NTNU
Norwegian University of
Science and Technology

Instant Search Using Query Expansion with Pseudo-Relevance

Jonatan Lund

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Herindrasana Ramampiaro, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Todays web services generate large amounts of data. Users expect these services to have a search which returns relevant search results instantaneously. Term frequency-inverse document frequency (TF-IDF) is a common technique used in search engines to deliver relevant search results fast. However, with the increasing amounts of data users expect the search results to deliver even more relevant search results. Improving the search results can be done by expanding the user's query. Providing relevant information for query expansion may be a challenge, but using a technique called pseudo-relevance feedback has shown promissing results. Relevant search results are important, but equally important is speed. Research by Google [23] found that 0.5 seconds increased load times resulted signifiant less traffic.

This thesis investigates how query expansion can be implemented together with pseudo-relevance, to deliver more relevant search results. Research on how to improve search results is not new, but the focus is rarely on speed. Thus our study focuses on speed, with a requirement to deliver search results within 100 ms, which is the maximum acceptable amount of time before users will notice the delay.

The final implementation showed promising results, and were able to deliver search results within within the requirement of 100 ms.

II

Sammendrag

Dagens webtjenester genererer enorme datamengder. Brukerene av disse tjenestene forventer at søkene leverer relevante søkeresultater umiddelbart. En mye brukt teknikk i søkemotorer kalles term frequency-inverse document frequency(TF-IDF), som er i stand til å levere relevante søk fort. I dag inneholder enkelte tjenester så mye informasjon at det trengs mer avanserte metoder for å levere relevante søkeresultater. En måte å forbedre søke på er ved å utvide søket til brukeren, men en utfordring er å finne relevant informasjon som kan brukes i en søkeutvidelse. Ved å kombinere søkeutvidelse med pseudo-relevant tilbakemelding har man funnet ut at søket vil bli mer relevant. I tillegg til relevante søk er det viktig at søket går fort. Forskning gjort av Google [23] viste at et halvt sekund lenger ventetid på et søkeresultat førte til markant mindre trafikk.

Denne oppgaven undersøker hvordan søkeutvidelse kan bli implementert sammen med pseudo-relevanse for å levere mer relevante resultater. Dette i seg selv er ikke ny forskning, men mye av forskningen på feltet fokuserer ikke på hastighet. Implementasjonen beskrevet i denne oppgaven vil ha et hovedfokus på hastighet, og vil ha et krav om at søkeresultatene skal være tilgjengelig hos brukere innen 100 millisekunder. 100 millisekunder forsinkelse er det øverste taket før en bruker vil oppfatte forsinkelsen.

Implementasjonen av søkeutvidelsen viser lovende resultater, og er i stand til å levere resultater godt innenfor kravet på 100 millisekunder.

Contents

List of Figures

List of Tables

1 Introduction

The first section in this chapter describes the motivation behind the master thesis. Next, the problem specification is derrived from the motivation. Lastly, the master thesis layout is described.

1.1 Motivation

The amount of data generated by web applications are continuously increasing. Making all the data avaiable for users, requires specialized search engines. Most of today's search engines implement a technique called term frequency-inverse document frequency when searching for documents. This technique is effective, and returns relevant search results most of the time, but with the ever growing data size web applications strive to deliver even more relevant search results. Personalization is a common approach to deliver more relevant search results. If a user in Trondheim search for the word `resturant`, resturants in Trondheim are most likely more relevant compared to resturants in Oslo. Today an increasing number of software companies try to personalize their services. A few examples of personalized services are Facebook's News Feed [13], Netfix's movie suggestion [19] and Spotify's Discover Weekly [27].

A second important factor for interactive tasks such as search, are speed. Interactive tasks have a requirement of 100 ms before the user recognizes the delay [4]. This means that search results have to be available within 100 ms from the user typed the query. High latency may lead to users abandoning your site, and revenue is easily lost. Returning results fast is more important compared to the number of results returned. While conducting latency experiments, Google found that users said that they wanted 30 search results instead of 10 [23]. However, their tests also showed that users who recieved 30 search results gave less traffic compared to users who recieved 10 search results. The difference between the search results were load times. 10 search results took 0.4 seconds to load and 30 search results took 0.9 seconds to load. It shows that speed is an important factor in search.

Personalized search engines already exists. Google[1] and Bing[2] were the two largest search engines in 2016 according to NetMarketShare [25]. However, neither Google nor Bing are open source.

[1] `https://www.google.com`
[2] `https://www.bing.com/`

A common approach to improve the search result is to extend the user's query with more terms, or adding additional information like the user's position. Research has found it diffiucult to select good terms to expand [18]. On the other hand, tags have often been found to provide good expansion terms [5]. In this master thesis query expansion will be implemented together with a technique called pseudo-relevance. An important requirement for the implementation will be to deliver search results within 100 ms and that the implementation is scalable.

This master thesis builds on and extends the work done by Rudihagen [26] and my project report [21]. Rudihagen researched how an instant personalized search could be achieved. The research achieved a personalized search which returned results of higher relevance to the user compared to TF-IDF. However, the results from the work showed that the implemented method did not meet the requirement of interactivity [26]. An important factor for the latency was the number of round trips between the web server and the search engine. In my project report [21], an implementation is described which reduced the number of round trips between the web server and the search engine from four to two. However, the implementation may be improved even further by implementing query expansion with pseudo-relevance directly on the search engine.

This master thesis will explore how the implementation described by Rudihagen and in my project report can be improved even further. The next section describes three research questions based on the motivation introduced in this section.

1.2 Problem Specification

This project report tries to answer the following research questions, with the main focus being research question 2 and 3.

1. How to make an improved search in terms of relevance compared to TF-IDF?
2. How to develop an improved search which scales with an increasing amount of data?
3. How to delevop an improved search that fulfills the interactive latency requirements?

1.3 Structure

This master thesis consists of 5 chapters: Background, State of the Art Survey, Implementing Query Expansion in a Search Engine, Evaluation and a Conclusion. The Background chapter starts with an overview of the underlying technologies used in the implementation. Furthermore, this chapter contains sections which describe basic search engine techniques, relevance feedback and query expansion. The chapter State of the Art Survey describes related reasearch on how to improve result relevance. The theoretical basis for this master thesis is also included in this chapter. The chapter Implementing Query Expansion in a Search Engine describes the implementation in Lucene and the implemented Elasticsearch implementation. Important configurations are also described. Next, the chapter Evaluation outlines

how the experiments were conducted and the results from the experiments. The last section discusses the results and makes a comparison with the work by Rudihagen and my the project report. Lastly, the Conclusion chapter gives a brief summary of the results and the implementation. The last section suggests further work that, if implemented, will improve the current implementation.

2 Background

This chapter presents basic theory and techniques behind search engines. First the software used in the implementation are explained, and is then followed by a detailed description on how the documents are scored inside the search engine from a query. Finally, relevance feedback is described, as well as how it may be used with query expansion.

2.1 Underlying Technologies Used in the Implementation

This section outlines the different technologies used in the implementations. There were two separate implementations; one using the Lucene and one using Elasticsearch. Node.js was used as a part in the experiment with Elasticsearch.

