NTNU

Norwegian University of
Science and Technology

Phrase searching in text indexes

Asbjørn Alexander Fellinghaug

Master of Science in Informatics
Submission date: June 2008
Supervisor: Svein Erik Bratsberg, IDI
Co-supervisor: Øystein Torbjørnsen, FAST

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

Asbjørn Alexander Fellinghaug
fellingh@stud.ntnu.no
asbjorn@fellinghaug.com

Phrase searching in text indexes

Abstract

This master thesis focus at the challenges within phrase searching in large text indexes, and to assess alternative approaches to cope with such indexes.

This goal was achieved by performing an experiment, based on the theory of using bigrams consisting of stopwords as additional index terms. Realizing the characteristics within inverted index structures, we utilized stopwords as indicators for severe long posting lists. The characteristics of stopwords proved valuable, and they were collected based on a already established index for a subset of the TREC GOV2 collection.

In alternative approaches we outlined two "state of the art" index structures, specifically designed to cope with phrase searching challenges. The first structure - nextword index - followed a modification of the inverted index structure. The second structure - phrase index - utilized the inverted structure in using complete phrases as index terms.

Our bigram index focused on the same manipulation of the inverted index structure as the phrase index, using bigrams of words to drastically cut posting lists lengths. This was one of our main goals, as we identified stopwords posting list lengths to be one of the primary challenges with phrase searching in inverted index structures. Using stopwords to create and select bigrams proved successful to enhance phrase searching, as response times substantially improved.

We conclude that our bigram index provides a significant performance increase in terms of query evaluation time, and outperforms the standard inverted index within phrase searching.

Preface

This is a master thesis for the Master of Science program at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisors, Svein Erik Bratsberg and Øystein Torbjørnsen for valuable feedback, suggestions and ideas, as well as many entertaining talks. They gave me inspiration and assistance throughout this work. Also, I would like to thank fellow researchers and students at IDI for support. Also, a big thanks to my girlfriend for support during the numerous long working days. Also, the espresso machine next to my working hall for providing me with precious caffeine on those long days.

A big thanks to my parents for encouraging me to reach for a master degree, as well as the rest of my family.

Contents

List of Figures

List of Tables

1 Introduction

This project intends to evaluate different aspects of performing phrase searching in text indexes. That is:

- different index structures
- characteristics for indexing with respect for phrases
- characteristics for searching with respect for phrases

Also, this project will evaluate different ways of phrase searching, and briefly look at new approaches within this subject. Many research papers cover index structures, and their characteristics for efficient searching and indexing. This is mainly because of the more widespread understanding of the importance of search, and that it is absolutely necessary for businesses to embrace this technology. It is a highly hot subject, and therefore it is devoted a lot of research.

> *Search technology has proven it's importance in the modern world..*

The future is hard to predict, but one thing is certain - there will always be a need to search, both in business and personal life.

An increasing growth of information created every day needs to be searchable. That is because it may be of value by other people, but they do not know it since they are not aware of it. This poses some tough challenges for the search engines which is to index and provide a rapid search service for the people and businesses. We will explore some of these challenges during this master thesis - some in more depth than others.

This master thesis is organized into 9 chapters. Chapter 2 gives an introduction to the problem which this report will cover, in addition to limit the problem domain. Chapter 3 provides a whole set of concepts and definitions which is of great importance within the problem domain of this report. In Chapter 4 we will dive into the industrial search engine data structure called inverted index structure. We will explore the most known aspects and look at how this structure is used in relation to phrase query evaluation. We will also explore some state of the art material within phrase query evaluation in Chapter 5. Next is our experiment in Chapter 6 whereas we implemented a specialized inverted index structure to facilitate phrase query evaluation, and we shall thoroughly discuss the results. In Chapter 7 we will present and explore the results retrieved from the experiment. In Chapter 8 we discuss our bigram index vs the "state of the art" material presented in

Chapter 5. Finally, in Chapter 9, we will present our conclusion for the problem statement which this report is concerned, along with the results from the experiment carried out.

2 Background and problem statement

This master thesis is concerned with challenges within the search technology subject, with a extensive focus on phrase searching in large text indexes. We shall explore our initial problem statement along with an descriptive background for exploring this problem.

The background for this report is the increasing usage of phrases when performing searching, and the rapid growth of information which needs to be easily available for others. Now, performing phrase searches are expensive, as we will explore during this master thesis, so we need to look at different solutions to meet this problem.

We will explore the commonly used and acknowledged search engine *database*, namely the inverted index, in Section 3.4 and Chapter 4. Along with the inverted index, we shall also explore some other more special purpose structures in Chapter 5, which empathize the challenges introduced by phrase querying.

We will here first go through the problem statement and background, and then have a look at the problem definition. At the end we will briefly go through how we will take into consideration the ranking performed, and also the motivation behind this master thesis.

2.1 Problem background

The problem background arises from the realization that the growth of information is exponential nowadays, and all that information should be searchable to be of any value for others. Non-searchable information is only of value by the people knowing exactly where to find it. So, search engines index a lot of information that is made available, and the users need to query the search engines to find it. Now, if the indexed information is of middle size, then users could most likely retrieve only the interesting documents by a single term query such as “computer”. However, if the indexed information is huge (multiple gigabytes of text), the users would meet some difficulties. To comply with this obstacle the user needs to express a better query, so that non-interesting documents would be omitted. Such queries could be phrase queries, which has an growing user mass, according to [12] in their section of query properties. Phrase queries is briefly covered in chapter 3.2.1. These phrase queries have the benefit of better discrimination of documents than regular term queries, but as will be shown, comes with an additional cost.

2.2 Problem definition

The problem definition is mainly to evaluate a set of state-of-the-art techniques for searching and building efficient inverted index files which promotes phrase searching as well as standard term searching. These index files are efficient in terms of disk space, processing needs, and how well they actually fulfills the information need for the end-user. These are all aspects that we will later investigate into greater extent.

If we now dive a little deeper into this problem definition, there may be some clarification needed. An information retrieval system (IR) has to retrieve its information units fast - a delay for multiple seconds would be unacceptable disregardless of which query there is (in some scenarios also milliseconds is unacceptable). And, of equal importance is to retrieve relevant information in reponse to the query. If any of these aspects fails, the IR system would have a serious problem.

So, this problem definition is to compare the different ways of performing phrase searching, with respect to the aspects of response time, disk space consumption and other efficiency factors. We will more concrete look at specific approach, where we want to create a index based on pair of words as index terms. We believe such an approach would yield efficiency gains, and this is the foundation for our experiment introduced in Chapter 6. Based on the results provided by our experiment, and introduced in Chapter 7, we will hopefully be able to conclude whether or not this approach will yield any gains in the context of search technology and phrase query evaluation.

2.3 Models

All IR-systems rely on a background model which supports the intention of the IR-system and thus provides structures and techniques to support them. Common characteristics for all of these models is that index terms is identified words of semantic value from the document collection. Also of importance is that the models consider various terms of different importance, and the interpretation of words somewhat differ. Next we will introduce two very much discussed models in the search community.

2.3.1 Boolean model

The boolean model introduced in [22] is a information retrieval model which is based on boolean algebra. Its concept somewhat lies in the name of the model; it is based on binary weights of documents. The boolean model retrieves documents based on a simple match/no-match presumption. A basic search process in an IR-system built on the boolean model yields a boolean expression where query terms must be present to represent a hit. If none

hits are found, then nothing would be retrieved. If a (strict) match is found, then we have a hit and that particular document would be retrieved.

Some decades ago this model was highly popular in the search communities, but the drawbacks of the model was too substantial for it to survive in the field. The major drawback is its foundation; the binary selection of documents based on strict queries. By strict we mean queries where query terms must be present to represent a hit. This basics of the model does not yield a good retrieval performance, since documents which semantically would match based on the context would be discarded based on the binary search decision. Nowadays, when people talk about the binary model, most ones relate it with databases and strict data retrieval. And, this has its root in another drawback; the model is too data oriented, while the semantics are left out.

The boolean model works very well for databases where relevance and ranking of hits is not a necessity. One example could be searching in a repository for a document with a tagged code, identifier or something like that. For example search for a document containing the unique identifier code "ISBN 0-201-39829-X". There has been some attempt to enhance the boolean model, such as the *Extended Boolean Model* described in [22]. This enhanced boolean model implements one of the predecessors major drawbacks; the functionality of partial matching and term weighting. Despite this drawback elimination, the extended boolean model has not achieved any widespread usage.

2.3.2 Vector space model

The *Vector space model* is a model based on the realization that the *Boolean model* in section 2.3.1 is not sufficient in retrieval of documents based on a loose coupling between words and their semantics. The coupling is somewhat non-existing. The VSM model was introduced in 1975 by the information retrieval scientist Gerard Salton, and is thus not to be considered as a state-of-the-art model

The vector-space model introduces a new approach which allow partial matching between documents, instead of the regular boolean matching model described above in section 2.3.1 (either a match or no match; data retrieval). The way this is accomplished is that the VSM *(Vector-Space Model)* sees a document as a vector which contains one entry per unique term in the hole document collection (each dimension corresponds to a separate unique term). A query is also to be looked at as a document (hence a vector). Now we have our document collection which is a huge set of vectors representing one document each, and then we have a query which is also to be looked at as

a document. A highly common approach to calculate the relevance between a query and a document is to calculate the angle between the two vectors using the dot-product[1]. A formula for this calculation is shown in equation 1 in section 2.4. This formula is part of the ranking scheme frequently used in VSM based IR-systems which will be introduced below.

Theoretically the VSM has a drawback in that the index terms are assumed to be independent of each other, which may imply lack of semantically encapsulation by the model. If we think about it, then this drawback is the reason for many of the scalability issues the inverted index suffers from, which will be more clear in Chapter 4. In the VSM each index term is independent, and thus is indexed as a single searchable unit, means that we end up with a waste amount of data. Given that the VSM is enable to "perfectly" cope with this drawback, and that the model would recognize the dependencies between index terms, then phrase searching would be substantial better in terms of processing needs, disk space consumption and its ability to fulfill the users information needs.

2.3.3 Other models

Other models could be such as the *Probabilistic model* described in [22] and [14]. This model tries to cope with the information retrieval problems with an probabilistic framework. Another model could be the *Fuzzy Set Model* described in [22]. This model considers each query term defines a *fuzzy* set, and that each document has a "degree of membership" towards this fuzzy set. Often is a thesauri used in such IR-models as the fuzzy set model, since we in a thesauri can lookup broader, narrower and direct relationships between terms, and thus achieve the possibility to retrieve documents somewhat semantically matching the user query.

There is many more models that could be of value, but we will not consider them in this report.

2.4 Ranking

In the process of evaluating different approaches for searching, the question of the ranking scheme will occur. Now, this report will discuss and explore the subject of ranking schemes, since it is quite large and covers much material outside the scope of this report. So to clarify one assumption made in this report: we will always assume that the best possible ranking scheme is used, and will therefore achieve optimal ranking.

[1] `http://en.wikipedia.org/wiki/Dot_product`

However, since the subject of ranking is quite important within search technology, albeit a little out of the scope of this report, we will only briefly go through one very basic scheme. This ranking scheme lies within the widely used and acknowledge *VSM*[2] thoroughly discussed in [14] and [22], and somewhat introduced above in section 2.3.2. The reader is recommended to read the above section before proceeding here.

In VSM the ranking between a document and query is measured as the angle between the vectors. This measurement is very often calculated with the *dot-product*. The formula for the *dot-product* is outlined in equation 1. In this equation the d_j represents the document vector, and q the query vector.

$$sim(d_j, q) = \frac{d_j \cdot q}{|d_j||q|} \tag{1}$$

The dimensions in a document vector (each entry that is) contains the weight which that particular term has. This weight could be calculated in a number of ways. One way is to use the number of occurrences as the weight, so the more frequent the term is, the higher the weight. An alternative calculation which is widely used as a measurement for term weight is the *tf-idf*[3] covered in [14] [22] and introduced in [25]. *Tf-idf* stands for "*term frequency - inverse document frequency*" and is a statistical measurement which has proven its strength in calculating the importance of a term to a document in a document collection.

The *term frequency* is believed to represent that the more frequent a term occurs inside a document, the more relevant it is to the context of that document. The *inverse document frequency* is believed to represent a measurement for whether a term which occurs in many documents in the collection, is useful for distinguishing between a relevant and non-relevant one. So, this term weighting measurement needs to balance these two effects; *tf* and *idf*. In Equation 2 we present a possible approach for calculating the weight of an document entry (term). This calculation approach was introduced by G. Salton in [25].

$$w_{i,j} = f_{i,j} \cdot log\frac{N}{n_i} \tag{2}$$

Note that $f_{i,j}$ represents the term frequency within document j, N the total number of documents in the collection, n_i number of documents where the entry (term) occurs, and at last the i represent the term. So, equation 2 tells us that the term weight of term i in document j is derived from the

[2] `http://en.wikipedia.org/wiki/Vector_space_model`
[3] `http://en.wikipedia.org/wiki/Tf-idf`

term frequency of term i in document j, times the logarithm of the number of documents in the collection divided by the number of documents where the term i occurs.

Equation 2 is only one possible approach for the term weight calculation. In [17] Zobel and Moffat briefly consider other calculation approaches.

We will assume that the best possible ranking scheme is in use in this report. Note that in the Apache Lucene library (described in section 3.8) which we will be using in our experiment later on, the ranking scheme in use is *tf-idf*[4]. The author will encourage the reader to have a look at other ranking approaches such as Google's *PageRank*[5] thoroughly described in [18].

2.5 Zipf's law

Zipf's law is a model as described in [28] and [22] which tries to capture the frequency distribution among the terms within the document collection. In [28] it is stated that the frequency of the i-th most frequent word is given by the equation 3 times the frequency of the most frequent word in the document collection.

$$\frac{1}{i^{\theta}} \tag{3}$$

If we here assume that the θ value is equal to 1 (for large collections $\theta > 1$) we can state that the frequency of a word is inversely proportional to its rank, where rank is defined by the equation 3.

Position	Word	Frequency	Rank
1	the	6	1
2	of	4	0.5
3	to	2	0.33
4	and	1	0.25

Table 1: A very simple example of Zipf's law which captures the frequency distribution among some simple words. We assume θ equals 1 here.

In Table 1 is a very simple example where we have the 4 most frequent words sorted by descending frequency. In the column *Rank* we have calculated their $\frac{1}{i}$ where i is that particular words position in the sorted list of

[4] `http://lucene.apache.org/java/2_3_0/api/org/apache/lucene/search/Similarity.html`

[5] `http://en.wikipedia.org/wiki/PageRank`

words.

2.6 Heap's law

Heap's law is in the field of linguistics used to describe the growth of the vocabulary element in an inverted index structure (see section 3.4). Heap's law is thoroughly described in [11], and somewhat briefly in [22].

The main concept in Heaps law is that the more information that is gathered, the growth of new term discovery would decrease. In the beginning of an index process for a given IR-system the growth of term discovery would be substantial. However, the more text that comes in, the more of the terms would already have been discovered. Figure 1 clearly illustrates this observation in the Heaps law. The x-axis represents the text size, and the y-axis represents the number of distinct index terms in the vocabulary.

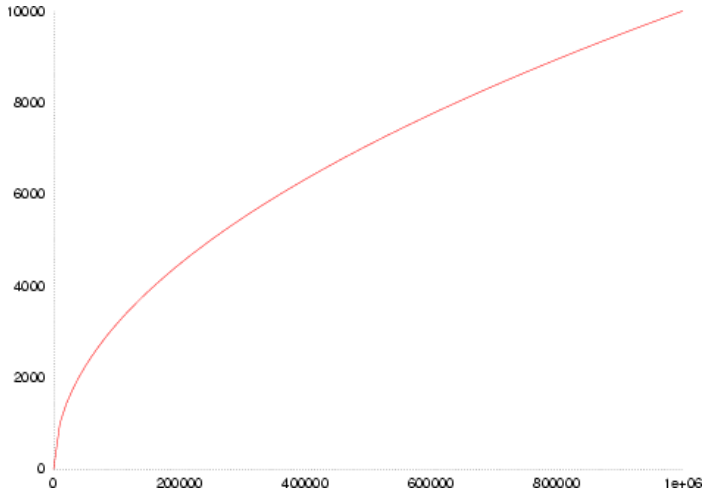

Figure 1: Illustrates a typical Heap's law plot diagram from Wikipedia [15].

This growth description provided by Heap's law can also be mathematically formulated, as shown in Equation 4. As can be noted there are two parameters, K and β. Usable values of these two parameters would highly depend on the linguistic characteristics of the document collection. In [22] and the article describing Heap's law in [15], recommended values for K is between 10 and 100, for β between 0.4 and 0.6.

$$V_R(n) = Kn^{\beta} \tag{4}$$

The implication of Equation 4 enables us to model the growth of the vocabulary, and thus the construction of the search engine can adapt to this

behavior.

2.7 Motivation

The motivation in this master thesis is (*the desire*) to enhance searching in general, but with specifically focus on phrase searching. Everyone has, one time or another, experienced the results given to them by a search engine that not completely fulfill their needs. There could be several reasons for this:

- the search engine finds no relevant documents for the user
- the results are presented in a non-user friendly manner
- the query gives too many results

The latter one is the most common one since the query does not discriminate enough to disregard unwanted documents, and thus poses a huge challenge for the search engines.

One approach that has proven its strength is phrase searching where you specify a certain phrase within quotation marks (Example: *"to be or not to be"*). Now, a certain phrase is much more discriminating than a set of terms disregarding the order. That is in the nature of a phrase query, it is a set of terms in an given sequence where the order matters.
One requirement with phrases is that the user which constructs the phrase has to have some knowledge regarding the information the user seeks. That is, the user need to know that a specific phrase occurs, or has a high probability for occurring in documents which encapsulates the wanted content. In a very specific phrase such as "Search engines need to evaluate queries extremely fast", the user has some knowledge about the wanted documents. In a more vague phrase query such as "search engines" the user has much less knowledge, but is still able to construct a phrase query based on the information needed. It is all about how the user expresses its information need with regards to the background knowledge of the user. The more discriminating the expressed information need (query) is, the better results in terms of relevance and precision is achieved.

3 Concepts and definitions

This chapter will clarify some concepts that we will use throughout this report, and that we think would be of great importance for you. Also, we will look at some important definitions.

We will not go deeply through these concepts and definitions since most of them will be more deeply explored later on. For an even more in-depth information regarding these concepts and definitions, we would recommend you to have a look at our bibliographic list at the end of this report.

3.1 N-gram

An *n-gram* is by definition a subsequence of n-items from a sequence. The items can here be letters, words, numbers, etc. By this, we can say that the sub-sequence "or not" from the sequence "to be or not to be" is a *n-gram*. Different kinds of *n-grams* have received its own names:

- **unigram** A *unigram* is when the n-gram size is 1. That is, there is only one item in the sub-sequence.
- **bigram** A *bigram* is when the n-gram size is 2. Following the previous pattern, there is only two items in the sub-sequence.
- **trigram** A *trigram* is when the n-gram size is 3. As previous, there is now only three items in the sub-sequence.
- **n-grams** A *n-gram* is when the n-gram size is 4 or above. That is, there is 4 or more items in the sub-sequence.

In [2], there is discussed the early usage of n-grams in regards to term-based search techniques. The usage of n-grams was introduced to provide resilience to noise such as misspelling. The idea was to decompose terms into word fragments of size n, then perform matching based on these fragments to determine a match if it exists. In this report, we will not deal with n-grams of letters (characters) such as these fragments, but instead concern ourself with n-grams based on words. N-grams based on words is a combination of words which are near each other in a sentence, or semantically related such as "computer science", or "in the". These two bigrams (size equals two) is a combination of two words, and the first one represents a very semantic meaning in contrast to the two terms "computer" and "science" by themselves. The bigram "in the" does not bring much semantic, and is purely constructed based on their ordering in the sentence. We will explore n-grams features (with a focus on bigrams) more in our experiments in Chapter 6.

3.2 Queries

A query is a representation of an information need by a user. The user want to find some kind of information, and then expresses this quest in a particular representation, which is by definition a query. So, queries are basically textual representations of information needs.

There are however different kinds of representations for different kinds of information needs. Some needs are expressed by single terms such as: *car 1971 pontiac lemans*. In this query there is four terms which is an expression for the information need. The user wants to find documents which contains information about 1971 models Pontiac LeMans cars, and not cars in general or the "American lemans sport car race series". This query would most likely not yield any good efficiency in terms of relevant documents. Note that what is here referred to as a term query is also referred to as a keyword query (multiple terms) in other literatures, whereas a single term query denotes a query of one term.

Another representation could be: *"Pontiac LeMans 1971"*. In this query we specifically search for documents with the terms "Pontiac", "LeMans" and "1971" in that particular order. Other representations can be a user's desire for finding pictures of the car, or finding a video sequence of the car in action. These are more advanced ones, and is part of *Multimedia Information Retrieval*[6] which will not be discussed in this report. We will have a closer look at two kind of queries here, along with the familiar Boolean query. Note that each query type here imply the usage of ranking, in that the final result list would be ordered by the score each hit is assigned. A boolean query often signalize a strict usage of boolean operators to locate documents, however there is assumed that boolean queries also take ranking into consideration.

3.2.1 Phrase query

A phrase query is basically a simple term query expressed as a phrase, and a phrase often carries more semantic encapsulation than a term query. For example, say that you want to find the famous quote from *William Shakespear's, Hamlet*: "to be or not to be". Here is two examples:

1. boolean query: to be or not to be.
   This query will tell the search engine to look for all documents which has these terms in it, regardless of where they occur next to each other in the document. Some search engines may also find documents where not all terms occur. If no boolean operator such as `AND` or `OR` is given,

[6] Multimedia IR is a fairly large subject, handles typically audio, images and video.

then the search engine has a predefined operator set. This default operator differs somewhat depending on the search engine.

Another thing to notice here is that the terms are all typical stopwords. For these reasons, the hit list of this particular query would be huge. A test at `http://www.google.com` gives us 592 000 000 pages, and a test at `http://www.yahoo.com` gives us 7 300 000 000 pages. The waste difference here is because of different approaches to handle such queries. We think Google use implicit `AND` boolean operator between each query term, thus shrinking the result set. Also, each search engines internal index vary in size which would also impact the result set.

2. phrase query: "to be or not to be."
   This phrase query will tell the search engine to look for this particular arrangement of all the terms in the query; the phrase. It will never look for pages with only a subset of the terms "to, be, or, not, to, be". A test at `http://www.google.com` gives us 2 710 000 pages in return, and `http://www.yahoo.com` gives us 2 560 000 pages.

These two examples clearly show the importance of phrase queries, and their impact on the results from the search engine. We will explore similar results later on.

3.2.2 Term query

A term query is, in contrast to phrase queries, a simple query based on a semantic and discriminating term (one word). The query could for example be as simple as: "*Computer*", without the quotation marks. The issue with term queries is that you need to use quite discriminating terms to get good efficiency (*recall / precision*) scores. Now, this may work very well within relative small collections, but in case of a huge collection (like the web?), then we will have problems.

A very simple example could be to query for "*computer*", which in Google gives us 1 190 000 000 hits, and Yahoo gives 3 190 000 000 hits. Both result sets are vastly large since the term query is not particular discriminating and the total size of the collection (the web?) is enormous.

3.2.3 Boolean query

A boolean query is a simple query combined of multiple query terms separated by a boolean operator such as `AND, OR, NOT`, or whitespace indicating the usage of a default boolean operator. Most search engine use the `AND` operator, and thereby demand that all the query terms must be included in the document to indicate a match. There is also techniques where the `AND`

operator is used if there is located hits, and if not sufficiently enough hits is located the `OR` operator would be used. This technique introduce a sort of ranking, meaning that the first documents in the result set has all the query terms, and those further down the list could be based on a subset of the query terms.

Lets take a look at two very simple examples:

1. boolean query: computer science
   This boolean query gives us a total of 193 000 000 hits with the usage of the Google search engine. The field of computer science is very large, so we are not surprised by the waste number of hits found.

2. phrase query: "computer science"
   This phrase query gives us a total of 88 100 000 hits with the usage of the Google search engine. Notice the difference between this and the previous term query, which is quite significant. However, even with the usage of a phrase query for "computer science" the hits is fairly large, so considering extending the phrase query would narrow down the result set. In Section 3.2.4 we shall see additional approaches to address this problem.

This clearly show the differences with the usage of boolean and phrase queries. As already stated, there lies a difference in the semantic encapsulation between boolean and phrase queries. Basic boolean queries carry only the semantics of the query terms itself, regardless of the semantic relation they may share. The term(s) within a boolean query is not directly related (unordered), and therefor the query will have a more vague semantic. A phrase query however, is by nature carrying the semantics of all the terms in the ordering they have. A phrase query like "new york" differs substantially from the boolean query "new york" (without the quotation marks). Even worse would have been if the search engine interpreted the boolean query with an `OR` operator, thus including documents which also contains one of the query terms.

3.2.4 Combined queries

This section is merely to introduce the concept of combined queries, where we by "combined" mean the usage of different query types to express a more precise information need. The introduction of this concept has become more visible since the information amount in web search engines is so enormous, and the users need new ways of express their information needs. So, these queries are simply different types of queries which can be expressed as a sort of boolean query where each query term is in fact another query type.

Take this combined query as an example:
`computers "python programming" norway`. Here we have a boolean query consisting of three query terms, whereas each query term is in fact another query. The first term is a single term query. The second is a phrase query consisting of two ordered terms, and the last one is a single term query. This combined query is better illustrated in Figure 3.2.4, and note that there is assumed a default boolean operator such as `AND` or `OR` between each query term if just whitespace is used..

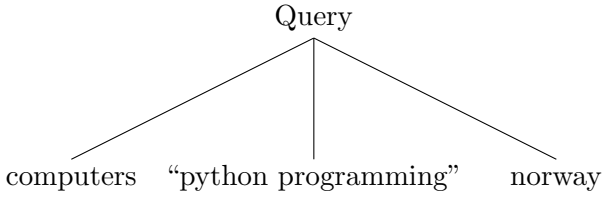

Figure 2: A hierarchy showing a combined query, where each node has an implicit boolean operator such as *AND, OR, NOT*

3.3 Recall and precision

Recall is an IR performance measure which represents the fraction of relevant documents in a set of retrieved documents. Let R represent a set of relevant documents, and $|R|$ the number of documents in R. Now, assuming a answer set A is retrieved in response to some query, and $|A|$ the number of documents in A, and that $|Ra|$ represents the number of documents in the intersection of the two sets, R and A. Then we have a formal calculation of recall illustrated in Equation 5.

$$Recall = \frac{|Ra|}{|R|} \tag{5}$$

Precision is another IR performance measure which represents the faction of retrieved documents that is relevant. Relevant would here be in context of the query. In Equation 6 we see the formal calculation of precision, whereas we follow the same notation as for recall.

$$Precision = \frac{|Ra|}{|A|} \tag{6}$$

3.4 Inverted index

An inverted index (also called *inverted file*) is a simple structure used to index a text collection in order to provide fast searching capabilities. It is a

combination of two elements, the vocabulary (also called the index) and the occurrences (also called posting lists).

3.5 Signature File

A signature file is another index structure which was very much used before the inverted index became available. A signature file is basically an index structure based on hashing which represents the "signature" in the sentence "signature file". The hash function in a signature file is the function which maps words to bit masks. An example could be that the word "computer" maps to the following bit mask → `01000111`, that is that `h`("computer") = `01000111` where `h()` is our hash function. In [14] and [22] there is an extensive introduction of signature files which goes through the structure, searching and constructing capabilities. Just to make you clear on what signature files really are, we will briefly introduce the underlying structure and some characteristics.

The hash function maps words to bit masks. Now, a single text document has its own signature file constructed by hashing all the words occurring in the document. First in the construction phase the text is to be divided into blocks. Now, the document is a set of blocks which in turn holds all the words in the document. Each block in the document gets its own signature by bitwise OR-ing together the hashes from all the words inside that particular block. Now we have a signature file which is composed of the sequence of all the text block hashes.

The searching process in a signature file is very straight forward. First we get the query where we perform hashing on each query term, and then bitwise OR-ing them together into a bit mask which represents the query. And then we compare the query bit mask with all the text blocks bit masks. Assuming W represents our query bit mask, and B_i the bit mask of text block i, then the comparison statement can be written like this:

$$W \& B_i = W \tag{7}$$

Note that the & is the bitwise AND operation. So, if all the bits set in W are also set in B_i, then we *may* have a hit. Note that we say *may* have a hit, since there is a probability of a *false-match*. The reason for this *false-match* is in the nature of the signature file structure. When we construct a text block hash, then our hashing function could construct a hash which has the similar bits set as in another word hash. The likelihood of this occurring is related to how much overhead you are willing to sacrifice in the signature file. The less likelihood of a *false-match* indicate a large signature file, since the hashes for each text block would need to contain more bits.

Signature files has some interesting characteristics, such as that they are more efficient the larger the query (in number of terms that is). For single term queries, the signature files are not efficient. According to [22] signature files improve phrase searching, and is the only indexing scheme which does this (disregarding the state-of-the-art indexing scheme which is discussed in Chapter 5). However, signature files are only efficient with relative small document collections.
One important observation derived from many acknowledge books and articles such as in [14, 22, 17], is that inverted indexes almost always outperforms signature files. This observation is explained in that signature files is likely to yield a lot of *false-matches* which has to be sorted out by retrieving the documents from disk.
To minimize the risk of *false-matches* one can increase the amount of bits used in the bitmasks, however this imply larger index and more data to retrieve for each search. Of even more importance is that ranking is not supported. This restriction in contrast to the inverted index makes the signature file approach less appealing. Also complex queries with operators such as disjunction or negation is hard to support.

3.6 Hybrid solutions

Hybrid solutions in regards to indexes is typically different kind of combinations of index structures. The motivation for hybrid solutions is to support various types of queries, such as:

1. term query
2. boolean query
3. phrase query
4. similarity query
5. image query (multimedia types)
6. ...

