NTNU
Innovation and Creativity

A generic and flexible Framework for focusing Search at Yahoo! Shopping

Trond Øivind Eriksen
Anne Siri Korsen

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Kjetil Nørvåg, IDI
Co-supervisor: Per Gunnar Auran, Yahoo! Technologies Norway AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Yahoo! Technologies Norway AS is developing a vertical search platform, Vespa, being used for more than 40 of Yahoo!'s vertical search services. This thesis aims to improve the users' perceived relevance when searching on Yahoo! Shopping. The proposed ideas should be implemented as a Searcher plug-in in the Query-Result-Server (QRS) in Vespa. The QRS has a chain of Java modules that modifies the query before it is sent to the search engine's back-end, and the result set before it is sent to the front-end of the application.

The work includes a theoretic and a practical part:

Theoretic part
- General overview of ranking and relevance in search.
- Adjustment towards ranking of structured data and use of knowledge from database environments.

Practical part
- Create a Searcher plug-in that analyses results from back-end and ranks the results in a way that improves the user's perceived relevance.
- Document how the improved relevance is achieved.
- Describe if the implemented solution will scale to the most visited Yahoo! verticals.

Assignment given: 20. January 2006
Supervisor: Kjetil Nørvåg, IDI

Abstract

Information retrieval is concerned with extraction of documents from a collection, according to the user's information need. The ranking returned by a search engine is determined by the relevance function in use. The amount of information stored digitally and being searched for on the Web, grows every day. As the document bases grow, relevance has never been more important.

There is a trend towards domain-specific search solutions, vertical search services, in the case of searching the Web. A vertical search service utilise semi-structured documents, i.e. documents which contain metadata describing the content. Semi-structured information retrieval is a hybrid between traditional information retrieval based on unstructured documents, and database retrieval based on structured content. Semi-structured documents imply the use of multiple criteria for how the returned documents should be ranked. This in turn arises questions like which criterion that is more important, and how to combine the results produced by the different criteria. This thesis addresses these challenges. We have studied relevance techniques for the purpose of identifying an approach to improve the perceived relevance at the Yahoo! vertical search platform, Vespa. In particular, Yahoo! Shopping has been the focus during problem elaboration, implementation, and evalution.

A plug-in is implemented in Vespa, providing a generic and flexible framework for hybrid search. Our solution allows for context queries, i.e. queries that include terms that describe the desired context, with no specific knowledge about the query language or document structure needed. Also, keyword and context terms in a query is treated differently, using the context terms only for focusing the search.

5 experiments have been performed to test our proposed solution. The results indicate that:

- A considerable improvement in retrieval performance is achieved for context queries. Much of the improvement is obtained by removing noisy hits from the result.
- The solution performs almost similar as the standard approach for non-context queries. However, these queries will suffer from a higher latency. The latency depends on the complexity of the domain.

Most search engines today either return thousands of answers to a user query, or, in $\sim 20\%$ of the cases, none. Our solution may provide as a solution to these challenges and thus help to improve the perceived relevance. It should be noted that the solution requires a reasonable labelling of the documents, in addition to training of the users in order to make them use context words in their queries.

The preliminary experiment results are positive, but are influenced by a reference collection somewhat adapted to our solution, and should therefore be complemented with experiments based on a full system implementation and a well-defined reference collection. The first step is to choose an appropriate labelling scheme for how the semantics of the documents and queries should be captured. Next, it would be interesting to experiment with the ranking of the results. Finally, the user interface should be extended in order to guide the user when submitting context queries.

Preface

This thesis is written by Trond Øivind Eriksen and Anne Siri Korsen as part of our master degrees at the *Department of Computer and Information Science (IDI)*, at the *Norwegian University of Science and Technology (NTNU)* in Trondheim.

The intention of this thesis is to improve the users' perceived relevance when searching on Yahoo! Shopping. First, a survey of ranking and relevance in search will be conducted. Next, the task aims to implement a flexible component, realised as a Java plug-in, that analyses results from back-end and ranks the results in a way that improves the user's perceived relevance.

During our work, we had the privilege to be supervised by Kjetil Nørvåg (Associate Professor) and Per Gunnar Auran (Senior Research Scientist, Yahoo!). Without their valuable guidance and feedback, this research and thesis would not have had the quality we are proud of presenting today.

Finally, we would like to thank Yngve T. Aasheim and Sigurd Gartmann, together with the other Yahoo! employees at the Trondheim office, for their continuous support and valuable feedback. Also, we are most grateful for the tasteful free lunch.

Trondheim, June 2006

Anne Siri Korsen Trond Øivind Eriksen

Contents

List of Figures

List of Tables

Listings

Part I

Background

Chapter 1
Introduction

"Attempt the end, and never stand to doubt; nothing's so hard but search will find it out."

– Robert Herrick

This chapter gives an introduction to the thesis, and describes the motivation, task definition and goals, thesis scope, and the thesis outline.

1.1 Motivation

Vast amounts of information are stored digitally and being searched for on the Web every day. According to [48], which focuses on quantifying the influence of Web search in people's daily Web access, 13.6% of all Web traffic is under the direct or indirect influence of search engines. Search engines also help users reach 20% more sites by presenting them in search results, which may be otherwise unreachable by the users.

Information retrieval is concerned with extraction of documents from a collection, according to the user's information need. The ranking returned by a search engine is determined by the relevance function in use. These functions define a set of features of the documents to assist the decision process. The features may be query dependent or query independent. Examples of features are freshness of the document, authority, i.e. a measure of a page's importance and popularity deduced from the link structure on the Web, and the weighted number of times the query terms occur in the document.

In the case of generic web search engines, a heterogeneous document collection, and diverse users, pose challenges in deciding which documents that are more relevant. Consider the query `brown`. Is the user more interested in the colour, the university, or a person with the surname *Brown*?

There is a trend towards domain-specific search solutions, vertical search services, in the case of searching the Web. Examples of vertical search engines at http://www.yahoo.com are *Images*, *Video*, *Audio*, *Directory*, *Local*, *News*, and *Shopping*, as can be seen in Figure 1.1. By focusing on one area of knowledge, with clear relationships between concepts and a limited document base, domain-specific search solutions may provide far superior results to generic search engines [5].

Figure 1.1: The Vertical Search Engines at http://www.yahoo.com

A vertical search service relies on semi-structured documents. Semi-structured documents capture the semantics as labelled fields usually represented as XML[1]. That is, we are moving from a pure information retrieval view towards a database centric view, where the structure of the data is known.

Traditional information retrieval techniques are based on a static document concept, that ignore the document structure and do not support structured queries. This impose serious restrictions if used for XML retrieval. Techniques originally used for database retrieval are neither appropriate, since answers are either right or wrong, i.e. there is no concept of partially relevant answers. The queries submitted are also very specific, requiring extensive knowledge of the structure of the data to be searched in. Finally, semi-structured documents imply the use of multiple criteria for how the returned documents should be ranked. This in turn arises questions like which criterion that is more important, and how to combine the results produced by the different criteria. This thesis addresses challenges related to semi-structured information retrieval.

1.2 Thesis Definition and Goals

Yahoo! Technologies Norway AS is developing a vertical search platform, *Vespa*, being used for more than 40 of Yahoo!'s vertical search services. This thesis aims to improve the users' perceived relevance when searching on Yahoo! Shopping. The proposed ideas should be implemented as a *Searcher plug-in* in the *Query-Result-Server (QRS)* in Vespa. The QRS has a chain of Java[2] modules that modifies the query before it is sent to the search engine's back-end, and the result set before it is sent to the front-end of the application.

The work includes a theoretic and a practical part:

Theoretic part

- General overview of ranking and relevance in search.
- Adjustment towards ranking of structured data and use of knowledge from database environments.

Practical part

- Create a Searcher plug-in that analyses results from back-end and ranks the results in a way that improves the user's perceived relevance.
- Document how the improved relevance is achieved.
- Describe if the implemented solution will scale to the most visited Yahoo! verticals.

1.3 Thesis Scope

This thesis is limited to studying search rankings for the Shopping vertical. Also, the main focus is on the domain of books, music, and videos. This is due to several factors. First, only considering a sub-domain made it easier to delve into the problem to be addressed. Further, a confined time perspective and restrictions in Vespa causing a less general implementation, forced restrictions of the task at hand. The domain of books, music, and videos were chosen due to a better structure of these documents than for other domains. To describe how the implemented solution can be

[1] XML (Extensible Markup Language) is a W3C initiative that allows information and services to be encoded with meaningful structure and semantics that computers and humans can understand, http://www.w3.org/XML/.

[2] Java is an object-oriented programming language created by Sun Microsystems, http://java.sun.com/

generalised to other domains, and if the proposed ideas will scale to the most visited Yahoo! verticals, is however considered interesting for the thesis.

Two systems is used for implementation and testing, and evaluation, respectively. Due to factors described in Chapter 10, it was not possible to obtain a perfect mirror of the real Shopping results. However, the system used for evaluation was as close to the real world as possible.

Finally, the evaluation will only consider raw relevance, i.e. the difference in ranking values with and without our solution. While user interface issues also contribute a great deal to perceived relevance, we will only treat this subject from a theoretic point of view.

1.4 Thesis Outline

This section briefly describes the outline of the thesis. The thesis has been divided into four different parts, as presented below.

Part I Background

- **Chapter 1** contains this introduction.
- **Chapter 2** presents a motivation of the thesis, elaborating relevant topics and initial thoughts about the challenges to be addressed.
- **Chapter 3** describes the research method used in this thesis.

Part II State-of-the-art

- **Chapter 4** introduces the most central concepts and techniques in the field of information retrieval.
- **Chapter 5** elaborates on concepts and techniques related to vertical searching, that is specific domain search tools.
- **Chapter 6** provides a short introduction to Yahoo's vertical search platform, Vespa.

Part III Implementation and Experiments

- **Chapter 7** discusses some of the preliminary ideas regarding the Searcher plug-in.
- **Chapter 8** describes the implemented Searcher plug-in. The motivation for how the improved relevance is achieved, and the value our solution adds beyond existing functionality in Vespa, is discussed. Finally, scaling to more visited Yahoo verticals is explored.
- **Chapter 9** presents the evaluation principles that are used in the experiments.
- **Chapter 10** presents the various experiments performed in this thesis. A discussion of the results is also provided.

Part IV Conclusion and Further Work

- **Chapter 11** presents a summary of the findings of this research, and concludes the work.
- **Chapter 12** outlines propositions for further work relevant to this thesis.

Chapter 2
Problem Elaboration

"The only good is knowledge and the only evil is ignorance."

–Socrates

This chapter elaborates on topics related to the thesis. First, a short introduction to the concepts of information retrieval and vertical searching is given. Secondly, an example of a search on Yahoo! Shopping is presented. Finally, some initial thoughts related to the problem to be addressed are provided.

2.1 Information Retrieval vs. Vertical Searching

Information retrieval (IR) is a sub-field of computer science that deals with the automated storage and retrieval of documents. While some users have a clearly defined objective with the searching activity, others have a diffuse idea of what they are looking for. The concept of search is generally used to cover both *recovery* of documents that we know exist, and *discovery* of information that we intuit exist [5]. Information retrieval is further discussed in Chapter 4.

A *vertical search* application is a domain-specific search system to serve the information needs of specific domains. Common for these services is that the documents to be searched are more structured than ordinary web documents, i.e. they include labelled fields usually represented as XML. Vertical searching is discussed more closely in Chapter 5.

In Listing 2.1 and 2.2, examples of an unstructured document and a semi-structured document are shown. The documents illustrate two different ways of presenting the information about the book and the video "The Da Vinci Code". The first example presents the information in a standard manner, without any specific form of structure. In the second example, the same document is represented using XML labels. An advantage of having semi-structured documents is that it makes it possible to utilise the information about the semantics, and thereby making the search more precise. In the second example, it is possible to search within specific labels of the document. This contributes to improve the retrieval precision (precision is explained in Chapter 4).

```
1 "The Da Vinci Code", a book by Dan Brown (2003)
2 Paperback
3 List price: $17.95
4 Note: A murder inside the Louvre and clues in Da Vinci paintings lead to the discovery of
5 a religious mystery protected by a secret society for two thousand years; which could
6 shake the foundations of Christianity.
```

Listing 2.1: An Unstructured Document

```
<document type="book">
    <title>The Da Vinci Code</title>
    <author>Dan Brown</author>
    <format>Paperback</format>
    <price>17.95</price>
    <year>2003</year>
    <desc>A murder inside the Louvre and clues in Da Vinci paintings lead to the discovery
    of a religious mystery protected by a secret society for two thousand years; which
    could shake the foundations of Christianity.</desc>
</document>
```

Listing 2.2: A Semi-Structured Document

2.2 Searching on Yahoo! Shopping

In this section, an example of a search performed on Yahoo! Shopping is presented. Figure 2.1 shows the Shopping search front-end at http://shopping.yahoo.com. The front-end is divided into three main panes; the *Query Pane* at the top, the *Refine Results Pane* to the left, and the *Results Pane* in the middle. In what follows, we will describe the main functionalities in each pane.

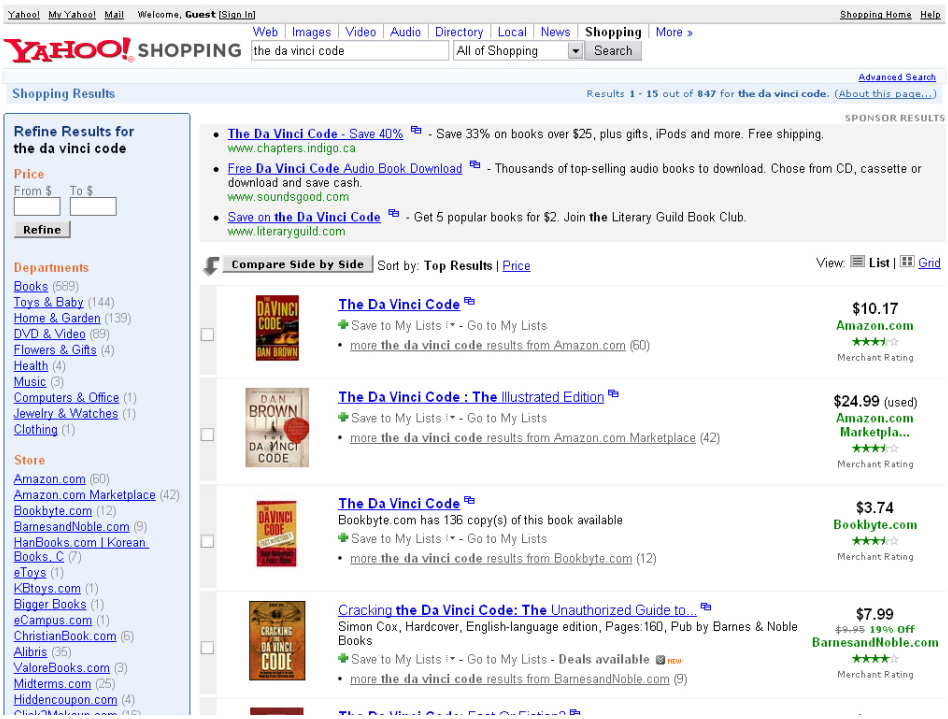

Figure 2.1: A Search for `the da vinci code` on Yahoo! Shopping

Query pane

- **The query field** – the text field where the user defines the query.
- **Department drop-down box** – by selecting one of the values in the department drop-down box, the user can search for products within the following categories: *Bargains*, *Beauty*, *Books*, *Clothing*, *Computers & Office*, *DVD & Video*, *Electronics*, *Flowers & Gifts*, *Jewelry & Watches*, *Music*, *Sports & Outdoors*, and *Toys & Baby*.

Refine results pane

- **Price** – by defining values for *Price from* and *Price to*, the user can narrow the search to only retrieve items that have a price that satisfies these values.
- **Departments** – by clicking on one of the departments (categories), the search is narrowed to only search for products within the respective department.
- **Store** – by clicking on one of the stores, the search is narrowed to only search for products within the respective store. Examples of stores are *Amazon* and *Wal-Mart*.
- **Brand** – by clicking on one of the brands, the search is narrowed to only search for products of a specific brand. Examples of brands are *Canon*, *Sony*, and *Dell*.
- **Group Products** – by clicking on one of the two *grouping* options, the retrieved products are either grouped by *all products* or *one product per store*.

Results pane

- **Sort by** – by clicking on one of the two *sort by* options, the retrieved products are either sorted by top results, i.e. relevance (default) or price.
- **View** – by clicking on one of the two *views*, the retrieved products are either viewed as a *List* (default) or as a *Grid*.
- **Compare side by side** – two or more products can be compared side by side by first checking the checkbox left to the image of each retrieved product, and then clicking on the compare-button.

2.3 Known Challenges

This section presents some initial thoughts regarding the problem to be addressed in the thesis. The challenges that will be discussed in the following are *treatment of context queries*, *focusing search*, and *utilising semantic relations*.

2.3.1 Treatment of Context Queries

One of the main challenges when searching on Yahoo! Shopping today is the treatment of *context queries*. We define a context query as a query that includes two or more terms, where at least one of the terms describes a certain context to focus the search within. Such a context can for example be *DVD*, *director*, or *artist*.

Consider the query `brad pitt movies`. Searching for this query on Yahoo! Shopping retrieves results that users may consider less relevant. Instead of retrieving movies with Brad Pitt, the first page of results is dominated by posters, t-shirts, and other accessories. This is because the query is treated as an AND-query, thus requiring that all query terms are found in the documents. However, the users are not necessarily looking for every document that contains all the query terms, rather documents that conceptually are movies of the Hollywood actor. Making a search engine understand this, may greatly improve retrieval. To cope with this challenge, [5] proposes to use cue words that tip the engine off to the context of a particular search. In this case, movies is a concept and not just a word found in one of the indexed documents. The cue words are next linked to clusters of results that might fulfil the concept of movie. These documents are found using metadata, i.e. labels describing the semantics.

2.3.2 Focusing Search

When users search on Yahoo! Shopping today, a lot of noisy hits, i.e. hits that are considered irrelevant to the query, are retrieved for many queries. If the top-k results contain several noisy hits, this may contribute to ruining the users' searching experience. The challenge of removing noisy hits can be overcome by various approaches. One approach is to find the most relevant category/categories to a specified query, and focusing the search towards this category.

Figure 2.2 visualises the effect of removing noisy hits. Here $\mathcal{A}$ is the result set before the search has been narrowed, and $\mathcal{A}'$ is after.

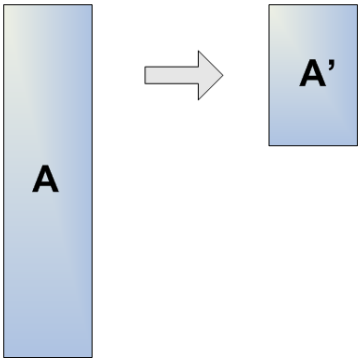

Figure 2.2: Narrowing the Result Set

Focusing search and thereby narrowing the result set, may have a large impact on perceived relevance when sorting by price. Today, when performing the search `canon digital camera` and sorting by price on Yahoo! Shopping, additional equipment to cameras like AC adaptors and camera cases, are highest ranked. The reason is obvious; too much noisy hits in the result set with low price. If the search could be focused, only retrieving digital cameras from Canon and ignoring accessory products, the price sort would be performed only on the products that actually are relevant given the query, yielding considerably improved perceived relevance.

2.3.3 Utilising Semantics

Semantics within and between XML documents should be exploited in order to improve perceived relevance. Within a single XML document, this means to decide which features or XML elements are most interesting. Between documents, the task is to determine semantic relations between documents or parts of documents.

As to exploiting semantics within XML documents, this can be done at query-time. For example, a user browsing for CDs by Madonna may use the search term `madonna`. When performing this search, the result set may include music by Madonna, books about her, or even videos. It is quite unlikely that the user is looking for all three. If the user in some way can identify whether she only is interested in books, music or videos with Madonna, the search would be considerably more precise. How context can be specified in the query, will be further discussed in Chapter 5.

Another approach of semantic exploitation is to boost hits where the query terms occur in the more specific fields. The motivation is that information in more specific fields is considered more descriptive of the documents. For the domain *books*, *music*, and *videos*, examples of specific fields are `author`, `artist`, `actor` and `director`. A less specific field is `desc` (see Listing 2.2). For the query `madonna`, this strategy will favour hits where *Madonna* occurs in the `artist` field. Note

that the difference from the approach mentioned in the previous paragraph, is that the user does not have to specify the desired context.

As to semantic relations between XML documents, it could be interesting to explore relations like:

- book ↔ film ↔ soundtrack.
- album → other albums of the same artist.
- film → other films of the same director or with the same actors.
- book → other books of the same author.

As an example, consider the query `pride and prejudice`. The result set contains various TV-series, films, soundtracks, and the book of Jane Austen. These hits should be connected in order to show their semantic relation. Often, only a subset of the first relation will be available, i.e. the film has no soundtrack, or is not based on a book. The idea behind the three last relations, is that a user that has shown interest in one item may also be interested in related items, i.e. items with the same participants.

Chapter 3
Research Method

"As an experimental scientist, it is important to stay intimately connected with research methods, data acquisition, data analysis and modeling."

– Norbert Scherer

The main objectives of this thesis are to obtain a general overview of ranking and relevance in vertical search, and implement a flexible component that aims to improve users' perceived relevance when searching on Yahoo! Shopping. In the following, we summarise the research methods used in this thesis.

Research in software engineering can be based on several different methods, usually answering empirical questions through controlled experiments. In [62], there are four general categories of research methods: *scientific method*, *engineering method*, *empirical method*, and *analytical method*. The scientific method builds a method based on observations. In the engineering method, current solutions are studied and changed appropriately. The empirical method proposes and evaluates a model through empirical studies. In the analytical method, a formal theory is developed, and results derived from that theory is compared with empirical observations. This thesis will use a hybrid approach between the engineering method and the analytical method. That is, the current solution, i.e. Vespa, is examined first, then a formal model based on state-of-the-art research is proposed and evaluated through empirical observations.

In Figure 3.1, the different phases of the thesis are visualised. These will be discussed in the following.

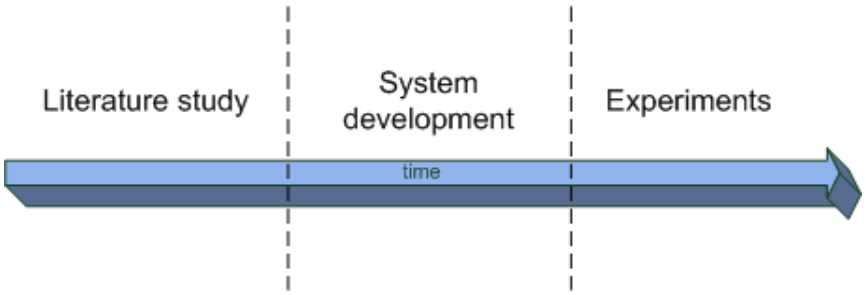

Figure 3.1: The Phases of the Thesis

Literature study As a general rule of thumb for selection of literature to base our work on, we decided to use articles from conferences and journals included in the DBLP[1] (Digital Bibliography & Library Project) server. The DBLP server provides bibliographic information on major

[1] http://www.informatik.uni-trier.de/%7Eley/db/index.html

computer science journals and proceedings. Examples of relevant and popular conferences that can be found in the DBLP server is SIGIR (Special Interest Group on Information Retrieval) [53], SIGMOD (Special Interest Group on Management of Data) [54], and VLDB (Very Large Data Bases) [17]. In addition, most referred publications and publication date were used as selection criteria. The most referred publications have been chosen in order to base the thesis on work that are acknowledged and accepted among other researchers. Papers, articles, books, and web pages with recent publication dates have been preferred to ensure that the research is up to date.

Finally, since we have no a priori knowledge of Vespa, a survey of this also needs to be carried out.

System development In system development, different models can be used based on the nature of the system that is to be implemented. Three popular methods are *the waterfall model* [51], *iterative development* [40], and *agile methods* [1]. The methods are often classified according to their adaptability or predictability. While adaptive methods focus on adapting quickly to changing realities, predictive methods, in contrast, focus on planning the future in detail. Agile methods are the most adaptive methods, the waterfall model is the most predictive, and iterative development is a mix of adaptive and predictive.

According to Boehm and Turner [8], agile methods should be used when there is low criticality, high requirements change, small number of developers, and culture that thrives on chaos. Since this research share many of the mentioned features, we will use agile methods in the development process. Agile methods attempt to minimise risk, by having short iterations in the development process. These iterations typically last one to four weeks. According to [7], agile methods also emphasize direct person-to-person communication rather than the heavy written documentation of the waterfall life-cycle.

Experiments A fundamental question concerning the results of an experiment is the validity of the results. Possible threats may be classified as threats to the conclusion, internal, construct, and external validity [62]. We will present the possible threats here. The validity of our results is discussed in Chapter 9 .

The conclusion validity deals with the correctness of the inferences drawn from the observations. Examples of threats are low statistical power, fishing, i.e. searching for a specific result, and heterogeneity of subjects.

