Design and Implementation of a Content-Based Search Engine

Ville H. Tuulos

Helsinki May 26, 2007

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Ville H. Tuulos

Työn nimi — Arbetets titel — Title
Design and Implementation of a Content-Based Search Engine

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	May 26, 2007	93

Tiivistelmä — Referat — Abstract

This thesis presents a system to find interesting textual content among tens of millions of documents. This is made possible by a novel content-based ranking method and a simple, structured query interface, which are presented in this thesis. The ranking method allows the user to utilize the full co-occurrence matrix of all words in the corpus to bring out relevant material. The user may explicitly define her conception of relevance by guiding the ranking with single words.

This thesis presents the design and implementation of the system. The basic formulation of the content-based ranking method is computationally rather expensive and therefore also an efficient algorithm is given. The index structures of the system have been specifically designed to support the ranking scheme. The system is distributable to a cluster of servers, allowing reasonable scalability.

We present three real-world deployments of the system. The largest of the deployments was a publicly available Web search engine, Aino, which covered over four million pages in the .FI domain.

ACM Computing Classification

H.3.1 Content Analysis and Indexing
H.3.3 Information Search and Retrieval
I.2.7 Natural Language Processing

Avainsanat — Nyckelord — Keywords
Text processing, Statistical information retrieval, Language models, Search engines

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library, serial number C-

Muita tietoja — övriga uppgifter — Additional information

Contents

List of Algorithms

List of Figures

List of Tables

Acknowledgements

Many people have provided help and inspiration for this thesis. First, I would like to thank Professor Henry Tirri for challenging me to develop a search engine in the first place and Professor Petri Myllymäki for his support throughout the process. I would like to thank my colleagues in the Complex Systems Computation Group, especially the following three people on our IRC channel `#finni`: Special thanks go to Antti Tuominen for HooWWWer and his 7517 supportive comments, Tomi Silander for his 3627 insightful comments, and Pekka Tonteri for building necessary infrastructure and helping me with his 3872 comments. Last but not least, a huge hug to my wife Heli Tuulos for helping me with the math.

1 Introduction

This thesis presents an efficient design and implementation of a content-based search engine. *Content-based* means that the system utilizes information available in the documents in a holistic manner to determine what might be interesting to the user. We focus on textual content that is written in a natural language as opposed to, say, images included in the documents. We call the presented system a *search engine*, as it contains components to retrieve and index documents, and it provides a mechanism to return a ranked subset of the documents according to the user's requests. By *efficient*, we mean that the system should be able to process millions of documents in a reasonable time and respond to queries with a low average latency.

This thesis consists of four main contributions:

- Design and implementation of a full-fledged search engine, Aino,
- a novel content-based ranking method, AinoRank,
- an efficient algorithm that implements the previous,
- and several applications of the system to real-world document collections.

A brief overview of relevant prior work is given in section 2. Then, Chapter 3 presents the ranking method, AinoRank. Chapter 4 introduces the architecture and the main elements of our system as well as several algorithms that implement the previous ranking method. Finally, Chapter 5 gives a summary of some real-world applications of the system. Chapter 6 concludes the thesis.

The starting point of this thesis is based on my personal background. I have been interested in natural language processing since I was fifteen. In those days, Neural Networks Research Center at the Helsinki University of Technology released the first demonstrations of the WEBSOM method [KHLK98], which visualizes document

collections using a model called Self-Organizing Map [Koh01]. I found this intriguing and soon I got an opportunity to take these ideas forward in a spin-off company. In this context, I helped to design and implement a content-based information retrieval platform [HT02], which was the first of its kind for me.

In 2003 I joined the Complex Systems Computation Group at the University of Helsinki. The group had a strong background in statistical and information-theoretic modeling, which matched well with my interests. The group was applying an advanced statistical model, Multinomial Principal Component Analysis (MPCA) [BJ06], to content-based search with promising results. We produced some nice demonstrations [TT04, BLP+04] which, however, raised new questions in my mind.

Two things bothered me both in WEBSOM and MPCA. First, I felt helpless with opaqueness of the methods. Both methods relied on certain elaborate theoretical assumptions that were not directly related to the modeled phenomenon, namely language. I did not feel comfortable with the exploratory "let's change some parameters and see what happens" approach. The problem of opaqueness was reflected also in the user interfaces. It was hard to tell exactly how to get desired results and in case of unsuccessful queries, how to improve them. I started to feel that natural language is a phenomenon so complex that we should avoid adding to this complexity in our models – at least until we understand the underlying data better.

The second unanswered question was about losing information. Practically all sophisticated methods in information retrieval seem to take as granted that most of the tokens do not carry any important information or they are redundant. Often features that are considered unimportant are thrown away either by explicitly using some stopword lists or by frequency limits, or by using some automatic feature selection or dimensionality reduction method. I could not see how one could decide what should be thrown away *a priori*, before seeing any queries. Naturally we could base the decision on some statistical measure but often the choice felt unjustified.

I felt that before I could continue with more sophisticated methods, I needed to understand the basics better. I wanted a system that would be transparent and whose inevitable fallacies could easily be circumvented by the user. Also I wanted a system that would not make any unjustified decisions before seeing what the user considers important in her queries. And due to my personal sense of aesthetics, I wanted a rather minimalistic, but robust and efficient implementation.

2 Prior Work

Aino's legacy comes from many sources. Firstly, there is the decades-long tradition of information retrieval. Secondly, there is almost an equally old tradition of statistical modeling of language. Finally, there is a rather recent trend of producing free search engines.

In the following, we first give an overview of different ranking methods. After this, we present some free search engines and information retrieval packages. The presented systems vary from drop-in search solutions to complex academic testbeds of state-of-the-art information retrieval methods. This selection is deliberate, as Aino falls somewhere in between these two extremes. To justify this claim, we present a rough comparison between the presented systems and Aino in the end of the section.

2.1 Ranking Methods

Consider that you are given a corpus of documents. Then you get a query consisting of one or more words. Now your task is to order documents in the corpus according to their relevance with respect to the query. This operation is commonly called *ranking*.

Besides ordinary words, in many cases the document corpus contains some auxiliary information that can be utilized in ranking. As of 2007, the three most popular Web search engines, Google [Goo07], Yahoo [Yah07], and MSN Live [MSN07], utilize hyperlinks in their ranking algorithms. A well-known example of a link-based ranking algorithm is Google's PageRank [PBMW98].

An approach based on hyperlinks is justified in the Web, as statistically hyperlinks tend to point at the content that many people find interesting [PBMW98]. Similarly to hyperlinks, other kinds of structures within the document can be utilized in

		Query type		
		Keyword	**Example**	**Natural Language**
Corpus type	**Hypertext**	Google, Yahoo	More Similar Pages by Yahoo, Google	Ask.com, QuASM
	Structured	XIRQL	SQL-QBE	Precise, NALIX
	Plain Text	Lucene, Aino	WEBSOM, Ydin	Various QA systems

Table 1: Different ranking approaches

ranking as well. For instance, some ranking algorithms such as [LYJ05, FG04], can utilize any structure encoded in XML.

Not only the corpus, but also the query may come in different forms. A typical Web search engine accepts a short list of keywords with some modifiers, such as quotes to denote phrases. In contrast to earlier systems, which required explicit Boolean operators to structure the query, current systems often rely on implicit, algorithm-specific heuristics to decide how to handle multiple keywords.

Some systems provide a function like "Query by Example" (QBE) or "More Similar Documents". In these systems, a short excerpt of text or even a full document may be used as a query. Some sophisticated systems let the user input the query or a question in natural language, e.g. in plain English.

Table 1 presents a survey of some existing systems that utilize different types of queries and corpora for ranking. Let us start with the systems based on hypertextual content: If the query interface is based on keywords, the system is similar to contemporary Web search engines, such as Google and Yahoo. Web search engines often provide a function called "similar pages", which resembles Query by Example functionality. In this case, however, the similarity measure is based on the topology of the link-graph and not on the textual content.

`Ask.com` is a major Web search engine that lets the user search for pages in plain English. The results are ranked with a link-based ranking scheme, similarly to keyword-based Web search engines. A different kind of approach is taken by the QuASM system [PBC+02] that uses HTML tables to find answers to questions given in plain English.

Nowadays majority of research on structured information retrieval focuses on XML and Semantic Web technologies; for an overview, see [Leh06]. Earlier, a substantial amount of related research focused on designing various extensions to the Simple Query Language (SQL). XQuery [XQu07] is a standard query language for retrieving different parts of an XML document. On top of this, various layers have been proposed to support keyword search, for instance, XIRQL [FG04].

Domain Relational Calculus [LP77] is an early example of a method that allows querying of structured (relational) data by example. It is followed by many query by example systems for SQL, such as the IBM's QBE [RG02]. Roughly speaking, these systems let the user specify a template or a set of constraints that the results must satisfy. Also, several research prototypes exist for compiling natural language queries to SQL, e.g. with Precise [PEK03] and to XML queries e.g. with NALIX [LYJ05].

Content-Based Information Retrieval is a vague concept, which is used to refer to various information retrieval settings and approaches. Often the concept refers to retrieval of non-textual data, such as images or videos [BR99]. In the literature, content-based queries are brodaly defined as "queries exploiting data content" [BR99].

Along these lines, we can interpret that content-based refers to systems that utilize the bulk content of the corpus in a holistic manner, instead of relying on some specific features of the data, such as links, tables, or other structural elements. According

to this interpretation, the last row in table 1 can be seen to refer to content-based systems.

The work presented in this thesis belongs to the category characterized by plain text corpora and keyword queries. Many other academic or open source search engines, such as the ones presented in the following section, belong to this category. Some ranking schemes benefit directly from longer queries that contain more information about the user's interests – in this case the system may work with single keywords but the best results are produced when a long example document is given as the query: Systems of this kind include WEBSOM [KHLK98] and many other system based on language modeling approach [PC98], such as Ydin [BLP+04]. Many traditional question-answering (QA) systems belong to the category on the lower right corner. For recent examples, see publications in the TREC Question Answering track [VB06].

An important characteristic missing from Table 1 is how the query actually affects ranking. Some algorithms, such as PageRank [PBMW98], assign a static score to each document independently from the query. The main benefit of this approach is that query processing becomes extremely efficient as the query is only needed to determine the matching subset of documents that are already ranked. We call this approach *static ranking*.

An alternative approach is to re-rank the corpus against every query – we call this *dynamic ranking*. This approach is more flexible, as it does not fix the ranking before seeing any query, but the computational cost may be high. In the following chapter, a novel dynamic ranking approach is presented.

2.2 Existing Systems

Considering the vast number of possible interpretations of "search" and the ease of implementing something resembling search, it is understandable that there is a plethora of software packages available for this purpose.

However, implementing a truly *scalable* search engine is not so trivial. We restrict the discussion to systems that claim to scale to millions of documents. Also, we require that the system should be rather complete, so an efficient implementation of, say, a bare inverted index is not enough. Furthermore, the system should not be domain-specific, which often allows remarkable optimizations by sacrificing generality. Instead, we are interested in systems that are aimed at indexing and querying unspecified, mostly unstructured corpora of natural language.

Comparing implementations is often rather unfruitful. The most relevant comparison deals with the results or the perceived quality of search. This is a deep subject, which is covered for instance by the TREC text retrieval competitions [VB06]. Comparing scalability or performance is difficult as well, since any attempt to index a huge number documents requires detailed tuning of the system, which in turn requires intimate knowledge of the implementation at hand. Comparing the performance of query processing is even more difficult, since each system balances the trade-off between ranking quality and speed differently.

However, by comparing systems we may clarify our own position. This is the main motivation for the following section, which will briefly introduce some systems resembling ours and their relationship to this work.

Ht://Dig

The first version of ht://Dig [Ht:95] was released already in 1995, which makes it one of the oldest software packages for searching web pages. Ht://Dig consists of

a collection of small command-line tools for indexing and querying web pages. It is mainly aimed at individual web sites, but technically it should be applicable to small intranets as well, given that the documents are suitably organized on disk. Ht://Dig is implemented in C++.

The system provides a simple query interface based on Boolean ranking [BR99] and a variation of it using approximate keyword matching. Although Ht://Dig was popular during its early days, its development has been ceased for years and it is surpassed by more modern systems. Despite the system's lack of technical or academic merit, its simple Unix-style architecture provided some inspiration for Aino.

Lucene

As of 2007, Lucene [Cut97] is the most widely used open-source search package. Lucene includes an efficient indexing mechanism and facilities for building highly scalable search engines, including a Web crawler Nutch. The main implementation is in Java, although partial ports to various programming languages exist. According to the project's web site, Lucene's ranking method is "a combination of the Vector Space Model (VSM) of Information Retrieval and the Boolean model to determine how relevant a given Document is to a User's query".

Developers are encouraged to integrate the Lucene's object-oriented architecture into their own search interfaces. Also, the architecture makes it possible to implement missing functionalities behind the given interfaces. In contrast, the design and implementation of Aino was dictated by our data-intensive ranking method, which required careful attention to performance.

Lemur and Indri

Lemur is a C/C++ toolkit for language modelling, which forms the basis for the search engine Indri [MC04]. They have been developed in a joint project by the Computer Science Department at the University of Massachusetts and the School of Computer Science at Carnegie Mellon University. In contrast to the above systems, they are targeted at the academic community needing a testbed for advanced information retrieval algorithms. Lemur / Indri supports a versatile INQUERY query language [CCH92] and ranking using inference networks [MC04].

Lemur's strong focus on statistical language modelling and its success in the TREC competition [VB06] make it inspirational for Aino.

Terrier

Terrier is a full-fledged search engine and information retrieval platform by the information retrieval research group at University of Glasgow [OAP+06]. Terrier implements various novel probabilistic ranking algorithms, such as the *Divergence from Randomness* model [AVR02]. It is targeted both at practical applications and academic research.

In the Aino's point of view, Terrier belongs to the same category of large-scale probabilistic information retrieval research platforms as Lemur. Terrier is implemented in Java.

Comparison

The presented systems are motivated by different goals, which are deeply reflected in the design of each system. However, some basic tasks are common to all the systems: Namely, they have to read input documents, perform some preprocessing

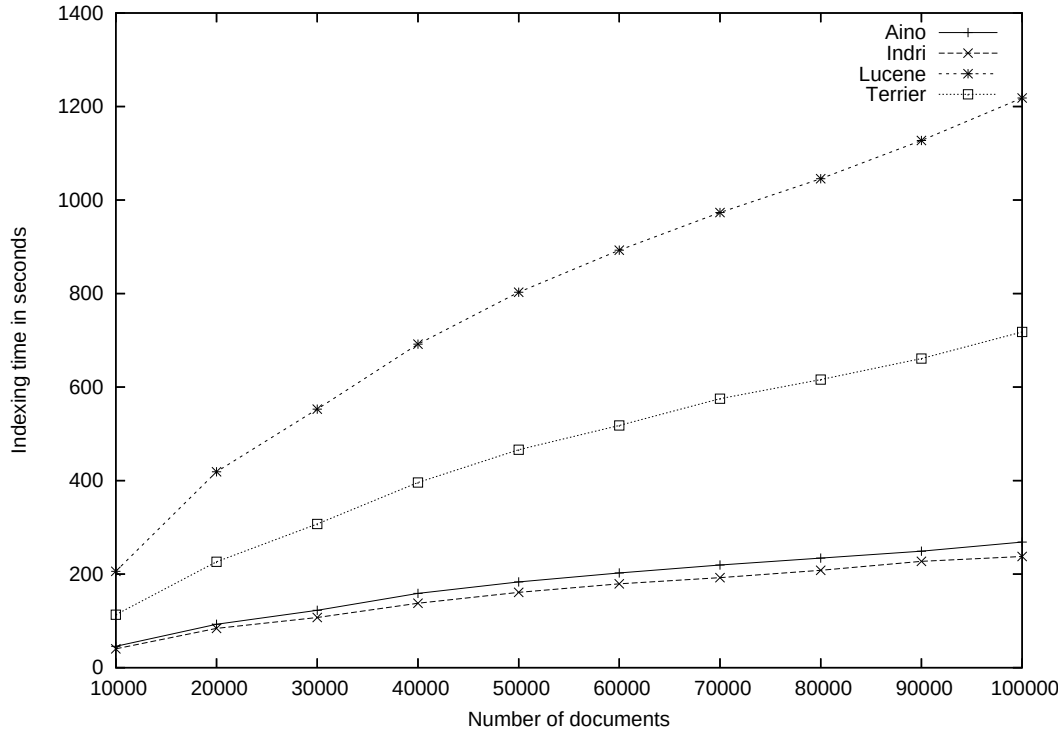

Figure 1: Indexing times with respect to size of the corpus

tasks, and construct an index.

Being efficient in these tasks is mostly a matter of careful engineering. In theoretical point of view, these tasks are more or less trivial. However, in terms of scalability this phase is essential: The system must achieve high indexing throughput to be able to handle millions of documents in a reasonable time. At the same time, the system must produce efficient index structures so that latencies in the query processing phase will be low.

We compared raw indexing performance of the presented systems. Unfortunately Ht://Dig had to be excluded from the benchmark due to its exceptional slowness; processing of the smallest data set took more than eight minutes. This might be due to the fact that it is designed to receive its input from an integrated Web crawler that is remarkably suboptimal in local environments.

The test collection consisted of 100,000 documents from Wikipedia, the free en-

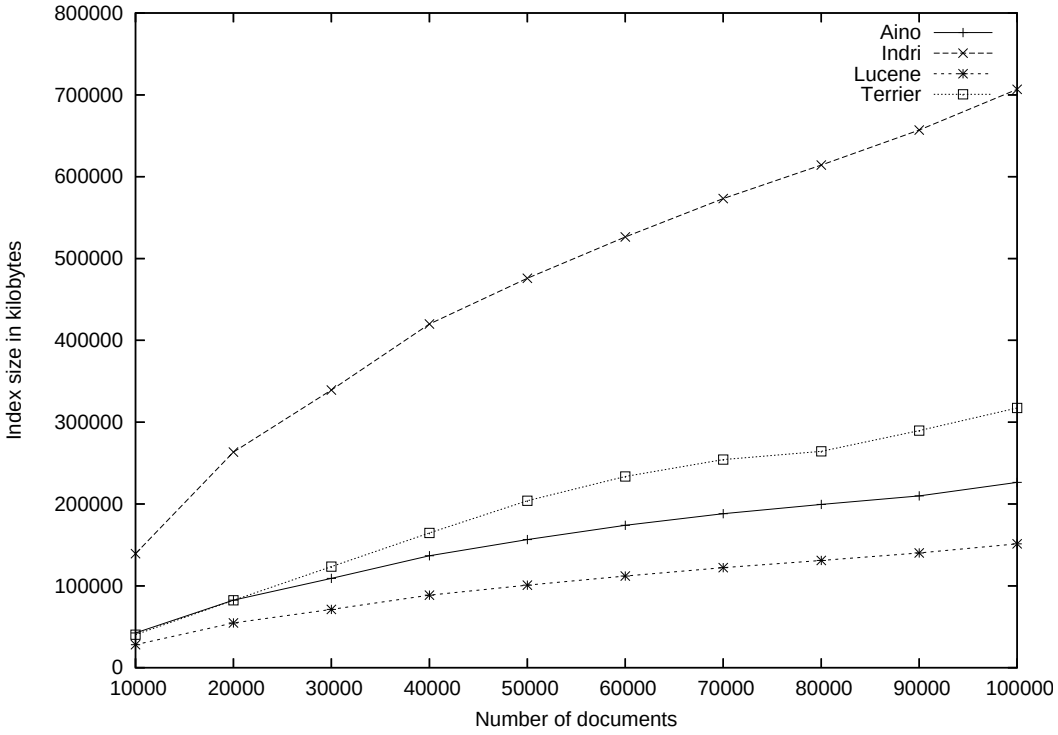

Figure 2: Index sizes with respect to size of the corpus

cyclopedia [Wik07]. We tested the indexing performance with varying number of documents. The results are shown in Figure 1. Each system was run with comparable default parameters. It is clear that the systems may be tweaked to achieve higher throughput but it should be reasonable to assume that Figure 1 depicts the overall trend. Here, Aino and Indri seem to scale rather logarithmically and they outperform the other two systems. This might be due to differences between C/C++, the language of choice for Aino and Indri, and Java, which is used by Terrier and Lucene.

We also measured sizes of the resulting indices; the results are shown in Figure 2. One should be extremely careful in interpreting the results, since the systems do not index exactly the same types of information. In this case, it is reasonable to believe that Aino, Terrier, and Lucene index approximately the same amount of information and their mutual differences are caused by different encoding methods.

Aino performs well in the comparison. It shows modest linear behavior with respect to the number of documents and a good balance between the indexing speed and the index size. Chapter 4, which describes the implementation, will give some reasons for this.

