Solbrille : a simple search engine

Arne Bergene Fossaa, Simon Jonassen, Jan Maximilian W. Kristiansen, Ola Natvig

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

Abstract

This report describes the design ideas, concepts and some of the implementation details of **Solbrille Search Engine**. The report also includes an evaluation of the search engine retrieval (qualitative) performance, both with and without clustering, and a number of suggestions for the further development and extension.

Preface

Solbrille Search Engines was designed and produced during the course project in TDT4215 ‘Web-Intelligence’ at Norwegian University of Science and Technology, Spring 2009. All the work was performed by the four group members listed as authors of this report. Some of the ideas behind the search engine were inspired by **Brille** (buffer management) and **Terrier** (modular query processing) search engines.

The authors would like to thank Truls A. Bjørklund for providing the source code of **Brille** which was the main inspiration source in an early phase of the search engine development.

Arne Bergene Fossaa, Simon Jonassen, Jan Maximilian W. Kristiansen, Ola Natvig
Trondheim, 30th March 2009

Contents

List of Figures

Chapter 1

Introduction

something fishy

Chapter 2

System Requirements

The project assignment stated by the TDT4215 cours staff was to create a search application, implemented either in Python or in Java, consisting of a basic system and an extended system. Only approved libraries could be used in the final application, other libraries not listed on the course web page could be approved by contacting the staff members. Some of the specific challenges were phrase search and proximity.

The project requirements stated by stuff were as follows:

- The preprocessing should include tokenization, stopword removal and stemming.
- The indexing and query retrieval should use the cosine vector model and inverted files that must be stored and loaded on startup.
- The resulting application should use a clearly defined query language.
- The query result should be sorted and presented to the user according to the similarity ranking, the result should also include a link to the source document.
- The final implementation should be evaluated on time collection with 10 defined queries.
- The extended system should use a clustering technique to improve the search quality, the document clusters must also be ranked according to a similarity measure.
- The project, report and presentation deadlines were set to 39 days (5 weeks) from the project start.
- The number of group members were limited to maximum five persons.

Chapter 3

System architecture

The system is designed around three major data structures, or indices. These are the occurrence index which store inverted lists with positions. The content index stores the content of the documents, and is used for snippets and clustering. The last index is the statistics index which is used to store document statistics which may be used to calculate relevance.

All these indices are wrapped by one class `SearchEngineMaster`. The external interface for the system uses this master class to feed documents and to execute queries.

3.1 Statistics index structure

The statistics index is a mapping from document id to a statistics object containing information such as most frequent term, document vector $tf * idf$ length and number of unique terms.

3.2 Content index structure

The content index stores the content of the documents indexed, these documents are stored as lists of tokens. These are the raw tokens of the documents nothing is added or removed. By concatenating any consecutive subsequence of these tokens a section of the original document will be produced. This property are used when extracting snippets.

3.3 Occurrence index structure

The occurrence index consists of two parts, the dictionary and the inverted file. The dictionary contains the terms of documents, and a pointer into the inverted file. The inverted file contains the inverted lists for each term in alphabetical order. The inverted list of a term contains the documents occurrences that contain the term sorted on increasing document id. Each document occurrence contains a list the positions within the document where the term occurred. The index structure is shown in Figure 3.1.

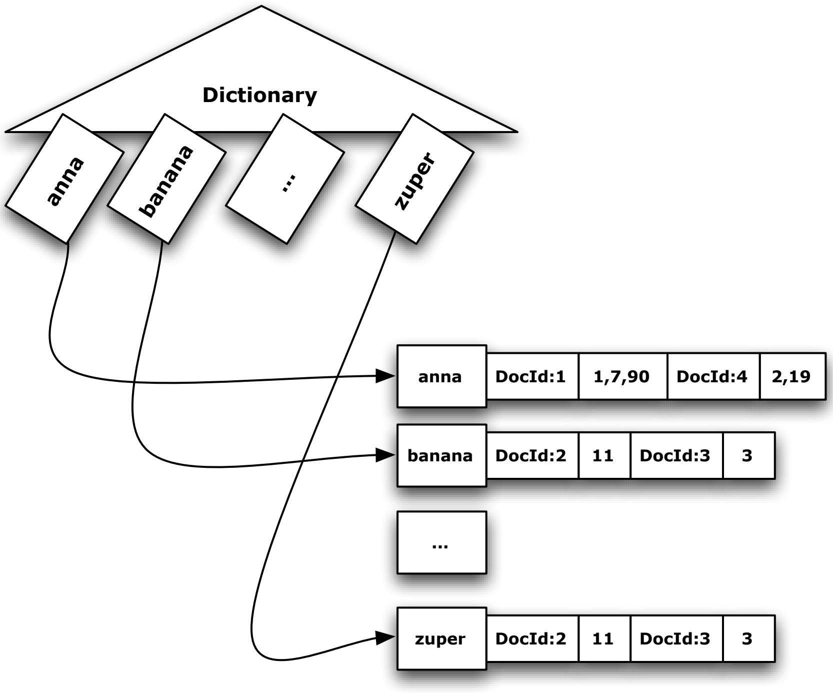
Dictionary
anna
banana
...
zuper
anna | DocId:1 | 1,7,90 | DocId:4 | 2,19
banana | DocId:2 | 11 | DocId:3 | 3
...
zuper | DocId:2 | 11 | DocId:3 | 3