The internal validity is concerned with the relationship between the treatment and the outcome, i.e. that the treatment causes the outcome.

The construct validity tries to discover if the experiment settings are able to reflect what is actually evaluated. These threats relate to the design of the experiment.

The external validity is concerned with generalisation outside the scope of the study. The threats to external validity is reduced by making the experimental environment as realistic as possible. Examples of threats are homogeneity of subjects, and experiment setup.

Part II

State-of-the-art

Chapter 4
General View of Search

"I don't search, I find."

– Pablo Picasso

The field of *information retrieval (IR)* is concerned with finding relevant documents in a large collection of documents. The concept of relevance is related to a user information need, and is consequently a subjective matter.

IR is not a new field of study. For approximately 4000 years, humans have organised information for later retrieval. Many of the techniques presented in this chapter are based upon research done in connection with IR in library systems. Lately, the field has got a lot of attention with the development of the World Wide Web. IR is a broad interdisciplinary field that draws on many other disciplines, such as cognitive psychology, information design, information architecture, human information behaviour, linguistics, semiotics, information science, computer science and librarianship. As a consequence of its broad nature, information retrieval is a domain of high complexity.

This chapter introduces the most central concepts in the field of IR. The chapter is mostly based upon [4].

4.1 Information Retrieval Overview

Most Web search engines today use the crawler-indexer architecture, shown in Figure 4.1. The crawler-indexer model separates the tasks of searching and indexing. The user front-end consists of the user interface and the query engine, and deals with searching. The Web front-end consists of the crawler and the document engine, and is concerned with indexing of the documents.

In order to make documents searchable, they need to be fetched into the system. How documents are fetched will be discussed closer in Section 4.2.

Text operations are performed in order to prepare the data for further analysis. Such operations can include lexical analysis, removal of stopwords and stemming. Text operations will be further elaborated in section 4.3.

The user specifies his needs for information through the user interface. The requirements are usually formulated as a set of words describing the desired information. Additionally, most search interfaces offer advanced search possibilities like proximity, phrase, and boolean search. Often, the retrieval system performs some operations at the query before searching the document base. An operation may be removal of unnecessary words, or addition of words to further specify the query. Query operations are explained in section 4.4.

The next step is to search the document base for matches to the query. The documents are usually represented by index terms. An index term can be any word appearing in the text. In that case, a full text logical view of the documents is employed. For efficiency reasons, it is more common to

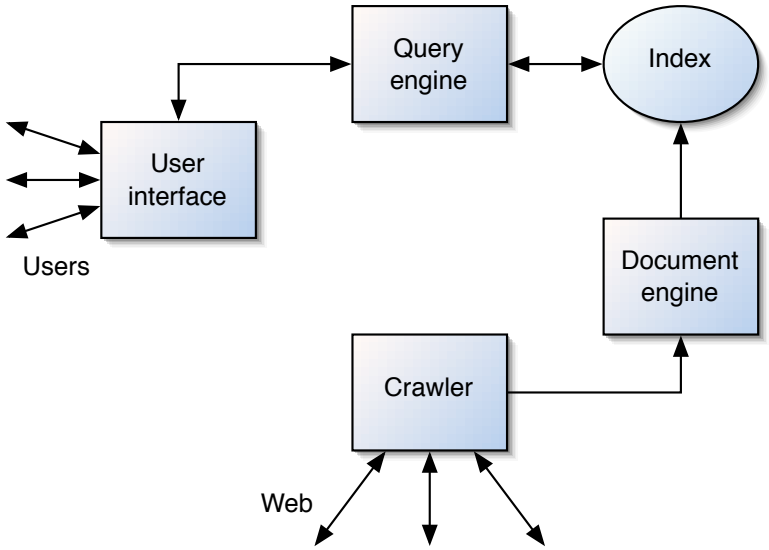

Figure 4.1: Crawler-Indexer Architecture [4]

use keywords which have some meaning of its own, or that have an important semantic meaning, as index terms. Once the logical view of the documents is defined, an index of the text is built. There are many index structures that can be used when searching over large volumes of data, the most popular structure is the inverted index. Indexing is described in Section 4.5.

The result set of relevant documents is made out of the documents that best match the query. Matches are ranked according to a ranking function, i.e. rules of relevance, specified in advance. Different ranking functions are developed which use different criteria to decide if there is a match, a partial match or not a match. In the consensus search, a single ranking function is utilised for all queries. The routing task uses different ranking functions for each query. Ranking is further investigated in Section 4.6.

Even though the information retrieval process can be a goal-oriented activity from the arise of a need to the need is satisfied, the variations are many. The search process depends on many factors; personal characteristics like knowledge and skills, the contents of the search task, the search system, the information domain to be searched within, and the results that are achieved.

4.2 Document Fetching

Documents may be fetched into the system in various ways. In the case of Web search, the crawler traverses the Web, sending new or updated pages to the search engine where they are indexed. The general technique is to start with a set of URLs, follow links found on pages in a breadth-first or depth-first manner, and fetch copies to forward to the search engine. The set of URLs initially provided, may affect the overall content of the document base, and should thus be given careful thought. Other common sources for fetching of documents are databases and file systems, e.g. for enterprise and desktop search.

4.3 Text Operations

Not all words are equally significant for representing the semantics of a text. In the general case, nouns are the single group of words which are most representative of the content of a text. Therefore, the queries and documents are often pre-processed to determine the terms to be used as query terms or index terms. This section covers three pre-processing operations; lexical analysis, elimination of stopwords, and stemming and lemmatisation. Note that for query pre-processing, operations like proper name and phrase recognition, and spell checking are also often performed.

4.3.1 Lexical Analysis

A *lexical analysis* is the process of converting a stream of characters (the text of documents) into a stream of words (the candidate words to be adopted as index terms). This is called *tokenization*, and includes among other factors the treatment of numbers, hyphens, punctuation marks, and the case of letters. In Table 4.1, some examples of challenges related to tokenization are presented. It is important that the converting process is performed in a correct manner, in order to avoid indexing the tokens wrong. For example *state-of-the-art* could be extracted and indexed as one token or four tokens (*state of the art*). If the tokens are not indexed correctly, it could cause problems at retrieval.

Table 4.1: Challenges with Tokenization

Type	**Examples**
Numbers	*123,456.78*
Case of letters	*Bush* vs. *bush*, *Bank* vs. *bank*
Punctuation marks	m.p.h, .NET
Hyphens	*state-of-the-art* vs. *state of the art*

4.3.2 Stopwords

Stopwords are words that occur so frequently in documents that they are not useful for distinguishing one document from another. Stopwords are therefore normally filtered out as index terms. The benefit of removing stopwords is that it considerably reduces the index size. The drawback is that it may reduce recall for the queries containing stopwords, for example for the query `"to be or not to be"`. Table 4.2 shows some English and Norwegian examples of stopwords.

Table 4.2: English and Norwegian Stopwords [49]

English stopwords	**Norwegian stopwords**
I, a, about, an, are, a, at, be, by, com, de, en, for, from, how, in, is, it, la, of, on, or, that, the, this, to, was, what, when, where, who, will, with, und, the, www	av, begge, bra, da, denne, der, deres, det, din, disse, du, eller, en, for, fra, ha, i, ikke, inn, kan, men, navn, nei, ny, og, om, opp, oss, part, rett, slik, som, til, under, ut, var, vi

4.3.3 Stemming and Lemmatisation

This section presents two approaches for converting words to their root forms, namely *stemming* and *lemmatisation*.

Stemming

Stemming is an algorithmic method for reducing a word to its root form, or stem, by removing typical suffixes of the word. The algorithm checks the ending of the word, and if this ending corresponds with a typical ending in the given language, the ending is removed from the word. Porter's Stemming Algorithm, is an example of a stemmer. The stemmer uses different stemming rules, which vary from one language to another. Below, two of the most used stemming rules for the English language are presented:

$$sses \longrightarrow ss$$

$$s \longrightarrow \phi$$

Table 4.3 presents some stemming examples. Stemming reduces the size of the document representation, that is, the number of distinct terms needed for representing a set of documents. The method also increases recall, as syntactic variations of words are removed from the documents.

Table 4.3: Stemming

Word	Stemmed word
Streets	Street
Cars	Car
Caresses	Caress
Attacking	Attack

A negative aspect with stemming is that there may be situations where a part of a word gets removed even if it is not a suffix as a consequence of inflection. For example, the word *news* is stemmed to *new* by using Porter's Stemming Algorithm. This is a clear weakness, but still most words will be stemmed to a unique stem for this word, and the effect of the problem is therefore reduced. One way to avoid such error situations, is to use a different method, called lemmatisation.

Lemmatisation

Lemmatisation is closely related to stemming, but instead of using an algorithmic approach, lemmatisation uses a dictionary. By using lemmatisation, lists of inflected forms are used to map words to their primary form. This method will always give the right primary form of the word as long as the word and its inflected forms are found in the list. Table 4.4 presents examples of lemmatisation.

A potential weakness with lemmatisation is that unusual words may not be in the list. This often happens when a lemmatiser for a new language is implemented.

Table 4.4: Lemmatisation

Related words	Lemmatised word
walk, walked, walks, walking	walk
am, was, are, is, were, been	be

4.4 Query Operations

Formulating a query for an information need is difficult and the users often need to reformulate the queries for effective retrieval. This suggests that the initial query should be extended and reweighed in order to focus the search towards the relevant documents. This section examines three approaches for improving the initial query formulation, namely *relevance feedback*, *local analysis*, and *global analysis*.

Relevance Feedback is a popular query reformulation strategy. The idea is to ask the user to identify the relevant documents in the original result set. Important terms from these documents are used in the new query formulation. In addition, the importance of these terms is enhanced.

A distinct approach is to create a description of the cluster of relevant documents automatically. This includes identifying terms which are related to the original query terms. Such terms may be synonyms, stemming variations, or terms which are close to the query terms in the text. The analysis can be performed on the documents retrieved by a given query, or all documents in the collection. The two approaches are named local and global analysis, respectively.

Local Analysis is based on expanding the query with terms from local clusters built from the local document set, i.e. the documents initially retrieved for a given query. Three approaches for building the local clusters are:

- co-occurrence of terms inside documents.
- distance, i.e. number of words, between co-occurring terms.
- the similarity of neighbourhoods, i.e. local clusters, between terms.

A term s_u, which belongs to a cluster associated to another term s_v, is said to be a neighbour, or a searchonym of s_u. Neighbour terms often represent keywords which can be used to extend the query formulation in an unexpected direction, rather than merely complementing it with missing synonyms.

Global Analysis is based on a similarity thesaurus which identifies term relationships based on the whole document collection. A thesaurus consists of a set of terms used as indices, and for each term, a set of related words, e.g. synonyms, or terms induced by patterns of co-occurrence within documents. Here, a term is used in the sense of a concept. A term may be individual words, groups of words, or phrases. Usually, small text windows of a fixed size are used to build the thesaurus, instead of whole documents. Terms which are closest to the whole query, not individual query terms, are selected for query expansion.

4.5 Indexing

The brute force method when searching for a query, is to scan the documents sequentially. Sequential, or online search, is appropriate when the document size is small, and it is the only choice if the document collection is highly volatile, or the index space overhead cannot be afforded. A second option is to build indices, i.e. data structures, over the documents to speed up the search.

The most common indexing technique is inverted indices. An inverted index consist of the vocabulary, which is the set of all different words in the document, and for each word, a list which stores all positions where the word appears. Figure 4.2 illustrates the concept of inverted indices.

The indices vary with respect to size and the information contained. The index scheme specifies which document elements are to become searchable fields. Several index profiles may be built for different purposes. For instance, it is common to build separate indices to support phrase search.

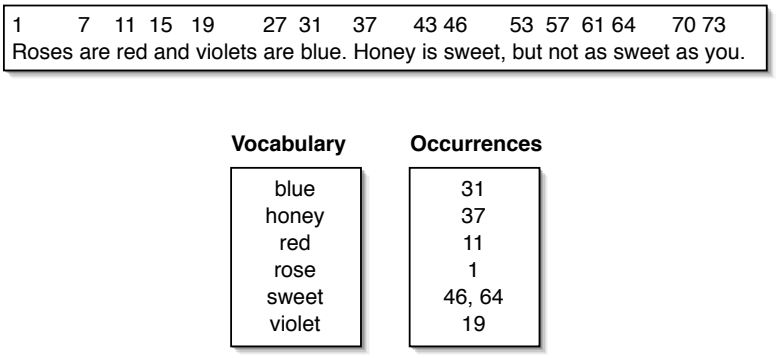

Figure 4.2: Inverted Index

The occurrences can refer to characters, words, or blocks of text. Block addressing reduces the space required by the index. The text is divided into blocks, and the occurrences point to the blocks where the words appear. This leads to smaller pointers because there are fewer blocks than positions. Also, multiple occurrences of a word inside a single block are collapsed to one reference. On the other hand, an online search over the qualifying blocks has to be performed if the exact occurrence positions are required, for instance for a proximity query.

Searching in an inverted index consists of three steps. First, the terms in the query are separated and independently searched for in the vocabulary. Second, the lists of occurrences for each term are retrieved. Third, the occurrences are manipulated to solve phrases, proximity, or boolean operations. The vocabulary is usually stored in lexicographical order, which is cheap in space and can be binary searched in $O(\log n)$ cost for a single term.

The most time demanding operation in an inverted index search, is the merging or intersection calculation of the lists of occurrences. Suffix arrays are more efficient in solving complex queries, or if the text can not be seen as a sequence of words. Its main drawbacks are a costly construction process, that the text must be available at query time, and that the results are not delivered in text position order. Unless complex queries are important, inverted files perform better for word-based applications [4].

4.6 Ranking

For searching and ranking of documents, the classic models, *the boolean model*, *the vector model* and *the probabilistic model*, are commonly used. The main focus in this thesis will be on the vector model, as this is the most used retrieval model today [43, 47]. A complete overview of a taxonomy of retrieval models is visualised in Figure 4.3.

4.6.1 The Boolean Model

The boolean model is based on set theory. It uses boolean expressions to describe queries. A document is modelled by a binary vector. The vector contains one element for each index word in the document collection. Each element is set to 1 if the word is present in the document, and 0 otherwise.

The query is made up of terms, combined using the Boolean relations AND, OR, and NOT:

$$Q = t_a \wedge (t_b \vee \neg t_c)$$

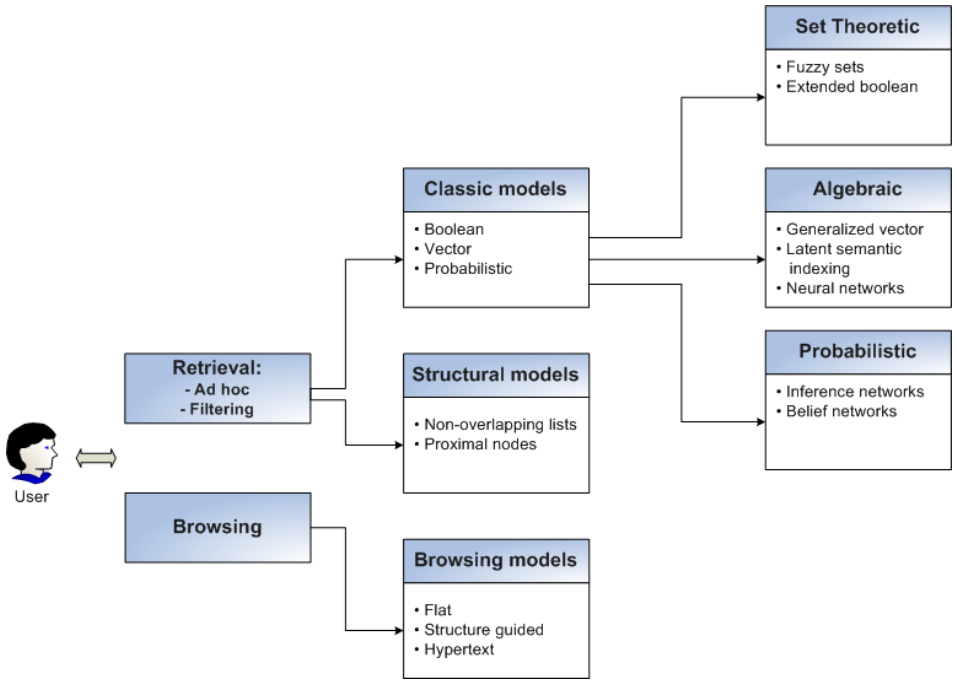

Figure 4.3: Taxonomy of Retrieval Models [4]

In order to decide if a document is relevant or not, the query is converted into its disjunctive normal form:

$$Q_{dnf} = (1,1,1) \vee (1,1,0) \vee (1,0,0)$$

If any of the conjunctive components (t_a, t_b, t_c) of the query are present in the document, the document is considered relevant to the query.

The boolean model is simple, and was popular in the early days of information retrieval. Two major drawbacks have led to less attention lately. First, the model uses binary weights of the terms, i.e. the model predicts the documents to be either relevant or non-relevant, with no notion of partial match to the query conditions. Second, it is often difficult to describe an information need as a boolean expression [4].

4.6.2 The Vector Model

The vector model is an algebraic model. Both the documents and queries are represented as vectors of weights, one weight for each index term in the document collection. More formally, for a collection with t index terms, a document D and a query Q are represented as:

$$D = (w_{d1}, w_{d2}, w_{d3}, ..., w_{dt})$$

$$Q = (w_{q1}, w_{q2}, w_{q3}, ..., w_{qt})$$

The similarity between a query and a document is usually calculated by the Cosine measure:

$$sim(d_j, q) = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t}(w_{i,j})^2 \times \sum_{j=1}^{t}(w_{i,q})^2}}$$

The most popular way to calculate the weights is to use some variation of the tf-idf weighting scheme. The *term frequency* (tf) of a term t_i for a document d_j is given as

$$tf_{i,j} = \frac{freq_{i,j}}{max_l freq_{l,j}}$$

where the maximum is computed over all terms which appear in the document d_j.

The *inverse document frequency* (idf) is defined as

$$idf_i = log\frac{N}{n_i}$$

where N is the total number of documents in the collection, and n_i is the number of documents in which term t_i appears.

Term frequency measures how well the term describes the document content, while inverse document frequency is a measure of the term's ability to distinguish the relevant documents from the non-relevant documents. The intuitive idea is that a term which occurs many times in a document, may provide as a good measure of how well the term describes the document content. On the other hand, a term which occurs in many documents is not a good discriminator in order to separate the relevant documents from the non-relevant ones. Despite its simplicity, the vector model yields good retrieval performance with general collections [4].

Two extensions of the vector model, the *generalized vector model* and *latent semantic indexing*, are presented next.

Generalized Vector Model

The generalized vector model is based on less restrictive interpretation of term-to-term independence. In particular, the index terms vectors are assumed linearly independent[1], but not pairwise orthogonal[2]. In contrast, the classic model often interprets independence of index terms as pairwise orthogonality. The generalized vector model was proposed by Wong, Ziarko and Wong [63].

Possible patterns of term co-occurence (inside documents) are represented by *minterms*. These minterms are given by

$$\begin{aligned} m_1 &= (0, 0, \ldots, 0) \\ m_2 &= (1, 0, \ldots, 0) \\ m_3 &= (0, 1, \ldots, 0) \\ m_4 &= (1, 1, \ldots, 0) \\ &\vdots \\ m_{2^t} &= (1, 1, \ldots, 1) \end{aligned}$$

The minterm m_1 points to the documents containing none of the index terms. The minterm m_2 points to the documents containing solely the index term k_1. Further, the minterm m_{2^t} points to the documents containing all the index terms. The central idea in the generalized vector space model is to introduce a set of pairwise orthogonal vectors $\vec{m}_i$ associated with the set of minterms, and to adopt this set of vectors as the basis for the subspace of interest. The $\vec{m}_i$ vectors are

[1] Linear independence: Let $S = v_1, v_2, \ldots, v_n$ be a set of vectors in the vector space V. The set S is a linearly independent set if none of the vectors $v_1, v_2, \ldots, v_n$ is expressible as a linear combination of the other vectors in S.

[2] Pairwise orthogonality means that for each pair of index term vectors $\vec{k}_i$ and $\vec{k}_j$, we have $\vec{k}_i \cdot \vec{k}_j = 0$. For example, the vectors $(1, 0, 1)$ and $(1, 0, -1)$ are pairwise orthogonal. As long as the vectors are not 0, pairwise orthogonality implies linear independence.

$$\begin{aligned}\vec{m}_1 &= (1,0,\ldots,0,0)\\ \vec{m}_2 &= (0,1,\ldots,0,0)\\ &\vdots\\ \vec{m}_{2^t} &= (0,0,\ldots,0,1)\end{aligned}$$

where each vector $\vec{m}_i$ is associated with the respective minterm m_i. Index terms are correlated by the $\vec{m}_i$ vectors. For instance, the vector $\vec{m}_4$ is associated with the minterm $m_4 = (1,1,\ldots,0)$ which points to the documents in the collection containing the index terms k_1, k_2, and no others. If such documents do exist in the collection, the minterm m_4 is *active* and a dependence between the index terms k_1 and k_2 is induced.

To determine the index term vector $\vec{k}_i$ associated with the index term k_i, the vectors for all minterms m_r in which the term k_i is in state 1 and normalise, are summed up. This gives

$$\vec{k}_i = \frac{\sum_{\forall r,\ g_i(m_r)=1} c_{i,r}\ \vec{m}_r}{\sqrt{\sum_{\forall r,\ g_i(m_r)=1}\ c_{i,r}^2}}$$

where

$$c_{i,r} = \sum_{d_j \mid g_l(\vec{d_j})=g_l(m_r)\ for\ all\ l} w_{i,j}$$

Notice that the internal product $\vec{k}_i \cdot \vec{k}_j$ can now be used to quantify a degree of correlation between the index terms k_i and k_j:

$$\vec{k}_i \cdot \vec{k}_j = \sum_{\forall r \mid g_i(m_r)=1 \wedge g_j(m_r)=1} c_{i,r}\ \times\ cj,r$$

The usage of index terms dependencies to improve retrieval performance continues to be a controversial issue. In fact, despite the introduction in the 1980s of more effective algorithms for incorporating term dependencies, there is no consensus that incorporation of term dependencies in the model yields effective improvement with general collections. Further, the generalized vector model is more complex and computationally more expensive than the classic vector model [4].

Latent Semantic Indexing

Latent semantic indexing (LSI) seeks to reduce the term-document space, and attempts to solve the synonomy[3] and polysemy[4] problems that plague automatic information retrieval systems. Where the other retrieval models are based on index term matching, LSI is based on concept matching. The advantage of concept matching is that documents could be retrieved even when they are not indexed by the specified query terms. The main idea of LSI is to map each index term vector and query vector into a lower dimensional space which is associated with concepts. The intention is that retrieval in the lower dimensional space may be superior to retrieval in the space of index terms. In [58], a simple example that describes LSI is presented. In the following this example will be presented.

[3]The semantic relation that holds between two words that can (in a given context) express the same meaning/-concept.

[4]The ambiguity of an individual word or phrase that can be used (in different contexts) to express two or more different meanings/concepts.

M	doc 1	doc 2	doc 3	doc 4	doc 5	doc 6
cosmonaut	1	0	1	0	0	0
astronaut	0	1	0	0	0	0
moon	1	1	0	0	0	0
car	1	0	0	1	1	0
truck	0	0	0	1	0	1

Figure 4.4: Term-by-Document Matrix M [58]

In Figure 4.4, an example of a term-by-document matrix M is visualised. The example shows five different terms for six different documents, and if each term is represented (denoted by 1) in the document or not (denoted by 0). If term vector similarity is used to compare the documents in the example, there is a similarity between the documents 4, 5 and 6. This is because these documents are about cars and/or trucks. However, for documents 2 and 3, the words "cosmonaut" and "astronaut" would not be recognised as the same word. This semantic weakness is avoided by using LSI.

LSI is a text application of the mathematical technique Singular Value Decomposition (SVD) [6]. SVD splits the matrix M into the product of three matrices by the least square method [6, 44]. The formula is given below, and further visualised in Figure 4.5,

$$\mathbf{M = TSD^T}$$

where T is a Term-by-m matrix, S is an m-by-m matrix, and D is an m-by-Document matrix. The m rows and columns can be thought of as the concept space.

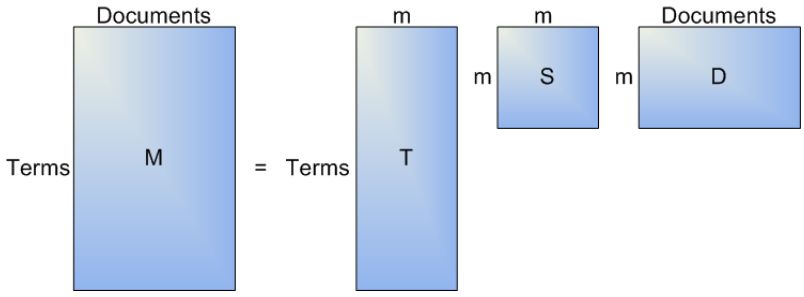

Figure 4.5: Singular Value Decomposition

The three matrices can be used as follows. Given some terms, the vectors of concepts from T can be retrieved. Further, these vectors can be used to retrieve related concepts from S. Finally, the concepts can be used to retrieve documents from D.