For term- and boolean queries there is usually the standard inverted index which promotes this kind of use. For more advanced ones such as phrase queries, a more enhanced index that supports rapid search based on phrases could be used, however a standard inverted index would also fulfill the task. Other examples could be queries which has special characteristics, such as a very long boolean query with one or more stopwords. These kind of queries would be very expensive to execute in a standard inverted index. However, if we have some form of analyzation for all incoming queries, and then send queries that performs best on a standard inverted index to that index, and queries that may be rewritten to phrases to an phrase enhanced inverted

index, and so on. This basically introduces the concept of implementing a kind of routing mechanism, which maps each query up to the index type which it performs best on.

Using the approaches described above, there is a possibility of saving processing speed in form of lookup time in the index. This approach is very much found in larger search engines nowadays. In [23], the author emphasize that analyzing (linguistic techniques) queries and possibly rewrite them to better suit the index in use, is done nowadays. In [12] the authors look into the subject of using hybrid solutions to provide a fast searching process in terms of phrase querying.

In Chapter 5 we will also go a little deeper into hybrid solutions, as it is an important subject. We will also look at different example approaches towards hybrid solutions that could be used, based on the introduced index types.

3.7 Document analysis

Document analysis is the process of analyzing the content of documents (also called "document preprocessing"). This is a core process in information retrieval systems, and has received a lot of scientific research over the past decade. If we have poor document analysis in an IR-system, it will have an impact on the search effectiveness since the indexed information will not reflect the information in the document collection in a good semantic way. Not being able to encapsulate the document collection information in a search engine makes the search engine worthless.

So, what lies in this core process which is of such great importance for an IR system? Document analysis is composed of text operations which are used to extract and manipulate the content of documents in such a way that the semantics are encapsulated in the index. These text operations are often referred to as a form of NLP (*Natural Language Processing*). You can see a very simplified graphical illustration in Figure 3.

Text operations such as *stemming*, *stopword* elimination, index term *selection*, as illustrated in Figure 3, is a very common approach to document analysis. In [22] Baeza-Yates and Ribeiro-Neto gives a simple yet broad introduction to such text operations. The following three subchapters (3.7.1, 3.7.2 and 3.7.3) will cope with these text operations, and we would like to point out that these operations are not the only ones who are being used in document analyzing. Others worth to mention is the usage of thesaurus to support index term selection in a much broader perspective, and the usage of n-grams (chapter 3.1) to encapsulate the concept of the document (i.e. to

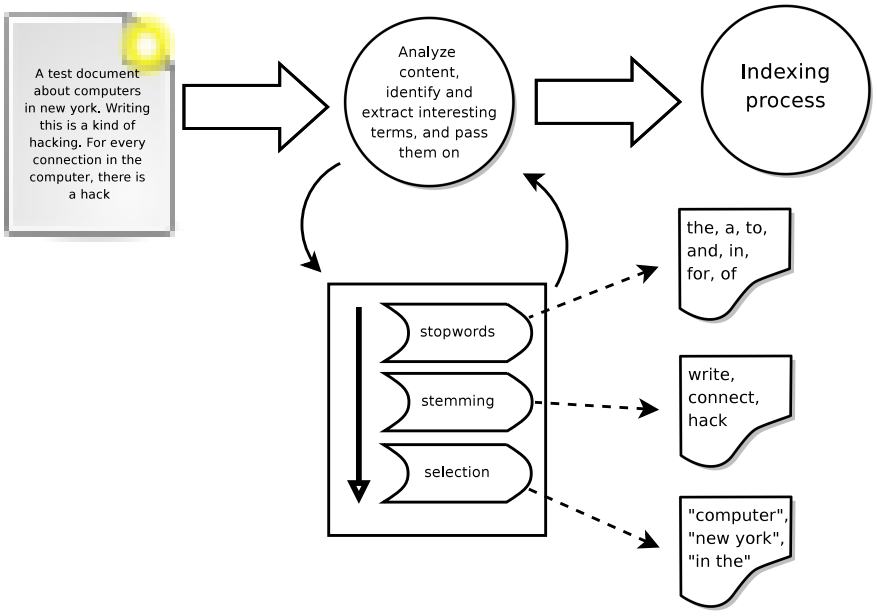

Figure 3: Illustration of a typical document analysis process

model the document).

3.7.1 Stemming

In a user query there is often a combination of multiple words which together forms the information need (the query). These variations between words in different queries often share common characteristics, that is the same root word with syntactical variations (grammar). In a very simple analysis process the analyzer would make an distinction between the word "computer" and "computers" (plural), which would construct two distinct terms, while the concept of the word is basically the same. Now, if we consider a document of many thousands of words in which a big portion is the same basic word with syntactical variations, then it is easy to see that we would waste vocabulary space if we did not consider the common concept they represent. This root word is referred to as the *stem* portion of a word. The *stem* is what is left after we remove the *affixes* (*prefixes and suffixes*). Let us have an example:

hack *hacking hacked hacker*

Here the word *hack* is the *stem* (the root word) of hacking, hacked and hacker. So, we basically save three index terms in the vocabulary here, ergo the indexing structure will occupy less storage space. However, despite these

apparent advantages, most search engines nowadays do not adopt stemming in their document analysis process. The reason for this is that researchers have come forward with some conflicting conclusions regarding the usage of stemming. Especially a researcher named *W. B. Frakes* did some research within the usage of stemming, and the results he came up with was inconclusive. [22] describes into more extent what was included in the experiment performed by *W. B. Frakes*.

3.7.2 Stopword elimination

The definition of a stopword is a frequent word that carry little semantic on its own, and thus is of little value to any IR-system. A high frequency for a stopword also means that a query for a stopword would eventually result in a very large result list. In the English vocabulary we have for example the *very* frequent word "the", which occur in almost all English documents. The number of documents retrieved for a query like "the" would be $|N|$ (*often even much larger*), where N is all the documents in the collection. We can therefore safely tag this word as a stopword based on its frequency, and for the reason that it does not bring any meaning by it self. In a sentence it brings meaning, but not on its own. In Figure 3 you can see how the stopword elimination has identified stopwords in the input document(s), and thus discarded these words from being put into the index structure.

In an IR-system when there is an index which holds the vocabulary and the posting lists, there would be a very skew balance in the posting lists. This is because of their nature - each posting list holds the occurrences of a particular word in the whole document collection. Given a stopword, which would occur quite frequently, the posting list for that particular word would be huge compared to a low-frequent one such as the word "TREC".
If we applied stopword elimination in the document analysis phase, we would enable large space savings in the index. In [2] and [22] there is claim to obtain a compression in the inverted index structure by 40% or more (in disk space savings). These savings are optimization factors that should not easily be disregarded in IR-systems, since I/O traffic is usually a bottleneck. Some disadvantages with the usage for stopword elimination is that it may reduce Recall (Section 3.3) in the context of IR-systems.

As mentioned, stopword elimination will according to [2], [14] and [22] reduce recall in an IR-system. This disadvantage is very obvious when we look at the query "to be or not to be", where all the words except maybe "not" and/or "be" will be stopped. This query would yield a very bad recall. As elegantly stated in [14]: "the index creator is by stopword elimination anticipating that future users will not be interested in these terms". For this reasons, most search engines nowadays do not adapt stopword elimination.

It could be of interest to mention that search engines nowadays adapt auxiliary structures to cope with the stopword overhead in the index structure. These auxiliary structures will be explored in Chapter 5.

3.7.3 Index term selection

Index term selection is the process of identifying and selecting good index terms which encapsulate a comprehensive model of the document being analyzed. Quoting this process in this simple sentence above makes it sound very easy and straightforward, however that is not the case. A straightforward approach would be to identify all words (terms) in the document, and then let them represent the content of the document. Now, given a query for `internet cookie` could then match documents that describes food recipes for cookies which is available on the internet, when the real intention was to find documents regarding the internet information cookie used in web browsers. This is an good illustration of semantic flaws which follows a straightforward approach.

Other approaches in index term selection could be to extend the single term identification algorithm above to also include infrequent terms which are semantically related to the terms in the document. That is to extend certain terms which have a semantic value with others synonyms or related words. Techniques such as the usage of thesaurus is commonly used in these approaches. Also manual annotation of related terms (also called keywords) could be used, however not in the same degree since it is expensive in contrast to an automatic approach.

In [22] there is described an approach where noun are identified as the semantic carrying terms, and thus only they are identified and included in the index term selection process. Terms such as verbs, adjectives, adverbs, connectives, articles and pronouns are simply discarded based on their lack of semantics. Term clustering is also used in cases where nouns appear nearby each other, so that they encapsulate the concept of that particular sentence or context. A very good example of this is a sentence which include the nouns *computer* and *science*, which is clustered into *computer science*.

3.8 Apache Lucene

Apache Lucene is an open-source search and indexing technology framework, and has become quite popular within the last couple of years. The name "Lucene" comes from the creator wife's middle name[7]. There was in the

[7] Doug Cutting is the original creator of Lucene.

beginning only one implementation in the Java programming language[8], so this is the main language which the framework is available in. There now exists several ports to other programming languages[9].

Apache Lucene is not a single search application that we can just setup and run. It is a complete search engine library which contains all the necessary functions to both index and search a document collection. With this library one can create a search engine which comply with user defined requirements and other special needs. The library is very simple to use, and is despite its simplicity very fast and efficient. The library provides out-of-the-box default values so that one can quickly be able to index and search. However, it is quite simple to extend the library to enhance indexing performance, search domain, document analyzers, etc.

In the introduction of our experiment later on, we will further explain the power and beauty of Apache Lucene. We shall also explore Lucene performance, both in terms of indexing and searching with a focus of enhancing phrase searching capabilities.

3.9 TREC GOV2 Collection

The TREC[10] Collections is a standard "testbed" to judge information retrieval systems. The collections provided (called "tracks") are each designed for their own utilization domain; Web, Blog, Enterprise, Legal, Genomics, etc. The track we use in our experiment is the *Terabyte Track*, which is based on a web data crawl from web sites in the `.gov` top-domain.

This *Terabyte Track* is also called the GOV2 collection. The GOV2 collection is a crawl that was performed in early 2004, and grow up to 426GB in size and contained as much as 25.2 million documents (web pages). In the IR-community it is quite common to do benchmarking based on this collection, and so have we done in our experiments.

[8] Java programming language. `http://java.sun.com/`

[9] Look at `http://wiki.apache.org/lucene-java/LuceneImplementations` to find out which programming languages Lucene is ported to.

[10] Text REtrieval Conference (TREC). `http://trec.nist.gov`

4 Inverted Index

The inverted index is a very much acknowledged information structure which promotes fast term lookup. The reader is recommended to first read Chapter 3.4 for a brief and understandable definition of an inverted index. Basically an inverted index is a data structure whose primary goal is to simplify and enhance the search task of a text document collection. Recall from Chapter 3.4 that the inverted index is a combination of two elements:

- the *vocabulary (index)*
- the *occurrences (posting lists)*

The first element, the vocabulary, is a list of all the unique terms in the text document collection. In Figure 4 on page 34, the leftmost box illustrates this element. The other element, the occurrences, is a set of lists where each list is related to one unique term in the vocabulary. Inside these lists there is what we call postings (hence the lists is referred to as *posting lists*).

The postings represents pointers to all the occurrences of that particularly term in the text document collection. A term in the vocabulary has a pointer to a posting list in *the occurrences* part of the inverted index (as graphically illustrated in Figure 4). Each entry in a posting list typically contains a unique document identifier (denoted as d_j, where j is the unique document identifier), along with the term frequency within that document (usually denoted as $f_{i,j}$ where i is the term and j the document), and maybe also a list of positions to where the term occurs inside the document. In Figure 4 we clearly see the pointer from a term in the vocabulary to its posting list in the occurrences.

Just to make the concept of a posting list simpler, we have a denoted a very simple posting list for an imaginary term. Table 2 is a very simple posting list where each entry is another list (vector) where we typically have three items: document identifier, term document frequency, and a list of occurrence positions within a document. In the posting list the i is the term id and j is the document id, and the list items is sorted increasingly based on the document identifier j.

entry 1	entry 2	entry ...	entry j
$< d_{i,1}, f_{i,1}, [3, 19, 43] >$	$< d_{i,2}, f_{i,2}, [23] >$	...	$< d_{i,j}, f_{i,j}, [59, 88, 102] >$

Table 2: A very simple posting list. i is the term id, j is the document id. List items is sorted incrementally based on the document identifier j.

The document id represents the pointer to which document that a particular term occur in. The posting lists in an inverted index is by definition also

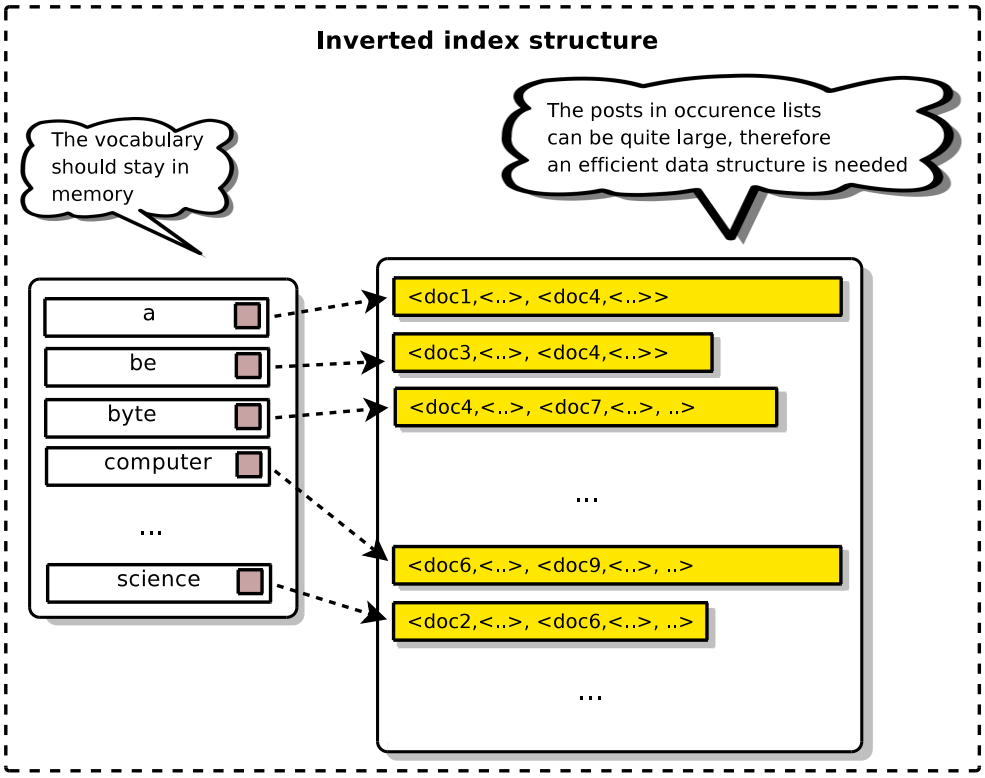

Figure 4: Standard inverted index structure where the two core elements and their relation is in focus.

referred to as an inverted list. An inverted list is built up from the presence and positions of a term in a collection of documents, so it is basically a natural inversion from the text into a list of words and their occurrences. An example can be the inversion of Table 3, which represents a set of documents where each row is a document, that would produce Table 4 which is the (simplified) generated inverted index. A thorough description of these core elements in the inverted index structure can be found in [14, 22] and [2] along with comprehensive illustrations and examples.

If in-document positions is being used (in posting list entries that is), then a given position there either refers to a word or a character. Most search engines nowadays dealing with text documents use word references, not character[11]. The index type search engines often used is said to be a word oriented index. A document oriented index type however, does not store word positions, but only document identifiers. Look at the example posting list above without the extra list within each post entry to denote the positions.

[11]Bioinformatics (genetic) search engines may use characters references

Documents	Words
doc 1	searching is for 1337 people
doc 2	programmers, also know as codemonkeys
doc 3	1337 people is codemonkeys

Table 3: A table of documents and their content. This type of table is in some circumstances referred to as a forward index

Words	Documents
1337	<doc1>,<doc3>
also	<doc2>
as	<doc2>
codemonkeys	<doc2>,<doc3>
for	<doc1>
is	<doc1>,<doc2>,<doc3>
know	<doc2>
people	<doc1>,<doc3>
programmers	<doc2>
searching	<doc1>

Table 4: A table of terms and their occurrences. The rightmost column can be looked at as an inversion of Table 3, and the hole table is a very simple inverted index structure.

In document oriented indexes it is difficult to support proximity or phrase queries, since there is a need to retrieve the document text online and then sequentially scan it to locate phrases or proximity matches. This will in most cases add a substantial time delay in an already strict time schedule because of the I/O delay of fetching the document. It would also implicate a significant storage overhead since all the documents needed to be stored in their original condition. By original condition we mean before it has been piped through the "Document Analysis" process described in Chapter 3.7. In word oriented indexes the search engines has the advantage of simply fetch postings lists and find matches based on the word positions. In case of phrase queries, multiple posting lists must be fetched and then merged together based on the order of the query terms and the word positions in the posting lists. This is a whole lot simpler (with phrase and/or proximity queries in mind) with a word oriented index than with a document oriented index, and is also the preferred way of supporting these kinds of queries.

It should be noted that the document oriented index structure would occupy much less space than the word oriented. This is because the document oriented index only needs to record the document identifier, not the term positions inside the document, thus saving a whole lot of pointer space. In

cases where we have stopwords like “the”, which is in almost every document in the collection, we would save ourselves as much pointers as there is documents. If we have 1 million documents who all have the term “the” inside, then we would at least save ourselves 1 million pointers in the posting list of that particular term. In most cases, the term “the” would occur multiple times inside a document, so we would actually save ourselves much more than 1 million pointers overall. Despite this seemingly disadvantage with the usage of word oriented indexes, this is widely in use due to its simple support of phrase and proximity queries.

4.1 Resource usage

The inverted index resource performance has been deeply explored in [2, 14, 22, 23] where the focus is mostly on the occurrences part, since it is the most resource demanding. Of great importance is balancing resource performance versus search speed performance, and deciding which sacrifices should be made in this process. It is widely known that there is limited main memory available in a computer, and thus great care should be taken with regards to what is to be stored there.

We want to utilize all main memory available such that data which is frequently accessed, is available without the delay of external devices. One may think that the internal OS cache strategy might do this for you, and that is correct. However, often this approach is not sufficient. The vocabulary is something we would want to put in the main memory, if there is sufficient space though. Also caching of posting lists which would sufficiently speed up the search task is of interest here. Approaches and experiments regarding caching of results and posting lists is widely explored in [16], and a set of interesting architectures of different levels of caching is being discussed.

Lets consider the first element in the inverted index structure; the vocabulary. The vocabulary is only a collection of all the unique terms in the text document collection. In addition to the terms, there is the need to store the pointer to which posting list the vocabulary term is related to. Also, in order to reduce I/O traffic towards slow external storage devices and thereby save significant main memory, it is normal to include the overall frequency for each term in the vocabulary. The benefits of including the frequency is gained in cases where multiple query terms are present.

Given the phrase query “howto search”, we have two unique query terms; “howto” and “search”. Now, inside the search engine, it is highly attractive to locate which query term that has the lowest frequency (rarest term), because the higher the frequency the larger posting list needs to be fetched

and decoded from the occurrences. The phrase query term "howto" gives 34 300 000 hits with Google, and "search" gives 5 250 000 000 hits. Clearly the term "howto" needs to be fetched first in light of this ratio, and the entries are decoded and a temporal structure is constructed. Then the second rarest terms posting list is fetched: "search". A merge process will begin that will merge the entries in this posting list against the temporal structure. Only entries in the temporal structure that has a match in the posting list will be leaved untouched, the rest will be discarded. So, this merge process only purge entries from the temporal structure, it *never* appends. The savings here are quite significant in terms of main memory, and also in I/O traffic. If a high frequent term had been processed first, then the temporal structure would have occupied more space than available (swapping to disk). The phrase query above has a total of 19 400 hits according to Google. This hit count differs somewhat from the frequency of the query terms alone. If we had chosen to process the query term "search" first, our temporal structure would have occupied 5 215 700 000 more entries than with the query term "howto". Just as a little sidenote: it is unlikely that the temporal structure in this example would hold all the entries. Only a subset of the most relevant ones based on the query ranking, since the user would never browse all the 19 400 hits at once.

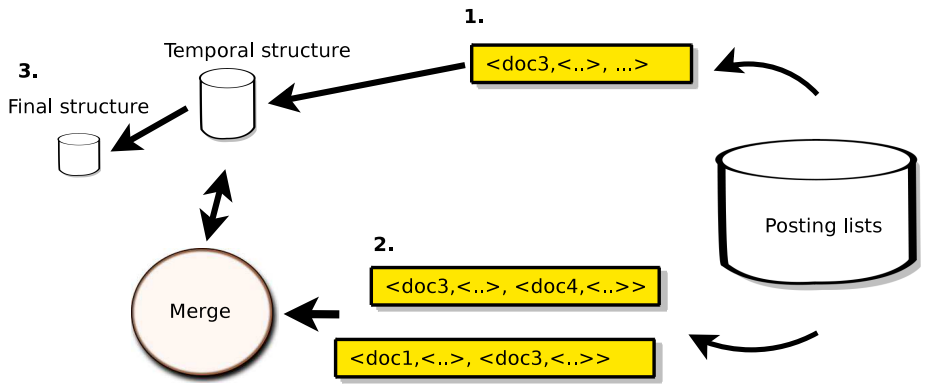

Figure 5: Illustrates query term evaluation with the fetching order of posting lists and the merge process into a temporal structure.

In Figure 5 there can be seen the fetch and merging of posting lists such as described here. The operations performed in each number in the figure can be summarized as:

1. Fetch posting list of rarest query term, and construct temporal structure.
2. For remaining query terms; fetch posting list and merge with already constructed temporal structure.

3. Output final structure where all entries has a hit based on the query.

Term	Frequency	Pointer
a	983 342	(disk access pointer)

Table 5: A simple illustration of an item in the vocabulary.

A simple example of an entry in the vocabulary can be seen in Table 5. Here we simply have a term stored in its ordinary form (a sequence of characters; a string), a (imaginary) term frequency (integer), and a pointer to its posting list. Since the term in the vocabulary is unique, it is usually stored as it is (*compression could have an impact here; see Section 4.4*).

The term frequency is very unbalanced in the vocabulary as a consequence of the indexing of low- and high frequent words (and maybe stopwords), and may vary between 1 to as many documents there is or more, depending on whether the frequency counts document hits or in-document hits. In Section 4.4 we explore this kind of behavior and look at how we can use compression to enhance both the searching and indexing. The pointers in the vocabulary will need to address all the posting lists in the occurrences element of the inverted index structure. Since there will only be as many posting lists as there are terms in the vocabulary, then a pointer will usually be 32-bits.

The vocabulary is stored using some sort of clever data structure such as a *B-tree*[1], *Suffix tree* [22], *Hash table* or a simple sorted list. The issue is the lookup speed in finding the terms, so in case of *B-tree*, *Suffix tree* and *Hash table* there will be an additional space overhead associated with the vocabulary structure.

Lets now consider the second element in the inverted index structure; the occurrences. The occurrences which holds the posting lists is the space consuming part of the structure. We especially talk about disk space usage, and therefore also optimal CPU utilization based on the time-consuming task of fetching these posting lists. A decade ago there was a limited amount of CPU cycles available in computers, hence a focus on saving processing was quite imperative. Operations such as decoding[12] the posting lists required a lot of CPU power, and was by that time very expensive in these terms.

One technique that proved to yield substantial gains for those computers was the skipping [20]. This technique will not be discussed into detail here, but its main goal was to provide an additional "index" on top of a posting list.

[12]Typical decompress and sense/understand the information

This "index" reduced the decoding step, which in turn saved much processing time for the CPU. Nowadays however, the CPU is not the bottleneck[13], but the disk is. To fetch posting lists from a disk requires multiple disk accesses which comes with a price, namely time. Disks have not gone through the same development as the CPUs, and therefore suffers from slow access rates compared to operations within the CPU. This skipping technique is better described in [20, 14], and somewhat in [5, 12].

A posting list for a particular term is a set of entries. Each entry encapsulates the document identifier, the term frequency and the word positions. This posting list is sorted based on the document identifier, so that the first entry in the posting list denotes the smallest document identifier match for that particular term, as illustrated in Table 2. For each next entry, the document identifier is guaranteed to be larger than the previous one. When evaluation of ranked queries the posting lists for the query terms might yield some difficulties due to its size, and that it is sorted based on the document id. Given a very large posting list, the list has to be retrieved and decoded. This is because it is necessary to calculate document weights and then rank the matches in order to find hits based on relevance.

In case of ranked query processing, we would like to only retrieve and decode entries in a posting lists that is likely to be relevant, thus discard the rest. Techniques for such an list reordering to promote this capability could be *frequency-ordering* and *impact-ordering*. These techniques are somewhat described in [5], and in much more detail in [17]. When considering ranked phrase query processing however, it is not so simple to predict which occurrence of the term which will be in a phrase query. Thus such techniques as list reordering is not considered to be effective, and therefore discarded. In phrase queries, processing word positions inside a posting list entry only needs to be decoded in case of a document match. If no document match is in place, the processing will move on to the next entry until the end of the posting list is reached. The whole posting list needs to be retrieved from a disk.

4.2 Inverted index construction

The construction of an inverted index structure is actually a simple process, to some extent described in the beginning of Chapter 4. Outlined in [17] is the single key challenge with index construction that the data volume involved cannot be held in main memory, and thus requires additional techniques which facilitates inversion of such large data volumes without the need to store the whole temporal index in main memory.

[13] `http://en.wikipedia.org/wiki/Bottleneck_%28engineering%29`

As so elegantly outlined in [14], the whole construction process to create an inverted index is essentially an matrix transposition. In the beginning of Section 4 we introduced table 3 and 4. In table 3 we have a table where each row is a document, and each column is essentially a term (word). To construct an inverted index structure for such a document collection, we need to perform a matrix transposition from table 3 to table 4, whereas each row is transposed into a column and vice-versa. Now, the key challenge is the document collection size, which is quite large - much larger than available main memory - and thus we assume that the inverted index structure would occupy more main memory than available. In the construction process there is also taken extensively usage of compression techniques, as we shall see in section 4.4. Now we will have a closer look at some inversion techniques that has proven its strengths in response to our large data volumes which is to be indexed.

In-Memory Inversion. The in-memory inversion is a simple inversion technique which first constructs a template form of the inverted index based on a pass over the document collection. This template contains vocabulary terms and their frequencies, and also allocated blocks already created by using the term frequency in the collection. Then a second pass over the collection is required, and the pointers and such are so appended to the template. This approach is taking full advantage over the random-access capabilities of main memory in that the skeleton is already constructed. This technique works only when the main memory is about 10%-20% greater than the complete size of the inverted index. Also, if we is to construct a word oriented index to facilitate phrase querying, then the index size would increase quite significantly and pose a challenge to keep in main memory.

However, it is possible to extend this technique to allow a larger index than available main memory. This extension lays out the index template on disk, and then creates a series of partial indexes in memory and move each in a skip-sequential manner to the template laid out on disk. This extension thus enables this technique to handle quite large collections. One important thing to notice about this technique is that it requires two passes over the collection, which implies huge I/O costs and possible a lot of document analysis processing (see 3.7). In section 4.4 we shall see that taking advantage of compression will further improve this technique in the inversion process.

Sort-Based Inversion. The sort-based inversion technique is as the name indicates - an inversion method that is based on sorting. In its simplest term the technique creates an array of triples of the type $< t, d, f_{d,t} >$ on disk

in document order and sorted by term order. This array is then used to create the final inverted index. The transformation from the array to the final index is then to merge blocks from the array, which would then give the final index. This merge process is easy to see, as the triples are laid out on disk in document order and sorted by term order. Also here the usage of compression comes in handy. In this technique the vocabulary element must be kept in memory, which constraints the amount of data which can be processed at the same time. The larger the collection the larger the vocabulary, hence more main memory would be occupied by the vocabulary. This requirement also constraints which applications this technique is useful for, since large collections such as the TREC Web Data track has a large vocabulary and thus occupies large portions of memory.

Merge-Based Inversion. The merge-based inversion technique has focused on the problem of holding the complete vocabulary in main memory at all time. The key idea here is that we want to create a mixture of small indexes (*partial indexes*). The way these are created is that we continuously read in documents from the collection and index them. If the main memory has reached its capacity or a fixed threshold, the index including its vocabulary is flushed to disk as a partial index. This partial index is a complete index with its own vocabulary and occurrences part. Once the flushing to disk is completed, the index in memory is deleted, and a new index is created based on the new read of documents. This process continues until the whole collection has been processed, then a new merge phase between the partial indexes is initiated. The time to generate these partial indexes is $O(n)$ where n is all the words inside each document in the collection, and there will be a total of $O(n/M)$ partial indexes, where M is the size of the main memory. This new merge phase merges two indexes at a time; it starts off with merging the vocabularies, and for each time the same term occurs in both indexes, a merge between their posting lists will occur. A more in-depth understanding of these inversion techniques can be found in [22, 17]. As outlined above, there exists several inversion techniques used to create indexes. Most of the steps that typical happens in an inverted index construction is summarized in Listing 1.

4.3 Scalability

The scalability of the inverted index structure with regards to the indexing process depends on the inversion techniques discussed above in section 4.2. The last technique, *Merge-Based Inversion*, scales particularly well for a range of collection sizes. The partial indexes can easily be distributed, and the merge process is also highly scalable. Also, for an inverted index structure to scale regardless of document collection sizes, it needs to take advan-

```
1  create a partial index in memory
2  while there are more documents to index do
3    if the memory is full then
4      flush the current partial index
5      create new partial index in memory
6    end if
7
8    Add a new document to current partial index
9  end while
10
11 flush the current partial index
12 merge all partial indexes into one final index with multi-way
       merge
```

Listing 1: General steps for constructing inverted indexes

tage of data structures and algorithms which facilitates distribution among several nodes[14].

