead, unrelated web sites (especially those not being hyper-linked to each other) can suddenly become related, because they cover a similar topic base.

7 Experimental Results

The implementation was tested on a set of 7300 documents, extracted during 4 months from various news sites as described above. The plain text of those articles summed up to 32.8 megabytes of uncompressed, unformatted textual data, 7% of which consisted of context information only. 87% of the extracted documents had context information associated with them. On average, the context data amounted to 7.3% of the total data size.

Text files were collected on a server, in a SQL database. From there, they were extracted as plain text files, then processed by the full text database implementation. Context information was stored in separated files, if it was available.

Zobel et al. suggest in [51] that an artificial text collection, created by a software utility, can be used to conduct performance tests on text retrieval systems. An artificial text collection has the advantage that it can be created with unlimited size and vocabulary, with similar statistical properties to real text collections. Of course, such a synthetic text collection is by no means a substitute for a real database, but it can be used to benchmark the performance of a text retrieval system in terms of speed and index size. Of course, it cannot be used to examine the effectiveness of ranking algorithms or to estimate precision and recall, nor can it be used for semantic analysis [51].

Unfortunately, tests regarding precision and recall (chapter 2.3) have not been conducted, since none of the benchmark text collections were available. Furthermore, these text collections would lack context information, and thus would not be suitable for a meaningful performance evaluation.

7.1 Processing Time and Data Size

All performance tests were conducted on a personal computer system with one gigabyte of memory and an AMD Athlon XP 1600+ CPU, running from a IDE hard disk drive. This system specification is far from optimal for this type of application, which requires especially fast hard drive access due to the big volume of data read from and written to disk. However, as all test runs of the database implementation and the subsequent MySQL test runs were conducted on the same machine, the hardware limitations are not relevant for a direct, relative comparison of the evaluated database systems [34]. Appropriate hardware could boost the overall performance, resulting in better test results and system speed.

The indexing and signature generation process for 7300 documents, with a total raw text size of 32.8 megabytes with context information, took 10 minutes and 37 seconds – while large parts of the inverted index were stored

in a memory cache. Alone the subsequent writing of the index data to disk required another 4 minutes and 45 seconds, the resulting data files amounted to 298 Megabytes on disk (of which an estimate of 42% was unused disk space due to data fragmentation).

The test runs showed that the implementation of the data file storage needs some further improvements. Data is stored in sequences of data blocks with a fixed minimal size. Once written to disk, the sequence of data blocks remains as it is. One data object that is stored to disk will always be stored in the same data blocks in the data file, with new blocks being added as the object grows in size (as it happens to inverted lists during indexing). As the indexing process continues, several list entries are repeatedly written to disk, then re-read from disk to add more data. With each cycle, one data block is added in the data file, resulting in a lot of seeking operations whenever the object needs to be written to or read from disk. This presented the most severe performance bottleneck of the implementation.

The text collection contained 201704 distinct words, resulting in a dictionary file of 4.4 megabytes. Due to the two-byte representation of unicode data in memory[28] and some additional data management variables, the dictionary required 10 megabytes of memory after being loaded. This suggests that with modern hardware, keeping a dictionary in memory is not likely to present a resource problem.

By storing the same signature data (see chapter 6.3.1) in two inverted indexes, this data becomes redundant which of course increases the overall index size. This could be compensated by using data compression and a record level semantic index.

7.2 Performance Comparison

The database implementation was compared against the performance of MySQL. For this test, a different, much bigger text collection has been used. The same text collection was repeatedly indexed in both database systems[29]. While indexing performance of MySQL was far better than the indexing speed of the inverted index, the query performance of MySQL, as compared to the inverted index, dropped below an acceptable level very quickly.

For this test run, a collection of 3373 text files was used, resulting in a total collection size of 95 megabytes and a total of 210989 distinct words.

[28] On disk, textual data is represented in UTF-8 encoding, which uses a variable amount of bytes for one single character

[29] For an inverted index, adding the exact same text numerous times, is the worst case testing scenario: after the first run, every single word and document needs to be fetched from and written back disk.

The text files were taken from the user documentation of a Linux installation, but did not contain any context or text markup information. Also, signature calculation was disable during this test. The collection was processed in three passes, with query speed evaluations at the end of each pass.