To visualise the documents in a two dimensional space, the document coordinates can be found by calculating the matrix:

$$\mathbf{B = S_{k\times k} D_{k\times n}}$$

where k is the top rows, and n is the number of documents.

In Figure 4.6, the matrix $B = S_{2\times 2} D_{2\times 6}$ (top), and the document correlation matrix $B^T B$ (bottom) are presented. The document correlation matrix shows how similar the documents in the collection are. The closer to 1.00 the values are, the more similar are the documents. In the example, there is a similarity between documents 2 and 3, since their $B^T B$ value 0.88 is close to 1.00. This similarity shows the strengths of LSI. Even though document 2 and 3 have no words in common, there is a high possibility that both documents are about the same topic, namely space travelling.

B	doc 1	doc 2	doc 3	doc 4	doc 5	doc 6
dim 1	-1,62	-0,60	-0,04	-0,97	-0,71	-0,26
dim 2	-0,46	-0,84	-0,30	1,00	0,35	0,65

B^TB	doc 1	doc 2	doc 3	doc 4	doc 5	doc 6
doc 1	1,00					
doc 2	0,78	1,00				
doc 3	0,40	0,88	1,00			
doc 4	0,47	-0,18	-0,62	1,00		
doc 5	0,74	0,16	-0,32	0,94	1,00	
doc 6	0,10	-0,54	-0,87	0,93	0,74	1,00

Figure 4.6: The Matrix B and the Document Correlations Matrix B^TB [58]

4.6.3 The Probabilistic Model

The probabilistic model attempts to determine the probability of a document being of interest to the user, based on the user query.

The similarity measure is defined as the ratio between the probability of a document being relevant to the query, and the probability that it is non-relevant:

$$sim(q, d_j) = \frac{P(R|d_j)}{P(\bar{R}|d_j)}$$

Using Bayes' rule, and noting that the prior probabilities for relevance and irrelevance are identical for all documents in the collection, the expression can be reformulated as:

$$sim(q, d_j) \sim \frac{P(d_j|R)}{P(d_j|\bar{R})}$$

Assuming that the presence or absence of a term is the only indicator of relevance, as well as the independence of index terms, $P(d_j|R)$ can be written as

$$P(d_j|R) = \prod_{k_i appears} P(k_i|R) \prod_{k_i does not appear} P(\bar{k_i}|R)$$

where each k_i represents an index term appearing in the query.

$P(k_i|R)$ is initially set to the same value, usually 0.5. The model assumes that the initial distribution of index terms within non-relevant documents, initially is equal to the distribution of these terms within all documents in the collection. This yields the initial estimates

$$P(k_i|R) = 0.5$$

$$P(k_i|\bar{R}) = \frac{n_i}{N}$$

where n_i is the total number of documents containing the term t_i, and N is the total number of documents.

The inital assumptions are refined each time a query is posed to the system:

$$P(k_i|R) = \frac{|r_i|}{|r|}$$

$$P(k_i|\bar{R}) = \frac{n_i - |r_i|}{N - |r|}$$

Here, $|r_i|$ is the number of retrieved documents containing the term t_i, and $|r|$ is the total number of retrieved documents.

In addition, user feedback may be used to decide which documents are relevant and non-relevant.

4.6.4 Structural IR Models

Structural IR models use information about the structure of the text, such as chapters, sections, titles, bold text, and so on, combined with content information. The structure based models differ in how they balance between model expressiveness and efficient query evaluation. This section briefly mentions two structure based models.

Non-Overlapping Lists divides the text in non-overlapping regions, collected in a list. There may be multiple ways to divide the text, for instance chapters and sections. In the case of HTML[5] documents, the text may be divided according to the various HTML tags. The text regions are split into separate lists, i.e. distinct lists for chapters and sections.

Proximal Nodes is based on a strict hierarchy composed of the document structure, in contrast to the flat lists in the previous model. The consequence is that the model allows for queries taking the relation between the structural parts into account.

4.6.5 Web Search and Pagerank

Even if each document available in the Web is fairly unstructured, the hyperlink structure between the documents can be utilised to develop new ranking schemes. Algorithms exploiting the hyperlink information show significantly increase in retrieval performance. In its most simple form, the traditional ranking models are extended to also include pages that point to a page in the answer set, and less usual, pages pointed to by a page in the answer set. Figure 4.7 illustrates this page structure.

More sophisticated algorithms use the number of links that point to a page as a measure of its popularity and quality. Also, links on a popular page should count more than links on random pages. An algorithm taking this approach is Google's[6] PageRank [4, 46]. The PageRank of a page a is defined as

$$PR(a) = q + (1 - q) \sum_{i=1}^{n} PR(p_i)/C(p_i)$$

where page a is pointed to by pages p_i to p_n. $C(a)$ is the number of outgoing links of a page a. q is the probability that the user jumps to a random page. The probability for following a random link at the current page is thus $(1 - q)$.

A drawback with PageRank is that it can easily get spammed up with sophisticated search engine marketing techniques and click fraud. To cope with this, a search engine spam solution called SNAP has been proposed [5]. SNAP is a new breed of search engine that ranks sites by, among other things, how many times they have been clicked by prior searchers. The technique is for example being used by Google today for advertising on the main page.

[5] HyperText Markup Language (HTML), is a coding language for publishing hypertext on the World Wide Web, http://www.w3.org/MarkUp/

[6] http://www.google.com

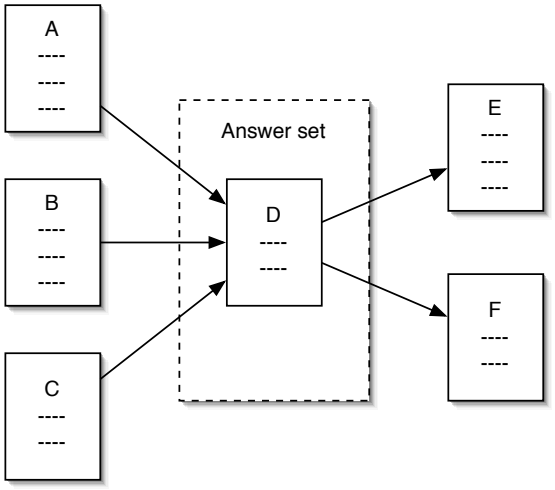

Figure 4.7: Link Structure on the Web

4.7 Retrieval Evaluation

Even though the performance of an IR system often is a highly subjective matter, it is important to evaluate the relevance of the retrieved documents. *Recall* and *precision* are the single two most popular evaluation parameters. Let $|R|$ be the number of relevant documents for a query, and $|A|$ be the number of documents in the answer set. $|R_a|$ is the number of relevant documents in the answer set. Figure 4.8 visualises the different sets.

Recall is the fraction of the relevant documents which has been retrieved:

$$Recall = \frac{|R_a|}{|R|}$$

Precision is the fraction of retrieved documents which is relevant:

$$Precision = \frac{|R_a|}{|A|}$$

As an example, consider the set R_q of relevant documents for query `q`:

$$R_q = \{d_3, d_4, d_5, d_6, d_{11}, d_{12}, d_{16}, d_{17}, d_{18}, d_{20}\}$$

Table 4.5 shows the recall and precision values for the retrieved documents. Figure 4.9 plots the highest precision value for each recall level, known as the precision versus recall figure.

To evaluate retrieval performance for more than one query, the average precision at each recall level is computed:

$$\bar{P}(r) = \sum_{i=1}^{N_q} \frac{P_i(r)}{N_q}$$

Here, $\bar{P}(r)$ is the average precision at a given recall level r, N_q is the number of queries, and $P_i(r)$ is the precision at recall level r for the i-th query.

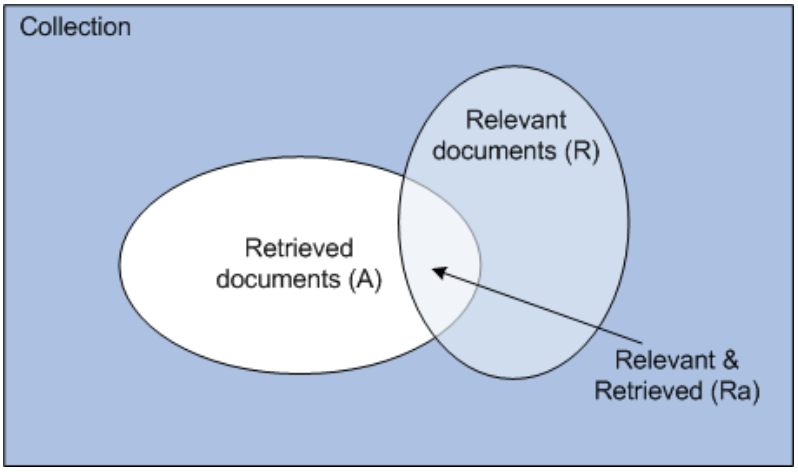

Figure 4.8: Precision and Recall

Table 4.5: Precision and Recall

Rank	Document Number	Relevant	Recall	Precision
1	d_{11}	Yes	10%	100%
2	d_{12}	Yes	20%	100%
3	d_{13}	No	20%	67%
4	d_{14}	No	20%	50%
5	d_9	No	20%	40%
6	d_{10}	No	20%	33%
7	d_7	No	20%	29%
8	d_{16}	Yes	30%	38%
9	d_8	No	30%	33%
10	d_6	Yes	40%	40%
11	d_{17}	Yes	50%	45%
12	d_4	Yes	60%	50%
13	d_{15}	No	60%	46%
14	d_{20}	Yes	70%	50%

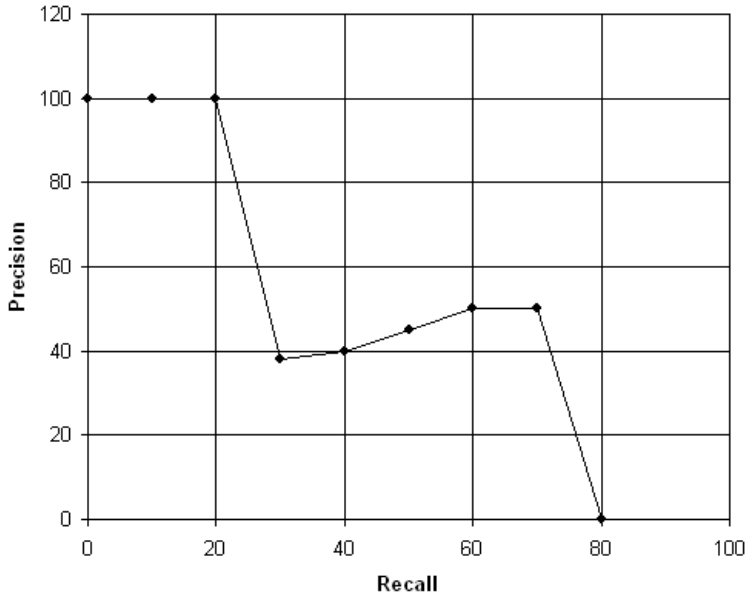

Figure 4.9: Precision versus Recall Plot

Averaging the performance over a set of queries has the disadvantage of hiding anomalies for distinct queries. To evaluate a retrieval algorithm for each query in an example set, single value summaries of the precision versus recall figures are useful. Three approaches for calculating a single value summary of a precision versus recall figure are examined next.

Average Precision at Seen Relevant Documents is calculated by averaging the precision at each recall level. In the above example, after each new relevant document is observed, the precision values are 100, 100, 38, 40, 45, 50 and 50. Thus, the average precision at seen relevant documents is:

$$(100 + 100 + 38 + 40 + 45 + 50 + 50)/7 = 60.4$$

Precision-at-k (P@k) is the precision at k returned documents. In the example, P@3 is 67%, because 2 among the first 3 documents in the ranking are relevant. A special case of P@k is *R-Precision*, where $k = R$ is the total number of relevant documents. The R-precision in the example is 40%, because there are 10 relevant documents in R_q, and 4 among the first 10 documents in the ranking are relevant. We will use P@k to evaluate of our solution.

Precision Histogram is used for comparison of two ranking models. The histogram is calculated by taking the difference between the P@k values of algorithm A and B for each query. If the histogram shows a positive value for the i-th query, then algorithm A yields a better result than algorithm B for query number i and vice versa.

Recall and precision require knowledge of the total number of relevant documents to be calculated. In large collections, information about this number is rare. To make it possible to perform relevance evaluation using precision and recall, reference collections exist where experts manually have selected the relevant documents.

Chapter 5
Searching the Vertical Web

"Judge a man by his questions, rather than by his answers."

- Voltaire

There is a trend towards more specific domain search tools, vertical searches, in the case of searching the Web. Some examples are image search, news search and shopping search, which are standard in every major search engine today. The advantage of splitting the different domains is that the document base often will be more homogeneous and semi-structured. Even if the structure of HTML-documents has been used for ranking purposes for a long time (as seen in Section 4.6.4), semi-structured documents differ because they do not only capture the structure of the documents, but also the semantics, usually represented as labelled XML fields. That is, we are moving from a pure information retrieval view towards a database centric view, where the structure of the data is known.

Traditional information retrieval techniques are based on a static document concept, ignore the document structure, and do not support structured queries. This impose serious restrictions if used for XML retrieval. Up to recently, most research regarding XML in search has taken the database point of view [12]. This approach is not feasible when it comes to information retrieval, since answers are either right or wrong, i.e. there is no concept of partially relevant answers. Also, the queries submitted are very specific, requiring extensive knowledge of the structure of the data to be searched in. Section 5.1 presents work regarding XML in the context of information retrieval where queries typically are vague and the answers are more or less "right".

Semi-structured documents generally lead to multiple criteria for how the returned documents should be ranked. This in turn arises questions like which criterion that is more important, and how to combine the results produced by different criteria. The latter task is named ranking aggregation in the case of score-based rankings. Section 5.2 addresses these challenges.

5.1 XML Retrieval

Listing 5.1 shows three example XML documents representing shopping items at http://shopping.yahoo.com. Note that the documents are not encoded exactly the same way as Yahoo! Shopping does today. Anyway, they could be encoded this way, and we will use this representation for presentation purposes. Figure 5.1 visualises the documents in the form of an XML tree.

The ability to represent the semantics of data creates a great opportunity for better information retrieval [23]:

- Documents may be categorised by scheme, and search limited to a particular XML scheme of interest. For instance, a search for *The Da Vinci Code* may be limited to the *video* document type in the example documents.

```
<document type="book">
    <year>2003</year>
    <title>The Da Vinci Code</title>
    <author>Dan Brown</author>
    <desc>A murder in the Louvre...</desc>
    <price>17.95</price>
</document>

<document type="music">
    <title>Bad</title>
    <song_list>
        <title>Bad</title>
        <title>The way you make me feel</title>
        <title>Speed Demon</title>
    </song_list>
    <artist>Michael Jackson</artist>
</document>

<document type="video">
    <title>The Da Vinci Code</title>
    <desc>A murder in the Louvre...</desc>
    <year>2006</year>
</document>

<document type="video">
    <director>Peter Jackson</director>
    <title>King Kong</title>
    <format>DVD</format>
</document>
```

Listing 5.1: XML Documents

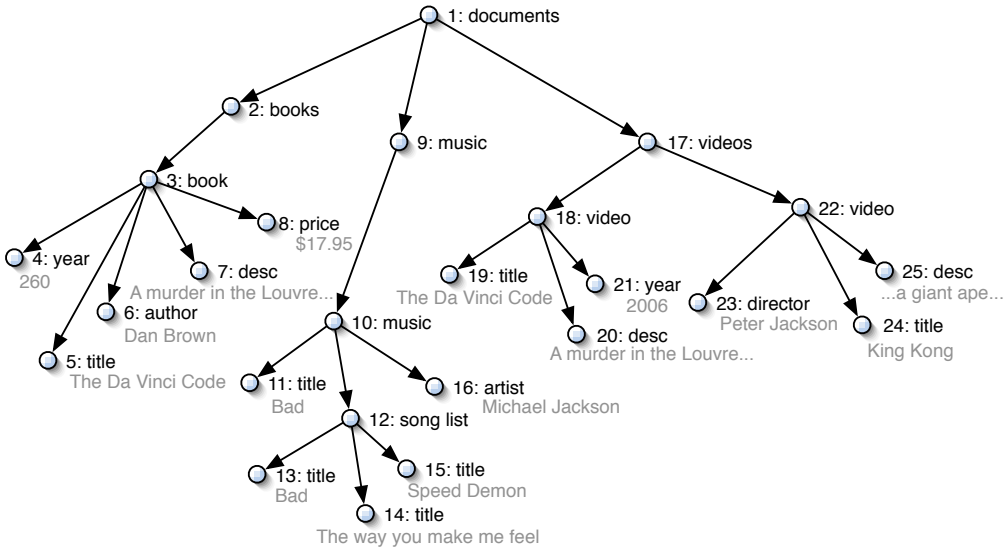

Figure 5.1: XML Tree

- Ambiguous words can be distinguished by the XML labels they appear in. For example, the name *Jackson* can be either the artist or the director.
- Special data types like numeric attributes may be separated and treated appropriately. An example is the *price* element, which should have other search predicates (like =, < and >) than string fields.
- Relevance can be computed based on structural proximity rather than keyword proximity. For instance, the nodes 19 and 20 are more structurally related than the nodes 5 and 20, even though they are equally similar as to keyword proximity.
- Only the part of the document of interest may be returned. For example, in an article document collection, only the relevant sections may be returned, instead of the whole article.

This section is divided into three parts, examining XML query languages, indexing, and ranking respectively.

5.1.1 XML Query Languages

In order to benefit from a semi-structured document collection, a lot of research explore *Content and Structure queries (CAS)* [19, 27, 50, 60, 61]. Such queries restrict the context of interest by referring to the document structure. The ability of posing highly specific queries is however at the cost of a complex query language and the need for knowledge about the document structure.

Due to its simplicity, *Keyword-Only queries* (Content-Only queries in INEX[1]), have become popular, especially in the case of search applications for common people. In contrast to structural queries, where elements that match structural requirements in the query are identified and then ranked, a part of keyword-only query processing is to determine semantics from syntax.

An approach somewhere in between is *Keyword and Label queries*. In this case, the user submits keywords, and optionally, XML label names where the keywords are to be found.

The three mentioned query language paradigms are explored next in the context of XML retrieval.

Content and Structure Query Languages

Most work considering content and structure query languages are centred around the XQuery [61] language proposed by the W3C group[2]. XQuery combines the features of two languages named XQL [19] and XML-QL [50]. XQL represents a document-centric view, retrieving elements from the original documents fulfilling the specified condition. On the other hand, XML-QL takes the data-centric view, offering a number of operators for restructuring the result, similar to standard database query languages. Yet another variation used in INEX, is XPath [60], a subset of XQuery, with an addition of an *about(path, string)* predicate. The about predicate specifies a certain context to be about a specific content. Table 5.1 illustrates some of the main operators of the XPath language.

An INEX CAS query may take two possible forms [55]:

`//A[B]` : Return A labels about B.
`//A[B]//C[D]` : Return C descendants of A where A is about B and C is about D.

Some examples of INEX CAS queries applied on the example documents in Listing 5.1, are presented in Table 5.2.

[1] INitative for the Evaluation of XML retrieval, http://inex.is.informatik.uni-duisburg.de/
[2] The World Wide Web Consortium (W3C), http://www.w3.org

Table 5.1: The XPath Query Language

Operator	Description
nodename	Selects all child nodes of the node.
/	Selects from the root node.
//	Selects nodes in the document from the current node that match the selection no matter where they are.
.	Selects the current node.
..	Selects the parent of the current node.
@	Selects attributes.
[]	Selects a specific node or a node that contains a specific value, specified as a predicate inside square brackets.
*	Matches any element node.
@*	Matches any attribute node.
node()	Matches any node of any kind.
|	Computes two node-sets.
+	Addition.
-	Subtraction.
*	Multiplication.
div	Division.
=	Equal.
!=	Not equal.
<	Less than.
<=	Less than or equal to.
>	Greater than.
>=	Greater than or equal to.
or	Boolean or.
and	Boolean and.
mod	Modulus.

Table 5.2: INEX CAS Queries

`//book[about(., a murder in louvre)]` Returns *book* elements that mention *a murder in Louvre*.
`//book[about(., a murder in louvre)]//title` Returns *title* elements of *books* that mention *a murder in Louvre*.
`//*[about(., a murder in louvre]` The retrieval engine must deduce the most appropriate element about *a murder in Louvre* to return.
`//*[about(*//title, the way you make me feel)]` Returns *any* element containing a *title* element about *the way you make me feel* as a descendant.
`//video[about(*//director, peter jackson) and year > 2000]//desc[about(., a giant ape]` Returns *description* elements about *a giant ape* from *videos* directed by *Peter Jackson* after year *2000*.

The query language XIRQL [27] is based on XQL. XIRQL extends XQL by means of a number of information retrieval related issues:

- Weighting of document and query terms. The weights are used to compute a retrieval status value, yielding a ranked list of results.
- Relevance-oriented search, i.e. only content are requested, not the type of elements to be retrieved. In this case, XIRQL relies on the *structured document retrieval principle* which states that a system should always return the most specific part of a document answering a query, unless otherwise stated [14].
- Vague predicates for various data types. Examples of vague predicates are "near" in the context of locations, and "broader" and "narrower" in the context of a thesaurus.
- Semantic relativism, meaning that the query language should take into account that a particular information could be encoded in different ways, for instance may information be encoded as either an XML element or an XML attribute.

The structure of a query may be applied in three ways [23]:

- *Proactively structured queries* are fully structured when first submitted by the user.
- *Reactively structured queries* are unstructured in the beginning and obtain structure by the user in an interactive process.
- *Automatically structured queries* are annotated with structure automatically.

As easily seen by the above query examples, fully structured queries may be fairly complex. In order to submit proactively structured queries, the user needs knowledge about the document structure. This is oftentimes not the case.

Reactively structured queries may be obtained by means of special adapted user interfaces. [23] suggests a multiple step interface where the user first chooses the appropriate structure and then fills data into a form. Another approach towards helping the user to add structure to a query, is autocompletion. Autocompletion may be used in a number of ways. Figure 5.2 shows an example where the user interface shows the possible contexts in which the query terms exist in the index. The user may then choose one of the suggested structured queries.

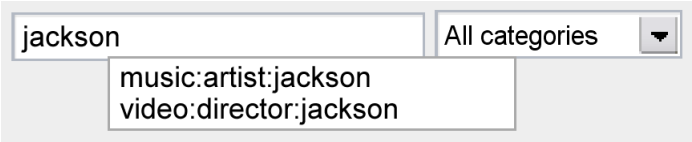

Figure 5.2: Autocompletion

Finally, the data may be explored in order to automatically produce a structured query from an initially unstructured one. An approach is to analyse the query and the result set in order to boost hits where the query terms appear in specific XML elements. For instance, the query `jackson director` could boost hits where the keyword `jackson` appears in the `director` element. This approach is user friendly, but could be distracting if the automatically added structure produces unexpected results.

Keyword-Only Queries

Keyword querying has emerged as the main paradigm for information discovery, especially in the Internet. The main advantage of keyword search is its simplicity. There has been several proposals of keyword based XML retrieval systems. This includes among others Nearest Concepts [52], XRANK [33] and XKSearch [64]. Schema-free XQuery [41] allows users to express queries

differently, depending on their knowledge of the document structure. The query expression may range from keywords-only, to regular XQuery queries.

Keyword and Label Queries

Although the golden mean, there has been little work focusing on simple query languages based on keywords, and optionally, XML label names and structure. However, there are a couple that are worth mentioning.

[12, 16] presents query languages that utilise both keywords, and optionally, the XML labels. Both proposals emphasize being simple.

[12] argues that XML documents should be searched via queries of the same nature, an idea from traditional information retrieval. I.e. XML documents should be searched via XML fragments. Listing 5.2 shows an example of an XML fragment query.

```
<document type="video">
    <director>Peter Jackson</director>
</document>
```

Listing 5.2: XML Fragment Query

In XSEarch [16], a query term consists of a keyword, a label, or a combination of a label and a keyword written in the form *label:keyword*. A query term may be required or optional.

We will explore keyword and label queries in our implementation. The queries will be annotated with structure automatically.

5.1.2 XML Indexing

Structured retrieval makes it possible to focus the search towards specific contexts, and dynamically adjust the granularity level of the returned documents. In order to achieve this, the index has to store both content and structure information.

A naive approach is to treat each XML element as a document and apply traditional information retrieval techniques. This solution has several drawbacks [33]. First, the inverted list would need to store both the element that directly contains the keyword, and all of its ancestors. This leads to space overhead. Additionally, if an element appears in the result set, all of its ancestors will also appear. Finally, regular approaches do not take structural proximity into account. This section presents several methods exploring properly indexing of XML documents.