Node.js[1] version 7 was chosen as the web server, because the author has knowledge of the technology, and it contains a rich package manager called NPM. By utilizing open source libraries through NPM, more time could be spent implementing the algorithms for query expansion. Node.JS utilizes the V8[2] JavaScript engine.

2.1.1 Lucene

Lucene[3] is an open-source full-text search engine library written in Java. According to Lucene [17] the index size is only about 20-30% of the original text. Lucene's search have features such as ranked search, field search and faceting, to mention a few.

Lucene exposes the low level API's which give very much control over the inner workings of Lucene. Table 2.1 shows an example of a low level data structure inside Lucene called an inverted index. The table contains a list of all the possible terms inside a text. The second column is the frequency each term has. Lastly, the documents column list all the documents where the given term occurs. An inverted index requires more resources when indexing, but the data structure is

[1] https://nodejs.org
[2] https://developers.google.com/v8/
[3] https://lucene.apache.org/

effective when searching. The query expansion which is used in this master thesis requires information stored in the inverted index.

To illustrate how the inverted index works, an example query may be the term `blue`. Using table 2.1 the inverted index shows that the term has a frequency of three, and that the term occurs in document 1, 2 and 3. The search result would then contain documents 1, 2 and 3. This is a simplified example and a more detailed description on how search engines work are found in a later section.

One important drawback with Lucene is that it is not scalable across multiple machines. However, scalable across multiple machines are one of the advantages of using Elasticsearch.

Term	Frequency	Documents
blue	3	Document 1, document 2, document 3
sky	1	Document 2
clouds	4	Document 2, document 3, document 4, document 5
rain	2	Document 2, document 5
plane	2	Document 1, document 4
sunset	4	Document 1, document 2, document 3, document 4
Sum	16	

Table 2.1: Example of an inverted index inside Lucene

2.1.2 Elasticsearch

Elasticsearch[4] v5 was used as the search engine in the implementation described in this master thesis. The search engine has proven the ability to scale up to petabytes of data [1]. Elasticsearch is open source and built on top of Lucene. Lucene is the search engine itself, and Elasticsearch provides functionality for distribution and a REST API interface.

The following topics describe basic termonology used in Elasticsearch. This information is needed to understand some of the results and observations, and the implementation described in chapter 4 and chapter 5.

Cluster

One or more servers connected together is called a cluster. Elasticsearch indices are divided into shards which are distributed across the servers in the cluster. Queries are also distributed to all the servers in the cluster, which again increases the performance. Elasticsearch is responsible for distributing the shards across the physical servers, and makes sure that replica shards are not on the same physical server. The cluster used in this master thesis consisted of one server.

Node Types

Elasticsearch has four different node types: master node, data node, ingest node and tribe node. The master node is responsible of handling administrative cluster

[4] `https://www.elastic.co/products/elasticsearch`

tasks such as creating or deleting an index, tracking online and offline nodes and deciding how the shards should be distributed. Data nodes are responsible for holding the shards and executing queries. Ingest nodes are used as pre-processing nodes. A tribe node is handling the coordination of querying and indexing. Tribe nodes is also known as coordination nodes. All nodes in a cluster are also an coordination nodes.

Sharding

The stored data in Elasticsearch may grow to become larger than the hardware on a single machine can handle, both in size and in numbers of requests. To mitigate the problem Elasticsearch splits each index into multiple segments called shards. Each of these shards may be distributed across multiple nodes. This provides higher performance when indexing new documents and when searching the documents. The shards may also be duplicated to support higher query volumes and availability. Shard duplicates are also known as replica shards.

There is important distinction between an Elasticsearch index and a Lucene index. A Lucene index is an index which contains the inverted indices and holds all the data. An Elasticsearch, index on the other hand, consists of one or more shards. One Elasticsearch shard is the same as a Lucene index.

When the cluster consists of multiple nodes, a query strategy is required. Figure 2.1 illustrates how an index may be distributed across three nodes. The figure shows a simplified view on how a query is distributed across three nodes. First the query arrives a coordination node. The coordination node parses the query and determines which nodes hold shards for the given index. Each shard determines locally which documents are most relevant from the query. Metada from all the shards are then sent back to the coordination node. On the coordination node all retrieved metadata are used to calculate the global result. After the global result has been calculated, all the shards are queried for the documents from the global result. After the documents are retrieved, the result is returned to the client. As the size of the search result is increased, the response time will also increase. This is a result of more calculations on each shard, more data transfered, and more work to be done when merging the results.

Approximate Values

As a result of Elasticsearch's distributed nature, some queries will return estimated values. Aggregations in Elasticsearch is an example which may return estimated values.

Table 2.2 has a list of words across three different shards, with the term frequencies inside the parenthesis. When a client queries for the top three terms, the query is distributed to the shards. Each shard then returns their local top three terms. Given Table 2.2, the shards will return the terms listed in table 2.3. On the coordination node the global top three terms are calculated. The top three terms are listed in table 2.4, which are `blue`, `sky` and `clouds`. However, `blue`, `sky` and `clouds` are not the actual top three terms. If the coordination node had all the knowledge in table 2.2, the top three terms would be `blue`, `sky` and `insta`,

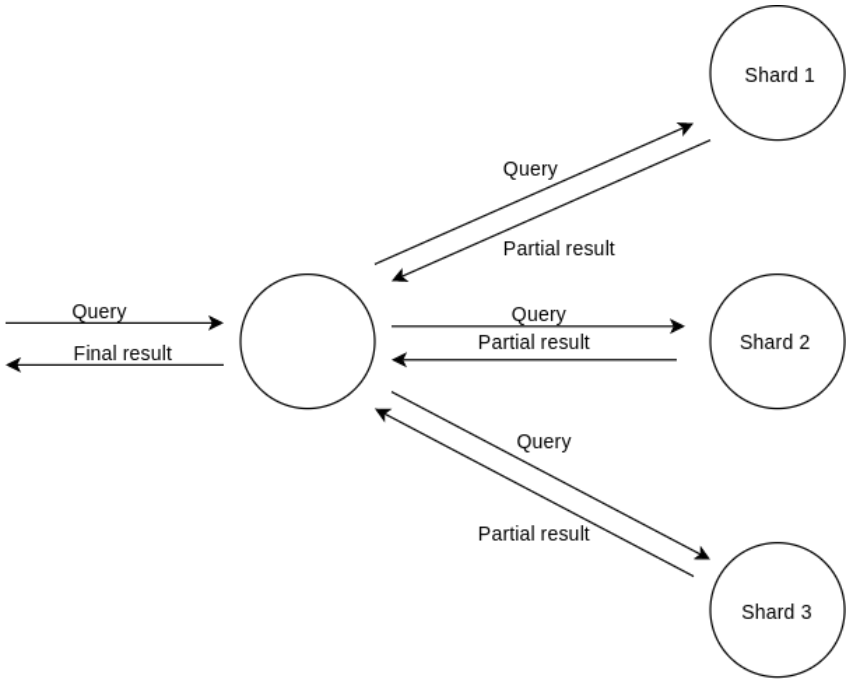

Figure 2.1: Distributed Elasticsearch query.

with the term frequencies 60, 32 and 22 respectively. With global knowledge both the frequencies and one of the top 3 terms have changed.

The example explained above is exaggerated to illustrate what might happen in Elasticsearch's distributed architechture. As the number of documents increase, the error smooths out and will be less likely to happen. Even though the result is not exact, it is good enough for most use cases. Elasticsearch can be configured to return more exact results, at the cost of longer response times.