Details about test results are shown in tables 11 and 12. The overall result indicates that, while MySQL shows better full text indexing performance, the query evaluation speed in the inverted index is – by far – superior, especially for large collections. The test queries were chosen to result in a lot of matching documents. Due to the Linux-oriented nature of the collection, the search terms were *"linux"*, *"root"* and *"computer"*.

	MySQL	**Monastery**	
		Reading	Indexing
1st pass	3:15	2:11	9:26
2nd pass	3:27	2:36	17:38
3rd pass	3:44	5:05	27:12

Table 11: Indexing times needed by MySQL and Monastery during the repeated indexing of the same text collection for three times. The time needed to read the data from disk and to add the data to the inverted index is listed separately for the Monastery. Times shown for MySQL are total times needed to process the text collection.

	query	**1st pass**	**2nd pass**	**3rd pass**
Monastery	linux	$<$ 1 sec.	3 sec.	5 sec.
	computer	$<$ 1 sec.	1 sec.	1 sec.
	root	$<$ 1 sec.	1 sec.	2 sec.
MySQL	linux	$<$ 1 sec.	53 sec	92 sec.
	computer	$<$ 1 sec.	3 sec.	10 sec.
	root	$<$ 1 sec.	14 sec	51 sec.

Table 12: Times needed to evaluate the search queries after each indexing pass. The test queries were chosen to return a large result set. 6051 documents contained the term *"linux"*, 1720 contained *"computer"* and 5787 contained *"root"*. The number of occurrences doubled in the second run, and tripled in the third run.

8 Related Work

The problem of information overflow and document retrieval is as old as the computing industry. Motivated by the need for automated indexing systems suitable for large amounts of textual data, there have been a lot of projects related to document retrieval, full text index performance improvements, semantic data representation and processing – all for a wide variety of applications.

This section will give a broad overview over some topics of interest that are relevant for an automated text processing system in the context of the web and full text search.

8.1 Improving the Inverted Index

The inverted index has been a subject of research for many scientists during the last years, and a lot of work has been done with the goal to increase index performance and reduce processing costs. Independent from each other, many of these projects suggest similar fundamental principles to improve the performance. Numerous publications by Zobel et al. [52, 40, 18, 48] and others [6, 31, 7, 12] suggest to use data compression to speed up indexing and retrieval, at the same time as reducing storage requirements.

One of the strong points of the inverted index is the possibility to compress its contents. Data compression has the disadvantage that all fetched records must be decompressed as they are retrieved, but this decoding can be very fast. Moreover, large parts of the decompression can be avoided by compressing single records only, so the main factor for performance limitations during data retrieval is the disk speed, and not the time needed for data decompression [51]. The smaller the amount of data that needs to be read and written to disk, the better is the overall performance. The time needed to read compressed data from disk and then decompress it in memory been shown to be less than the required time to fetch the same, uncompressed data from disk, without decompression [34, 40].

Various methods of data compression have been examined with inverted indexes, ranging from word-level compression [12] over various run-length encodings of binary data [52] to more expensive arithmetic coding algorithms, which require more computation time but achieve better compression rates [6, 31]. Dvorský et al. describe a run-length word level compression algorithm. A similar approach, but applied to complete records in the inverted index, is shown by Zobel et al. in [52].

Some index compression techniques also have disadvantages, and cannot be combined with other methods to improve performance. For example,

Google uses the *bzip2* algorithm for data compression [6] – a method which makes it impossible to exploit a possible query performance gain by compressing small pieces of data only – as shown by Scholer et al. [40]. The reason for this is that *bzip2* operates on very large data blocks, where it achieves the best compression rates. However, a lot of index compression approaches have the significant disadvantage of considerably increasing the costs for index updates.

The inverted index can also be divided into smaller segments, allowing to create it even with very limited resources – a method referred to as *single pass indexing* as shown by Heinz et al in [18]. The idea is to keep as much data as possible in memory, until it is depleted and everything needs to be flushed to disk, thereby creating a new index segment. After that segment has been written, a new one is started.

8.1.1 Additional Features

Other projects aimed at increasing the flexibility of the inverted index in order to implement additional new features. With the smallest entity in the database being one single word, there are some limitations to the types of queries that are possible with the inverted index. For example, it is impossible to search for substrings within words, like *"motor"* in *"motorway"*. An additional problem are different grammatical forms of the same word, like *"car"* and *"cars"*: as the string representations are not equal, those words are not automatically recognized as being equivalent.

