Solbrille : Bringing Back the TIME

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

Wordcount: 6545

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's retrieval 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 the 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

The assignment given for the project in TDT4215 *Web-Intelligence* were to develop a search application using an inverted index file, a vector ranking model and a clustering technique to improve the search. All the background knowledge for these topics is partially covered by the course curriculum which includes the bible of information retrieval, "Modern Information Retrieval" [BR99], and a number of papers.

However, [BR99] gives rather a broad overview of the field of IR than a detailed description of how inverted index based search works. For this purpose, a reader may be consider to look at [ZM06] and [WMB99], which describe both basic concepts and advanced performance improvement techniques.

Two another knowledge sources used early in this project were Terrier [OAP+06] and Brille [Bjø07] search engines. The inspiration from Terrier lies in a modular design of the processing components, as it will be demonstrated later. Terrier also splits query processing into several steps such as preprocessing, matching, scoring and postprocessing. A modular design allows developers to change small fractions of code (modules or components) in order to evaluate a completely new/different architecture concept.

Brille Search Engine was originally produced during a similar project at the same course couple of years ago, and later altered by one of its authors, Truls Amundsen BjÃ¸rklund, as a part of his Ph.D. research. The version known to the authors of this report is an outdated version accessible from Truls' Home Page (`http://www.idi.ntnu.no/~trulsamu/brille.tar.gz`).

As the available version was mainly focused on hierarchical indexes and index freshness [Bjø07], the corresponding search engine was more a fast index handler with a good buffer manager. (According to rumors, Brille handles TREC GOV2 more effectively than Lucene Search Engine).

The ideas taken from Brille are within Buffer management and performance issues of Java's standard implementation of ArrayLists (which is quite storage inefficient and slow). The concept of the search engine itself, feeding, indexing, processing and presenting were conducted by the authors on their own.

1.1 Target Goals and Motivation

As two of the four group members are graduate students, and three of four are actually working with search, the main motivation was not just complete the project, but actually produce a well working search engine from scratch. However, as there exist a large number of freely available search engines such as Lucene, Solr, Terrier, Zettair, MG4J, etc., it is not the product, but the process and the ideas behind are of the highest value. Also due to the project limits it was impossible to achieve a great execution performance of the resulting product, but all the decisions to be presented will always consider about how the performance can be improved later, and how the search engine application can be extended to become a worthy product.

1.2 Reading Guide

So far this chapter has introduced the assignment itself and listed the background theory and inspiration sources used for this project. Section 2 will now present a short summary of the assignment text. The design and the system architecture will be outlined in Section 3. Section 4 will describe the application implementation at a very high level. The application will be further evaluated in Section 5. Any conclusions and a number of suggestions to further improvements will be listed in section 6.

Chapter 2

System Requirements

The project assignment stated by the TDT4215 course 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 staff 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 on disk and loaded on startup.
- The resulting application should use a clearly defined query language.
- The query results 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

To satisfy the requirements specified in the previous section a system can be designed in terms of a number of data structures used to store the index and statistics data and a number of major components. For data the data structures it is chosen to implement three types of indices - content index, occurrence index, and document statistics index. The major components of the system are chosen to be a buffer manager to handle disk I/O, a feeder chosen to feed the text files into the system, an indexer chosen to construct and store index, a query retrieval part chosen to process textual queries using the stored index, score the candidate results and generate *snippets* for the top-scored results, a clustering component that may perform result clustering and cluster scoring based on provided snippets, and finally a front end part chosen to be a dedicated web-server. In addition the system has to contain a number of components for performance evaluation, unit testing and debugging.

3.1 Index Structures

The system is designed around three major data structures, or indices. These are the occurrence index which stores 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.

3.1.1 Document and 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.

3.1.2 Content index structure

The content index stores the content of the documents indexed, these documents are stored as lists of raw document tokens, with nothing being added or removed. By con-

catenating any consecutive subsequence of tokens a section of the original document can be produced. The last property is used when extracting snippets.

3.1.3 Occurrence index structure

The occurrence index consists of two parts, a dictionary and an inverted file. The dictionary contains the indexed terms, and a pointer into the inverted file. The inverted file contains an inverted list for each term stored in alphabetical order. The inverted list of a term contains all the documents occurrences of this term sorted by increasing document ID. Each document occurrence contains a list the positions within the document where the term occurs. The index structure is shown in Figure 3.1.