A simple approach is taken in [12]. The authors suggest to use pairs of the form *(term, context)* as indexing units, instead of single terms as in traditional information retrieval. However, the answers returned consist of entire documents.

[27] proposes to use predefined XML elements as roots of index objects. Figure 5.1 shows the index objects of the XML elements *book*, *author*, *music*, *artist*, *video*, and *director* in dashed boxes. The index objects are disjoint, such that each term occurrence is considered only once. The main drawback of the work of [27] is that the concept of index objects is static. [30] generalises the model to support the retrieval of nodes at any granularity level. This is achieved by treating all basic XML elements as indexing objects.

In [42], a compact indexing tree, Ctree, allows for efficient retrieval for queries with any structure constraint and to retrieve nodes at any level. The Ctree construction consists of two steps. The

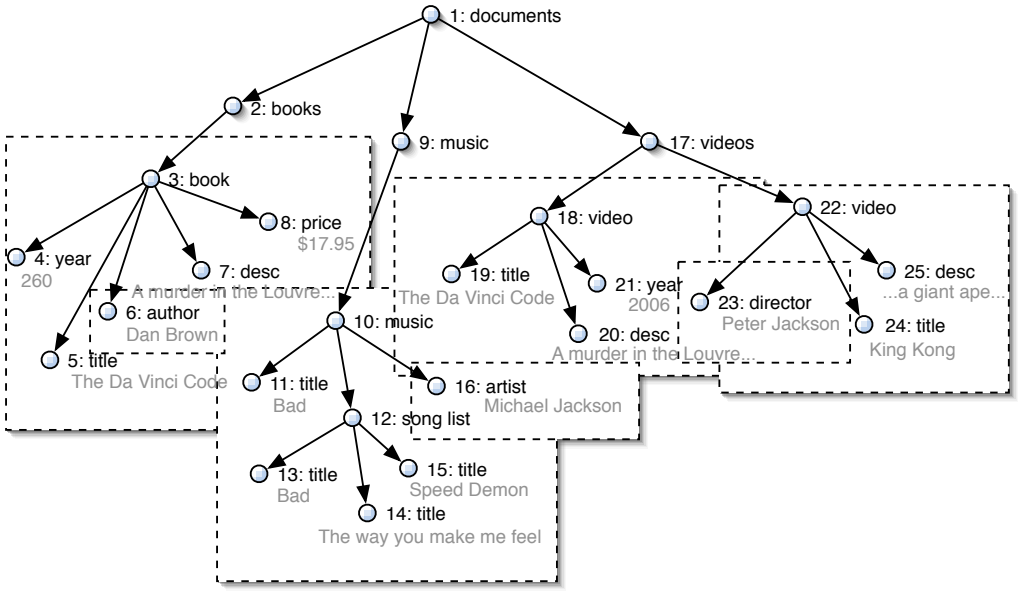

Figure 5.3: Indexing Nodes

first step is to cluster equivalent nodes from the XML tree, i.e. nodes with equal label paths (the list of labels of the nodes on the path from the root to the node), into groups. There is an edge from group A to group B if the label path of group A is the longest prefix of that of group B. Next, each group is associated with an array where the array index represents a list of equivalent nodes ordered by their preorder in the XML tree and the array values point to the corresponding parent elements. The last step preserves the hierarchical relationships among the individual nodes in the XML tree. Figure 5.4 shows the creation of a Ctree from the example XML tree in Figure 5.1.

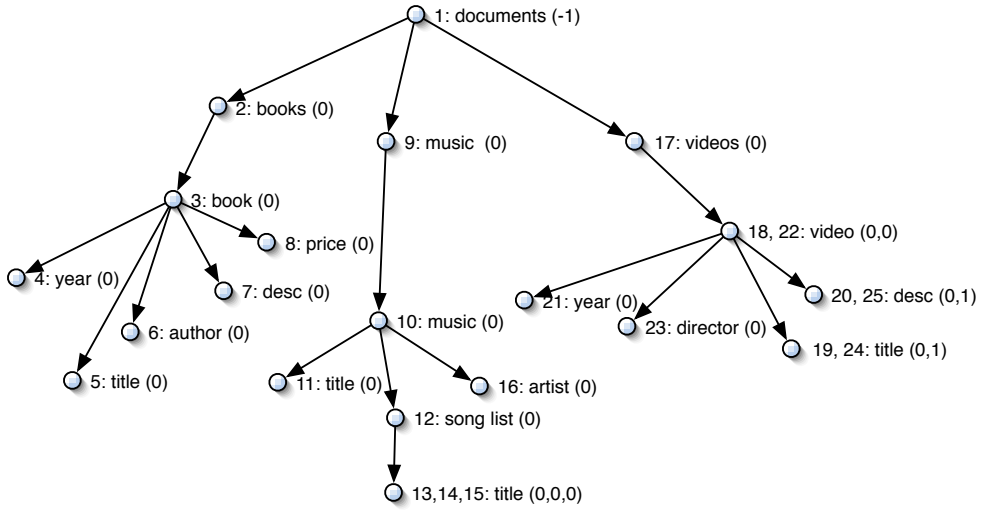

Figure 5.4: Ctree

The mentioned work aims to support retrieval at any (or specific) levels in the XML tree. Additionally, there has been some research regarding indexing that attempts to speed up computation of semantic relationships. The idea of semantic relationships is presented in the next section. We

will however cover the indexing theory here.

Most approaches take the Dewey numbering as a starting point. In Dewey numbering, each node is assigned the number of its parent concatenated with a number that represents its position among its siblings. Since the number of an ancestor is a prefix of the number of a descendant, anchor-descendant relationships are implicitly captured in Dewey numbering. The Dewey numbering of the nodes in our example document tree is shown in Figure 5.5.

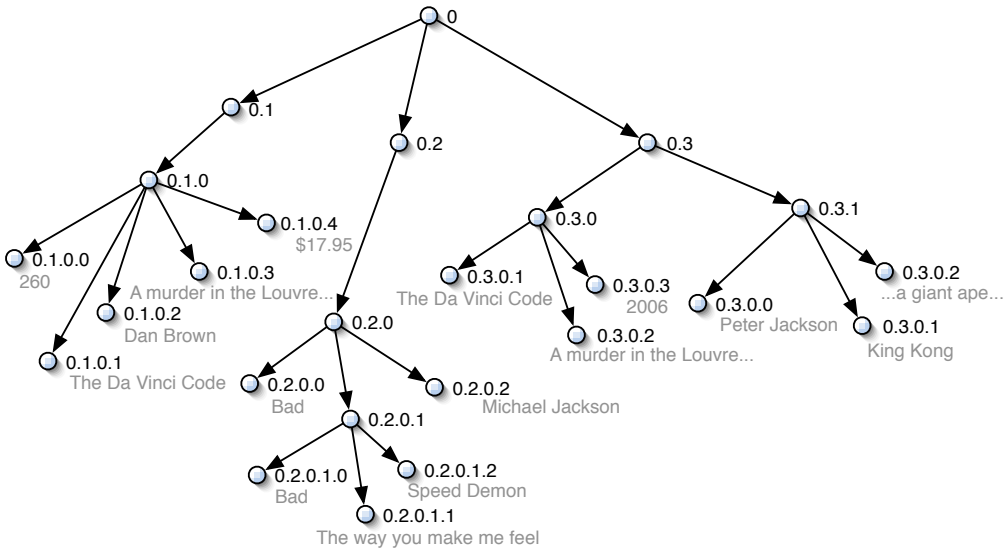

Figure 5.5: Dewey Numbering

[33] propose several inverted lists based on the Dewey numbering. The *Dewey Inverted List (DIL)* for a keyword, contains the Dewey number of all the XML elements that directly contain the keyword. For each keyword, the DIL is sorted by the Dewey number. A part of the DIL for our example is shown in Figure 5.6.

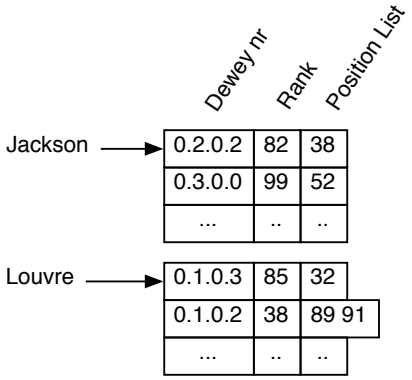

Figure 5.6: Dewey Inverted List

In addition to a traditional merge-join of the query keyword inverted lists, the result elements have to be inferred from the descendants. This is done by computing the longest common prefix, i.e. the deepest common ancestor, of the Dewey numbers in the different lists. The algorithm works in one single pass over the inverted lists. If the lists are long, this may be expensive, especially if the user only wants the top few results. A *Ranked Dewey Inverted List (RDIL)* is ordered with respect to

the rank value instead of the Dewey number. This way, higher ranked results are likely to appear first in the inverted lists, at the cost of more complex query processing. The performance of RDIL strongly depends on the keyword correlation, because in the case of low correlation, one keyword may appear in an element with high rank while another keyword may appear in an element with low rank. Since the keyword correlation seldom is known a priori, the authors suggest a *Hybrid Dewey Inverted List (HDIL)* that consists of the DIL and a small fraction of the RDIL. This is reasonable because RDIL is likely to outperform DIL only if it scans a small fraction of the full inverted list. The query evaluation uses RDIL until the estimated remaining time for RDIL is more than the estimated remaining time for DIL, then the query processing switches to DIL. A similar approach using B+-trees is presented in [64]. The problem of merging several ranked lists is examined in section 5.2.

A different approach is taken in XSEarch [16]. For each keyword, the paths in the documents that lead to that keyword are stored. Additionally, the set of nodes reachable by each path is stored. Similarly, for each label, the paths in the documents that lead to that label and the nodes reachable by those paths are stored. Finally, a path index is used to determine if the paths leading to keywords or labels are interconnected, and if so, the size of their interconnection tree. This size is used for ranking, as will be seen in the next section.

5.1.3 XML Ranking

As mentioned earlier, a structured document collection offers a new dimension when it comes to ranking, namely structural proximity. As mentioned, the *structured document retrieval principle*, states that a system should always retrieve the most specific part of a document answering the query [14], in case the query does not specify what kind of element to retrieve.

Work considering XML ranking mainly take two different approaches. The first part of this section covers work regarding adaption of the tf-idf weighting scheme to XML retrieval. The second part focuses on how to interpret the degree of semantic relationship between potential result nodes.

Other suggestions for XML ranking are ElemRank [33], a hyperlink metric similar to PageRank (see Section 4.6.5), extended to XML, and weighting of the different XML elements in order to mark some elements as more important than others [42].

Extensions of the tf-idf weighting scheme

Traditional information retrieval precalculates the measures *term frequency* and *inverse document frequency*, and stores them in the index. All documents in the base are considered when calculating the values. In the case of structured documents however, it could be desirable to calculate term frequency and inverse document frequency considering only the specific documents or parts of documents of interest. This may have serious impact on the ranking produced, specifically if the text statistics in the various domains differ to a high degree. For example, consider a document base containing books covering various categories. A query which specify the category of interest should return a ranking based on statistics from that category only.

A common method, explored in [27] among many others, is to associate each index object with term statistics. Searching may be performed at the level of the indexing nodes and hierarchical combinations of those. In the latter case, a technique named *augmentation* is used to downweight term statistics when the terms are propagated upwards in the document tree. The motivation is that content that is more distant in the document tree, is less important than content close to the context node.

Semantic Relationship

In all of the mentioned keyword-only search approaches [33, 41, 52, 64], keywords and labels are treated in the same way. Each query term is matched against every word of the document, even if this is a label name. The document structure is utilised during query processing, by deciding the degree of semantic relationship between the parts of the document that match the query. More semantically related results are ranked higher.

In the case of label and keyword queries, the XSEarch system [16] decides that an interior node satisfies a search term $l : k$ if it is labelled with l and has a descendent that contains the keyword k. Semanic relationship determines the ranking of the results. In [12], only approximate matching between the query and the documents is required. There is no requirement of semantic relationship.

Semantic relationship is most common implemented as some variation of the *Lowest Common Ancestor (LCA)*. LCA is defined as the deepest node in a tree that is an ancestor of two given leaves.

XSEarch [16] introduces the concept of interconnected nodes. Interconnected nodes are the set of connected nodes, where no two internal nodes have the same label name, and the root node is the LCA of leaf nodes. The size of the relationship tree is used as a factor in the ranking formula such that smaller sets are ranked higher because nodes close together are likely to be more meaningfully related. Interconnected nodes require that the same entities always have similar label names, and that the document tree contains exactly one logical hierarchy.

XKSearch [64] relies on the *Smallest Lowest Common Ancestor (SLCA)* semantics. The set of SLCAs is the set of nodes that contain the keywords either in their elements or in the elements of their descendants, and have no descendant node that also contains all keywords. The same variant is used in Schema-Free XQuery [41], although named as the *Meaningful Lowest Common Ancestor Structure (MLCAS)*.

The difference between the various approaches is easiest captured by some examples. Consider nodes 19 and 25 in Figure 5.1. Their LCA is the node 17. The nodes 19 and 25 are not interconnected nodes because there are nodes in their relationship tree that have the same label name, i.e. *video*. However, nodes 19 and 20 are interconnected nodes with node 18 as their LCA. Next, consider the query `da vinci code louvre`. The set of SLCAs consists of the nodes 3 and 18, which is the set of roots of all smallest answer subtrees.

5.2 Ranking Aggregation

Semi-structured documents generally lead to multiple criteria for how the returned documents should be ranked. The user may be indifferent regarding which criterion is more important, or he may prefer one criterion to another. This raises questions about expressing preferences, combining preferences, and querying preferences. This section addresses these challenges.

5.2.1 Expressing and combining Preferences

Preference expressions may be either *qualitative* or *quantitative*. The qualitative approach specifies preferences directly using relations, e.g. “I prefer X to Y”. In the quantitative approach, preferences are specified indirectly using scoring functions, e.g. “I like X with score 0.5 and Y with score 0.3” [15].

Note that the qualitative approach is more general than the quantitative one, since every scoring function can be defined by preference relations. An example is provided in Table 5.3. The preference "if the same ISBN, prefer lower price to higher price" gives the preferences "book 3 to book 1" and "book 1 to book 2". There is no preference between the first three books and book 4. A scoring function would have to assign equal score to book 4 and book 1-3, but this contradicts the preferences between the first three books. Thus, the preference can not be represented as a scoring function.

Table 5.3: Expressing Preferences

Book Nr	ISBN	Merchant	Price
1	0618517650	amazon.com	$63.00
2	0618517650	christianbook.com	$74.99
3	0618517650	walmart.com	$61.70
4	0739307312	www.fictionwise.com	$6.99

Another advantage of the qualitative approach is that it represents a natural way of expressing wishes. However, the qualitative approach suffers from tie resolution when multiple preferences are to be combined. Consider the example in Figure 5.7. Preference 1 states that "book 1 and book 2 are preferred over books 3-6", while preference 2 says that "books 3, 4, 1 and 6 are not preferred". The *pareto combination* only preserves those orders in consensus, while the *priority combination* emphasizes one of the preferences, in this case preference 1.

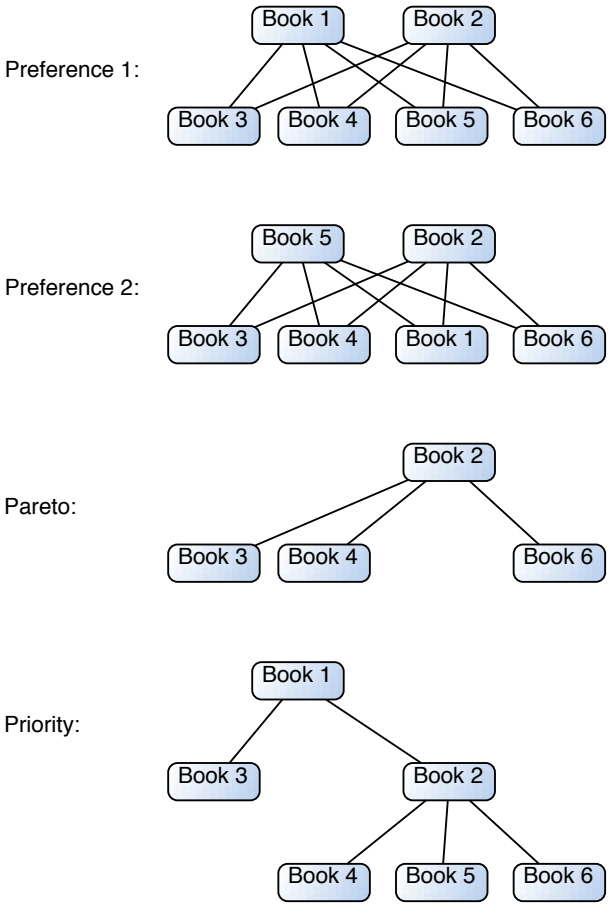

Figure 5.7: Preference Combination

In the case of quantitative preferences, combination is performed by means of a combining function [2]. Table 5.4 and 5.5 show two preference functions, i.e. mapping from records to numeric scores. An example combination function may state that "if price is higher than $70.00, then return max of preference function 2, otherwise return max of preference function 1". The record "(0618517650, amazon.com, $63.00)" will then return 0.8, which is max of preference 1.

Table 5.4: Quantitative Preference 1

ISBN	Merchant	Price	Score
*	amazon.com	*	0.8
0618517650	*	<$70.00	0.7
0618517650	*	>$90.00	0.3

Table 5.5: Quantitative Preference 2

ISBN	Merchant	Price	Score
*	walmart.com	*	0.2
0618517650	*	*	0.9

There exist efficient methods for computing the top-k answers, i.e. the k top hits with the highest scores, as we will see in the next section. However, it is not obvious how to specify scores and combining functions. Also, the total ordering of scores is not always reasonable. A solution may be to let the user specify qualitative preferences, which are mapped to score-based preferences that can be used for query processing.

5.2.2 Querying Preferences

[37] proposes the *Best-Matches-Only (BMO)* query model for processing of qualitative preferences. The model is based on the *better-than graph*. A better-than graph contains an edge $y \rightarrow x$ if there exists a preference $x < y$. Nodes in the graph without a predecessor are named *maximal elements*. x and y are unranked if no directed path exists between x and y. The better-than graph for the tuples in Table 5.3 is shown in Figure 5.8.

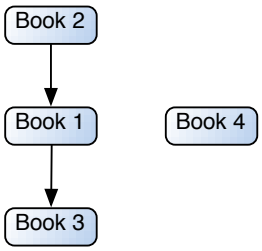

Figure 5.8: Better-Than Graph

The BMO model retrieves perfect choices, i.e. the maximal elements, if present. These points constitute what is called the skyline. If there are multiple criteria that have to be combined, the pareto preferences are used as the basis for the better-than graph. Plenty of researchers have addressed the problem of answering BMO without fully computing the pareto preferences [10, 39, 56]. We will not examine this problem further, because both qualitative preferences and best-matching only, are less relevant in the context of Web search. The rest of this section examines the *ranking aggregation* problem, i.e. combination of several score-based orderings.

The ranking aggregation problem is heavily studied related to middleware scenarios, or meta search. Meta search connects several databases or document collections in order to answer user queries. Another application area is comparison of several ordered lists in order to obtain knowledge of how the various ranking features contribute to the overall ranking. Typically, only the top-k answers are of interest.

The algorithms are highly driven by minimising total access cost

$$c_t = s \cdot c_s + r \cdot c_r$$

where c_t is the total access cost, s and r are the number of sorted and random accesses, and c_s and c_r are the cost of sorted and random access. The various algorithms specialise in different scenarios, ranging from cheap sorted and random access cost, to sorted or random access not possible. Some algorithms are presented in the following. Table 5.6 explains some basic notation related to the task of ranking aggregation.

Table 5.6: Basic Concepts related to Ranking Aggregation

Notation	Explanation
D	Domain
τ	An ordered list of a subset $S \subseteq D$. A full list, permutation, contains all the elements in D. A partial list contains a subset of the elements in D. A top-k list contains the k top ranked elements in S.
$\tau(i)$	The rank of $i \in D$. Higher numbers indicate better ranks.
τ_T	The projection of τ with respect to $T \subset D$.
$\sigma \succeq \tau$	The extension σ of a top k list τ is a permutation such that $\sigma(i) = \tau(i)$ for all $i \in D_\tau$.

Borda's Method

A naive algorithm for obtaining an aggregated list from several ordered lists, is to assign a score to each item corresponding to the position in which the item appears in the ordered lists. The aggregated list is sorted with respect to a total score of each item.

In particular, in the case of full lists $\tau_1, \tau_2, ..., \tau_m$, Borda's method [9] assigns a score $B_i(c)$ equal to the number of candidates ranked below c in τ_i. The total Borda score is defined as:

$$B(c) = \sum_{i=1}^{m} B_i(c)$$

Borda's method may be thought of as assigning an m-element position vector to each item, and sorting the items by the L_1 norm of these vectors. In general, the items could be sorted by the L_p norms, the median, or the geometric mean.

The naive algorithm has two main drawbacks; it is linear in database size, and it is not clear how to extend it to partial lists.

Fagin's Algorithm

Fagin [26] first proposed an algorithm to the ranking aggregation problem not needing to access every item in the database. In this context, a database consists of m lists, each of length N which is the number of items in the database. Each item is associated with m scores $x_1, x_2, ..., x_m$ where

$x_i \in [0,1]$ for each i. The x_i value may be interpreted as the degree of relevance for the i-th dimension. The m lists are sorted in descending order by the x_i fields.

Fagin's algorithm consists of three steps:

1. Do sorted access to each of the m sorted lists until there is a set H of at least k items that has been seen in each of the m lists.
2. For each item R in H, do random access to each of the lists to find the scores x_i of R.
3. Apply an aggregation function t to compute the overall score $t(R) = t(x_1, x_2, ..., x_m)$ for each item R in H. The output of the algorithm is an ordered list of the k items with the highest scores.

Popular aggregation functions are *min*, *max*, and *average*. An aggregation function is *monotone* if $t(x_1, x_2, ..., x_m) \leq t(x_1^{'}, x_2^{'}, ..., x_m^{'})$ whenever $x_i \leq x_i^{'}$ for every i. An aggregation function is *strict* if $t(x_1, x_2, ..., x_m) = 1$ whenever $x_i = 1$ for every i. All of the mentioned aggregation functions are monotone, but only min and average are strict. Fagin's algorithm is correct for monotone aggregation functions, and optimal with high probability in the worst case if the aggregation function is strict and the orderings in the sorted lists are probabilistically independent.

The Threshold Algorithm / Quick-Combine / Multi-Step

The threshold algorithm was suggested independently as *The Threshold Algorithm* [26], *Quick-Combine* [31] and *Multi-Step* [45]. The algorithm is an improvement of Fagin's algorithm, and goes as follows:

1. Do sorted access to each of the m sorted lists. As an item R is seen in some list, do random access to the other lists to find the scores x_i of R. Apply an aggregation function t to compute the overall score $t(R) = t(x_1, x_2, ..., x_m)$ of R. Remember item R and its score $t(R)$ if this score is one of the k highest scores seen.
2. Let $\hat{x_i}$ be the score of the last item seen in list L_i. Stop when at least k items have been seen whose score is at least equal to *the threshold value* $\tau = t(\hat{x_1}, \hat{x_2}, ..., \hat{x_m})$.
3. The output of the algorithm is an ordered list of the k items with the highest scores.

The threshold algorithm may be further improved by exploiting the distribution of scores relative to the ranks on which they occur. The idea is to force the threshold value to decrease quickly, increasing the probability of newly found items having a grade above the threshold value. This is achieved by preferring lists showing a behaviour of declining scores most rapidly relative to the ranks for evaluation. In the same way, lists that represent more important query terms may be preferred.

The stopping rule for the threshold algorithm always occur at least as early as the stopping rule for Fagin's algorithm. The threshold algorithm is *instance optimal* for all monotone aggregation functions. An algorithm $\mathcal{B}$ is instance optimal if $cost(\mathcal{B}, \mathcal{D}) \leq c \cdot cost(\mathcal{A}, \mathcal{D}) + c^{'}$ for every algorithm $\mathcal{A}$, database $\mathcal{D}$ and constants c and $c^{'}$. Finally, the threshold algorithm is cheap in space, only requiring bounded buffers.

The threshold algorithm may be modified to a θ-approximation algorithm, $\theta > 1$, by transforming the stopping rule to halt when at least k items have been seen whose grade is at least equal to τ/θ.

No Random Access / Stream-Combine

Both Fagin's algorithm and the threshold algorithm rely on random accesses. In some scenarios, random accesses are expensive or simply not possible. [26] presents algorithms for both cases,

named *Combined Algorithm* and *No Random Access (NRA)* respectively. A similar approach as NRA is suggested as *Stream-Combine* in [32]. We will explore the case of no random accesses, as this is true for text retrieval systems.