To a degree, this can be solved by using stemming algorithms that find the basic form for each word, in all its different grammatical variations [47, 51] – which introduces the problem of discovering the language of a text: stemming algorithms are heavily defendant on the language, and some might require external databases for irregular verbs. The basic, "reduced" form of the word is then added to the index, at the same time reducing the size of the vocabulary. On the other hand, by "reducing" each word to its basic form, some queries might yield incorrect search results, especially when words are used inside phrase queries. The phrase *"I am the king of pop"* is not equal to *"I* be *the king of pop"*.

An other possibility is the implementation of synonym lists for the vocabulary, a *thesaurus*, listing equivalent words. To a degree, such lists can be calculated automatically as shown in LSI (ref. chapter 4.1.1 and [23]).

8.2 Web Content Mining

Although the Web contains a huge amount of data, the representation of information on each web site is different. The fully automated identification of semantically similar data (such as content, navigation links and links to related web sites) is a very important problem with many related publications. Monkville is using a semi-automated tool to extract this kind of information, but there are projects dedicated to the fully automated discovery of semantical types of information on web sites.

Several principal approaches exist for web content mining. First of all, there are fully automated approaches (some are mentioned below), that are suitable for information extraction in a specific problem domain. Some of them perform inductive information extraction, by generating grammars based on training samples. The extraction method used with Monkville is semi-automated, meaning that the information extraction program is created during an user-interactive process (in this case, using the *Lixto Visual Wrapper*). It is also possible to use high-level programming libraries for content extraction from HTML documents. However, such approaches tend to be highly dependant on the web site layout and are therefore not suitable for information extraction based on a wide variety of web sites.

Yin et al. [49] define a web site as a collection of basic elements, with each element having a different semantic role. Their goal is to *"design a system that can classify the elements that make up a web page"* [49], allowing to extract only certain elements when needed, and to facilitate web information extraction.

A similar approach was shown by Liu et al. in [26] who present a very effective technique to determine the type of website components (called *records*). Their classification is based on two different observations about data records on web sites and a string matching algorithm, and is also able to discover noncontiguous data records.

Other methods, as the one shown by Embley et al. [13] are more focused on the hierarchical structure of HTML documents. Records, or record boundaries can be defined via subtrees of nested HTML tags, by locating a subtree containing a records of interest, then identify separator tags within that subtree to determine record boundaries. Embley et al. use several different heuristic algorithms for this identification and then combine the results of the heuristics into one consensus candidate.

8.3 Semantics in Databases

The idea of semantical information in a database is not new. Some publications focus on an explicit representation of semantical information in a database [21], using an object oriented approach to represent facts and relations between them. To a degree, this is very similar to the RDF proposed in the *semantic we*b, only that it is applied on data storages instead of web sites. Using such databases, it is possible to infer over the represented facts, effectively creating a generic expert system.

Semantics can also be exploited to create specific applications, like grouping legal documents based on properties related to specific jurisdiction and different laws in different countries [29], or building retrieval systems for very specialized domains [9] such as patent law or medical information systems.

The representation of semantics in databases is not necessarily bound to purely textual data, but can be achieved by using alternative data storing methods, for example encoding facets in a tree-structured fashion like XML. Neven et al. [36] explored the possibilities to store and operate on attribute grammars for query and tree-walking automata, focused on an XML-based applications. Their investigation of first-order logic expressiveness as well as the complexity of query evaluation and optimization problems showed that, *"although attribute grammars as well as query automata are quite expressive they are quite complicated formalisms and do not seem to be the basis for an easy-to-use pattern language"* [36] that can be implemented in a retrieval system.

8.4 Machine Learning from Natural Language Text

A big problem for automated systems is to make implicit semantic information explicitly known to the computing system [25]. A first step to do this is to remove irrelevant components of the text in a web site, before processing its contents. This can be done by an analysis of web pages that detects the functions of site elements, like navigation links, advertisement, links to related sites and the content itself, as proposed by Yin et al. in [49].

But even with noise elements removed, analyzing natural language text is complicated and requires sophisticated, heuristic algorithms.

8.4.1 Extraction of Semantic Information

Usually, semantic content is extracted on a document level, thereby evaluating single documents only. But there are also other approaches that take the context of web pages into account: as users browse different web sites

and follow links, they follow a *"coherent semantic path"* [42] that can be used to determine context information for web pages, as described by Sreenath et al. [42]. The basic assumption is that an author of a web page cannot completely define its semantics, or semantics emerging through the context in which the web site is being used. Semantics are content-sensitive, and user *browsing paths* over a collection of web sites provide the necessary context to derive semantics, which can be used to improve the retrieval effectiveness of semantics from web pages.