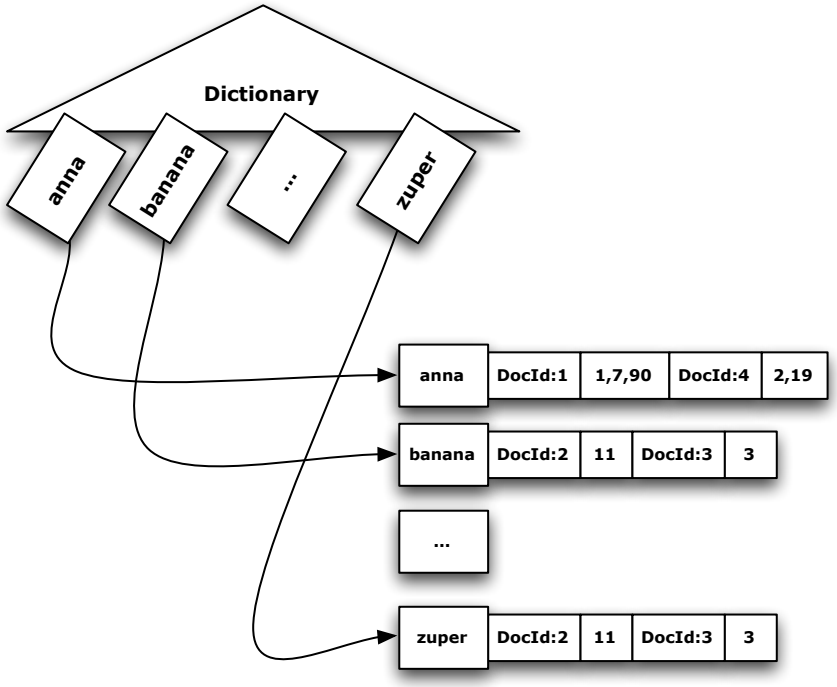

Figure 3.1: Occurrence index structure.

3.2 Buffered IO

To solve one of the major bottlenecks of search, namely disk I/O a buffer manager can be used. A buffer manager can split files into blocks and, when a part of a file is requested for reading or writing, it may fetch it entirely from disk into a memory buffer, eventually saving time on later requests to the same file area. However, the memory used by a buffer manager is limited to a certain number of buffers and a buffer size; newly fetched buffers

has to replace already stored buffers. The easiest way to handle buffers replacement is to use a Least Recently Used (LRU) replacement policy. In addition, the buffer manager has to implement flushing of dirty buffers, but the technicalities of the buffer manager implementation are quite irrelevant for the context of this assignment and, if needed, they can be obtained from the source code itself.

3.3 Feeding pipeline

An important processing stage during the document indexing is feeding of the data into the indexer. The approach of transforming of the input data into a stream of document terms is chosen to use a *feeding pipeline*. The pipeline proposed as a solution is based on two main concepts. Frist, the structure chosen to represent a document is an object containing various objects with various keys (fields), which is basically a map. Second, a pipeline processor can transform a field in a document structure, and place the result in an other processor.

An example of a feeding processor is illustrated by Figure 3.2. Finally, the output of the feeding pipeline is submitted to the indexer process described in next section.

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.

3.3.1 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:** breaks the documents into tokens. These tokens are raw document tokens with no characters removed from the text, the text is only split into pieces.
- **PunctuationRemover:** removes punctuation and whitespace characters from tokens.
- **Stemmer:** reduces tokens to their stems.
- **Termizer:** creates inverted documents, collecting position lists for each term in the document.

3.4 Index building

When building the indices, all the documents fed into 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 was 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. However, during this merge, the statistics for each document have to be re-calculated and stored in the statistics index. Finally, when the merge is completed, the dictionary is updated so that the pointers in the dictionary point into the newly created inverted file.

3.5 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 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 has to 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 results 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