As mentioned earlier in this section, macro-scale benchmarks of this kind are often unfruitful and unfair. Due to modular architecture of each system, it should not be difficult to replace inefficient algorithms with efficient ones, at least in theory. Moreover, most of the systems focus on improving the ranking quality, not efficiency of indexing. Thus, one should not underestimate quality of the other systems based on this comparison but it should be interpreted as a further motivation for the design and implementation of Aino instead.

3 AinoRank

This chapter forms the core of this thesis. We start with our desiderata for content-based ranking. Then, Section 3.2 introduces the theoretical basis for the method. Section 3.3 formulates the actual method step by step, starting from a simplified case. We then proceed to explain how the method appears to the user in Section 3.4. Finally, we bring all the pieces together in Section 3.5 that gives the final formulation for AinoRank.

3.1 Design Criteria

AinoRank was motivated by the following design criteria: We need a content-based ranking algorithm that is *usable*, *predictable*, *robust*, and it should *scale* to millions of documents. In the following, we provide background for each of the criteria.

Users of Web search engines seldom browse the search results beyond the top ten hits. Based on 15,000,000 clicks on search results in the AOL Web search logs [AOL06], we know that the first result is 3.5 times more popular than the second, which in turn is four times more popular than the tenth result. In total, the top ten results account for 90% of all clicks in the data and the top-100 for 99% of them.

Thus, an important **usability** goal for our ranking algorithm is to get the most relevant documents to the top ten set. However, the user's conception of relevance, i.e. what the user feels important, is often weakly mediated in the query: The query may carry little information (e.g. "cat"), it may be broad ("foreign politics"), or ambiguous ("apple" – computer or fruit?). Yet in all these cases we can assume that the user has a specific information need, although it is not well articulated in the query. In order to be truly usable, the system should take into account this discrepancy between the true information need and the actual query, while

respecting the user's desire to use only minimum effort to formulate the queries.

It is reasonable to assume that our ranking algorithm will not be perfect in the first place; it is likely to miss relevant documents. In these cases, the user should be able to circumvent errors by rephrasing the query. Rephrasing is likelier to improve the results if the user is able to **predict**, at least to some degree, how the system behaves. Traditional information retrieval systems, which were based on Boolean queries, were excellent in predictability once the user had become familiar with Boolean expressions. On the other hand, behavior of a sophisticated statistical model may seem opaque even to the system developers. Predictability is not just a usability goal – it makes also system development easier.

By **robustness** we refer to several criteria: The system should be resistant against noise, as otherwise it can not be used with arbitrary web pages. The system should not be highly sensitive to parametrization. It should perform similarly on different corpora, so that it does not require extensive domain-specific customization. Likewise, the system should perform similarly with different queries i.e. its behavior should not change drastically with varying input. In particular, the system should take all occurring words into account, regardless of their frequency. These criteria try to ensure that the system is easy to deploy in different environments, it adapts to various use cases, and its behavior does not change abruptly.

There is not a single good definition for **scalability** in information retrieval. In some sense dynamic ranking is inherently non-scalable, since in the worst case, ranking cost increases linearly with respect to number of documents. However, we can assume that the computational power of commodity hardware increases faster than the number of interesting documents to be searched, at least outside the Web. In many cases, one would like to trade CPU cycles for higher quality results. Our goal should be a method that can easily benefit from new hardware in terms of faster results and from larger corpora in terms of higher quality results.

Lacking a good formal definition, we define that a scalable ranking method should handle the largest intranet with hardware costing $40,000 at most. This definition was inspired by a keynote talk at the SIGIR 2005 conference, which was given by a Google's Distinguished Engineer Amit Singhal. As of 2007, we can assume that the largest intranets contain some 10-50 million searchable documents.

3.2 Co-occurrence Matrix

Let us begin with some definitions. In the following discussion, the basic atom is *token*. Informally, we may think that tokens are words, but this conception may be misleading. For instance, whitespace-delimited substrings that are extracted from a random Web page often do not correspond to words in any language. Consider the following "words" from Wikipedia:

`solvayprocessworks504r, pageantofsteam, addc3, lamanna`

These are valid tokens. What actually counts as a token is determined by the *tokenizer*. Tokenizer splits documents to tokens as a part of the preprocessing pipeline. Depending on its parametrization, it may allow digits, dashes, apostrophes, or other special characters to occur in tokens.

A *document* is a N_d-length finite sequence of tokens.

$$d = (t_1, ..., t_{N_d}). \tag{1}$$

A document may naturally contain multiple occurrences of the same token. If we lose the order in the document, we get a multiset

$$D' = \{t_1, ..., t_{N_d}\}. \tag{2}$$

Like with ordinary sets, order is ignored but multiplicity of tokens is explicitly significant. This gives us a so called *bag of words* representation for document, which is dominant in the information retrieval tradition [BR99].

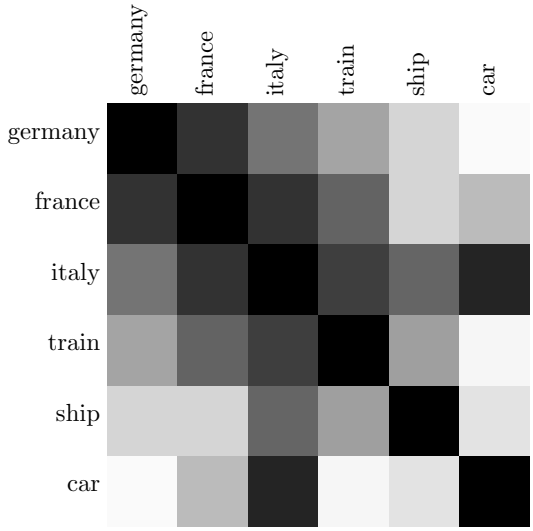

Figure 3: A co-occurrence matrix. Dark cells depict frequent co-occurrences.

If we ignore token frequencies within a document, we may cast multiset D' to an ordinary set D. An alternative document representation is given later on. The set of all D is called a corpus $\mathcal{C}$. The set of all distinct tokens that occur in $\mathcal{C}$ is denoted by $\mathcal{T}$.

When two distinct tokens belong to the same bag of words, we say that the tokens co-occur.The most fundamental concept in this thesis is the *co-occurrence matrix*. The co-occurrence matrix tells how many times any two tokens co-occur in the corpus.

For each token $t \in \mathcal{T}$ we may define an *inverted set* i.e. the set of all documents in which t occurs. Let I_t denote the inverted set for the token t, formally

$$I_t = \{D \in \mathcal{C} | t \in D\}. \tag{3}$$

By definition, I_t is non-empty for each $t \in \mathcal{T}$. Now let us pick two tokens $m, n \in \mathcal{T}$. If the intersection

$$A = I_m \cap I_n \tag{4}$$

is non-empty, words m and n co-occur i.e. they appear at least once in the same

document. We may enumerate all possible token pairs in a $\mathcal{T} \times \mathcal{T}$ matrix Λ and define

$$\Lambda_{ij} = |I_i \cap I_j|. \tag{5}$$

The co-occurrence relation is naturally symmetrical: If token m co-occurs with token n, n co-occurs with m equally often. Thus the upper and lower triangular matrices contain the same information i.e. the co-occurrence matrix is symmetric. This can be easily seen in Figure 3 that contains a co-occurrence matrix of six words. Since a token always co-occurs with itself, the diagonal values correspond to the number of documents in which the token appears in the corpus $\mathcal{C}$.

Technically, the co-occurrence matrix grows quadratically with respect to the number of tokens. Even with medium-scale corpora, its space requirements are enormous. A 100,000 document subset of Wikipedia contains approximately one million tokens[1]. If we used only one bit per value on average, the matrix would require approximately 10^{12} bits or 116 gigabytes of space – or half of this if only the lower triangular matrix is saved. Still, the space requirement is huge for any Intranet-scale corpora. Although one can get terabytes of disk space nowadays off the shelf, the average sizes of RAM and CPU caches are measured in giga- and megabytes, which makes any operations on the full matrix slow.

Fortunately, most of the tokens are infrequent. Figure 4 shows the distribution of frequencies of the aforementioned subset of Wikipedia. About 64% of tokens occur only in one document. Infrequent tokens are likely to co-occur with frequent tokens, but each of them co-occur only with a small number of other rare tokens. This can be seen in Figure 5 that shows the number of co-occurring tokens for each token in the Wikipedia subset. Infrequent tokens co-occur typically only with 10-200 other tokens. The median number of co-occurrences per token is 154 in this corpus. This

[1]A good rule of thumb for the relationship between the size of a Web corpus and the number of tokens in it is that asymptotically every new document introduces one new token.

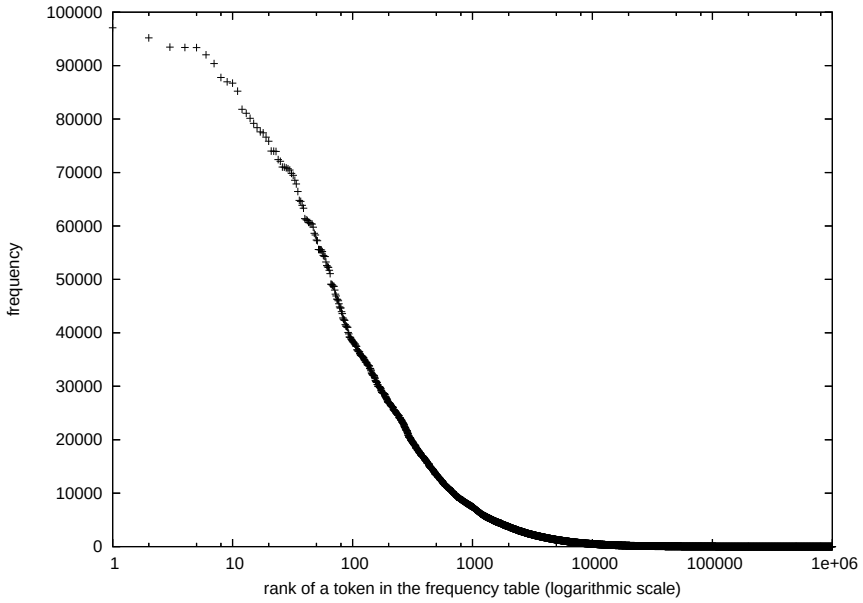

Figure 4: Zipfian distribution of frequencies of Wikipedia tokens

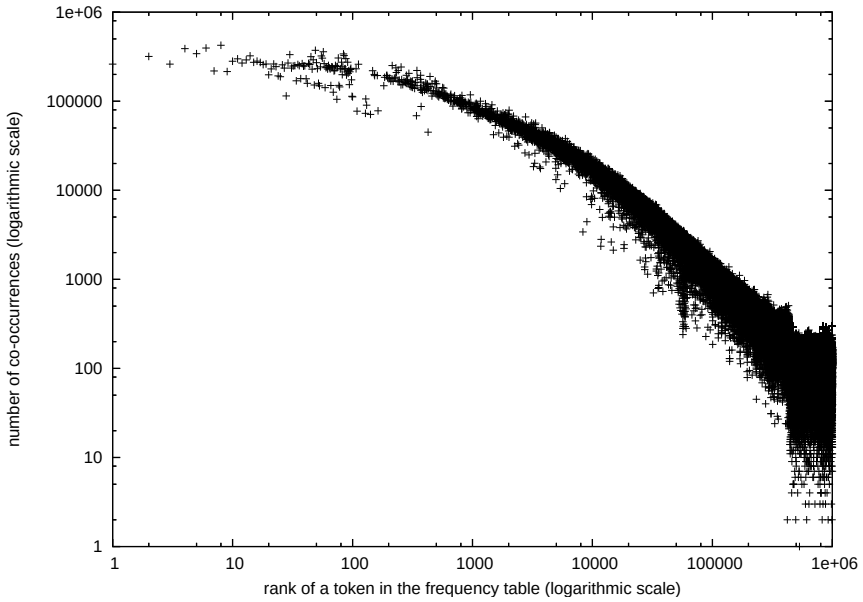

Figure 5: Number of co-occurring tokens for each Wikipedia token

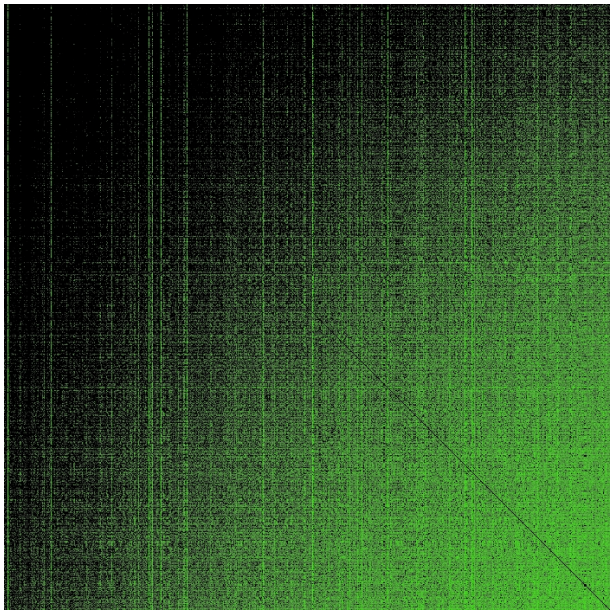

Figure 6: Co-occurrence matrix of 3000 Wikipedia tokens

implies that the co-occurrence matrix is sparse. In the Wikipedia subset, only 0.1% of the matrix is populated.

One should note that the downward curve is not very steep in Figure 5. Especially the first 10,000-50,000 most frequent tokens, which often correspond to common English words, have lots of co-occurrences with other tokens. For the purpose of content analysis, this is beneficial. A high number of co-occurrences translates to a rich characterization of the usage of a token. On the other hand, one can see that only few tokens co-occur with more than 10-20% of all tokens in the corpus, which implies that the tokens have discriminative power.

In addition to Figure 5 that shows the number of *distinct co-occurrences* for each token, it is useful to have a general idea how the absolute values of the co-occurrence matrix Λ are distributed. Figure 6 shows a co-occurrence matrix of 3000 tokens from the Wikipedia subset. The tokens are ordered according to their frequency in the full Wikipedia; the highest frequency being 52,000 and the lowest 2200 – token

frequencies decrease from left to right and from top to bottom. The dark area in the upper-left corner corresponds to high Λ values between the most frequent tokens, which are in the range from 100,000 to 130,000 co-occurrences. In the lower-right corner, the co-occurrences are in the range from 40,000 to 60,000. The visible vertical stripes correspond to tokens that are frequent in the full Wikipedia, but not frequent in the subset that was used to generate the matrix. As only few displays can show over 1000 dots per inch, the image is quantized to fit into smaller space.

We can derive some empirical probabilities from the matrix. The prior probability for seeing a token m in the corpus is

$$P(m) = \frac{\sum_{n \in \mathcal{T}} \Lambda_{mn}}{\sum_{l \in \mathcal{T}} \sum_{n \in \mathcal{T}} \Lambda_{nl}} \tag{6}$$

We can also define the conditional probability of seeing token m given that we have seen n:

$$P(m|n) = \frac{P(m,n)}{P(n)} = \frac{\Lambda_{mn}}{\sum_{l \in \mathcal{T}} \Lambda_{nl}} \tag{7}$$

This probability will have a central role in the following ranking scheme. Moreover, since items of Λ are unbounded sizes of intersections, they are not convenient for visualization as such. In the following, we will visualize co-occurrence data as $P(m|n)$ distributions instead. Note that in contrast to raw co-occurrence matrices the conditional distribution tables are not symmetrical, since in general $P(m|n) \neq P(n|m)$.

Semantics

Justifying the *semantic* validity of any formal ranking method is difficult, as long as we do not have a widely accepted formalism for natural language or human cognition. Actually, this can be seen as one motivation for using rather simple and pragmatic models: We cannot justify the use of complex models over simple ones, especially if the complex model is not as usable, predictable, robust and scalable as the simple model, even though the simple model might be clearly inadequate

semantically. In other words, since we can not justify any model in semantic terms, there is no reason to use a more complex model than the simplest one that can be justified in pragmatic terms. More discussion about this viewpoint for content-based search can be found in [TS05].

Co-occurrence data has a long history in linguistics. An early seminal work in the field, "Methods in Structural Linguistics" by Zellig Harris [Har60], states that

> The main research of descriptive linguistics, and the only relation which will be accepted as relevant in the present survey, is the distribution or arrangement within the flow of speech of some parts or features relatively to others. The present survey is thus explicitly limited to questions of distribution i.e. of the freedom of occurrence of portions of an utterance relatively to each other.

Taking "freedom of occurrence" as the basis for research is an attractive approach, as it allows objective and rigorous study of the language based on empirical data. Clearly, there are some implicit rules or conventions in the language which constrain how and which individual units may occur with each other, at least statistically.

For instance, consider two real-world examples in Figure 7. The figures contain two manually selected subsets of tokens from the Reuters corpus of news articles [LYRL04]. In the first example, two thematically different sets of tokens were chosen: The first three words are about agriculture and the next three words about metallurgy. The example shows that if one sees a word related to agriculture in a document, it is much more probable to see another word related to agriculture in the document than a word related to metallurgy. This phenomenon agrees with the common sense.

The second example in Figure 7 shows five tokens that are closely related to oil industry and five other tokens that are common words in English. If you see a common

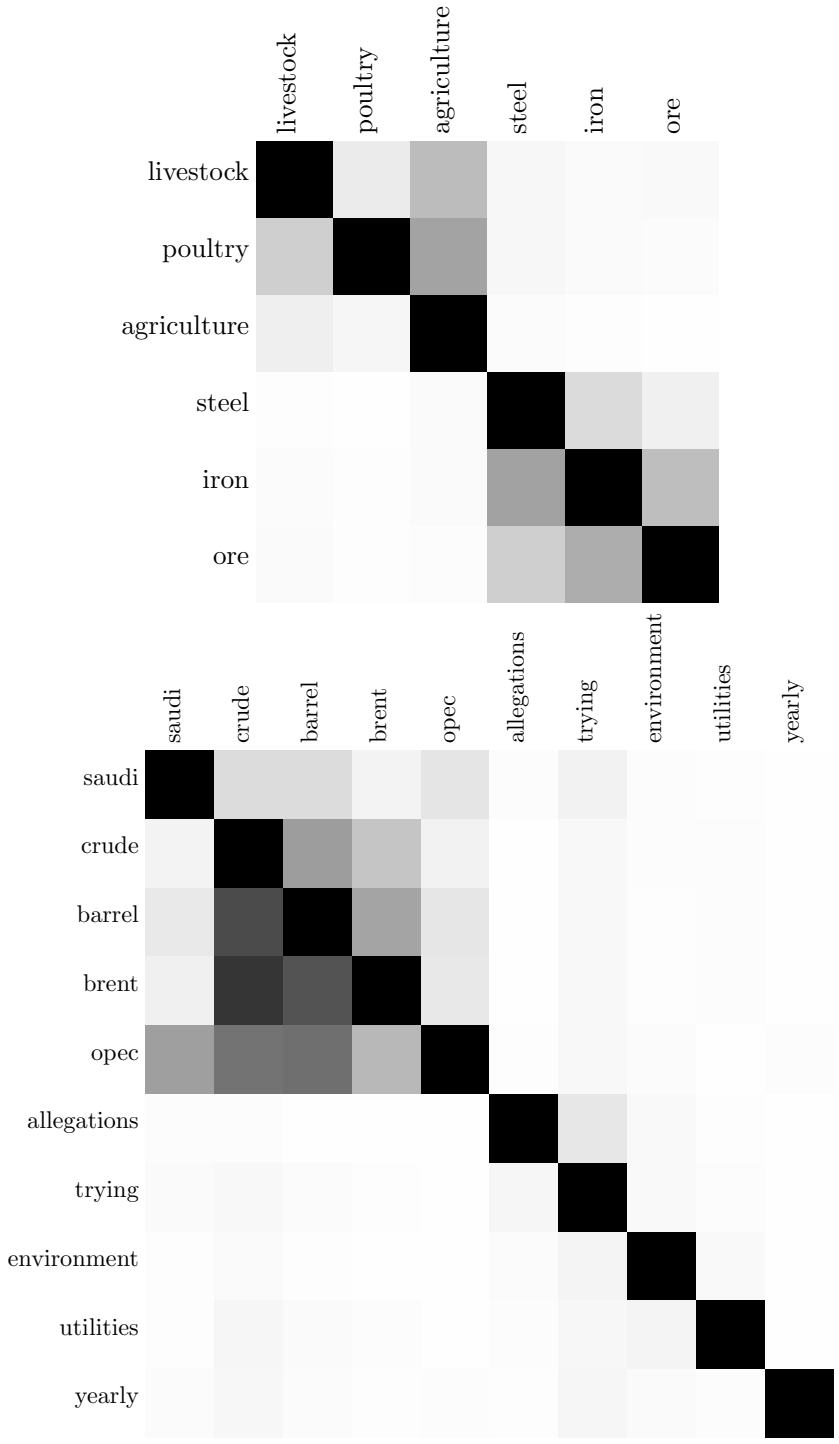

Figure 7: Conditional distributions of some tokens from the Reuters corpus

English word in a document, you can not be confident that the document is about oil, or you can not know what other common words might appear in the document. However, if you see a word related to oil, with moderately high confidence you can expect to see other oil-related words in the document as well. This exemplifies the discriminative power of tokens that are highly related to a certain topic or theme.