Most systems that determine semantic information from natural language text are based on statistical evaluations of element repetitions and distributions, either looking at global average values of word distributions to aquire the semantic context of single words, or by processing textual data in multiple passes in order to remove noise elements. Latent Semantic Indexing (LSI, see chapter 4.1.1) is an algorithm to derive semantic information from the global text collection. By heuristically evaluating word distributions and proximity it is able to detect word synonymity and polysemy [23, 45]. "Semantic noise" can be filtered by processing documents in multiple passes, and keeping only words and word sequences that appear more than once. This can be done by finding *n-grams* in any text, as shown by Mladenic et al. [32, 33].

8.4.2 Text Classification

Text Classification, often also referred to as Document Clustering, is the challenge to *"discover meaningful groups of documents where those within each group are more closely related to one another than documents assigned to different groups."* [24] The resulting document classification can be used to facilitate the organization of large document collections into semantically related categories.

While there are supervised text classifiers, the goal generally is to create a fully automated, unsupervised classification system. Usually, this is achieved by first training a classification algorithm like the bayesian classifier [33, 19], using a relatively small set of documents and a predefined classification of topics. The problem is to find suitable parameters, based on which the classifier can determine the most significant components of a text. Based on those parameters, the bayesian classification determines the relationship of any document to the given topics.

One parameter that has been used with success in combination with bayesian classification are repeated n-grams in a text [32, 33], which are also used in the Monastery (ref. chapter 6.3.1 and algorithm 2 on page 58). Mladenic et al. showed that by using n-grams for bayesian classification, it is possible to find suitable topic groups with high accuracy for new web sites

– provided that there is a predefined topic ontology and a sufficient set of documents to train the classifier.

A related, but slightly different problem is the automatic definition of topics based on the documents alone, without using a predefined set of possible topics or categories. This adds a new component to the problem of document clustering: determining relevant components of a text, and forming meaningful topic descriptions based on those [8, 11].

8.4.3 Abstract Generation

Automated summarization has received a lot of attention in recent years. [17] Summary Generation is a problem within a similar scope as text classification, but with a different goal. The desired behavior of a good summarization algorithm is to considerably reduce the size of a document, while preserving its content. [20]

Similar to text classification, the algorithms used often require some form of training based on predefined documents and summaries thereof. The problem is to find a good mapping of a high dimensional feature space (given with the documents themselves) onto a much smaller feature space, the summary [10]. To achieve this, noise elements in a text need to be filtered out efficiently through a combination of statistical methods and filtering algorithms (like n-grams). The key features (content) of the documents need to be preserved by their summaries [20], even though they usually are much shorter.

Han et al. [17] propose a summary generation based on "important words", that is, key terms in a document that were evaluated as being important by some form of semantical analysis. The extraction system generates summary sentences according to the word meanings, using a variety of evaluations to determine the importance of a word.

A similar proposal was made by Liu et al. [27] who focus on non-content features of key resources and introduce a pre-selection method, to locate key resources on a page using a probabilistic decision tree, using independent features including document length, url type and the amount of distinct words.

8.5 The Semantic Web

The Semantic Web is the vision of a global information network, which is linked in such a way that it is easily processable for machines. The goal is to create a global database of explicit semantic data [4], that usually is given only implicitly within natural language text, encoded in HTML pages. Facts

and relations are represented in triples of URIs, which are used to uniquely identify entities. The syntax is called *"Resource Description Framework"* (RDF) and can be given in various well defined forms. Facts and relations between them are described in an ontology description language, called RDF Schema [37].

RDF is a data description language that allows to define classes and relations. There are some predefined basic ontology languages based on RDF, which are already established by the semantic web task force. They are mostly tailored for specific object domains, like *DAML+OIL*[30] or *OWL*[31]. They define the *vocabulary* to describe objects, facts, properties and relations between them – for example facets like *"Snoopy is a dog, and belongs to Charly Brown"*. Relations like cardinality ("exactly one" – *"Snoopy is exactly this dog, and no other one"*), equally and symmetry can be expressed with the help of OWL and other ontology description languages.