	Shard 1	**Shard 2**	**Shard 3**
1	blue (10)	blue (20)	blue (30)
2	clouds (9)	clouds (12)	sky (25)
3	sky (5)	insta (5)	insta (15)
4	plane (4)	plane (3)	plane (10)
5	insta (2)	sky (2)	rain (3)

Table 2.2: Term list with the corresponding term count.

2.2 Basic Search Engine Concepts

A common approach for search engines is to use *term frequency* (TF) and *inverse document frequency* (IDF) to calculate a documentś relevance based on a query.

	Shard 1	**Shard 2**	**Shard 3**
1	blue (10)	blue (20)	blue (30)
2	clouds (9)	clouds (12)	sky (25)
3	sky (5)	insta (5)	insta (15)

Table 2.3: Term list of the top three terms returned to the coordination node.

	Returned result
1	blue (60)
2	sky (30)
3	clouds (21)

Table 2.4: Top three terms which are returned to the client.

Documents with the highest TF from a query, are believed to be the most relevant. On the other hand, the most common words are removed as they do not contain information about the topic. TF and IDF alone is a simple model, and Elasticsearch uses a more sophisticated model. Elasticsearch's document scoring model is described in the following subsections. This section describes how Elasticsearch scores its documents and is based on the documentation found on the website [6].

2.2.1 Term Frequency

Term frequency is the number of times a term is mentioned in a document. A document containing a term multiple times is probably more relevant than a document containing fewer occurences. However, in this work a term is only present once in each document, and the reason is described in greater detail in Chapter 4. Term frequency calculation is given by Equation 2.1.

$$\mathbf{tf} = \sqrt{frequency} \tag{2.1}$$

Term frequency calculation in Elasticsearch.

2.2.2 Inverse Document Frequency

Inverse document frequency describes how many times a term is present in all the documents. Terms with high frequencies are often less relevant. E.g. the terms "a" and "an" often appear in a sentence, but should not be given a high score even though they appear numerous times.

$$\mathbf{idf} = 1 + \log\left[\frac{numDocs}{docFrequency + 1}\right] \tag{2.2}$$

Inverse Document Frequency calculation in Elasticsearch.

2.2.3 Document Normalization

A title field is likely to be shorter compared to a description field. As a result, the description field possibly contains more instances of a given term. To account for longer fields, document normalization is used. Elasticsearch's implementation is illustrated in equation 2.3.

$$\textbf{normalization} = \frac{1}{\sqrt{numTerms}} \tag{2.3}$$

Normalization.

2.2.4 Document Score

After calculating term frequency, inverse document frequency and document normalization, the factors are multiplied together. A document's score in Elasticsearch is given by the Equation 2.4.

$$\textbf{documentScore} = tf \times idf \times normalization \tag{2.4}$$

Final document score.

2.2.5 Vector Space Model

The theory presented earlier only describes how to score a single term, but user queries may contain multiple terms. Search engines often apply a technique called *Vector Space Model* on queries with multiple terms. The vector space model represent the query and the document as vectors. The vector is a an array which holds term weights. Vector similarity is calculated between the query and the documents. The documents which are most similar is then returned from the search. The most common technique to calculate similarity is called cosine similarity.

2.2.6 Multiple Term Query

Elasticsearch's underlying technology Lucene, combines the boolean model, TF/IDF and vector space model to score queries with multiple terms against documents [11]. Equation 2.6 shows how each document is scored against a multiterm query. Table 2.5 explains each variable in equation 2.6.

The variables *queryNorm*, *coord* and *getBoost* are described in greater details in the following paragraphs. *queryNorm* or *query normalization factor* are used to make results from different queries comparable. The factor is calculated using equation 2.5. *sumOfSquaredWeights* is determined by adding the idf value of all the terms in the query, and squaring the result afterwards. As a result, every document will have the same query normalization factors.

The *coord* variable stands for *coordination factor*. With the factor, documents which contain most terms from the query will be ranked highest. Without the factor documents with more matching terms would still be ranked higher. However,

$$\mathbf{queryNorm} = \frac{1}{\sqrt{sumOfSquaredWeights}} \tag{2.5}$$

Equation for calculating the query normalization factor

$$\begin{aligned}\mathbf{score}(\mathbf{q}, \mathbf{d}) = & coord(q,d) \times queryNorm(q) \\ & \times \sum tf(tind) \times idf(t)^2 \times t.getBoost() \times norm(t,d)\end{aligned} \tag{2.6}$$

Equation for scoring documents when searching with multiple terms. Each variable is described in table 2.5.

the boost factor gives documents with more matching terms an even higher score compared to documents with less matching terms. For instance, a query might contain two terms, with a term weight of 3. If the score where calculated without the boost factor, documents with one matching term would recieve a score of 3 and documents with two matching terms would recieve a score of 6. Calculating the score with the boost factor, documents with one matching term recieves a score of $(3 \times 1)/2 = 1.5$, and documents with two matching terms recieve a score of $(6 \times 2)/2 = 6$.

Lastly, the variable named *getBoost* is used to make some field impact more on the document score, compared to other fields. In Elasticsearch there are two different methods of boosting: query time boosting and index time boosting. Index time boosting means that all terms in a specified field, will recieve the boost factor during indexing. Query time boosting, on the other hand, calculates and adds the boost factor when the query is running on the Elasticsearch node.

Variable	Description
t	term
d	document
q	query
score	document score from a given query
coord	coordination factor
queryNorm	query normalization factor
tf	term frequency
idf	inverse document frequency
getBoost	boost factor used on the query
norm	document normalization factor

Table 2.5: Variable descriptions for equation 2.6.

2.3 Relevance Feedback

The idea behind *relevance feedback* is to use the result from the initial query to extract relevant information from the top-k documents. Once the information is extracted, a new query is executed with extracted information. Results from the second query are returned to the user. The assumption is that the second query returns documents which are more relevant to the user.

In *Modern Information Retrieval the concepts and technology behind search* [3] the authors define *relevance feedback* as: "when the user explicitly provides information on relevant documents to a query," and *query expansion* as: "when information related to the query is used to expand it" [3, p. 177]. In other words to use *relevance feedback*, input from the user is needed. For example the user could be given the task to mark whether the documents are relevant or not. In practice it is often difficult for a user to determine the result's relevance. For *query expansion* information like position and tags may be used to expand the query. A more detailed explanation of query expansion is described in section 2.4.

Relevance feedback is divided into three main categories *explicit feedback*, *implicit feedback* and *pseude-relevance feedback*, and is introduced in the next two subsections.

2.3.1 Explicit vs Implicit Feedback

Explicit feedback data are retrieved directly from user interaction. An example would be if the user selects the section "graphic cards" in an online store. From the interaction the user explicitly states that the search should only contain graphic cards. Another approach is to use data from a user search. If a user clicks on a search result, the result may be regarded as relevant. Even though the result may not be relevant, it is a good indication. The problem with explicit feedback, is that it requires interaction from the user.

Implicit feedback on the other hand, does not require any involvement from the user. Examples of implicit user data are collecting the documents from a search result that are opened by a user, and measure time spent viewing a document.