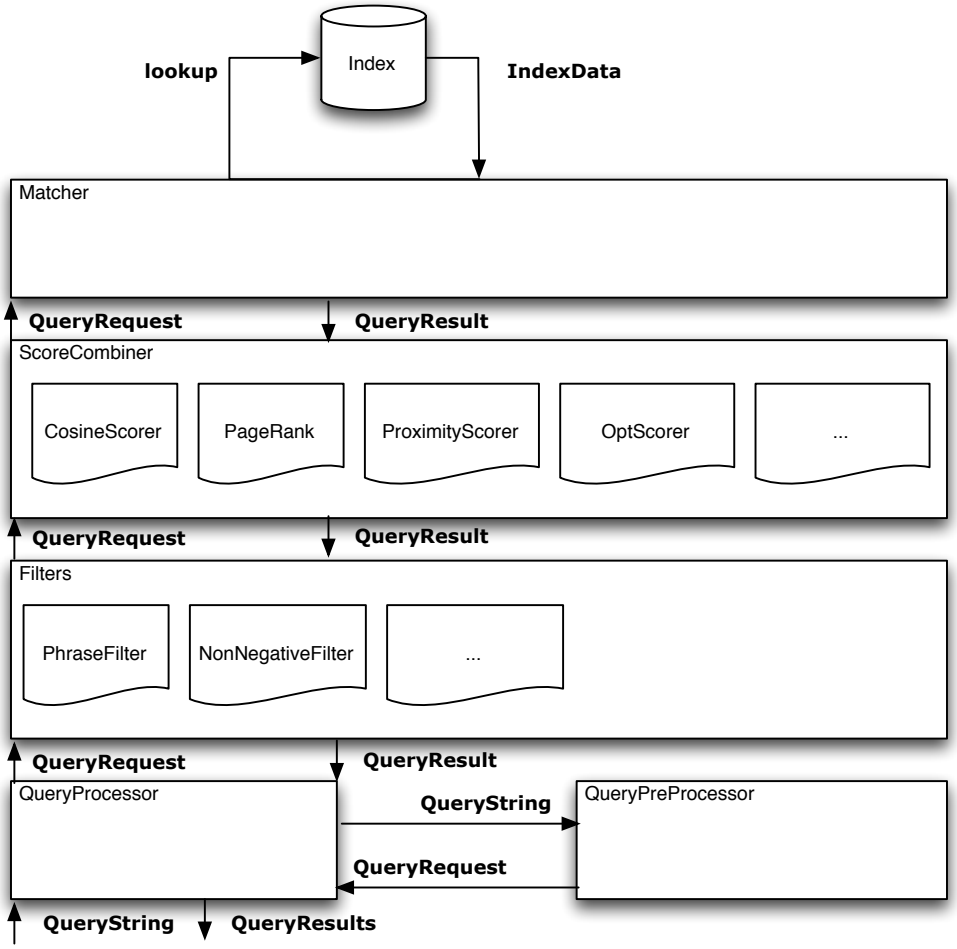

Figure 3.3: Query processing subsystem.

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 query processing structure proposed for an initial implementation of Solbrille is illustrated by Figure 3.3. The rest of this section explains the basic processing components and the ideas behind.

3.5.1 Matcher component

The matcher component is the part of the processing pipeline which performs the actual lookups in the inverted index. The matcher extracts the various terms referenced in a query and executes one lookup for each term. 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 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 Postnote)

To support the full range of boolean queries a tree of *AND*, *OR* and *NAND* matching components could be constructed. This would be a much more flexible way of building result candidates. Merging all functionality for doing matching into one component also leads to one piece of very complex code.

The reason a more general scheme was not used were that an index term could have been looked up multiple times. However, this is not a issue with the proposed design of the 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 the performance impact of this general scheme is close to nothing.

3.5.2 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 have to be stored in the query requests and query results, since the access to global statistics on demand may be too slow.

3.5.3 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

[1] the optimal model with 1.0 in both precision and recall, an inside joke

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.

Critique and an Alternative Approach (A Postnote)

The approach described above was 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 alternative solution as the one mentioned in Section 3.5.1 with respect to multiple index lookups applies here as well.

3.5.4 Final Details for Result Scoring

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.

3.5.5 Snippet extraction

To ensure a pleasant user experience, search results should contain a short piece of the original document. This snippet could be the beginning of the document, however, since

the occurrence index contains the positions of the various terms it should be possible to select more suitable snippets.

The simple scheme selected for the snippet extraction is that the position lists for all the terms in the query are merged to one list. Then a *snippet window* which contains the greatest number of queries terms. The length of the window is a predefined constant, and the unit of the window dimensions is number of terms.

Critique and an Alternative Approach (A Postnote)

Selecting the window with the highest number of matching tokens is not a good measure of snippet relevance. Common words and stop-words will be counted as much as any other word. Treating the various possible snippet-windows as documents and calculating their relevancy to the query could be a better solution, however, efficient algorithms to do this have not been studied.

To produce visual appealing search result presentations the length of the snippet windows could have been calculated in number of characters rather than number of tokens. Using tokens as the unit of measure the variance of the snippet lengths might be high.