The intent of the semantic web is to enhance the usability and usefulness of the current world wide web and its interconnected resources, by providing some common ontologies and class descriptions that can be used by a wide number of independent web sites, in different contexts. Based on the used fact representation ontologies, it is possible to perform automated reasoning over facts, even if they are represented in different ontology languages and contexts.

Documents *"marked up"* with semantic information contain machine-readable information about the human-readable content of the document, such as title, author, abstract – or it could be purely metadata, representing a set of facts like resources and services. Common ontologies (metadata vocabularies) describe how to mark up documents so that the marked up information can be processed by computers [46].

A popular application of the Semantic Web is Friend of a Friend (or FoaF), which describes relationships among people and other agents in terms of RDF.

[30] DAML: the *DARPA agent markup language*, OIL: *ontology inference layer*. DAML+OIL combines the features of both

[31] OWL: acronym for *Web Ontology Language*, a markup language for publishing and sharing data using ontologies on the semantic web

9 Summary

The ever growing internet and the amount of textual data that is available on the web present a problem which is often referred to as *information overflow*: due to the large number of web sites, it is hard to locate sites that cover a specific topic. To solve this problem, automated full text search engines need to be implemented that help to navigate in the internet.

This can be achieved by implementig *document retrieval systems* – databases specialized on the processing of large amounts of text, with the possibility to perform high performant searches on all stored documents, and with the ability to evaluate the relevance of the documents to the search query. The core component of any document retrieval system is the text indexing method used, which also defines the functionality of the system to a very large degree. Several indexing methods were explained in chapter 2, with the subsequent focus on the *inverted index* due to its superior properties.

An implementation of a document retrieval system on very large text collections requires a lot of thought, as the problem of full text search is not trivial to solve. The large amount of data that needs to be processed presents problems of its own, as explained in section 2.4. The indexing method is the most important characteristic property of any retrieval system, it defines the overall speed, ressource requirements and system features. The concept of the inverted index, as explained in chapter 3, shows good performance and the highest degree of flexibility – which is the reason why it is usually chosen for text retrieval implementations. It also supports a lot of semantical text evaluations that are necessary for result ranking, as explained in chapter 4.

The practical implementation of a document retrieval system, as presented with *Palaver* in chapter 5, revealed a series of problems to which the literature does not present any solutions. Those problems are for example related to the actual representation of textual data itself, which might be given in multiple different languages, scripts and encodings. Also, more fundamental problems like finding a good definition for *one word*, that also works with multiple languages and scripts, need to be solved. Technial issues, like finding a good mapping function to calculate an unique identifier for each distinct word, usually are not even mentioned in relevant publications.

Text processing systems are inherently limited in the degree to which they are able to grasp the semantic concepts of text in a natural language [25]. Some existing approaches to determine semantic properties of documents do exist, but this fundamental limitation prevails.

Monkville demonstrates a new approach to increase the quality of semantic processing in a document retrieval system. By adding external context information to the full text database, the quality of semantic evaluations can

be increased. Chapter 6 illustrates the basic concept used to improve search functionality as well as semantical analysis of the documents, resulting in a wide variety of new possibilities for full text search on web sites.

The result is a content-based navigation system that can be used to navigate through the world wide web, based not on hyperlinks or keyword searches, but on semantic content. Sites that do not refer to each other by placing hyperlinks suddently can become linked semantically, because they have similar content.

The shown approach does not require fact extraction or any inference on logic relations, as for example in applications of the semantic web. It rather uses existing utilities to extract semantically related context information from semi-structured sources, like from news on web sites. Further work remains to explore the degree and quality of the semantical relationships that can be detected by including such context information, as well as ways to effectively navigate in a semantically linked world wide web.

The presented implementation still leaves room for improvements. Further work will have to focus on increasing the performance of data storages and disk throughput. An additional focus will have to remain on additional sources of high quality semantical information. With the semantic web, new sources of context information could become available, but ways to exploit the vast amount of semantic data that already is available need to be explored too. Where original semantic content is not present, it could be created by semi or fully automated text processing or information extraction systems.

The questions that remain to answer are not only the ways how to store semantic information in an inverted index, but the origin and quality of the semantics themselves. Sites that contain semantical relevant context information do not use a representation that is supposed to be interpreted by computing systems, but by human readers. Finding suitable techniques to discover such information and to filter out irrelevant web site components will be a very important topic for the future, not only in the scope of this project.

Once it is generally available, semantic context information can have a great impact on the quality of internet search engines.