In the linguistic point of view, the use of co-occurrence data has some drawbacks: Firstly, it can not *explain* how language is generated in the first place – for elaboration, see for instance Noam Chomsky's influential works [Cho69, Cho72]. Secondly, the real-world co-occurrence data contains a complex mixture of several linguistic phenomena from phonetics and morphology to stylistic and pragmatic issues. In other words, the data seems extremely noisy for rigorous study.

For content-based search, the above issues are not critical. A search engine does not have to process language similarly to the human brain. In our approach, this is deliberately avoided since mimicking the brain inadequately often leads to serious problems in usability and predictability, which were our original design criteria. As long as we do not have adequate machinery for handling semantics computationally, it is better to leave semantics to the user and give the user a powerful, but non-intelligent tool[2]. In our case, the tool could be characterized as "an automated token relationship analyzer".

Regarding the second issue above, the mixture of different phenomena can be seen as an asset for content-based search. In information retrieval, the primary uses for co-occurrence analysis have traditionally been word-sense disambiguation and automatic thesaurus construction [BR99]. In addition to these, co-occurrences can be used, for instance, in automatic style analysis [UK05] and language recognition [CT94]. These applications are possible due to the richness of phenomena that are

[2]We call this tool-centric approach to language modelling *practically adequate*. For elaboration, see [TS05]

present in co-occurrence data.

The previous methods of information retrieval are not perfect. Depending on the measure, their accuracies are typically in the range from 50% to 95%. For instance, this is the case with a survey of methods for style analysis in [UK05]. False results are often attributable to co-occurrence data that contains many interleaved linguistic phenomena, as mentioned above, and the model is unable to de-noise or de-interleave the data in order to solve a particular task perfectly.

However, if we give the user tools that are powerful enough to handle the data smoothly, she may use them to solve the aforementioned tasks "manually", on case by case basis. The benefit of this approach is that the user is able to interpret the results, despite the interleaved levels of information, and act accordingly, since the human brain is perfectly tuned to the task of producing and understanding natural language. Metaphorically, one can consider machinery for efficient co-occurrence analysis as the engine of a car. Whereas the traditional information retrieval aims at building a perfect auto-pilot to drive the car, we want to give the user a perfect steering wheel.

Statistical Models and Co-Occurrence Data

Many statistical models for information retrieval are based on the co-occurrence data. Strong statistical dependencies are abundant in the data, as Figure 5 above suggests. Often dependencies can be given a natural interpretation by a human observer, as in the case of Figure 7 above. As the data is a rich and complex mixture of dependencies due to the various linguistic phenomena it captures, many different statistical model families are able to capture at least some of the dependencies. Generally speaking, any captured dependency is meaningful to a human observer, since, after all, all the dependencies were generated by the human brain in the first

place. Thus, it is not surprising that almost any statistical model, which is applied to co-occurrence data, seems to produce at least some meaningful results.

For exploratory data analysis, the above phenomenon is beneficial to some degree. For example so called Word Category Maps, which are based on the *Self-Organizing Map* model [Koh01], can be used to visualize and cluster frequently co-occurring tokens [HPK95]. The results are often intriguing and easily interpretable to a human observer. However, it is difficult to formulate a rigorous way to use the model, say, for word sense disambiguation, which would work *without* the human observer. Similarly, if the observer wanted to analyze some other dependencies in the data, in contrast to the ones that the model happens to capture, it is not clear how the model should be modified.

The results are dominated by implicit and explicit modeling assumptions and the chosen parameter estimation procedure, whose relation to semantics is not often fully understood and thus difficult to tune. For users and applications, such as content-based search, models seem rather *opaque*. This observation motivated us to take predictability as one of the design criteria – the user should be able to see through the model in order to be able to use it in an efficient manner.

Another modeling dilemma is directly related to the nature of search: What information we can afford to lose *a priori*, before seeing any query? For instance, various methods of *dimensionality reduction* are commonly used in information retrieval for de-noising and to reduce the computational burden of the models [BR99]. The aim is to reduce the number of distinct tokens in $\mathcal{T}$, either by grouping similar tokens together or by filtering out the ones that have no statistical significance.

However, as with the Wikipedia subset above, over half of the tokens may occur only once in the corpus. They have hardly any statistical significance. Yet if the user searches for a rare token, say, a model number of a microchip, the system should be

able to rank the results accordingly and maybe return other microchip data-sheets that contain some related information. This motivates the criterion of robustness. The quality of the results should not drop drastically with rare queries.

Some models are based on the working assumption that an unknown generative process has produced the tokens in documents – these models are called *latent-variable models*. The model is used to estimate these unknown, or latent, factors. In the case of co-occurrence data, the factors often correspond to semantically meaningful themes or topics that appear in the corpus. For instance, in the case of Figure 7, a latent variable model would likely recognize that tokens livestock, poultry, and agriculture are generated by a common topic. In this sense, a latent variable model can be used to group tokens together to form higher-level "concepts". Commonly used models for this task include *Multinomial Principal Component Analysis* [BP03], *Latent Dirichlet Allocation* [BNJ03], and *Probabilistic Latent Semantic Analysis* [Hof99]. Theoretically, these models are closely related to each other [BJ06].

When a probabilistic model is used for content-based ranking, a so called *Language Model* formalism is dominant nowadays. In this formalism, document ranking is based on *query likelihood* [PC98]:

$$\log P(Q|D) = \sum_{q \in Q} \log P(q|D), \tag{8}$$

where Q is the query, possibly consisting of several tokens q. In this approach, the model is used to estimate probabilities of form $P(t|D)$ i.e. the probability of a token t to appear in document D. Again, infrequent tokens pose a difficult challenge to the model: How to estimate $P(t'|D)$ for all $D \in \mathcal{C}$, if t' occurs only in one document, and thus $P(t'|D) = 0$ for most of the documents? This problem is known as the *Zero Probability Problem* in the literature [PC98]. The problem is often solved by various *smoothing* methods that try to spread the probability mass more uniformly over the tokens. The chosen smoothing method is a major factor that distinguishes

different language models.

One can use a latent variable model as a language model. In this case, the query becomes conditional on the latent variables, $\mathcal{Z}$:

$$\log P(Q|D) = \sum_{q \in Q} \sum_{z \in \mathcal{Z}} \log P(q|z) P(z|D). \tag{9}$$

Some examples of this approach applied to content-based ranking can be found in [BPT04, BLPV05]. Here the latent variables $\mathcal{Z}$ can be seen as a bottleneck, which compresses global co-occurrence statistics to a smaller, fixed number of dimensions, $|\mathcal{Z}|$. The largest latent-variable models that have been built for search have included at most hundreds of topics [BLPV05]. Due to relative sophistication of the models, scaling them up is a major challenge.

Based on the above, a latent-variable model can be seen as a dimensionality reduction method. However, in this case tokens are not filtered out, but instead they are grouped together by $\mathcal{Z}$. For instance, consider the following observation from [BP03]:

> For instance, a "Mother Teresa" component was discovered whose extreme documents were all about Mother Teresa but whose general role in other documents was to isolate facets of motherhood and elderly women.

This result is semantically understandable. However, what if the user was specifically interested in geriatrics and elderly women? In this case, the model might estimate that the query is probably about Mother Teresa and rank the results accordingly. Similarly, a latent-variable model could easily find the two topics in Figure 7, namely farming and metallurgy. Yet if the user was *specifically* interested in poultry, which is a subset of livestock, grouping it under the topic of farming would be an over-generalization.

These issues of over-generalization and the lack of semantic resolution are inherent to latent-variable models, at least in the context of search. The problems are greatly amplified when a latent-variable model of, say, 500 topics, is applied to a Web corpus of millions of documents. Generally speaking, we can not judge beforehand which features or tokens can be grouped together or filtered out until we see the query.

The intention of the previous discussion is not to undervalue the statistical methods and models for information retrieval. The discussed methods are valuable for various purposes, such as clustering, summarization, data mining, and even domain-specific search where lots of *a priori* information on relevance is available. However, the points made are applicable to general, Web-like, search. To a large extent, the design criteria of *predictability*, *robustness*, and *scalability* were motivated by the need to address the issues discussed above. The following section will describe our solution.

3.3 Method

This section presents *AinoRank*, a content-based ranking method. The method in itself is simple, which is deliberate. The central questions are whether the method can be seen to fulfil the original design criteria and whether it is possible to implement the method in a scalable manner. We start with a simple special case and gradually extend the description to cover the whole method.

Consider that we have a corpus of documents, $\mathcal{C}$. The user submits a query, $q \in \mathcal{T}$ to the system. We require that the query token occurs in the corpus at least once. We start with a simplified assumption that the query consists of only one token, q. The ranking method assigns a score, $\mathcal{S}'_q(D)$ for each document $D \in \mathcal{C}$. The system returns a ranked sequence of documents $(D_1, D_2, ..., D_N)$ so that $\mathcal{S}'_q(D_i) > \mathcal{S}'_q(D_{i+1})$.

In this case, AinoRank assigns score to a document as follows

$$\mathcal{S}'_q(D) = \lambda_D \sum_{t \in D} P(q|t), \tag{10}$$

where λ_D is a document-specific normalization term and $P(q|t)$ is estimated using the full co-occurrence matrix as specified in Equation 7.

We can formulate scoring of all $D \in \mathcal{C}$ in matrix form as follows. Let $\mathbf{D}$ be a $\mathcal{C} \times \mathcal{T}$ matrix containing the documents. A row from the $\mathcal{T} \times \mathcal{T}$ conditional distribution table corresponding to the query q is denoted by $\mathbf{q}$. The document normalization factors $[\lambda_D], D \in \mathcal{C}$ are entries of a diagonal matrix $\mathbf{N}$. Now scoring can be formulated as follows

$$\mathbf{s} = \mathbf{NDq}, \tag{11}$$

where $\mathbf{s}$ is a $|\mathcal{C}|$-dimensional vector containing the document scores.

This formulation for $\mathcal{S}'_q(D)$ is similar to so called *Translation Model* for information retrieval [BL99]. Translation Model is defined as follows:

$$T(q|D) = \sum_{t \in D} l(t|D) r(q|t), \tag{12}$$

where $l(t|D)$ is *document language model* and $r(q|t)$ *translation model*, following the terminology of [BL99]. Now if we assume that the language model is uniform for all $t \in D$, we can let $\lambda_D = l(t|D)$. Furthermore, if we define that the translation model is based on the empirical probability $P(q|t)$ derived from the co-occurrence matrix, we can equal $T(q|D) = \mathcal{S}'_q(D)$.

Intuitively, the idea behind the above formulation is the following. We want to give a high score to documents that contain *either* the query token q *or* many co-occurring tokens $\{t|\Lambda_{tq} \neq \emptyset, t \in \mathcal{T}\}$. Since co-occurring tokens contain many synonyms,

hypernyms, and hyponyms of q, this scheme should reward relevant documents even though they would not contain any occurrences of the query token q.

Naturally the co-occurrences include non-relevant tokens as well. We try to weight relevant tokens against the non-relevant ones with $P(q|t)$. The idea is that for relevant tokens, Λ_{tq} should be large and Λ_{tt} (the frequency of token t) not be much larger than that. In other words, to gain a high weight, token t should appear *only* in the same documents as q, i.e. preferably $I_t \subseteq I_q$. Very frequent words have large Λ_{tq} but also their Λ_{tt} is large and thus $P(q|t)$ becomes small. In probabilistic terms, we give high weight to term t if seeing it makes seeing q in the same document probable. Correspondingly, we give a high score to document D if it contains many tokens related to q.

The above scheme favors long documents. As each token in a document can only increase the score, the more tokens the document has, the higher score it will get. This is not beneficial. Here the normalization term λ_D comes into play. A straightforward fix is to take the average of the weights, or let

$$\lambda_D = \frac{1}{|D|}. \tag{13}$$

This definition coincides with the uniform document language model, $l(t|D)$ above which assumes that seeing each token $t \in D$ is equally probable. Semantically this means that we favor documents that contain the maximum number of informative i.e. rare words that co-occur with the query. This does not always produce desirable effects, as we will see later on.

We employ also another approach to handle documents of varying lengths. The lengths of web pages vary from one token (e.g. `error`) to full dissertations and books of hundreds of pages. Lengthy documents are problematic not only for ranking, but also for co-occurrence statistics. We defined the number of co-occurrences between two tokens as $\Lambda_{ij} = |I_i \cap I_j|$ i.e. the number of common documents in which they

appear. A long document, say a dissertation about bioinformatics, may speak about genetics in the first part and about machine learning in the latter part. Following the above definition, the document would increase co-occurrence counts between tokens related to genetics and machine learning, which lessens topicality of the co-occurrences.

Even if a long document is only about one topic, the current formulation is sub-optimal. For instance, a two hundred page book about global warming is a rich source of topical co-occurrences. The document might mention token `ozone` a hundred times, but each co-occurrence count between `ozone` and all the other tokens in the book would increase only by one.

To overcome these problems, we *segment* the documents. Each document is split into fixed-size segments during tokenization. In our experiments, a typical segment size has been 300 tokens. The choice of the segment size is based on semantical and technical considerations. In the semantic sense, the segment size defines the *context* of a token. The chosen size should reflect our belief on the extent of a token's influence over the document. The segment size of 300 is based on the assumption that a token's semantic context covers some 2-3 paragraphs.

On the other hand, the smaller the segment size, the fewer distinct co-occurrences a document will produce. In an extreme case, the segment size of two reduces the co-occurrence matrix to a bigram-like model. Each distinct co-occurrence increases the space requirement of the index, as seen in Chapter 4.

Formally, we may extend the original document definition in Equation 1 to include the segments. Document d is split to $\lceil \frac{N_d}{K} \rceil$ segments as follows

$$s_d^i = (t_{Ki+1}, ..., t_{\min(N_d, K(i+1))}), i \in \left[0, \left\lfloor \frac{N_d}{K} \right\rfloor\right], \tag{14}$$

where K is the segment size and i gives the segment's position in the document.

Correspondingly, we may define a segment set, $S_D^i = \{t|t \in s_d^i\}$. Usually we are not interested in the actual position of a segment in the document and S_D^i becomes S_D that may refer to any segment in document D. The set of all segments in document D in denoted by $\mathbf{S}_D$. Inverted sets take each segment into account separately

$$\overline{I}_t = \{S_D|t \in S_D, D \in \mathcal{C}\}. \tag{15}$$

Other terms, such as Λ and $P(q|t)$, which are based on inverted sets, are derived equally for I_t and $\overline{I}_t$. We use an overline to denote the segment-based version of a term. Since conceptually the segment-based approach is not much different from the original formulation, differences being mainly in the co-occurrence matrix and in the document score, we may use the original formulation for clarity in the following sections. All the results are applicable to the segment-based approach as well.

We can now give a new definition of the document score,

$$\overline{\mathcal{S}}(D) = \frac{1}{|\mathbf{S}_D|} \sum_{S \in \mathbf{S}_D} \sum_{t \in S} \overline{P}(q|t). \tag{16}$$

Since segments are of equal length K (although the last segments in documents may be shorter), there is no need for a segment-specific normalization term, like λ_D above. However, documents may contain a varying number of segments, so for the document score we take the average of the segment scores.

One could consider also a more sophisticated way to segment documents. The segments might be overlapping or they could be of varying size depending on their contents. Overlapping segments would be expensive to implement with the current approach, as shown in Chapter 4, and their benefits are not clear.

Advanced segmentation schemes for content-based search have been considered in the literature. For instance, [CYWM04] compares ranking quality between non-segmented documents, fixed-size segments, a segmentation scheme based on HTML tags, and a scheme based on the visual layout of a web page. In the study, fixed-size

segments return consistently the best, or the second best results, right behind an expensive layout-based approach that was introduced in the paper. Especially, the fixed-size segments improved results compared to full documents.

3.4 Query Interface

Consider that instead of a singular q, we have a set of query words, $q \in Q$. The original scoring function in Equation 10 could be easily extended as follows,

$$\mathcal{S}'_Q(D) = \lambda_D \sum_{t \in D} \prod_{q \in Q} P(q|t). \tag{17}$$

However, this is not the only option. Think of the semantic viewpoint: What is the user's intention when she adds more tokens to the query? We have two alternatives: Either she wants to *extend* the query to cover more documents, or she wants to *constrain* the query to a more specific subset of documents. The former is called disjunctive and the latter conjunctive query.

Traditional keyword-based search engines, such as Google, are usually conjunctive, since in the disjunctive scheme a long query can easily produce impractically many results. Moreover, the higher number of documents match the query, the tougher it is for the ranking method to ensure that the most relevant ones are ranked on top – especially if the query is ambiguous. However, this is not an issue with AinoRank, as we can affect the ranking behavior and the result set size independently from each other, as we will describe in the following.

Keys & Cues

The first design criterion for AinoRank was usability. Even though our ranking method might be highly predictable, robust, and scalable, ultimately input from

the user defines whether the results are of any use – the right answer to a wrong question does not help the user much.

As stated in the design criterion, the query interface should take into account the discrepancy between the user's true information need and the actual query. Moreover, since another design criterion of ours, predictability, required that the user must be able to re-phrase the query easily if the original query failed, we must not resort to guess-work with respect to the user's intentions. If the system was eager to guess, and it guesses wrong, and the user tries to re-phrase the query only to see another bad guess, the query interface becomes a frustrating cat-and-mouse play for the user.

In order to better understand the problem, let us consider what the users typically search for. Based on a sample of 100-200 queries from a query log of the AltaVista Web search engine, Rose and Levinson analyzed and interpreted typical search goals of the Web users [RL04]. The results, as presented in their paper, are summarized in Table 2.

First, consider that no ranking method was available: Search engine would return the documents that match the exact query tokens in some random order. Which of the query types in Table 2 would likely return a relevant document among the top ten results in this case? Navigational (1), locate (2.4), and download (3.1) queries might work due to proper nouns in the query. Some query types, such as undirected (2.2) and obtain (3.4), might sometimes work if the query contained a rare, specific word. If the goal was really broad or vague, such in undirected (2.2) and entertainment (3.2) queries, the results might be satisfactory because of the "anything goes" attitude.

This is the baseline. Under no circumstances should a sophisticated ranking method produce *worse* results than no ranking at all. This requirement might feel trivial,

Search Goal	Description	Example
1. **Navigational**	My goal is to go to specific known website that I already have in mind. The only reason I'm searching is that it's more convenient than typing the URL, or perhaps I don't know the URL.	`"aloha airlines", "duke university hospital", "kelly blue book"`
2. **Informational**	My goal is to learn something by reading or viewing web pages.	
2.1 Directed	I want to learn something in particular about my topic.	
2.1.1 Closed	I want to get an answer to a question that has a single, unambiguous answer.	`"what is a supercharger", "2004 election dates"`
2.1.2 Open	I want to get an answer to an open-ended question, or one with unconstrained depth.	`"baseball death and injury", "why are metals shiny"`
2.2 Undirected	I want to learn anything/everything about my topic. A query for topic X might be interpreted as "tell me about X".	`"color blindness", "jfk jr"`
2.3 Advice	I want to get advice, ideas, suggestions, or instructions.	`"help quitting smoking", "walking with weights"`
2.4 Locate	My goal is to find out whether/where some real world service or product can be obtained.	`"pella windows", "phone card"`
2.5 List	My goal is to get a list of plausible suggested web sites (i.e. the search result list itself), each of which might be candidates for helping me achieve some underlying, unspecified goal.	`"travel", "amsterdam universities", "florida newspapers"`
3. **Resource**	My goal is to obtain a resource (not information) available on web pages.	
3.1 Download	My goal is to download a resource that must be on my computer or other device to be useful.	`"kazaa lite", "mame roms"`
3.2 Entertainment	My goal is to be entertained simply by viewing items available on the result page.	`"xxx porno movie free", "live camera in l.a."`
3.3 Interact	My goal is to interact with a resource using another program/service available on the web site I find.	`"weather", "measure converter"`
3.4 Obtain	My goal is to obtain a resource that does not require a computer to use. I may print it out, but I can also just look at it on the screen. I'm not obtaining it to learn some information, but because I want to use the resource itself.	`"free jack o lantern patterns", "ellis island", "lesson plans", "house document no. 587"`

Table 2: Search goals by Rose & Levinson [RL04] based on AltaVista queries

but considering the heterogeneity of the goals, it is easy to focus on optimizing one of them and forget the others. For example, one might build a sophisticated question-answering system to serve open informational needs (2.1.2) and the resulting system might be unable to serve simple navigational or download queries that contain a single product number. Naturally, if the system does not aim at being a general-purpose search engine, specificity is justified.