2.3.2 Pseudo-Relevance Feedback

Retrieval of data to use relevance feedback requires either explicit or implicit user interaction. Manually involving the user in the search is undesireable. To avoid this, an approach called pseudo-relevance feedback can be used. Using implicit feedback requires a system which does the data collection and post process the information. Pseudo-relevance, on the other hand, uses information from the first search, and thus leads to a simpler implementation.

Often the top-k documents are used to find pseudo-relevance for query expansion. However, the top-k documents are in many cases not relevant, and thus not suitable data for query expansion [18]. Section 2.4 describes a method to extract information from the top-k documents which is regarded as relevant information.

2.4 Query Expansion

When a user searches using the query "Super Bowl" the day after the sport event has taken place, it is likely that the user wants information about the event from the previous day. The query "Super Bowl" is likely to also return documents from previous years. If the search engine could be able to notice that recent documents also contains the term "2016," the extra term could be added to the query. The new query "Super Bowl 2016" is likely to rank documents from this year's Super Bowl higher, as a result of the extra term. This technique is called query expansion.

The idea behind query expansion is to add more terms to the user's query, and then use the extended query on the search engine. According to literature, a query expanded search does improve the results [3, ch. 5]. Even though research shows promising results, query expansions require explicit information which in practice often is difficult to acquire. E.g. in a free text search users expect to automatically recieve search results without having to answer questions or to filter the result to provide query expansion data. On the other hand, according to Efron [5], hashtags provide an excellent way to acquire the explicit information needed for query expansion.

There are different methods of query expansion, and this report describes one technique called Kullback-Leibler divergence. The implementation is described in chapter 4.

2.4.1 Kullback-Leibler Divergence

Kullback-Leibler divergence (KL) measures how well distribution P(t) represents the distribution Q(t). The variables in distribution P(t) and Q(t) are explained in the bullet points bellow.

- *numberOfTimesInTopKDocuments* is the number of times a term is present in the top-k documents
- *numberOfTermsInTopKDocuments* is the number of terms in total in the top-k documents
- *totalNumberOfTimesInCollection* is the total number of times a term is present in the data collection
- *totalNumberOfTermsInCollection* is the total number of terms in the data collection

Equation 2.7 explains how to calculate the distribution P(t), and equation 2.8 explains how to calculate distribution Q(t).

$$\mathbf{P} = \frac{numberOfTimesInTopKDocuments}{numberOfTermsInTopKDocuments} \tag{2.7}$$

Term	**numberOf-TimesIn-TopK-Documents**	**numberOf-TermsIn-TopK-Documents**	**totalNumber-OfTimesIn-Collection**	**totalNumber-OfTermsIn-Collection**
blue	1	42	14	298,962
2016	2	42	143	298,962
clouds	3	42	31	298,962
sea	1	42	34	298,962
water	1	42	24	298,962

Table 2.6: The returned numbers for each of the top 5 terms excluding the term "sky".

$$\mathbf{Q} = \frac{totalNumberOfTimesInCollection}{totalNumberOfTermsInCollection} \tag{2.8}$$

Computing the Kullback-Leibler divergence for a term is given by equation 2.9.

$$\mathbf{KL}_D[P(t), Q(t)] = P(t) * \log[\frac{P(t)}{Q(t)}] \tag{2.9}$$

Kullback-Leibler Divergence.

Kullback-Leibler Divergence Example

To illustrate how KL score is calculated, an example for the search term "sky" is displayed. The expanded query may consist of up to 5 terms. To keep this example short, only the top 5 terms are calculated and the term "sky" is excluded from the calculations.

Table 2.6 contains information extracted using the data set from my project report. Using equation 2.9 and the information in table 2.6, the KL score for each term is calculated and is shown in table 2.7. The table 2.7 lists the terms descending according to their score. From the original query we have the term "sky" and we may use 4 additional terms to complete the expansion. The top 4 terms are "clouds", "2016", "blue" and "water", which results in an expanded query containg the terms: "sky", "clouds", "2016", "blue" and "water".

Term	**Score**
clouds	0.46678
2016	0.21908
blue	0.14836
water	0.13553
sea	0.12723

Table 2.7: KL divergence score of each term.

3 State of the Art Survey

This chapter gives an overview of some of the research related to this master thesis. The work by Rudihagen and my project is presented in the first section, and the last section presents other related research.

3.1 Previous Work

This master thesis is a continuation of the work done by Rudihagen [26] and my project report [21]. Rudihagen studied how search engines could return more relevant search results, but at the same time deliver the results fast enough to be used in live searches. He looked at how Google Play[1] ranked search results. Google Play is the official app store for Android phones. He found that Google Play's search results ranked the most popular apps highest. Ranking the most popular apps highest, will result in relevant results in many cases, but it will also make less popular apps almost disappear. By utilizing the techniques Kullback-Leibler divergence and Bayesian classification, he was able to return more relevant search results. However, the latency in his implementations ranged from 80 ms to 600 ms. This means that most of the time the search implementation is too slow to be used in a live search. The requirement for interactive applications is 100 ms.

The sequence diagram from Rudihagen's query expansion implementation can be seen in figure 3.1. The sequence diagram shows that the query expansion has two round trips from the web server to the search engine, and two round trips from web server to the database. This is a total of four round trips from the web server to collect the data needed for the query expansion algorithm. According to Rudihagen the measured latency was between 150 ms - 600 ms, and 238 ms in average.

To improve Rudihagen's implementation, the assumption was to decrease the number of round trips. This assumption is based on the results of Rudihagen's other implementations. His other two implementations had one and two round trips, with average latencies of 92 and 108 respectively. This suggests that the number of round trips has a significant impact on the measured latency.

Based on the experience from Rudihagen's master thesis, the project report [21] implemented query expansion with less round trips. The project report describes an implementation which was able to achieve a total of two round trips between

[1] `https://play.google.com`

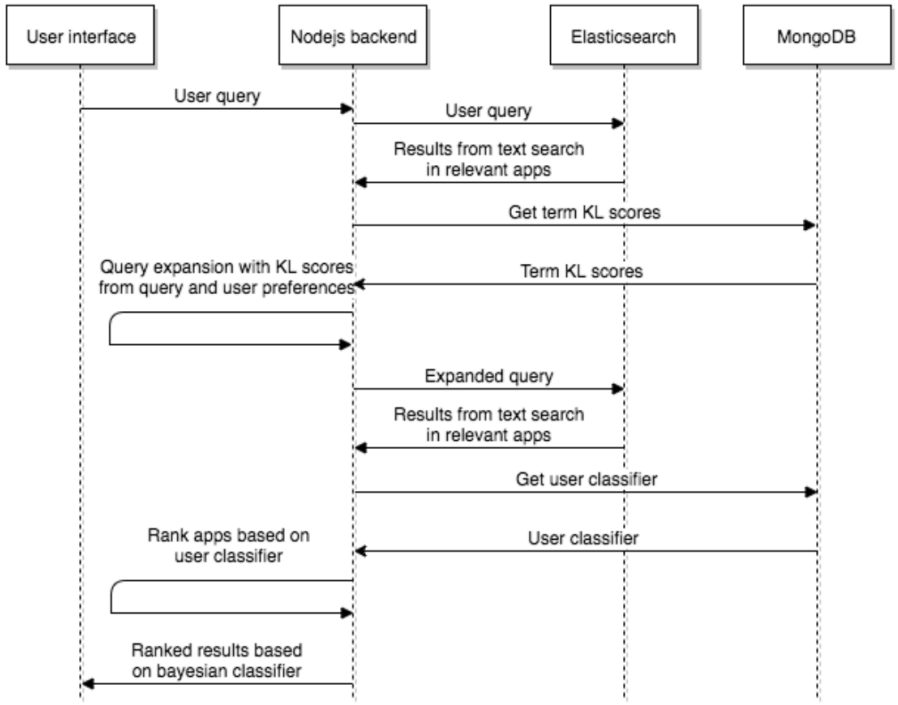

Figure 3.1: Sequence diagram from Rudihagen's implementation of query expansion. Figure taken from [26].

the web server and the search engine. The measured response times were within the limit of 100 ms. An important remark is that all the tests described were done locally. If the tests were conducted with the server placed at a hosting provider, the response times would most likely have been higher.