NRA is described in the following:

1. Do sorted access to each of the m sorted lists. As an item R is seen in some list, upper and lower bounds are calculated. The upper bound is calculated by substituting the last value obtained in list i, $\hat{x}_i$, for each missing score. In the same way, the lower bound is computed by substituting each missing score with the value 0. The current top-k list contains the k items with the largest lower bounds, ties are broken using the upper bounds.
2. Halt when at least k objects have been seen, which lower bounds are greater than the upper bound scores of all unseen items, that is, the upper bound of the last values obtained in the lists so far.

Note that the algorithm does not output the grades of the items. If the sorted order is of interest, this has to be determined by computing the top-1 list, top-2 list, ..., top-k. The cost of finding the top-k items in sorted order is thus $k \times \max_i C_i$, where C_i is the cost of finding the top-i items. NRA is instance optimal for monotone aggregation functions.

MPro / Upper

The last scenario is the case when sorted access is not supported. MPro [13] and Upper [11] provide solutions to this problem. We will not investigate this problem further as it is somewhat outside the scope of this thesis.

Optimal Ranking Aggregation

The concept of *optimal ranking aggregation* depends on what distance measure that strives to be optimised. In the following, a closer look at two approaches to optimal aggregation will be examined, each optimal in the meaning of minimising two popular distance measures; *the Kendall tau distance* and *the Spearman footrule distance*.

Optimal ranking aggregation is considered less relevant in the context of the thesis. The theory discussing distance measures may however be useful as to further evaluation of our solution.

The Kendall Tau Distance In the case of measuring the distance between two permutations, the Kendall tau distance is the number of times in which the two lists disagree regarding the ordering of two items:

$$K(\sigma, \tau) = |\{(i, j) \quad | \quad i < j, \quad \sigma(i) < \sigma(j) \quad but \quad \tau(i) > \tau(j)\}|$$

The measure is normalised by dividing by the maximum possible value $\binom{|S|}{2}$.

The Kendall tau distance of σ to several lists $\tau_1, ..., \tau_k$ is given by:

$$K(\sigma, \tau_1, ..., \tau_k) = \frac{1}{k} \sum_{i=1}^{k} K(\sigma, \tau_i)$$

In order to generalise the Kendall tau distance measure to partial lists, [24] introduces the Kendall tau distance with penalty parameter

$$\bar{K}_{i,j}^{(p)}(\tau_1, \tau_2)$$

where $0 \leq p \leq 1$. τ_1 and τ_2 are two top k lists, such that $\sigma_1 \succeq \tau_1$, $\sigma_2 \succeq \tau_2$, and σ_1 and σ_2 are permutations of $D_{\tau_1} \cup D_{\tau_2}$. Table 5.7 describes the four possible cases in which i and j may appear in the two lists, together with the penalty value for each case.

Table 5.7: The Kendall Tau Distance Generalization to Partial Lists

Case	**Penalty**
Both i and j appear in both lists:	
i and j are in the same order.	0
i and j are in the opposite order.	1
Both i and j appear in τ_1, and i appears in the τ_2:	
$\tau_1(i) > \tau_1(j)$.	0
$\tau_1(i) < \tau_1(j)$.	1
Note that this case generalise to *both i and j appear in one list, and one of i or j appears in the other list.*	
i appears in one list, and j appears in the other list.	1
i and j appear in one list, but neither i nor j appears in the other list.	p

The optimistic approach, $p = 0$, in the last case, gives the minimising Kendall distance:

$$K_{min}(\tau_1, \tau_2) = \min(K(\sigma_1, \sigma_2))$$

Similarly, the neutral approach $p = 1/2$, gives the average Kendall distance:

$$K_{avg}(\tau_1, \tau_2) = E(K(\sigma_1, \sigma_2))$$

The Kendall tau distance is also called the bubble sort distance because it corresponds to the number of times two items are swapped by using bubble sort to transform one of the lists into the other.

The Spearman Footrule Distance The Spearman footrule distance is the sum over all elements $i \in S$, of the absolute difference between the ranks of i in the two lists:

$$F(\sigma, \tau) = \sum_{i=1}^{|S|} |\sigma(i) - \tau(i)|$$

The measure is normalised by dividing by the maximum possible value $\frac{|S|^2}{2}$.

Like the Kendall tau distance, the Spearman footrule distance may be extended to several lists and partial lists. In the case of partial lists, [24] replaces all missing elements in each of the lists with a location parameter l. An intuitive choice for l is $k + 1$.

Kendall/Kemeny Optimal Aggregation A lot of research regarding finding the best consolidation of several ordered lists is carried out in the context of social choices [22]. The Condorcet alternative was first proposed by Marie J. A. N. Caritat and Marquis de Condorcet in 1785 [18]. They stated that if there exists some alternative that defeats every other in pairwise simple majority voting, then this alternative should be ranked first. The extended Condorcet criterion, due to Truchon [57], says that if there is a partition $(C, \bar{C})$ of S such that for any $x \in C$ and $y \in \bar{C}$, the majority prefers x to y, then x should be ranked above y.

A Kendall optimal aggregation of a collection of partial lists $\tau_1, \tau_2, ..., \tau_k$, minimises $K(\pi, \tau_1, \tau_2, ..., \tau_k)$ over all permutations π. The aggregation obtained by optimising the Kendall distance is also called the Kemeny optimal aggregation after the person who first proposed it [35, 36]. Kemeny optimal aggregations have the desired property of satisfying the extended Condorcet criterion.

Computing the Kendall optimal aggregation is NP-hard [21]. The authors therefore introduce the concept of a locally Kendall optimal aggregation. A list π is a local Kendall optimal aggregation if it is impossible to reduce the total Kendall distance by flipping an adjacent pair. Like the Kendall optimal aggregation, also the local version satisfies the extended Condorcet criterion. A local Kendall optimal aggregation can be computed in time $O(kn \log n)$ as this problem is similar to finding a Hamilton path in a majority graph for $\tau_1, \tau_2, ..., \tau_k$. A majority graph has an edge (x, y) from x to y if a majority of the τ's that contain both x and y, rank x above y.

The researchers further propose that by applying *the local Kemenization* procedure to any initial aggregation, a ranking is obtained that is maximally consistent with the initial aggregation, and simultaneously satisfies the extended Condorcet criterion. This makes it possible to benefit from the initial aggregation method. The local Kemenization procedure picks an element from the top of the inital ranking, places it at the bottom of the local optimal ranking, and bubbles it up as long as the majority of the τ's agree.

Footrule Optimal Aggregation The Kendall distance for full lists σ, τ, may be approximated by the Spearman footrule distance [20]:

$$K(\sigma, \tau) \leq F(\sigma, \tau) \leq 2K(\sigma, \tau)$$

It follows that if σ is a Kendall optimal aggregation of full lists $\tau_1, ..., \tau_m$, and $\sigma^{'}$ is a footrule optimal aggregation, then [21]:

$$K(\sigma\prime, \tau_1, ..., \tau_m) \leq 2K(\sigma, \tau_1, ..., \tau_m)$$

[25] proposes an algorithm, *median rank aggregation*, that gives an optimal solution for a notion of distance similar to the footrule distance. Median rank aggregation sorts the items based on the median of the ranks in each list. The algorithm can be summarised as follows: Access the ranked lists from the m voters, one item of every list at a time, until some item is seen in more than half of the lists - this is the winner. The procedure continues until k items are discovered. The algorithm is instance optimal and uses no random accesses.

Chapter 6
Vespa – The Yahoo Vertical Search Platform

"Far better an approximate answer to the right question, than the exact answer to the wrong question, which can always be made precise."

– John Tukey

In this chapter, a short introduction to *Vespa*, the Yahoo! vertical search platform, is provided. The chapter is based on the Vespa documentation. Note that because of sensitivity matters, some simplifications have been made in the description of Vespa.

Vespa is a generic, scalable platform, used in more than 40 of Yahoo! search applications. It supports search applications with varying:

- **data size** - from data fitting on one machine to Web-scale.
- **query load** - from a single machine to clusters handling thousands of queries per second.
- **query formality** - from raw end user input to deeply nested logical combinations of terms and conditions.
- **data diversity** - from one type to tens or hundreds of different kinds of data formats, each processed in a unique way.
- **data formality** - from raw pages of text to hundreds of formal fields.

Figure 6.1 presents an overview of Vespa. The *content* module controls fetching of data into the system. To search this content, Vespa creates an index structure from the input data (*indexing*). Search results are served from this index as responses to incoming queries. The machines, processes, and services involved in the entire pipeline are controlled from a single *administration server*.

In Figure 6.2, an overview of the search core in Vespa is presented. The search core is responsible for:

- finding all documents matching an incoming query by using the indices. Indexing is covered in Section 6.3.
- calculating the rank score (relevance) of each hit. Ranking will be examined in Section 6.4.1.
- performing database operations like aggregating information over all the hits using attributes, and sorting, if requested in the query. These operations will be explained in section 6.4.2.
- selecting the subset of hits to return to the user.
- processing and returning the document summaries of the selected hits.

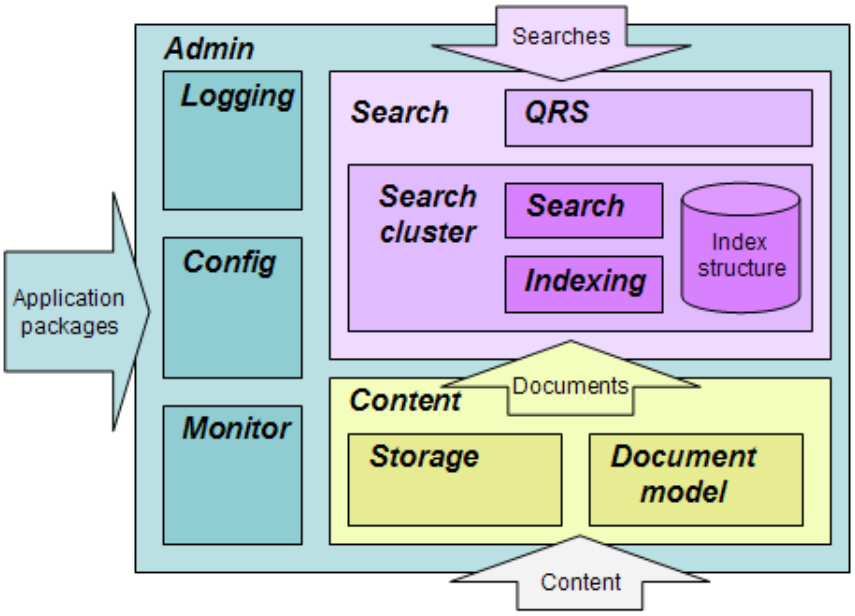

Figure 6.1: The Main Components in Vespa

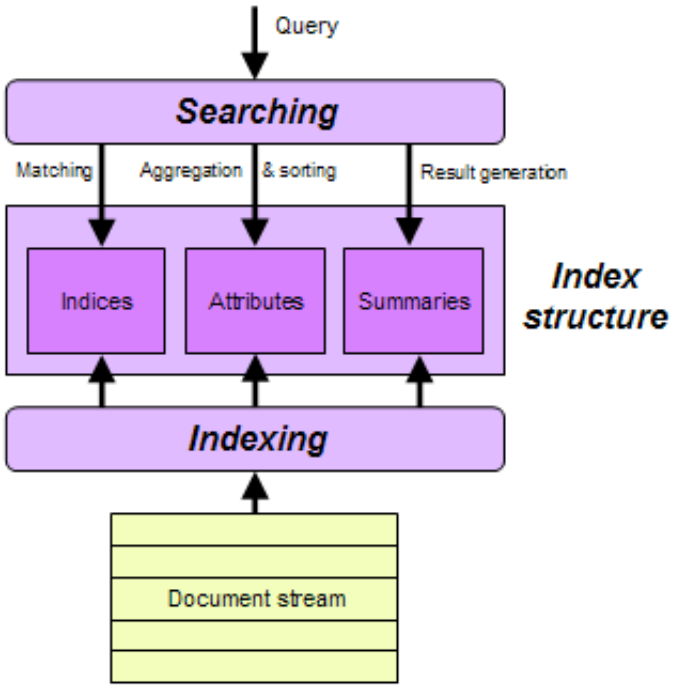

Figure 6.2: Vespa Search Core

6.1 Document Fetching

Data fed into Vespa is in the form of documents. Documents may be fetched from a number of different sources, mainly using two different approaches. In the first approach, Yahoo! Shopping gets product information from shops that pay money to get their products indexed. These records end up in a database, and are next dumped to XML files that are fed into the Vespa feeder. An example of the structure of the XML files is illustrated in Listing 6.1. In the second approach, a crawler is used to fetch documents. The crawler extracts data from HTML pages on the Web, in a similar way as presented in Section 4.2.

```
<document type="book">
    <title>The Da Vinci Code</title>
    <author>Dan Brown</author>
    <desc>A murder in the Louvre...</desc>
    <price>17.95</price>
    <year>2003</year>
</document>
```

Listing 6.1: A Shopping Document in Vespa

The content system (see Figure 6.1) in Vespa is responsible for handling content until it is delivered for indexing. Index structures are partitioned and duplicated by means of three dimensions. The number of *search clusters* depends on the variety of document types. Typically, there is one search cluster for each document type. Each search cluster consists of a number of *rows* and *columns*. All machines at the same row are duplicates of each other. Thus, the number of rows needed depends on the query load or redundancy requirements. The number of columns is determined by the number of documents. The documents are routed to the appropriate search clusters via *dispatch* nodes.

In Listing 6.2, a simple example of how Vespa XML documents are fed, is illustrated. Each label directly contained in the `vespafeed` top-level label represents a command to be sent to the Vespa distributor. In this case, a `startoffeed` message is sent first, then an add document message (`vespaadd`), a remove message (`vesparemove`), and finally an `endoffeed` message. The labels `vespaadd` and `vesparemove` may contain a number of documents to be added to the index. Each document must have a specified document type, matching a search definition (explained in Section 6.3).

6.2 Text Operations

This section briefly presents some text operations in Vespa. The operations discussed in the following are stopwords, normalising and stemming/lemmatisation.

6.2.1 Stopwords

Stopwords are not removed in Vespa. The most frequent words for different languages are rather assigned lower ranks when indexing. Words that occur too frequently in documents can be ignored.

```
<vespafeed>
   <startoffeed>
      <name>book</name>
   </startoffeed>

   <vespaadd>
      <document type="book">
        <title>The Da Vinci Code</title>
        <author>Dan Brown</author>
        <desc>A murder in the Louvre...</desc>
        <price>17.95</price>
        <year>2003</year>
      </document>
   </vespaadd>

   <vesparemove>
      <documentid type="book">
        <title>Angels & Demons</title>
        <author>Dan Brown</author>
        <desc>High atop the steps...</desc>
        <price>10.96</price>
        <year>2001</year>
      </documentid>
   </vesparemove>

   <endoffeed>
      <name>book</name>
   </endoffeed>
</vespafeed>
```

Listing 6.2: A Vespa XML Feed

6.2.2 Normalising

Normalising in Vespa will cause accents and similar decorations which are often misspelled, to be normalised the same way both in documents and queries. Table 6.1 provides examples of how some special characters are normalised in Vespa.

Table 6.1: Normalising in Vespa

Normalising Examples			
à → a	ç → c	ñ : n	ù → u
â → a	è → e	ò → o	ú → u
á → a	é → e	ó → o	û → u
ã → a	ê → e	ô → o	ü → u
ä → a	ë → e	õ → o	ý → y
å→ a	ì → i	ö : oe	ÿ →y
æ→ a	í → i	œ→ oe	ß→ ss
	î → i	ø → oe	þ→ th

6.2.3 Stemming/Lemmatisation

Both algorithmic methods and dictionary look-ups are currently in use for finding the root forms of the words. Words are converted to stems both when indexing and searching.

There are three modes of stemming; *none* stems none of the words, *nouns* stems only the nouns, and *all* stems all words. Stemming mode may be set on the level of fields, indices, or for all searches.

6.3 Indexing

When documents are transmitted by the content system to the indexing nodes, Vespa builds an index structure of the data. The index structure consists of three major parts:

- **Indices** - Inverted indices used to efficiently match queries to documents and calculate the query dependent rank (see Section 6.4.1) of a document. Indices are stored on disk and cached partially in memory.
- **Attributes** - Selected document fields whose values are stored in memory for each document. Attributes are used to do hit sorting and aggregation (see Section 6.4.2).
- **Document summaries** - The document fields which should be included when a document is returned to the front-end. Summaries are stored on disk.

The index structure to be built for a given document type, is described in the *search definition (SD)* file. Search definitions describe the application's document types, and how they should be indexed, ranked, searched, and presented. All Vespa applications must create at least one search definition that contains at least one document type. Each document type contains a list of all the fields that make up the logical information units of the document, e.g. title and description.

In Listing 6.3, a simplified example of an SD file is provided. This search definition makes it possible to feed documents having `title`, `artist` and `popularity` as field names. The search definition can also define transformations of the document data to be done during indexing.

The indexing controller (see Figure 6.1) is responsible for receiving incoming documents from the content system, initiating indexing, and controlling which revision of the index that should be used for searching at any time. There are three modes for indexing; *batch*, *incremental*, and *real-time*. In batch indexing, all documents have to be fed every time the index is updated. The Shopping vertical currently uses batch indexing. Real-time indexing feeds documents as they become available. News search is a typical real-time installation, as yesterday's news is today's history. Incremental indexing supports partly updates, and yields better control than real-time indexing because it provides index revision numbers.

```
# A basic search definition - called music, should be saved to music.sd
search music {

  # It contains one document only - called music as well
  document music {

    field title type string {
      indexing: summary | index # How this field should be indexed
      index-to: title, default  # Field is stored in two indices
      rank-boost: 9000          # Ranking importance of this field
      rank-type: about          # Type of ranking settings to apply
    }

    field artist type string {
      indexing: summary | attribute | index
      index-to: artist, default
      rank-boost: 7000
      rank-type:about
    }

    # Increase rank score of popular documents regardless of query
    field popularity type int {
      indexing: summary | staticrank
    }

  }

}
```

Listing 6.3: A Search Definition File

6.4 Ranking

This section first describes how documents are ranked in Vespa. Next, the database operations sorting, collapsing, and aggregation are presented.

6.4.1 Dynamic and Static Rank

Ranking is the calculation of a rank value per document that matches a query, used for ordering the result set in the absence of sorting or other reordering operations. Ranking should be used when the primary intent of the search is to find the documents that are most relevant to the query terms.

Document rank can either be based on the query, assigned preferences, or both. The two approaches are named *dynamic rank* and *static rank*, respectively.

- **Dynamic rank** is a summed score of how well the different parts of the document matches the query. The rank value is based on tf-idf measures, as well as in what field of the document the match occurs. Each field defined in the search definition file that participates in the dynamic rank generation, may contribute with different strength in the dynamic ranking. In the example in Listing 6.4, the `title` and the `desc` field are assigned a rank boost of 9200 and 8200, respectively. This will give the title field higher influence on the overall ranking than the description field. In addition, the rank value is higher as earlier the query terms occur in the document. For XML documents, this feature is calculated on a per field basis. The dynamic rank is also based on the proximity of the query terms, phrasing, freshness, and location.
- **Static rank** is a numerical value assigned to a document that is independent of the query evaluation. Each document is assigned one or more static rank values. Static rank value(s) to use for a document type is specified in the search definition file. These values are based on document field values only, for example editor assigned priorities, connectivity (e.g. number of inbound links), date modified, product popularity, or preferred provider.

```
1  field title type string {
2      ...
3      rank-boost: 9200  # Ranking importance of this field
4      ...
5  }
6
7  field desc type string {
8      ...
9      rank-boost: 8200  # Ranking importance of this field
10     ...
11 }
```

Listing 6.4: Dynamic Boost in a Search Definition File

The document rank is calculated as a function of the dynamic rank and static rank

$$R = f(r_D, r_S)$$

where R is the total rank, r_D is the dynamic ranking and r_S is the static rank value.

6.4.2 Sorting, Collapsing, and Aggregation

Sorting, *collapsing* and *aggregation* are traditional database operations that are implemented as a part of Vespa. In this section, these techniques will be explained.

Sorting

Sorting is the explicit ordering of documents on a list of field values. Sorting should be used for searches where the user would like to see a list of documents ordered by a field value or feature. Note that ranking may be one of these values. Sorting may also be combined with ranking, for example by sorting on price and then rank if price is equal.

Collapsing

Hits can be sorted into buckets by performing collapsing. Collapsing is a feature which sorts hits into buckets and returns only the most relevant hit(s) in each bucket. Collapsing cannot be combined with sorting, since both are reordering operations over the result set, and a sorting operation would therefore break the collapsing or vice versa. A useful application of this operation is to collapse on categories, displaying the most relevant hits for each category in the top-k result.

Aggregation

Aggregation is the process of collecting statistical information about the entire result set for a given query, in addition to the regular hits. *Unique aggregation* gives a count for each of the different values for a particular field. *Bucket aggregation* returns a count for the number of hits in each bucket. The buckets may be either fixed width, or the desired number of buckets may be specified. In the last case, the width of each bucket is adjusted so that the count of hits in each bucket is distributed as evenly as possible. A useful application of this operation is to aggregate on categories, and then in the front-end displaying the count of the hits within each category in the result set.

6.5 Customised Query and Result Processing

The *Query Result Server (QRS)* accepts search requests on HTTP, and returns results in a Vespa XML format, or at any format customised by the application. In addition, it is responsible for:

- parsing incoming queries.
- doing query transformations.
- sending queries to the dispatch nodes of multiple search clusters, and blending the results returned into one final result.
- processing results.
- caching results.

These operations are by default carried out as specified in the search definitions and the query, but can also relatively easily be customised by the application by writing *Searcher plug-ins*. Developing such a plug-in will be the main focus in the practical part of this thesis.

Inside the QRS, all queries and their results pass through a *search chain*, visualised in Figure 6.3. The search chain consists of *Searcher* plug-ins which receive queries and return results. The plug-in receives a query, processes it, sends the query to the next Searcher in the chain, receives the result, processes it, and returns the results. Listing 6.5 shows an example of a Searcher plug-in written in Java. The method call `super.doSearch()` sends the query to the next Searcher, and returns the result. The example plug-in resorts the result by title.

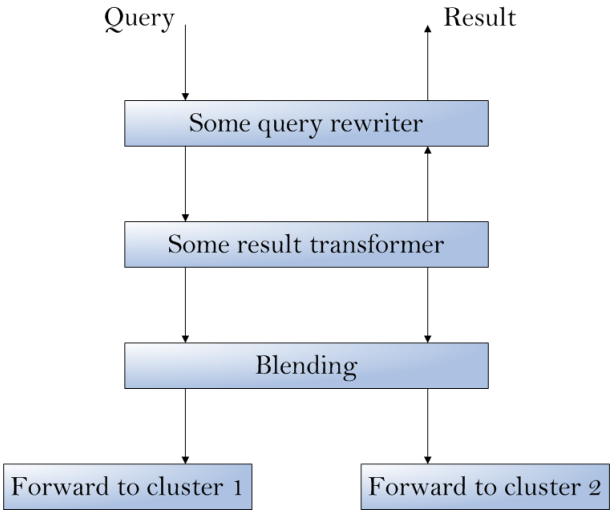

Figure 6.3: QRS Searcher Chain

```
1  package com.yahoo.example;
2
3  import java.util.List;
4  import java.util.Collections;
5
6  import com.yahoo.prelude.*; //The QRS API
7  import com.yahoo.prelude.searcher.HitOrderer;
8
9  public class TitleOrderingSearcher extends ChainedSearcher {
10
11     private TitleOrderer titleOrderer=new TitleOrderer();
12
13     public Result doSearch(Query query,int offset,int hits) {
14         Result result=super.doSearch(query,offset,hits);
15         result.setHitOrderer(titleOrderer); // Causes resort too
16         return result;
17     }
18
19     private static class TitleOrderer extends HitOrderer implements java.util.Comparator {
20
21         /** Called by result when a sort is needed */
22         public void order(List hits) {
23             Collections.sort(hits,this);
24         }
25
26         /** Called by Collections to carry out the sort */
27         public int compare(Object o1,Object o2) {
28             Hit hit1=(Hit)o1;
29             Hit hit2=(Hit)o2;
30             return (hit1.getProperty("title").compareTo(hit2.getProperty("title")));
31         }
32     }
33 }
```

Listing 6.5: A Searcher Plug-in

Part III

Implementation and Experiments

Chapter 7
Preliminary Ideas Regarding The Searcher

"The best way to have a good idea is to have lots of ideas."

– Linus Pauling

In this chapter, we present the preliminary ideas we have had regarding the Searcher plug-in. In the following sections, we present the various ideas, and provide a discussion of why we did not implement them in our solution.

7.1 Focusing the Search towards Main Categories

In this section, we present the approach of focusing the search towards the categories that are considered to be most relevant to the query. First, the approach is described, followed by its pros and cons.