3.6 Extended System

For the extended system, a clustering method that is based on suffix trees is used. The method is presented in [ZO98], which have been used as a reference when implementing the method.

Suffix Tree

A suffix tree is a representation of a text which speeds up several string algorithms. In a suffix tree, all suffixes of a text is held in a trie to speed up string comparisons, meaning that algorithms like substring comparison can be done in $O(m)$ time (where m is the size of the substring). In a naive implementation of a suffix tree, creation time and space would be $O(n^2)$, but when using an algorithm called Ukkonen's algorithm, both time and space can constraints can be reduced to $O(n)$. More information can be found in [Gus97].

Usually in a suffix tree, characters is chosen as the most atomic part of a text. For this project words was chosen as the most atomic part. The primary reason for this is that substring matches is not of intersect, since term frequency is used to score matches. To support using words as atomic parts, and to support having multiple documents in a tree, a variant of suffix trees called Generalized Suffix Tree is used . This is a tree that resembles a normal suffix tree, but which also has a bit set on each node, which describes which documents contain that node.

Suffix Tree Clustering

The clustering method itself is based on the method as presented in [ZO98]. What is basically done is that for every document in the result set, the snippet is extracted and put into the suffix tree. For all terms and phrases that exists in more than one document, the tree will then contain a node where the cardinality of the bit set is more than 1. A simple iteration over the tree will then extract all common phrases and terms. This is again used to create clusters of documents, where all the documents have a set of phrases in common.

Since there are usually a lot of "garbage" nodes (common words, words/phrases that only occur in one document, or nodes which contains every result in the result set), every node is scored. The score function depends on the number of words in phrase, if the words are common or not, the score calculated by the scorer and the number of documents that share the phrase.

The N best nodes is then chosen and joined. All nodes where more than half of each set are in both sets are then joined. For the joined sets, all nodes where the phrases are simple subsets of each other is removed. For example, if "president kennedy" and "kennedy" containes the same documents, "kennedy" is removed, to leave the most expressive of the two.

Finally, all the clusters (joined nodes) are re-scored, and all clusters which has a score that wis more than 1/5th of the maximum cluster score.

3.7 Front-End Design

The main responsibilities to the front-end is to handle and provide search functionality to a user. This can be done through embedding Jetty in the system. Jetty is a compact java web server which supports both embedded and stand-alone deployment of applications. Embedding Jetty gives us advantages such as reducing the complexity of the development environment, requires no installation, and in-line configuration of the web server. However, it increases the complexity of writing views because of the in-line configuration.

Jetty is configured and started by a front-end component. The idea is to have two context paths, where a context path is a domain where a given set of services are accessible. The two different context paths are used for servlets and serving of static content such as documents and images. The currently proposed servlets are:

- `A search servlet`: This is the main servlet that handles the search functionality of the application.
- `A feeder servlet`: A hook for feeding the search application with documents.

The current system proposal embeds mark-up generation in the servlets. This is a huge disadvantage as the code complexity of each servlet increases drastically. A future improvement would be to introduce the Model-View-Controller architectural pattern, and

separate the servlet and the markup. Another improvement would be to not embedding jetty, and create a generic web application which is able to run using any java web server. However, this would require a installation of the chosen web server.

3.8 Testing

Test-driven development is proposed to be used to ensure the quality of the most important components. During development, the tests can be designed and made before each component is implemented. A test is composed of one or several test cases, and in order to pass, all conditions must be correct. **JUnit** is used for implementing test cases, and they will be located in a separate web-tree.

3.8.1 A Postnote

The current test coverage is rather limited. Currently, only the *feeder*, *index*, *buffering* and *system* components are covered by tests. This is due to the fact that test development takes time, and were substituted by manual testing at the end of this project when time became limited. In future, the test coverage can be improved.

Chapter 4

Implementation

This chapter describes some of the implementation specific matters. However, it's not the purpose of this chapter to give a total understanding of all the implementation specific details. To get a in depth understanding the source code should be consulted.

4.1 Programming language and code

The *Java* programming language were chosen as the platform for the solution. In addition there are *HTML*, *JavaScript* and *CSS* code in the frontend.

4.2 Structure of solution

There are five six level packages in the **Solbrille**'s source tree. These packages represent separate pieces of functionality and their roles may be summarized as following.