A Database Manual

Startup, Shutdown and Help

1. Start the program (f.i. PALAVER.EXE on Windows)
2. Connect with a terminal program to local port `1200`. If possible, disable telnet control sequences and use a raw connection. You can use any number of concurrent client connections.
3. Once connected, the database should reply with "`Hello, I am ready`" to the client, the main program should write a message to the console reading "`Client accepted: <hostname>`"
4. At any point, enter "?" or any command followed by "?" to get a list of available commands, parameters or short usage information
5. Enter `shutdown` to terminate the program

Adding Documents

1. To add one specific file with one specific URI only, enter:
   `add myDocumentURI path/to/file.txt`
2. To recursively add the contents of a directory, enter:
   `add myURIprefix path/to/directory/` (note the trailing `/`)
   The string "myURIprefix" will be prefixed to the full path names to the document files. The combination of the URI prefix and the path name will be the document URI.
3. To process context information, put all context keywords for one document into a raw text file, with the same filename as the original text file, but with `.context` appended to it. For example, context keywords for `/data/text.txt` would go into `/data/text.txt.context`

Querying

All queries are case insensitive.

1. `search <term> [additional terms]`
   Performs a full-text search in document contents (bodies) for all terms passed as arguments. Unless specified otherwise (see below), terms are logically OR-ed and the result is the sum of all documents that contain one or more of the search terms.

2. `searchtopic <term> [additional terms]`
   Same as `search`, but performs a full-text search in document *contexts* and *signatures* only, not in document bodies.

3. `query <term> [additional terms]`
   `[ CTX[+|-] <contextterm> [additional contextterms] ]`
   Search in document bodies and/or context and signatures. query terms before `CTX` are used for document content search, the terms after are used for context search. The optional `+` or `-` character after `CTX` follows the boolean syntax described below and can be used to include or exclude documents with a specific context.

Boolean Query Syntax

Search terms can be prepended by one optional `+` or `-` each. If neither `+` nor `-` are given, the system will perform a logical OR operation on the query result, thus merging all resulting documents for sub-queries.

List of examples of parameters for the `query` commands:

`peach apple`	results contain "peach" **OR** "apple"
`peach +apple`	results contain "peach" **AND** "apple"
`peach -apple`	results contain "peach" **BUT NOT** "apple"

Phrase queries

Word sequences that are part of the document signature (that is, sequences that appear at least two times in a document) can be used in phrase queries. To search for a phrase, enclose all words that are part of that phrase in quotes (`"`). For example,

```
query "I am the king of pop" CTX+ music
```

Returns all documents that contain the phrase "I am the king of pop" AND have "music" in their context. To use boolean operators with phrases, insert the `+` nor `-` character *after* the quote. Thus:

```
query Jackson "+I am the king of pop"
```

to search for documents that contain *"Jackson"* AND the phrase *"I am the king of pop"*.

List of Commands

- `load <database>`
  Load the full text database from disk
- `save <database>`
  Save the full text database to disk
- `stat`
  Display database status and statistical values
- `shutdown`
  Terminate the database and disconnect all clients
- `add`
  Alias for `mercator add`
- `mercator add <URI> <file/dir>`
  Add one or multiple documents to the database. If the last character of the filename is a directory separator (`/`), it is assumed to be a directory, and adding will happen recursively within that directory.
- `search`
  Alias for `librarian search`
- `librarian search <word> [ [+|-]additional words ] ...`
  Perform a full text search for search words within document bodies only, optionally with boolean operators
- `searchtopic`
  Alias for `librarian searchtopic`
- `librarian searchtopic <word> [ [+|-]additional words ] ...`
  Perform a full text search for search words within document contexts only, optionally with boolean operators.
- `query`
  Alias for `librarian query`
- `query <term> [ [+|-]additional terms] ... [ CTX[+|-] <contextterm> [ [+|-]additional contextterms] ... ]`
  Search in document bodies and/or context and signatures. query terms before `CTX` are used for document content search, the terms after are used for context search. The optional `+` or `-` character after `CTX` follows the boolean syntax.