In the subject of scalability there is a very important factor that effects almost every aspect of search technology, namely the disk space consumption. An index which requires a substantial large disk overhead (overhead as in extra data per entry in the index) has drawbacks in that more I/O traffic must be performed to fetch the wanted data. The disk space consumption needs to be linear, so a distribution of the processing over n nodes would cut the whole processing M into $\frac{M}{n}$ smaller processes. A lower disk space consumption than linear is unlikely to be found, and an exponential consumption is certainly not desirable, as the need for extra nodes to process the data would be out of control. So, an inverted index structure providing linear disk space consumption is a requirement, as the I/O traffic would in such a case also be distributed (disregarding network bandwith).

Also of great importance in scalability and the inverted index structure is its ability to handle a vast amount of queries. Large web search engines nowadays utilize distribution in large scales to handle the query influx, and the inverted index thus needs to be scalable to provide this.

In [23] there is discussed different partition schemes for partitioning the inverted index. Partitioning an inverted index is the approach used to distribute parts of the index to other nodes, and thus make the inverted index distributed. There is two common schemes:

- document partitioning

[14] A node is in general a network connected computer

- term partitioning

In Figure 6 we can see a simplified inverted index structure.

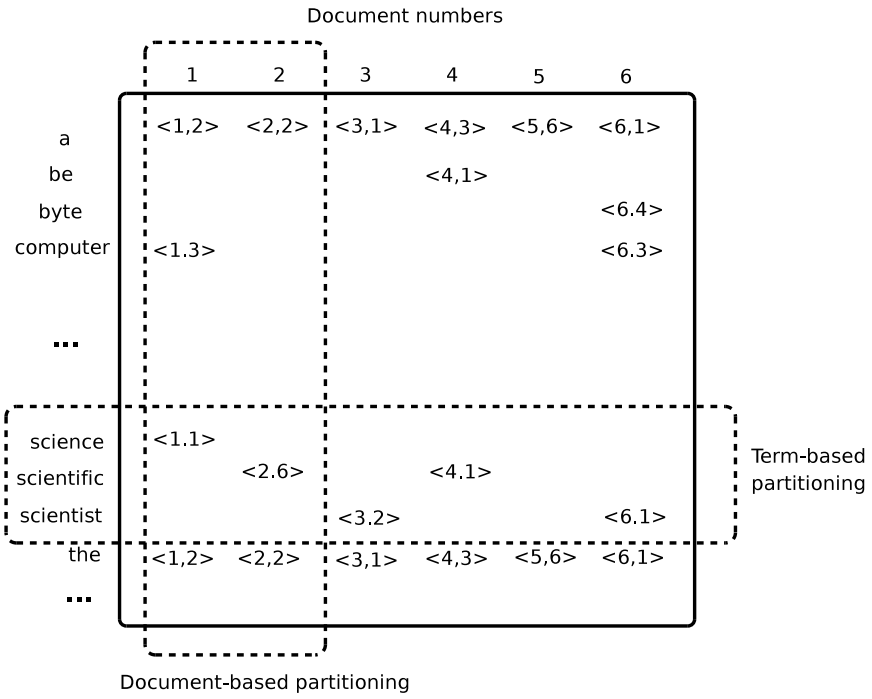

Figure 6: A figure encapsulating the inverted index as a whole, and how the document-based and term-based partitioning scheme works on the inverted index structure. The idea of this figure is borrowed from [17].

The figure is somewhat self-explaining; to the left we have the vocabulary, and at the top we have the document identifiers (numbers). Inside the main-box we have the postings, where the first digit is the document identifier, and the second is the frequency. Notice however the bounding boxes by the different partition schemes. The document-based partition scheme encapsulates a set of document identifiers, and handle only requests concerning those documents. The term-based partition scheme encapsulates a set of terms from the vocabulary, and handle only requests concerning those terms. From [22] there is concluded that most circumstances considers the term-based partition scheme to perform better than the document-based.

4.4 Compression

The usage of compression in inverted indexes is a necessity because of the large amounts of information involved. Almost every aspect of inverted indexes is affected by compression in that less amounts of data is needed to be

retrieved from disk and therefore savings in memory, disk space consumption and processing is achieved. In [14] compression is one of the key discussions, and a thoroughly discussion regarding overall aspects of compression effects on inverted indexes from [17] will be explored in-depth here. Another very interesting article regarding inverted index compression is [16], whereas coding schemes such as the ones brought forward here is thoroughly discussed. Also of interest within the subject of IR-systems and compression is the aspects brought forward in [21] and the experiments outlined in [7].

Compression in IR systems Compression has not always been so attractive in the area of indexes. In the times where the CPU was a fragile and limited resource, compression yielded little gains. This is because of the additional processing that is required by compression, lowered the overall performance of computers at that time. Compression was though present, but not in the same degree as now. In present times we are so lucky to have a very huge resource pool of CPU cycles. However, the disk is the bottleneck, meaning the disk access rates has not gone through the same development as the CPU. So, the gap between CPU and disk has become larger which affects the design of search engine architectures. In [9] there is a quite comprehensive example which illustrates the differences here. First is to fetch a single byte from memory to CPU, which has a delay of 12-150 CPU cycles. If we were to fetch a single byte from disk to CPU, we would see a delay of 10 000 000 CPU cycles. It should be evident that the benefits of implementing compression in inverted indexes is substantial and should not be avoided.

In the inverted index structure there are a lot of integers which needs to be represented and stored. Examples are the term frequency (f_t) in the vocabulary, the document identifier (d_j where j is the document number), or the word positions. Anyway, all these integers needs to be stored in the index to make it complete. The majority of data in the inverted index structure are integers, therefore we strive to keep the integer space as small as possible. The main question is how large should such numbers be? What space range do we need to support?

A very naive implementation would use 32-bit integers. This imply that all numbers in the index would occupy 4 bytes, regardless of the number being 1 or one million. This implementation would work, but it is not the most space efficient. There would be a lot of waste as word positions unlikely would be as large as 4 294 967 296 ($= 2^{32} \Rightarrow$ `int` type) for example. One simple, yet very space efficient approach here is to support integer coding by variable number of bits. With such an approach, the number limit is infinite, and the space consumption will scale from 1 bit to as much as is

needed. In contrast to the other approach of using a fixed number of bits (32-bit integer), this approach does not waste space nor set limits of max values. However, sufficient large numbers demand space, and in an inverted index there will be large numbers present. We will look at this problem after some introduction of coding schemes used in the field of compression.

Coding schemes. Some popular encodings used is *unary, Elias gamma(γ) and Elias delta(δ)*, whereas the Elias encodings are introduced in [8]. These are all called *parameterless codes*, and is thoroughly discussed in [14, 17]. By parameterless we mean that they have a fixed parameters which counts for all compressions, and if a coding is parametrized it means that the parameters used can be tailored depending on the characteristics of the data to be coded. Also one important coding scheme is the *Golomb code* introduced in [10].

Unary is a highly simple variable-bit code which encodes a integer x, where $x > 0$, as $x - 1$ bits as either "`1`" or "`0`" bits. The code is always terminated by an opposite bit. If "`1`" bit is being used to code the x value, then the terminating bit is "`0`". Unary is the simplest coding. Gamma and delta is slightly more complicated. The integer 4 in unary is `1110` - notice the terminating "`0`" bit. In the second column in table 6 we have some coding examples over a range of integers for unary.

The gamma code[15] is only capable of encoding positive integers, as implied above in the unary coding. The process of encoding a integer x is to extract two values; first find the highest power of 2 that x contains ($\Rightarrow 2^i$), and then the second value is the remaining binary digits of the integer x. We now have two separate values. Encode i in unary as described above; use i "`1`" bits followed by one "`0`" bit. Then append the remaining binary digits to the unary encoding, and we have a gamma coding of the integer x. As can be seen from table 6, we have coded a sequence from 1 to 6 integers, and the results for the gamma is in the third column.

The delta coding[16] shares such requirements that it can only code positive integers. The delta coding is somewhat similar to the gamma coding in that it also extracts two values from the integer to be coded. The first value is the highest power of 2 that the integer contains ($\Rightarrow 2^i$). The i from the first value is then coded as $i + 1$ in gamma coding. The second value is the remaining binary digits from the highest power of 2 of the integer. Then we have two separate values, and to represent it as a delta code we have to

[15] See `http://en.wikipedia.org/wiki/Elias_gamma_coding` for more examples
[16] See `http://en.wikipedia.org/wiki/Elias_delta_coding` for more examples.

Integer	Unary	Gamma	Delta
1	0	1	1
2	10	10:0	100:0
3	110	10:1	100:1
4	1110	110:00	101:00
5	11110	110:01	101:01
6	111110	110:10	101:10

Table 6: Example coding of a sequence of integers with unary, gamma and delta coding. Note that the ":" is only for illustrative purposes.

concatenate the two values. The fourth column in table 6 shows examples of delta encoding of a sequence of integers.

The briefly mentioned variable bit coding has the disadvantage that it is bit oriented. By this we mean that the computer needs to inspect each bit to decode such a value. Now, since the basic unit of access is multiples of eight bits (one byte), the process of inspecting each bit would be rather costly. Instead we rather inspect whole bytes, which will simplify the process. Based on this observation, we can now move on to a very simple coding scheme used to encode and decode variable length byte-aligned codes. This coding scheme can easily be summarized here: if $x <= 128$, then a single byte is used to represent the number $x - 1$ in binary, with a leading "0" bit. If $x > 128$, then the low-order seven bits of the number $x - 1$ is packed into a byte with a leading "1" bit, and the quantity (x div 128) is recursively coded the same way. The pseudo algorithm for these coding scheme can be seen in listings 2 and 3.

```
1 set x = x - 1
2 while x >= 128
3   write_byte( 128+x mod 128)
4   set x = (x div 128) - 1
5 write_byte(x)
```

Listing 2: Simple encoding with the variable-length code scheme

Just to mention a few advantages with this type of coding (bytewise) is that it promotes fast reading of compressed data streams, since it only needs to *step* through the stream for the kth subsequent code. This advantage is highlighted in [9], were it is proven that queries perform more than twice as fast as with bitwise codes.

The golomb coding[17] scheme is somewhat different than the gamma and

[17] `http://en.wikipedia.org/wiki/Golomb_coding` for more examples

```
1 set b = read_byte()
2 set x = 0
3 set p = 1
4 while b >= 128
5   set x = x+(b-127) * p
6   set p = p * 128
7   set b = read_byte()
8 set x = x+(b+1)*p
```

Listing 3: Simple decoding with the variable-length code scheme

delta scheme. Golomb coding is very efficient scheme which is *parametrized*. This imply the usage of an parameter in the encoding / decoding process of the scheme, and this parameter is calculated based on the length of a posting list. The encoding and decoding process in this scheme shares some characteristics with the *Elias gamma* and *Elias delta*, in that it creates two values based on the input value. These two values are created with the help of the parameter called M, and is a fixed value. Based on the input value we find a quotient and the remainder.

$$\text{quotient} \Rightarrow q = int(\frac{N}{M})$$

$$\text{remainder} \Rightarrow r = N\%M$$

As the previous coding schemes, the encoded value has a first and a second value, whereas the first is the *unary* encoded value of the quotient, and the second is in binary encoding. Note that if the M is not the power of 2, then it has to be truncated. This is achieved based on a logical statement: set $b = \lceil \log_2(M) \rceil$, if $r < 2_b - M$ then code r as plain binary using $b-1$ bits, else if $r \geq 2^b - M$ then code $r + 2^b - M$ in plain binary using b bits.

As with the *Elias gamma* and *Elias delta*, the output encoding is on the form "firstcode:lastcode". An example can be to encode the value 31, and set $M = 10$. Then $q = 1110$, since the $int(\frac{31}{10}) = 3$, and 3 in unary is 1110. The remainder is then 1, and following the description above we then get 001. So, given an input value of 31, the golomb coding scheme would output "1110:001".

Compression in vocabulary The vocabulary part of an inverted index structure is the first part of an index which is searched in order to retrieve the pointers to the posting lists in the occurrences part. Since the vocabulary is to hold all the unique terms in the document collection, it may become quite large. The size of the vocabulary is somewhat limited, as described in 2.6, where the vocabulary growth is described as a mathematical

expression. However, given a collection such as web data, the vocabulary will contain millions of entries. Each entry contains a unique term, the overall term frequency (f_t), and a pointer to a posting list in the occurrences part. These three elements inside an entry in the vocabulary needs storage space, especially the term. One standard approach in the vocabulary is to have the terms lexicographically sorted. With this in mind, we can assume that a lot of the nearby terms in the vocabulary may share some syntactical similarities. Have a brief look at table 7 which illustrates a small set of entries from a given vocabulary. Notice how all the terms are lexicographically sorted.

Term	**Frequency**	**Pointer**
...	...	...
connect	$freq$	`0xXXXXX`
connected	$freq$	`0xXXXXX`
connecter	$freq$	`0xXXXXX`
connecting	$freq$	`0xXXXXX`
connection	$freq$	`0xXXXXX`
connective	$freq$	`0xXXXXX`
...	...	...

Table 7: A table illustration a small piece of entries in a given vocabulary. Note that the term frequency is not written here.

One key observation here is that the "root" (see *stemming* in section 3.7.1) is shared between multiple terms. Take for example the term "connected" which is basically the term "connect" with the appended suffix "ed". The same goes for the term "connecter", which is the term "connected" without the ending "d" and with the suffix "r". The pattern here is that we, given similar terms, can express one term based on the previous with the usage of suffixes. This key observation will provide us with compression in regards to the vocabulary, and therefore also better utilization of the main memory.

Recall that it is very important that the vocabulary is very fast accessible, and therefore large portions should be kept in memory. However, a straight forward approach where all terms in vocabulary is coded with the usage of prefix and suffix offsets would imply a sequential scan to decode a term - which is not what we want. To approach this problem, we can have a fixed interval where we index the whole vocabulary term without the usage of prefixes and suffixes. Say for example that every 512 *th* entry is stored as it is.

One other aspect is that given a vocabulary of many millions of terms, which

would occupy most of the memory available, we could hold only each 512 *th* entry in the memory, and perform binary search between them. Now, in cases where the index term is between entry x and $x+512$, a read operation reading in all $x+511$ entries would be initiated, and all the entries would be buffered. This approach utilizes the main memory into more extent, and also promotes the buffering of frequently accessed posting lists to be performed. Note that vocabulary entries between two 512 *th* multiples which is frequently accessed would be buffered for long times, and not so frequent entries would usually be deleted given that the main memory has reached its capacity or a given timeout has occurred.

This vocabulary approach to compress and thereby utilize the memory is currently being used by the open-source search technology *Apache Lucene* in Section 3.8. The idea of holding each n *th* entry as "it is" in the main memory is one approach described and used in [23]. It should however be noted that not all search engines implement such structures in the vocabulary. Some takes advantage of distributing the vocabulary into a distributed hashtable, and thereby achieving $O(1)$ lookup time in each partition of the vocabulary. The network I/O delay will however play a role here. Others however just loads the entire vocabulary in the main memory and thereby accepts to sacrifice memory space which could have been used to hold posting lists.

Compression in the occurrences The occurrences holds the set of posting lists, which is the consuming part of the inverted index structure. In Figure 4 and 5, and Table 2, there is illustrated different examples of posting lists and how they are presented. One key observation in these illustrations is that given a posting list containing more than one entry, all the entries is sorted increasingly based on the document identifier. A simple list of document identifiers extracted from a posting list is shown below. Notice the order they are in.

```
10, 21, 23, 28, 31, 34, 35, 39, 73
```

We can see that each document id is a standalone integer, which each occupy 4 bytes. This is the least space efficient approach. If we now implement the *d-gap* approach, we will have this list of document identifiers.

```
10, 11, 2, 5, 3, 3, 1, 4, 34
```

This new improved document id list is the same as the previous one, but instead of denoting each document id as itself, we denote the gap between its document id and the previous one (showing just the increment value). With this approach we can represent document identifiers with much smaller numbers, hence their representation does not need to be a standard integer. It is in these circumstances we takes advantage of the coding schemes already

introduced in the beginning of this section. By representing these *d-gaps* with a coding scheme such as *Elias delta*, *Elias gamma* or *Golomb* we can harvest significant savings.

Take for example the need to represent the gap between document id 21 and 23, which is 2. The number 2 does not need 4 bytes to be represented. If we look at the Table 6, we can see that the number 2 needs 3 bits using *Elias delta* and 4 bits using *Elias gamma*. So, overall the most efficient representation of the number would be to use a byte, since we then eliminate the need to inspect each bit, but rather multiples of bytes.

The usage of compression in posting lists is proving even more efficient in case of common words (stopwords). Stopwords contains the largest posting lists because of its nature; it is a frequent word in the collection, and therefore contains a lot of document pointers. Of even more interest is that given the large frequency of common words, then the gaps between each document identifier in the posting lists would be on average very small; 1 or more.
Take for example the *very* frequent term `the`, which occurs in almost every document. Now, if we were to store a integer for each document identifier, and we have 10 million documents in our collection, then our posting list for this term (given document oriented index) would need 4 *byte* $\times 10000000 = 40000000$ *byte* (≈ 38 *MB*) just to store the *d-gaps*. We would also need to append the in-document frequency, and possible also word positions which would yield even more storage needs.

If we however could represent each document identifier as a *d-gap*, and to code each *d-gap* in *Golomb, Elias delta or Elias gamma*, then our posting lists would only be a fraction of the above calculation and thus promote a much faster retrieval. The indexed stopwords will always point to the consuming posting lists in the occurrences part, but we would significantly reduce this space overhead with the usage of compression. To even further reduce the costs there is the possibility to discard stopwords. The main disadvantage with that is that the IR-system would most likely loose effectiveness as described in Section 3.7.2, and thus lowering the overall performance for the end users. Especially phrase queries would be effected by stopword elimination, as described in Section 3.7.2.

Word positions Word positions only apply for word oriented indexes. And as stated, indexes which is to efficiently offer proximity and/or phrase searching capabilities needs to store word positions in order to find intersections between posting lists. Now, the concept of word positions is pretty much the same as document identifiers - they are both stored in increasing order. Introducing word positions in indexes impose huge storage overheads.

Stopwords, which already account for a large part of the occurrences part, would considerable increase given the nature of a posting list. Each entry that already contains a document identifier (gap) and an in-document frequency, would also contain a new list consisting of each word position inside that document. A very naive implementation would maybe apply integers for each word position too. There is fortunately very simple to see the downside of such a naive implementation. Word positions shares some characteristics with the document identifiers, and one is the significant savings of implementing *d-gaps* - we will from now on call this type of gap for *w-gaps*, as done in [17]. Taking advantage of coding schemes such as *Elias gamma, Elias delta* and *Golomb* here would lower the storage costs considerably.

An interesting observation done in [17] is that only a handfull of the most common words in the occurrences account for more than 10% of the total index size. One can therefore imagine the processing needed to handle query evaluation involving such terms.

4.5 Author comment

The inverted index structure is one of the most researched and discussed elements in the search engine architecture. It is therefore highly interesting to read such acknowledged research papers within this subject as this chapter is built on, knowing the widespread usage of the structure.

The inverted index structure has many advantages in contrast to structures such as signature files, and the structure holds and scales for query types such as boolean queries containing simply query terms. However, with the introduction of phrase query, which is the overall subject of this master thesis, the inverted index structure has some downsides. These downsides are neglectable for small document collections, but given a sufficient collection, which is often found nowadays, the downsides are more obvious. In [14] there is outlined that the inverted index structure outperform structures such as signature files and bitmaps, but the challenges in phrase querying are not taken into account in relation to large document collections. The same goes for the subject in [22].

In the research work [13] the issue of phrase queries in inverted indexes is discussed, and approached further. This is one of the earliest research work which addresses this problem. More thorough work has been accomplished in [5] with the introduction of *nextword index structures*, and then we have the introduction of *phrase indexes* in [12], and the substantial gains of combining structures into hybrid IR-systems. The latter approach is also more discussed in [13].

5 Related work

This chapter will explore related work in the area of inverted index structures which promotes efficient phrase search capabilities. Sections in this chapter must be read with the previous chapter in mind; the inverted index structure.

The two first sections in this chapter explores two somewhat similar structures. Both these structures builds on the *inverted index structure*, taking advantage of the concept of *mapping an information unit to its occurrences*.

The first section discusses a structure called *nextword index* introduced in [13] which explores some of the characteristics of terms in the vocabulary. Its main focus is on pair of terms; a *firstword* and a *nextword*. The second section discusses a structure called *phrase index*, introduced in [12]. The meaning of this structure somewhat lies in the name; it focuses on phrases and how to index them as single searchable units.

The last section has a primary focus on combined solutions which is meant to extract and utilize the advantages of the other structures, and by that create an IR-system which supports various kinds of query evaluations. A very attractive feature with such an approach is that each query evaluation type is highly efficient since each type has its own especially tuned structure, that performs very well for such queries. This is in contrast to having only one index structure that needs to encapsulate and support all kinds of queries, such has having a single standard inverted index.

5.1 Nextword index

The nextword index is a quite interesting index structure, which exploits some of the characteristics of phrase searching within a document collection. The nextword index takes advantage of phrase queries of length 2 or more, and where there is a stopword within. This stopword is quite significant, since it signals a huge occurrence list that costs to retrieve. This cost is in terms of I/O traffic and processing needs, which counts for the majority of the time delay in IR-systems. Now, we want to keep the costs below a satisfactory level to cope with phrase searching in large text indexes, and therefore will strive against minimizing the retrieval of large occurrence lists (posting lists).

5.1.1 Index introduction

The special characteristic of the nextword index is that it is similar to the standard inverted index in all terms, except for high frequent terms (stop-

words). What the nextword index does, in case of stopwords, is to "hook" a stopword up to a new list, instead of a posting list such as in the inverted index. This new list holds all terms that has been identified to occur right after that particular stopword. Consider this small set of phrase queries of length 2:

Example: "new york", "the yankees", "computer science"

In essence, these phrase queries is combined of two queries, a firstword and a nextword. Considering the phrase query "new york" we see that the usual approach, where the posting list for each word inside the phrase query is retrieved, would imply a significant time delay in case of stopwords. However, since we in the nextword index has "hooked" a list to the stopword, instead of the usual posting list, we will be able to lookup the subsequent word right there.

The main purpose is to reduce the costs of retrieving large posting lists for stopwords. When we have a phrase query containing a stopword and a subsequent word, then instead of retrieving the large posting list for the stopword, and then for the subsequent word, we want to retrieve the preprocessed posting list for the phrase *firstword+nextword*. The posting list for "new york" is in all likelihood much smaller than the individual posting lists for the terms "new" and "york".

In Figure 7 we have graphically outlined the nextword index structure, in the same manner as with the standard inverted index in Figure 4. Observe that given the lookup of a stopword, we would need to retrieve the nextword list for that particular stopword (if not already fetched). The nextword list can be stored in advanced data structures such as a HashMap, B+ tree or just stored sequentially. A good data structure for such a list would enhance the lookup speed, as well as the storage space needed.

Another significant gain introduced by the nextword index structure is the savings of disk accesses in case of phrase queries which does not have any hits in the document collection. This saving is achieved by that if the *firstword+nextword* pair does not yield any hits in the nextword index, then that evaluation could terminate without the need to fetch any posting lists. In a standard inverted index the fetching of posting lists would have been performed, since the only way to establish whether there is a hit or not is to decode and merge the posting lists. The nextword index needs a single disk access in case of a non-existing phrase query, since it needs to fetch the nextword list from disk, as illustrated in Figure 7.

The nextword index structure is in [13] introduced as a phrase browsing

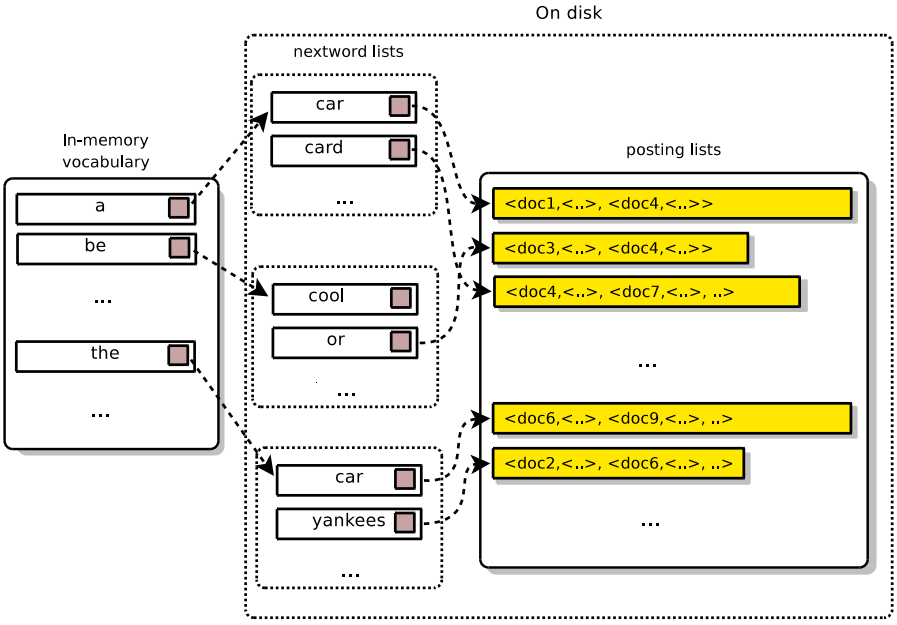

Figure 7: A figure illustration the nextword index structure.

structure, and the usage of stopwords as firstwords is neglected based on that it would produce a vastly large nextword list and thus lowering the performance. However, the experiment outlined in the research paper proved the opposite due to the savings in retrieval of posting lists. In [5] there is introduced the usage of stopwords as firstwords, and the savings that comes along with such an approach.

In case of phrase queries with lengths beyond 2 terms, there follows a slightly different query evaluation strategy. First the evaluation needs to select which firstword+nextword pairs to use, then evaluate them individually. This selection process is thoroughly discussed in [13]. Given an example query such as “programming in python”, a good approach is to use “in” and “python” as the firstword+nextword pair, and the query term “programming” on its own. Looking up the firstword+nextword pair is easily performed, and then the corresponding posting list is retrieved. If the firstword+nextword pair does not yield any hits, then the query evaluation can be terminated, thus saving two additional disk accesses in fetching the posting lists for the term “programming” and the firstword+nextword posting list. If the pair has hits, then its posting lists is fetched, along with “programming” terms posting list, and then they are merged to identify phrase hits.

5.1.2 Index performance

The nextword index performance is thoroughly explored in [13], where the nextword index structure was experimented with. In the outlined experiment it was measured towards the MG[18] search engine described in [26], and proved to perform from 1.6 to 2.5 times faster than the MG search engine, depending on the characteristics of the phrase queries used.

The query evaluation speed have sufficiently increased with the usage of the nextword index structure, but so is also disk space consumption. In the experiment outlined in [13] there is observed that the index consumes approximately 60% of the indexed data, which is not a desirable factor for an index. Note that this is given the usage of a nextword index for a whole document collection alone, without the utilization of stopwords as firstwords. There are, however compression techniques such as those described in Section 4.4 which can lower this consumption. And, the experiment implementation of the nextword index structure in [13] was only a prototype. Even with a reduction of the nextword index, it is in most cases twice as large as a standard inverted index, and thus poses disk consumption challenges, given an nextword index of the whole collection.

5.1.3 Author comment

This index structure is very interesting since it builds an index inside an index to reduce the posting list for the firstword, and instead spread it out based on the firstword+nextword pairs. It is basically an extension of the standard inverted index with a focus on more efficient phrase searching.

The way this nextword index structure approaches the challenges with phrase searching is highly interesting since it takes advantage of the characteristics of stopwords in the standard inverted index, and utilize this to sufficiently lower the response time for phrase query evaluation.

The authors personal believe is that this index structure is an elegant approach towards the phrase searching challenges, and should be further explored in more real life IR-systems. The author have yet to found IR-systems utilizing this kind of index structure.

[18] Managing Gigabytes

5.2 Phrase index

The phrase index is - like the nextword index - pretty much the same as the standard inverted index. However, a fundamental change in the vocabulary makes the phrase index more suitable to address specific phrase searching challenges, in that the response time is sufficiently lowered. This performance enhancement is due to reduced disk accesses and I/O bandwith, along with reduced processing and decoding of vastly large posting lists.

5.2.1 Index introduction

What separates the phrase index from the standard inverted index lies in the vocabulary. In the inverted index vocabulary we use single words (*unigrams*[19]) as searchable units (terms), but in the phrase index we have neglected the usage of single words. Instead it uses whole phrases as searchable units. Also, in a phrase index there would be no need to store in-document positions, which lowers the disk space consumption considerably. The reason for have the in-document positions in the inverted index is to support phrase queries by merging word positions and find matches. This will not be necessary in the phrase index.

In Figure 8 we have outlined characteristics for the phrase index. Notice the similarity between the phrase index and the standard inverted index outlined in Figure 4.