The project report had two different implementations, one without query expansion and one with query expansion. The implementation without query expansion is used as a baseline. The query expansion implementation had a latency increase of about twice the time compared to the baseline implementation. In a real world environment the increased latency may exceed the 100 ms interactive requirement.

3.2 Other Work

The following section outlines other related research which have been done to deliver more relevant search results.

3.2.1 Query Dependent Pseudo-Relevance Feedback Based on Wikipedia

Pseudo-relevance feedback uses information from the top-k documents to compute query expansion terms. However, the top-k documents are not always relevant and may introduce noise to the data set. *Query Dependent Pseudo-Relevance Feedback based on Wikipedia* explores query dependent expansion on data from Wikipedia [29]. In the paper they found that different field weights had a significant impact on the precision performance. From the results they found high weights on the fields "Links" and "Content" to yield very good results. Increasing the weights on the "Overview" field decreased precision results. The research explored three different methods for query expansion: relevance model, strategy for entity/ambiguous queries and field evidence for query expansion.

3.2.2 Fuzzy Search

With a standard term search, the user has to spell the term correctly as a misspelled term will most likely yield no or few results. E.g the term "sceinci" should also retrieve results on the term "science." There are two character edits required to retrieve the correct term "science." First, switch the two characters "c" and "e" to achieve "scienci." Secondly, change the character "i" to "e" to acquire the term "science."

A study found that 50% of users reformulate their queries, and close to a third of these users reformulated their query three times or more [2]. To handle this problem a technique called fuzzy search may be used [22].

A commonly used method with fuzzy search is Levenshtein distance. Levenshtein distance calculates the number of characters edits which are required to transform one string into another. Editing a string includes substitution, insertion and deletion. Levenshtein distance can be expanded with the method Damerau to allow character transposition.

Elasticsearch uses Damerau-Levenshtein distance to calculate edit distance [8]. The Damerau-Levenshtein distance corresponds to the number of character edits. A distance of 1, means one character has to be changed.

3.2.3 Twitter Query Suggestion Engine

Fast Data in the Era of Big Data: Twitterś Real-Time Related Query Suggestion Architecture is a paper that describes the architecture behind Twitter's suggestion engine [24]. One of the most important requirements for Twitter was to provide relevant search results within minutes of an event taking place. More precicly they wanted to register trending hashtags within 10 minutes of an event happening, and deliver the results in real-time.

The search assistance engine has multiple features to aid the users while searching. Temporal information is searched to deliver information which has been popular the last day, and popular information aggregated over time. Secondly, they have a spelling correction system, which suggests corrections for common misspelled queries.

4 Implementing Query Expansion in a Search Engine

After implementing query expansion in the project report, the challenge was to implement query expansion directly into the search engine. As the Elasticsearch source code is poorly documented, the choice was to implement query expansion with Lucene and then finally port it into Elasticsearch. After implementing query expansion using Lucene, Elasticsearch's documentation was inspected to learn how to make the implementation scalable. It was discovered that Elasticsearch has a plugin API which can be used to extend Elasticsearch's functionality.

This chapter gives an overview of the query expansion algorithm before the implementation with Lucene and Elasticsearch is explained.

4.1 Implementation

Two different platforms were used during the implementation. The initial implementation used Lucene as the search engine.

4.1.1 Algorithm

The algorithm used for query expansion is equal on both platforms, but they have some platform specific differences which are explained in their respective subsections. The algorithm is available as pseudocode in appendix A.7.

The algorithm starts by sending a term search to the search engine. The result from the initial search is often referred to as top-k documents, where k stands from the number of documents in the results. The photos from the result are then looped through to extract all the terms. Each term is stored in a hash map for fast retrieval. On every iteration the term is checked against the hash map. If the term does not exists, the term is added to the hash map. The key is the hash map itself, and the value is an object which stores information about the total number of times the term occurs in the whole collection, the total number of terms in total in the collection and the number of times the term is present in the top-k documents. In the opposite case where the term already exists, the term counter for the current term is incremented. The counter for the number of times the term appears in the top-k documents are incremented by one.

After looping through the photos the information about how many terms there are in the whole collection is retrieved from the search engine. Now all the information required to calculate the KL-score is available. All the keys in the hash map are now iterated through to retrieve every single term. In every iteration the equation 2.8 is used to calculate the KL-score. An array is used to store objects which holds information about which term it is and the corresponding KL-score. Subsequently, the array is sorted from high to low according to their KL-score.

Next the expanded search terms need to be generated. A new string array is created to hold the new search terms. First the old search terms are added to the array, then the new expanded terms are added to the array. In this implementation a maximum of ten terms may be added to the term search.

Lastly, the expanded terms are used in a new term search. The search engine is queried for the terms, and the result from the search is returned directly to the client without any modifications.

Algorithm Complexity

An important aspect of every algorithm is to analyze its complexity. The algorithm explained in 4.1.1 has two different inputs we have to consider. Firstly, the number of terms in top-k documents, and secondly, the number of unique terms. This analysis only examine the algorithm itself, and does not account for the time it takes to retrieve data from the search engine.

The algorithm contains a total of three loops and one sorting algorithm. Initially, all the tags are iterated through, which means the size is equal to the number of tags in the top-k documents. The second loop is of the same size as the number of unique terms. This number has a best case of only one unique tag and a worst case of T unique terms. Lastly, all the unique tags are sorted using a function inside the Java util library called `Arrays.sort()`. `Arrays.sort()` uses an algorithm called merge sort, which has a worst case of $O(n \log n)$.

Combining all the loops described above, the algorithm complexity given in equation 4.1. T is the number of tags, and U is the number of unique terms.

$$\mathbf{f}(\mathbf{T}, \mathbf{U}) = \mathcal{O}(T + U + U \log U) \tag{4.1}$$

Algorithm complexity for the algorithm explained in subsection 4.1.1.

4.1.2 Lucene Implementation

This subsection describes how query expansion was implemented using the Apache Lucene java library. The Lucene implementation was done using version 6.4.0 of Lucene [14].