- `related`
  Alias for `librarian related`

- `librarian related <documentURI>`
  Find documents that are semantically related to the document referenced by `documentURI`

B Figures, Tables and References

List of Figures

List of Tables

References

[1] Mysql 5.0 reference manual, section 12.7: Full-text search functions. *http://dev.mysql.com/doc/refman/5.0/en/fulltext-search.html.*

[2] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. In *VLDB '01: Proceedings of the 27th International*

Conference on Very Large Data Bases, pages 119–128, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[3] R. Baumgartner, G. Gottlob, M. Herzog, and W. Slany. Interactively adding web service interfaces to existing web applications. In *SAINT*, pages 74–80. IEEE Computer Society, 2004.

[4] B. Berendt, D. Mladenic, M. van Someren, M. Spiliopoulou, and G. Stumme. A roadmap for web mining: From web to semantic web. In *EWMF*, pages 1–22. Springer, 2003.

[5] T. Berners-Lee. Primer: Getting into rdf & semantic web using n3. *http://www.w3.org/2000/10/swap/Primer*, 2005.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. Technical report, Stanford University, Computer Science Departement, 1998.

[7] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB*, pages 192–202. Morgan Kaufmann, 1994.

[8] H. Chen and V. Dhar. A knowledge-based approach to the design of document-based retrieval systems. In *Proceedings of the conference on Office information systems*, pages 281–290. ACM Press, 1990.

[9] L. Chen, N. Tokuda, and H. Adachi. A patent document retrieval system addressing both semantic and syntactic properties. In *Proceedings of the ACL-2003 Workshop on Patent Corpus Processing*. ACL, 2003.

[10] W. Chen, X. Chang, H. Wang, J. Zhu, and Y. Tianshun. Automatic word clustering for text categorization using global information. In Myaeng et al. [35], pages 1–11.

[11] J. J. Chua and P. E. Tischer. Hierarchical ordering for approximate similarity ranking. In N. Zhong, Z. W. Ras, S. Tsumoto, and E. Suzuki, editors, *ISMIS*, volume 2871 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2003.

[12] J. Dvorský, J. Pokorný, and V. Snásel. Word-based compression methods and indexing for text retrieval systems. In J. Eder, I. Rozman, and T. Welzer, editors, *ADBIS*, volume 1691 of *Lecture Notes in Computer Science*, pages 75–84. Springer, 1999.

[13] D. W. Embley, Y. Jiang, and Y.-K. Ng. Record-boundary discovery in web documents. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 467–478, New York, NY, USA, 1999. ACM Press.

[14] C. Goutte and É. Gaussier. A probabilistic interpretation of precision, recall and -score, with implication for evaluation. In D. E. Losada and J. M. Fernández-Luna, editors, *ECIR*, volume 3408 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2005.

[15] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *STOC*, pages 397–406, 2000.

[16] P. Gulutzan. The full-text stuff that we didn't put in the manual. *http://dev.mysql.com/tech-resources/articles/full-text-revealed.html*.

[17] D. Han, T. Noguchi, T. Yago, and M. Harada. Summary generation centered on important words. In Myaeng et al. [35], pages 48–60.

[18] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *JASIST*, 54(8):713–729, 2003.

[19] C.-M. Hung and L.-F. Chien. Text classification using web corpora and em algorithms. In Myaeng et al. [35], pages 12–23.

[20] W. Jung, Y. Ko, and J. Seo. Automatic text summarization using two-step sentence extraction. In Myaeng et al. [35], pages 71–81.

[21] N. Katayama, A. Takasu, and J. Adachi. A database with an explicit semantic representation. In *Proceedings of the 7th international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 323–332. Gordon and Breach Science Publishers, Inc.

[22] A. J. Kent, R. Sacks-Davis, and K. Ramamohanarao. A signature file scheme based on multiple organizations for indexing very large text databases. *JASIS*, 41(7):508–534, 1990.

[23] M. Kobayashi and K. Takeda. Information retrieval on the web. *ACM Computing Surveys*, 32(2):144–173, 2000.

[24] C. Kruengkrai, V. Sornlertlamvanich, and H. Isahara. Document clustering using linear partitioning hyperplanes and reallocation. In Myaeng et al. [35], pages 36–47.

[25] D. D. Lewis. Natural language processing for information retrieval. *ACM*, 39(1):92–101, January 1996.

[26] B. Liu, R. Grossman, and Y. Zhai. Mining data records in web pages. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–606, New York, NY, USA, 2003. ACM Press.

[27] Y. Liu, M. Zhang, and S. Ma. Effective topic distillation with key resource pre-selection. In Myaeng et al. [35], pages 129–140.