7.1.1 Approach

One of the first preliminary ideas we had was to focus the search towards the categories relevant to the query. The motivation was to narrow the result set to the categories relevant to the query. A positive aspect with this approach was to reduce the price sort problem previously discussed in Section 2.3.2. By narrowing the search down to the main categories, the price sort would to a higher degree be performed on the intended products. For example, when performing the search `canon digital camera`, the main category would be *digital cameras*. Hence, products like AC adaptors and camera cases (belonging to the *accessories* category) would not appear in the result set, and thereby not be included when sorting on price.

The various approaches we examined were:

- *Analysing the retrieved results.* Two approaches for analysing the retrieved results to find the main categories, is to consider the categories with the highest *normalised relevance* or top *hit count* as the main categories. The normalised relevance is found by taking the total relevance score within each category and dividing this by the number of hits within the same category. The hit count is simply the number of hits in each category. That is, the frequency of each category represented in the result set. The search is focused towards the main category/categories, reducing possible noise of hits belonging to other categories that are considered less relevant to the query.

  Both normalised relevance and hit count can be computed at query time or pre-computed from harvested data. Pre-computation yields better time performance. On the other hand, computation at query time ensures that a result set to analyse exists for every query.

- *Click-through logs.* To find the main categories of different queries, *click-through logs* can be used. Click-through data consist of the query, the ranking presented to the user, and the set of links the user clicked on. Click-through logs may give precise results as the logs are likely to include information based on users' opinions, such as the relevant document(s) to the specific search.

7.1.2 Discussion

Focusing the search towards main categories may have positive effects in that it is likely to reduce the amount of noise in the result sets for different queries. However, the approach has one big pitfall. If the search is focused towards categories that are not relevant to the query, this is likely to result in a worsened perceived relevance for the user. This can easily be illustrated by an example. In Table 7.1 we present the top-10 result set for the query `madonna artist`[1]. Table 7.2 shows the hit counts and normalised relevance for each category. As shows from the two tables, `Home & Garden` and `Toys & Baby` have the top hit count, and `Clothing` and `Books` have the highest normalised relevance. However, the query `madonna artist`, indicate that the user is interested in the music by the artist *Madonna*. Hence, the main category for this query should be *Music*. By using highest normalised relevance or top hit count, we see that the search will be wrongly focused for both approaches. Note that the user also could be interested in the painting Madonna. In this case, the *Home & Garden* category found by top hit count is the correct main category.

Table 7.1: Finding Main Categories for the Query `madonna artist`

Rank	Item	Category
1	Various Artists - The Electronic Tribute To Madonna	Music
2	Art Poster Print - Black Madonna Watercolor (Gic)	Home & Garden
3	Tribute To Madonna: Virgin Voices / Various	Music
4	Art Poster Print - Cowper Madonna	Toys & Baby
5	Art Poster Print - Soft Madonna	Toys & Baby
6	The Electronic Tribute To Madonna (Various Artists)	Music
7	Raffaello Sanzio Canvas Art Madonna On Grass	Home & Garden
8	Cavallucci, Antonio Prints - Madonna	Home & Garden
9	Madonna	Home & Garden
10	Madonna of Port Lligat	Toys & Baby

Table 7.2: Categories, Hit-Count, and Normalised Relevance for the Query `madonna artist`

Category	Count	NormRel
Home & Garden	404	565
Toys & Baby	385	404
Music	311	691
DVD & Video	29	404
Books	16	696
Electronics	11	498
Clothing	4	878

Considering only the top-k when using the different approaches to focus the search, could give better results as to improving the relevance of the result set. The reason for this is that the top-k results are more likely to be more relevant to the query.

[1]The results used in the table were retrieved when performing a search for `madonna artist` at `http://shopping.yahoo.com`, 19.05.2006.

We discarded this idea, even though the top-k approach could be a simple approach for improving perceived relevance. One reason was that we found the approach to have too little academic depth in order to make it constitute an entire thesis. The approach could also confuse the user if a search was focused wrong. Hence, some additional information in the graphical user interface could prove helpful with such an approach. The information should include a short motivation for the retrieved results. Using click-through logs to focus the search towards the main categories relevant to a query, could turn out to be a good solution. But since we did not have such click-through logs to our disposal during the period when we worked with this thesis, the approach was not investigated any further.

7.2 Semantic Relations between Books, Music and Videos

In this section, we present the approach of finding semantic relations between retrieved documents for different queries. First, the approach is described, followed by its pros and cons.

7.2.1 Approach

Finding semantic relations between books, music, and videos, as described in Section 2.3.3, could be approached in several ways. We initially investigated the use of text mining techniques. The idea was to look for certain words occurring in specified XML fields. Examples of such words could be `[Soundtrack]` or `O.S.T.` (Original SoundTrack) in the `title` field, or `by Original Soundtrack` in the `artist` field. Listing 7.1 shows two documents that are semantic related when performing a search for `the da vinci code`. The hits that are semantic related should be given a boost, or in some way have their relation displayed in the front-end after the search has been carried out.

```
<document type="music">
    <title>The Da Vinci Code / O.S.T.</title>
    <artist>Original Soundtrack</artist>
    <format>CD</format>
    <price>$11.99</price>
    <song>Dies Mercurii I Martius; The Paschal Spiral...</song>
</document>

<document type="video">
    <title>The Da Vinci Code</title>
    <director>Ron Howard</director>
    <format>DVD</format>
    <desc>A murder in the Louvre...</desc>
    <year>2006</year>
</document>
```

Listing 7.1: Semantic Related Documents

7.2.2 Discussion

The approach of finding semantic relations between books, music and videos could have a positive effect for the user as to perceived relevance. However, this approach has clear weaknesses since the

task of mining the text for semantic relations is very difficult. The reasons for this are variations in ways of saying that a document is a soundtrack, and also that oftentimes this information is simply missing in the `title` or other relevant fields. This problem could be avoided by adding an extra label to the documents, and thereby providing structure to potential semantic relations. Such a label could include information about related documents. In Listing 7.2, we present an example of how this could be realised. The first document is semantically related (`semrel`) to the second document with id 456 (`docid`), and vice versa. If a document is semantically related to more than one document, then this could be solved by using a comma notation, e.g `<semrel>123,981</semrel>`. It is also possible to make the semantic relations more specific, and include a label for each relation, e.g `<soundtracksemrel>123</soundtracksemrel>` and `<booksemrel>981</booksemrel>`.

```
<document type="music">
    <docid>123</docid>
    <title>The Da Vinci Code / O.S.T.</title>
    <artist>Original Soundtrack</artist>
    <format>CD</format>
    <price>$11.99</price>
    <song>Dies Mercurii I Martius; The Paschal Spiral...</song>
    <semrel>456</semrel>
</document>

<document type="video">
    <docid>456</docid>
    <title>The Da Vinci Code</title>
    <director>Ron Howard</director>
    <format>DVD</format>
    <desc>A murder in the Louvre...</desc>
    <year>2006</year>
    <semrel>123<semrel>
</document>
```

Listing 7.2: Labelled Semantic Related Documents

7.3 Boosting Hits with Match in Specific XML Fields

In this section we present the approach of boosting hits that match the query terms in specific XML fields. Specific fields are considered to be fields containing a known type of information. First, we describe the approach, and then its pros and cons are provided.

7.3.1 Approach

The approach of boosting hits with match in specific XML fields was as follows; the more specific an XML field was considered, the more the hit should be boosted if one or more query terms occurred in that field. The motivation is that information in more specific fields is considered more descriptive of the documents. Examples of fields that are considered more specific are:

- `author` – The author of a book.
- `artist` – The artist of a music album.
- `actor` – The actor of a movie.
- `director` – The director of a movie.

Fields that are considered to be less specific are:

- `title` – The title describing the product.
- `desc` – A textual description of the product.

Consider the query `madonna`. In Listing 7.3, two of the retrieved documents relevant to the query are presented. Both of these documents will be retrieved. The second document (*Confessions On A Dance Floor*) will however be boosted and ranked higher than the first document, since the query matches the content in the specific field `artist`.

```
<document type="book">
    <title>Madonna: The Biography</title>
    <author>Robert Matthew-Walker</author>
    <format>Paperback</format>
    <price>$9.74</price>
</document>

<document type="music">
    <title>Confessions On A Dance Floor</title>
    <artist>Madonna</artist>
    <format>CD</format>
    <price>$7.98</price>
    <songs>Hung Up; Get Together; Sorry</songs>
</document>
```

Listing 7.3: Two Documents with the Term `madonna`

7.3.2 Discussion

The approach of boosting hits with match in specific XML fields could help improve perceived relevance for several queries, but is to a great extent already supported in Vespa today. However, this approach led us in the direction towards the idea we realised in the Searcher plug-in. In brief, our solution analyses the query and focuses the search towards specific parts of the XML tree. The solution will be more closely elaborated in the following chapter.

Chapter 8
The Searcher Plug-in

"If Edison had a needle to find in a haystack, he would proceed at once with the diligence of the bee to examine straw after straw until he found the object of his search...
I was a sorry witness of such doings, knowing that a little theory and calculation would have saved him ninety per cent of his labor."

- Nikola Tesla

This chapter describes the Searcher plug-in, `XSearcher`, which we developed in order to test some of our proposed ideas. Section 8.1 describes its functionality and gives a demonstration of the motivation behind the improved relevance of the solution. Next, section 8.2 focuses on the value our solution adds beyond existing functionality in Vespa. Section 8.3 presents the technical details concerning the implementation. Section 8.4 outlines how the implementation can be generalised to other domains. Finally, section 8.5 elaborates on if the solution will scale to more visited Yahoo! search verticals.

8.1 How the Searcher works

As seen in Chapter 5, there have been a lot of proposed solutions regarding an XML query language. In the case of structured queries, these require too much knowledge about the document structure and a complex query language. The keyword-only approaches do not allow for knowledge about the structure at all. Some research exploit simple query languages that may include structure constraints [12, 16, 41]. We will investigate this approach further.

Our method, in its most generic form, treats a query term as a context term, i.e. terms that describe the desired context, if the term matches the name of an XML label in the document structure. In that case, the search is focused towards the XML subtree having the relevant label as the root node. For simplicity, and due to the fact that the Yahoo! Shopping users only submit 2.3 query terms on average[1], we decided to consider maximum one context term per query. After studying the query logs obtained from Yahoo! Shopping, it seems reasonable to consider the first or the last term as a possible context term. In case both these terms are context terms, the last term is preferred. This choice was done based on statistics from the query logs. Some examples will help clarify the approach. Table 8.1 presents several queries along with an explanation of what would be retrieved from the XML tree used in our implementation shown in Figure 8.1.

Our implementation is less generic than the approach described above, only considering XML labels specified in advance as context terms. Particularly, we are not considering the `title` and

[1] Calculated from Yahoo! Shopping query logs.

Table 8.1: Search Approach

Query	Search Approach
`jackson director`	A search for the term `jackson` in the `director` field.
`jackson video`	A search for occurrences of `jackson` in the subtree below the `video` node, i.e. the `actor`, `director`, `title`, and `desc` fields.
`jackson`	If no context terms are found in the query, the whole document tree is used as search context.
`the director movie`	Returns *movies* about *The director*. The fact that only the last query term is considered to be a context term, makes it possible to retrieve results with the title *The director*.

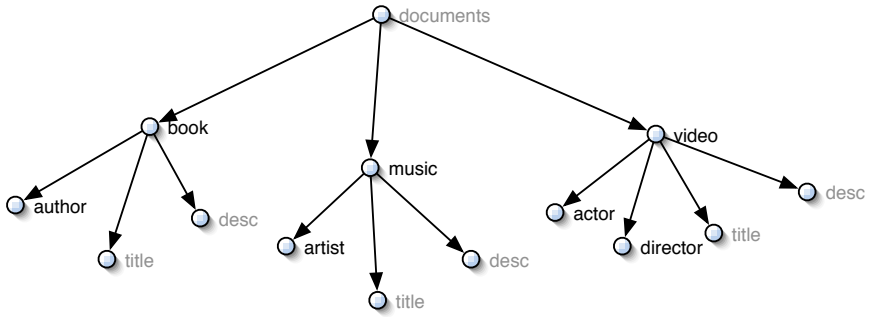

Figure 8.1: XML Tree for the Implemented Solution

`desc` fields due to the ambiguity of these.

Our solution has two premises:

- *A reasonable XML labelling.* According to [28], less than 20% of people choose the same term for a single well-known object. The article also finds that word usage tends to follow Zipf's distribution. Zipf's distribution states that a few words are used very frequently, the vast majority only rarely [4]. The system may thus be made more robust by including an ontology[2], such that different terms may be mapped to the same label. Our implementation makes use of a simple ontology, performing the mappings in Table 8.2, in addition to plural to singular mappings. The labelling scheme of the documents to be searched in is one of the most vital issues regarding our solution, and will be further explored in Chapter 12.
- *That the users submit queries containing context queries.* This requirement deals with training of the users. Among more than one million random user queries submitted at Yahoo! Shopping, 0.67% contained one of our context words. In 2.00% of the queries, the user had chosen the book, music, or video category in the department drop-down box. However, we believe that in order to improve search, a minimal amount of information has to be submitted. The users can be trained to use context words in their queries, by for instance getting presented suggestions of such queries in the front-end. A similar approach is in use at Yahoo! Shopping today, as *Also try ...* suggestions based on similar queries submitted by other users. Also, the users are likely to continue to use context words if they experience good results by utilising them when searching.

In case the search is performed at an intermediate node, the results from each leaf node have to be aggregated to obtain the final result. The results are merged one level at a time, i.e. if the whole document tree is used as the search context, the subresults for book, music, and video are obtained first and then merged to a final result set. We will evaluate two aggregation functions,

[2] A controlled vocabulary that describes objects and the relations between them in a given domain.

Table 8.2: Ontology

Alias	Concept
dvd	video
movie	video
album	music

namely *average* and *max*.

8.2 Value added beyond Existing Functionality in Vespa

Our solution extends the existing functionality in Vespa by means of two dimensions:

- *Context queries.* The first extension is about handling context queries. This is in fact possible in Vespa today by the notation `director:jackson`, which focuses the search towards the fields indexed to the `director` index. The search definition specifies in which indices the different fields should be searchable. Today, there is a separate index for each XML field, in addition to a full index covering most of the fields. The main drawback of today's method is that the user needs to know about both the query notation and the index structure. For example, the query `book:da vinci code` will not return any results, because there is no index named `book`.

  In addition, our solution solves the problem of treating context words in the query as keywords. Consider the query `brad pitt actor`. The user is not necessary interested in documents where all three query terms occur, but rather documents about the *actor* Brad Pitt. That is, the term `actor` should be treated as a context term used for focusing the search, greatly improving the retrieval.

  XSearch uses a simple query language where no structure knowledge is required. However, by submitting a reasonable amount of information describing the desired context, the user may perceive greatly increased relevance. We believe that in order to further improve search, and particularly in the direction of database retrieval, a minimum of information about the user's need has to be required. The importance of the specificity of the user query increases as the document corpus grows. More experienced web users due to increased age of the Web, also supports this choice of direction.

- *Hybrid search.* Vespa is developed from a platform used for web and enterprise search, and is thus heavily influenced by techniques used in this context. A typical characteristic of web search engines is that they return thousands or even millions of hits for each query. This contrasts the fact that users normally only view the few top ranked hits. Also, $\sim 20\%$[3] of the queries are futile, i.e. queries that returns zero answers. Our solution provides a generic and flexible framework for hybrid search, which also relies on methods originally used for data retrieval. Focusing of search, and different treatment of context terms as keyword terms, may provide as a solution to the mentioned challenges and thus improve the perceived relevance.

8.3 Implementation Details

A UML class diagram of the implemented solution is provided in Figure 8.2. *XSearch* is turned on and configured by setting parameters in the search URL[4]. An example query URL is

[3] From Yahoo! research.
[4] Uniform Resource Locator, the global address of documents on the World Wide Web.

`http://host/?query=madonna&xsearch&top-k=10&hits=50&aggrfunc=avg`. The various parameters are explained in Table 8.3. Table 8.4 describes the methods of `XSearcher` in detail. Figure 8.3 shows, somewhat simplified, how the query changes and the results are retrieved and merged for the query `madonna music`.

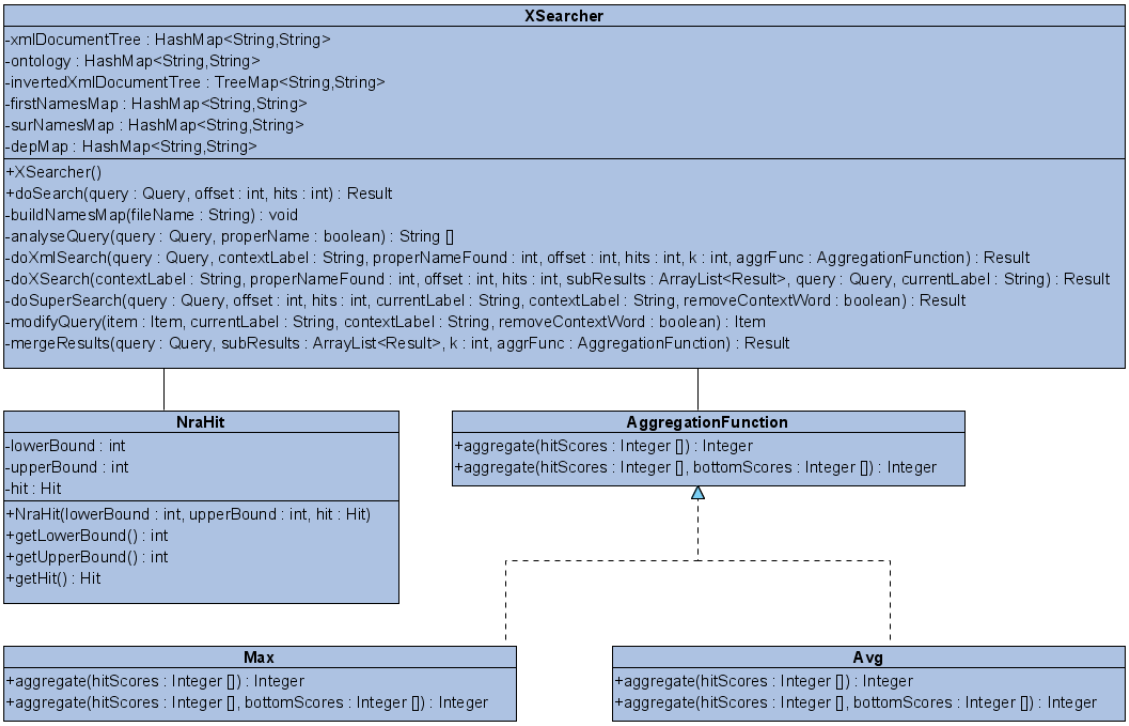

Figure 8.2: UML Class Diagram of the Implemented Solution

Table 8.3: Input Parameters

Method	Description
`xsearch`	Turns on *XSearch*.
`top-k`	The desired number of returned results.
`hits`	The number of hits requested in order to retrieve each subresult. Note that this number may have impact on the ranking of the merged result.
`aggrfunc`	Sets which aggregation function that should be used when merging the subresults. The parameter may be set to either `avg` or `max`.

The rest of this section discusses two challenges as to Vespa, which we were facing during the implementation. The first is concerned with inadequate labelling of the documents to be indexed. The Shopping vertical uses six types of document schemes. Three of these; *book*, *music*, and *video*, are labelled in an appropriate way. For example, the music document type contains separate fields for `artist` and `song`. The rest of the document types; *paidmerchant*, *freemerchant*, and *fastupdate*, have an inappropriate labelling scheme, only providing generic fields as `txt` and `int`. Book, music, and video only constitute about 8% of the total number of documents. In order to have the possibility to use the entire document base as a test collection, we implemented a workaround that changed a search from `actor:keywords` to `desc:keywords AND desc:actor` for these document types. This was also done for the fields `author`, `actor`, and `director`. Additionally, a department filter was added to focus the search towards the desired category. In the case of the keywords being proper names, the keywords were rewritten as a phrase. Listing 8.1 shows a typical example of the content of a `desc` field.

Table 8.4: The Methods in the `XSearcher` Class

Method	Description
`XSearcher()`	The constructor initiates the data structures. Three main data structures are used: • A hash map storing mappings from context terms to their Dewey numbers. • A hash map that acts as an ontology, storing mappings from synonyms or plural forms to the terms' base form. • A sorted hash map storing mappings from each Dewey number to its XML label. The first two mentioned hash maps are used during query analysis, in order to recognise context terms and express the desired search context by means of the Dewey number of its root node. The sorted hash map is used to traverse the part of the XML tree to be searched in, retrieving subresults, and merging these when appropriate.
`doSearch()`	This method is called each time a search is performed. It determines the values of the input parameters, and initiates query analysis and searching by subcalls to `analyseQuery()` and `doXmlSearch()`.
`analyseQuery()`	The method recognises context terms.
`doXmlSearch()`	Traverses the XML structure from the root context node found in `analyseQuery()`, performs a search by calling `doXSearch()` for each leaf node, and merges the subresults by calling `mergeResults()`.
`doXSearch()`	This method is a workaround, caused by inadequate labelling of parts of the document corpus. The problem will be further explained later in this section. For the moment, this method only forwards the search specifications to `doSuperSearch()`.
`doSuperSearch()`	Modifies the query by calling `modifyQuery()`, and sends it to the next Searcher in the chain.
`modifyQuery()`	The method traverses the query tree, and performs the following modifications: • Removal of a keyword recognised as a context term. • Addition of the context term as index labels, e.g. `music:madonna`. • Addition of a department filter, e.g. `department=books`. This is a part of the above mentioned workaround.
`mergeResults()`	Merges the subresults by performing the No Random Access algorithm from Section 5.2. The top-k hits are sorted according to their lower bound values. Objects of the `NraHit` class are kept in a hash map, sorted according to `NraHitOrderer`, a static, private class inside `XSearcher`, which first sorts by lower bound, then by upper bound. The method uses the `AggregationFunction` as a common interface to the specific aggregation functions.

```
1 <desc>Visuals highlight this creature-fest based on Oscar-winning director Peter Jackson's
2 remake of the classic movie. Captured in the wilds of Skull Island.</desc>
```

Listing 8.1: The Content of a `desc` Field

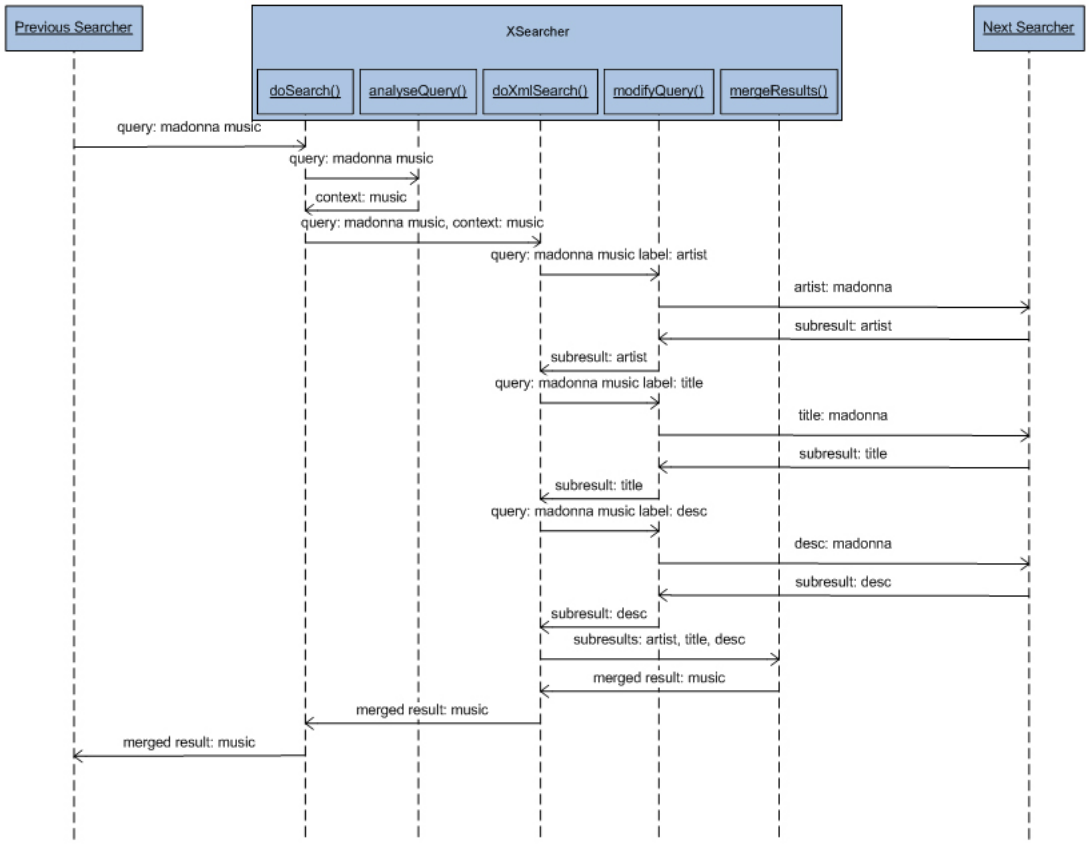

Figure 8.3: Message Diagram for the Query `madonna music`

The second challenge is regarding aggregation of the merged results. Table 8.5 shows the titles from the sorted lists returned from searching the `director` and `desc` nodes for the query `jackson video`. The relevance returned for each hit is shown in parentheses. Note that items with similar titles in the same list, represent similar items from different merchants. The max function works more like a simple merge of the lists, returning one of the items having relevance equal to 1000 as the top item. Intuitively, the max function should perform well, as hits returned from searching the `director` node would probably be more relevant and thus be assigned higher relevance. Unfortunately, each returned ranking from Vespa is normalised so that the relevance values range from 0 to 1000, and then balanced according to the field's rank-boost value. Thus, even if the hits returned from the `desc` node are less relevant, the relevance values will be about as high as the hits returned from the `director` and `actor` nodes. Due to the challenges related to the max function, we decided to also explore the average function. The average function rewards hits appearing in several lists, returning "Pulp Fiction (UMD Mini For PSP)" as the top item (shown in *italic style* in the table). Redundant data in the XML fields is necessary for the average function to work. The `desc` node often duplicates the information from the other nodes. If redundant information across XML fields is desired, is however a open question.