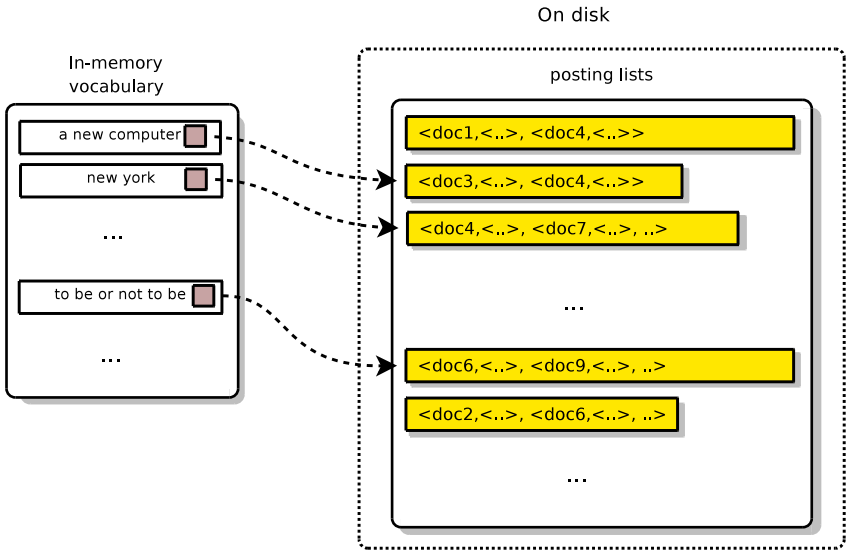

Figure 8: A figure illustration the phrase index structure.

[19] A unigram is an N-Gram of size 1. `http://en.wikipedia.org/wiki/N-gram`

The phrase index provides, with the characteristics described, a very fast approach towards evaluating phrase queries. The process to evaluate a phrase query is simply to lookup that particular query term in the vocabulary (which should stay in memory), and if there is a hit, then the posting list is fetched and processed. This posting list would be much smaller than the posting lists, which would have been retrieved in a inverted index approach. An example query: "Olympic games".

- **phrase index:** Lookup in vocabulary. If a hit, then retrieve and process posting list, or else terminate phrase query evaluation.
- **standard index:** Lookup "Olympic" in vocabulary, retrieve and decode posting list. Lookup "games" in vocabulary, retrieve and decode posting list. Merge posting lists to find hits where the term sequence "Olympic games" occurs is in that order.

The former approach requires only one disk access, given that there is a hit in the vocabulary. If no hit is found, then no disk access would be performed. In the latter approach, there are two disk accesses, and each disk access has a substantial I/O load since the posting lists is quite large. The benefits with the phrase index are very attractive.

With the introduction of a phrase index, one might wonder if it is suitable to index a whole document collection? Due to the amount of potential phrases in a document collection, such an index would be prohibitive in construction time and disk space consumption. So, if you can not use a phrase index to reflect the whole document collection, what can you do with it? In [12] there is discussed a common approach where only a small subset of all possible phrases is indexed in the phrase index. For the phrase index to yield most gains in respect to the IR-system, it would need to index frequently occurring phrases, since those poses large costs to the IR-system. By continuous monitor the query input to the IR-system (typical query logs), one might be able to analyze and extract frequently occurring phrase queries, and thus predict their future usage. Also phrases, which is hard to evaluate in a inverted or nextword index, would be attractive to index. The benefits of indexing these phrase queries in the phrase index would be substantial. This kind of phrase index is referred to as a *partial phrase index*, since it is by nature *partial* in respect to the amount of phrases which is indexed in it.

5.2.2 Index performance

The performance of a *partial* phrase index is substantial, in respect to that its primary focus is on resolving phrase queries. This is also its major constraint - no other query types (see Section 3.2) is provided.

In a query evaluation, the phrase index demonstrates its power in case of frequently occurring or difficult phrase queries. The query evaluation process is straightforward and significant fast, and the savings when these kinds of phrase queries are indexed is substantial.

5.2.3 Author comment

This index structure focuses primarily on phrases, both in terms of queries and searchable terms. It is therefore an interesting twist to use complete phrase queries as the entries in the vocabulary, for then to support direct lookup for those phrases. This approach yield a very fast determination whether the phrase query is indexed or not, thus allowing early termination of the evaluation.

The author find the similarity between the phrase index and the nextword index interesting, in that they both take advantage of a preprocessed posting list for a phrase. This is a common performance characteristic they share. In case of the nextword index, only phrase queries of length 2 brings such a posting list, as longer phrase queries brings more posting lists to process. Nevertheless, this signals the importance of preprocessed and narrow posting lists, which is some of the essence of this master thesis.

5.3 Combined solutions

Hybrid solutions within phrase searching can be thought of as combinations of the techniques previously discussed to provide a full feature IR-system. Since both the inverted, nextword and phrase index poses constraints and severe performance challenges within their own areas, an approach to combine them and take advantage of their characteristics would be interesting.

In [12] there is considered three combinations:

- Combined inverted and nextword indexes
- Combined inverted and phrase indexes
- Three-way index combination

We will in the following sections explore these combinations, as they bring valuable thoughts when considering the main goal of this master thesis.

5.3.1 Combined inverted and nextword indexes

This combination takes advantage of the nextword and the standard inverted index, as thoroughly introduced in [12]. In Section 5.1 we discussed in which cases the nextword index performs best, and it was concluded that the largest gains is when using stopwords as *firstwords*. So, whenever a word identified as a stopword occurs, then it is selected as a firstword along with the next word in the document as the nextword. This forms the firstword+nextword pair. In addition to hold the nextword index, a complete inverted index is constructed to reflect the whole document collection.

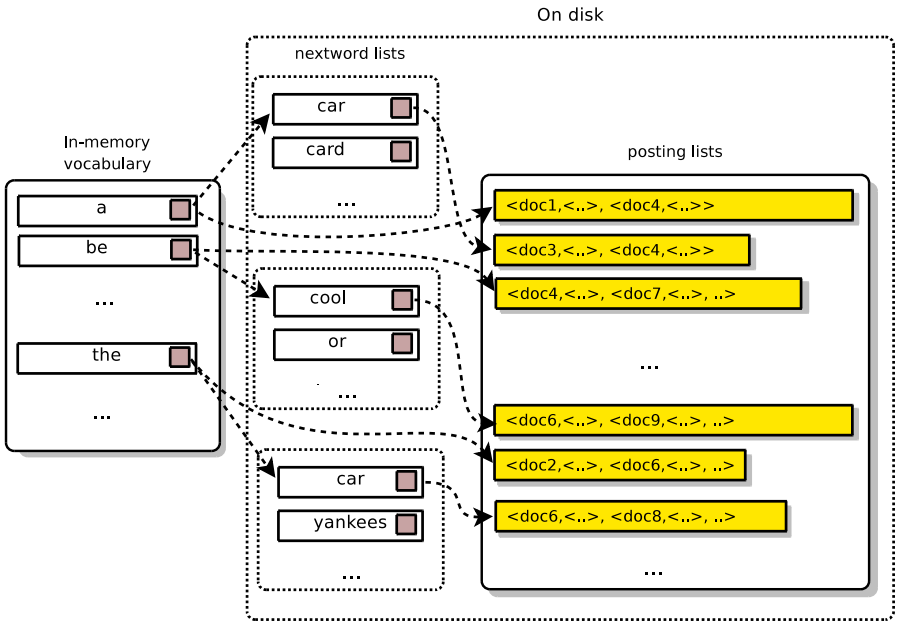

Figure 9: A figure illustration the combination of the inverted index and the nextword index structure

In Figure 9 there is illustrated this combined structure. Notice the similarity with Figure 7. The only aspect separating this combined index with the nextword index in Figure 7, is that the vocabulary holds pointers to both the nextword lists and posting lists. Stopwords in the vocabulary has pointers to both its nextword list and posting list, and the other words has just pointers to their posting lists.

A simple phrase query evaluation with such a combination would work like this:

1. Identify any pairs in the query, in which the firstword is a stopword

2. Fetch the posting lists for each firstword+nextword pair
3. Fetch the posting lists for the rest of the query terms
4. Process the posting lists in increasing order to find the phrase query hits

This simple step-by-step procedure tells us that we use the nextword index for all the firstword+nextword pairs which we manage to identify, and then the inverted index for the rest. The gains is in that all occurrences of a stopword signalize the usage of the nextword index, and the nextword index handles these kinds of phrase queries significant fast. The rest of the query terms in the phrase query is handled by the inverted index, which performs quite well for such query terms.

5.3.2 Combined inverted and phrase indexes

This combination takes advantage of the phrase index and the standard inverted index. Also this approach is thoroughly described and discussed in [12]. With the phrase index, this combination would promote direct lookup of indexed phrases. If the phrase query is not indexed in the phrase index, then the query would be evaluated using the standard inverted index.

The twist with this combination is that we, as discussed in Section 5.2, do not consider a complete phrase index, but a *partial* phrase index. The selection of which phrases that should be indexed in the partial phrase index is usually done by considering two factors: their frequency in the query logs, and their degree of difficulty in terms of query evaluation. An example of a hard phrase query is “to be or not to be”.

Consider the example query “new york”. In terms of the phrase index, the phrase query needs to be treated as a single query term, and then looked up in the phrase index vocabulary. If a hit is found, then its posting lists is fetched. If no hit is found, the query is to be evaluated against the standard inverted index.

5.3.3 Three-way index combination

This three-way index combination takes advantages of both previous combined approaches into a single combination. The motivation behind such an combination is the eager to further harvest phrase searching gains, and that the techniques described in this chapter provides such gains in an combined manner. This three-way combination is introduced in [12]. Based on the results provided by the experiment in the report, this combination yields the best gains in terms of response time and disk space consumption.

In this three-way combination the query evaluation would proceed as follows:

1. Use the partial phrase index to check if the phrase query has been indexed as a single term. If a hit is found, fetch and decode the corresponding posting list. If no hit, then continue to next step.
2. Use the combined inverted and nextword index described in Section 5.3.2, and evaluate the query.

These procedures in this three-way index combination yields substantial efficiency gains. According to experiments performed in [12], phrase searching in text indexes is between 60%-80% faster than using a standard inverted index. There are variable factors though, such as how many stopwords to use in the nextword index to identify firstwords, and how to effectively select good phrase queries to be indexed in the partial phrase index. In Figure 10 this new three-way combination is graphically outlined, where we have two main entities; the partial phrase index and the combined nextword- and inverted index.

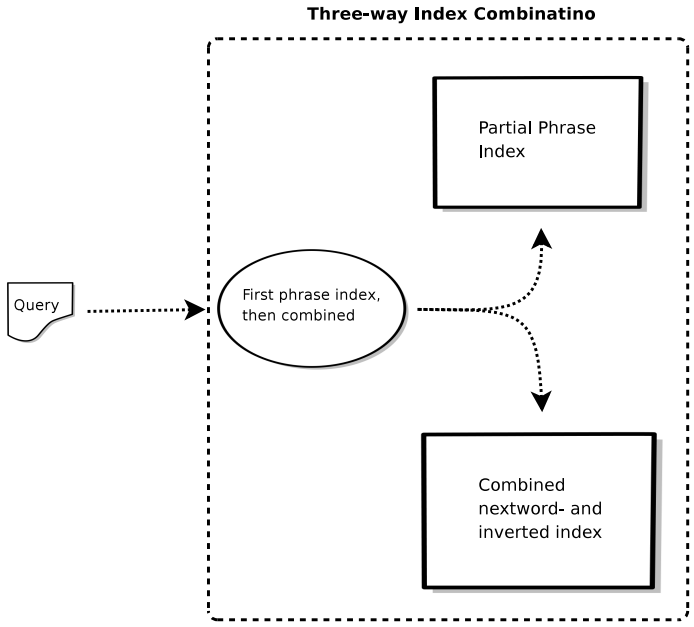

Figure 10: A figure illustration the three-way combination of the partial phrase index and the combined nextword- and inverted index.

Table 8 below is basically a replica of the table summarizing space and time effectiveness in [12]. The reason for duplicate this table and presenting it here, is to give the reader a sense of the characteristics for each index structure outlined in this chapter; both individual and combined.

Scheme	Index Size	Avg. Query Time
Inverted	1429 MB	1.04 sec
Inverted (stopping)	1002 MB	0.20 sec
Nextword index	2816 MB	0.02 sec
Inverted + nextword, 24 firstwords	1943 MB	0.31 sec
Inverted + phrase, top 10 000	1442 MB	0.89 sec
Combined	1956 MB	0.26 sec

Table 8: A table with the summarized results from the experiments performed in [12]. The document collection used was TREC WT10g, and it contained 1.67 million documents and is 10.27 GB in size. The queries used in order to extract response times was from the Excite query log, reflecting real life queries.

Note that the stopped inverted index in Table 8 have used stopword elimination to reduce the index size and response times for evaluation of stopword query terms. The "top 10 000" notation used in "Inverted + phrase, top 10 000" denotes the usage of the 10 000 most frequent occurring phrase queries, based on the Excite query log.

Table 9 below is another replica of the table summarizing a query test towards the three-way combined approach. It is easy to observe that the results in Table 8 and Table 9 is related.to each other.

		Numbers of Firstwords						
Query length	Inv. ind	3	6	12	24	48	96	192
All	1.04	0.41	0.30	0.27	0.26	0.24	0.22	0.20
2	0.35	0.14	0.14	0.14	0.13	0.13	0.12	0.11
3	1.62	0.64	0.39	0.31	0.28	0.25	0.22	0.17
4	3.73	1.32	1.00	0.92	0.88	0.84	0.81	0.74
5	3.80	1.62	1.13	1.06	0.95	0.89	0.85	0.77
6	4.67	1.78	1.39	1.33	1.21	1.12	0.91	0.81
7	4.99	2.31	1.41	1.41	1.37	0.78	0.63	0.56

Table 9: A table in [12], with the average query times (in seconds) of $\approx$ 66 000 queries. The three-way combined approach described in Section 5.3.3. Used the 10 000 most common phrases, based on the Excite query log, and a nextword index based on 0 to 192 stopwords.

5.3.4 Author comment

The combined indexes outlined represents some very interesting combinations of a set of state of the art index structures. Each combination has some appealing characteristics, whereas the nextword index introduces a very in-

novative idea. Realizing that each of these state of the art structures works quite well within their usage domain, which is frequent and/or hard phrase queries, have pushed forward these combined approaches. Other query types such as Boolean and term queries is not supported by the structures, except for in the combined approaches.

The first combined approach, inverted- and nextword indexes, provides strengths in a domain of tough phrase queries. Also, all additional queries is supported since the index structure is an "extended" inverted index. A concern within this combination is disk space consumption, and the additional disk access needed to fetch the nextword list for a firstword. However, based on the results, these concerns are addressed.

The second combined approach, inverted and phrase index, embrace the problems regarding frequently occurring and/or hard phrase queries. The partial phrase index provides rapid phrase query evaluation in case of an already indexed phrase. Here is the standard inverted index used as a fallback index, in case of a negative hit in the partial phrase index. The drawback is that there is two separated indexes, whereas one for the partial phrase index and one for the inverted index. If a query evaluation in the phrase index is terminated due to a negative hit, then the evaluation process would need to perform additional lookups in the inverted index in order to evaluate the phrase query.

The three-way combined index gathers the benefits from the two previous described approaches, and describes a combination where there is two indexes. By utilizing the partial phrase index first, and if necessary the second index. The second index is basically the combination of the inverted- and the nextword index.

The latter approach is the most appealing one, based on the results extracted from the experiment in [12]. However, all outlined index structures in this chapter brings interesting characteristics, which altogether enhances phrase searching.

6 Experiment

This chapter will cover all aspects regarding the experiment undertaken in this master thesis. One important factor for both me and my supervisors was to include a practical part in this report - we wanted to have some coding.

In the experiment outlined below we will look into the open-source search technology framework named Apache Lucene, somewhat discussed in Section 3.8. The search framework encapsulates all the typical aspects of a standard search engine, and provides helpfull techniques which helped us significantly in this experiment. Also, using this framework saves us from the additional work of creating our own search framework, which would have introduced an unsurpassable time delay.

In Section 6.1 we briefly go through what the experiment should include based on the problem definition and the overall scope of this master thesis. An experiment without any goal is not a good experiment, so in Section 6.2 we present a overall goal of our experiment. Section 6.3 will deeply present our choice of search engine framework. We will also explain the index structure used, since its highly relevant for this report. We will also look at how the search engine provides phrase searching. Section 6.4 presents one core aspect of this experiment; the index structure we want to *create* without modifying the existing inverted index structure. Section 6.5 introduces the document collection we will use to analyze and test our implementation. In Section 6.6 we outline the actual implementation of the experiment which also will cope with the architectural and performance choices made.

6.1 What shall be done?

The experiment will attempt to test and evaluate phrase query evaluation on different types of inverted indexes. This process will include a document collection, and that collection would be indexed based on these inverted index types. As stated in the problem definition in Section 2.2, we will look at both the indexing and the query process to evaluate how these inverted indexes performs. This implies that we will index the document collection for each inverted index type we use in our experiment. From there on perform a set of queries and monitor search engine, and extract information that can be useful in the evaluation.

Index process So, first aim is to create the search engine with the help of our open-source search technology framework. Here lies the need to create a good architecture to support our task. One important factor outlined in

Section 3.7 is the process of analyzing and extracting information from the documents in the collection. This process is the foundation for the index, since each indexed term, which together constitute the index, needs to be identified in that process. We will later look deeper into this subject since it is highly relevant for our *Bigram index*. After the indexing task has been completed, and the appropriate measurements done, we can then start to focus on the search process.

Search process To be able to perform an effective search process and retrieve relevant measurements we would need a set of queries. This set of queries could potentially be extracted from the document collection, but that would introduce more work for us, so we focus on applying a predefined query set instead. Then the search process can be executed. Now, along with the search process we would also need to measure and extract information to be able to evaluate the process later on, so this requirement must be fulfilled.

Evaluation Finally what needs to be done is to gather all the information extracted from each previously performed processes, and then analyze and measure how well each process did. This last phase here is very important, since it would give a final answer to whether the experiment was successful or not. Criteria for indexing is the time spent on indexing a document collection, the throughput of documents pr second (document-pr-second rate) and the initial disk space consumption. For searching, the performance criteria is how much time each phrase query takes, whereas each phrase query is categorized based on its characteristics. In 6.6 these criterias will be even further outlined.

Test platform The platform which we will perform our experiment on is listed in Table 10. All the hardware characteristics of interest should be included here.

6.2 The goal

The main purpose of this experiment somewhat shines through the overall goal of this master thesis; to experiment and measure different inverted index types and especially evaluate their effectiveness in regards to execute phrase query evaluation. What also lies in the goal of this experiment is to carefully look at the characteristics such as disk space consumption and processing needs. These aspects would be significant in the evaluation of the various inverted index types in the end, thus we need to inspect and measure the process of indexing and querying our indexes.

Our computer specification	
CPU:	Intel^{tm} Pentium 4 3.0GHz 1MB cache
Memory:	1 x 1GB
Disk:	1 x 80GB SAMSUNG HD080HJ 7200 RPM
OS:	Ubuntu Desktop Gutsy v7.10
Java version:	6.0 (v1.6.0_03)
Java args:	-server -Xms256M -Xmx768M
Python version:	2.5.1

Table 10: Shows the hardware specification of our test computer. Note that we set initial java heap size to 256MB, and maximum to 768MB.

6.3 Search framework

The search framework used in this experiment is better known as *Apache Lucene*, which is briefly mentioned in Section 3.8 and more thoroughly introduced in [6]. To further outline the *Apache Lucene* framework we will here introduce the implemented index structure, and thus also the characteristics of this particular index. Since this element is one of the core elements within search technology, and that it somewhat identify and categorize the search framework, we have devoted some time and space for this subject.

One common misconception regarding Lucene is that its a ready-to-use application like desktop search tools or a web search engine. Lucene is more like a software library which provides all the features and techniques available to facilitate search capabilities in your application. As a software library it has an overall goal; to provide search capabilities such as indexing and searching documents. It does not concern nor care about your application, since that is not its goal. All the application logic is not important, it is more the other way around. Your application needs to take advantage of, and integrate, Lucene to be able to harvest the search technology "fruits" provided.

In Figure 11 we can see how a typical integration between an application and Lucene might work. The arrow denotes the communication in the form of commands to search and index documents that is defined by the application owner. In its physical form the *Apache Lucene* software library is a single JAR[20] file.

In Section 6.3.1 the inverted index structure will be thoroughly introduced along with some illustrations to make the introduction as comprehensive as possible without too much complexity. Also of importance in this section is to focus on the characteristics of the index structure. Section 6.3.2 will

[20] **J**ava **AR**chive. Used to encapsulate multiple files into a single one.

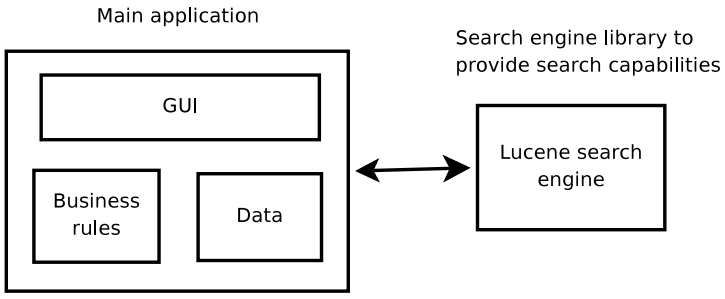

Figure 11: An illustration of how Lucene is integrated into applications, since it is not a single ready-to-use search application.

outline all there is to query evaluation within *Apache Lucene*, especially phrase query evaluation. It is very important to have a grasp of whats happening from the moment the query enters to the response is sent to the user. In Section 6.3.3 there will be some discussion regarding the search framework performance in terms of disk space consumption, indexing and query evaluation speed based on the already explored characteristics and such.

6.3.1 Index structure

The index structure within *Apache Lucene* is based on the standard inverted index structure thoroughly outlined in Chapter 4. The index is built around the concept of VSM (*Vector Space Model*) presented in Section 2.3.2, hence Lucene is able to list all documents that contain a specific term.

Inside the index structure there are three fundamental concepts which the index consist of:

- **Document** The index is a sequence of documents.
- **Field** A document is a sequence of fields.
- **Token** A field is a sequence of tokens, whereas a token is a pair of strings. The first is the fields name and the second is the tokens text. A token encapsulates a term, along with some additional information.

So, as summarized by the list above is that an index in Lucene contains a sequence of documents, and a document contains a sequence of fields, and a field contains a sequence of tokens. This index structure can be illustrated in Figure 12, which shows the relationships between each fundamental concept.

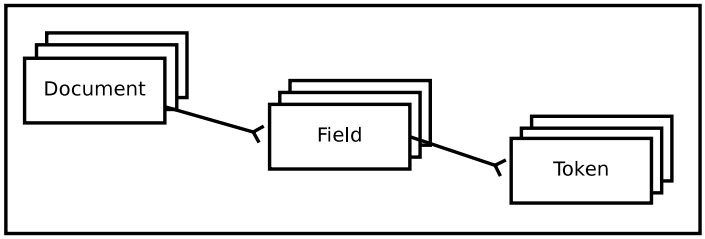

Figure 12: The Apache Lucene index structure, illustrating the fundamental components and their relationships. A document has a sequence of fields, a field has a sequence tokens.

To give a better grasp of this index structure, we have provided a simple example illustrated in Figure 13. In this figure there is shown the transformation between a textual document towards the presentation in the Lucene index structure.

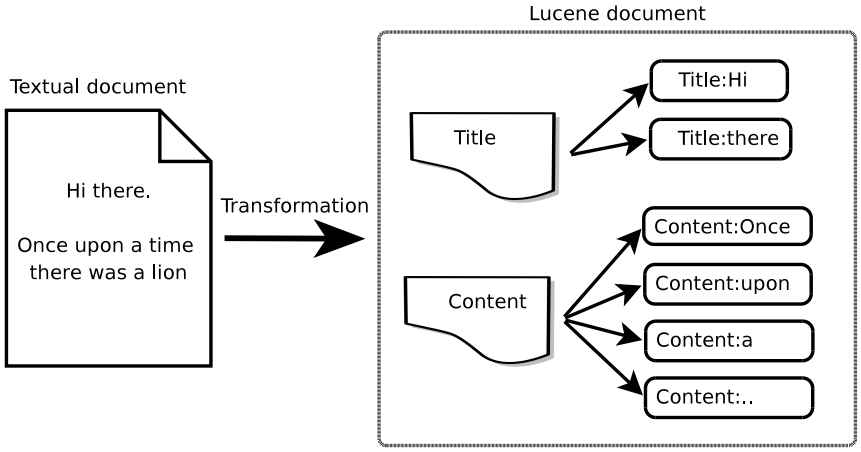

Figure 13: Illustrates a text documents presentation in the Apache Lucene index structure, highly based on the structure outlined in Figure 12.

Since an index construction might consume more main memory than available, there is often written sub-indexes to disk. In Lucene, this is also the common approach. Each sub-index, hereby called segment, is actually a complete index structure capable of being searched in. An index for a large collection will after construction contain multiple segments, each being searchable.

Vocabulary As all inverted index structures, this structure also has a vocabulary containing all the unique terms identified in the document col-

lection. That is, after the structure has been indexed. Now, as discussed in Section 4.4 regarding compression of the inverted index, we mentioned an approach where the vocabulary could be somewhat compressed and enhanced for faster lookup. There were two important concepts:

1. First the compressed presentation of each term, whereas a term is denoted as a prefix of the previous term.

2. Second is an index of the vocabulary that was achieved by having each n th term in its original form, and fetched to the memory

This is somewhat illustrated in Figure 14, and perhaps even better in Figure 15. The numbers are purely for the sake of appearances. The reason for this is that it is unrealistic to assume that the whole vocabulary would fit in main memory, nor that it would be efficient in regards to the posting lists needed to be fetched in response to a query. Think that each item in the memory is in its original form, and the following entries on disk is prefixes (if possible) of the one in memory.

A binary search of the in-memory vocabulary ($O(logN)$) would give us an idea of where the term is. If it is not in memory, we have a pointer to the disk allowing us fetch a bulk of terms from disk. There would be some kind of caching available here, so frequently accessed terms would always remain in memory.

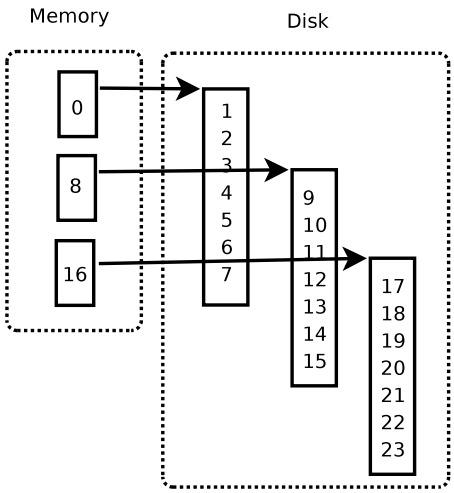

Figure 14: A illustration of the vocabulary structure in the Apache Lucene index. The vocabulary is in a way also indexed. Each number in the figure is just representing an entry which contains a unique term in the collection.

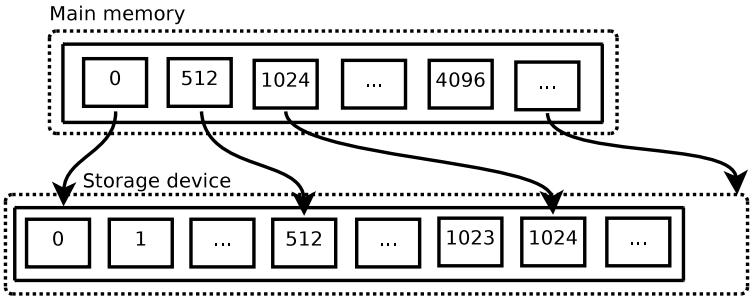

Figure 15: Another illustration of the vocabulary structure in the Apache Lucene index. The vocabulary is in a way also indexed. Each 512^{th} entry is here stored in memory.

Occurrences In Lucene the occurrences is pretty much represented as described in Section 4, that it is a set of posting lists. Lucene has an approach to facilitate compression and fast decoding of entries in the posting lists. For instance Lucene uses *d-gaps* to effective store document identifiers, and the same approach also accounts for storing word positions inside documents. Both these techniques saves significant storage space, and therefore also enhances a faster reading of postings lists.

Another interesting feature in Lucene's posting lists is called *Payloads*. Payloads is in its simplest term an array of bytes which is stored in the occurrences part. A payload may account to a document entry or a word position entry. The concept of payloads in Lucene's sense is that it can store arbitrary metadata related to entries in posting lists. This arbitrary metadata can further be used to enhance scoring calculation or describe other aspects of entries in posting lists. Payloads introduce an opportunity to implement other features in posting lists which could potentially have an impact on the hits found in reponse to a query. However, the concept of payloads imply more data in posting lists, and thus a slower query evaluation. An approach in Lucene tries to counteract the lowering of query evaluation throughput by storing the payloads in a compact manner *(byte array)*. It is important to specify that if a payload is not set (disabled), then there would be no additional space overhead in posting lists. Therefore the payloads in Lucene is quite space efficient.

The occurrences part of the Lucene index structure has, in regards to both posting list entries and word position entries, taken advantage of *skipping* to save decoding processing when performing query evaluation. As mentioned in Section 4.1, skipping yield significant gains in IR-systems when there was limited CPU cycles, but not so much in present computers because the gap

between CPU and I/O devices in terms of speed. However, skipping will provide some savings with the penalty of a little more complex structure.

In the Lucene index structure there are several files which are created to hold different data regarding the vocabulary and the occurrences part, and some more. Numbers which are represented as integers in these inverted index files holds constraints over the maximum allowed representation. This restriction somewhat limits the amount of terms and documents that can be addressed to a 32-bit quantity[21]. If we were to build an enterprise search system capable of indexing over 4 billion documents, then this restriction would constitute a significant problem.

A very good (and likely to be implemented in the future) alternative would be to replace the standard integer with a large value (64-bit), but this would again only skew the maximum allowed representation further down the road. A 64-bit integer would consume a lot of space, and can represent numbers up to 18 446 744 073 709 551 616. Possible a better alternative would be to embrace the `VInt`. `VInt` is a variable integer. The `VInt` is equal to the variable length coding scheme discussed in Section 4.4, and thus capable to represent all possible numbers and in addition achieve efficient compression ratio.

6.3.2 Query evaluation

Query evaluation within Lucene is basically what happens from the time a query is entered the IR-system, the processing is performed, and possible hits is found and returned to the user. The query evaluation is a very important aspect in search technology, since its the part of most importance from a user perspective. Fast retrieval and high effectiveness are two factors which accounts for the users satisfaction with the system.