Let us make another thought experiment: Take a sheet of paper and cover the first two columns in Table 2. Now, assume that the example queries were shown to you in some random order. Could you assign each query to the corresponding search goal just by looking at the queries? If you had no previous knowledge, how would you know whether `pella windows` refers to a Web site (1), to a location (2.4), to a computer program (3.1), or to an off-line document (3.4)? Similarly, given the query `travel`, would you return the Web site `www.travel.com` (1), an encyclopedia page about traveling in general (2.1.2), or blog articles about traveling experiences of an individual (2.3)?

As a human being can not guess intentions of another reliably, it feels unreasonable to try to solve the question computationally. One can not expect that the user provides enough information to make the queries unambiguous and her intentions clear in the first place. The only reliable source that can explicate the intentions is the user herself. Our approach is to let the user to use the system as a tool that does not make any guesses but it does what it is told to do in a predictable way. It is left on the user's explicit responsibility to command the tool to return desired results.

We call this approach *Keys & Cues*. The core concept is that the user may control the set of matching documents and their ranking independently from each other. In this sense, the query interface is structured or multi-faceted: Query tokens may have different roles in the query string, namely, each token is either a key or a cue.

Similar ideas and rationales for this *structured query* approach can be found e.g. in [KJ98, Bro95].

Keys correspond to the standard keyword interface, like, say, in Google. *Cues* are used to rank the documents that match to the keys, using for instance Equation 16 above. Note that due to the nature of the ranking scheme, cues do not have to occur as such in the matched documents. Instead, all tokens that co-occur with the cues in the corpus are taken into account in ranking. One can think that keys are used to *filter* the desired content from the corpus and cues are used to *sort* the results.

It would be infeasible and nonsensical to try to distinguish keys and cues automatically in the query. The roles of the tokens depend fully on the user's implicit intention. All tokens $t \in \mathcal{T}$ are equally valid as keys and cues. The distinction depends on the user, on query by query basis. Since the distinction between the roles must be explicit, we reserve a special character, usually a slash '/', to denote the cues. Another approach would be to let the user type keys and cues in two separate input boxes.

Consider the following queries that exemplify various use cases of the query interface:

apple /computer Key `apple` is ambiguous. The user may explicate her intention and *disambiguate* the query with a single cue. In this case, all returned documents contain token `apple` and the top-ranking results are about OS X, iPods, and MacBooks etc.

computer /apple In this case, all documents containing token `computer` are returned. The top-ranking results talk probably about the Apple products as in the previous case, but they could be also about some similar products, such as BeBox or Windows Vista, which are often characterized in the apple-like terminology.

cluster /k-means The disambiguating cue can be also a specific term from the

desired context. In this case, documents related to various clustering methods are ranked on top – not only the ones that discuss about the k-means method.

book /buddha Cues can be used to specify the desired *theme*. In this case, the user is interested in religious books. The opposite query, `buddha /book` would return information specifically about Buddhism.

site:en.wikipedia.org /biology On the Web, a key can be used to specify a site of interest. A cue can be used to sort contents of the site in a desired way.

george bush /foreign /politics Cues can specify an *abstract concept* that would be difficult to express in keywords.

nokia /suxors /rulez Since any tokens, also rare ones, can be used as cues, one can use untypical forms of words or even misspellings to specify a certain *style* for documents. In this case, the top-ranking results are probably opinionated forum comments about mobile phones.

airplane /b52 A rare token as cue is often a good way to specify a *detailed topic* of interest – in this case, large bomber planes.

test /19 /23 /29 /31 Since the ranking scheme is rather straighforward and mechanical, it is also very *flexible*. In this case, the top results are probably about number theory and primality tests.

saddam hussein /uno /dos /tres Since tokens of one language are highly co-occurring, one can use any foreign words as cues to specify a desired *language* for the top results.

/how /tall /is /the /eiffel /tower Unintentionally, one can pose *questions* in plain English and surprisingly often get an answer to the question within the top-ranking results. The reason for this is that common interrogative words,

such as `how` above, get only a small weight due to their high frequency – similarly to other common words as `is` and `the` above. Thus, only topical words, `tall`, `eiffel`, and `tower` in this case, get any remarkable weight and they bring the documents that are related to the question on top. The small effect caused by the interrogative tokens is often enough to raise the explicit answers to the question amongst the very first results.

If the user does not specify any cues, we use keys as cues. The rationale is that a query like `computer /computer` or `george bush /george /bush` should return in some sense most prototypical examples of documents that match to the keys. Moreover, this way the user may start to use the system similarly to other well-known search engines that do not let the user to affect the ranking explicitly.

Table 3 shows some strategies how *Keys & Cues* may be used to achieve the search goals of Table 2. Three strategies are used: First, if the query contains a proper noun or other specific token, typically cues are not needed (1). Secondly, if the user is interested in a particular topic, the most specific, and the most infrequent, token is used as the key and the broader specifiers as cues (2.1.2, 2.3). Thirdly, if the desired information should appear in a specific context or it should be presented in some particular style, indirect cues may be used. For example, in case of general interest, one may use some general, non-specific words as cues (2.1.1, 2.2). If a commercial context is desired, words related to selling and buying often produce the wanted result (2.4). Similarly, any word that refers to the desired use or origin of the information, may be a good cue (3.1, 3.4). If the results should appear in a web page of particular style, one may utilize previous knowledge of the structure of such pages to formulate matching cues (2.5, 3.2, 3.3). Note that these strategies are given only as illustrative examples of use. In practice, each user finds best strategies of her own through trial and error.

Search Goal	Keys & Cues example
1. **Navigational**	`"aloha airlines", "duke university hospital", "kelly blue book"`
2. **Informational**	
2.1 Directed	
2.1.1 Closed	`"supercharger /what", "2004 election dates /list"`
2.1.2 Open	`"baseball /death /injury", "metals /why /shiny"`
2.2 Undirected	`"color blindness /cause /effect", "jfk jr /life /work"`
2.3 Advice	`"smoking /help /quit", "weights /walking"`
2.4 Locate	`"pella windows /shop /sell", "phone card /street /address"`
2.5 List	`"travel /list /http /com", "amsterdam universities /links"`
3. **Resource**	
3.1 Download	`"kazaa lite /download", "mame roms /zip /gz"`
3.2 Entertainment	`"free xxx /porno /movie", "l.a.  live /camera"`
3.3 Interact	`"weather /cookies /javascript", "measure converter /login /register"`
3.4 Obtain	`"free jack o lantern patterns /images", "lesson plans /handout", "house document no.  587 /scanned"`

Table 3: Some strategies to achieve search goals with Keys & Cues

The *Keys & Cues* query interface corresponds to the original design criteria as follows. It takes into account the user's implicit information need by letting the user control both the set of returned documents (*keys*) and their ranking (*cues*) separately. The user may steer the results freely towards the desired search goal by using the two facets. In this sense, the system should be *usable* in many situations.

Considering *robustness*, it is important that also rare tokens are allowed as cues, as this allows the user to experiment with the system using her own name or with her favorite dog breed etc. When the user sees how the system performs in a field that is already familiar to her, she may transfer this understanding to other, more general cases.

The user should be able to form a mental model of the system, either consciously or unconsciously. The ranking scheme is easy to explain in layman terms: *"You can affect how the results are sorted by giving the system some cues about your interests. The cues may be any words. The more words a document contains that are related to your cues, the higher rank it will get."* Now, if the system returns undesired results, the user may re-phrase the query with other cues. Since the user has some idea how the system works and she may trust that the results are robust regardless of the query, she can make a sophisticated guess or *predict* which cues would produce best results in her particular situation.

The next section shows how the system tries to ensure predictability and robustness in case of multiple cues.

Multi-Token Cues

Traditionally, the quality of a ranking scheme has been measured by two metrics: *Precision*, or how many of the results are relevant, and *recall*, or how many of all relevant results were returned [BR99]. However, the user does not judge results in

	Precision	**Recall**
Add / Remove	kw conjunctive	kw disjunctive, AinoRank
Modify	AinoRank	

Table 4: Mapping from the query interface to the ranking behavior

these terms. The user can either feel that the results are bad (low precision) or she may want to see more good results (low recall).

A textual query interface gives the user two ways to affect the results: She may either add and remove query tokens or she may modify any existing tokens. Table 4 summarizes the user interface options (add / remove, modify) and the ranking behavior (precision, recall), which the user may want to tune via the query interface. Entries "kw conjunctive" and "kw disjunctive" show how a keyword-based search engine, that utilizes either a conjunctive or disjunctive query interface, works. Note that in a "kw conjunctive" system, like in Google, there is no easy way to increase recall. Similarly increasing precision is difficult in a "kw disjunctive" system.

If only ranking is considered, there is one way to map controls to behaviors, which is clearly more usable than the others: If the user wants more results, she adds *new tokens* that describe what kind of *new results* should be returned. If the user wants to *change the results*, she may *change the query tokens*. The term AinoRank represents this mapping in Table 4. Consider the other alternatives: If recall was to be increased by modifying tokens, the user is challenged with a difficult semantic question, namely which token could possibly produce more results. Correspondingly, it would feel strange to increase precision by adding more words, since usually one wants to increase precision if the results are bad, i.e. the current query did not produce good results. In this case, it feels most natural to change the query instead.

Based on the discussion above, we can now re-formulate Equation 17, which was our

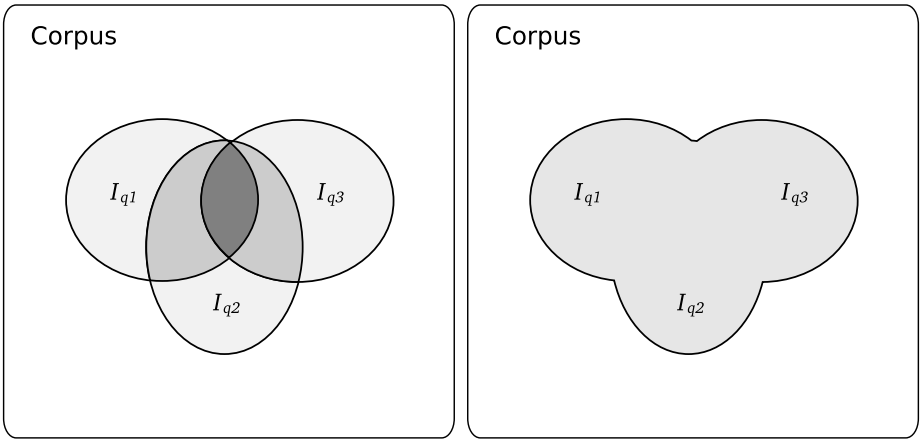

Figure 8: Alternative Venn-diagrams for multi-token cues

first attempt to handle multi-token queries. Now we know that each token $q \in Q$ should *extend* the coverage of the ranking. The ranking should take into account *more* "semantic categories", as defined by the new tokens. For instance, consider that the user has ranked the results first with token `Chevrolet` and she notices that the results are too specific, as she is interested in American cars in general. She adds tokens `Hummer` and `Lincoln`. Now she would vaguely expect that the results are ranked according to what is common in Chevrolets, Hummers, and Lincolns – namely that they are all American cars. Our ranking scheme should aim at this result. Furthermore, the scheme should ensure that individual tokens in the query do not interfere with each other, so that the user may add any new tokens to the query independently without unexpected side-effects.

Let us remind how $P(q|t)$ definition reduces to inverted sets

$$P(q|t) = \frac{\Lambda_{qt}}{\Lambda_{tt}} = \frac{|I_q \cap I_t|}{|I_t|}. \tag{18}$$

Consider that we have multiple cue tokens, $W = \{q_1, ..., q_n\}$. Correspondingly, we have inverted sets, $W_I = \{I_{q_1}, ..., I_{q_n}\}$. The Venn diagram in Figure 8 illustrates the case for the three first tokens. If cue tokens are semantically related, it is likely that there exists a set of documents that contains occurrences of all cues. This set

is the intersection $\widehat{W_I} = \bigcap_{I \in W_I} I$, depicted by the darkest area on the left Venn diagram. In the case of Equation 17, the weight is multiplied for the tokens that occur in the intersection, $\{t \in D | D \in \widehat{W_I}\}$, since each cue token $q \in W$ amplifies the effect of these tokens in the summation. Correspondingly the relative weight of the other tokens is decreased. Thus, if the user adds new cue tokens that are highly overlapping with the previous cue tokens, she may inadvertently amplify the weight of the old cue and the cue coverage does not extend as the user intended. Since the user can not know which tokens are overlapping, the ranking mechanism must include measures against this phenomenon.

The current solution in AinoRank is as follows. Given cue tokens Q, we define the corresponding *cue set*, $\mathbb{Q}$, as follows

$$\mathbb{Q} = \bigcup_{q \in Q} I_q. \tag{19}$$

A cue set corresponding to W above is depicted on the right in Figure 8. In this scheme, a new cue token q_{n+1} changes the cue coverage only to the extent that the token q_{n+1} provides new information. If it happens that a new token is highly related to the previous ones, say $I_{q_2} \subset I_{q_1}$, it does not add any new information and the cues are unaffected, which the user can see easily.

To handle multi-token cues in the desired manner, we do not estimate individual cues $q \in Q$ independently. Instead, we estimate the probability of the cue set as a whole:

$$P(\mathbb{Q}|t) = \frac{|\mathbb{Q} \cap I_t|}{|I_t|}. \tag{20}$$

The document score is estimated accordingly

$$\mathcal{S}'_{\mathbb{Q}}(D) = \lambda_D \sum_{t \in D} P(\mathbb{Q}|t). \tag{21}$$

A philosophically inclined reader may contrast the concept of cue set to that of *extension* in semantics and semiotics [Lac96]. In the literature, the idea of representing

concepts as sets [Kor85] or areas in space [Gar00] is not new. Even though the above formulation of cue sets is justified only in pragmatic terms, one may consider it as a reminiscence of many previous, semantically more justified, approaches.

3.5 Discussion

We are now ready to formulate the final document score for AinoRank,

$$\begin{aligned} \mathcal{S}_D^{\mathbb{Q}} &= \frac{1}{|\mathbf{S}_D|} \sum_{S \in \mathbf{S}_D} \sum_{t \in S} \overline{P}(\overline{\mathbb{Q}}|t) \\ &= \frac{1}{|\mathbf{S}_D|} \sum_{S \in \mathbf{S}_D} \sum_{t \in S} \frac{|\overline{\mathbb{Q}} \cap \overline{I}_t|}{|\overline{I}_t|} \\ &= \frac{1}{|\mathbf{S}_D|} \sum_{S \in \mathbf{S}_D} \sum_{t \in S} \frac{|(\bigcup_{q \in Q} \overline{I}_q) \cap \overline{I}_t|}{|\overline{I}_t|}. \end{aligned} \tag{22}$$

This final score is based on segmented documents and cue sets. Formally, the proposed search engine works as follows. The user submits a query via the query interface. The query is a tuple $(\mathcal{K}, Q)$ where $\mathcal{K}$ is a sequence of key tokens that are used to filter a matching set of documents from the corpus. We may define this *result set* as follows

$$\mathcal{R} = \{D \in \mathcal{C} | match(D, \mathcal{K}) = 1\}, \tag{23}$$

where the function *match* returns 1 if the given document D matches with the given keys $\mathcal{K}$. The actual behavior of *match* is explained in section 4.1. Documents in the result set are then scored using the second item of the query tuple, the cues Q. Based on Equation 22 above, we get a scored result set in which each document is accompanied by its score

$$\mathcal{S}_{\mathcal{R}} = \{(D, \mathcal{S}_D^{\mathbb{Q}}) | D \in \mathcal{R}\}. \tag{24}$$

The results are then ranked according to descending score

$$\tau(\mathcal{S}_{\mathcal{R}}) = (D_1, ..., D_{|\mathcal{R}|}),\ \mathcal{S}_{D_i}^{\mathbb{Q}} > \mathcal{S}_{D_{i+1}}^{\mathbb{Q}}. \tag{25}$$

The ranked results τ are finally shown to the user.

There are well-known algorithms to implement operations in Equations 23 and 25, namely keyword matching and sorting, efficiently. In the following chapter we will show that it is possible to implement operations required by Equations 22 and 24 in a moderately scalable manner as well.

4 Implementation

On the surface level, a Web search engine is a simple device, at least when compared to a full-fledged relational database system. Its core data structures may be immutable as all queries are read-only. Data and queries are mostly unstructured and the supported query language is minimal compared to, say, Simple Query Language for relational databases [RG02]. Few precautions are needed against data loss, since the data is backed up in the Web anyway.

Scale is the challenge. Even medium-sized intranets, or subsets of the Internet, are measured in tens of millions of documents nowadays. The system may have to serve tens of millions of queries per day. Another great challenge for Web search is crawling, or how to collect the Web pages in the first place. Despite these challenges, current commercial web search engines manage to provide a remarkably smooth user experience.

Given that the core challenges are "solved" by commercial offerings in the case of static ranking, this thesis focuses on a further challenge: We investigate whether dynamic ranking, utilizing co-occurrences in the corpus, could be a feasible option for an intranet or for small-scale Web search in the future.

To study the question, we have implemented an efficient search engine, Aino. Aino includes a full, distributable preprocessing pipeline from character set normalization and language recognition to HTML tag removal and tokenization. Based on the output of the preprocessing pipeline, efficient indices are built that are optimized for content-based ranking. The indices are explained in detail in Section 4.1 below.

Query processing, which includes keyword matching and ranking, is described in the following sections 4.2 and 4.3 – the former presents a straightforward brute-force algorithm for the ranking scheme that was presented in Chapter 3, and the latter an optimized version of it. Finally, we show briefly in section 4.4 how the

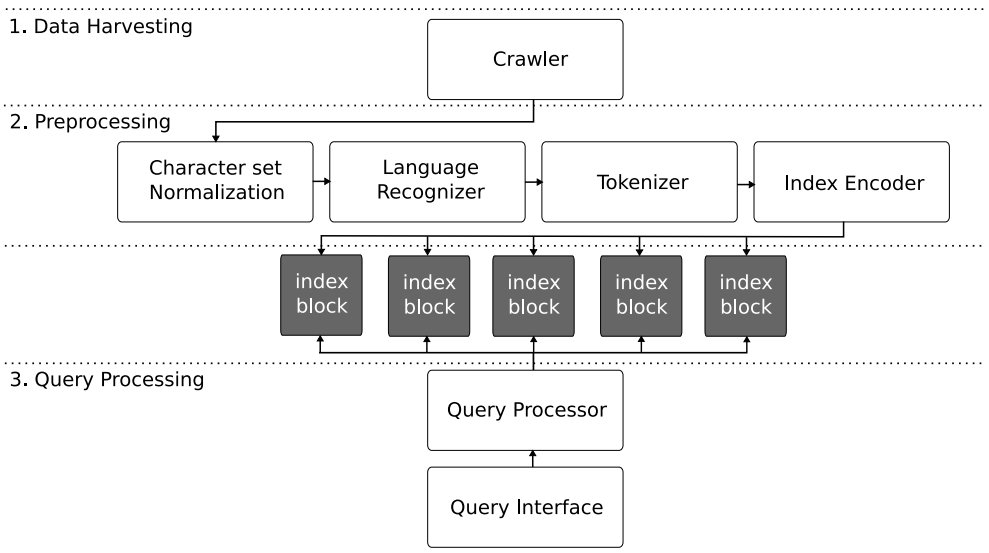

Figure 9: Architecture of Aino

system is distributable to a cluster of servers.

Architecture

A short introduction to the anatomy of a search engine is needed in order to follow the next sections. The subject is well covered in several books, e.g. [WMB99, BR99]. The description of the early Google architecture [PB98] gives a concise technical overview of some implementation issues as well.

Figure 9 illustrates the essential parts. We divide the process in three distinct phases:

1. Data harvesting or crawling
2. Preprocessing and indexing
3. Query processing

During the first phase, data is collected for indexing. After all, a search engine can not search for anything it has not seen before. Depending on the application

domain, data may originate from the Web, an email archive, or from any other source. Collecting or crawling Web pages is a complex issue in itself and it is not covered in this thesis. However, Aino is accompanied by an advanced Web crawler called HooWWWer [Tuo05], which was developed in a sister project to Aino.

