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. Bj¿rklund 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 **??**.

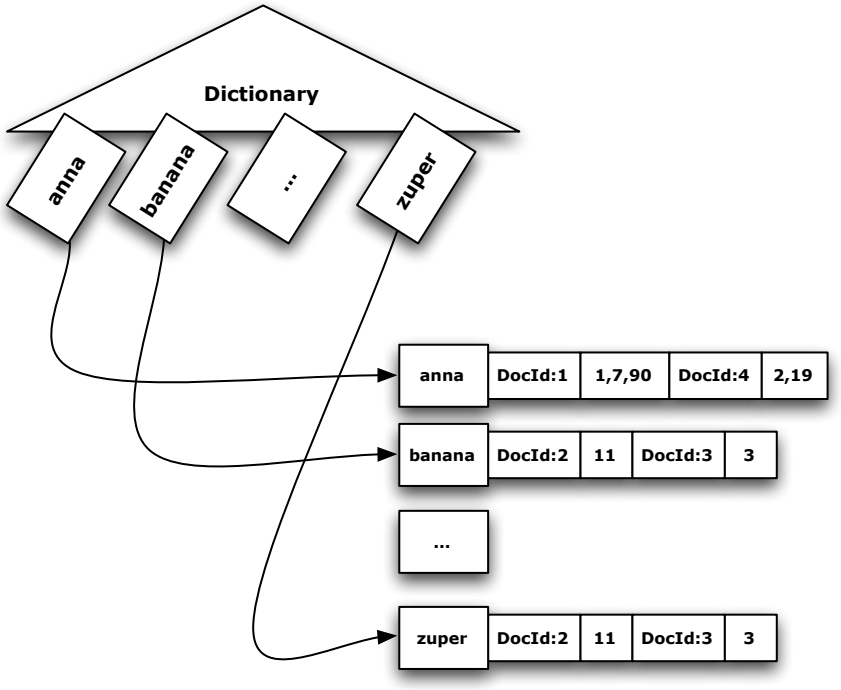

Figure 3.1: Occurrence index structure.

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 **??**.

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 Implemented feeding processors

To be able to solve our task we have implemented 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.

Chapter 4

Implementation

Chapter 5

Evaluation Experiments and Results

blablabla

Chapter 6

Conclusions and Further Work

whatever