Speed The first factor, fast query evaluation, is basically to have an efficient query run-time. By this we mean that the process of fetching the needed information, for then to be decoded and calculated, needs to run as efficient as possible. We have already went through obstacles such as significant time delays with I/O traffic and CPU processing. What is of special focus with the run-time of query evaluation, is how fast the data can be accessed; hence the I/O is important. If we need to fetch large amounts of data, which in addition is stored disorderly, then this will have a high impact for the run-time factor. If the data to be fetched is stored contiguous - a desirable situation - then reading could be performed in a single operation,

[21] See `http://java.sun.com/javase/6/docs/api/java/lang/Integer.html#MAX_VALUE`

without the need to perform multiple seek operations on disk. This would also account for appending data to lists on disk. As also thoroughly discussed, the gap between the CPU and hard disks in speed has grown in such a way that we have a large pool of CPU cycles available, but the hard disk unit is in contrast noticeable slow to retrieve the wanted data. Especially of interest is the seek delay for storage devices, which accounts for the major part. Take for example the retrieval of a single byte from main memory to the CPU, which cost between 12-150 CPU cycles. The retrieval of a single byte from disk to the CPU costs 10 000 000 CPU cycles. It is easy to see the waste gap between fetching from main memory and external devices, such as disk. In Section 4.4 we presented the attractiveness of compression, as a result of this gap. The more compact and compressed the data is stored, the less is required to read from disk, thus fewer CPU cycles are wasted in waiting for data.

In Lucene there are approaches to deal with challenges such as these. Taking a widespread usage of caching beyond what is performed by the operating system has proven to be very effective, and is thus implemented and provided through the API. Especially interesting is to cache results for frequent queries along with list caching. Often an LRU[22] policy is sufficient in cases of caching, but in [16] it is concluded that there exist other policies that will achieve significantly higher cache ratios. Just to mention a few other policies: Least Frequently Used (LFU) and Optimized Landlord (LD).

As discussed in Section 6.3.1 regarding the index structure of Lucene, we introduced the vocabulary structure of Lucene along with its characteristics for compression and effective utilization of main memory, somewhat illustrated in Figure 14 and Figure 15. This approach makes it easy to binary search the index in memory, and to locate and retrieve the entries which is not in memory from disk. Since the vocabulary entries is stored in compressed form, the amount of data to retrieve from disk is limited. A query evaluation would typically need only a small set of vocabulary terms. If the terms have been queried before, they would probably be cached and therefore fast accessible. If not, a disk access pr term may be required, which is the downside of this vocabulary structure. However, there exists an approach where a "warm-up" phase to force the IR-system to fetch and cache typical high-frequent words would yield significant savings in run-time query evaluation. A "warm-up" phase is typically a program running a predefined set of queries on the search engine, and thereby forcing it to perform the "heavy lifting" before allowing users to query.

Next in a query evaluation process would be to fetch the posting lists based

[22] *Least Recently Used* is a cache algorithm

on the term frequency. Now, in Lucene there is a division between a list of documents that contains a given word, and a list of word positions related to a given term which occurs inside a document. This occurrences part is therefore divided into two parts, the list of documents, and the list of word positions.

1. The list of documents is stored efficiently within its own file which also contains pointers to their word position list.

2. The list of word positions is like with the list of documents also stored within its own file. This file is accessed through pointers from the list of document entries.

Both these parts, which together constitute the occurrences of an inverted index, takes advantage of the *gaps* between document identifiers and word positions values, as discussed in Section 4.4 regarding the compression of occurrences. This introduces significant savings. As mentioned above, the "warm-up" phase would not only yield run-time savings for the vocabulary, but also for the occurrences, since posting lists would be fetched and therefore cached (by the OS and possibly also by the search engine).

Lucene will first parse the entered query in an attempt to categorize the query type. Recall from Section 3.2, that we have different kinds of queries. If Lucene can route a query type to the module which it is optimized for when handling such a query, then that would in the end perform much better than a "generic" query handler. We have several kinds of query types in Lucene:

- **TermQuery** is a query consisting of a single term.

- **RangeQuery** is a query who search for hits between a set of terms which is lexicographically ordered. A search for hits between dates is a good example here.

- **PrefixQuery** is a query searching for documents which contains term matching the beginning of the query, hence the "prefix".

- **BooleanQuery** is not a query in normal sense, but applied to combine *sub-queries* with the Boolean operators such as `AND`, `OR` and `NOT`. See Figure 3.2.4. Combined *sub-queries* constitute a regular query.

- **PhraseQuery** is a query taking into account the order of the query terms. Only documents matching the query terms with the same order they are in will be returned.

- **WildcardQuery** is a somewhat specialized query that allows querying for words with unknown characters. One example: "be*r" =>

matches both "beer" and "bear", despite the very large semantic gap. This query type is very expensive in terms of processing needs, since a lot of terms in the vocabulary would match a wildcard query, thus a large set of posting lists needs to be retrieved.

- **FuzzyQuery** is a query that provides a way to query for *similar* matches. The *similarity* can be based on *Levenshtein distance*[23]. One example could be "beer" which is *similar* to "bear", "beep", i.e.

After the appropriate query handling module has been selected, and the terms are fetched along with their posting lists, then the decoding and processing of a temporal structure is performed. The query handling module describes the internal policy in regards to how a match is found, and such.

Effectiveness The second factor, high effectiveness, is basically to have the sufficient information and calculations preprocessed so that it is easy to find the most relevant documents in response to the query. As mentioned above, Lucene builds on VSM, enabling it to calculate document weights and thereby rank the hits found. This process could be quite substantial if we had to process all documents in the collection (a naive approach) for each query to be able to give a good ranking.
However, in Lucene, much of the data needed to perform fast calculation of document weights are already preprocessed when the documents are added to the index. One important type of data is the term frequency, which is stored in the vocabulary along with other terms. So, when a document is to be added into the Lucene index, all the identified terms are added to the vocabulary along with their term frequencies. In case a term already exists in the vocabulary, then solely the frequency is updated. The same thing goes for other major operations on indexes, such as `update` and `delete`.

Other aspects in Lucene which promotes query evaluation effectiveness is the possibility to *Boost* a document, a field or a term. What lies behind *Boosting* within Lucene is to affect the scoring procedure which is performed for each query in such a way that a document, field or term gets a higher weight than the other occurrences. This *boost* technique thus enables us to favour a specific term in contrast to a non-boost term. If we have a collection within the domain of computers, then we would boost the term "cache" in contrast to the term "copyright". Another more business like example would be to boost terms such as a product name, model number, or items on sale and such. Lucene also provides the possibility to boost a field that hold terms (actually token objects, but we frequently use terms to denote them). This is very useful for generic search engines where there is a division between content and title. A natural approach would be to boost the title field, since

[23] `http://en.wikipedia.org/wiki/Levenshtein_distance`

a hit in the title field would more likely reflect the context of the document. Another level up would be to boost the whole document. This approach can be very useful in cases where an identified document is important to be exposed, frequently searched in, of general interesting, etc.

6.3.3 Performance

Apache Lucene is a search engine framework which is meant to be significantly fast, regardless of the collection size or the query rate. The real world understanding based on practical usage and experiments has to some extent proven this overall goal of the search framework. It does scale well compared to other search frameworks, and along with its search technology functionality it does reflect a real search engine alternative. In [19] there is a comparison among Lucene and other open-source search engines which support our statement regarding Lucene.

We have until now discussed the search framework in general, along with more specific topics such as the index structure and the query evaluation. All these topics include in some degree observations regarding the search framework performance. In the topic of Lucene's index structure we discussed characteristics of the vocabulary and occurrences, along with how efficient those elements perform. In Lucene's query evaluation we went far to discuss the characteristics of speed and effectiveness in the process of query evaluation. The aspect of performance lies on both these issues.

One aspect which Apache Lucene that has not been covered is the ability to handle a distributed search. The technique of distributing a search engine, both in respect of indexing and searching, has received an increasing attention in the search communties. Because of the vast amount of information that is to be made searchable on the web or in large intranets, a single computer alone could never cope with such demanding data amounts. Instead the concept of distribution is widely applied. We will not look further into this subject here, since it is outside the scope of this report, and not provided by Apache Lucene search framework. However, one sub-project by Apache which uses Lucene as a search framework, has been able to implement distributed capabilities. This sub-project is called *Nutch*[24], and uses a distributed filesystem named *Hadoop*[25] that implements *MapReduce* to cope with the vast amount of information needed to be processed. MapReduce is introduced in [3], and the reader is recommended to get familiar with the technique.

[24] `http://lucene.apache.org/nutch/`
[25] `http://lucene.apache.org/hadoop/`

We would like to recommend the reader to read the comparison of different search engines in [19] performed by some well known researchers within the search technology community. It gives an idea regarding how to place Apache Lucene search engine compared to other open source search engines.

6.4 Bigram index

The Bigram index - the core of this experiment - is in its simplest term a standard inverted index. We have the standard vocabulary and occurrences elements, and terms and postings lists are organized as described in Section 6.3.1. So, what distinguishes a Bigram index from the standard index used in Lucene? Since there has been no alternation of the index structure nor code, then the major difference is that we have taken full advantage of the definition of a term in Lucene. Lucene defines a term as the unit of search, and that is precisely what it is. During the indexing process we do not only consider terms as single words, but also as bigrams consisting of multiple words.

Bigrams of words is a *N-Gram* of size 2, as defined in Section 3.1. It is important to note that we do not consider n-grams to be character oriented, but rather word oriented. By character oriented, a bigram would imply a subsequence of two letters of a word, as opposed to a subsequence of two words of a sentence as in word oriented. We focus merely on word oriented n-grams in this report.

Considering this main difference in our bigram index, there are some potential obstacles ahead. If we consider both single words and pair of words as terms, then our vocabulary would double in size, making it harder to hold the vocabulary in main memory. Below is a sentence from the sound track "Stiff Upper Lip" by the Australian rock band named "AC/DC", released in year 2000. Observe how an indexing process using a naive bigram indexing approach.

```
Well I was out on a drive on a bit of a trip
```

Gives these single word terms:

```
a, bit, drive, i, of, on, out, was, trip, well
```

And subsequently these bigram terms:

```
well I, I was, was out, out on, on a, a drive, drive on,
        on a, a bit, bit of, of a, a trip
```

As seen, the identified bigram terms based on the sentence above produce additional $n - 1$ terms, where n is the number of terms in the sentence. If

we have a document of 20 000 words, and thus 20 000 single word terms, then we would also have $20\ 000 - 1 = 19\ 999$ bigram terms with a sequential approach as shown above. Using this approach would not yield any significant improvement on efficiency either, since the size of the vocabulary would impose a large I/O overhead. Only a fraction of the bigrams would actually constitute a performance improvement. Also note that some of the identified bigram terms above brings some semantic meaning on its own, and in [24] it has been experimented with the usage of bigrams to perform text categorization.

A factor to keep in mind is the identification of bigrams. In a very naive approach one might consider the vocabulary of an already indexed collection. If the vocabulary consists of x distinct terms, then there would be $x \times x$ possible bigrams. If our vocabulary holds 1 000 000 distinct terms, then there would be 1×10^{12} possible bigrams. Imagine holding that vocabulary in main memory. A naive approach like this would not work, both because of the vast amount of bigrams and that only a fraction would hold any semantic value.

It is important to know what kind of queries that lowers the overall throughput in a standard inverted index. As already thoroughly discussed in previous sections, the observation of stopwords and their huge impact on the index structure is of great concern here. In this report our main goal is to look at ways to improve phrase searching in text databases, and this is where our Bigram index comes into play. Now, phrase querying our standard inverted index would in some queries - where no stopwords are present - yield a satisfactory performance. This is because the posting lists to retrieve and decode from disk is small / normal in size, and the I/O overhead would not have any significant impact on our search engine. However, if we perform a phrase querying where stopwords are present, then the overall performance would decrease significantly, and thus result in dissatisfied users since the results is not retrieved as quickly as expected. The posting lists for stopwords is so large that they constitute a bottleneck in the IR-system due to the I/O.

Based on the observation above, we can safely conclude that there will be none or insignificant gains in identifying all pair of words as a bigram term, and then index it. This raises another important question: How can we identify terms that we with certainty would know provide performance savings? Well, we already know that stopwords are responsible for the majority of the occurrences size in terms of long posting lists and word positions lists, so stopwords have some characteristics which we can utilize. We do not want to discard stopwords in our index, since that could potentially impact the semantics in phrase query evaluation and may include non-relevant documents and thus lower the precision.

```
1  read stream of words
2  previous_word = None
3  gotStopword = False
4  added_bigram = False
5
6  while word in stream of words do:
7    if previous_word and gotStopword:
8      new Bigram(previous_word+" "+word)
9      gotStopword = False
10     added_bigram = True
11   end if
12
13   if word is stopword:
14     gotStopword = True
15   end if
16
17   if gotStopword and previous_word and not added_bigram:
18     new Bigram(previous_word+" "+word)
19   end if
20
21   previous_word = word
22   added_bigram = False
23 end while
```

Listing 4: General steps to identify and construct a bigram term

What we want is to identify bigrams which contains a stopword. If we construct bigrams based on the constraint that a stopword occur in, then we would reduce the retrieval of the very long posting list of that particular stopword, and instead retrieve a posting list based specifically on the bigram term. That is, the posting list denoting which documents that a particular bigram occurs in, along with in-document positions. This posting list will in size be a fraction of the posting list for the stopword.

In Listing 4 the pseudo code for identifying and constructing a bigram term based on a stream of word is listed. It is assumed that we already have a dictionary of stopwords identified, thus checking a word for a stopword is easy and fast. The core principle in identifying and constructing a bigram term is to check every term. If a term is a stopword, then construct a bigram containing the previous term plus the stopword, and the stopword plus the next word in the stream.

Below is a simple sentence as example. Feeding it into our bigram identifier and constructor would generate terms based on single words and bigrams. Lets have a look:

```
This is an example text, without any meaning of its own.
```

The stopwords from this sentence are:

```
an, any, of, is, its
```

then we produce these bigram terms based on the new approach:

```
this is, is an, an example, without any, any meaning,
meaning of, of its, its own
```

By analyzing these identified bigram terms we can see that they all is constructed based on the a stopword. And that stopword is - alone - quite expensive for the search engine. We can also see some semantic rich bigrams in this set, such as `an example, any meaning`.

Having analyzed and discussed the principles and goal of our bigram index, let us now have a look at how we implemented it, both in terms of Lucene and in general.

6.4.1 Implementation

We have now introduced our Bigram index and what distinguishes it from the standard inverted index. We will now show how we implemented it. First of all, we will follow the pseudo code in Listing 4, where bigrams are identified and constructed based on some rules. The main rule is to initialize a bigram if a stopword is observed, and then construct a bigram based on the previous term and the current stopword, and the current stopword and nextword. This can be summarized to that we for each stopword construct two bigrams; one containing the previous term plus stopword, and one containing stopword plus the next term.

Before we go further into the Java code and more inner Lucene logic, there is one important aspect that needs to be understood before proceeding. Recall from Section 3.8, we mentioned how simple there is to expand the library, and the concept of document analyzers. The key component which we shall discuss is the *document analyzer*, hereby referred to as the *Analyzer*. In Lucene the Analyzer is the class which handles the policy for extracting index terms from text. This may sound familiar, since it is what we have described in Section 3.7 where we described document analysis and index term selection.

It is now established that our primary focus is index term selection within document analysis. What specifically is defined to be accomplished in the Java class `Analyzer` in Lucene is that it shall return a `TokenStream` object that the indexing logic will use to create the index.

A TokenStream is an enumeration over all identified terms based on a stream of words. Note that the terms - searchable units - is referred to as Tokens[26] in Lucene, and that the TokenStream will enumerate over Tokens. In order to understand how we create our bigram and standard index with the usage of these Analyzers, we first need to understand how Analyzers work inside Lucene.

Analysers Lucene comes with some built-in Analyzers that have somewhat different purposes. We will not cover all of them here, but rather encourage the reader to explore them. We will however introduce three Analyzers, whereas two is used in our experiment, and the other one is not. Lets have a look at the latter one first.

StandardAnalyzer is a very widely used Analyzer which performs well under most circumstances. StandardAnalyzer is quite efficient in identifying different kind of index terms. The Analyzer takes advantage of JavaCC[27] to identify a wide range of index terms of various lexical types, such as:

- Alphanumerics
- E-mail addresses
- Numbers
- CJK (Chinese, Japanese, Korean) characters
- Acronyms
- Names

StandardAnalyzer is also constructed to discard stopword, if wanted. Now, StandardAnalyzer performs its purpose quite well, but in this experiment we will only need to identify single words, and there are therefore some downsides. A major downside is the time issues in relation to the wide range of parsing capabilities. So, in simple terms, it is too slow for our purpose.

SimpleAnalyzer is another very widely used Analyzer. It performs quite well using standard text, and where the interest of identifying index terms lies strictly in finding words based on alphabetical characters. So the main operation performed by SimpleAnalyzer is to identify words, whereas all other characters will signal a whitespace, and implicit the creation of a new Token. The Analyzer is very fast, since it builds on a range of other classes where each class is especially designed for its own purpose. An attempt to illustrate the relations SimpleAnalyzer has towards other classes in the Lucene package is illustrated here in Figure 16.

[26] A Token object contains the terms text along with the start and stop offsets inside the text

[27] Java Compiler-Compiler. A parser generator capable of perform effective lexical parsing.

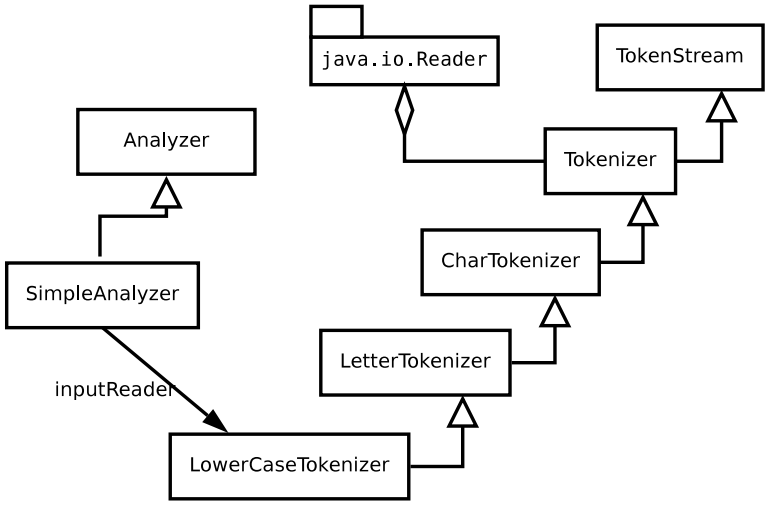

Figure 16: Shows how the SimpleAnalyzer uses the stream of text (inputReader) and a chain of objects to return valid Tokens based policy of the Analyzer.

SimpleAnalyzer extends the abstract Analyzer in Lucene, which includes a method to return a TokenStream enumerator. In SimpleAnalyzer this TokenStream enumerator is created by returning a new object of LowerCaseToknizer, which in turn extends LetterTokenizer and passes on the "inputReader" to its constructor. LetterTokenizer extends CharTokenizer, and passes once again on the "inputReader" object. CharTokenizer, in turn, extends Tokenizer, and Tokenizer extends TokenStream. Now, the actual construction of Token objects based on the input stream from "inputReader" is constructed in "CharTokenizer". That is pretty much the basic of SimpleAnalyzer and what is illustrated in Figure 16.

DocumentAnalyzer is an Analyzer which we created, that also serves as a factory class. DocumentAnalyzer holds a private class named `PhraseFilter3`, and this particular class holds all the logic in regards to identify and create Bigram Tokens. The `PhraseFilter3` utilize the SimpleAnalyzer described above, and inspects the Tokens created by the SimpleAnalyzer. This makes the `PhraseFilter3` quite fast, since SimpleAnalyzer perform minimum operations on the text.

Having a grasp of what Analyzers do is essential in order to understand the procedure of creating the bigram index.

Creating the index Lucene has the advantage of being designed with modularity in all levels. This becomes clear when we can *manipulate* the searchable units (Tokens) which are to be inserted into the index structure. Manipulating the construction of Tokens is in essence what we have done to be able to construct our Bigram index.

One important note regarding how Lucene handles phrase querying is that each Token object has some additional data containing positional information. Given an input document and the creation of Token objects for each identified word, a Token object would contain a value denoting the position relative to the previous Token. Lets again consider the simple example sentence:

```
This is an example text, without any meaning of its own.
```

Now, each word would imply the creation of a Token object (`Token('this')`, `Token('is')`, `Token('an')`, ..). A Token object has by default the positional value set to one, which tells us that this Token is "one word position" relative to the previous one. So `Token('is')` is a Token object which is one word position to the right of `Token('this')`, given that they are created in the order of the sentence. This approach is quite straightforward, and it adds the possibility to append additional terms at the same word position. We can add the tokens `Token('intention',0)`, `Token('purpose',0)` right after the token `Token('meaning')` (where the number 0 indicate that the word position has not changed, since the term is in the same position as the previous term.

Inside a Token object there is also offset information denoting the subset of offset values inside the source text which constitutes the actual word or pair of words. Note that this offset information regards character positions within the document.

For `Token('example')` in the example sentence above, we would get a "start offset" equal 12, and "stop offset" equal 18. This tells the IR-system that `Token('example')` starts at character position 12 and ends on position 18. If we consider the example sentence above as a document, then our positional information for the terms "this, is, an" are: `Token('this', start=1, stop=4)`, `Token('is', start=6, stop=7)` and `Token('an', start=9, stop=10)`.

In case of using the DocumentAnalyzer, we would need a set of stopwords defined. Given the example sentence, we could say that the terms `is`, `an`, `any`, `of`, `its` is stopwords. Now, using the DocumentAnalyzer would produce these tokens: `Token('this',start=1,stop=4)`, `Token('this is',start=1,stop=7)`, `Token('is an',start=6,stop=10)`,

..

The general process of identify and constructing bigrams based on a input stream of words is illustrated in Figure 17. The basic steps from the input of a set of documents to the insertion of the Tokens into the index is outlined here. Note that we have a process for parsing documents without performing any form of analyzing to identify index terms, since only pure text is identified. This is in cases where the documents requires careful handling, in case of PDF documents, HTML pages and such. This step would output a stream of characters, most likely with some additional metadata (such as a description of either the text is title, body, author, description). The next process performs the analyzing on the stream of text from the parses.

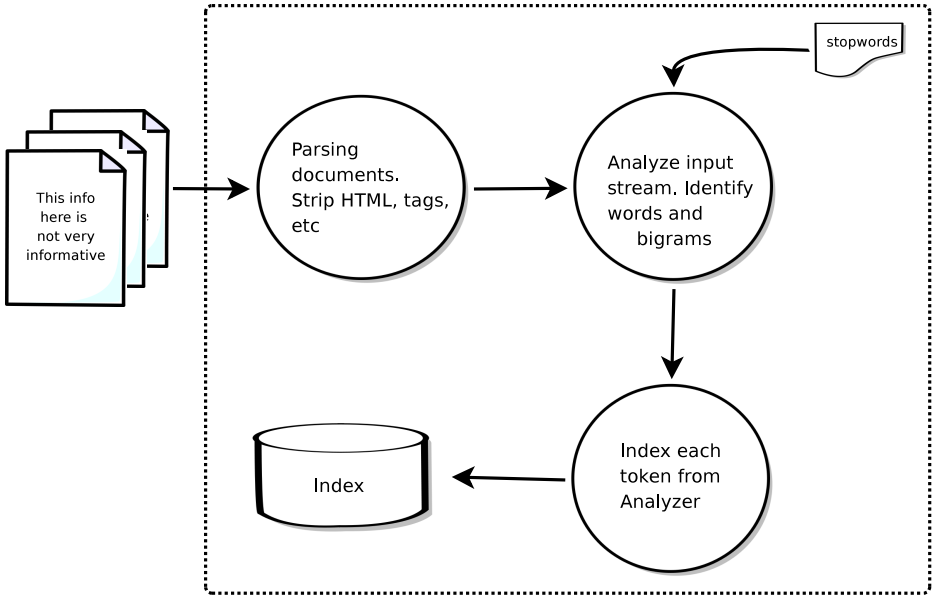

Figure 17: The process of constructing a bigram index, from the point of documents to the writing of the inverted index.

6.4.2 Performance

The Bigram index performs somewhat similar to the standard inverted index since its basically the same internal structure and data types. The only difference is that we discard indexing of single stopwords, and instead introduce indexing of bigrams based on stopwords. This difference means that we are faced with an even large vocabulary, which can be problematic if not properly handled. However, we have the advantage of holding significant smaller posting lists.

By discarding indexing of stopwords in our index we can save ourselves for considerable large posting lists in the occurrences part. This is one of the main advantages we achieve with the usage of the Bigram index. This advantage means that we can retrieve posting lists faster from external devices (ignoring disk access rates) along with much faster decoding of such large posting lists. Consider the posting list for term "the", which is many times the number of documents in the index. In addition there is the word positions that ordinary needs to be stored inside each entry of a posting list (given the usage of word oriented index). So, as can be seen, discarding indexing of stopwords is highly attractive.

However, a drawback could be the restriction of querying a single stopword. Even though such a case is very rare and thus gives no semantic meaning, it could happen. In such a case, the user query would be routed to another index or given a error message in return. So, a restriction in the Bigram index is that querying after single stopwords, such as "in" or "the" would not yield any results since those terms is not in the vocabulary. However, word sequences containing those terms is indexed. Other drawbacks worth mention is the vocabulary that will have to hold a much larger set of terms in contrast to the standard inverted index. This would in turn also mean that there would be less main memory available to hold postings lists and such.

Lets illustrate the threshold for the amount of terms in vocabulary. Assuming that a term entry in vocabulary uses 33 bytes. In this the actual term representation takes 168 bits, where we allocate an average Java char array of size 10 based on UTF-8[28] representation, in addition to a Lucene VInt byte to hold the length of the term. Next we have a 64 bit to hold the term count, and a 32 bit pointer to the posting list. Altogether 264 bits $\Longrightarrow$ 33 bytes. Next we assume we have 2 GB of available main memory, which is not unusual on a standard desktop computer these days. In Equation 8 below we can see that based on the assumptions made above we can hold 60 606 060 terms in memory.

$$\frac{2 \times 10^9}{33} \approx 60\ 606\ 060\ \textit{terms in memory} \tag{8}$$

Now, this number is a little unrealistic since we then would not have any memory left to hold posting lists and/or results. Also, it would be a huge waste to hold all these terms in memory, since only a fraction would be accessed frequent enough. Given the approach Lucene use regarding vocabulary and memory, we would hold an "index" over the vocabulary terms. Say

[28] `http://en.wikipedia.org/wiki/UTF-8`

that we hold each 512^{th} entry in main memory, thus lowering the memory usage for a vocabulary of 60 606 060 terms into to $\frac{60\ 606\ 060}{512} \approx 118\ 371$ terms. This approach dramatically lowers the main memory usage by the vocabulary. There should be noted that in addition to hold each 512^{th} entry in main memory, there would also be some mechanism to fetch and hold high frequent terms, since their very likely to be queried.

6.5 Document collection

In our experiment we wanted to have a well acknowledge and used collection, which has been a part of many other benchmarkings of various kind. Based on the research on IR-systems and all of it characteristics, we found the TREC collection very attractive. The TREC collection has outlined itself as one "standard testbed" in IR-systems, and thus was a natural choice for this experiment.

The scope of this master thesis is to look at phrase searching in text indexes, and we know phrase searching is widely used on web collections due to the index size. Thus, our selection for a testbed needs to reflect this. The TREC GOV2 collection meets these requirements, and is the collection of focus in this experiment. The importance of a good document collection is substantial, since it would effect the effectiveness and efficiency if the collection did not reflect real life data. If the IR-system is angled into a specific domain of interest, then the testbed collection should reflect that too.

Below we will further inspect the document collection of our choice, and how we managed to extract a pure text stream which is then passed into the Analyzer.

6.5.1 TREC GOV2

The TREC GOV2 collection is a standard testbed within the IR research community, and thus makes it highly attractive for our experiment. It provides a crawl of all `.gov` domains from 2004, altogether 426GB of web data.

Now, one of the major characteristics that makes this collection attractive is that its a web crawl, and therefore provides real life web data. As already known, phrase searching is commonly used on web data, since the waste amount of information on the web makes the querying for specific data harder, and thus requires more discriminating queries. The GOV2 collection has already been extensive used within both searching and indexing benchmarking, and since we shall, into some degree, explore both processes we found it to suite our experiment quite well.

6.5.2 Parsing

A GOV2 document is not a plain HTML webpage, as it is more an encapsulated webpage surrounded by SGML tags that provides additional metadata to the webpage, including the HTTP response header. In Figure 18 there is a slightly stripped GOV2 document that illustrates the usage of SGML tags to encapsulate metadata of the webpage, and then the HTML codes for the actual webpage. One can see that a GOV2 document is encapsulated between `<DOC>` and `</DOC>`.

Figure 18: A slightly stripped GOV2 document

```
<DOC>
<DOCNO>GX000-00-0002361</DOCNO>
<DOCHDR>
http://nuclear.gov
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://nuclear.gov/default.html
Date: Tue, 09 Dec 2003 21:21:34 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Wed, 08 Aug 2001 12:37:51 GMT
ETag: "80f9e8ef620c11:902"
Content-Length: 1954
</DOCHDR>
<html lang="en">
<head>
<title>Office of Nuclear Energy, Science &#38 Technology - DOE</title>
</head>
[HTML tags and information here]
</HTML>
</DOC>
```

The process of extracting the textual information, and not the HTML tags and other non-interesting data, is performed based on a library provided by the MG4J project (Managing Gigabytes for Java) described in [27]. What we will describe here can be outlined in Figure 17, where our focus is the entry process which deals with the parsing of documents. It is a preprocessing that must occur to be able to feed the rest of the IR-system with a pure text stream. In the MG4J project there is a subpackage which has a single purpose, which is to parse files containing GOV2 documents. A single file contains up to thousand GOV2 documents, and therefore the parser

provided by the MG4J project only needs to have a list of files to be able to parse the documents, and then it will notify whenever it has extracted text ready to be fetched. The parser is also able to parse and decode GZip[29] compressed files, since most GOV2 files is compressed by the UNIX *compress* software to save disk space. GZip is the free implementation of the old UNIX *compress*. In other words, the GOV2 parser is quite flexible.