Indexing

Lucene has multiple data types which may be stored, but the types used in this implementation were `StringField`, `LongPoint` and `TextField`. All the photo tags were indexed using the `TextField`. The two fields `TextField` and

`StringField` are quite similar, but have some important differences. Listing 4.1 displays the `StringField` configuration code from Lucene's Github repository [15]. The first function `setIndexOptions(IndexOptions.DOCS)`, configures the inverted index to only store the documents containing the term. Term frequencies and vector are not stored with this option. `setStored( true )` tells the Lucene index to store the original value. Lastly, the function `setTokenized(false)` informs Lucene to store the string value as a single token.

Listing 4.1: Lucene's `StringField` index configuration.

```
1    setStored(true);
2    setTokenized(false);
3    setIndexOptions(IndexOptions.DOCS);
```

Listing 4.2 shows the `TextField` configuration code from Lucene's Github repository [16]. The difference between the `StringField` and the `TextField` code are line two and three. `setTokenized(true)` tells the Lucene analyzer emit a token for each word in the given string. Most important is line three which tells Lucene to store data about what documents contain a given term, a given terms frequency in each document and the term's position in the original text. Calculating the KL-score of a term requires the term's total frequency across all documents.

Listing 4.2: Lucene's `TextField` index configuration.

```
1      setStored(true);
2      setTokenized(true);
3      setIndexOptions(IndexOptions.
           DOCS_AND_FREQS_AND_POSITIONS);
```

Query Expanded Search

Figure 4.1 shows a sequence diagram for the Lucene implementation. As the sequence diagram shows, the java program first recieves the query. The query is then sent to the Lucene index as a multi term query. Lucene then returns the top-k documents from the index. Calculating the KL-scores requires information about how many times each term appears in the collection, and the total number of terms in the collection. As Lucene exposes a low level interface, this information may be retrieved directly from the inverted index. The function `totalTermFreq` from the class `IndexReader` returns the number of times a term is present across all documents. The inverted index also holds information about the total number of terms in the collection. This number is retrieved by using the function `getSumTotalTermFreq` from the same `IndexReader` class. After all the necessary information has been gathered the KL-score is calculated, and the new expanded search terms are generated. Finally, the multi term search is sent to the index. The final result is then returned back to the client.

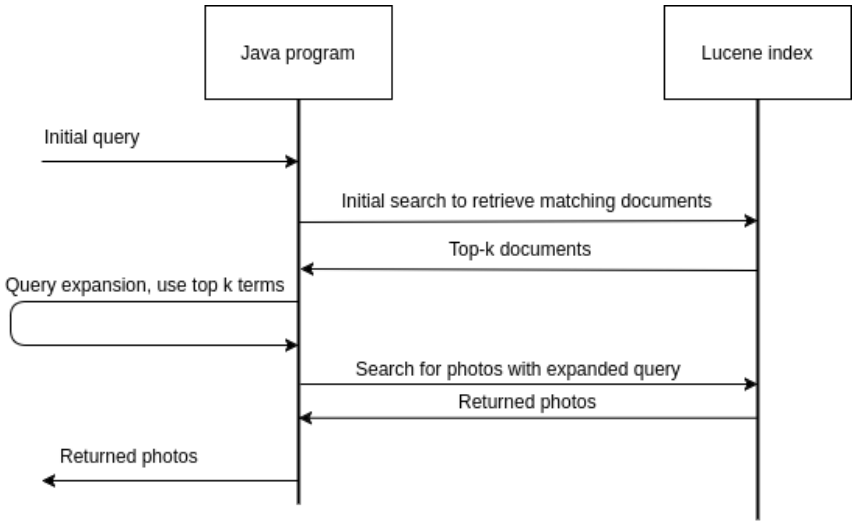

Figure 4.1: Sequence diagram for the Lucene implementation.

4.1.3 Elasticsearch Implementation

To achieve scalability across multiple machines, the second implementation of query expansion was done using Elasticsearch. The initial thought was to implement query expansion directly into Elasticsearch's core. However, during the research phase it was discovered that Elasticsearch has a plugin API.

The query expansion plugin for Elasticsearch is almost identical to the implementation described in subsection 4.1.1 and in subsection 4.1.2. Thus, this subsection will only describe the differences. The differences are mostly how the documents are indexed and how the queries are structured.

Figure 4.2 displays the sequence diagram for the implemented Elasticsearch plugin. First, the query arrives the coordination node in the Elasticsearch cluster. The query is parsed and distributed to all the relevant shards. Each uses the plugin to calculate their local top-k documents. Metadata of each shard is returned to the coordination node, which again merges the results and calculates the global top-k documents. Lastly, the actual documents are retrieved from all the shards, and the final result are returned.

Elasticsearch Plugin API

Elasticsearch has its own plugin API [7]. The plugin API can be used to extend Elasticsearch's index and query capabilities. The implementation described in this master thesis creates a plugin which extends the REST API with query expansion. The plugin API has two main categories: core plugins and community contributed plugins. Core plugins are plugins which are a part of the Elasticsearch project. These plugins are developed by the Elastic team. Community contributed plugins, are plugins outside the Elasticsearch project. These plugins are developed by the community. When a plugin is installed, the installation is distributed to all the

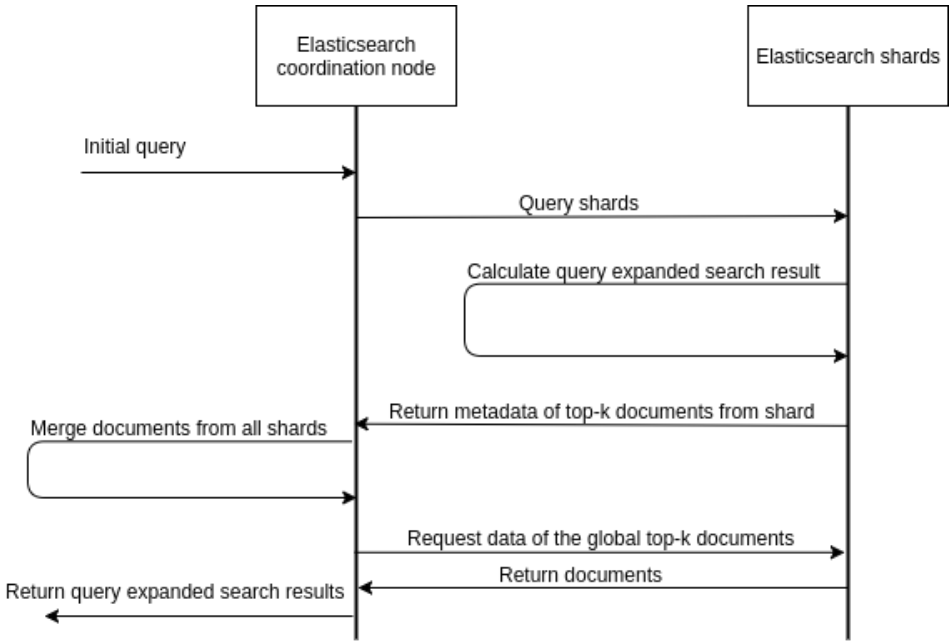

Figure 4.2: Sequence diagram for the Elasticsearch implementation.

nodes. As the installation is distributed, all the nodes are able to act as a coordination node, which also makes the plugin API scalable. The plugin implementation described in this thesis belongs to the community contributed plugins.