- `com.ntnu.solbrille.buffering`: Used to do buffered IO and thread safe sharing of IO resources (Section 3.2).
- `com.ntnu.solbrille.feeder`: Used in the document preprocessing stage. This package contains two sub-packages: `outputs` for content targets, and `processors` for content processors (Section 3.3.1).
- `com.ntnu.solbrille.index`: Contains the utilities for creating index structures, and three specific index implementations `content`, `document` and `occurence` (Section 3.1.1-3.1.3).
- `com.ntnu.solbrille.query`: The query processing pipeline, contains sub packages for preprocessing, matching, scoring and filtering.
- `com.ntnu.solbrille.utils`: Utilities used in various parts of the solution. Contains a sub package for some special *iterators* used during indexing and querying.
- `com.ntnu.solbrille.frontend`: The web frontend for the search engine.

The relationships between these packages is summarized in Figure 4.1.

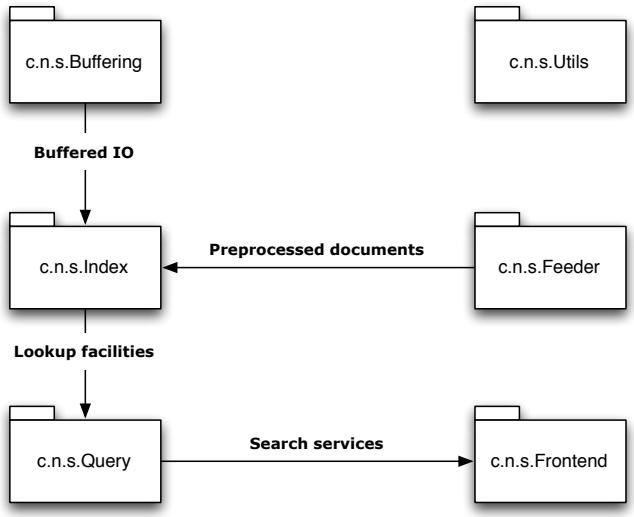

Figure 4.1: High level packages.

4.3 Index building and update algorithm

The algorithm used for incrementally building of the occurrence index is, as mentioned in Section 3.4, a two phase algorithm. Initially one index update is built in memory, then when the index is flushed this update is merged with the existing index.

The way this is managed to assure concurrent searches and index updating is that the system uses two inverted files. One inverted file which is searched, and another one which is updated. To make the index consistent the concept of a *Index phase* is introduced. A index phase is the identifier of the current searchable index. Each entry in the dictionary contains a double set of inverted list pointers, the current active set of pointers are determined by the parity of the current index phase. While merging the old inverted list and the new index update the inactive pointer of each dictionary entry is updated. When the update completes; the index phase is incremented and the active set of inverted list pointers are swapped (due to parity).

4.4 External libraries

To ease the implementation some external libraries have been used. These libraries have been run past the course staff for verification.

- **Snowball stemmer:** For stemming in the document and query preprocessing stage.

- **Carrot2 suffix tree:** For suffix tree clustering in the extended system.
- **htmlparser.org:** To strip away *HTML*-formatting from documents, to extract links for static link analysis etc.
- **Jetty:** A compact web-server to produce the front-end user interface.

4.5 Deployment and binaries

The **Solbrille** search engine is deployed as two standalone applications, one *ConsoleApplication* and a *FronEnd* which runs a web-server. The console application includes the functionality required to feed documents to the system, in additional some convenience methods for feeding the TIME-collection. In Section A.1 describes how to run the search-engine.

Chapter 5

Evaluation Experiments and Results

The system has 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 first 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.

Query	Clusters
buddhist	{"south viet nam roman catholic president ngo dinh diem" "quang duc" "buddhist monk" "virtual martial law" "flag" "religious freedom"}, {"diem government" "buddhist leaders"}, {"sister law mme ngo dinh nhu" "ngo dinh nhu has" "mme nhu has banned" "diem brother" "nine" "church" "buddhist schools"}
war	{"war south viet nam" "viet cong"}, {"international red cross"}, {"government"}, {"mekong delta" "diem"}, {"various" "sino soviet split" "maneuver" "lenin"}, {"mao tse tung"}, {saudi arabia jordan" "egypt saudi arabia" "imam" "yemeni" "war yemen" "saudi arabia has" "sallal" "tribesmen" "veteran" "badr" "egyptian" "san" "team" "limit" "wide" "little war"}
britain	{"osteopath stephen ward" "profumo case" "john profumo was" "mentor" "christine keeler" "trial charges"}, {"government"}, {"harold macmillan"}, {"national"}, {"intention" "common market britain" "efta" "tariff" "outer" "regard" "join"},{"sir alec douglas home" "luton" "plain" "proof" "campaign" "seat" "labor years" "next day"}