One concern with the usage of the MG4Js GOV2 parser was its performance. Not keeping the search engine waiting for incoming text to index is a requirement, so we wanted a parser that could maintain a document feed to the indexer. Before we went all the way to implement the GOV2 parser, we wanted to test how many documents it could parse pr second. So, we created a simple Java program that triggered a parser process, and ordered the parser to parse 1000 files, each containing approximately 1000 GOV2 documents. We performed millisecond sampling before starting to parse the documents, and after the parser had completed. Then the difference between the samples was calculated. After three tests of the parser, whereas we wiped out the internal cache and paging made my the underlying OS (Ubuntu Linux in our case) for each time, we measured the overall parsing pr second to be ≈ 750 GOV2 documents. This speed is sufficient enough for our search engine, so we settled with the usage of this parser.

In the MG4Js GOV2 parser there is possible to adjust the buffer size used in the parsing process. We have experimented with various buffers to find out which one gave the best results. The result of ≈ 750 GOV2 documents was based on a buffer of size 16KB. In Table 11, we can see our results along with the range of buffer sizes which we experimented with. Between each run we cleared the internal cache and paging by the underlying operating system, so that it would not affect the results.

In Figure 19 we have created a graph with the buffer sizes along the x-axis, and the document-pr-second (dps) rate along the y-axis. The table and figure clearly outline that using 16KB (16383 bytes) as buffer size will give the best performance.

One may notice the performance decrease when using buffer sizes of more or less than 16KB. The believed reason for this is that allocation larger buffers would imply larger latency for the parser to receive data. In case of a 64KB buffer size, the parser would for each GOV2 document allocate the usage of 64KB, while the GOV2 document only needs 16KB. This large buffer have a negative impact on the parsing process, since the reader of the GOV2

[29] `http://en.wikipedia.org/wiki/Gzip` and `http://en.wikipedia.org/wiki/Compress`

MG4J GOV2 parser tests			
Buffer size (KB)	Files	Documents	DPS
1024	x	x	x
2048	x	x	x
4096	100	89771	680
8192	100	89771	680
16384	100	89771	685
32768	100	89771	669
65536	100	89771	568

Table 11: Table presenting some test results for fetching and parse TREC GOV2 files, and identify GOV2 documents. Tested with various buffer sizes to see the impact on the overall dps (document-pr-second rate).

document then will wait until the whole GOV2 document is read into the buffer. Ergo, the rest of the parsing process needs to wait for the input data. Also, choosing a too low buffer size imply that the buffer is filled up too fast. This means that the parser receives data fast - maybe too fast, implying that the parser will be too loaded for handling the data, and therefore the I/O systems would have to wait for the buffer to be emptied. A mean value is desireable, and in this case 16KB was choosen.

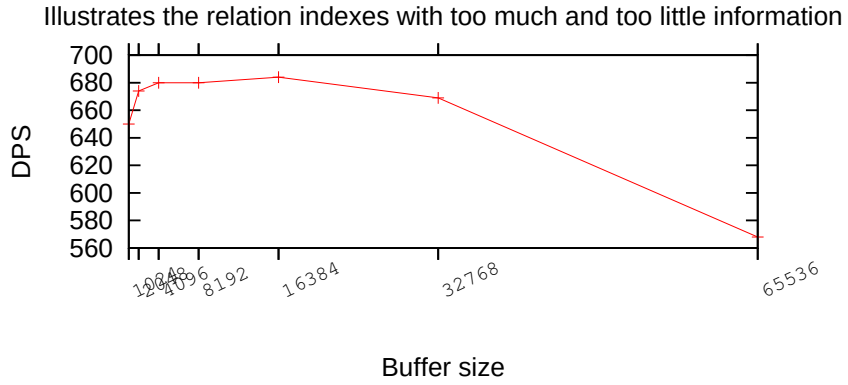

Figure 19: The process of constructing a bigram index, from the point of documents to the writing of the inverted index.

6.6 Implementation

As any respectful search engine, we needed a name for our search engine to achieve a certain level of personality. In our experiment the name is *Baldr*, named after the well known god in Norse mythology, and son of the chief

god Odin. The name *Baldr* was chosen by the author in a random approach, were the author read some articles on Wikipeda[30] and the name *Baldr* emphasized itself. *Baldr* - our search engine - will be outlined into more detail in this section.

We have previously introduced the main goal of this experiment along with the index structures, document parsing and analyzing. Now, we shall look at more specific details for the running code of our search engine, and how we solved the challenges on our path, such as keeping the indexer busy with indexing the documents. One major challenge in the indexing process is to achieve a satisfactory documents-pr-second rate (dps), that is the throughput of the indexer. Even though this was not a very important goal of this experiment, we felt an urge to achieve this. One approach was to have an efficient buffer management in order to reduce *Garbage Collection* performed by the Java Virtual Machine. Garbage collection of unreferenced objects in the Heap of Java Virtual Machine is costly, and should thus be avoided if possible. Also, the garbage collection procedure is periodically runned, so there is a possibility to run out of available memory between each run. In other terms, creation and deletion of objects should not occur frequently along the lifetime of the indexing process. More on this subject in Section 6.6.1 where we also describe our bufferpool structure. Also, the Apache Lucene v2.3 has better support than previous versions to keep the garbage collection at a minimal in that it allows reassigning content to Fields, and thus enables Document objects to be reused.

In the searching process we focus on measuring the time used to query a predefined set of phrase queries. This is our main concern, since we then can observe the responsiveness of the IR-system in relation to the query. We used a query set made available on the TREC TeraByte Track website[31]. This query set contained altogether 50 000 queries of both single term and potentially phrase queries. We say "potential" phrase queries, since queries in the query set is basically sequences of words, not explicitly phrase queries. To filter out queries which we could use in our query test, we parsed the query set file and discarded each query of length one, leaving only queries qualified as phrase queries. Of 50 000 queries we had only 38 799 valid ones, which was initially used in this experiment. Of 38 799 queries the search engine only had results for 7922, roughly 20%. More on the results from this experiment in Section 7.

[30] `http://www.wikipedia.org`
[31] `http://trec.nist.gov/data/terabyte/05/05.efficiency_topics.gz`

6.6.1 Overall design principles

In the beginning of the development phase of Baldr we had the already mention goals and requirements to follow. As in any development project there is design principles to sort out, and we wanted to face those before we started any major coding. It is important to note that the implementation needs to support two elementary tasks; indexing and searching.

- **Write search engine in Java**. In Table 10, we also noted this. This design principle is highly supported by the fact that our search technology framework, Apache Lucene, is written in Java (and also ported to other languages). Also, the Java programming language is well understood and handled by the author. Usually, C/C++ is the preferred language for such applications, but we chose not to for the reasons mentioned.

- **Take advantage of Apache Lucene library**. This is thoroughly presented in Section 6.3, and needs no further introduction.

- **Use Apache Commons Collection implementations.** The Apache Common Collection library is a package containing more powerfull collection implementations than those provided by the *Java Collection Framework*[32]. This library was used as our buffers in our bufferpool implementation.

- **Take advantage of a bufferpool to hold reusable Lucene Documents**. This principle yield efficiency gains in the indexing process, in that we reused already created Lucene Document objects and thus save ourselves from garbage collection the document objects.

- **Built with phrase querying in mind**. Both the indexing and the searching part of this implementation is built with efficient phrase querying in mind. In the subpackage regarding indexing this imply efficient handling and identifying of index terms, and regarding the approach to query the index in the searching subpackage. In Section 6.3.2 we introduced different query evaluation approaches in the searching subpackage which is taken advantage of in this design principle. The Bigram Index is heavy used in the indexer subpackage to support this principle.

.

[32] `http://java.sun.com/docs/books/tutorial/collections/index.html`

6.6.2 In the beginning

In the very beginning of the development of this search engine, we did not had a clear vision of what we wanted to do besides experimenting with a search engine and its abilities to handle phrase queries. What was not clear was which search engine to use, or if it would be a necessity to develop our own search engine. Since the author was somewhat familiar with the Apache Lucene search engine library, and the supervisors agreed that we should avoid spending time constructing our own search engine if we had the chance - then the choice of using Lucene was early concluded.

It took some time before we agreed upon what exactly to do in our experiment, besides the somewhat abstract goal / motivation described in Chapter 2. After some reading and a good conception of the underlying concepts introduced in [23], underpinned by research papers regarding phrase searching, pair of words and the inverted index structure in [4, 5, 12, 13] and [17], we concluded that we should have a focus on the characteristics of bigrams and how to use them in order to provide more efficient phrase searching capabilities. Especially the nextword structure described in Section 5.1 and introduced in [13] was highly attractive. We searched for implementations of this nextword structure, but none was to be found. As a consequence, we chose to experiment with our own somewhat similar structure - the Bigram index introduced in 6.4. In Figure 20 there is given a very abstract view of the main modules in Baldr - just to simplify the overall comprehension of the search engine.

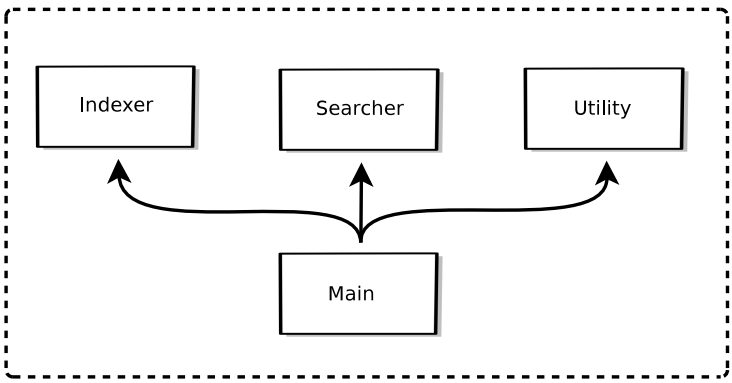

Figure 20: Illustrates the main modules in the Baldr search engine from a very abstract view. Made only for perceptual purposes

Main entry The main entry is basically the Java file which is the glue in the system. It is initialized by the usage of Apache Ant[33], an automating software build tool, which simplifies the process of compiling, running and cleaning up the code. In Figure 20, the "main" component is the main entry in Baldr, and is responsible for providing the underlying modules such as Indexer, Searcher and utility tools to the initializer. Apache Ant is in Baldr supported by an XML file, there each possible action towards the search engine is specified, along with the appropriate arguments to map a request to an action in Baldr.

One design pattern we took advantage of in Baldr is the usage of a "Singleton". A Singleton is in simple terms a *"global point of access"*. What distances a Singleton object from another object is that there is only **one** instance of a Singleton, and never two or more. Our Singleton holds references to our Indexer and Searcher, and thus provides simple access to those resources for our main entry. Another feature is that instances of the Searcher or Indexer would only be instantiated whenever a request for the object is retrieved, and all subsequent request would only return references to the already existing object.

Also, another somewhat similar class in Baldr is named Definitions.java. This class holds a large set of static variables, thus centralizing and unifying access to global parameters and other needed references. It may be argued that some of the parameters and references should be held in our Singleton object, however the class name (Definitions) somewhat defines the scope of the parameters and references, and is thereby naturally stored there.

Indexer Our approach to develop Baldr started with the indexing part, since it is the first part that needs to function to create an index. At that time we did not have the document collection we initially ended using, however the author had access to an very old TREC collection from 1994 which was used as a testbed for the early versions of the indexer. Problems regarding efficiency was early experienced, such as slow documents-pr-second rates as a consequence of frequent garbage collection. We created and "deleted" document objects for each new TREC document. Also, our parser for these old TREC documents was not very efficient, as it used many Java String objects in an ineffective approach, resulting in high garbage collection costs. Altogether, the efficiency for this indexer was not great, but we harvested some experience - especially the author.

Spending many hours reading the source code for the Apache Lucene trying

[33] `http://ant.apache.org`

to understand the stuff beneath the hood, paid of in the long run. Being able to sufficiently modify the parameters which Lucene uses in the indexing process turned out to yield sufficiently performance gains. Adjusting the parameters for main memory usage, and the threshold for number of documents to be indexed before the index is flushed to disk had an huge impact on the dps throughput (document-pr-second rate). These parameters varies based on the underlying hardware; amount of main memory, CPU (cycles, L1 & L2 cache, cores), I/O systems, etc. On our hardware we suffered from some slow CPU and I/O system, so we needed to take this into consideration when adjusting the search engine parameters.

One challenge in our search engine was to balance the input stream of GOV2 documents with Baldr's ability to consume the documents objects, in addition to utilize the main memory in a good manner. We sorted this approach by having the parsing of GOV2 documents in its own thread, and a buffer queue to hold all documents processed by the parsing thread. Also, the indexing procedure is fired off in its own thread. In Figure 21, these working threads are illustrated. Keeping both of these threads busy in their own way was a goal to achieve a good dps throughput. One aspect within threads which can sufficiently slow down the performance is the timing issues related to data synchronization. In Figure 21 the shared buffer between the "producer" and the "consumer" can be seen, and all inserts and deletes require locking the buffer from external modification. If our parser inserts a new document object, then the buffer will be locked. Hence the indexing procedure will not be able to retrieve and delete a document object from the same buffer at the same time. The indexing procedure would then have to wait until the parser has finished inserting the document object. This is a core challenge within threads, and introduces some extra time delay and complexity.

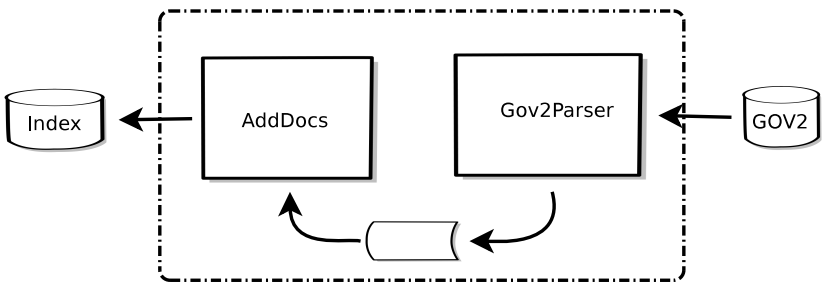

Figure 21: Illustrating the two working threads in our Indexer, and how they are connected

Searcher Having finished the Indexer implementation, thus being able to construct a complete index of the document collection, we needed to construct the other core part to enable searching in the index. Of great importance was to support the experiment goal to extract measurements regarding our large scale query tests. One thought was that we needed to "simulate" more than one user to query Baldr, so we needed threads which had their own query sets, and could execute queries simultaneous. These queries would then query Baldr and collection measurements along the way. We did not want the collection of measurements to affect the performance of Baldr or anything else, so we just stored time usage and numbers in an internal Java HashMap data structure, without any computation performed on it.

We used two threads, each having half of the original query set. For each query we stored two things: the query in its original form as the HashMap key, and a vector as the value. The vector contains the rewritten query based on the Analyzer used, the hit count retrieved from Baldr, and the time Baldr used to process the query. Recall from Section 6.6 that we have 38 799 usable queries, and only 7922 gave hits in return. This means that the HashMap containing the measurements will only contain 7922 entries, whereas each entry is a vector of three elements (String, Int, Long). Based on the 7922 result entries, we have calculated that the average query is 2.5 words, where the average word is 7 characters. See Appendix B.1 to see how this was explored. Now, assuming an String is on average $\lceil 2.5 \times 7 \rceil \approx 18$ chars (36 bytes; Java char â 16-bits) plus the overhead of an int (4 bytes) presenting the String length, altogether 40 bytes. And, another Int consuming 4 bytes and a Long consuming 8 bytes. In the end, a vector entry consumes approximately $40 + 4 + 8 = 52$ bytes, and the whole HashMap holding the measurements takes $52 \times 7922 = 411944$ bytes, or $\approx$ 412KB. Note however that additional space consumption in regards to the structure of the HashMap is needed, such as the load factor, the index over each key and buckets assignation. Therefore this usage approximation is not to be interpreted as final. This data size would only occupy a fraction of the available memory, and thus hold a minimal affect on the performance of Baldr.

When the Searcher is finished, the measurements is collected from the threads and concatenated into a single structure, and then written to a result file on disk.

Utility The utility module does not provide any single major functionality, such as the Searcher or Indexer. However, we have chosen to store the small modules that we have been used to extract and test the system with here. One quite handy module is the HighFrequentTerms, which loads a

specified Lucene index and extracts the vocabulary. For each term in the vocabulary, there is also extracted the term frequency, where word positions is considered a hit. Based on this module we were able to locate the 10 and 100 most frequent terms in the document collection, and thus use them in order to construct different kinds of Bigram indexes to experiment on.

Another module in the utility module is the TestTRECDocReader. This module has a single purpose of taking in a set of TREC GOV2 files, and then extract the textual information. In these steps there are performed measurements to calculate the average performance of the TREC GOV2 parser being used. We used this module to calculate the dps rate outlined in Section 6.5.2.

6.6.3 Architecture of Baldr

The architecture of Baldr is quite straightforward, where our concerns have been on the two major components: Indexer and Searcher. A very abstract view can be seen in Figure 20 on page 91. With this figure in mind, we have created a more UML similar diagram trying to visually illustrate the relations and such between classes and packages (speaking in Java terms). This diagram can be seen in Figure 22.

One observation from Figure 22 is that the Indexer part is not encapsulated into its own Java package, such as the Searcher part. This is basically a design flaw from the early stages in the development of this search engine, and should ideally be rewritten. However, due to time restrictions, this has not been a priority.

Bufferpool One other architectural aspect in Baldr worth mention is our solution to address the high load of construction and deletion of *GOV2Documents* in the Indexer. Outlined in Section 6.6.1 is our bufferpool implementation, which is in simple terms two buckets representing document holders (buffers). The reason for using two buffers is based on the two states our GOV2Document objects can have; "free" and "ready".

- "free" is a state when a document object has already been indexed, or is unused. This state signals that the object can be retrieved from the buffer, and used in the Gov2Parser thread to hold a GOV2 document extracted from the parsing of GOV2 collection. A "free" document object may hold the information from a previous (already indexed) GOV2 document, however that information would be overwritten when the Gov2Parser sets new values on the document object. After new values

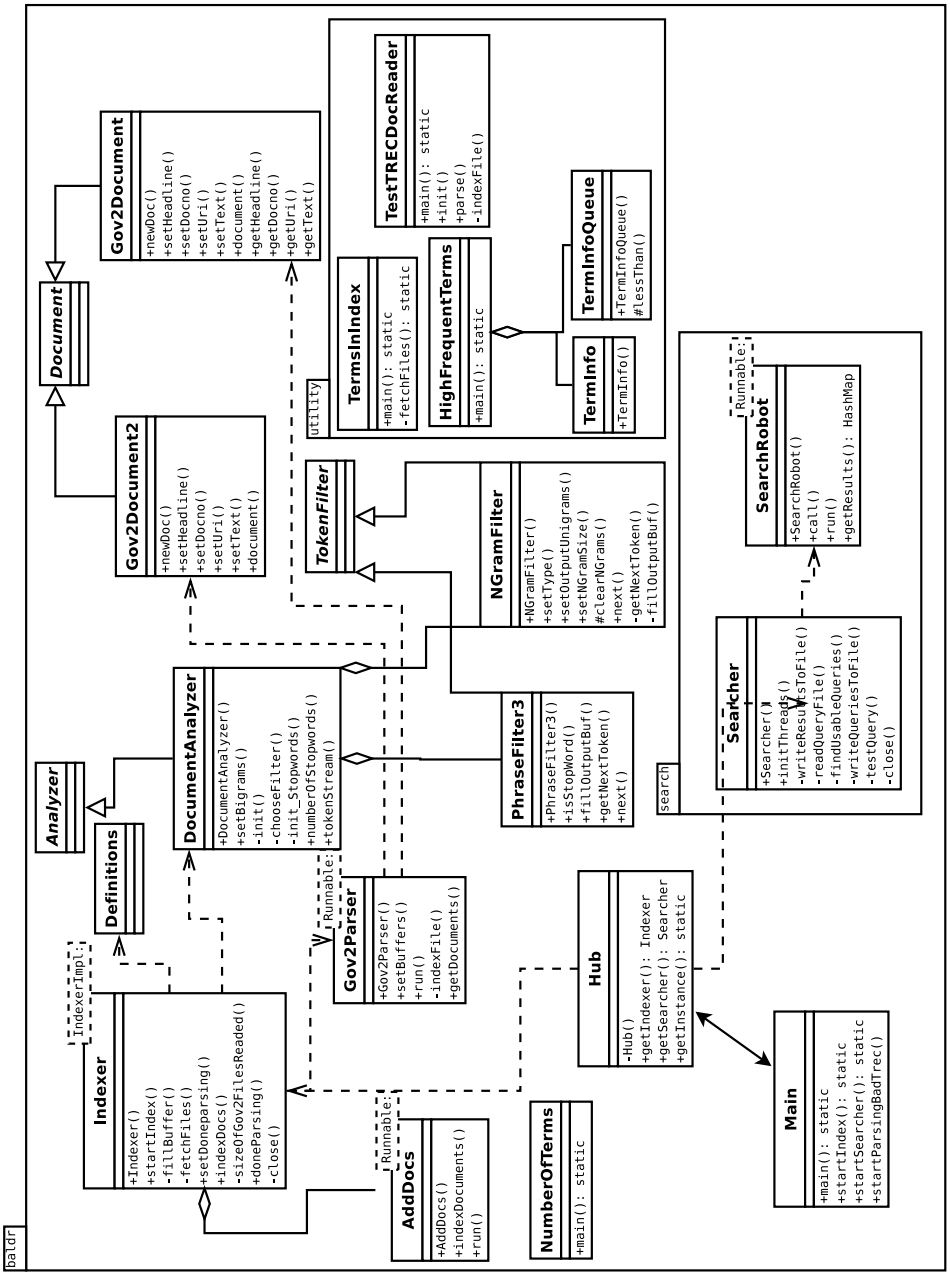

Figure 22: Overall architecture illustration of the Baldr search engine created in this experiment.

has been set on the document object, it changes state to "ready" and is inserted into the appropriate bucket.

- "ready" is a state when a document object is ready to be indexed. By

this, the document object parameters has been set with information from a parsed GOV2 document based on the GOV2 collection. The internal class "AddDocs" illustrated in Figure 22 will fetch document objects from the "ready" bucket, and then index the information each object holds as a single document. After the document object has been indexed, the object is appended to the "free" bucket and thus changes state.

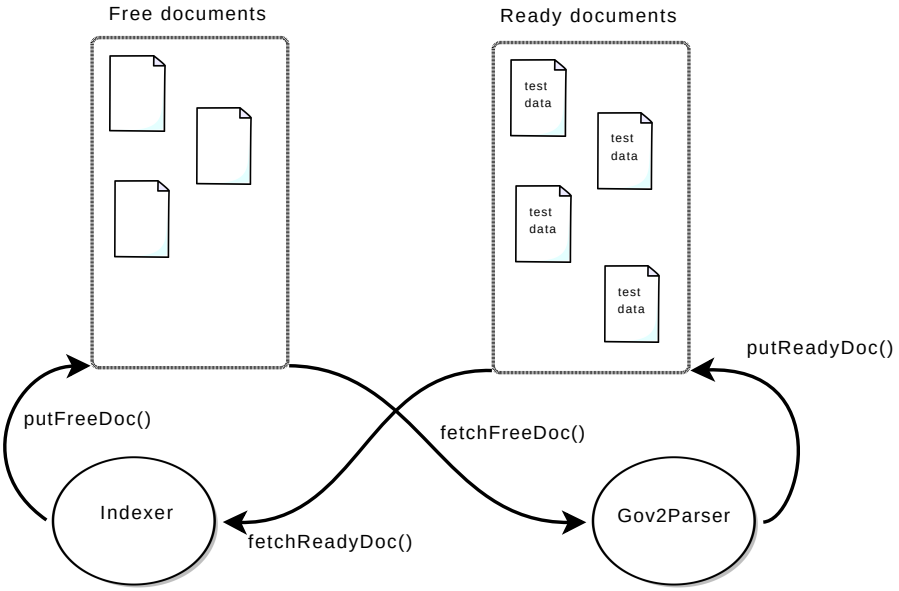

Figure 23: Illustration of the architecture of our bufferpool, and how operations is performed on them.

In Figure 23, the two buckets "free" and "ready" is illustrated along with the processes operating on them. When Baldr is initialized, then the "free" bucket is filled with empty GOV2Document objects. After the parser has started processing and the stream of parsed GOV2 documents is available, then the Gov2Parser fetches a "free" GOV2Document object from the "free" bucket. The Gov2Parser so fills it with the parsed GOV2 document, then appends it to the "ready" bucket. Another thread named "AddDocs" will continuous fetch GOV2Document objects from the "ready" bucket, and then index them. After a GOV2Document object has been indexed it will be appended to the "free" bucket, as it is free to be reused. This is the "circle of life" in Baldr with respect to the Indexing process.

The performance gains this approach introduced for Baldr proved significant, since we managed to keep garbage collection at a minimum. The creation of new objects in Java introduces some time delay in response to

the operations needed to create an object. Allocating the memory on the Java heap, and the initialization of all the defined variables, and evaluation of other defined actions based on the object *constructor* is time-consuming operations. These factors further delay the process of creating new objects, thus signalizing the performance gains in minimizing such operations.

The two buckets (buffers) is two instances of the `BoundedFifoBuffer` provided by the Apache Commons Collection[34] library. This buffer implementation is very efficient, providing operations such as `add()`, `remove()` and `get()` in $O(1)$ time. The `BoundedFifoBuffer` is of fixed size, thus saving dynamic memory allocation. Our two buffers is of size $n = 5000$; capable of holding 5000 GOV2Document objects each. However, when initialization of the Indexer the "free" bucket are filled with GOV2Document objects, thus saving the creation of objects during the indexing process.

6.6.4 Possible future improvements of implementation

During this experiment we have identified some aspects that can de denoted as "future" improvements. Some of these aspects affects the performance of Baldr, and should therefore be addressed. However, due to time constraints this was not performed during this master thesis. We will first introduce the ones which affects the performance.

I/O utilization Baldr is very much an I/O bound application, since all operations is heavily affected by the speed of retrieving data from disk. One improvement in Baldr could be to take advantage of the new I/O APIs in Java (NIO), which is specifically directed towards intensive I/O operations.

Less garbage collection In the indexing process there is continuous created Token objects to represent indexable terms (searchable units). The creation of Token objects is performed in the Analyzer, and therefore should our Analyzer take advantage of reusing already created Token objects. As of Apache Lucene version 2.3, reusing of objects such as Tokens, Documents and Fields is provided, and our Analyzer should take advantage of this. Further reduce the garbage collection performed during indexing would severely increase the throughput of Baldr.

Parser Further improving the parser to provide even faster parsing would increase the input of documents to the Indexer, thus the Indexer would not need to wait for documents to index. Possibly create a customized parser

[34] http://commons.apache.org/collections/

would yield some advantages, and utilize more I/O intensive APIs such as the Java NIO.

7 Results

In this chapter we will have an in-depth analyzation of the statistics provided by the collected measurements from the experiments performed on Baldr. Some of these statistics will be calculated from the measurements, and then provided in a comprehensible manner.

In this chapter we will refer to our standard inverted index as our *Unigram index*, since it is created with single words in mind. In case of our bigram indexes, we will refer to them as *Bigram10* and *Bigram100*, whereas the number signalize the amount of stopwords used.

All results could have potential sources of errors, and those sources should be discussed and addressed. We will outline the identified sources, and discuss how we approached them. In Section 6.1 we described our test platform, summarized in Table 10 on page 66, and all results provided in this chapter is highly influenced by that particular test platform. Performing the same tests on another platform with different I/O systems, CPU, RAM, etc, would have given other results. Regardless of the difference between results from one platform to another, as long as there is possible to extract the overall results that prove or disprove the impact of our design.

Before each test was executed, a Linux[35] specific command was executed. The main goal with this command is to clear the internal cache and paging carried out by the underlying operating system. To retrieve reliable measurements from each test, this was a necessity.

```
$# echo 3 > /proc/sys/vm/drop_caches
```

Now, the underlying operating system would have to read all new files from external devices, such as the hard drive. Executing our tests after the execution of the command above, would add the time delay introduced by the I/O traffic, which is the most realistic approach. This signalize that these tests would give more reliable results.

We performed a very quick and simple test to illustrate the difference between clearing and not clearing the internal cache in the OS. In Section B.2 in appendix, there is a code snippet outlined which is the Python program we used here. Combined with the usage of the *NIX specific `time` command, we have a powerful tool. We input a file - here named `99` - which is 17MB in size. The first try is after we have run the Linux command above - notice the `sys` time. Operations that goes through the kernel would affect the `sys` time; operations such as opening, reading and closing files.

[35] This only works for the 2.6 Linux kernel

```
fellingh: #; time python test.py 99
real    0m2.098s
user    0m0.104s
sys     0m0.060s
```

The second try is runned right after the first one, without the clearing of internal cache and paging. Notice how the `sys` time has approximately halved.

```
fellingh: #; time python test.py 99
real    0m0.153s
user    0m0.124s
sys     0m0.028s
```

The internal caching made by the underlying operating system is quite handy in regards to normal use of the operating system, but for our experiment it would shatter our results. So, it is crucial that we approached this.