The purpose of the next phase, indexing, is to extract all necessary information from the raw document stream produced in the first phase to efficient data structures, which are then utilized by the query processing in the third phase. Documents may be in various formats and they may have been written in various languages using exotic character sets. The first step is to normalize all documents into a common format, typically to raw text encoded in Unicode or ASCII. Raw text is then fed to a language recognizer, which uses 4-grams to detect language with high accuracy. The basic method is explained in [CT94]. Aino uses a highly optimized version of the method.

The next step is tokenization, which was briefly introduced in section 3.2. During tokenization raw text is split into substrings or tokens, which become atomic elements of documents. Depending on parametrization, tokenizer may take into account compound words, phrases, dates, and numbers. Tokenizer maintains a mapping from tokens to 32 bit identifiers. At this point in the preprocessing pipeline, raw text stream is converted to a stream of integer identifiers. In practice, segments of a document are represented as fixed-length lists of integers.

Tokenized feeds the encoded documents to the indexer. Indexing consists of several phases during which various parts of the index are constructed. Traditionally, the main purpose of the index has been to speed up query processing with an *inverted index*. Inverted index is a mapping from tokens (token IDs) to all documents that contain at least one occurrence of the corresponding token. Using this index, search engine can quickly return all documents that contain tokens of the query. In our case, the index is also used for ranking. The index of Aino is explained in detail in

the following section.

Size of the index grows in proportion to the size of the indexed corpus. Instead of having a single monolithic index, the index is split into smaller blocks, as depicted in Figure 9. These blocks may be distributed to several servers, which allows distributed query processing. The preprocessing pipeline is executed as a batch process, independently from query processing. In a Web search engine, the indices might be updated e.g. once per night. After the indexing pipeline has finished, the index blocks stay constant: Query processor uses them only to match and rank documents according to queries.

Query processing consists of two parts: First, the front-end is responsible for showing the Web interface of the system and for processing incoming HTTP requests. Secondly, there is the back-end that interacts with the indices. The back-end is also responsible for *decorating* the results, which mainly involves generating keyword-in-context (KWIC) [BR99] descriptions for matching documents. Each description is an excerpt from the document that includes at least some of the query tokens. Moreover the query processing back-end is responsible for *caching* the results to speed up processing of frequent queries.

When the user submits a query to the system, the query processing proceeds as explained in section 3.5. However several technical factors make the processing more complicated than the theoretical setting: First, the query must be parsed and validated and it must be checked against the query cache. Secondly, query processing is typically distributed to a cluster of servers, which requires a mechanism to dispatch new queries to the cluster and collect and combine the results. This is explained in section 4.4. Finally, ranked and decorated results are rendered to a HTML page and returned to the user.

Some details were omitted in the above description. Most importantly, Aino sup-

ports stemming of inflected word forms using the Snowball stemming software [Por01]. This increases quality of ranking especially for languages in which inflected words occur frequently, such as in Finnish. Due to language recognition, tokens in each document may be stemmed according to the stemming rules specific to the document's language.

Both the original inflected form and the corresponding stem are stored in the index. This ensures that the document is findable although the stemming process may be inaccurate, or if the user is interested in the inflected form in particular. Thus, stems are used mainly to improve semantic quality of the co-occurrence statistics. In the index, stems are stored as special *meta-tokens*. Meta-tokens may include also other document-specific information, such as date, author, or category. Meta-tokens are used in ranking similarly to ordinary tokens.

4.1 Index

Index is an immutable data structure whose purpose is to organize searchable data to such a layout that search operations can be performed efficiently. In our case, index must support efficient keyword matching and collection of co-occurrence statistics.

The following sections describe how we implemented the index. First, different parts of the index are characterized. Secondly, encoding of the index data is explained and justified. Thirdly, we explain the two main operations on the index, query matching and collection of co-occurrence statistics.

Structure

The index consists of three sections: inverted index, forward index, and position index. Each section has a separate table of contents (TOC) that maps a key to an address in the section body. The index file begins with a header that stores sizes of

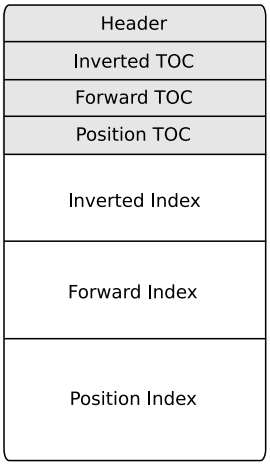

Figure 10: Index structure

the sections and some other bookkeeping information. This structure is presented in Figure 10.

Using the notation from the section 3.2, body of the inverted index stores $\overline{I}_t$ for each $t \in \mathcal{T}$, i.e. for each token t a list of segments in which t occurs. TOC of the inverted index stores tuples (t, B), where B is an offset to the corresponding $\overline{I}_t$ in the body. Since the tuples are ordered by t, we can find $\overline{I}_t$ for any t in $O(\log |\mathcal{T}|)$ time using binary search.

Body of the forward index stores the document segments, or S_D for each $D \in \mathcal{C}$. TOC of the forward index contains tuples (D, B), where B again points at the corresponding segment in the body. Given a segment ID s, the corresponding S_D can be found in the constant time simply by referring to B in the sth entry in the TOC. The key D in the tuple specifies the document ID which this segment belongs to. Thus, the mapping from a segment ID to the corresponding document ID, $s \rightarrow D$, is a constant-time operation as well.

Position index stores locations of occurrences for each token in each document. This information is needed by the phrase queries that require the query tokens to appear in a fixed order in the matching documents. The body contains a list of tokens and

lists of their positions. TOC contains tuples (s, B), where B points at the body as above and s is the ID of the first segment that belongs to this document. Thus, also the mapping from a document ID to its segments, $D \rightarrow s$, is a constant-time operation.

If the corpus is large, a separate index is built for each set of K documents, where K typically varies between 100,000 and 2,000,000 documents, depending on the nature of the corpus. Each index block should fit into the main memory of the server that hosts the index, so that expensive disk IO can be avoided. This form of parallelization is rather efficient as only the TOC of the inverted index contains overlapping entries between the index blocks.

The TOC's and the index sections are saved to a file consequently. When the index is loaded, the header and the TOC's are memory mapped to the process' address space. The area that stays constantly in memory is shown in gray in Figure 10. Parts of the section bodies are brought to the process' address space on demand basis. This makes possible to use indices that are larger than 3GB, which is the maximum size of the address space available for a single process in Linux. Thus the only limitation for the index size is the total size of the TOC's and available disk space. However, indices that do not fit into the main memory are impractically slow for any realistic multi-user setting.

Encoding

The index is made of lists of identifiers. An identifier list, or an array of 32-bit integers, is the only data type used in the index. Each index section employs lists for a particular purpose: Inverted index contains lists of segment IDs for each token ID, forward index contains lists of token IDs for each segment, and position index contains a list of positions for each token in a document.

A straightforward approach would be to encode the lists as such. A plain array of CPU word-width values is extremely efficient to access. The downside is that an index that uses 32 bits per value takes a lot of space. Moreover, the space is wasted for nothing, since the index seldom contains up to 4 billion segments or tokens, which would justify the need for 32-bit values. Before building the index, one could calculate the maximum number of bits needed per value and use, say, only 11 bits per item. However, the benefits of easy and efficient addressing would be lost since the items would not be aligned to byte boundaries. Also, quite likely most of the values are smaller than the maximum, thus using the maximum number of bits per value is wasteful.

A well-known solution to this problem is *Delta Coding* [BR99]. First, list of identifiers is sorted to ascending order

$$(v_1, v_2, v_n), \text{where } v_i > v_{i-1}. \tag{26}$$

Now we may equivalently present the list as differences (or deltas, hence the name) of each pair of items

$$(w_1, w_2, w_n), \text{where } w_i = v_i - v_{i-1}. \tag{27}$$

The benefit is that even though individual items may have arbitrarily large values, their differences map to a smaller scale. In practice, in the index small deltas are vastly more common than large ones, which can be seen in Figure 11. The figure shows the empirical distribution of deltas for the aforementioned Wikipedia subset. The upper distribution is for the forward index, showing deltas for token IDs, and the lower graph shows deltas of segment IDs in the inverted index. The peculiar peak in the upper graph is due to a gap in the token ID mapping: Here the most frequent tokens are assigned an ID below 2000 and the other tokens an ID above that. The peak corresponds to the delta between the frequent and non-frequent IDs.

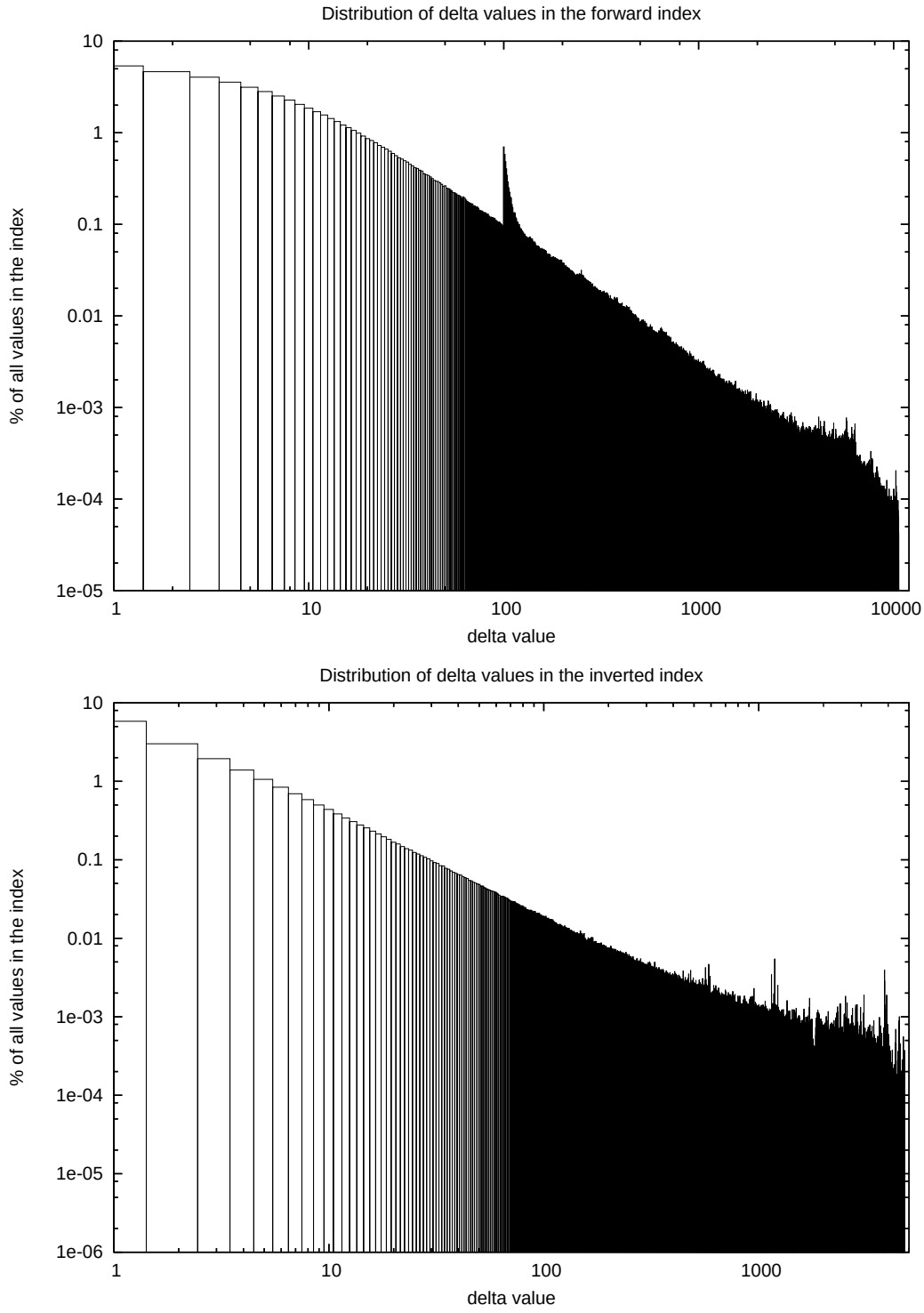

Figure 11: Distribution of delta values

We need an efficient way to encode the deltas so that the frequent small deltas are assigned a small number of bits per value. We might use a general-purpose *entropy encoding* scheme, such as *Huffman* or *Arithmetic coding* to assign codewords to values according to their probability of occurrence [Mac03]. However, since the distribution of values is known beforehand and it follows roughly exponential distribution so that small values are much more probable than large ones, we may use some static coding method for speed and simplicity.

Golomb coding is a static coding scheme that is optimal for geometric distributions [WMB99]. Each input value x is divided into two parts, the quotient q and the remainder r:

$$\begin{aligned} q &= \left\lfloor \frac{(x-1)}{M} \right\rfloor, \\ r &= x - qM - 1. \end{aligned} \tag{28}$$

The quotient q is encoded in unary coding and the remainder r in truncated binary encoding that is a slightly more efficient version of the normal binary code. Here M is a tunable parameter. In practice, we estimate the best M for each list to be encoded and store it together with the list.

If we choose M to be a power of two, division becomes a bit-shift operation and the remainder can be found with a single bit-mask operation. This special case, which yields an efficient implementation, is called *Rice coding*. This code, together with delta coding, is used to encode lists of identifiers in the index.

Compared to plain 32-bit values, Rice coding gives compression ratio of approximately two in the forward index and over three in the inverted index. Thus, the encoding makes the index considerably smaller than the corresponding index with 32-bits per value. However, disk space is not a scarce resource nowadays. Indices are not moved during their lifetime, so the space requirement is not a major consideration with this respect either. In contrast, we are concerned about speed of

usage.

Space and time efficiencies are interrelated via the memory hierarchy in modern computers. The closer an item is stored to the CPU, the faster it can be accessed. A modern X86-64 CPU has 16×8 bytes of space in the general-purpose registers, which is the fastest storage area, and typically 64KB in the L-1 cache and one megabyte of space in the L-2 cache that are the next fastest storages. A server in a computation cluster may have 4GB of RAM and some 500GB of local disk space, which is partly cached to the main memory by the operating system.

However, there are many factors, such as quality of the implementation, compiler settings, and configuration of the operating system, which affect how efficiently the memory hierarchy can be utilized. To better understand the behavior of different encodings in practice, we have made some empirical measurements with the Wikipedia subset. The measurements were taken on a server with 2.1GHz Intel Core 2 Duo CPU with 2MB L-2 cache and 4GB of RAM.

To simulate real index usage, we performed two kinds of test runs: First, all items in the index were accessed sequentially from the first to the last (label "full" in the graphs). Secondly, items were accessed sequentially but randomly skipping some 10-20 items, which simulates processing of a subset of the index ("random"). Each data point is an average of four test runs, each of which was performed with a different random seed. Identical test runs, with identical random seeds, were performed for a delta / rice -encoded index ("rice") and for an index with 32-bit values ("32bit"). Both the indices contained identical data. Index size varied from 100,000 segments to 100,000,000 segments. The largest index took 19GB with 32-bit values and 5.3GB with rice encoding.

Figure 12 shows processing times for different test runs when the index is small enough to fit into the disk cache as whole. One can notice that a sequential full

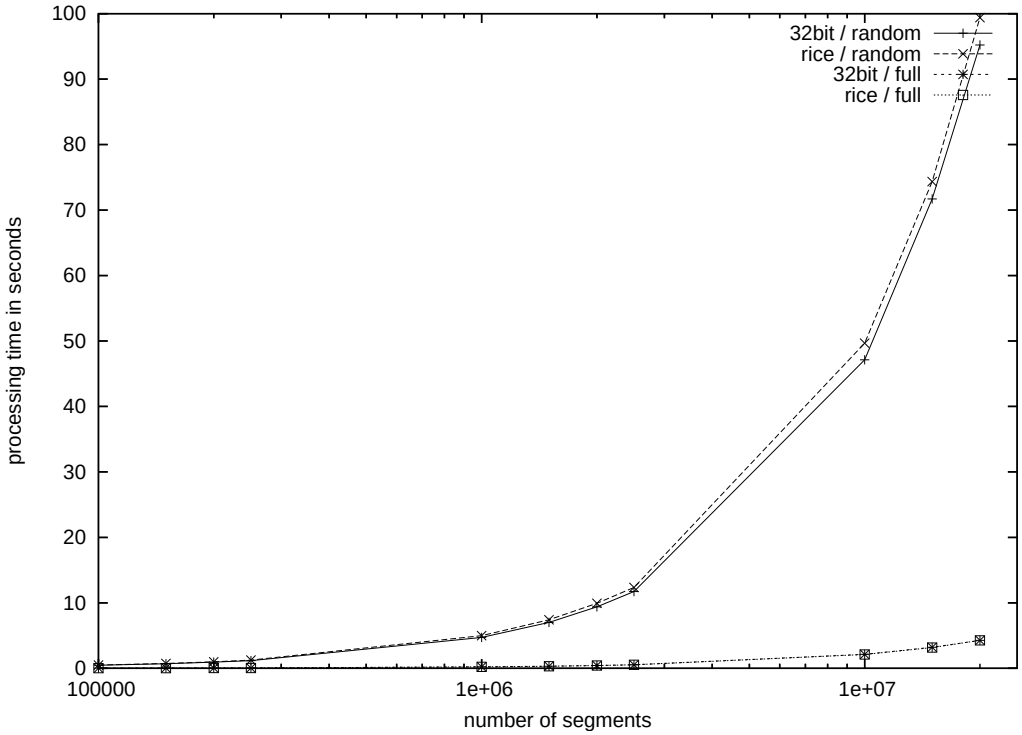

Figure 12: Decoding performance with an in-memory index

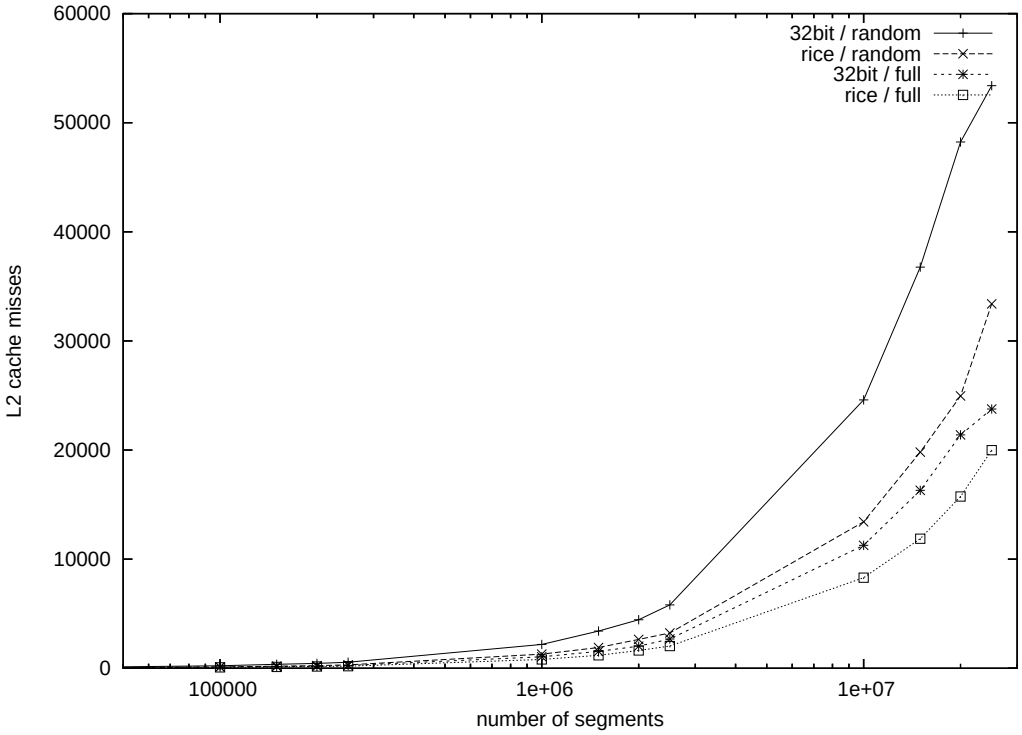

Figure 13: Frequency of L2-cache misses in decoding

sweep over the index is the most cache-friendly, and fastest, way to access the data. Random access becomes increasingly expensive with large indices, when the disk cache does not already contain the requested data and it must be fetched from the disk first.

The most important characteristic of Figure 12 is that the processing times are approximately the same for the rice-encoded index and the 32-bit version. This is not evident, as decoding the rice / delta-encoded values takes 20-30 times more CPU cycles than using the 32-bit values directly. The main cause for this is that the rice coding compensates the larger decoding cost with more efficient cache utilization. Given compression rations of 2-3, we may fit twice as many rice-encoded document segments in the L-2 cache compared to the sparser segments using 32-bit values.