Figure 3.1: Occurrence index structure.

3.4 Index building

When building indices, all documents fed to the system are given a unique document id from a global counter.

The occurrence index is built in two phases, in the first phase documents are converted to inverted lists and combined into one inverted list representing one index update.

When the index is flushed, the index update that were built in the first phase is merged with the existing inverted index. Since the inverted lists are sorted on term, document and position merging the lists is a trivial task.

During this merge, the statistics for each document is calculated and stored in the statistics index.

When the merge is completed the dictionary is updated so that the pointers in the dictionary points into the newly created inverted file.

3.5 Feeding pipeline

The feeding pipeline of our solution is modeled around two main concepts. A document structure which is a object containing various objects with various keys (fields), it's basically a map. A processor is a processing unit in our feeding pipeline which transforms a field in a document structure, and places the result in another. An example of a feeding processor is shown in Figure 3.2.

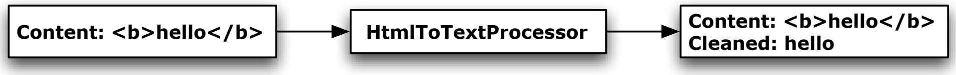

Figure 3.2: Feeding pipeline example, a html to text processor strips away html from the input field “content”, and puts the result into the “cleaned” field.

3.5.1 Feeding processors

To be able to solve the task the feeding pipeline has to implement multiple feeding processors. These are:

- **HtmlToText:** strips away HTML tags.
- **Tokenizer:** brakes the documents into tokens. These tokens are the raw tokens of the documents, that is, no characters are removed from the text, the text is only split into pieces.
- **PunctuationRemover:** removes punctuations and whitespaces from tokens.
- **Stemmer:** reduces tokens to their stems.
- **Termizer:** creates inverted documents, collecting position lists for each term in the document.

3.6 Query Processing

The query processing part of the system is designed to be both flexible and extensible. The main idea is to have a query processor which uses a query preprocessor and a (number of) query processing components to produce and sort the query results. A query preprocessor uses a number of built-in stages (similar to those used for the document processing) to transform a textual query into a query structure.

A query processing component, on the other hand, is an abstract component that may be used to pull query results. Each component may include another component as a source, and it is possible to construct a chain of such components. To begin with, a minimal number of processing components include:

- **Matcher:** produces a number of combined results for a given query. These match inclusion and exclusion of terms based on AND, OR and NAND modifiers.
- **Score Combiner:** pulls from its source and scores them according to a number of score modules that can be added to the system. The system may implement

a number of different similarity models (both dynamic and static) and a score combiner can calculate a weighted average from these.

- **Filters:** may filter out some of the results based on a number of abstract filters that can be later implemented and added to the system.

The most important point here is that the query processor does not have to know what components are used in the query processing, and a system implementation is free to extend the number of components, add several instances of the same component in the processing pipeline or change the processing order of these.

In future, the system may be extended by a merger that enables to merge a number of processing pipelines into one. In this case, the system may become distributed. For example, it may run a number of basic matchers, scorers and filters on a number of different nodes, then results from these may be pulled together by a new merger component which may combine the results. If required, the processing pipeline may be extended by a couple of new filters and scorers used to re-weight and filter the combined results. However, due to the project duration limits the target search engine is chosen to run only on a single computer.

The rest of this section explains the basic processing components and the ideas behind.

3.6.1 Matcher Component

The matcher component is the part of the processing pipeline which performs the actual lookups into the inverted index (OccurenceIndex). The matcher extracts the various terms referenced in a query and executes one lookup for each of the terms. It also recognizes the modifiers associated with the terms (*AND*, *OR* and *NAND*). The matchers purpose is to create candidate results for further filtering and ranking. A candidate result is a document which satisfies the query. That is, a document where *all* terms with a *AND* modifier is present and *no* terms with a *NAND* modifier. In addition the terms with *OR* modifiers are appended to the documents which satisfies the *AND* and *NAND* requirements.