Accessing the extended REST API is done sending a POST request to the URL `http://localhost:9200/_expansion`. This URL assumes that the server is running locally and that Elasticsearch's default port 9200 is used. `/_expansion` is the new REST API extension. To send a query, the POST request need to have a body as shown in listing 4.3. The body consists of a json object with one key value pair. The key is `search_query`, and the value is the desired query string. Query strings with containing multiple terms are divided into separate terms. The terms are extracted by splitting the query string on the character "space".

Listing 4.3: The POST request body for the implemented query expansion.

```
1 {
2   "search_query": "search terms"
3 }
```

Indexing

Many storage solutions require the stored data types to be defined before inserting data. Elasticsearch has support to both predefined data types and data types determined at index time. Data types determined while indexing are called dynamic mapping, and predefined data types are called static mapping. Dynamic mapping

are useful when prototyping and developing. However, dynamic mapping may assign wrong types to a field. For instance one might have a geo location point. A geo location point will most likely be sent to Elasticsearch as two floats: latitude and longitude. It is fine to store the values as floats, but Elasticsearch has a separate data type for latitude and longitude called Geo-point.

With static mapping, on the other hand, every mapping type is defined before indexing. Static mapping ensures that each field is assigned the correct mapping. The mapping used in this master thesis is available in appendix A.2.

Searching

All the searches in the Elasticsearch implementation is similar to the searches described in the Lucene implementation, but there are a few important differences. The query expansion algorithm requires three different types of search: single term query, multiple term query and field stats query.

The first step of the algorithm is to retrieve the top-k documents. In Elasticsearch this is done as a multiple term query, and the implemented Java code is available in appendix A.4.

Retrieving the number of times a term appears in the collection may be done through Elasticsearch's count API. However, this API is not available through the Java client library. Instead this information may be fetched using a normal single term query, and the Java code used can by found in appendix A.3. Metadata from the search result contains the needed information. Listing 4.4 displays an example on how the returned result may look. The array `hits` is intentionally left out to keep the example short, as the array contains the search result itself. The search result consists of information about how long the search took, how many shards searched, the total number of hits, the maximum returned document score and the search result itself. In listing 4.4 there is a field called `total`, which indicates the total number of hits the term query produced.

Lastly, the query expansion also requires to know the total number of terms in the collection. This information can be retrieved using Elasticsearch's field stats API. The API exposes native Lucene index function, for retrieving metadata from the inverted index.

Listing 4.4: Example of the metadata returned by Elasticsearch.

```
{
  "took": 3,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 2912,
    "max_score": 9.73782,
    "hits": [
```

```
13            ...
14        ]
15    }
16 }
```

Architecture

The architecture of the implementation can be seen in figure 4.3. As described earlier, the query first arrives a coordination node before it is parsed and sent to the data nodes in the cluster. Node 2 is enlarged to display what this master thesis has implemented. In figure 4.3 the node contains a plugin called "Query Expansion," together with data called shard 1 and replica 1. When a request of a certain format arrives, the plugin handles the request. The format is explained in section 4.1.3. After the plugin has calculated the top-k documents, the result is returned to the coordination node, and later returned to the client.

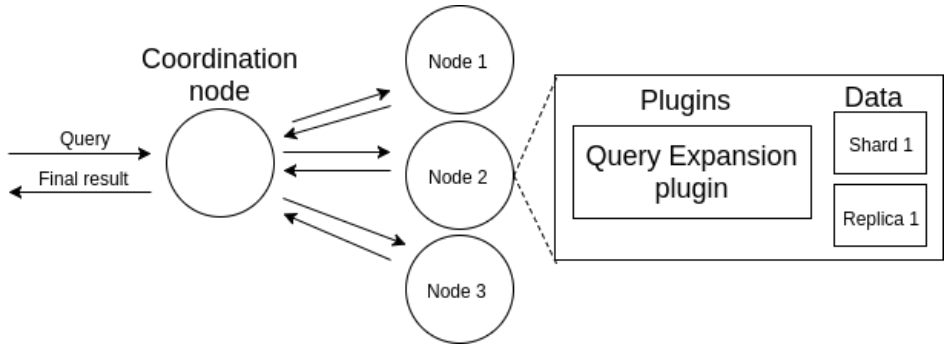

Figure 4.3: Architecture for the implemented Elasticsearch plugin. The figure is an example of a cluster setup with a view inside one of the nodes in the cluster.

5 Evaluation

This chapter explains how the implementations were tested together with the results. The first section explains how the Lucene and the Elasticsearch experiments were conducted. Section 5.2 shows the results and how the results were obtained, and section 5.3 discusses the results. Lastly, section 5.4 evaluates the research questions against the results.

5.1 Experimental Setup

Query expansion using pseudo relevance turns out to yield better results in terms of relevance compared to a TF/IDF search, which is the common approach in today's search engines. Even though query expansion delivers more relevant search results, the results have to be available fast enough to be used in interactive environments. In other words, the results have to be available within 100 ms. Both implementations will be evaluated against a baseline search. The baseline search will in both cases be a multi term search.

Query expansion requires the search engine to process an initial query, and is then sending a new query to retrieve the final result. As the search requires two searches, the query expansion implementation will naturally yield higher response times. The most interesting results are how much slower the query expansion is compared to the baseline search.

5.1.1 Data Set

Both experiments use the data set described in this section. The data set consits of photo data gathered from the Flickr API[1]. Flickr was chosen as the data source as tags have been found to provide strong data for query expansion [5]. All the photos were retrieved using Flickr's endpoint for fetching the last published photos. However, the API limits the number of requests per hour. The photos were gathered over a period of about three months, March 2017 to the end of May 2017. The data set used in the experiment consists of 4,561,816 photos and 9,993,411 tags. Of all the photos, a total of 1,708,324 photos have 1 or more tags, which means that 37 % of the photos have tags. Counting all the photos with at least one tag, each photo has 5.8 tags in average. Every tag is user created by the person

[1] `https://www.flickr.com/services/api/`

who posted the photo. Flickr also has an analyzer which ads additional tags, but these tags are not available through the API.

The pie chart in figure 5.1 shows the number of tags on each photo, and the chart shows that photos with 5 tags are the largest group. Most of the photos have multiple tags, which improves the tag generation. If the top-k documents only have one of the same tag, the algorithm will not generate additional tags. As a result, more tags results in better data for the query expansion algorithm.

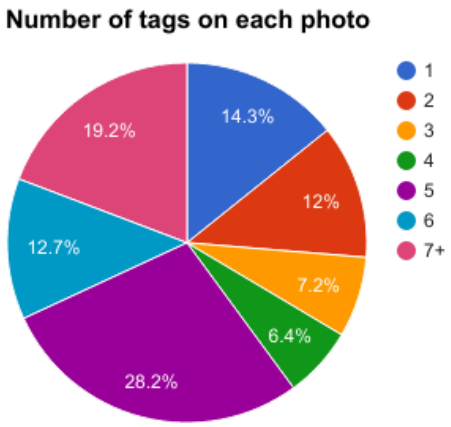

Figure 5.1: Pie chart that shows the distribution on how many tags each photo have.

5.1.2 Lucene Experiment

All the data types indexed in the Lucene experiment can be seen in table 5.1. Even though only the tag fields are used in the algorithm, more fields are indexed to make the experiment closer to a real world application. `StringField` will be indexed as a single token, which is useful for aggregating or filtering on the field. `TextField` is indexed for full-text search. With the full-text search indexing the field also stores the term frequencies, which is required for the query expansion algorithm. The reason for using `LongPoint` on a date is to make it efficient to range searches.