Note that the problems regarding aggregation makes it impossible to use the attractive median rank aggregation algorithm from Chapter 5 since there is no guarantee that the same items will be found in more than half of the lists.

Table 8.5: Sorted Lists to be Aggregated

Hit Nr	`director:jackson`	`desc:jackson`
1	Snow White and the Seven Dwarfs (1000))	Peter Jackson's King Kong: The Official Game of the Movie (PSP) (894)
2	XXX: Vin Diesel - DVD (New) (1000)	Peter Jackson's King Kong: The Official Game of the Movie (PSP) (894)
3	The Frighteners (1000)	Peter Jackson's King Kong: The Official Game of the Movie (PSP) (894)
4	Stalked Movie (1000)	GOSSIP (894)
5	Chattahoochee (1000)	UMD Movies (various vendors) Cursed PSP Movie (894)
6	The Lord of the Rings: The Motion Picture Trilogy (1000)	STAR WARS - EP. 6 REVENGE OF THE SITH WALL CLOCK (894)
7	The Oscar odds (978)	Wildstyle with Wil Power Break Dance DVD (894)
8	The Lord of the Rings - The Motion Picture Trilogy (Special Extended DVD Edition (978)	Journey - Warren Miller DVD (894)
9	XXX: State Of The Union (978)	*Pulp Fiction (UMD Mini For PSP) (894)*
10	Basic (978)	Awful Green Things From Space (Revised Edition) (894)
11	Snow White and the Seven Dwarfs (Disney Special Platinum Edition) Movie (978)	In Search of The Perfect Mountain, Tour Edition - VHS (894)
12	Chattahoochee Movie (978)	COOL HAND LUKE (894)
13	*Pulp Fiction (UMD Mini For PSP) (957)*	COOL HAND LUKE (894)
14	Pulp Fiction (UMD Mini For PSP) (957)	LORD OF THE RINGS - RETURN OF KING (894)
15	S.W.A.T. / XXX (Full Screen Editions) (DVD) (957)	1989 Baseball All Star Game Anaheim CA 7/11/89 DVD (894)

8.4 Generalisation of the Implementation

There are several modifications to Vespa that should be done in order to extend our solution to other domains. In this section, we will outline how the system could be generalised to not only include *books*, *music*, and *videos*, but also other shopping items.

The first step is to apply a proper labelling to the XML documents which is the basis for a proper index structure. Consider the domain of *electronics*. A subset of an XML tree based on the category structure currently in use at Yahoo! Shopping is shown in Figure 8.4.

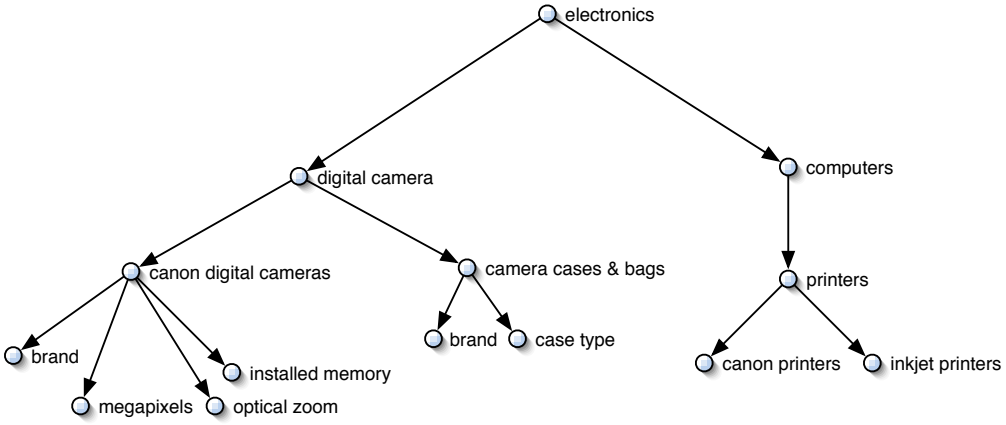

Figure 8.4: XML Tree for the Electronics Domain

The category structure is maintained as a list of category ids in the `category` field. The list contains an id for each category the item belongs to, as shown in Listing 8.2. Consider the query `canon digital camera`. The query terms `digital camera` should be treated as context terms, thus focusing the search towards the `digital camera` node. This can be done by performing a mapping from context term to category id, and filtering by category id. Alternatively, the XML structure may be modified, in order to capture the category hierarchy. An example of how this could be done is presented in Listing 8.3.

```
<document>
    ...
    <category>electronics digital camera</category>
    ...
</document>
```

Listing 8.2: The Content of the `category` field

```
<document>
    <electronics>
        <digital_camera>
            ...
        </digital_camera>
    </electronics>
</document>
```

Listing 8.3: An XML Structure capturing the Category Hierarchy

The `category` field is only one example of a field that should be given special attention. Consider

camera properties like *brand*, *mega pixels*, and *zoom*. Today, this information is contained in generic fields like `txt`, `int`, and `str`, as shown in Listing 8.4. Similar to the category case, a mapping from possible context term(s), e.g. *brand*, to the actual field containing this information, needs to be performed. The solution may be made cleaner by always labelling the fields with the corresponding context term(s), the same way as in the category example. The resulting XML document is presented in Listing 8.5.

```
<document>
    ...
    <str1>Canon</str1>
    <str2>PowerShot</str2>
    <txt1>Secure Digital Card</txt1>
    <int1>7100</int1>
    ...
</document>
```

Listing 8.4: The Content of the Generic Fields

```
<document>
    ...
    <brand>Canon</brand>
    <product_line>PowerShot</product_line>
    <memory_card>Secure Digital Card</memory_card>
    <mega_pixels>7100</mega_pixels>
    ...
</document>
```

Listing 8.5: An XML Structure capturing the Context of the Fields

The XML document representation needed for query analysis, may be built and maintained dynamically during indexing, either by treating all fields as searchable XML nodes, or by manually deciding which fields that should be searchable and which that only should be included for presentation purposes or such, as suggested in [42]. Also, indices need to be built for each field that should be searchable.

An important issue is how to decide which context term(s) that should be chosen to label each field. This challenge is closely related to analysis of the query, in order to decide which terms that are context terms and which that are not. We will elaborate more on these problems in Chapter 12.

8.5 Scaling to other Verticals

A last issue that should be considered is to which degree the solution will scale to more visited Yahoo! verticals. The answer depends on the running time of our solution.

The time consuming part of the implementation is the traversal of the XML tree with retrieval and merging of subresults. The traversal of the XML tree and merging of subresults depend on the number of nodes to be traversed, and the number of hits to be merged. This part is thus independent of the index size. The retrieval of subresults use the existing functionality of Vespa and is outside the control of our implementation.

The worst case running time for the traversal of the XML tree and merging of subresults is

$$O(bf, levels, hits) = \frac{bf^{levels-1} - 1}{bf - 1} \times hits \times \sum_{n=1}^{hits} bf \times n \times log(bf \times n)$$

where bf is the average branching factor, $levels$ are the number of levels in the XML tree, and $hits$ are the number of hits to be merged. $\frac{bf^{levels-1}-1}{bf-1}$ is the number of nodes in the XML tree. The $hits$ variable depends on how many hits that are retrieved for each subresult, and characteristics such as the rank score distribution of these lists, which decide when the aggregation algorithm breaks. $\sum_{n=1}^{hits} bf \times n \times log(bf \times n)$ is the cost of sorting the hits found so far.

Table 8.6 shows the average latency ratios between *XSearch* and the standard search. The first column presents the ratio for non-context queries. The second column contains the ratio for context queries where the context term is either *book*, *music*, or *video*. The last column shows the ratio for context queries where the context term is *author*, *artist*, *actor*, or *director*. The queries are the same as we used in the evaluation, and will be described in Chapter 9. Three runs were performed, because latency varies depending on system load.

The results show that the ratio decreases when context words deeper down in the XML tree are included in the query. This is reasonable as a more narrow search requires fewer subresults to retrieve and merge. The running time confirms these findings.

An interesting observation is that the context words *author*, *artist*, *actor*, and *director*, give less latency than the original search. Consider the query `jackson director`. The standard approach will obtain lists for both query terms and merge these. *XSearch* reduces the query to `jackson` and perform the search in the `director` index. Thus, both less query terms and a smaller index contribute to less latency.

Table 8.6: Latency Experiment

Query Type	**Ratio**
Non-Context Queries	10.5
Book, Music, Video	2.7
Author, Artist, Actor, Director	0.7

As elaborated in this section, the question if our solution will scale to more visited verticals depends on the complexity of the domain, i.e. the size of the XML tree, rather than the number of queries and index size.

Chapter 9
Evaluation Principles

"All life is an experiment. The more experiments you make the better."

– Ralph Waldo Emerson

In order to determine if a search system is desirable, it is necessary to evaluate the system and make comparative assessments. This chapter presents the evaluation principles that will be used as a basis for the evaluation of our solution. Section 9.1 describes how the quality of the solution is being measured. In Section 9.2, we elaborate what has been used as a reference collection. Validity assessments regarding the evaluation principles are presented in Section 9.3.

9.1 Effectiveness Measure

To evaluate the quality of our solution, we used $P@k$ returned documents as the effectiveness measure. As described in Chapter 4, $P@k$ is defined as the proportion of relevant documents in the top k results (T_k). The formula for $P@k$ is given below:

$$P@k = T_k/k$$

The main reason that we used $P@k$ as effectiveness measure is that it is a well-defined way to measure and compare search applications [4]. Also, users are usually more interested in the precision of the results displayed in the first result page, rather than the overall precision [38]. According to [4], the technique is considered a useful parameter for observing the behaviour of an algorithm for each individual query in an experiment. Additionally, it is possible to compute an average $P@k$ figure over all queries.

The rest of this section provides an example of calculating $P@k$. Table 9.1 shows three different top-*10* results for the query `q`; the reference set, the list retrieved for the standard search, and the list retrieved when using *XSearch*. Finally, Table 9.2 presents the $P@k$ for the k top documents, both for the standard search and when using *XSearch*.

9.2 Reference Collection

To make comparative assessments, some kind of reference is needed to benchmark against. For testing our solution, we made a reference collection constituting of the relevant documents for different queries. The collection was used as a benchmark, or optimal set. Since our focus is on the domain *books*, *music*, and *videos*, we selected queries where the user at query-time had focused the search within one of these categories, i.e. selected one of these categories in the department drop-down box. Of a total of 22427 queries related to our domain, 150 random queries were

Table 9.1: Three Top-*10* Lists for the Query `q`

Reference set		Standard Search		Using XSearch	
Rank	**Document**	**Rank**	**Document**	**Rank**	**Document**
1	d_1	1	d_2	1	d_1
2	d_2	2	d_1	2	d_2
3	d_3	3	d_{13}	3	d_3
4	d_4	4	d_{25}	4	d_4
5	d_5	5	d_{36}	5	d_6
6	d_6	6	d_5	6	d_5
7	d_7	7	d_{17}	7	d_{17}
8	d_8	8	d_{12}	8	d_{12}
9	d_9	9	d_{14}	9	d_{14}
10	d_{10}	10	d_{33}	10	d_{23}

Table 9.2: P@k Top Documents for the Query `q`

	Standard	**XSearch**
P@1	0 %	100 %
P@5	40 %	80 %
P@10	30 %	60 %

extracted from a Yahoo! Shopping query log to be used as testing queries. 17 of these queries were discarded, as they retrieved no results. Of the remaining queries, 42 queries were book queries, 45 were music queries, and the last 46 queries were video queries. Appendix A lists all the queries used for evaluating our solution.

Below we present some query examples. For the first query, `the da vinci code`, we see that `department=books` is included in the query-string. This means that the user has focused the search, by selecting that he only wants to search within the book department. For the two other queries, the search is focused towards the *videos* and *music* departments.

```
?query=the+da+vinci+code+department=books
?query=brad+pitt+department=videos
?query=madonna+department=music
```

As presented in the previous section, we decided to use Precision-at-k (P@k) to evaluate the quality of our solution. This measure does not take the ranking of documents into account, but rather how many of the top-k retrieved documents that are relevant. As to our reference collection, we first tried to manually remove irrelevant hits. We refrained from this because of the following reasons:

- *Underlying document structure.* Irrelevant hits are often retrieved as a consequence of inadequate labelled documents. An example of inadequate labelling is forum posts where a book has been discussed. Such forum posts should not be labelled as a book, and in fact never be indexed. Further, we experienced inappropriate labelling for DVDs, as both computer games, music videos, and videos were in the same category. These documents could profit by being differentiated in some way, making them easier to separate. The inadequate labelled documents often causes undesirable hits, but it can be argued that it is "algorithmically correct" that they are retrieved. Therefore, we chose not to remove these hits.

  Another group of hits that caused trouble is duplicates, i.e. identical products from different merchants. It is not in our interest to distinguish different merchants. Thus, similar items should be treated as the same product. However, since there is no product id in the documents, it is impossible to automatically decide if two products actually are identical.

- *Subjectivity.* In addition, manually browsing through retrieved results constituting of hundreds, or even thousands of hits, is a time consuming job. To obtain a best possible reference collection, it is a great advantage having more than two persons to perform this task. This to ensure that the subjectivity of individuals is reduced. Making a reference collection for various queries is often a task that is outsourced, because of its high need of resources.

Our approach for making the reference collection was as follows:

- Use the extracted queries for our domain.
- Modify queries containing proper names to phrase-queries, and thereby removing as much noise as possible.
- Add department filtering (`department`) to the queries in order to focus the search within the relevant category.
- Store the top-100 retrieved results for the various queries, and define this as the optimal result set. An exception is experiment 3 described in Chapter 10, where the optimal result set contains all the retrieved results.

Note that a positive aspect with this approach, is the possibility of testing our solution for far more queries. If we had chosen to manually remove noisy hits, the consequence would have been a much lower statistical basis for our results. On the other hand, the reference set is somewhat specially adapted to the solution to be tested, which should be taken into consideration when analysing the results.

9.3 Validity Assessments

Possible threats to the validity of the experiment results may either be accepted, or tried to be avoided. This section presents the relevant threats according to our evaluation principles. The threats are categorised according to the four types of threats to the validity presented in Chapter 3.

A possible threat to the conclusion validity is low statistical power. 133 test queries should be enough to give a good hint about the effect of our solution. However, the queries are chosen to fit our domain and is thus not completely random. This restriction was necessary because it was impossible to implement a general solution due to limitations in Vespa. A common method for evaluating search systems is to first measure average change for a large set of queries, and then inspect some hundreds of those that got their result set changed more closely.

Second, the subjectivity of measures influences the conclusion validity. In order to define a reference set of optimal results to a query, we first tried to manually decide which documents that should be considered relevant and not. Since we were only two persons in much the same circumstances of life, this approach would have been a large threat, and was one of the reasons we rejected it. Generally, subjectivity is a threat to every evaluation of search, as the concept of relevancy is a subjective matter. The final reference set was made using objectively criteria, and is thus easy to reproduce, assumed that a similar system, like the one we used, is available.

A final threat to the conclusion validity is the fishing threat, i.e. that the researchers may influence the result by looking for a specific outcome. This threat is handled by treating the independent variables, i.e. which queries that should be used as test queries, and how the reference set was defined, as random and objectively as possible.

A possible threat to the construction validity is that the experiment construction does not reflect what is actually going to be evaluated. This threat is handled by choosing an experiment setup that is based on common practice in the field of information retrieval.

Last, the extent of the experiments is a threat to the external validity. The number of test queries and choice of reference collection, are due to limited resources. Also, a full system implementation should be available before too much effort are put into testing.

Chapter 10
Experiments and Results

"There is no such thing as a failed experiment, only experiments with unexpected outcomes."

– Richard Buckminster Fuller

In this chapter, we describe the experiments performed in order to test our Searcher plug-in, along with their results. As stated in the thesis goals, the thesis aims to improve the users' perceived relevance for the Yahoo! Shopping vertical. The experiments will shed light on this issue. The main motivation of the experiments described in this chapter is therefore to investigate if our solution yields better results than the standard search.

First, Section 10.1 describes the experiment setup. Next, Section 10.2 provides an overview of the different experiments. Sections 10.3 to 10.6 elaborate the different experiments we conducted. Finally, Section 10.7 gives a summary of the results.

10.1 Experiment Setup

This section includes the setup of the experiments conducted in this thesis. In the following, we describe the system, the indices, and the configuration used when performing the experiments.

The experiments were run at a system using 52 Dell Poweredge 2650 boxes with the following specifications:

- 2×2,8GHz processor (Intel Xeon)
- 2GB main memory (DIMM DDR)
- 336GB disk space

During our work, we used two different indices to test our system. The first index was mainly used for testing the different modules of the system as they were implemented. The index, consisting of 3 million documents, was a subset of the real index used at Yahoo! Shopping. The documents were fetched from Yahoo! Shopping, using 10000 random queries from a Shopping query log. The QRS does not return all elements of the XML documents. Missing static rank fields, and dissimilar index size causing different term statistics, made it impossible to reproduce the same results as Yahoo! Shopping.

We were given the opportunity to test our solution against a mirror of the complete Yahoo! Shopping index at the end of the project. The index included $\sim$ 60 millions documents, similar to the real index as of ultimo February, 2006. Even though the indices at that time were similar, we experienced that getting identical results as Yahoo! Shopping was an impossible task. The main reason for this is due to external systems on the top of Vespa at http://shopping.yahoo.com, providing reordering of the results according to business objectives. In addition, we did not use all

the QRS plug-ins that Yahoo! Shopping uses today. This was due to that these plug-ins handle grouping tasks, an aspect not considered in this thesis.

In spite of all the challenges mentioned above, the configuration of the system and the index was as identical to the Yahoo! Shopping setup as possible. The index is a large test collection in the context of academic work, and although business objectives are important in the real world, it is of minor interest in academic work. Even though it was impossible to perform the tests on a system 100% identical to the "real world", we ensured that we had a reference point, as good as possible, to the real system.

10.2 Experiment Overview

This section presents an overview of the experiments, that is, which experiments that were chosen, and why they were chosen. The different parameters used in the different experiments were explained in Table 8.3. The combinations of the different parameters are infinite. In order to curtail the number of combinations, we first performed an experiment to find the aggregation function (`aggrfunc`) and the number of hits (`hits`) that gave the best results. Further, it was desirable to see the effect of our solution, and compare the results with the standard search. In addition, as the main task with our solution is to remove noisy hits, this had to be tested. Finally, we wanted to ensure that the differences between the standard search and our solution should be minimal when context words were not used in the queries. This was considered to be an important aspect in order to deploy our system.

In the light of the above mentioned requirements, we defined the following experiments:

- **Experiment 1** – Tuning of aggregation function and hits parameter.
- **Experiment 2** – Testing our solution.
- **Experiment 3** – Elimination of noise.
- **Experiment 4** – Searching without context words.

The experiments will be further elaborated in the following sections.

10.3 Experiment 1 – Tuning of Aggregation Function and Hits Parameter

In this section, the first experiment is described. The purpose of the experiment was to find the optimal aggregation function and hits parameter to be used in the subsequent experiments.

10.3.1 Approach

In the first experiment, we varied with the two aggregation functions `max` and `avg`. For the aggregation function `max`, only one run was performed with the hits parameter set to 10, since the top-k results returned from the `max` function will not change for $hits >= k$. For the aggregation function `avg`, the hits parameter was set to 10, 20, and 50. All 133 test queries were used in the experiment. The other parameter values used in the experiment are presented in Table 10.1.

Table 10.1: Parameters used in Experiment 1

Parameter	Value
top-k	10
xsearch	true
hits	10|20|50
aggrfunc	avg|max

10.3.2 Results and Discussion

In Table 10.2, test results for the first experiment are presented. The table shows the $P@k$ values for the book, music, and video queries. The last column shows the $P@k$ values for all 133 queries.

The results show that the differences between the two aggregation functions are minimal. Also, the `avg` function does not perform better as the hits parameter grows. This conclusion was surprising, because we had expected that a higher hits parameter would yield better results, as the aggregation algorithm would take more hits from each subresult into account. A likely explanation is that duplication of information in XML fields described in Section 8.3, does not necessarily denote a better hit.

It is worth noting that irregularities in the rankings possibly affect the experiments to a high degree. Due to the configuration of the Shopping vertical, many hits are assigned a high and equal relevance value. The ranking of hits with equal relevance value is affected by the order they are returned from the back-end. The effect may be that a large part of the top-k results changes when the same query is submitted twice, especially when k is small. This in turn affects the sorting in the aggregation algorithm, as the sorting of hits with equal lower and upper bound is determined by the sequence they are discovered.

The results show that the $P@k$ values are somewhat higher for the aggregation function `avg` and the hits parameter set to 10. We will therefore use these settings in subsequent experiments.

Table 10.2: Experiment 1 - Tuning of Aggregation Function and Hits Parameter

Aggregation Function	Hits	Books	Music	Video	All
max	10	77.1%	82.7%	88.8%	82.9%
avg	10	77.1%	83.0%	88.8%	83.0%
avg	20	74.3%	75.9%	92.3%	80.8%
avg	50	76.2%	77.0%	90.0%	81.1%

10.4 Experiment 2 – Testing our Solution

The second experiment consisted in testing our solution. The purpose of the experiment was to test the effect of our solution when utilising context words, and compare it with the standard search.

10.4.1 Approach

In the second experiment, we varied with different top-k values, using $k = 1$, $k = 3$, and $k = 10$, to compute the $P@k$. We performed 3 runs with the standard approach, and 3 runs using *XSearch*.

Table 10.3: Parameters used in Experiment 2

Parameter	Value
`top-k`	1|3|10
`xsearch`	true|false
`hits`	10
`aggrfunc`	avg

Each run included the 133 test queries. The parameter values used in the experiment are shown in Table 10.3.

10.4.2 Results and Discussion

In Table 10.4, the results when using *XSearch* is presented. Table 10.5 shows the results from the standard approach. The results clearly indicate that the $P@k$ values are far better for our solution, compared with the standard approach. The percentual improvements when considering all queries, are 34.7% for $k = 1$, 29.2% for $k = 3$, and 25.7% for $k = 10$.

A contributing factor to the difference in the results is due to that the query terms are treated as an AND-query in the standard search. Given the query `the da vinci code video`, *XSearch* modifies the query to the following AND-query `the da vinci code` and focuses the search only retrieving videos of "The da vinci code". The standard approach does not modify the query. The context word, video, is included as an AND-term, resulting in poor retrieval quality. Also note that the reference collection described in Section 9.2, is somewhat adapted to favour this experiment.

Table 10.4: Experiment 2 - Using *XSearch*

Top-k	Books	Music	Video	All
1	78.6%	86.4%	90.7%	85.2%
3	77.0%	85.6%	89.1%	83.9%
10	77.1%	83.0%	88.8%	83.0%

Table 10.5: Experiment 2 - Using the Standard Approach

Top-k	Books	Music	Video	All
1	50.0%	36.4%	65.1%	50.5%
3	51.6%	41.7%	70.9%	54.7%
10	56.9%	43.7%	71.4%	57.3%

10.5 Experiment 3 – Elimination of Noise

The third experiment tested elimination of noise. The purpose of the experiment was to investigate to which degree *XSearch* was able to remove noisy hits from the retrieved results.

10.5.1 Approach

In order to test if *XSearch* eliminates noisy hits, we performed three different runs, calculating the $P@k$ for the 133 test queries. The value used for k for each query, was the number of hits in the optimal result set (*optreslength*). In the first run, we used *XSearch* and context queries. For the second run, we used standard search and swapped the context term with a department filter. The department filter focuses the search within a desired category, and is the similar approach to *XSearch* and context queries in Vespa today. In the third run, we used the standard approach without the department filter. Table **??** shows the parameters used in experiment 3.