Table 5.1: Cluster tags for clusters returned in the TIME document set

5.2 Extended System

It is hard to fairly evaluate the success of a clustered results versus a non-clustered result, papers like [ZO98] usually compare the results to other clustering algorithms. There are several reasons for this. First, the clustered result contains information that the non-clustered result does not contain (the name of the cluster, the knowledge that the results are clustered together). Secondly, if standard precision/recall evaluation is used, several problems arise. The most important is in which order the results should be tried. Since the course of action for a human user using a clustered result would be to first check the label of the cluster, and then check the results, evaluation of the results only would not make sense. Whatever is done to evaluate the results will need human interaction.

Evaluating simple queries with suffix tree clustering shows that short queries are usually the best when trying to find good clusters. With longer queries, there are more "garbage" nodes, which not necessarily contains documents which should be clustered together.

Because of the problems with using standard precision/recall to score clustered queries, as well as suffix tree clustering not being good on long queries (and the queries in the test set was quite long), it was chosen not to do a quantitative test with the suffix tree clustering method.

If a simple qualitative test is done, by doing random queries like "buddhist", "war" or "britain", many (but not all) of the clusters that are returned have articles that are grouped together. The examples can be seen in 5.1.

The suffix tree implementation still needs a bit of tweaking when it comes to the

scoring function that selects the ranking of the clusters, as well as testing with a more diverse data set. However, the clustering method still fulfills its mission, clustering articles with similar subject.