[28] Y. Lu, X. Liu, H. Li, B. Zhang, W. Xi, Z. Chen, S. Yan, and W.-Y. Ma. Efficient pagerank with same out-link groups. In Myaeng et al. [35], pages 141–152.

[29] R. Álvarez, M. Ayuso, and M. Bécue. Statistical study of judical practices. In *Law and the Semantic Web*, pages 25–35, 2004.

[30] V. Mäkinen. Compact suffix array – a space-efficient full-text index. *Fundam. Inform.*, 56(1-2):191–210, 2003.

[31] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *WWW*, pages 396–406, 2001.

[32] D. Mladenic and M. Grobelnik. Word sequences as features in text-learning.

[33] D. Mladenic and M. Grobelnik. Mapping documents onto web page ontology. In *EWMF 2003, LNAI 3209*, pages 77–96. Springer-Verlag Berlin Heidelberg, 2004.

[34] A. Moffat and J. Zobel. What does it mean to measure performance? In X. Zhou, S. Y. W. Su, M. P. Papazoglou, M. E. Orlowska, and K. G. Jeffery, editors, *WISE*, volume 3306 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2004.

[35] S.-H. Myaeng, M. Zhou, K.-F. Wong, and H. Zhang, editors. *Information Retrieval Technology, Asia Information Retrieval Symposium, AIRS 2004, Beijing, China, October 18-20, 2004, Revised Selected Papers*, volume 3411 of *Lecture Notes in Computer Science*. Springer, 2005.

[36] F. Neven and T. Schwentick. Automata-and logic-based pattern languages for tree-structured data. In L. E. Bertossi, G. O. H. Katona, K.-D. Schewe, and B. Thalheim, editors, *Semantics in Databases*, volume

2582 of *Lecture Notes in Computer Science*, pages 160–178. Springer, 2001.

[37] S. B. Palmer. The semantic web: An introduction. *http://infomesh.net/2001/swintro/*, 2001.

[38] B. A. Ribeiro-Neto, J. P. Kitajima, G. Navarro, C. R. G. Sant'Ana, and N. Ziviani. Parallel generation of inverted files for distributed text collections. In *SCCC*, pages 149–157. IEEE Computer Society, 1998.

[39] R. Sacks-Davis, A. J. Kent, K. Ramamohanarao, J. A. Thom, and J. Zobel. Atlas: A nested relational database system for text applications. *IEEE Trans. Knowl. Data Eng.*, 7(3):454–470, 1995.

[40] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR*, pages 222–229. ACM, 2002.

[41] S. M. Shafi and R. A. Rather. Precision and recall of five search engines for retrieval of scholarly information in the field of biotechnology. *Webology, available at http://www.webology.ir/2005/v2n2/a12.html*, 2(2), 2005.

[42] D. V. Sreenath, W. I. Grosky, and F. Fotouhi. Using coherent semantic subpaths to derive emergent semantics. In M. G. Negoita, R. J. Howlett, and L. C. Jain, editors, *KES*, volume 3215 of *Lecture Notes in Computer Science*, pages 173–179. Springer, 2004.

[43] R. K. Stasiu, C. A. Heuser, and R. da Silva. Estimating recall and precision for vague queries in databases. In O. Pastor and J. F. e Cunha, editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 187–200. Springer, 2005.

[44] A. Tomasic, H. Garcia-Molina, and K. A. Shoens. Incremental updates of inverted lists for text document retrieval. In R. T. Snodgrass and M. Winslett, editors, *SIGMOD Conference*, pages 289–300. ACM Press, 1994.

[45] B. Trevor, E. Weippl, and W. Winiwarter. A modern approach to searching the world wide web: Ranking pages by inference over content. In *INAP*, pages 316–330, 2001.

[46] Wikipedia. The semantic web. *en.wikipedia.org/wiki/Semantic_web*, 2005.

[47] H. E. Williams and J. Zobel. Searchable words on the web. *International Journal of Digital Libraries*, to appear.

[48] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[49] X. Yin and W. S. Lee. Towards understanding the functions of web element. In Myaeng et al. [35], pages 313–324.

[50] B. Yuwono and D. L. Lee. Search and ranking algorithms for locating resources on the world wide web. In S. Y. W. Su, editor, *ICDE*, pages 164–171. IEEE Computer Society, 1996.

[51] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.

[52] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full text databases. In L.-Y. Yuan, editor, *VLDB*, pages 352–362. Morgan Kaufmann, 1992.