Modern CPUs include a Performance Measurement Unit (PMU) that collects various statistics on the CPU performance during actual workload. We used OProfile module for Linux [Lev02] to sample the number of L-2 cache misses with different test runs. The results are depicted in Figure 13. As suggested by Figure 12, the full sweep over the index manages to use the cache more efficiently. In this case, rice encoding produces approximately 50% less cache misses, which is comparable to the actual compression ratio. When segments are processed in short random intervals, rice encoding is clearly more efficient. In this case, rice encoding increases the probability of hitting the next segment in the cache, as a larger number of segments fits into the cache at once. Correspondingly, a new segment request can easily reach over existing 32-bit segments in the cache and cause an expensive cache flush.

Similarly, a dense coding scheme is beneficial if the index is much larger than the available disk cache. This is the case in Figure 14. Whereas in the case of Figure 12 the index was memory mapped as whole for processing, here the index is processed in a block-by-block manner. The disk cache may selectively keep some pages in memory. As the runs were repeated four times for each data point with different

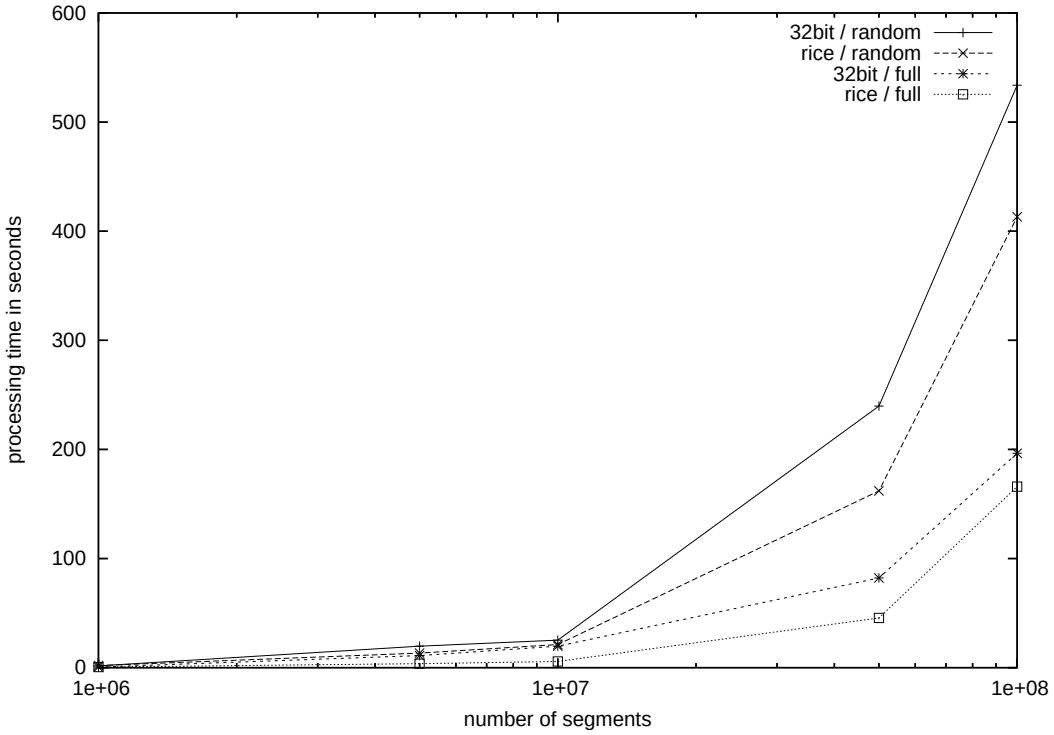

Figure 14: Decoding performance with an on-disk index

random seeds, rice-encoded index benefited from a larger amount of segments in the cached pages.

To conclude, with large corpora dense encoding is justified. If the index is small enough to fit into the main memory with 32-bit values, a dense encoding scheme does not bring any benefits. However, if the corpus is large, dense encoding makes possible to process more segments in the main memory of a single server with minimal performance hit. In any case, one should make sure that the index fits into the main memory as the gap between the disk and CPU speed is enormous.

Operations

In section 3.5 we briefly introduced function $match(D, \mathcal{K})$ whose purpose was to determine whether query keys $\mathcal{K}$ match with a document D. In the following, we describe how *match* is implemented. Our "query language" supports four different

match operations:

keyword(s) Given a query $Q = (q_1, q_2, ..., q_n)$, each token is first mapped to the corresponding token ID using binary search in the lexicon. Then inverted lists are retrieved from the inverted index. The result set is formed as follows:

$$\mathcal{R} = \bigcap_{q \in Q} \overline{I}_q. \tag{29}$$

In total, keyword matching requires two $O(\log |\mathcal{T}|)$ binary search operations per query token and the final intersection that requires $O(|Q|M \log N)$ operations where $M = |\min_{q \in Q} \overline{I}_q|$ and $N = |\max_{q \in Q} \overline{I}_q|$ when intersection is implemented with binary search.

negation Aino supports negation operator, $-$, which excludes the given token x from the result set. This is implemented by treating set $\overline{I}_x$ as its own complement when computing the keyword intersection as above.

range The system for e-mail search, which is described in section 5.2, lets the user retrieve all emails received over a given time period. This is implemented by assigning each document (e-mail) a meta-token that corresponds to the timestamp of an e-mail. A range of consecutive meta-tokens IDs were reserved for this purpose. In this case, a range query was implemented as follows:

$$\mathcal{R} = \bigcup_{i=F}^{L} \overline{I}_i, \tag{30}$$

where F and L are the start and end date IDs correspondingly. The time requirement is $O((L - F) \log |\mathcal{T}| + |\mathcal{R}|)$.

phrase A phrase query, e.g. "George W. Bush" is matched using the position index. Let $P = (p_1, p_2, ..., p_n)$ denote the phrase tokens. First, the intersection of inverted lists of $p \in P$ is computed as with normal keywords. This returns a list of documents that contain all the phrase tokens in some order. For each

document in the intersection, we then validate that the tokens occur in the desired order.

Let L_p denote a list of locations for token p in a document D. Now we may pick the rarest phrase token in the document, $M = \min_{p \in P} |L_p|$, and check the locations of the other tokens with respect to it. Since the check can be done with binary search, the order of phrase tokens can be validated in $O(M(|P| - 1) \log N)$ time per document, where $N = \max_{p \in P} |L_p|$. However, since $|P|$ and M are typically really small, almost always less than ten, the validation is a reasonably fast operation.

Next, we explain the second operation supported by the index, namely collection of co-occurrence statistics for content-based ranking. Recall the original definition for the co-occurrence count between tokens i and j in Equation 5, namely $\Lambda_{ij} = |I_i \cap I_j|$. Using the inverted index, we could evaluate this value in a straightforward manner in $O(log|\mathcal{T}| + M \log N)$ time, where $M = \min(|I_i|, |I_j|)$ and $N = \max(|I_i|, |I_j|)$. Here we assume that a simple set-intersection algorithm is used: For each item in the shorter inverted list, we check whether the item exists in the longer list using binary search. The additional $log|\mathcal{T}|$ term is due to finding the inverted lists in the first place.

However, in order to rank documents with respect to a query token q, we would have to evaluate all possible co-occurrences with q, which would require $O(|\mathcal{T}|M \log N)$ time where $N = \max_{t \in \mathcal{T}} |I_t|$ or $O(|\mathcal{T}|N)$ if hashing was used. Given that number of tokens $|\mathcal{T}|$ may be in the order of millions and the longest inverted list N in the same scale, this cannot be considered a particularly efficient approach.

Let us consider another approach. We may utilize the fact that we are interested in *all* tokens that co-occur with a given query token q, namely $T_q = \{t \in D | D \in I_q\}$. As one can easily see from this formulation, set T_q can be constructed by going

through all the documents in I_q. Moreover, one can compute all values $\Lambda_{qt}, t \in T_q$ by going through documents $D \in I_q$ and counting occurrences of all tokens in the documents. By using the forward index, we can collect all co-occurrence statistics for q in $O(K|\overline{I}_q|)$ time, where K is the segment size.

Multi-token queries are handled in a similar manner. Instead of going through documents $D \in I_q$, we process the documents in the cueset, $D \in \mathbb{Q}$. An interesting side-effect of this approach is that the cost to collect the statistics is directly proportional to the frequency of the query token. We may utilize this fact to speed up processing, as we will suggest in section 4.4.

4.2 Brute-Force Algorithm

Recall the formal overview on how our search engine processes a query in section 3.5. First, the query is parsed to keys $\mathcal{K}$ and cues Q. Then, the result set $\mathcal{R}$ is formed that includes documents that match to the keys $\mathcal{K}$. The cues Q are used to form a cue set $\mathbb{Q}$ of documents from which the co-occurrence statistics are collected. Using the collected statistics, documents in the resultset are scored, ranked, and finally shown to the user.

In a full-fledged search engine, query processing involves many other non-trivial details in addition to the above operations, such as caching and snippet generation. Most of these tasks are handled by Query Interface, as depicted in Figure 9. However, here we concentrate on operations that are directly related to AinoRank, which is the main contribution of this thesis.

We may divide query processing to three phases:

1. Matching
2. Scoring
3. Ranking

Algorithm 1 Brute-force scoring algorithm

```
1: procedure BruteScore(Q̄)
2:     Ω ← ∅
3:     for all S_D ∈ Q̄ do
4:         for all t ∈ S_D do
5:             Ω[t] ← Ω[t] + 1                    ▷ Accumulate |Q̄ ∩ Ī_t|
6:         end for
7:     end for
8:     for all t ∈ Ω do
9:         Ω[t] ← (1/|I_t|) Ω[t]                  ▷ Normalize the token scores
10:    end for
11:    return Ω
12: end procedure
```

Matching, which produces the result set $\mathcal{R}$, is a rather straightforward operation, as described in the previous section. When the inverted index and the position index are used, involved operations are not particularly demanding computationally. The core operation is set intersection – a state-of-the-art algorithm to this problem is presented e.g. in [BY04].

In the following, we present an algorithm for scoring, plus a sketch of an optimized version, and two different algorithms for ranking. The first algorithms for scoring and ranking, which are presented in this section, are rather straightforward, brute-force incarnations of the ideas presented in the earlier sections.

Algorithm 1 is used to compute a scoretable

$$\Omega = \{P(\mathbb{Q}|t)|P(\mathbb{Q}|t) \neq 0, \forall t \in \mathcal{T}\}, \tag{31}$$

where $P(\mathbb{Q}|t)$ is defined as in Equation 20. Considering the numerator $|\mathbb{Q} \cap I_t|$ in Equation 20, one can notice that the score can be non-zero only if $S_D \in \mathbb{Q}$. Thus, it suffices that the algorithm loops through all $S_D \in \mathbb{Q}$. Nominator values are accumulated on line 5 and the final score is computed on line 9. The algorithm finishes by returning the final scoretable.

Let us analyze time complexity of Algorithm 1. Individual scoretable items $\Omega[t]$

Algorithm 2 Brute-force ranking algorithm

```
1:  procedure BruteRank(Ω, R)
2:      S_R = ∅
3:      for all D ∈ R do
4:          S_D^Q ← 0
5:          for all S ∈ S_D do
6:              for all t ∈ S do
7:                  S_D^Q ← S_D^Q + Ω[t]          ▷ Accumulate the document score
8:              end for
9:          end for
10:         S_D^Q ← S_D^Q 1/|S_D|                 ▷ Normalize the document score
11:         S_R ← S_R ∪ (D, S_D^Q)
12:     end for
13:     τ(S_R) ← sort(S_R)
14:     return τ(S_R)
15: end procedure
```

can be accessed in $O(\log|\mathcal{T}|)$ time using any tree-structured array. Our current implementation uses Judy arrays that are based on cache-efficient 256-ary digital tries [Bas04]. Token frequencies $|I_t|$ are stored in the inverted index and they can be accessed similarly to inverted lists. The first two loops require a sweep over the forward index. In total, lines 1-7 can be performed in $O(K|\overline{\mathbb{Q}}|\log|\mathcal{T}|)$ time where K is the segment size. Score normalization on lines 8-10 can be performed in $O(|\mathcal{T}|\log|\mathcal{T}|)$ time. It is worth of noticing that the only query-dependent factor in the time requirement is $|\overline{\mathbb{Q}}|$, which corresponds to the frequency of query tokens.

Algorithm 2 presents a brute-force ranking algorithm that assigns scores to documents based on the scoretable Ω. The algorithm implements the document score of Equation 22. The algorithm loops through documents in the given resultset $\mathcal{R}$ and for each document D, it loops through its segments. Each token in the segments may increase the document score $\mathcal{S}_D^{\mathbb{Q}}$ by an amount specified by the token score $\Omega[t]$. The document score is normalized by inverse number of segments. Finally, the documents $\mathcal{S}_{\mathcal{R}}$ are sorted according to their scores and returned to the caller.

Time complexity is dominated by the loops. A single document can be scored in

$O(K|\mathbf{S}_D| \log |\mathcal{T}|)$ time, on lines 4-10. Correspondingly, the full resultset is scored in $O(|\mathcal{R}|KN \log |\mathcal{T}|)$ time where $N = \max_{D \in \mathcal{R}} |\mathbf{S}_D|$. Sorting can be performed in $O(|\mathcal{R}|)$ time, for instance with Radix sort [CLR92], since integer scores can be used as sort keys.

In total, the cost of ranking with algorithms 1 and 2 is as follows.

$$\begin{aligned} O(\text{ranking}) &= \overbrace{O(K|\overline{\mathbb{Q}}| \log |\mathcal{T}|)}^{scoretable\ \Omega} + \overbrace{O(|\mathcal{T}| \log |\mathcal{T}|)}^{\Omega\ normalization} + \overbrace{O(|\mathcal{R}|KN \log |\mathcal{T}|)}^{document\ scoring} + \overbrace{O(|\mathcal{R}|)}^{sorting} \\ &= O(log|\mathcal{T}|(K\,(|\mathbb{Q}| + |\mathcal{R}|N) + |\mathcal{T}|) + |\mathcal{R}|). \end{aligned} \tag{32}$$

The cost is dominated by two large factors: $|\mathbb{Q}|$ and $|\mathcal{R}|$. The former factor is directly proportional to frequency of cue tokens and the latter is directly proportional to frequency of keys. Hence, a query in which both keys and cues are specific, for instance "haggis /E113", can be evaluated with a reasonably small number of steps whereas a really broad query, say "apple /food", takes a lot of resources to evaluate. Fortunately, this phenomenon hints at many kinds of optimizations.

4.3 Top-$\mathcal{K}$ Algorithm

In order to understand how the previous algorithms could be optimized, let us see how they behave in practice. Figure 15 shows how token scores Ω, produced by Algorithm 1 above, are distributed in case of three different queries with varying frequencies. The most frequent query token, `age`, occurs in some 50,000 documents in the Wikipedia subset. The second most frequent token, `neolithic`, occurs in 500 documents and `damiens` in ten documents. The queries have 185,000, 17,000, and 1050 co-occurring tokens, correspondingly. A notable characteristic in the distributions is that after a large number of tokens with a score of 1.0, there is a steep drop in the scores. This suggests that only a small portion of co-occurring tokens might have a notable effect in the document scores.

This hypothesis is further supported by Figure 16 that shows how score mass cu-

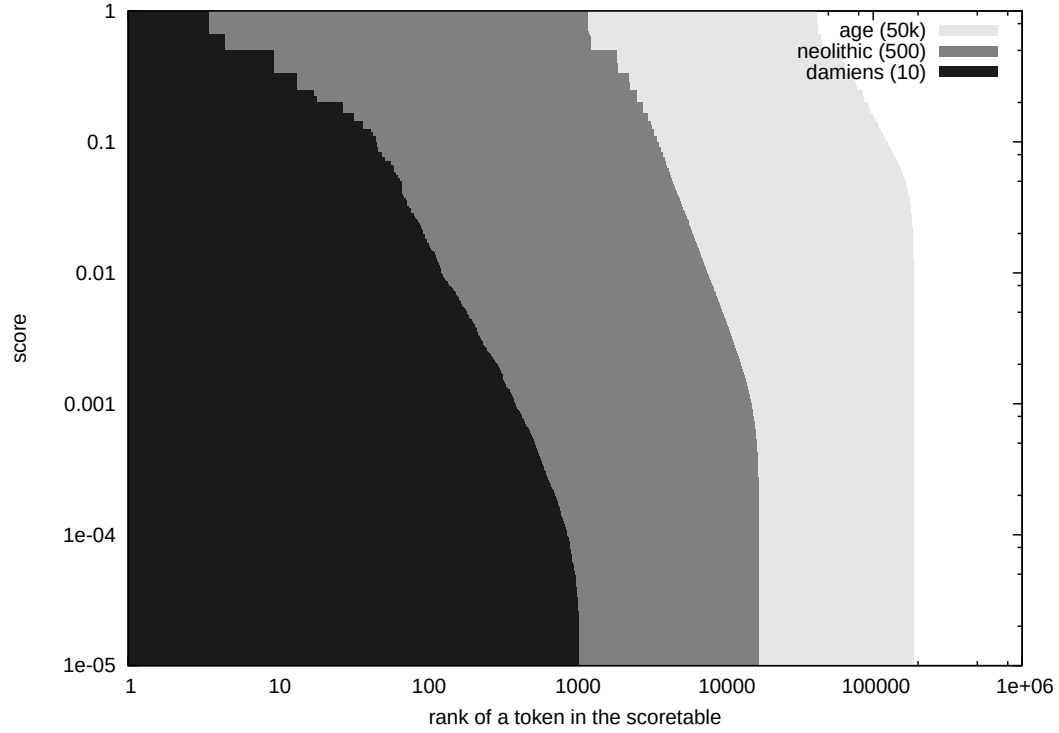

Figure 15: Distribution of token scores Ω for three queries

mulates with the above queries. This figure shows only the best half of the scores of Figure 15. One can see that the best half of the scores make almost 100% of the total score mass, besides the most frequent query `age` in which case the percentage is about 90%. Based on these figures, it seems that results of the previous algorithms could be approximated by processing only the highest scoring tokens of Ω.

Above we recognized that that the size of the result set $|\mathcal{R}|$ is a major factor in the cost of ranking. Each of the documents $D \in \mathcal{R}$ has to be scored token by token and finally the results must be sorted. Then, the ranked results are presented in a traditional Web search interface that shows only ten results at time, starting from the top ten results.

In practice, the users rarely bother to see more than the first ten results, as we exemplified in Section 3.1. Thus, to a large extent, computing *exact scores* for all documents in $\mathcal{R}$ is meaningless, as all we need is the *best* $\mathcal{K}$ *results in order*.

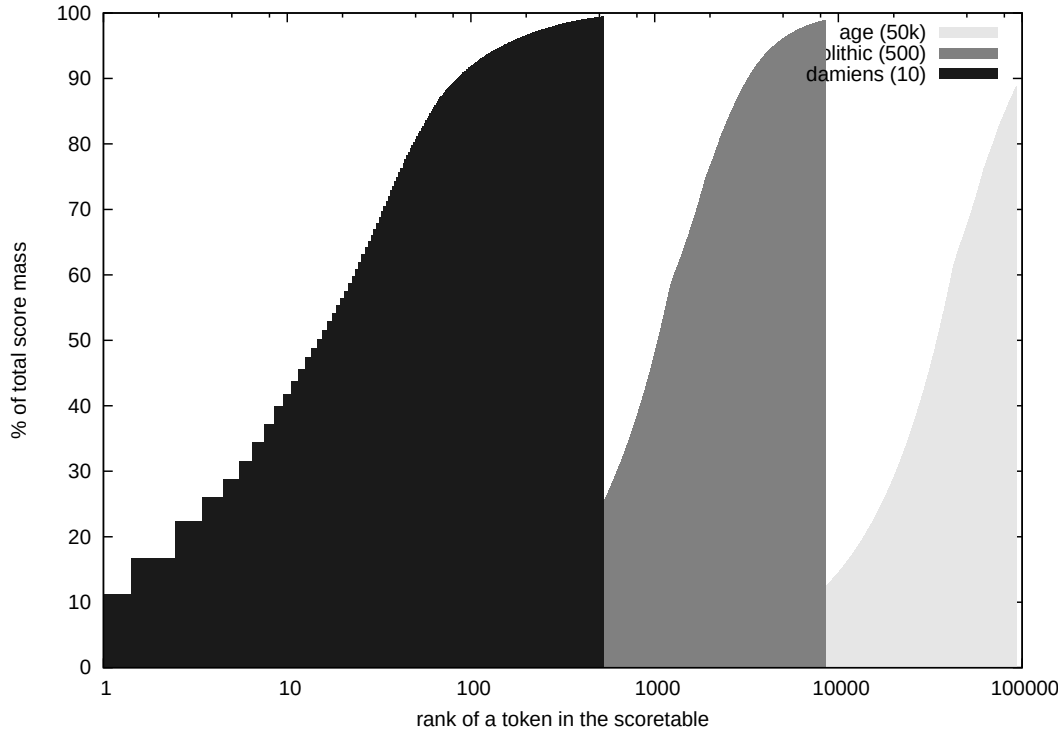

Figure 16: Cumulative score mass of the best half of token scores Ω for three queries

Given that the document score function is a linear sum of form

$$S_D = \mathcal{Z}_D \sum_{S \in \mathbf{S}_D} \sum_{t \in S} \phi_t, \tag{33}$$

where ϕ_t is a token score independent of the document and $\mathcal{Z}_D$ is a document-dependent normalization factor, the highest ϕ_t are likely to determine the highest S_D. Theoretically, high token scores could distribute uniformly over the resultset, but in practice in real-world corpora high scoring tokens tend to be also highly correlated. Thus, high scores tend to cumulate to a small number of documents. This phenomenon is visible in Figure 17 that shows how document scores are distributed over the top-100 documents after the highest scoring half of Ω has been taken into account, which accounts for almost 100% of the score mass. The figure suggests that one could find the highest scoring $\mathcal{K}$ documents just by considering a high scoring subset of Ω. This would bring considerable savings in time as we would not have to process the full resultset $\mathcal{R}$.