In Section 7.1 we will inspect the measurements collected in regards to the indexing procedure. In Section 7.2 we will present the query results, based on the searching procedure. In Section 7.3 we evaluated the results and measurements presented in Section 7.1 and Section 7.2.

7.1 Index construction

The index construction is our first test in our experiment, which provides us with detail knowledge regarding the performance constraints and characteristics introduced by our design.

7.1.1 Constructing the index

Constructing the index was performed by running a simple command towards the Apache Ant building tool, which glues the requested libraries and parameters together. We made available some simple commands to create different indexes, whereas one for the standard inverted index without any stopword identification nor stopping, along with commands for the other indexes as well.

Our approach towards collecting the stopwords and take advantage of their characteristics included the execution of `HighFrequentTerms` in the utility subpackage - see Figure 22. Running this module against the standard inverted index extracts the 100 most frequent stopwords, which we inserted into a file called `stop_words.txt`. When constructing the bigram index based on 10 stopwords, only 10 of the most frequent stopwords is used to

identify and discard terms from the document collection, as well as create new terms. The same goes for the bigram index based on 100 stopwords. When only using the top 10 terms, we needed to comment out the other 90 terms using the commonly used # sign. Ideally the software should have taken this into consideration, but it was never prioritized.

In Table 12 we see the results collected from the index constructions performed.

Index construction					
	Average time	DPS	Size	Documents	Terms
Unigram	32 min 36 sec	≈ 491	1.8GB	960 223	5 797 853
Bigram10	48 min 34 sec	≈ 330	2.5GB	960 223	10 744 459
Bigram100	58 min 14 sec	≈ 275	2.9GB	960 223	17 352 061

Table 12: Table presenting some statistics regarding the index construction of three different runs. The unigram index contains single stopwords in the vocabulary, while the bigram indexes do not.

From Table 12 we can see that all indexes have the same document numbers - which is logical and good, since they all used the same document collection. The construction of the standard inverted index without any concern for the stopwords (using SimpleAnalyzer) took 32 minutes and 36 seconds, resulting in a DPS rate at 491 documents. The processing steps performed for each document are:

- parse a GOV2 document from the collection
- fetch a "free" `GOV2Document`, then insert the parsed GOV2 document data. At last, add the `GOV2Document` to the "ready" bucket.
- Add the "ready" `GOV2Document` to the Apache Lucene
- Apache Lucene will run the `GOV2Document` through the Analyzer, and the output Tokens will be indexed

Observing the bigram indexes for 10 and 100 stopwords, we can see that the DPS rate somewhat decreases. This is actually logical and quite understanding, as both the complexity and the computation increases in the Analyzer used to construct the Bigram index. In the standard inverted index the SimpleAnalyzer is used to identify Token objects, and SimpleAnalyzer's policy is to create a new Token whenever a non-character is found. However, the Bigram index Analyzer (DocumentAnalyzer) has a somewhat more complex policy, and thus requires more processing. And, as an additional note, the more stopwords identified, the more bigram Tokens would be produced.

Also note the difference between each index's disk space consumption. Knowing that our document collection is ≈ 16 GB, then the 1.8 GB occupied by the standard inverted index is not that much (≈ 11% of the document collection). The Bigram index with 10 stopwords occupy 2.5 GB, increasing the ratio between index and collection to ≈ 15%. The Bigram index with 100 stopwords occupy 2.9 GB, further increasing the ratio between index and collection to ≈ 18%.

This drastic increase in disk space is a consequence of the amount of vocabulary entries, and thus the equal increase in posting lists. However, the posting lists length has decreased, which is the effect we are searching for. To create a comprehensible illustration to see the effects of our Bigram index, we created a graph with the 100 most frequent terms and their frequency for each index constructed. This graph can be found in Figure 24.

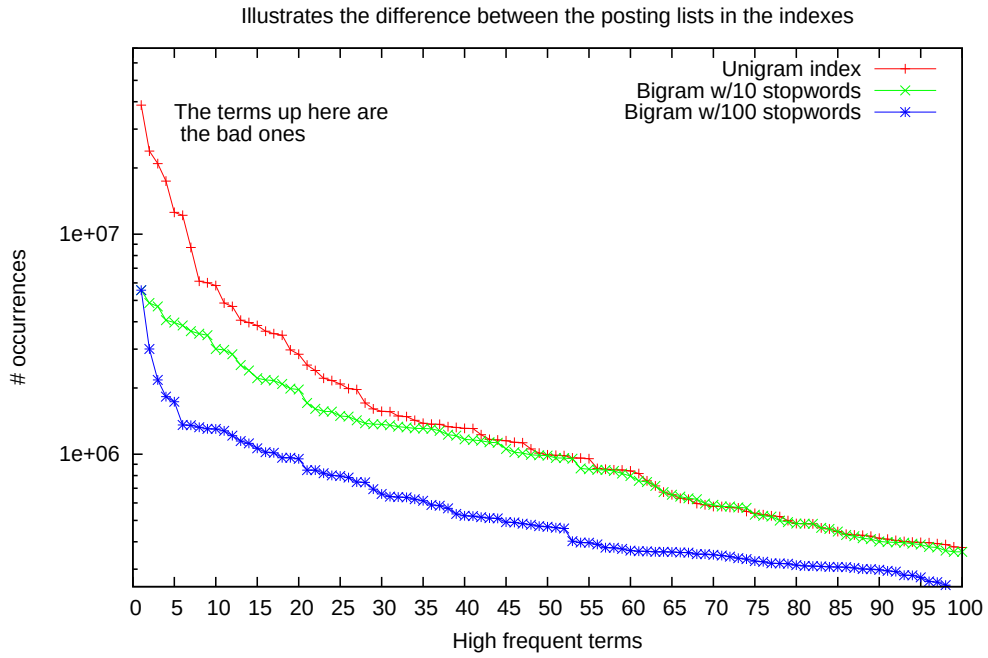

Figure 24: A graph illustrating, for each index, the 100 most frequent terms and their occurrences. Notice the gap between the bigram curves and the unigram curves at the leftmost side of the x-axis.

In Figure 24 the x-axis is the terms (searchable units), in decreasing order based on the frequency, and the y-axis represents the frequency for a term. One obvious observation is that the standard inverted index curve has a significant higher frequency for the most frequent terms, as opposed to the curves belonging to the Bigram indexes, consisting of 10 and 100 stopwords. This graph summarize the main purpose with our Bigram index in an ele-

gant way, and the gains are graphically outlined.

One important factor to have in mind when inspecting Figure 24, is that the Bigram indexes holds a quite large set of vocabulary entries. From the Table 12, we can see that the Bigram10 index has approximately 85% more vocabulary entries than the standard inverted index. And the Bigram100 index has approximately 200% more vocabulary entries than the standard inverted index. This factor is a drawback with the usage of Bigram index. Even so, the frequently accessed posting lists are substantial smaller, hence the I/O bandwidth and processing needs in order to cope with such large posting lists is prevented.

7.1.2 Source of error

In the process of index construction there are some potential sources of errors, and those should be explored. The results credibility would be debatable if we chose to ignore factors such as sources of errors, so in response to defend our results it is crucial to highlight all potential sources of errors.

One early introduced and thoroughly explored source of error is the caching and paging made by the underlying operating system. However, this potential source of error has been addressed in all operations where it is a required.

One source of error which the author early experienced was that the list of high frequent terms (stopwords) was not accurately calculated in the `HighFrequentTerms`. The early approach was to count *only* the number of documents which the term occurred in, without any regards to the number of occurrences inside each document. After this source of error was discovered and addressed, we extracted a new accurate stopword list. The consequence of not approaching this error would have had an impact on the results retrieved, as the stopwords is a central factor in the construction of the Bigram indexes.

Another source of error which the author experienced was that the Bigram indexes was not reflecting the content of the document collection, as there were some queries on the standard inverted index that did not give any results on the Bigram indexes. This error was quite a nutcracker, as there was no obvious errors in the code. The author compared the effected documents across indexes and the document collection, and noticed in the end that only the Bigram indexes were effected. A common characteristic for Bigram indexes is that it holds more Token objects (more terms) than the standard inverted index, which was the main source of this error. Apache Lucene has a default internal integer specifying the maximum length of a Field. Recall from Section 6.3.1 and Figure 12 that a Document object (GOV2Document)

is a set of Fields, where each Field holds Token objects that represent all terms identified and in relation to that particular Field from a document. Now, if a document has many terms in a standard inverted index, then it would have even more in the Bigram index. So, in case of this error, we had breached the internal integer threshold in Lucene, and thereby the rest of the terms for the document was discarded. The default maximum Field length was set to 10 000 Tokens. We increased it to 180 000 Tokens, which was a sufficient threshold, and thus resolving the error. In another document collection, also this threshold could be too low.

Other potential sources of errors in the Indexing process could be software bugs both in the Apache Lucene or in our implementation of the Bigram index, which we have yet to discover. The Apache Lucene library is widely in use in many different applications around the world, so most of the bugs are identified and hopefully approached. However, one may never say that software is bug-free, so we can never exclude the possibility for errors related to bugs in the software. A more likely entity that could contain bugs is in our Bigram index, since it has not gone through such extensive testing and usage as the Apache Lucene library. But, so far it has proven to be reliable.

7.2 Query tests

Here we will introduce the results which we managed to extract from our query tests in our experiment. These results will provide knowledge regarding the performance of our approach towards enhancing phrase searching in text indexes.

In order to make the results as comprehensible as possible, we have constructed both graphs and tables to present them. Also, the information is simplified wherever possible.

7.2.1 Querying the indexes

Querying the indexes was performed by the Searcher part, outlined in Section 6.6.3. In the querying process we used the set of queries outlined in Section 6.6, along with the stages performed to narrow down our initial query set from 50 000 queries to 38 799 queries.

As also specified in Section 7.1, we here performed the Linux specific command clearing the cache and paging performed by the underlying operating system. Based on which type of query test we wanted to perform, we needed to take into consideration the `stop_words.txt` containing the 100 most fre-

quent terms from the standard inverted index. When querying the Bigram10 index, we needed to comment out the other 90 stopwords, and uncomment them when querying the Bigram100 index.

7.2.2 Query results

For each query run there is collected a whole lot of measurements. To extract comprehensive and understandable results, we needed to parse and perform calculations over the measurements. In Appendix C.1, we have printed the source code for our program which calculates and presents this results. It is a simple Python program, which takes the measurements file created by the Searcher as input, and outputs a simple, yet information rich presentation of the results.

Standard Inverted Index First query test was against the standard inverted index. Since no stopword considerations is needed in this test, we could simply ignore the `stop_words.txt` for now.

In Table 13 below, we have the statistics calculated and presented by our Python program.

Query testing unigram index		
Query length	**Occurrences**	**Avg response time**
2	5352	186.94 ms
3	1848	448.39 ms
4	550	848.93 ms
5	130	1058.50 ms
6	27	1686.74 ms
7	7	1452.85 ms
8	3	1322.66 ms
9	5	2388.00 ms

Table 13: Table containing results from our query test performed on the unigram index.

In Table 13 we have three main measurements:

- *Query length* We have categorized all phrase queries in the measurements file created by the Searcher. We have phrase query lengths from 2 to 9.

- *Occurrences* We have here the number of phrase queries which gave a positive hit in Baldr. Notice that this number is related to the query length.

- *Average response time* This is an average calculation based on all the response times for each query within that query length category.

One clear observation from the table is that the longer the phrase queries is, the greater average response time is achieved. The background for this is that for each extra term in the phrase query, the whole posting list needs to be fetched and processed. Given that the extra term is a high frequent term (stopword), then the posting list would imply a lot of I/O bandwidth and processing time. The average response times outlined in the table somewhat reflect the general understanding of the standard inverted index structure.

Bigram10 Index Next query test was towards the Bigram index built with the 10 most frequent stopwords identified. This index is, from Table 12, 2.5GB in size, and contain 10 744 459 terms ($\approx$ 85% more than the amount of terms in the standard inverted index).

Query testing bigram10 index			
Query length	**Occurrences**	**Avg response time**	**Avg stopwords**
2	5352	151.14 ms	0.020740
3	1848	280.40 ms	0.222403
4	550	397.88 ms	0.587273
5	130	541.96 ms	0.946154
6	27	1299.00 ms	1.000000
7	7	737.57 ms	2.571429
8	3	431.00 ms	2.333333
9	5	1234.60 ms	3.000000

Table 14: Table containing results from our query test performed on the bigram10 index. We use the top 10 stopwords derived from our unigram index.

In Table 14 we have presented the processed measurements from the Searcher. This table is very much the same as Table 13 - with one minor difference. We have in this table in addition calculated (on average) how many stopwords there are in each phrase query. Note that the amount of stopwords used in the index (10 stopwords here) is also used in this calculation. This additional column in the table provides us with an understand of the queries used. Keeping this in mind while interpreting the average response time within a phrase query categorization, could be helpful in understanding the

results.

Observing the average reponse times in Table 14, we see the same trend as in Table 13. The longer phrase query in use, the greater response time is achieved. Again, this is quite understandable and logical. However, comparing the results for each phrase query category between this table and Table 13 shows a sufficient performance boost in the Bigram10 query test. For some categories the average response time is halved, which suggests a performance boost of approximately 50%.

Bigram100 Index This query test was approached in the same manner as the previous test, except for that we in this test used the 100 most frequent stopwords, instead of 10.

From Table 12 this index is 2.9GB in size, and contains 17 352 061 terms ($\approx$ 200% more than the standard inverted index, and $\approx$ 62% more than the Bigram10 index). So, this index is somewhat larger in terms of size and indexed terms, but due to the characteristics of the Bigram index it will hold shorter posting lists. This is graphically illustrated in Figure 24, where the Bigram100 curve shows the smallest occurrence values for its high frequency terms.

In Table 15 we have the output of our results from our measurements. As with the previous table, also this table includes the average number of stopwords in each phrase query category.

Query testing bigram100 index			
Query length	**Occurrences**	**Avg response time**	**Avg stopwords**
2	5352	127.75 ms	0.084081
3	1848	197.53 ms	0.459416
4	550	282.54 ms	0.994545
5	130	364.73 ms	1.546154
6	27	437.07 ms	2.111111
7	7	519.00 ms	3.285714
8	3	453.00 ms	4.000000
9	5	1347.80 ms	4.000000

Table 15: Table containing results from our query test performed on the bigram100 index. We use the top 100 stopwords derived from our unigram index.

Also in Table 15 we can observe the same characteristics as in the previous tables; the average response time increases in relation to the phrase query length. And, comparing this table against Table 14 signalize that we achieve

even further performance gains with the Bigram100 index. This is quite interesting, as it somewhat confirms that our approach yields performance gains in terms of phrase querying.

Comparing phrase query response time			
Query length	**Unigram**	**Bigram10**	**Bigram100**
2	186.94 ms	151.14 ms	127.75 ms
3	448.39 ms	280.40 ms	197.53 ms
4	848.93 ms	397.88 ms	282.54 ms
5	1058.50 ms	541.96 ms	364.73 ms
6	1686.74 ms	1299.00 ms	437.07 ms
7	1452.85 ms	737.57 ms	519.00 ms
8	1322.66 ms	431.00 ms	453.00 ms
9	2388.00 ms	1234.60 ms	1347.00 ms

Table 16: Table containing results derived from Table 13, Table 14 and Table 15.

In Table 16 we have summarized the average response times from each test run. This table gives an extensive view of the benefits of bigrams in case of phrase searching, and how it affects the query process.

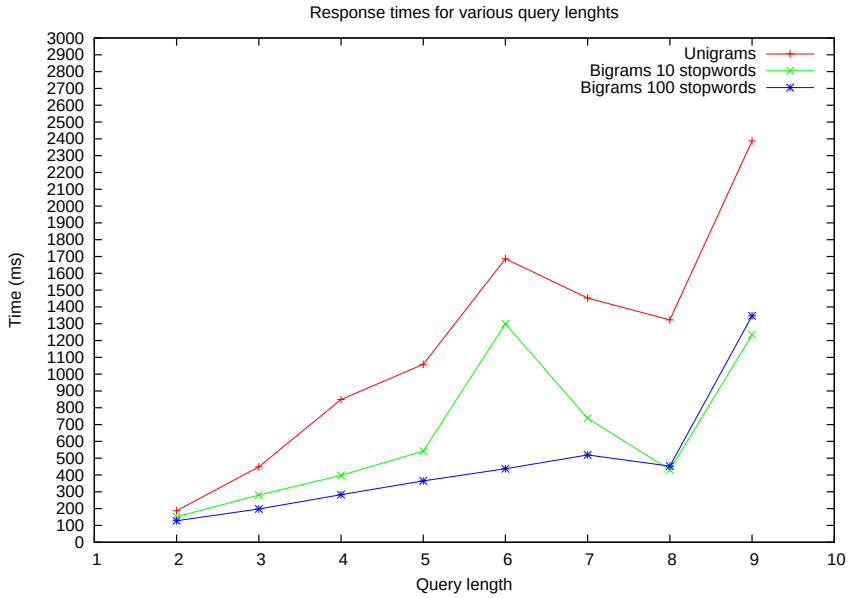

Figure 25: A graph illustrating, for each index, the increase in response time when phrase queries become longer. Is based on the results from Table 16

In Figure 25, we have created a graph based on the results presented in Table 16. This graph shows that relationship amongst phrase query lengths

and response times, and further confirms the efficiency gains introduced by the bigram index approaches, along with the utilization of stopwords.

In the IR-community there are frequent *bad* queries used as examples to illustrate different points. One of these queries is the famous *William Shakespeare's* quote from the play Hamlet: "*to be or not to be*". We have tested querying our indexes with this famous quote in Table 17.

Performance of a phrase query *Query: "to be or not to be"*		
	Time usage	**Savings**
Unigram	6081 ms	N/A
Bigram10	3029 ms	$1 - \frac{3029}{6081} \approx 50\%$
Bigram100	1136 ms	$1 - \frac{1136}{6081} \approx 81\%$

Table 17: Table illustrating the difference as well as the performance gains in case of a typical *bad* phrase query containing many stopwords.

It should be noted that this phrase query is typically referred to as a bad query, in terms of the frequent usage of stopwords. Therefore the gains of querying the Bigram indexes instead of the standard inverted index would be much clearer with queries such as these. In case of a phrase query without any stopwords at all, the gains would be more or less insignificant.

7.2.3 Sources of errors

In the process of querying our indexes, there are some factors that could potential resolve into sources for incorrect results.

As also outlined in Section 7.1 regarding sources of errors, the factor of caching by the underlying operating system should be taken serious. However, between each query test run we performed the Linux specific command introduced in the beginning of Chapter 7, thus resolving this source of error.

The vulnerability introduced by our `stop_words.txt` should also be taken into consideration here. Given that the wrong set of stopwords was being used in one of the tests, then those results presented above would be invalid. Extreme precaution was taken before each query test run, assuring that the correct set of stopwords was being used.

One observation made from the results is that the Bigram10 index performed somewhat better than the Bigram100 index in certain cases. This is best illustrated in the two lowest rows; phrase query category 8 and 9.

One theory for this is that we in Bigram100 used the 100 most frequent terms, instead of 10, thus there are more bigrams identified in the Bigram100 index. Take for example phrase queries of length 8, which we have 3 queries of. One characteristic they share in the Bigram100 index is that more bigrams could have matched within those queries, as opposed to the Bigram10 index where fewer bigrams could have had a match. This is because of that the Bigram10 index only holds bigrams of the top 10 stopwords. In case of our phrase queries of length 8, there could be bigrams which is outside the range of those top 10 stopwords. This brings along a greater likelihood for that the bigrams in the Bigram10 index have already matched in earlier phrase queries (since there are fewer of them, and they occur in more queries), and therefore has been cached inside Baldr and the underlying operating system. For the Bigram100 index however, there is a greater likelihood for that those matching bigrams has not yet been fetched, and thus the costs of fetching and decoding the posting lists comes into play.

It should be noted that the author has not verified this scenario, but it is not unlikely, so it should therefore be taken into consideration when interpreting the results. A larger set of phrase queries within those affected categories would most likely straighten out the results though.

7.3 Evaluation

Here we will evaluate the presented results from the indexing and querying tests performed in our experiment.

7.3.1 Indexing

In our indexing results presented in Section 7.1, we presented building statistics for our indexes. The gaps between each construction was presented, along with the causes for those gaps. There is an especial focus on the gaps between the standard inverted index and the Bigram indexes.

Maybe the best illustration between each index construction is in Figure 24 on page 103. In this figure, a decreasing list of "posting lists" lengths is outlined across the x-axis, starting with the largest posting lists in the index. It shows the drastic cuts in posting lists lengths introduced by our Bigram indexes. Also, Table 12 gives us some statistics for each construction, providing us with a conception of the building process.

By the introduction of the Bigram index, we clearly see the increasing disk space consumption along with the increasing vocabulary. This can be said to be the two drawbacks introduced by this approach. To cope with these

drawbacks, there is a need to take advantage of distribution. This is into some extent discussed in Section 4.3, whereas two approaches to partition the index across multiple nodes are discussed. Also worth mention is that Heap's law - discussed in Section 2.6 - also applies for the vocabulary in Bigram indexes. So, the vocabulary growth rate will straighten out over time, only a little later than with the standard inverted index. The two parameters in Heap's law, K and β, would also need some adjustments to model the vocabulary of the Bigram indexes.

7.3.2 Searching

The results presented in Section 7.2 is carefully calculated based on the measurements collected by the querying tests. These results gives insight into the querying process, and provides us with important knowledge about the influence that the various index types has.

One factor of great importance in this context is the performance in terms of response time. The response time is measured from the time the query enters the IR-system, to the time the response is retrieved (hit list). This factor, in relation to the actual phrase query used, along with its characteristics, gives valuable insight into how the search engine performs. That is, when this factor is compared across indexes.

Considering the tables with the query results presented in Section 7.2.2, we clearly see the trend introduced by the Bigram indexes. Enhanced phrase query capabilities is outlined in terms of drastic smaller response times for typical "bad" phrase queries, which contains stopwords. It should be noted that phrase queries has various degree of "badness" depending on the amount of stopwords, and the occurring stopwords frequencies.

An important thing is that our Bigram indexes only affects phrase queries that has stopwords in it, and that all other phrase queries would be better of using other approaches, such as combined approaches outlined in Chapter 5.

Below we have three graphs - one for each phrase query category. Figure 26 is based on phrase queries of length 3, and is basically a histogram, illustrating - for each index - the amount of phrase queries performing within various time ranges. The x-axis represents the response time in milliseconds, and the y-axis represents the amount of queries. Just to give a better understanding of the graphs, we can say that the larger amount of queries which is in the lower-end on the x-axis, the better that particular index performs, in terms of the query test. So, we are basically looking for a quite skew distribution of the queries among the x-axis.

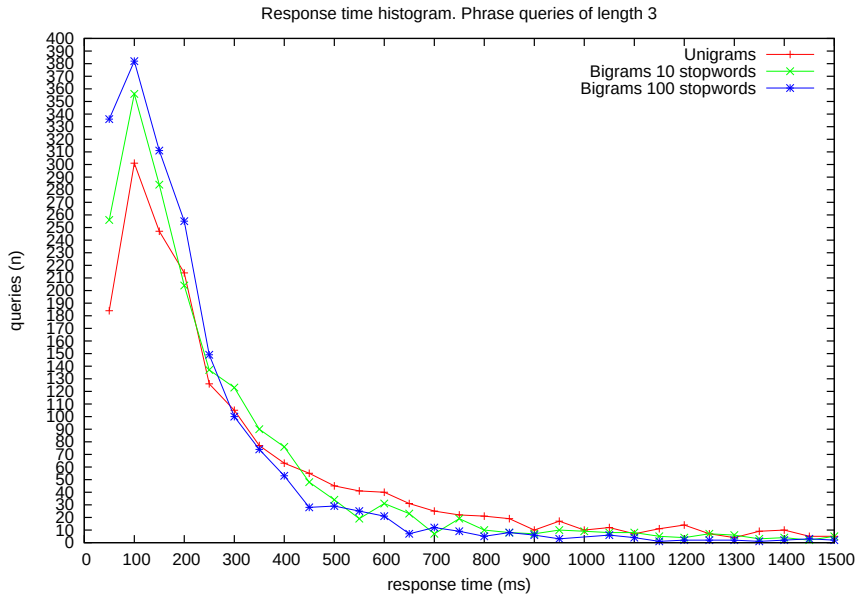

Figure 26: A graph illustrating the histogram for each index and the performance of its query response time. The x-axis represents the time, while the y-axis represents the amount of queries performing in that time range. Focus on phrase queries of 3 terms.

Notice the peaks for each curve in Figure 26, whereas each curve represents an index. This particular category included - according to Table 15 on page 108 - 1848 phrase queries, making the curves more reliable in reflecting real life performance. Also notice that the highest peak is hold by the Bigram100 index, and the second highest is hold by the Bigram10 index. The Unigram index (standard inverted index) has the lowest peak. This characteristic signalizes that the Unigram index has a more distributed curve among the x-axis, which in turn means that a large portion of its phrase queries performed within larger response time ranges. In contrast, the Bigram100 index has its major part of phrase queries within the lower part of the response time ranges, and thus gives overall better response times.

In Figure 27 we have the same graph as in Figure 26, but it is based on a different data set. The graph in Figure 27 is based on phrase queries of length 4, which we have 550 phrase queries of in our result set. In this graph we can see the curves and their peaks, along with their distribution among the x-axis. Just as with the previous graph, we can here see that the Unigram curve has a more equal distribution, in contrast to the Bigram100 index. Also, the Bigram10 index shows a better performance than the Unigram index.

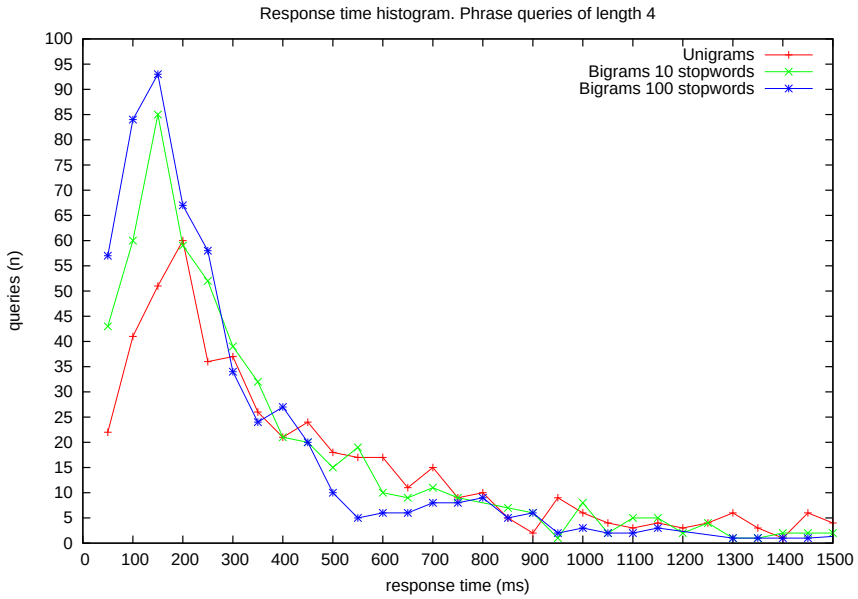

Figure 27: A graph illustrating the histogram for each index and the performance of its query response time. The x-axis represents the time, while the y-axis represents the amount of queries performing in that time range. Focus on phrase queries of 4 terms.

Notice that the amount of phrase queries in the category which is in focus in Figure 27 is much lower than the graph in Figure 26. This means that the y-axis holds a substantial lower range than in Figure 26, which in turn gives a more drastic image of the differences within the response time ranges. Nevertheless, the characteristics between the indexes comes forward in the same degree as in the previous one.

Observe that we in Figure 27 consider longer phrase queries (containing 4 query terms), and thus the process is a bit more complex, involving fetching and decoding of even more posting lists. This can be observed from the graph in that the Unigram curve has far less phrase queries within the lower reponse time ranges, which is caused by the need to fetch and decode long posting lists. In case of the curves of the Bigram10 and the Bigram100 indexes, we see that they somewhat maintain the *gap* between them and the Unigram curve.

In Figure 28 we are looking at the same type of graph as the two previous ones, despite the change of the underlying data set. In this graph we have considered phrase queries of length 5, which is a substantial smaller set of phrase queries than the others, as we here have 130 phrase queries.

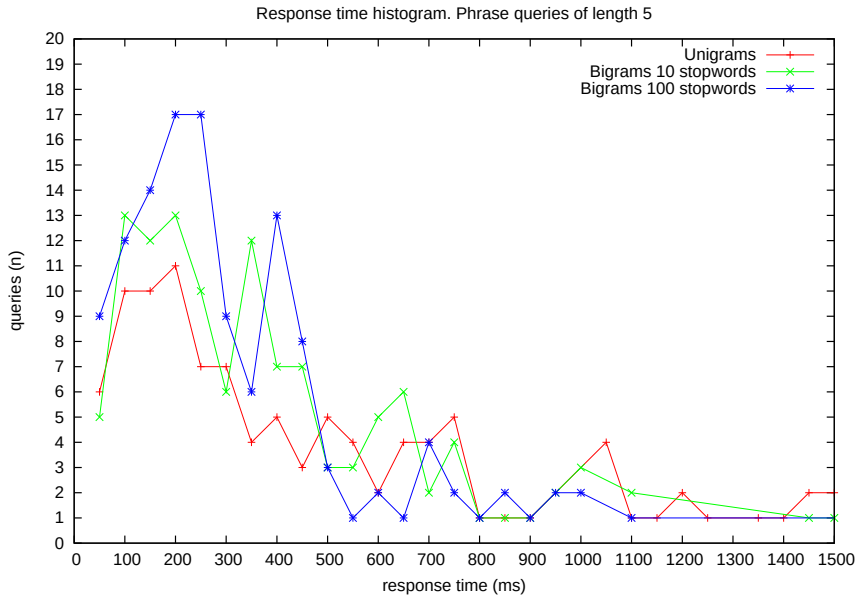

Figure 28: A graph illustrating the histogram for each index and the performance of its query response time. The x-axis represents the time, which the y-axis represents the amount of queries performing in that time range. Focus on phrase queries of 5 terms.