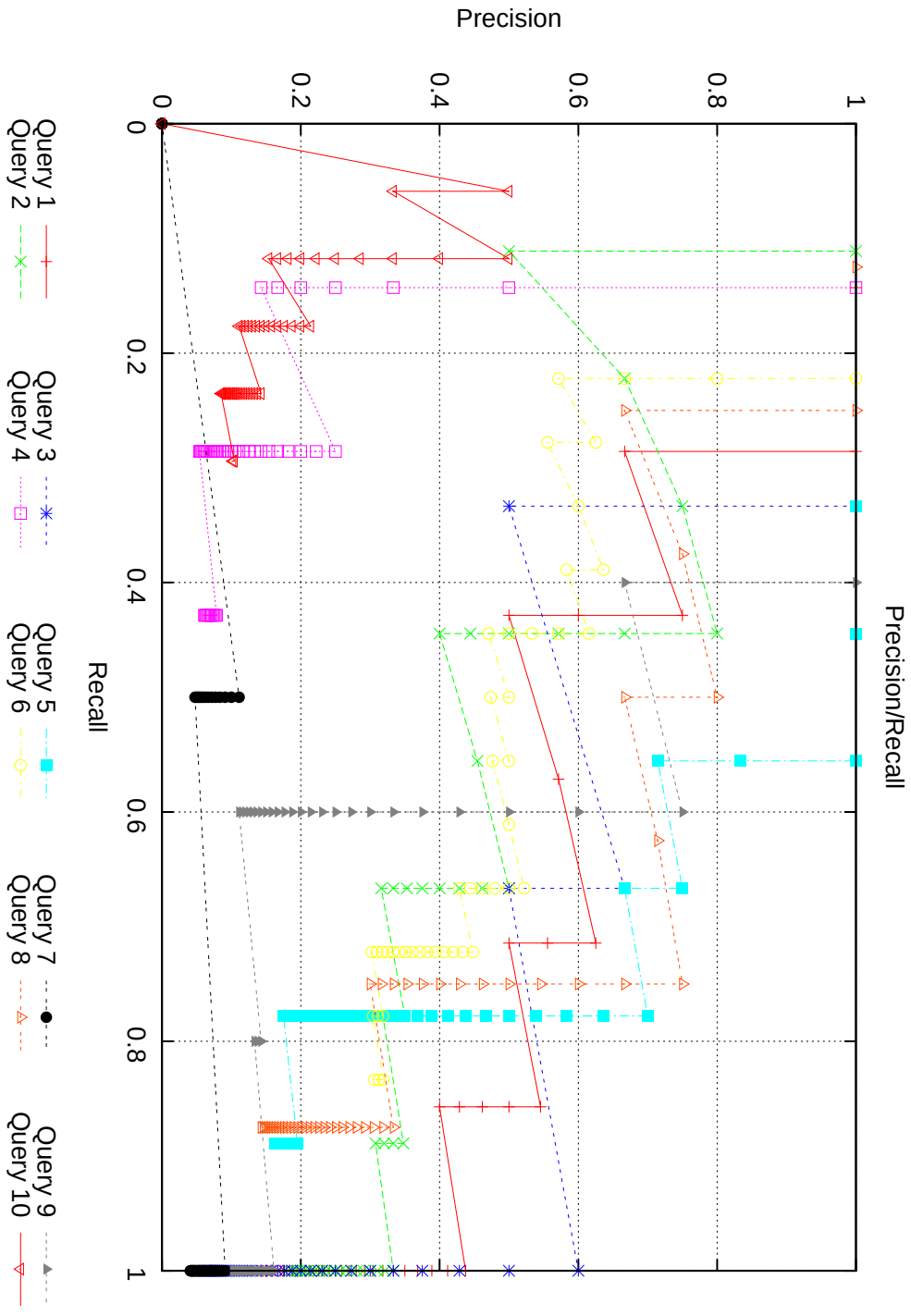

Figure 5.1: Evaluation Results with Cosine Scoring Model

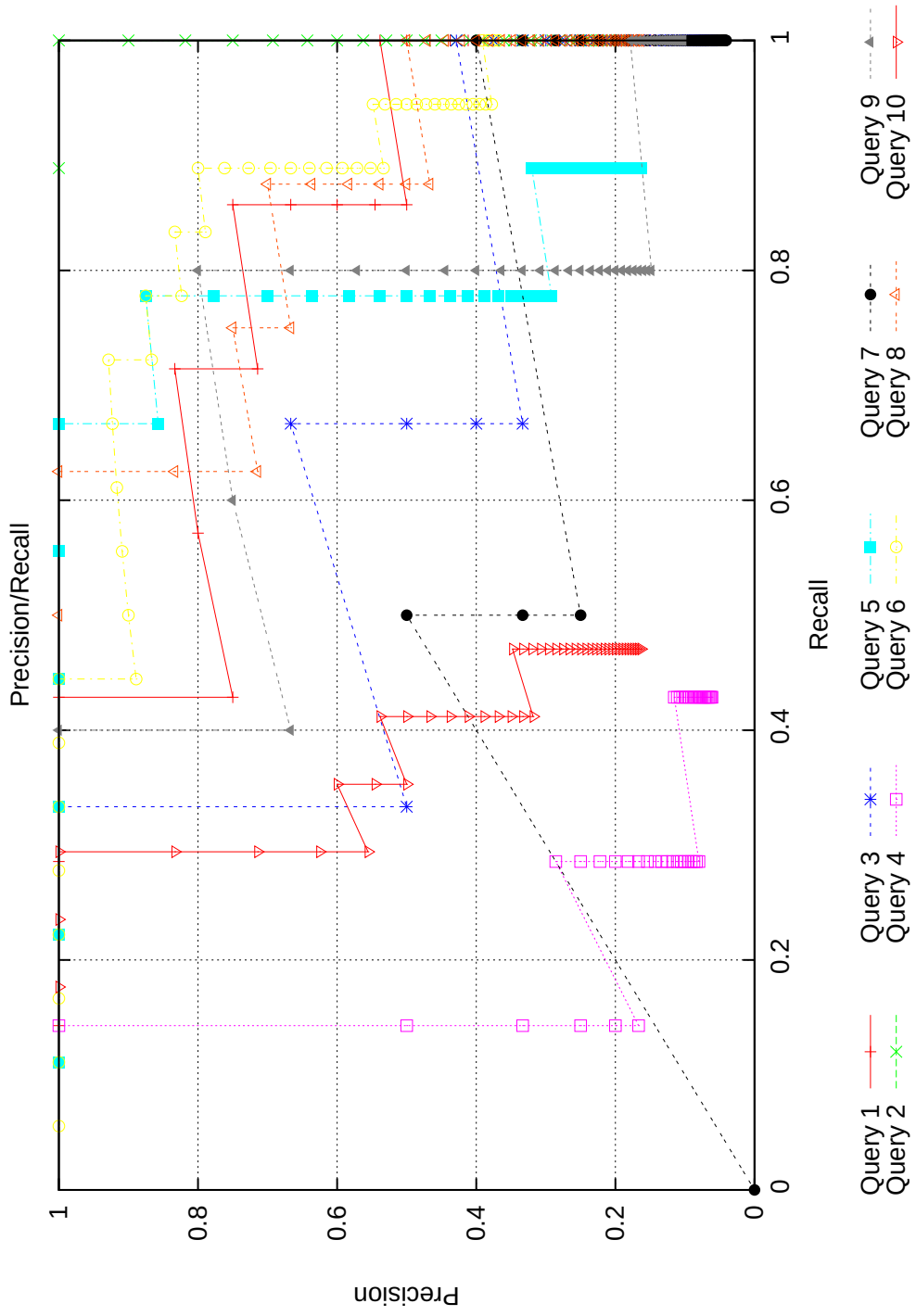

Figure 5.2: Evaluation Results with Okapi BM-25 Model

Chapter 6

Conclusions and Further Work

As it has been presented so far, **Solbrille** is already a working search engine designed and implemented according to the concepts covered by curriculum and sources mentioned in the introduction section. However, there are number of issues for further development, redesign and improvement:

- Feeder can be redesigned and improved to provide a better performance.
- Index reading and writing part may be improved to provide a better performance and handle concurrent index updates. More efficient methods for storing and retrieving statistics information may be required.
- Stop word removal should be implemented in future.
- Matcher can be redesigned according to the ideas proposed in the system architecture section. In future a matcher will also perform phrase matching. Another issues that can be considered here is the ordering in which the inverted lists are processed (query optimization) and skip lists.
- Scorer can be extended with link-based scoring schemes, proximity scorers, etc. At the moment the score values are not normalized by the implemented scorers, to be correct all the scorers has to return score values in the same range.
- Snippet generation and clustering may be improved to provide better performance and result quality
- Front-end may be improved to provide a better usability and performance
- Overall system performance with respect to processing time, disk and memory usage may be significantly improved. System profiling may be used to detect critical sections in the processing cycle.

The current implementation has only been tested on the provided version of TIME collection. In future the authors plan to index and evaluate the performance of **Solbrille**

on 500GB large TREC GOV2 collection. At the moment, as the statistics information is kept in memory, this is rather impossible. A redesign of some of the components may be important to be able to index and search in large document collections.

Also a number of advanced techniques such as caching (results, inverted lists, intersections, cluster data, snippets, etc), and distributed processing ([Jon08], [Ris04]) will be implemented and evaluated in future.