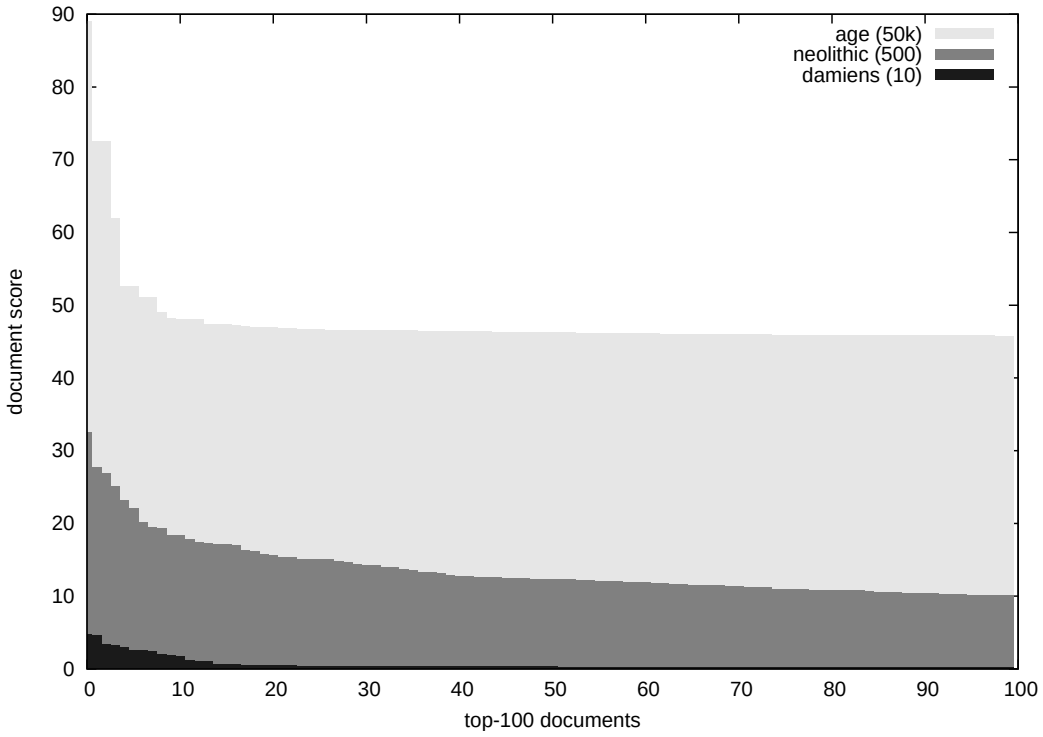

Figure 17: Distribution of document scores after the highest scoring half of Ω has been processed for three queries

Algorithm 3 Top-$\mathcal{K}$ ranking algorithm

1: **procedure** TopRank($\Omega, \mathcal{R}, \mathcal{K}, \Delta, \beta_{size}$)
2: $\tau(\Omega) \leftarrow sort(\Omega)$ ▷ Scores are sorted in descending order
3: $\alpha, \beta, \mathcal{S} \leftarrow \emptyset$
4: $i \leftarrow 0$
5: **repeat**
6: $i \leftarrow i + 1$
7: $t \leftarrow \tau(\Omega)[i]$ ▷ Pick the next best token t
8: **for all** $\{D \in I_t | D \in \mathcal{R} \wedge D \notin \alpha\}$ **do**
9: $\mathcal{S}_D \leftarrow \mathcal{S}_D + \Omega[t]$ ▷ Increase score for document D
10: **if** $\mathcal{S}_D > HeapMin(\alpha)$ **then** ▷ Top-$\mathcal{K}$ candidate?
11: $\mathcal{S}_D \leftarrow DocumentScore(D, \Omega)$
12: $HeapUpdate(\alpha,\ \mathcal{K},\ D,\ \mathcal{S}_D)$
13: **end if**
14: **if** $\mathcal{S}_D > HeapMin(\beta)$ **then** ▷ Top-β_{size} candidate?
15: $HeapUpdate(\beta,\ \beta_{size},\ D,\ \mathcal{S}_D)$
16: **end if**
17: **end for**
18: $\delta \leftarrow \sum_{j=i}^{i+\Delta} \Omega[j]$ ▷ Move score potential by one token
19: **until** $|\beta| = \beta_{size} \wedge HeapMin(\alpha) > HeapMin(\beta) + \delta$
20: $\mathcal{S}_{\mathcal{R}'} = \{D \in \mathcal{C} | D \in \beta\}$
21: $\tau(\mathcal{S}_{\mathcal{R}'}) \leftarrow BruteRank(\Omega, \mathcal{R}')$
22: **return** $\tau(\mathcal{S}_{\mathcal{R}'})$
23: **end procedure**

Algorithm 3 presents a solution that guarantees that the $\mathcal{K}$ highest ranking documents have the highest scores amongst all $D \in \mathcal{R}$ and they are returned in the correct order. Also, the next-best scoring set of documents is probably correct and in the correct order, although this is not strictly guaranteed.

The algorithm works as follows. Let us denote by α a candidate set of $\mathcal{K}$ highest scoring documents. The algorithm continues until we can be sure that α actually contains the best documents. Let us denote by β a set of documents that are likely to get promoted to α. The algorithm stops when there is no way for the worst document in β to get promoted to α.

The algorithm assumes that the token scores in Ω are sorted in descending order – this in ensured on line 1. The algorithm proceeds by processing one token at

time from Ω, from the highest scoring one to the worst (line 7). For each token, the score mass is distributed to all documents in which the token occurs, unless the document belongs to α already (lines 8-9). If a document's score is higher than that of the lowest scoring document in the α set, the document is promoted to α. Function *HeapUpdate* replaces the worst-scoring document in the given set α with a new document D, having score S_D. If the set contains less than $\mathcal{K}$ documents, no previous document is purged from the set. To speed up computation, we compute the exact final score of document D on line 11 before adding it to α. This aims at widening the score gap between the sets α and β. If the document score is not enough to reach α, it might be enough for β anyway (line 14) that is expanded using the same *HeapUpdate* function.

The score window δ, computed on line 18, is crucial for determining when α cannot change anymore. At any step i, δ contains the maximum amount of the score mass that can be still allocated to a single document. In other words, δ represents the available potential for any document to get promoted to the top-$\mathcal{K}$ set α. Since token scores Ω are sorted in descending order, the highest possible score mass, which is yet to be distributed to documents, is in range $[i, i + \Delta]$ where Δ is a free parameter. The end condition on line 19 terminates the loop when we have distributed score to a minimum number of documents β_{size} and when the lowest-scoring document in β_{size} could not get promoted to α even if it got the full potential δ of the score mass. Since all documents $D \notin \beta$ have lower scores than the ones in β, the end condition means that there is not any document that could raise to α and which would be not in β already. Necessarily, this implies that the set β contains the top-$\mathcal{K}$ documents. The actual top-$\mathcal{K}$ set is then determined by computing the exact document scores for all $D \in \beta$ using Algorithm 2.

There are two free parameters, Δ and β_{size}, that need to be set beforehand. The latter, β_{size}, does not affect correctness of the results, but it can affect the execution

time. The lower the value, the more difficult it is to grow the score gap large enough between β and α to reach the end condition. On the other hand, the higher the value, the more exact document scores must be computed on line 21. In practice, we have used values $\mathcal{K} = 10$ and $\beta_{size} = 1000$. This has also the positive effect that the first hundreds of results are likely to be correct, which should be enough even for the most exploratory user.

Parameter Δ controls the score potential. It represents our guess on the expected amount of next-best scoring tokens that a document can reasonably contain. If the algorithm was segment-based, i.e. I_t would be $\overline{I}_t$ on line 8, we could set $\Delta = K$ that would guarantee that the results are always exactly correct. However, it is highly unlikely that a document will contain high-scoring tokens *only*. Thus, typically we set $\Delta = 50$ that is our sophisticated guess on the maximum number of high-scoring tokens in a segment of $K = 300$ tokens. Note that a slightly incorrect setting might only trigger the end condition a bit too early, which seldom leads to drastically misleading results.

Let us analyze time complexity of the algorithm. Scoretable Ω can be sorted in $O(|\Omega|)$ time using e.g. Radix sort. Sets α and β are implemented as Fibonacci heaps [CLR92], which makes it possible to implement function $HeapMin$ in a constant amortized time and $HeapUpdate$ in $O(\log \beta_{size})$ time. The latter is practically a constant as well, since β_{size} is a fixed parameter.

If function *DocumentScore* is not considered (as it is not required by the algorithm), the loop on lines 8-19 operates in $O(|\mathcal{R}|)$ amortized time. Here we consider both $HeapMin$ and $HeapUpdate$ to be constant-time operations. In the worst case, the outer loop on lines 5-19 goes through all tokens in Ω, which results to $O(|\Omega||\mathcal{R}|)$ time complexity for both the loops together. Since $\mathcal{R}'$ is of a constant size β_{size}, the exact scores can be computed in $O(KN \log |\mathcal{T}|)$ time on line 21, where $N = \max_{D \in \mathcal{R}} |\mathbf{S}_D|$ as above.

Total worst-case time complexity $O(|\Omega| + |\Omega||\mathcal{R}| + KN \log |\mathcal{T}|)$ is unrealistically pessimistic. In practice, inverted lists for the highest scoring tokens are short, at least far from the $|\mathcal{R}|$ scale. Typically, the end condition is triggered after half of the tokens in Ω are processed. Thus, we can approximate that the average time complexity is in scale $\Theta((M+1)|\Omega| + KN \log |\mathcal{T}|)$ where M is the average length of inverted lists ($M = 47.25$ for the Wikipedia subset). Compared to the time complexity of the brute-force ranking algorithm in Equation 32 this is a remarkable improvement when $|\mathcal{R}|$ is large, say in the order of tens of thousands. If $|\mathcal{R}|$ is small, especially when $|\mathcal{R}| < \beta_{size}$, Algorithm 2 is likely to be faster and we may use it instead. In practice, we make the choice on query by query basis.

4.4 Discussion

Algorithms 1, 2, and 3 show that it is possible to use the full co-occurrence matrix for content-based ranking by computing parts of the matrix on fly. When the index has a suitable layout and it is encoded in an efficient manner, performing ranking on-line is not infeasible. Efficient dynamic ranking is made possible especially by the fact that the exponential distribution of token frequencies is reflected in the token and document scores, which leads to an efficient algorithm. Moreover, since a search engine may return only the top ranking results, only a small subset of the matching documents have to be scored, which reduces the computational load even more.

Figure 18 shows a comparison of the two ranking algorithms. On the x-axis, which is the rightmost horizontal axis, one can see $|\mathcal{R}|$ i.e. the number of matching documents. On the y-axis, which is the leftmost horizontal axis, one can see frequencies of the single query token. On the z-axis, the time used to rank the resultset either with Algorithm 2, BruteRank, or Algorithm 3, TopRank, is shown.

Time efficiency of BruteRank is linearly dependent on the resultset size, as implied

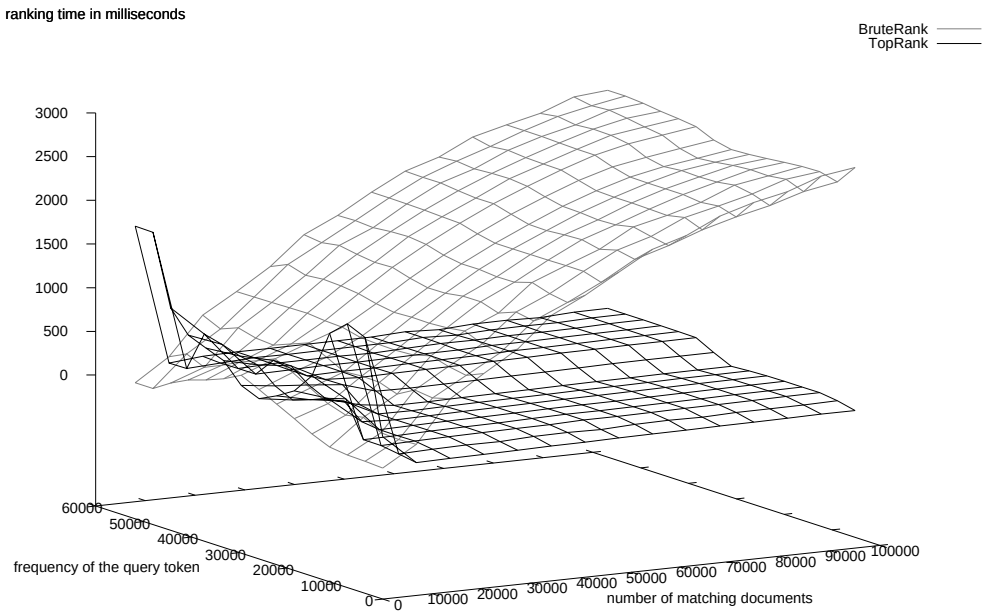

Figure 18: The two ranking algorithms compared

by Equation 32. In contrast, since TopRank ranks only the best $\mathcal{K}$ documents, the resultset size does not have a major impact on it, if the resultset is large enough. When the resultset is really small, TopRank is less efficient than BruteRank.

Both the ranking algorithms depend on the frequency of the query tokens via $|\Omega|$ i.e. the number of co-occurring tokens. In the figure, this can be seen as a small bump on the y-axis. However, the effect is logarithmic in the case of BruteRank and not very strong with TopRank either, even though in the worst-case time efficiency of TopRank is linearly dependent on $|\Omega|$. Actually, with TopRank the effect of $|\Omega|$ becomes stronger when the resultset becomes smaller, as a larger number of tokens in Ω have to be processed in order to differentiate the top ranking documents from the rest.

In practice however, the total ranking time is strongly dependent on $|\Omega|$, as the token scores Ω have to be computed via Algorithm 1, BruteScore. As BruteScore

uses an extremely straightforward approach to collect co-occurrence statistics, which is rather inefficient and wasteful, there should still be room for optimization. In the following, we present an idea how to improve Algorithm 1. After that, we briefly explain how query processing can be parallelized.

Dense Co-occurrences

One can see from Figure 5 that the 10,000 most frequent tokens have two orders of magnitude more co-occurring tokens than the other, less frequent ones. This implies that most of the entries in the forward index are dominated by the most frequent tokens, which is also a natural consequence of the Zipfian distribution of token frequencies, as shown in Figure 4.

The token scoring Algorithm 1 accumulates token occurrences one by one. If a query token is frequent and consequently size of the cueset $\mathbb{Q}$ is large, collecting the co-occurrence statistics is an expensive operation. A straightforward solution is to store a pre-computed co-occurrence matrix for the most frequent tokens separately. Given the Zipfian distribution of token frequencies, there is only a small amount of frequent tokens and consequently the size of the matrix is not astronomical.

Thus we split the forward index section (see Figure 10) into two: An explicit co-occurrence matrix for the N most frequent tokens and a sparse forward index for the infrequent ones. An alternative view to this dichotomy is that we use binary encoding to store frequent co-occurrences and a kind of a unary coding for sparse ones. Actually, one might see a peculiar correspondence between this approach and the two-part Golomb coding that was presented in Equation 28.

Considering Algorithm 1, this makes token scoring a constant-time operation for the N most frequent tokens, as for these, scoretable Ω is pre-computed in the index. Formally, this does not change the worst-case time complexity for the algorithm but

it improves its average-case performance significantly.

Parallelization

Keyword matching is a so called *embarrassingly parallel problem* – it can be parallelized with no particular effort: Each block of documents can be indexed and processed independently from others. If a static ranking scheme is used, document ranking can be distributed as easily. This is the secret behind scalability of the major Web search engines.

In our case, the situation is slightly more complicated. The index blocks depend on each other via the global co-occurrence statistics: First, each block of documents contributes to the global co-occurrence statistics. Secondly, document ranking within each block depends on the common scoretable Ω that is based on the aforementioned statistics. Fortunately, the token and document scoring algorithms can be performed in parallel and only the final results need to be shared.

We may distribute query processing to several servers or slaves, each of which handles a single index block. The query interface or the master node, which is typically hosted by a separate server, distributes tasks to the index nodes and receives the results over TCP or UDP.

In Section 4.2 we introduced the three phases of query processing, namely matching, scoring, and ranking. Correspondingly, distributed query processing can be divided in the following phases:

1. (M) Query distribution
2. (S) Computing co-occurrence statistics
3. (M) Computing scoretable
4. (M) Scoretable distribution
5. (S) Matching

6. (S) Ranking
7. (M) Merging results

Here (M) denotes an operation performed by the master node and (S) an operation performed by a slave. All operations marked with (S) can be performed in parallel. Essentially, these tasks are handled by the same algorithms that were presented above. Tasks performed by the master operate on outputs of the algorithms and their time complexity is either $O(|\Omega|)$ or $(|\mathcal{R}|)$ at most.

Communication costs are dominated by the phase 4, or scoretable distribution. In the worst case, we have to transfer scoretable of scale $O(|\mathcal{T}|)$ to each slave. The cost can be reduced with various encoding schemes and compression methods but given the gigabit Ethernets of today, this is seldom necessary.

The number of slaves required is naturally determined by the size of the corpus. In order to keep response times low, each index block should fit into the main memory (disk cache) of the slave that hosts the block. In practice, this limits the size of a single index block to some 1,000,000 - 2,000,000 documents when a slave has 4GB of RAM. An additional benefit of distributed query processing is that the system is more robust: In case that an index node crashes, the system continues to operate seamlessly with the remaining indices. The system is able to process queries as long as at least one of the index nodes is operational.

We have implemented two different distributed query processing systems for Aino: One in C using UDP multicasts and one in Erlang which is a functional programming language that has built-in support for distributed computing [AVWW96].

5 Demonstrations

In this chapter we introduce three demonstration systems that are based on Aino. The methods of this thesis were born in the Search-Ina-Box project at the Complex Systems Computation Group of the Helsinki Institute for Information Technology. In this project, several demonstrations were made to study and present promising applications of content-based search. The demonstrations presented here were used to evaluate quality and performance of AinoRank with various kinds of real-world corpora. Thus, this chapter serves also as the conclusion, showing how methods presented in this thesis can be, and are, applied in practice.

These demonstrations are implemented by the author, except the Web crawler HooWWWer by Antti Tuominen [Tuo05] that is used in the first demonstrator. The systems have been presented and described in several conference papers, which are only summarized here. The first demonstration, a public Web search engine covering the .FI domain, is described in part in [TS05]. The second system for searching e-mail archives is described in [PTBT05] and demonstrated in [TPT05]. The third system, which is used to search and analyze patents, is developed for the Patent Office of Finland who was a partner in the project.

5.1 Web Search

For any method of information retrieval, Web search is the grandest challenge of them all. There is an infinite amount of documents[3], data is extremely noisy, and any token or document may be relevant to some user. Aino was designed with this challenge in mind in the first place.

In 2005, we collected a corpus of some 4.2 million web pages from the .FI top level domain. For this purpose, a web crawler called HooWWWer [Tuo05] was developed

[3]Quite literally, if the crawler gets stuck in a cycle of the Web graph

Figure 19: Aino.hiit.fi: Front page

in a sister project of Aino. The corpus contained some 12 million tokens and the index took 5GB of space in total.