As with the previous graph, also Figure 27 suffers from the lack of enough phrase queries to give a more smooth presentation of the curves. Also, in Figure 28 there is a higher risk of being affected by the characteristics of the phrase queries performed, due to the lack of smoothness. Phrase queries containing non-stopwords would perform equally. If the larger portion of the 130 phrase queries shares this characteristic, then the gaps between the curves would decrease. In Table 14 and 15 there is denoted the average stopwords within each phrase query category. For Bigram10 there is on average 0.946154 stopwords in each phrase query, and for Bigram100 there is 1.546154. The difference is due to the different set of stopwords used.

7.4 Summary of findings

Based on the evaluations of the tables and the graphs presented, we can conclude that the gains introduced by the Bigram index is substantial. There are some constraints in relation to these gains introduced, such as that the input phrase queries has to contain one or more stopwords. In case of a phrase query without any stopwords, that query would most likely perform quite well due to the characteristics of the query terms. Non-stopword query terms has much shorter posting lists than stopwords, thus the I/O traffic

and processing needs would not introduce any significant time delay in the query evaluation. In case of a phrase query containing one or more stopwords, Baldr would be able to fetch posting lists for the identified bigrams in the query, thus introduce savings.

8 Discussion

In this chapter we will take the results and experience from our Bigram index experiments, and discuss them in relation to the Nextword- and Phrase indexes outlined in Chapter 5.

In Section 1 we stated that the intention of this master thesis is to evaluate different aspects of performing phrase searching, in addition to evaluate different approaches towards efficient phrase searching. In Chapter 5 we outlined different state of the art structures, which specifically has phrase searching in mind. In Chapter 6 and Chapter 7 we thoroughly explored and described our proposal for enhancing phrase searching, in addition to experiment with the proposal and evaluate the results. So, we have now a set of proposals, all with the same intention in mind. What we now need to do is to discuss whether there are benefits in using any of the proposals, and which one will be recommended.

8.1 Proposals

Based on the approaches in Chapter 5, along with the one we created and experimented with in Chapter 6 and Chapter 7, we have the following list of proposals:

- Nextword index
- Phrase index
- Bigram index
- Combined nextword- and inverted
- Combined phrase- and inverted
- Three-way combination

Each of the proposals in the list above is already thoroughly explored and introduced, so any further introduction will not be necessary.

8.2 Common characteristics

Here we will investigate and discuss the common characteristics the approaches may share or not. The motivation for this is to create a more visible distinction between them.

Since the characteristics between the nextword and the phrase index is explored in Chapter 5, we will not further outline them here. Hence, we will primarily focus on the common characteristics between the bigram index and the approaches outlined in Chapter 5.

Also note that the combined approaches would not be compared, as their applications differ with the bigram index, even though their primary goal is the same. Combined approaches enhances phrase searching indeed, but they also cope with the rest of the query types. Since the bigram index is only meant towards phrase queries, it would not match towards the combined approaches. However, a combined bigram approach would be of interest.

8.2.1 Nextword vs Bigram

Considering the nextword index and the bigram index, we may notice some common characteristics. Both is conceptually focused on the same thing: pair of words and the utilization of common words. The nextword index has a one-to-many relation for its firstwords and nextwords (a firstword can have many nextwords, hence the nextword list), while the bigram index considers word pairs as single terms in the index vocabulary.

The interesting common characteristics is that both focuses on the utilization of stopwords. The nextword index uses stopwords as firstwords in its vocabulary, and then "hooks" nextword lists up against those identified firstwords. Each complete firstword+nextword pair has its corresponding posting list. The bigram index uses stopwords to signalize the creation of bigrams. If a stopword occurs, then two bigrams is created. First the previous word plus the stopword, then the stopword plus the nextword. The resemblance is astonishing.

However, a structural difference is the implementation of the approaches. The nextword index imply a modification of the inverted index structure in terms of "hooking" up the nextword lists against the stopwords in the vocabulary. The bigram index, on the other hand, takes the same approach as the phrase index. Each bigram is indexed as a single searchable unit (term), and stopwords is neglected in the vocabulary since they are already covered by the bigram terms.

The drawbacks these index structures share is the increasing index size, as more information is stored in the index structure. This drawback is however counterbalanced by the significant increase in response time. But, considerations regarding scalability in terms of the increasing index size should be taken. If a sufficiently large document collection is to be indexed, then the index structure may be too much for a single computer to handle. There is no need to further outline the advantages, as they are made very clear.

8.2.2 Phrase vs Bigram

Considering the phrase index and the bigram index, we can see a common characteristic in that they both have the same inverted index structure. No modification such as those performed in the nextword index is done.

The phrase index considers a complete phrase query to be a single searchable unit (term), thus indexing whole phrase queries and place them in the vocabulary with its corresponding posting list in the occurrence part. The same approach is also performed in the bigram index, as it considers both single words and bigrams as terms. Bigrams, however is based on pair of words, somewhat similar to a phrase query of length 2. The main difference between the phrase index and the bigram index is the items in the vocabulary; the terms. The phrase index has exclusively complete phrases as terms in the vocabulary, while the bigram index has both bigrams and single words as terms. Notice that the bigram index vocabulary is magnitude larger than the phrase index vocabulary, as the bigram index also holds single words and bigrams, but not single stopwords. And that a phrase index only holds a predefined set of phrase queries in the vocabulary, not all possible phrase queries, as that would be prohibitive.

Nevertheless, both index types share this characteristic, and it is clear that both approaches has its advantages and disadvantages. The drawback with this approach is that its restricts the application for the index in that only phrase queries may be evaluated, and that a combined approach or a fallback index would be needed in case of other query types. The bigram index does however support other query types into some extent, but since the stopwords are neglected there would be difficult to query for stopwords. Queries containing only stopwords is thus very rare. The advantages with these structures is, beyond what is clear, its simplicity in that they do not modify the inverted index structure. This is an advantage, since it carry less complexity.

The phrase index described in Section 5.2 is meant to be an isolated index, in contrast to the bigram index which includes an stopped inverted index (no stopwords is indexed, except for in bigrams).

8.3 Performance

The bigram index results is thoroughly presented and evaluated in Chapter 7. The performance of the nextword and phrase index is presented in Chapter 5, along with the results in Table 8 illustrating their performance, in regards to the experiment performed in [12].

Comparing the results between our experiment and the experiment carried

out in [12] or [5] would be somewhat difficult. This is since they were carried out on different document collections, as well as different query sets. However, the characteristics they represents can into some extent be compared. We can easily see based on Table 9 for the "three-way combined approach" in Section 5.3.3, that it shares some characteristics with the tables Table 13, Table 14 and Table 15. However, these results is not very comparable.

What is possible to compare, in term of performance, is the overall stated performance enhancement for each index. Now, based on the evaluations and results from Chapter 7, we can see that the bigram index enhances query response times with 20%-80%, compared to the standard inverted index. In [12] there is stated that a complete nextword index is 50 times faster than using a standard inverted index. This can be shown by looking in Table 8, and comparing the average query time between an inverted index and the nextword index (1.04 sec vs 0.02 sec). In [5] a nextword index in combination with a inverted index is shown to cut the query evaluation of phrases to $\approx$ 60%. When using the "three-way combined approach" described in Section 5.3.3, the performance enhancement is between 60%-80% faster than using an standard inverted index.

In terms of disk space consumption, then the bigram index requires $\approx$ 18% of the document collection size when using 100 stopwords. Note that this requirement will increase when using more than 100 stopwords. Neither the nextword nor the phrase index can directly be compared to the bigram index in terms of index size, since neither one of them can be practical used in our experiment. Also, a complete nextword index has a severe disk requirement. According to [13], a complete nextword index requires 60% of the document collection. However, the "three-way" combined approach can be compared here. In [12] it is concluded that this approach requires 20% of the size of the document collection. As can be seen, the size requirements is very close, whereas the bigram index requires slightly less.

Considering operations performed on the structures, such as `add`, `update` and `delete`, then the nextword index will stand out in a negative perspective. This is due to the increased complexity with the nextword lists. The bigram index has an quite large vocabulary in contrast to the inverted index, but the postings lists for the index terms is drastic shorter (considering bigram terms). Thus, operations which includes `add`, `update` or `delete` on bigrams would claim less processing than in case of indexed stopword terms. The phrase index has a short vocabulary, since it only contains carefully selected phrases. In addition, the posting lists would be shorter than the most frequent term in the phrase query, thus imply far less processing.

9 Conclusion

This chapter covers the final conclusion based on the results and literature brought forward in this master thesis. Also the discussion regarding the bigram index and the other outlined approaches is taken into consideration.

This master thesis has covered some state of the art material in phrase search technology, as well as introduced a new phrase searching structure named "bigram index". We also performed a thoroughly experiment with our new phrase searching structure, and evaluated the presented results from both indexing and searching.

9.1 Current challenges

The challenges within phrase searching is from the very beginning highlighted in this master thesis. We described the inverted index and its evaluation process of phrase queries, and described in detail why such an approach does not scale well when considering vastly large document collections. Especially factors such as I/O, decoding and merging are taken into consideration when outlining the problems with phrase query evaluation.

The primary challenge which affects phrase searching performance in the standard inverted index is its vastly large posting lists. A particular set of index terms stands out in terms of large posting lists; high frequent terms, also known as "stopwords". The cost to fetch and decode such posting lists is tremendous, and highly influence the reponse time for queries which contains stopwords.

9.2 Phrase searching improvements

Throughout this master thesis we have presented possible improvements to cope with the current challenges within phrase searching. Innovative approaches introduced in [13] [4] [5] [12] and presented in Chapter 5, takes phrase searching challenges into consideration and approaches them. An even further innovative and interesting approach is the bigram index, which is introduced, experimented and evaluated in this master thesis.

In Chapter 8 we discussed the presented and introduced approaches towards phrase searching, in addition to combined approaches which is capable of working as a complete search engine. An especial focus was directed towards our bigram index vs the other presented indexes. Common to them all is that they cope with the challenge of vastly large posting lists, and thereby achieve significant enhanced phrase query evaluation. Thus, the applica-

tions for each index is somewhat different; as we found the bigram index as something between both nextword index, and phrase index, based on their characteristics.

The performance characteristics outlined in a summarized manner in Chapter 8, show us that the bigram index holds as similarly size requirement as the "three-way combined" approach. In direct comparison between the bigram and nextword index, we find that the nextword index requires much more in disk space due to more complexity and structure. A complete nextword index needs 60% of the document collection, while the bigram index (100 stopwords) only $\approx$ 18%.

9.3 Final conclusion

Based on the results presented in this master thesis, we can with confidence conclude that the bigram index provides significant improvements in terms of phrase query evaluation. These improvements yield great importance for search engines, given the desire to provide fast phrase query evaluation.

Summarized: when phrase searching, our bigram index provides a fast phrase query evaluation, in contrast to the ordinary inverted index. Also, the nextword and phrase indexes should be taken into consideration, as they both provide rapid phrase query evaluation.

9.4 Further work

In further work we will look at aspects in our bigram index which could be subject of improvements, and consider experimenting with other approaches in the hunt for faster phrase searching capabilities. We will outline the aspects here.

In our experiments with Baldr in regards to indexing and searching, we explicitly used 10 and 100 stopwords. An interesting thought could be to identify the ideal number of stopwords to use. Experimenting with more stopwords would give an even larger vocabulary, but the expensive posting lists would neglect since the "costly" index terms would be coped by the bigrams. This area would be quite obvious to explore in further work.

In this master thesis we outlined the nextword and phrase indexes, and discussed them in relation to our bigram index. It is very tempting to implement the nextword index, as it modifies the inverted index structure and introduces the nextword lists, which is a very innovative approach. This

would also make it a whole lot easier to compare it with our bigram index. Also the phrase index approach, which is easy to implement in the current bigram index, would have been interesting to experiment with.

In Section 6.6.4 we have identified some possible future improvements in our implementation. Taking these into consideration would naturally be part of a further work, as they are likely to yield further gains.

References

[1] Kjell Bratbergsengen. *Lagring og behandling av store datamengder.* Tapir akademisk forlag, 2003.

[2] Ophir Frieder David A. Grossman. *Information Retrieval, Algorithms and Heuristics, Second Edition.* Springer, 2004.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Google Labs*, 2004.

[4] Justin Zobel Dirk Bahle, Hugh E. Williams. Optimised phrase querying and browsing of large text databases. *http://www.csse.unimelb.edu.au/ jz/fulltext/acsc01.pdf*, 2001.

[5] Justin Zobel Dirk Bahle, Hugh E. Williams. Efficient phrase querying with an auxiliary index. *http://www.cs.rmit.edu.au/ jz/fulltext/sigir02bwz.pdf*, 2002.

[6] Otis Gospodnetiç Doug Cutting and Erik Hatcher. *Lucene In Action - A guide to the Java search engine.* Manning, 2005.

[7] Nivio Ziviani Edleno Silva de Moura, Gonzalo Navarro and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *ACM*, 2000.

[8] Peter Elias. *Universal codeword sets and representations of the integers.* IEEE Transactions on Information Theory, 1975.

[9] John Yiannis Falk Scholer, Hugh E. Williams and Justin Zobel. Compression of inverted indexces for fast query evaluation. *ACM*, 2002.

[10] S. W. Golomb. Run-length encodings. pages 399–401, 1966.

[11] H. S. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, 1978.

[12] Dirk Bahle Hugh E. Williams, Justin Zobel. Fast phrase querying with combined indexes. *http://portal.acm.org/citation.cfm?doid=1028099.1028102*, 2004.

[13] Phil Anderson Hugh E. Williams, Justin Zobel. What's next? index structures for efficient phrase querying. *http://www.csse.unimelb.edu.au/ jz/fulltext/adc99.pdf*, 1999.

[14] Timothy C. Bell Ian H. Witten, Alistair Moffat. *Managing Gigabytes: compressing and indexing documents and images, 2nd.* Morgan Kaufmann, 1999.

[15] Wikimedia Foundation Inc. Wikipedia. `http://www.wikipedia.org`.

[16] Xiaohui Long Jiangong Zhang and Torsten Suel. Performance of compressed inverted list caching in search engines. *ACM*, 2008.

[17] Alistair Moffat Justin Zobel. Inverted files for text search engines. *http://portal.acm.org/citation.cfm?id=1132959*, 2006.

[18] Rajeev Motwani Lawrence Page, Sergey Brin and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[19] Christian Middleton and Ricardo Baeza-Yates. A comparison of open source search engines. *http://wrg.upf.edu/WRG/dctos/Middleton-Baeza.pdf*, 2007.

[20] Alistair Moffat and Justin Zobel. *Self-indexing inverted files for fast text retrieval.* ACM, 1996.

[21] Gonzalo Navarro Nivio Ziviani, Edleno Silva de Moura and Ricardo Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEE*, 2000.

[22] Berthier Ribeiro-Neto Ricardo Baeza-Yates. *Modern Information Retrieval.* Addison Wesley, 1999.

[23] Knut Magne Risvik and Tor Egge. Scaling internet search engines: Methods and analysis. *http://www.pvv.ntnu.no/ kmr/risvik04phd.pdf.*

[24] Veclav Strnad Roman Tesar, Massimo Poesio and Karel Jazek. Extending the single words-based document model: A comparison of bigrams and 2-itemsets. *http://portal.acm.org/citation.cfm?id=1166160.1166197.*

[25] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw Hill Book Co., 1983.

[26] I.H. Witten T.C. Bell, Alistair Moffat and Justin Zobel. The mg retrieval system: Compressing for space and speed. *ACM*, 1995.

[27] Sebastiano Vigna. Managing gigabytes for java. `http://mg4j.dsi.unimi.it`.

[28] G. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.

A Inverted Index

Here we present our stopword terms extracted from the standard inverted index.

A.1 Ten most frequent

```
the
of
and
to
a
in
for
that
is
s
```

A.2 Hundred most frequent

Notice that the term "gov" occurs twice in the list below. This is due to that it frequently occurred in two Fields for our GOV2Document object. Logically, "gov" occurred once in the Field for the documents URL, since the TREC GOV2 is a crawl of the top-domain `.gov`.

```
the
of
and
to
a
in
for
that
is
s
on
or
be
by
with
are
this
as
at
from
```

not
an
it
will
information
have
i
may
data
u
other
c
e
state
all
has
new
if
no
gov
d
health
can
national
more
about
b
page
department
also
any
one
gov
http
home
its
office
federal
been
site
but
contact
center
do

```
available
act
each
date
www
development
agency
first
search
how
government
general
area
california
during
last
based
html
after
because
business
city
between
board
control
help
application
case
following
day
administration
current
access
community
american
privacy
back
```

B Code snippets

Outline various code parts from all aspects of the experiment. A mixture of both Python[36] and Java, whereas Java is used in the core parts, and Python

[36] http://www.python.org

is used to analyze the measurements extracted from the tests.

B.1 Calculate average query terms and terms length

In Listing 5 we calculate the average number of query terms used in all phrase queries used in our experiment. Notice the `tmp.txt` file, which contains for each line the query used.

```
1 import numpy
2 f = open('tmp.txt')
3 data = f.readlines()
4 t = [x.count(' ')+1 for x in data]
5 t.sort()
6 l = numpy.array(t)
7 print "Average query terms for each query: %f" % numpy.average(l
    )
```

Listing 5: "Calculate average query terms in result set from experiment"

In Listing 6 we calculate the average term length in all the query terms used in the query set.

```
1 import numpy
2 f = open('tmp2.txt')
3 data = f.readlines()
4 t = [len(x) for x in data]
5 l = numpy.array(t)
6 print "Average query term length: %f " % numpy.average(l)
```

Listing 6: "Calculate the average query term length for all the query terms used in the query set"

Common to both code snippets above is that they take advantage of scientific computing library named *NumPy*[37].

B.2 Test OS caching

Here is a very simple code that takes a filename as an argument, opens the file, read the contents, and then closes it. The code is outlined in Listing 7.

```
1 import sys
2
3 if __name__ == "__main__":
4     if not len(sys.argv) > 1:
5         print "Need input argument!"
6         sys.exit(0)
7     name = sys.argv[1]
8     try:
9         f = open(name, "r")
```

[37] `http://numpy.scipy.org/`

```
10          data = f.readlines()
11          f.close()
12      except Exception, e:
13          print "Will exit. Encountered error: %s" % error
14      finally:
15          sys.exit(0)
```

Listing 7: "Read a file"

C Code

Java and Python code from frequently used components in this experiment. The Java part is focused on the Baldr search engine and some utility modules, and the Python part is exclusively used to extract statistical information used to interpret the experiment.

C.1 Statistical fetcher

Python code for getStatistics.py

```
1  #!/usr/bin/env python
2
3  import sys
4  import os
5  import os.path
6
7  try:
8      from optparse import OptionParser
9  except ImportError, ie:
10     print "Missing 'OptionParser' module.\nWill exit."
11     sys.exit(0)
12
13 def readInResultFile(filename = None):
14     if not filename or not os.path.isfile(filename):
15         return dict()
16
17     query_results = dict()
18     f = open(filename,'r')
19     data = f.readlines()
20     f.close()
21
22     for line in data:
23         org_query, rewritten_query, hits, search_time = line.
               split(";")
24         hits = int(hits)
25         search_time = int(search_time)
26
27         nr_query_terms = len(org_query.split(" "))
28
29         query_results[org_query] = dict(q_len=nr_query_terms,
30                 rewritten_query=rewritten_query,
```

```
31                  hits = hits,
32                  search_time = search_time
33                  )
34
35      return query_results
36
37  def getStopwords():
38      f = open("stop_words.txt", "r")
39      data = f.readlines()
40      stopwords = []
41      for i in data:
42          if i.startswith("#"):
43              continue
44          stopwords.append(i.replace("\n",""))
45      f.close()
46      return stopwords
47
48  def fetchPlotsBasedOnQuery(results, length, withStopwords=False)
        :
49      """
50          Same as 'def fetchPlots()', but will use every single
                value!
51      """
52      if not isinstance(results,dict):
53          return False
54      if not isinstance(length, int):
55          raise ValueError("Invalid input value for parameter '
                length'. Got %s" % str(length))
56
57      stopwords = getStopwords()
58
59      # creates a fresh dict of search times
60      newDict = {}
61      for q in results.keys():
62          q_len = int(results[q].get('q_len'))
63          if q_len is not length:
64              continue
65
66          query_words = q.split(" ")
67          gotStopword = False
68          for w in query_words:
69              if w in stopwords:
70                  gotStopword = True
71              break
72          if withStopwords:
73              if not gotStopword:
74                  continue
75          else:
76              if gotStopword:
77                  continue
78
79          search_time = results[q].get('search_time')
80
81          # we want to construct a vector to hold
```

```
82          # the search_time along with the # of times it occurs
83          if newDict.get(search_time):
84              newDict[search_time] += 1
85          else:
86              newDict[search_time] = 1
87      sorted_keys = newDict.keys()
88      sorted_keys.sort()
89      for i in sorted_keys:
90          print "%i\t%i" % (i, newDict[i])
91
92  def fetchPlots(results, length):
93      """
94          Filter out only queries of size == length.
95          And, collect the results in buckets, where the values
                differ only by 50.
96          This is to 'smooth' the frequency in the graph.
97
98          note: we know that the response-time is not more than 15
                000ms (15sek),
99          so we've therefore used 'xrange(0,15000,50)' to create
                '0,50,100,150,..,15000'.
100     """
101     if not isinstance(results,dict):
102         return False
103     if not isinstance(length, int):
104         raise ValueError("Invalid input value for parameter '
                length'. Got %s" % str(length))
105
106     # creates a fresh dict of search times
107     newDict = {}
108     # creates the buckets to hold the values!
109     buckets = {}
110     for i in xrange(0,15000,50):
111         buckets[i] = {'value':0, 'count':0}
112
113     for q in results.keys():
114         q_len = int(results[q].get('q_len'))
115         if q_len is not length:
116             continue
117
118         search_time = results[q].get('search_time')
119
120         if newDict.get(search_time):
121             newDict[search_time] += 1
122         else:
123             newDict[search_time] = 1
124     sorted_buckets_keys = buckets.keys()
125     sorted_buckets_keys.sort()
126     for i in newDict.keys():
127         tmp = 0
128         for x in sorted_buckets_keys:
129             if i < x:
130                 buckets[x]['value'] += newDict[i]
131                 buckets[x]['count'] += 1
```

```
132                 tmp = x
133                 break
134     # normalize the numbers in each bucket. (divide by 50, which
            is the bucket size)
135     for i in sorted_buckets_keys:
136         if not buckets[i]['value']:
137             continue
138         print "%i\t%i" % (i, buckets[i]['value'])
139
140
141 def queriesWithStopwordsOfVariableLength(results, length,
        reverted=False):
142     # ugly function name!
143     if not isinstance(results, dict):
144         return False
145     if not isinstance(length, int):
146         raise ValueError("Invalid input value for parameter '
                length'. Got %s" % str(length))
147
148     phrase_types = dict()
149     stopwords = getStopwords()
150
151     interesting_queries = []
152
153     for q in results.keys():
154         q_len = int(results[q].get('q_len'))
155
156         # want only queries of length 2
157         if q_len is not length:
158             continue
159
160         # if stopword in query, append to interesting_queries[]
161         words_in_query = q.split(" ")
162
163         if not reverted:
164             for term in words_in_query:
165                 if term in stopwords:
166                     interesting_queries.append( q )
167                     break
168         else:
169             stopword_query = False
170             for term in words_in_query:
171                 if term in stopwords:
172                     stopword_query = True
173                     break
174
175             if not stopword_query:
176                 interesting_queries.append( q )
177     return interesting_queries
178
179 def queriesWithoutStopwordsOfVariableLength(results, length):
180     """
181         This is just a wrapper for 'def
                queriesWithStopwordsOfVariableLength()', by
```

```
182         just flipping the last argument in function call.
183     """
184     return queriesWithStopwordsOfVariableLength(results, length,
            True)
185
186
187 def countDifferntPhraseTypes(results):
188     if not isinstance(results, dict):
189         return False
190
191     phrase_types = dict()
192     phrase_avg_times = dict()
193
194     stopwords = getStopwords()
195     print "========== Got %i stopwords in memory! ==========\
            n" % len(stopwords)
196
197     for q in results.keys():
198         q_len = results[q].get('q_len')
199         if not q_len in phrase_types.keys():
200             phrase_types[q_len] = 1
201         else:
202             phrase_types[q_len] += 1
203
204         search_time = results[q].get('search_time')
205
206         stopwords_in_q = len([i for i in q.split(" ") if i in
                stopwords])
207
208         if not q_len in phrase_avg_times.keys():
209             phrase_avg_times[q_len] = [search_time, 1,
                    stopwords_in_q]
210         else:
211             stopwords_in_q = phrase_avg_times[q_len][2]+
                    stopwords_in_q
212             phrase_avg_times[q_len] = [phrase_avg_times[q_len
                    ][0]+search_time, phrase_avg_times[q_len][1]+1,
                    stopwords_in_q]
213
214         if not q_len in phrase_avg_times.keys():
215             phrase_avg_times[q_len] = [search_time, 1,
                    stopwords_in_q]
216         else:
217             stopwords_in_q = phrase_avg_times[q_len][2]+
                    stopwords_in_q
218             phrase_avg_times[q_len] = [phrase_avg_times[q_len
                    ][0]+search_time, phrase_avg_times[q_len][1]+1,
                    stopwords_in_q]
219
220
221     print "Summary of phrase lengths and occurences in query
            result file.."
222     total_phrases = 0
223     for i in phrase_types.keys():
```

```
224         tmp_avg_time = phrase_avg_times[i][0]
225         count = float(phrase_avg_times[i][1])
226
227         avg_time = float(tmp_avg_time / count)
228         avg_stopwords_in_q = float(phrase_avg_times[i][2] /
                count)
229         total_phrases += phrase_types[i]
230
231         print "phrase length: %i  Occurences: %s
                Average_search_time: %s
                Average_nr_of_stopwords_in_query: %f" % \
232             (i, str(phrase_types[i]).rjust(4), str(avg_time).
                    rjust(14), avg_stopwords_in_q)
233     print "Total nr of phrase queries with hits: %i" %
            total_phrases
234
235     return (phrase_types, phrase_avg_times)
236
237
238 if __name__ == "__main__":
239     usage = "usage: %prog [options]"
240
241     parser = OptionParser(usage=usage)
242     parser.add_option("-s", "--statistics", action="store_true",
             dest="statistics", default=False, help="Will output the
            statistics of the input result
243     parser.add_option("-o", "--fetchPlotBasedOnQuery", action="
            store_true", dest="plots_based_on_query", default=False,
            help="Will print out plot values
244             to be used in Gnuplot graphs. NOTE that you may
                    specify with -l or -r to have the plots based on
                    queries with or without stopwords. Default
245     parser.add_option("-p", "--fetchPlots", action="store_true",
             dest="plots", default=False, help="Will print out plot
            values to be used in Gnuplot gra
246     parser.add_option("-l", "--listOfstopwords", action="
            store_true", dest="listofstopwords", default=False, \
247             help="Will output a list of phrase queries which is
                    of variable length (default 2) and contains
                    stopwords in it.")
248     parser.add_option("-i", "--length", action="store", dest="
            phrase_length", default=2, help="Length of queries which
            is to be searched for. Only usefu
249             when used together with -l.")
250     parser.add_option("-r", "--listOfNoneStopwords", action="
            store_true", dest="listofnonestopwords", default=False, \
251             help="Will output a list of phrase queries which is
                    of variable lenght (default 2) and does NOT
                    contain stopwords in it.")
252     parser.add_option("-f", "--filename", action="store", type="
            string", dest="filename", help="This is required, since
            this is the result file.")
253     (options, args) = parser.parse_args()
254
```

```
255     if len(args) < 2:
256         if not options.filename:
257             parser.error("Need the result file to be able to do
                    stuff..")
258             sys.exit(0)
259
260         parsedResults = readInResultFile(options.filename)
261         if options.statistics:
262             countDifferntPhraseTypes(parsedResults)
263
264         if options.plots:
265             if not options.phrase_length or not int(options.
                    phrase_length):
266                 length = 2
267             else:
268                 length = int(options.phrase_length)
269             fetchPlots(parsedResults, length)
270
271         if options.plots_based_on_query:
272             if not options.phrase_length or not int(options.
                    phrase_length):
273                 length = 2
274             else:
275                 length = int(options.phrase_length)
276
277             if options.listofstopwords:
278                 fetchPlotsBasedOnQuery(parsedResults, length,
                        True)
279             else:
280                 fetchPlotsBasedOnQuery(parsedResults, length,
                        False)
281             # will exit here, so the section below won't get
                    executed due to
282             # 'options.listofstopwords' and 'options.
                    listofnonestopwords'.
283             sys.exit(0)
284
285         if options.listofstopwords or options.
                listofnonestopwords:
286             if not options.phrase_length or not int(options.
                    phrase_length):
287                 length = 2
288             else:
289                 length = int(options.phrase_length)
290
291             if options.listofstopwords:
292                 queries = queriesWithStopwordsOfVariableLength(
                        parsedResults, length)
293                 print "Queries of length %i with stopwords in it
                        : %i" % (length, len(queries))
294             else:
295                 queries =
                        queriesWithoutStopwordsOfVariableLength(
                        parsedResults, length)
```

```
296                 print "Queries of length %i without stopwords in
                        it: %i" % (length, len(queries))
297             x = 1
298             for i in queries:
299                 print ':'.join([str(x),i])
300                 x += 1
```

Listing 8: "Parse measurements from query tests and perform necessary calculations. Outputs statistics in a comprehensive manner. Also provides output of plot data to be used in GNUPlot."