As all four of the group members have expressed their interest in working with **Solbrille** as a hobby project, the project will be hosted by the Google Code (`http://code.google.com/p/solbrille/`) and the authors may add and complete tasks they would find important and interesting for a future development. There is a hope that at some point **Solbrille** will gain significant performance or become a working product, otherwise it will be a good toy for testing ideas and concepts.

Bibliography

[Bjø07] Truls A. Bjørklund. Experimentation with inverted indexes for dynamic document collections, 2007.

[BR99] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, January 1997.

[Jon08] Simon Jonassen. Distributed Inverted Indexes. `http://daim.idi.ntnu.no/show.php?type=masteroppgave&id=4197`, 2008.

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

[Ris04] Knut Magne Risvik. *Scaling Internet Search Engines - Methods and Analysis*. PhD thesis, 2004. Chair-Edward A. Fox.

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

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[ZO98] O. Zamir and O.Etzioni. Web document clustering: A feasibility demonstration. In *ACM International Conference on Information Retrieval (SIGIR)*, 1998.

Appendix A

Appendix

A.1 How to run Solbrille, a short guide

This section contains a short introduction on how to use **Solbrille**. The guide will include the steps required to install, feed and search, using the **Solbrille** search engine.

A.1.1 System requirements

Solbrille uses functionality which is only accesible using Java version 1.6. Any modern computer capable of running Java applications should be able to run **Solbrille**.

A.1.2 Installation

The electronic delivery from this project contains a folder called `delivery`, copy this folder to wherever you want to have **Solbrille** installed. This folder contains the **Solbrille** binaries, the TIME collection and some content files used by the web front-end.

A.1.3 Initial startup and feeding

To feed the TIME collection to the system you should start the console **Solbrille** application. Place a command-line inside the deploy folder on your system and start the `olbrilleConsole.jar`.

```
> java -jar SolbrilleConsole.jar
```

Then, to feed the time collection enter the `feedtime` command. When the log message: `INFO: Flushed index phase 1` appears the index has been built. Then exit the console application with the `exit` command.

A.1.4 Running the web front-end

To run the web front end the `SolbrilleServer.jar` has to be run.

```
> java -jar SolbrilleServer.jar
```

Then point your favorite web browser to `http://localhost:8080/servlets/search`.