The demonstrator was publicly available at `Aino.hiit.fi` for five months from May to September in 2005. Query processing was distributed to seven index nodes and one node acted as the query interface. One index node crashed during the demonstration period due to a disk failure, but the system experienced no downtime during its operation – a prime example of the additional robustness gained by distributed query processing. Another, updated demonstrator with improved algorithms and some five million documents was running on Spring 2007.

Figure 19 shows the front page of the `Aino.hiit.fi` web search engine and Figure 20 its result page. Table 5 shows some queries and the corresponding highest scoring entries of the scoretable Ω. The results show how high-scoring tokens correspond to

Figure 20: Aino.hiit.fi: Result page

Query	**Highest scoring entries of Ω**
akrr	akrr05, amklc, krbio, openconf
semanttinen	semanttisen, semanttisten, semanttista, semanttisesti
pragmatiikka	pragmatiikan, semantiikka, fonologia, kontrastiivinen
parturi	kampaamo, kampaamot, maahantuojat, kauneudenhoito
nenä	korva, kurkkutaudit, sisätaudit, naistentaudit
matala	korkea, pensasmainen, kasvista, lämpöisestä
äpy	äpyvän, rähästö, äpyn, wappulehti
yxin	voisittexte, iltajutust, listäkää, burggaballonkin
halonen	tarja, presidentti, tasavallan, halosen

Table 5: Highest scoring tokens for some queries in Aino.hiit.fi

various kinds of linguistic phenomena present in a rich corpus like the Web.

In the following, some example queries are shown with the corresponding top ranking results. The excerpts are the actual snippets produced by the search engine. More examples can be found in [TS05]. Notice how an ambiguous token `jukola` can be disambiguated by using different cues and how cues are used to hint at a certain disposition in the last example.

- `jukola /simeoni`
  1. *Simeoni, liuhuparta, valittaa se "ihmisparka, syntinen, saatana, kurja".*
  2. *Juhani, Tuomas, Aapo, Simeoni...*
  3. *Heikki Kinnunen (Aapo), Heikki Alho (Simeoni), Arno Virtanen (Timo), Ilari Paatso (Lauri) ja Juha Muje ...*
- `jukola /juoksu`
  1. *Nuorten Jukola 2002*
  2. *on tullut tutkittua suunnistuskarttoja (tio-mila, jukola, tanska jne.*
  3. *Jukola-katsastus...*
- `mcdonalds /kamalaa /yäk /kuvottaa`
  1. *Face it, you smell like McDonalds and Wallmart/ By killing you I'm akting globally, doing a small part.*
  2. *liha tulee ulkomailta (siis jos mun mcdonalds tietämys pitää paikkansa).*
  3. *McDonalds on vähän toisenlainen ongelma, terveydellinen ongelma.*
  4. *'ylikanallista mcdonalds'-kulttuuria, joka yksinkertaisesti hävittää molemmat kulttuurit?*

5.2 E-Mail Search

E-mails are a natural target for content-based search. They are textual, abundant, and they lack hyperlinks, which makes link-based ranking schemes unfeasible. In addition to search of the plain textual content, we were interested to study whether other types of information available in the emails, such as the sender, recipient, time and topic of the content could be used to enhance possibilities to find interesting

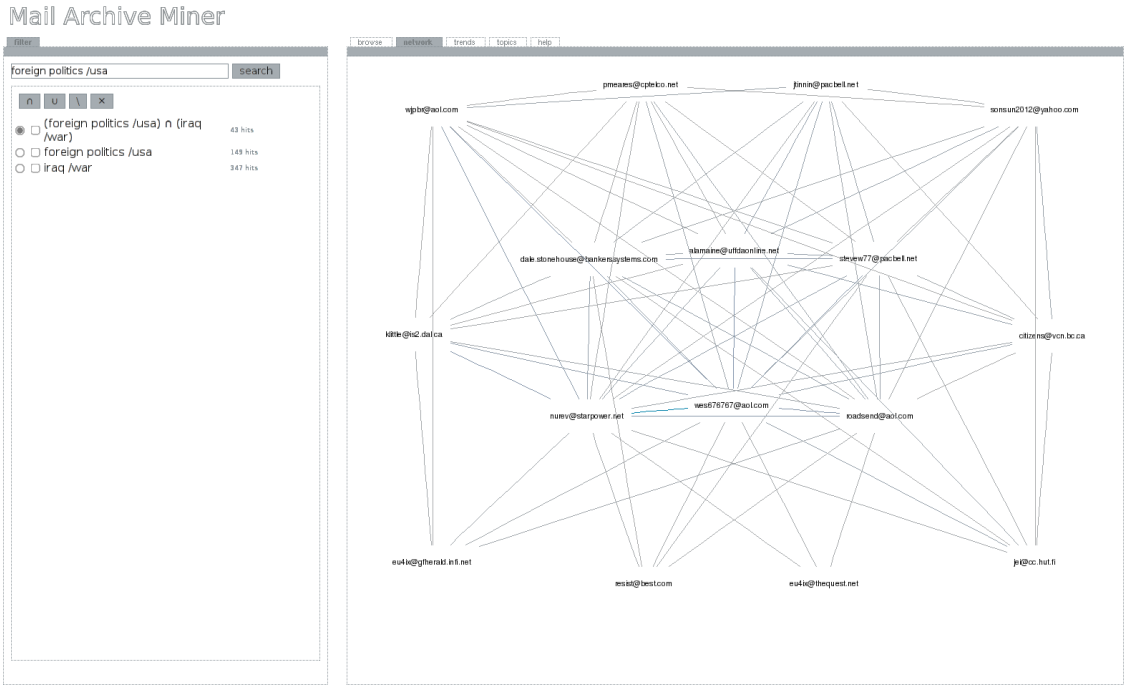

Figure 21: Mail Archive Miner: Social network analysis with AinoRank

content. We envisioned a system that was based on a powerful content-based ranking scheme, namely AinoRank, and on top of that, a rich set of tools was provided to combine, analyze, and refine the results. The system is described in detail in [PTBT05] and it was demonstrated first in [TPT05].

We got access to a large-scale email-archive that contained 20GB of emails from public mailing lists. An index of 2.5 million emails was created which included 4.4 million tokens in total. The index required 3.3GB of space. In addition to AinoRank, the system included a probabilistic model based on Multinomial Principal Component Analysis [BJ06], which was used to capture general themes or topics in the corpus.

On the leftmost list, Figure 21 shows three resultsets, the first of which is made by combining results of the two previous searches. The figure shows a social network that is automatically inferred from the chosen resultset. By choosing a node in the

graph, the user could restrict the results to a specific person or retrieve all emails whose style correspond to that of the chosen person.

5.3 Patent Analysis

The Patent Office of Finland was interested to evaluate feasibility of content-based search for patent documents. Similarly to emails and web pages, there are tens of millions of patent documents publicly available. On the average, a patent document is considerably longer and less noisy than an average email or a web page, which translates into higher quality co-occurrence statistics.

The patent office had some specific needs regarding the system. As it would be used together with a traditional patent database that supports only Boolean queries, it should suggest topical keywords given some tokens as seeds, which in turn could be used to query the original database. Fortunately, this requirement was rather straightforward to fulfill using a slightly modified version of the BruteScore algorithm. The user interface for this function with an example query can be seen in Figure 22.

We indexed a smallish corpus of some 100,000 European patents. The resulting index takes 1.2GB and consists of 1.4 million tokens. Ratio of the number of tokens to the number of documents reflects the exceptional length of an average document. In addition to the keyword suggest mechanism, the system includes functions to find characteristic and discriminative tokens in a patent application and a Keys & Cues -style interface for patent search which is shown in Figure 23.

Figure 22: Patent Cruncher: Keyword sets with AinoRank

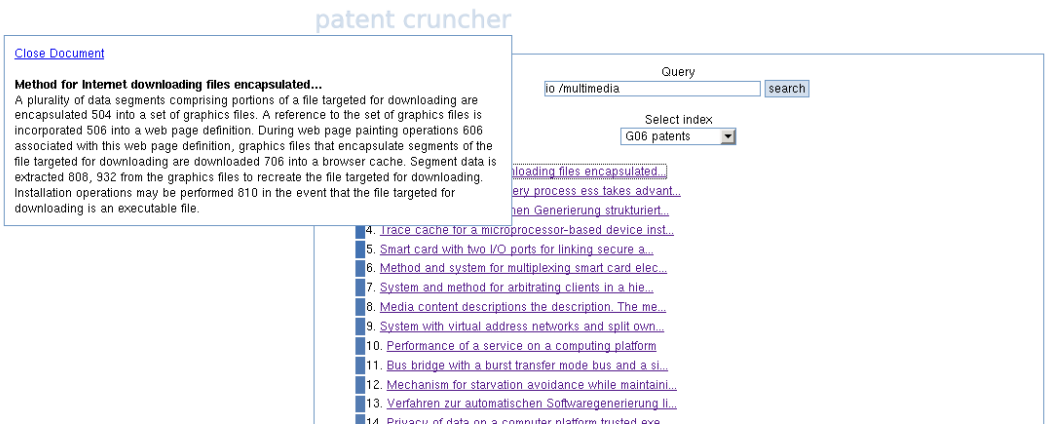

Figure 23: Patent Cruncher: Patent search

6 Conclusions

In the previous chapters we have described a system that helps the user to find interesting content among tens of millions of documents. We started by presenting theoretical and philosophical underpinnings of statistical content-based search. After this, we described a novel content-based ranking method, AinoRank, that comprises of a structured query interface and a document scoring function based on the full co-occurrence matrix of tokens. We gave an efficient algorithm to find the highest scoring documents in the corpus. Finally, we summarized three real-world applications of the system.

Certainly, this work is not conclusive. As we mentioned in the beginning, this work was motivated by the need to have a solid and understandable basis for more sophisticated approaches. Yet, the presented system has proved to be a well-placed stepping stone for various experiments in information retrieval, as exemplified by the Web, e-mail, and patent search applications that we described above.

There are two continuing lines of research: First, we believe that by further utilizing distributional features of data and the scoring functions, we may improve efficiency of the system drastically. Also we believe that the computational load could be reduced by approximating the results instead of computing exact scores and ranks for the documents.

Secondly, concerning usability of the system and quality of the results, we would like the system to be even more transparent to the user than it is currently. Furthermore, we would like to improve the ranking method semantically. The current method does not take the full advantage of distributional features of tokens, which could be derived from the co-occurrence statistics. Finally, we would like to clarify the connection between this method and various probabilistic and information theoretic formalisms.

References

AOL06 AOL search logs, `http://www.aolsearchlogs.com/`, 2006.

AVR02 Amati, G. and Van Rijsbergen, C. J., Probabilistic models of information retrieval based on measuring divergence from randomness. *ACM Transactions on Information Systems*, 20,4(2002), pages 357–389.

AVWW96 Armstrong, J., Virding, R., Wikström, C. and Williams, M., *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.

Bas04 Baskins, D., Judy arrays, `http://judy.sf.net/`, 2004.

BJ06 Buntine, W. and Jakulin, A., Discrete components analysis. In *Subspace, Latent Structure and Feature Selection Techniques*, C. Saunders, M. Grobelnik, S. G. and Shawe-Taylor, J., editors, Springer-Verlag, 2006, pages 1–33.

BL99 Berger, A. and Lafferty, J. D., Information retrieval as statistical translation. In *Research and Development in Information Retrieval*, van Rijsbergen, C. J., editor, Cambridge University Press, 1999, pages 222–229.

BLP+04 Buntine, W., Löfström, J., Perkiö, J., Perttu, S., Poroshin, V., Silander, T., Tirri, H., Tuominen, A. and Tuulos, V., A scalable topic-based open source search engine. *International conference on web intelligence, WI2004*. IEEE Computer Society, 2004, pages 226–234.

BLPV05 Buntine, W., Löfström, J., Perttu, S. and Valtonen, K., Topic-specific scoring of documents for relevant retrieval. *Workshop on Learning in Web Search (LWS 2005)*, 2005, pages 34–41.

BNJ03 Blei, D. M., Ng, A. Y. and Jordan, M. I., Latent dirichlet allocation. *Journal of Machine Learning Research*, 3, pages 993–1022.

BP03 Buntine, W. and Perttu, S., Is multinomial PCA multi-faceted clustering or dimensionality reduction? *Proc. 9th Int. Workshop on Artificial Intelligence and Statistics*, 2003, pages 300–307.

BPT04 Buntine, W., Perttu, S. and Tuulos, V., Using discrete PCA on web pages. *Proc. of the Workshop on Statistical Approaches for Web Mining (SAWM)*, M. Gori, M. C. and Nanni, M., editors, 2004, pages 99–110.

BR99 Baeza-Yates, R. and Ribeiro-Neto, B., *Modern Information Retrieval.* Addison Wesley, 1999.

Bro95 Brown, E. W., Fast evaluation of structured queries for information retrieval. *Proceedings of the 18th annual international ACM SIGIR conference*, New York, NY, USA, 1995, ACM Press, pages 30–38.

BY04 Baeza-Yates, R., A fast set intersection algorithm for sorted sequences. *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching, CPM 2004*, Sahinalp, S., Muthukrishnan, S. and Dogrusoz, U., editors, 2004, pages 400–408.

CCH92 Callan, J., Croft, W. B. and Harding, S. M., The INQUERY retrieval system. *Proceedings of DEXA-92, 3rd International Conference on Database and Expert Systems Applications*, 1992, pages 78–83.

Cho69 Chomsky, N., *Syntactic Structures.* Mouton, eighth edition, 1969.

Cho72 Chomsky, N., Acquisition of language. In *Chomsky: Selected Readings*, Oxford University Press, third edition, 1972.

CLR92 Cormen, T. H., Leiserson, C. E. and Rivest, R. L., *Introduction to algorithms.* MIT Press, sixth edition, 1992.

CT94 Cavnar, W. B. and Trenkle, J. M., N-Gram-based text categorization. *Proceedings of Third Annual Symposium on Document Analysis and Information Retrieval*, 1994, pages 61–175.

Cut97 Cutting, D., Lucene, `http://lucene.apache.org/`, 1997.

CYWM04 Cai, D., Yu, S., Wen, J. and Ma, W., Block-based web search. *Proc. of the 27th annual international ACM SIGIR conference*, New York, NY, USA, 2004, ACM Press, pages 456–463.

FG04 Fuhr, N. and Grobjohann, K., XIRQL: An XML query language based on information retrieval concepts. *ACM Transactions on Information Systems*, 22,2(2004), pages 313–356.

Gar00 Gardenfors, P., *Conceptual Spaces: The Geometry of Thought.* MIT Press, 2000.

Goo07 Google, `http://www.google.com/`, 2007.

Har60 Harris, Z., *Methods in Structural Linguistics, 4th edition.* University of Chicago Press, 1960.

Hof99 Hofmann, T., Probabilistic latent semantic indexing. *Research and Development in Information Retrieval*, 1999, pages 50–57.

HPK95 Honkela, T., Pulkki, V. and Kohonen, T., Contextual relations of words in Grimm tales analyzed by self-organizing map. *Proc of ICANN-95, International Conference on Artificial Neural Networks*, Paris, 1995, pages 3–7.

Ht:95 Ht://dig, `http://www.htdig.org/`, 1995.

HT02 Honkela, J. and Tuulos, V., GS textplorer – adaptive framework for information retrieval. *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference*, New York, NY, USA, 2002, ACM Press, pages 456–456.

KHLK98 Kaski, S., Honkela, T., Lagus, K. and Kohonen, T., WEBSOM – self-organizing maps of document collections. *Neurocomputing*, 21, pages 101–117.

KJ98 Kekäläinen, J. and Järvelin, K., The impact of query structure and query expansion on retrieval performance. *Proceedings of the 21st annual international ACM SIGIR conference*, New York, NY, USA, 1998, ACM Press, pages 130–137.

Koh01 Kohonen, T., *Self-Organizing Maps*. Springer-Verlag, third edition, 2001.

Kor85 Korfhage, R., Theoretical measures in P/Q document spaces. *Proceedings of the annual international ACM SIGIR conference*, 1985, pages 33–40.

Lac96 Lacey, A., *A Dictionary of Philosophy*. Routledge, 1996.

Leh06 Lehtonen, M., *Indexing Heterogeneous XML for Full-Text Search*. Ph.D. thesis, University of Helsinki, 2006.

Lev02 Levon, J., OProfile: System-wide profiler for linux, `http://oprofile.sf.net/`, 2002.

LP77 Lacroix, M. and Pirotte, A., Domain-oriented relational languages. *Proceedings of the Third International Conference on Very Large Data Bases*. IEEE Computer Society, 1977, pages 370–378.

LYJ05 Li, Y., Yang, . and Jagadish, H. V., NaLIX: an interactive natural language interface for querying XML. *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2005, ACM Press, pages 900–902.

LYRL04 Lewis, D. D., Yang, Y., Rose, T. and Li, F., RCV1: A new benchmark collection for text categorization research. In *Journal of Machine Learning Research*, volume 5, MIT Press, 2004, pages 361–397.

Mac03 MacKay, D., *Information Theory, Inference and Learning Algorithms.* Cambridge University Press, 2003.

MC04 Metzler, D. and Croft, W. B., Combining the language model and inference network approaches to retrieval. *Information Processing Management*, 40,5(2004), pages 735–750.

MSN07 MSN Live, `http://www.msn.com/`, 2007.

OAP+06 Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C. and Lioma, C., Terrier: A High Performance and Scalable Information Retrieval Platform. *Proceedings of ACM SIGIR'06 Workshop on Open Source Information Retrieval (OSIR 2006)*, 2006.

PB98 Page, L. and Brin, S., The anatomy of a large-scale hypertextual web search engine. *Proc. Seventh World Wide Web Conference*, Brisbane, Australia, 1998.

PBC+02 Pinto, D., Branstein, M., Coleman, R., Croft, W. B., King, M., Li, W. and Wei, X., QuASM: a system for question answering using semi-structured data. *JCDL '02: Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, New York, NY, USA, 2002, ACM Press, pages 46–55.

PBMW98 Page, L., Brin, S., Motwani, R. and Winograd, T., The pagerank citation ranking: Bringing order to the web. Technical Report, Stanford Digital Library Technologies Project, 1998.

PC98 Ponte, J. M. and Croft, W. B., A language modeling approach to information retrieval. *Research and Development in Information Retrieval*, 1998, pages 275–281.

PEK03 Popescu, A., Etzioni, O. and Kautz, H., Towards a theory of natural language interfaces to databases. *Proceedings of the 8th international conference on Intelligent user interfaces*, New York, NY, USA, 2003, ACM Press, pages 149–157.

Por01 Porter, M., Snowball stemmer, `http://snowball.tartarus.org/`, 2001.

PTBT05 Perkiö, J., Tuulos, V., Buntine, W. and Tirri, H., Multi-faceted information retrieval system for large scale email archives. *Proceedings of the IEEE/WIC/ACM Conference on Web Intelligence (WI 2005)*, 2005, pages 557–564.

RG02 Ramakrishnan, R. and Gehrke, J., *Database Management Systems*. McGraw-Hill, third edition, 2002.

RL04 Rose, D. E. and Levinson, D., Understanding user goals in web search. *WWW '04: Proc. of the 13th international conference on World Wide Web*, New York, NY, USA, 2004, ACM Press, pages 13–19.

TPT05 Tuulos, V., Perkiö, J. and Tirri, H., Multi-faceted information retrieval system for large scale email archives (description of a demonstration). *Proceedings of the 28th annual international ACM SIGIR conference*, New York, NY, USA, 2005, ACM Press, pages 683–683.

TS05 Tuulos, V. and Silander, T., Language pragmatics, contexts and a search engine. *International and Interdisciplinary Conference on Adaptive Knowledge Representation and Reasoning*, 2005.

TT04 Tuulos, V. and Tirri, H., Combining topic models and social networks for chat data mining. *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)*. IEEE Computer Society, 2004, pages 206–213.

Tuo05 Tuominen, A., HooWWWer homepage, `http://cosco.hiit.fi/search/hoowwwer/`, 2005.

UK05 Uzuner, O. and Katz, B., A comparative study of language models for book and author recognition. *Proceedings of the Second International Conference on Natural Language Processing - IJCNLP 2005*, volume 3651 of *Lecture Notes in Computer Science*. Springer, 2005.

VB06 Voorhees, E. M. and Buckland, L. P., editors, *The Fifteenth Text REtrieval Conference Proceedings (TREC 2006)*. NIST, 2006.

Wik07 Wikipedia, `http://en.wikipedia.org/`, 2007.

WMB99 Witten, I. H., Moffat, A. and Bell, T. C., *Managing Gigabytes - Compressing and Indexing Documents and Images*. Morgan Kauffman, second edition, 1999.

XQu07 XQuery 1.0: An XML query language, `http://www.w3.org/TR/xquery/`, 2007.

Yah07 Yahoo, `http://www.yahoo.com/`, 2007.