The matcher component allows us to support a limited subset of boolean queries, however, since it has no aspect of parenthesises the subset is rather small. For vector space queries, all terms are treated with *OR* modifiers.

Critique and an Alternative Approach

A problem suggested with a more general scheme were that index term would be read twice. However, this is not a case with the proposed design for buffer manager and processing component. Since the results are pulled one by one and combined as soon as possible, a repeated request to the same term will be cached in the buffer manager. Therefore this method can be quite efficient.

3.6.2 Score Combiner and Scorer Components

The idea with score combiner is to retrieve a number of score values from different scorers, and then combine these into a total score. The main intention here is that the system may implement different scoring models or even combine these. A scorer may implement a cosine model, a probabilistic model such as Okapi BM-25, a static link-analysis models such as PageRank or HITS, or even an Opt model[1].

To be fast, some of the document collection statistics later required by scorers has to be stored in the query requests and query results, since theaccess to global statistics on demand may be too slow.

3.6.3 Filters and Phrase Search

To demonstrate the idea with filters the system is intended to present two types of filters, a non-negative filter and a phrase filter. A non-negative filter is rather a demonstation of the concept and it can just filter out the results which have a score value less or equal to zero. However, using the scorer components specified above, this situation is impossible.

A phrase filter has a more practical application. The main intention is to filter results based on + and - phrases.

A simple case for the phrase search is either a phrase such as *+"kari bremnes"* or a phrase like *kari +"kari bremnes"*. The solution for both of these is to represent AND phrases as lists of terms in the query structure, match and score the corresponding terms just as normal terms, and then finally check if these terms occur in the required order using the document occurrence part of the inverted index.

A more problematic case for the phrase search is a query such as *kari -"kari bremnes"*. In this case the user wants to retrieve all the documents containing word *kari*, but not the word *kari* followed by the word *bremnes*.

The solution is to introduce a fourth modifier type, PNAND. As for the AND phrases, the query structure has to store a number of phrases represented as a modifier (AND or NAND) and a list of terms. All the terms represented in a phrase has the same modifier as the phrase itself, and terms having both PNAND and an other modifier have to loose the PNAND modifier. The matcher component has to perform a match on both NAND, OR and AND terms, and also PNAND terms interpreted as OR terms. Any scorer component has then to ignore PNAND terms, giving 0 in the score impact of these. Any results matching either NAND phrases of only PNAND terms have to be removed in the phrase filter.

Finally, in the current design a result says to be filtered out if one of the filters does not approve the result. This approach can be later extended to use a boolean expression of filter constraints, represented as a tree or similar.

[1] the optimal model, an inside joke

Critique and an Alternative Approach

The approach described above is chosen to be implemented, just to demonstrate the idea behind the filtering component. However, as the phrase filtering is now performed during all the stages of the query processing pipeline this method is rather impractical. A better solution that can be applied in future is to use a separate phrase matcher, and then combine the matched phrase results and the ordinary term results before sending these to a scorer component. The same critique mentioned in Section 3.6.1 with respect to multiple index lookups applies for this as well.

3.6.4 Final Details

To save memory the results pulled by the query processor are stored into a priority queue. As a query is expected to be a query string and a page range *startpage* end *endpage*, the maximum queue size is limited by *endpage*. When the queue size has reached this value a new candidate has to be compared with the least scored result contained in the queue. If the candidates value is greatest of these two, the candidate has to replace the least scored result in the priority queue.

When all the results are processed, up-to *endpage* - *startpage* results has to be extracted from the queue, sorted in descending order and returned to the user.

3.7 Extended System

3.8 Testing and Front-End Design

Chapter 4

Implementation

Chapter 5

Evaluation Experiments and Results

The system have been evaluated on the TIME collection ([TIM09]) provided by the course staff. The collection consist of 423 documents and 10 queries with specified relevant documents.

5.1 Basic System

The basic system has been evaluated using only precision and recall, no performance considerations has been made. The results from a default implementation using cosine scoring mode are demonstrated by Figure 5.1. The figure plots a line for each of the queries, x and y axis correspond to recall and precision, and point i on each line correspond to a (precision,recall) value after inspecting top i results. The diagram plots up-to 50 top results for each query.

The diagram is quite easy to understand by keeping in mind that a correct result will either increase both the precision and recall or keep them at 1.0. After the maximum number of correct results is achieved, the recall will stay at 1.0, while precision will drop. Wrong results always result in a drop in precision, while the recall value is kept at the same level.