Table 10.6: Parameters used in Experiment 3

Parameter	Value
`top-k`	optreslength
`xsearch`	true|false
`hits`	optreslength
`aggrfunc`	avg

10.5.2 Results and Discussion

In Table 10.7, the results from experiment 3 are presented. The results show that the $P@k$ values are far better for *XSearch*, compared with the two standard approaches. The percentual improvements when considering all queries, are 22.3% between the first two runs, and 39.8% between the first and the last run.

Table 10.7: Experiment 3 - Elimination of Noise

Search Approach	Books	Music	Video	All
`context-words&xsearch`	82.4%	90.8%	95.0%	89.4%
`standard+department`	59.3%	63.0%	79.1%	67.1%
`standard`	52.3%	51.3%	45.1%	49.6%

10.6 Experiment 4 – Searching without Context Words

The fourth, and final, experiment consisted in searching without context words. The purpose of the experiment was to ensure that the retrieved results of the standard search and *XSearch* should differ minimal when context words were not used in the queries. As earlier mentioned, this was an aspect that was considered important in order to deploy our system.

10.6.1 Approach

In this experiment, we investigated if *XSearch* had the same effect as the standard approach when context words were not utilised in the queries. We performed 3 runs with the standard approach, and then 3 runs using *XSearch*. Each run included the 133 test queries. The parameters used in the experiment are shown in Table 10.8.

Table 10.8: Parameters used in Experiment 4

Parameter	Value
top-*k*	10
xsearch	true|false
hits	10
aggrfunc	avg

10.6.2 Results and Discussion

In Table 10.9, the results when searching without context words are presented. The results show that the differences between the standard search and *XSearch* are minimal when context words are not utilised in the queries. This is promising for a later deployment of our solution. As seen in Section 8.5 however, these queries suffer from a higher latency.

Table 10.9: Experiment 4 - Searching without Context Words

Queries	Standard	XSearch
Books	48.33%	47.22%
Music	43.63%	43.87%
Video	45.79%	44.68%
All	45.92%	45.26%

10.7 Summary

In this chapter, we have described the experiment setup and elaborated the different experiments. The purposes of the experiments have been to find the best suited aggregation function and hits parameter, testing the performance of *XSearch* in the case of context queries, and to which degree noisy hits are eliminated. In addition, an experiment have been performed to ensure that the retrieved results of the standard approach and *XSearch* differed minimal when context words were not used in the queries.

In Section 2.3, some known challenges related to this thesis were described. The challenges included focusing search and thereby removing noisy hits from the retrieved results, and the treatment of context queries. Both of these problems relate to this thesis' goal of implementing a Searcher plug-in that should retrieve the results in a way that improves the user's perceived relevance. The results from the experiments 2 and 3, show that our solution to a great extent solve these challenges.

It should be noted that it would have been better to use an own test collection to tune the aggregation function and the hits parameter. This is because quality should not be measured from the same data as was used to tune the system. However, since the choice of aggregation function and hits parameter showed minor impact on the system, this have probably not influenced the results to a large degree.

Part IV

Conclusion and Further Work

Chapter 11
Conclusion

"A conclusion is the place where you get tired of thinking"

– Arthur Bloch

With billions of documents stored in today's web indices [29, 65], the problem of finding the relevant information is becoming more and more complex. The amount of digital information increases with 1% every week [59]. As the indices grow, relevance – the ability to find the needle in the haystack, rather than bury you in straw – has never been more important. Semantic search is by many referred to as the new era in search [5, 59]. Semantic search is characterised by the search engine being able to understand the meaning of both user queries and document content. The straightforward approach is to label the queries and documents with metadata, i.e. data describing the content, resulting in semi-structured data.

In this thesis, we have studied relevance techniques for the purpose of identifying an approach that improves the perceived relevance at the Yahoo! vertical search platform, Vespa. A Searcher plug-in has been implemented in Vespa for evaluating the suggested approach. The plug-in provides a generic and flexible framework for hybrid search, based on techniques proposed for XML retrieval:

- A simple query language, optionally including context queries, i.e. queries that include terms that describe the desired context [16, 33]. Keywords and context terms in a query is treated differently, using the context terms only for focusing the search.
- Enabling retrieval at different levels in the XML document tree, where term statistics are computed independently for each context [12, 27, 30, 42].
- An effective aggregation algorithm for merging subresults from various parts of the XML tree [26].

5 experiments have been performed in order to test our proposed solution. The results indicate that:

- A considerable improvement around 30% in retrieval performance is achieved for context queries. Much of the improvement is obtained by removing noisy hits from the result. Note that the solution also may reorder the results. The effect of this feature has however not been evaluated.
- The solution performs almost similar as the standard approach for non-context queries. The relative change is 0.66%.
- Latency is acceptable, reaching from about 10 times the standard than the standard latency for the most specific context queries. The difference in latency for various queries is due to that a more focused search requires fewer subresults to retrieve and merge.

It should be noted that the reference collection used for the evaluation is somewhat adapted to favour our solution.

The solution requires a reasonable labelling of the documents, in addition to training of the users in order to make them use context words in their queries. Even though research shows that

the average query length does not increase [34], we believe that in order to improve search, and in particular in the direction of database retrieval, a minimal amount of information has to be submitted. At the time that this is written, only 2.67% of the total amount of queries is affected by *XSearch*. This requires that the queries that already are focused towards the book, music, or video category by selecting one of these categories in the department drop-down box, are rewritten to fit our solution. The other queries will suffer from a higher latency. The latency depends on the complexity of the domain.

Most search engines today either return thousands of answers to a user query, or, in $\sim$ 20%[1] of the cases, none. Our solution may provide as a solution to these challenges and thus help to improve the perceived relevance. The preliminary experiment results are positive, but should be complemented with experiments based on a full system implementation.

[1] From Yahoo! research.

Chapter 12
Further Work

"The Alchemists, in their search of gold, discovered many things of greater value."

– A. Schopenhauer

This chapter presents some thoughts regarding future work. Mainly three issues are discussed, namely document labelling, ranking experimentation, and user interface guidance.

12.1 Document Labelling

As mentioned in Chapter 8, the implemented system includes several hacks caused by underlying restrictions. The most precarious issue is improper labelling of the XML documents. In order to provide semantic search, the label names need to capture the semantics of the documents' content. The ultimate question is how to choose which label name that best represents the meaning of the content, e.g. should a movie be labelled as *movie*, *DVD*, or *video*? We will in the following examine two different approaches to this problem.

[28] finds that *armchair naming*, i.e. the system designers personal favourite names, only succeeds in 10-20% of the cases. By collecting term usage from a sample of potential users, the most frequent word can be used as keyword. This method provides about two times better than armchair naming. Using several aliases may increase the success rate to 50-100%, depending on the domain. The authors argue that the problem should be viewed from the human-centred point of view. That is, the starting point should not be a set of system objects needing names, rather the process should be viewed as a set of user words needing system interpretations. This can be achieved either by collecting a number of terms for each object from a number of representative users, or to construct alias indices adaptively, on site, in use. Careful thought should be given to ambiguous terms, i.e. terms that may have more than more semantic interpretation. A solution may be to return the various choices to the user for further selection.

The second approach is presented in [5]. The labelling scheme technique proposed is to let anyone label anything, exactly the way they want it. On the long view, some items will be returned using any of the may possible aliases as a keyword. For example, movies will be labelled both as *movies*, *DVDs*, and *videos*.

In case of the Shopping vertical, the second approach is less suitable as they probably would prefer a uniform labelling scheme. A better solution would be to choose a labelling scheme, and put effort into building extensive alias indices.

12.2 Ranking Experimentation

The first step will be to choose a labelling scheme, and extend the `XSearcher` to the generic approach. This is necessary in order to obtain some more reliable test results.

Next, it would be interesting to experiment with ranking of the results. The choice of aggregation function and weighting of nodes are factors that should be explored. For example, consider the query `omen music`. As illustrated in Figure 12.1, the results consist of albums with *omen in the title* and albums from *the artist Omen*. Which results should be considered more important? Should for example hits from the `artist` node be considered more important than hits from the `title` node as the `artist` node is more specific than the `title` node? Or should hits from the `artist` node be considered less important since the `artist` node is treated as a context node and the query context word is *music*? Further, should soundtracks of the Omen movies be included in the results from the `title` node, or should the XML document tree be extended in order to separate albums and soundtracks as shown in Figure 12.2? As should be obvious from the last issue, the retrieval challenge is strongly connected with the labelling scheme chosen.

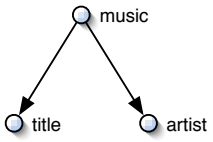

Figure 12.1: XML Tree for the Query `omen music`

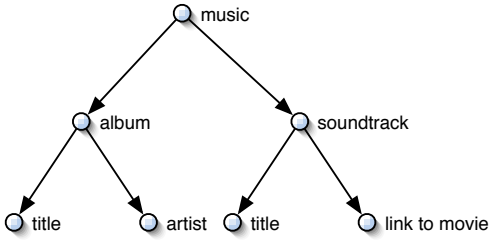

Figure 12.2: Extended XML Tree for the Query `omen music`

Another example of the retrieval problem is visualised in Figure 12.3. The figure shows the XML tree of the query `john denver sunshine on my shoulders`. In this case, "Sunshine On My Shoulders" is both an album title and a song title (occuring at different albums), being retrieved from different levels in the XML tree. Should the more specific song title or the less specific album title be considered as most relevant? As mentioned in Chapter 5, [14] proposes that the system in this case always should retrieve the most specific part of a document answering the query, known as the *structured document retrieval principle*. However, the Shopping vertical differs at this point, because the problem is not *which part* of the documents to retrieve, rather *which documents* to retrieve.

12.3 User Interface Guidance

Regardless of which approaches that are chosen both regarding document labelling and ranking of the results, it is important to be consistent. This is due to that the user always should experience the same processes happening when he performs similar actions. Various approaches for guiding the user through the user interface were presented in Chapter 5. The *Also try ...* functionality

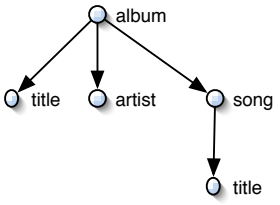

Figure 12.3: XML Tree for the Query `john denver sunshine on my shoulders`

used in Yahoo! Shopping today, is shown in Figure 12.4. A similar approach is currently in use at AllTheWeb Livesearch[1] and can be seen in Figure 12.5. The recommendations in both approaches are based on earlier user queries, but could easily be modified to mirror what can be retrieved in different contexts. A site probably taking this approach is the BitTorrent[2] source Seedler[3].

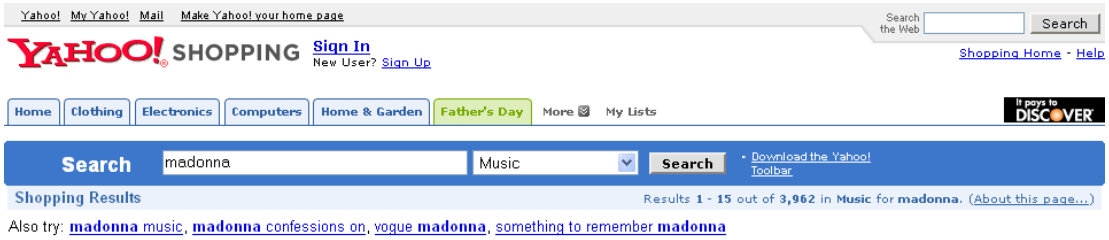

Figure 12.4: User Guidance on Yahoo! Shopping

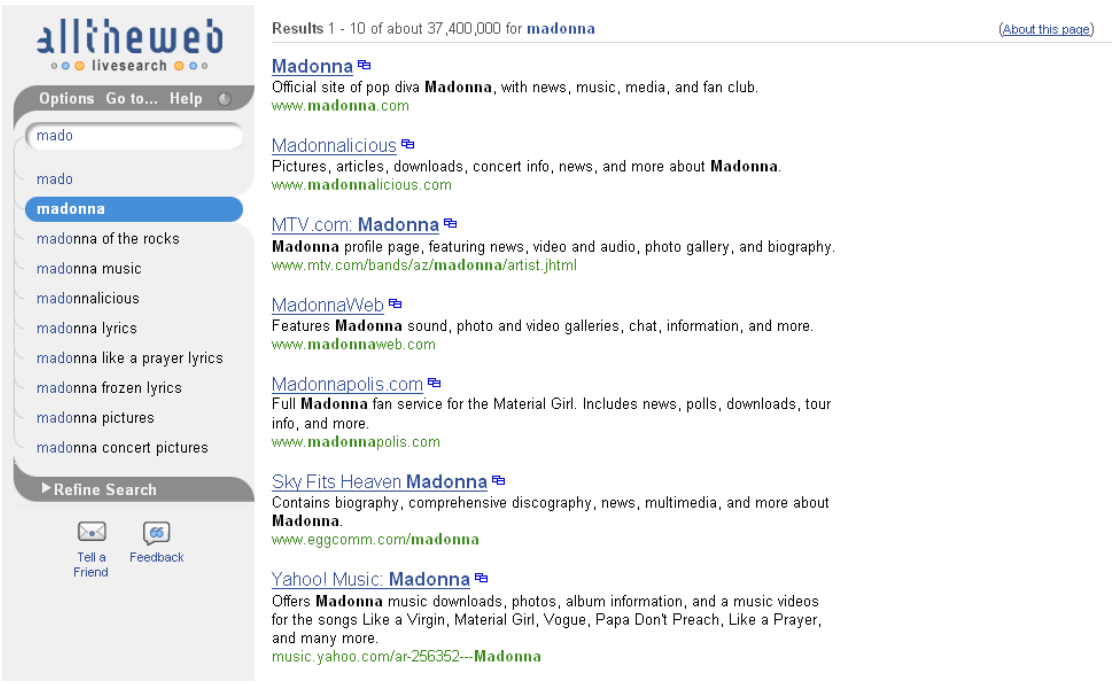

Figure 12.5: User Guidance on AllTheWeb Livesearch

[1] `http://livesearch.alltheweb.com/`
[2] BitTorrent is a peer-to-peer (P2P) file distribution tool.
[3] `http://seedler.org`

12.4 Query Analysis

In search, true success comes from understanding what the user's information need. [3] classifies query analysis along four dimensions:

- *Orthographic* – for example checking for typos and language.
- *Morphologic* – stemming/lemmatisation.
- *Syntatic* – for example entity or phrase extraction, anti-phrasing, and removing word-sense ambiguity.
- *Semantic* – applying general and specific thesauri and ontologies.

The last mentioned dimension, semantics, is highly related to the labelling scheme and should be explored in order to obtain additional improvement. This includes assessments of the rules used for analysis, i.e. that only the first or last query term should be considered as context terms, and the number of possible context terms. Also, in case keyword terms are recognised as proper names, this should be utilised in the search.

In addition, it is possible to apply domain specific linguistic analysis. As an example, consider the domain of books, music, and videos. Here, the XML elements `author`, `artist`, `actor`, and `director` always contain proper names. If a user is interested in books by the author Dan Brown, she could define the query `brown author`. For this query the search would be focused towards the relevant leaf node (author), since "Brown" is a proper name. In the opposite case, the search is focused towards the root node, just as in a standard search. This solves the problem of a focusing the search wrong due to context words occuring in the title of an item, as for instance the movie "The author".

Part V

Appendix

Appendix A
Test Queries

In this Appendix we present the 150 queries that were used when evaluating our solution. Listing A.1 shows the 42 book queries, Listing A.2 shows the 45 music queries, and Listing A.3 shows the 46 video queries. How the reference collection constituting of the relevant documents for different queries were made, is presented in Section 9.2. By following the steps described, the reference collection can easily be reproduced, assumed that a similar system like the one we used, is available.

```
naruto+1
barbie
gorden+west
managua
Microbiology+an+intro
midsummer+night+s+dream
oeuvre+zola
a+day+in+a+tropical+rainforest
Paradise+Lost
clown+fish
maxim+magazine
Geology
hurricane+rita
Yes%2c+you+can+still+retire+comfortably
Adult
Veterinary+Helminthology
vanity
food+calorie+counter
working+with+spoken+discourse
trumpets
grid+photos
The+Ezekiel+Option
differential+equations+by+polking
itzykson+field
Rita+Mulcahy
The+Guv'nor
top+ten+books
The+Phantom+Tollbooth
tsubasa
Books+for+Teens
battleships
Booty+magazines
overcoming+the+enemy
alexander+mccall+smith
green+gables
boston
women+with+attention+deficit+disorder
mathematical+quantization
```

```
ask+question
puma
performance+measurement+in+healthcare
semantic+web+owl
```

Listing A.1: The 42 Book Queries used in the Evaluation

```
reservoir+dogs+soundtrack
gloria+gaynor
I'd+Rather+Be+Alone
Rob+Zombie
babylon+by+bus
Bule+Bule
We+Will+Worship+by+Dennis+Jernigan+music
Van+Morrison
Die+Monster+Die
Satanic+Surfers
duke+of+earl+history
fabolous+ghetto
new+song
Wayne+Wonder
Nina+Sky
Alice+Cooper
mcqueen+street
Jonathan+Richman
gangster+records
GUITAR+STRINGS
DC+Talk
Tommy+Boy
forest+gump
Prince+-+Musicology
carlos+mata
now+18
Ry+Cooder
kalimba
magic+numbers
DMX
ethnic+instruments
polkas
Dave+Matthews
benabar
Mc+Magic
bone+thugs+n+harmony
creedence+clearwater
Music+for+Modern+Living+3
The+Green+Children
Get+rich+or+die+tryin+soundtrack
javed+akhtar
jack+johnson
Jonnie+Mitchell
linkin+park
come+closer
```

Listing A.2: The 45 Music Queries used in the Evaluation

```
andrew+keegan
john+lithgow
big+daddy
```

```
4  driver+education
5  the+incredible+mr+limpet
6  vanessa+angel
7  acrobat
8  greta+garbo
9  peter+jackson's+king+kong+production+diaries
10 sophie+marceau
11 daddy+yankee
12 agua+chocolate
13 Ilocos+Sur
14 laura+kightlinger
15 priscilla+barnes
16 jake+weber
17 kohls
18 demon+inuyasha
19 donald+o+connor
20 placido+domingo
21 sylvia+sidney
22 Cousins
23 wil+wheaton
24 shirley+maclaine
25 woody+allen
26 Sanyo
27 jane+march
28 jason+flemyng
29 tara+subkoff
30 rene+russo
31 ebey
32 Along+Came+Polly
33 dave+chappelle
34 eliza+dushku
35 steve+zahn
36 jane+campion
37 what's+up+doc
38 SEA+OF+LOVE
39 individuality
40 free+music
41 when+you+are+mine
42 daniel+auteuil
43 jean+louisa+kelly
44 liev+schreiber
45 ALICE+IN+WONDERLAND
46 digital+camcorder
```

Listing A.3: The 46 Video Queries used in the Evaluation

Bibliography

[1] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. New directions on agile methods: A comparative analysis. In *ICSE*, pages 244–254, 2003.

[2] Rakesh Agrawal and Edward L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD Conference*, pages 297–306, 2000.

[3] Fast Search & Transfer ASA. *Book of Search*. 2006.

[4] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[5] John Battelle. *The Search*. Penguin Group, 2005.

[6] Michael W. Berry, Susan T. Dumais, and Todd A. Letsche. Computational methods for intelligent information access. In *SC*, 1995.

[7] Konstantin Beznosov and Philippe Kruchten. Towards agile security assurance. In *NSPW*, pages 47–54, 2004.

[8] Boehm and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[9] J. C. Borda. Memoire sur les elections au scrutin. *Histoire de l'Academie Royale des Sciences*, 1781.

[10] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[11] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–, 2002.

[12] David Carmel, Yoëlle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. Searching xml documents via xml fragments. In *SIGIR*, pages 151–158, 2003.

[13] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[14] Y. Chiaramella, P. Mulhem, and F. Fourel. A model for multimedia information retrieval, 1996.

[15] Jan Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[16] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.

[17] Very Large Databases. *VLDB Endowment Inc.* `http://www.vldb.org`.

[18] Marquis de Condorcet. Essai sur l'application de l'analyse a la probabilite des decisions rendues a la pluralite des voix, 1785.

[19] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. Xml-ql: A query language for xml. In *WWW The Query Language Workshop (QL)*, `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`, 1998.

[20] P. Diaconis and R. Graham. Spearman's footrule as a measure of disarray. *J. of the Royal Statistical Society, Series B*, 39(2):262–268, 1977.

[21] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, pages 613–622, 2001.

[22] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation revisited, 2001.

[23] Daniel Egnor and Robert Lord. Structured information retrieval using xml. *Proceedings of the ACM SIGIR 2000 Workshop on XML and Information Retrieval*, 2000.

[24] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *SODA*, pages 28–36, 2003.

[25] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD Conference*, pages 301–312, 2003.

[26] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[27] Norbert Fuhr and Kai Großjohann. Xirql: A query language for information retrieval in xml documents. In *SIGIR*, pages 172–180, 2001.

[28] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.

[29] Google. ** * - Google search.* `http://www.google.com/search?hl=en&q=%2A+%2A&btnG=Google+Search`. Accessed May 09 2006.

[30] Torsten Grabs and Hans-Jörg Schek. Generating vector spaces on-the-fly for flexible xml retrieval. *Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval*, pages 4–13, 2002.

[31] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.

[32] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.

[33] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD Conference*, pages 16–27, 2003.

[34] Bernard J. Jansen and Amanda Spink. How are we searching the world wide web? a comparison of nine search engine transaction logs. *Inf. Process. Manage.*, 42(1):248–263, 2006.

[35] J. G. Kemeny and L. Snell. *Mathematical Models in Social Sciences*. Ginn, 1960.

[36] John G. Kemeny. Mathematics without numbers. *Daedalus*, 88:575–591, 1959.

[37] Werner Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

[38] Mei Kobayashi and Koichi Takeda. Information retrieval on the web. *ACM Comput. Surv.*, 32(2):144–173, 2000.

[39] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[40] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, 36(6):47–56, 2003.

[41] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[42] Shaorong Liu, Qinghua Zou, and Wesley W. Chu. Configurable indexing and ranking for xml information retrieval. In *SIGIR*, pages 88–95, 2004.

[43] J. Le Maitre. Indexing and querying content and structure of xml documents according to the vector space model. In *Proceedings of the IADIS International Conference WWW/Internet 2005*, volume II, pages 353–358, Lisbon, Portugal, October 2005.

[44] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.

[45] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29, 1999.

[46] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[47] Benoit Favre Patrice. Information retrieval on mixed written and spoken documents, 2004.

[48] Feng Qiu, Zhenyu Liu, and Junghoo Cho. Analysis of user web traffic with a focus on search activities. In *WebDB*, pages 103–108, 2005.

[49] RANKS.NL. *Stopwords in different languages*. `http://www.ranks.nl/stopwords/norwegian.html`.

[50] J. Robie, J. Lapp, and D. Schach. Xml query language (xql). In *WWW The Query Language Workshop (QL)*, `http://www.w3.org/TandS/QL/QL98/pp/xql.html`, 1998.

[51] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *ICSE*, pages 328–339, 1987.

[52] Albrecht Schmidt, Martin L. Kersten, and Menzo Windhouwer. Querying xml documents made easy: Nearest concept queries. In *ICDE*, pages 321–329, 2001.

[53] ACM SIGIR. *ACM SIGIR Special Interest Group on Information Retrieval Home Page*. `http://www.acm.org/sigir/`. Accessed February 01 2006.

[54] ACM SIGMOD. *ACM SIGMOD Online*. `http://www.sigmod.org/`. Accessed February 01 2006.

[55] Börkur Sigurbjörnsson and Andrew Trotman. Queries: Inex 2003 working group report.

[56] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[57] M. Truchon. An extension of the condorcet criterion and kemeny orders. *cahier 98-15 du Centre de Recherche en Economie et Finance Appliquees*, 1998.

[58] Henrik Tveit. *Towards an automated procedure for annotation of gene products*, 2004.

[59] Odd Richard Valmot. Søkemotoren overtar for databasen. *Teknisk Ukeblad*, (18):40–42, 2006.

[60] W3C. *XML Path Language (XPath)*. `http://www.w3.org/TR/xpath20/`. Accessed April 28 2006.

[61] W3C. *XQuery 1.0: An XML Query Language*. `http://www.w3.org/TR/xquery/`. Accessed March 17 2006.

[62] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[63] S. K. Michael Wong, Wojciech Ziarko, and P. C. N. Wong. Generalized vector space model in information retrieval. In *SIGIR*, pages 18–25, 1985.

[64] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, pages 537–538, 2005.

[65] Yahoo! *Yahoo! Search blog: Our blog is Growing up - And So Has Our Index.* `http://www.ysearchblog.com/archives/000172.html`. Accessed May 09 2006.