As Figure 5.1 shows, the total search quality is somewhat poor - for two out of ten queries the ferst result is already irrelevant, and the highest possible recall value at precision level 1.0 is 0.6, while it is impossible to achieve a precision higher than 0.6 for a recall level at 1.0. The best combination of precision and recall altogether is slightly off (0.8, 0.8)

As the Okapi BM-25 has also been implemented, it was quite interesting to evaluate its performance against the cosine model. As Figure 5.2 shows, Okapi results in a great quality improvement. Only one query fails to on the correct top results, but it contains both correct results within top five results. Most of the queries success to achieve a combine recall-precision value greater than (0.7, 0.7), and most important one of the queries success also to achieve a value at (1.0, 1.0), while the worst performance (observed for query number four) is just the same as for the cosine model.

5.2 Extended System

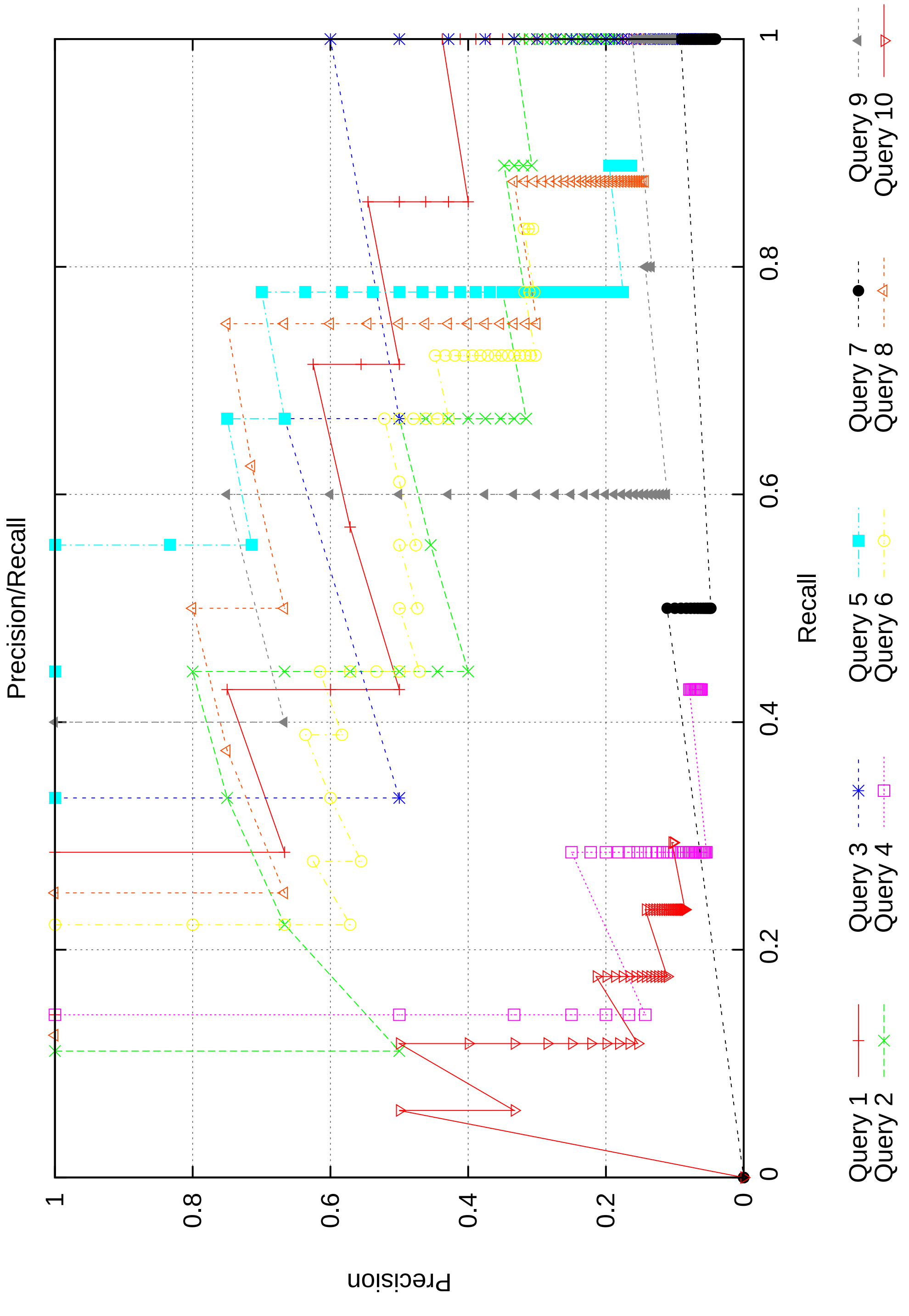

Figure 5.1: Evaluation Results with Cosine Scoring Model

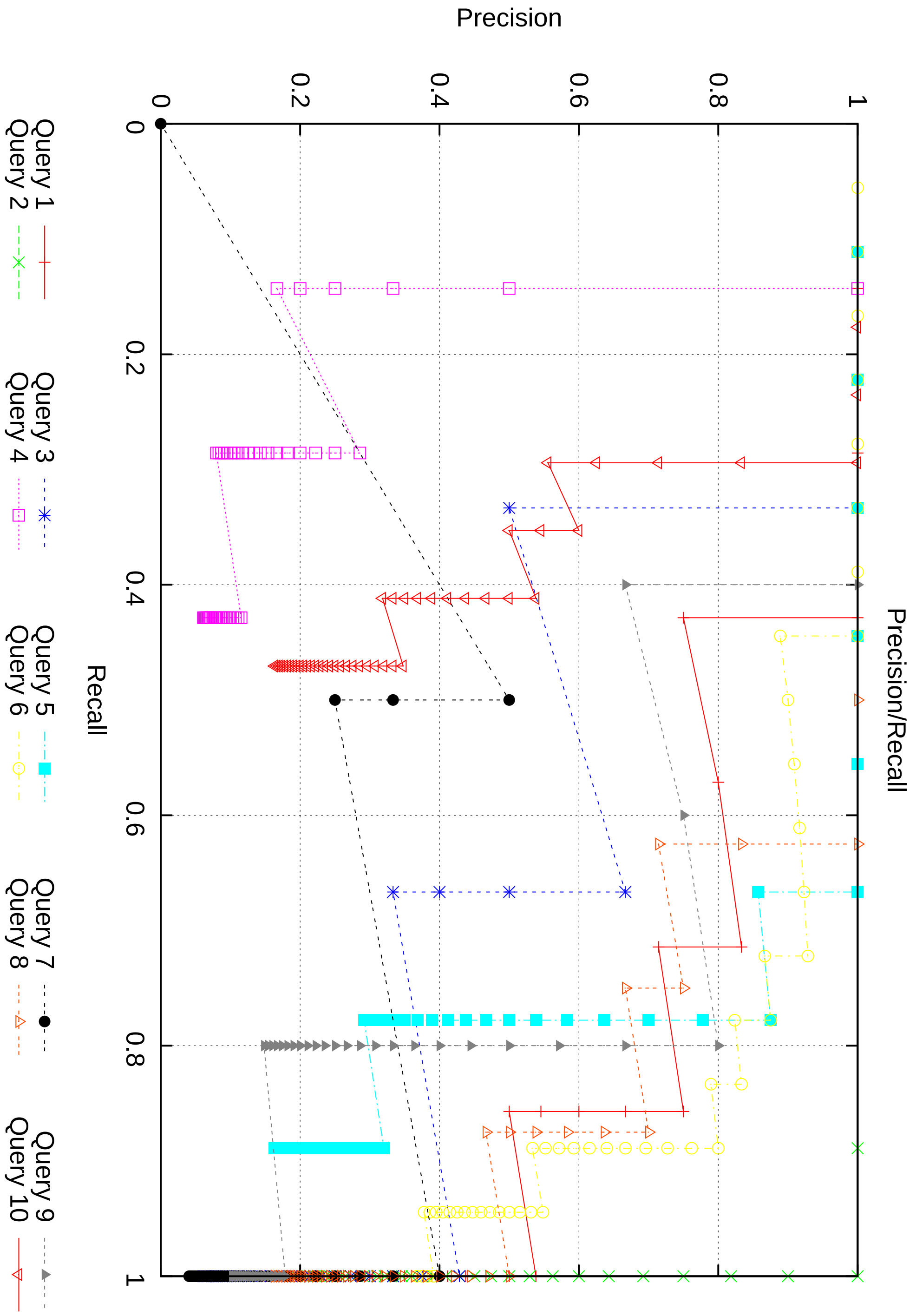

Figure 5.2: Evaluation Results with Okapi BM-25 Model

Chapter 6

Conclusions and Further Work

whatever

Bibliography

[TIM09] Time Collection supplied by TDT4215 course staff. `http://www.idi.ntnu.no/emner/tdt4215/collections/TIME-collection.zip`, 2009.