A different performance metric is used for the Lucene experiment, as Lucene is a Java library and is not a complete framework with a REST API compared to Elasticsearch. The Lucene implementation is evaluated by measuring the time from the query is sent to index until the result is retrieved. This is not a real world experiment setup, but this evaluation is used to reveal how much slower the query expansion algorithm is compared to the baseline query. To retrieve

Field	Type
id	StringField
title	TextField
description	TextField
dateuploaded	LongPoint
urls	Array of StringField
tags	Array of TextField

Table 5.1: Field types of every field used in the experiment.

an accurate measurement a Java method called `System.nanoTime()` is used. `System.nanoTime()` is the most precise method available in Java to measure elapsed time.

5.1.3 Elasticsearch Experiment

Most web applications today with a search field will have a web server and a search engine on the backend. Figure 5.2 displays a common setup for web application with search capabilities. The client sends a search to the web server which process the request and sends it to the search engine. After finding the most relevant documents, the result is sent back from the search engine to the web server, which again sends the result to the user. As mentioned earlier, the requirement for interactive applications are that no operation takes longer than 100 ms. Response time is measured from the request is sent from the client to the response arrives at the client. To verify that the query expansion plugin for Elasticsearch is fast enough to meet the interactive requirements, the experiment was setup as shown in figure 5.2. The client on the figure was the author's laptop, and the backend was setup to mimic a real world environment. The backend consists of two main components, a web server and a search engine.

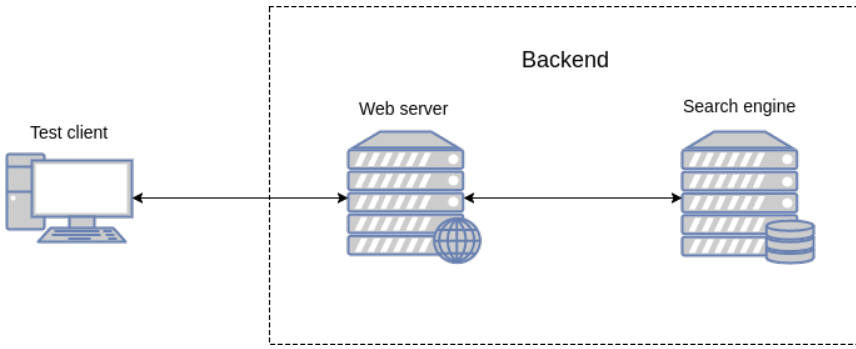

Figure 5.2: Overview of the Elasticsearch experiment setup.

As web applications often use cloud providers, the tests also needed to be conducted using cloud providers. The requirement set for the cloud provider was: need to be widely used, have servers in Europe and provide VPS services. Possi-

ble providers were: Amozon Web Services[2], Google Cloud Platform[3] and Digital Ocean[4]. Google Cloud Platform was chosen as the service provider, as you have more flexibility to choose between number of cores and the memory size and the author had knowledge of the platform. Tests were conducted using two Google Compute Engine instances. Web services always strive to place the servers as close to the users as possible. To make the experiment simulate a real world scenario both the instances were placed in the reagion called *europe-west1-c*.

Elasticsearch Instance

The instance running Elasticsearch had the following specifications: 2 vCPUs, 10 GB memory and 20 GB SSD. Elasticsearch's documentation [9] suggests that memory will be the most important resource in most use cases. As a result more memory was chosen over the number of CPUs. An important setting in Elasticsearch is the heap size. By default the heap memory size is set to 1 GB, but was changed to 5 GB in the test environment. A logical assumption would be to set the Elasticsearch to use all the available memory, except the memory needed for the operating system. However, Elasticsearch's underlying structure Lucene also needs memory. Lucene stores the data in separate files. The data structure inside the files is immutable, which makes them optimized for caching. With this strategy Lucene optimizes the underlying operating system's eager to hold small and often used files in memory. According to the Elasticsearch documentation [10], the heap size should be set to 50% or less, of the available memory.

Most operating systems today also comes with swapping turned on by default. If the operating system decides to swap, it would significantly reduce the performance. To avoid the problem, swapping was turned off on the Elasticsearch instance.

NodeJS Instance

The instance running NodeJS had the following specifications: 4 vCPUs, 4 GB memory and 10 GB SSD. On the web server we want to be able to handle as many requests as possible. The number of requests the server is able to handle are closely linked to the number of cores. That is why the Node.js instance has more cores at the cost of less memory.

Node.js is by design single threaded, which would make 3 of the cores on the Node.js server being idle. However, this problem can be solved by using tool called pm2[5]. pm2 has a feature called *cluster mode*, which may spawn multiple Node.js instances. To allow maximum CPU utilization, pm2 can be configured to spawn as many Node.js instances as the number of cores.

To test the implementation the Node.js web server implemented two different endpoints. One endpoint is used to test the query expansion plugin, and the other is used to test a multi term query.

[2] `https://aws.amazon.com/`
[3] `https://cloud.google.com/`
[4] `https://www.digitalocean.com/`
[5] `http://pm2.keymetrics.io/`

Performance Metrics

Evaluating the Elasticsearch plugin is done by measuring the response time from a client sends the request to the response arrives. The response time can be broken down into two separate parts: latency and execution, as shown in figure 5.3. In this thesis latency, is defined as the time it takes for a signal to go from the client to the web server and back. Execution time is defined from the web server recieves a request from the client, until the web server sends a response back to the client. This means that the execution time also includes the processing time on the search engine.

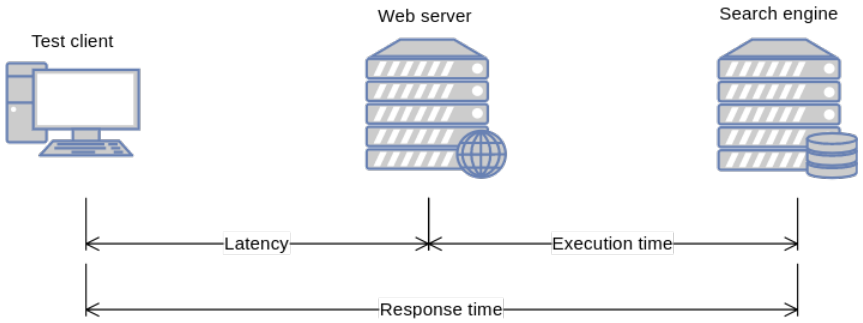

Figure 5.3: Overview of the different measurements used when evaluating the implementation.

5.2 Results

The results from the experiments in the project report [21], showed that the query expansion implementation had about 2 times longer latency compared to the baseline implementation. The latency was measured from the request left the user to the response from the server arrived.

All the results show that the first request is often the slowest. After the initial request, the response is cached by Lucene and makes all the subsequent requests a lot faster.

5.2.1 Lucene Results

The query expansion implementation was evaluated by measuring the time from the Java program sends the request to Lucene index, until a result is returned from the index. Before starting the tests, the index cache was prewarmed using the terms `square` and `insta`. The prewarming was done sending 10,000 requests to the Lucene index. 5,000 of the requests were multi term queries and the other 5,000 requests were query expansion queries. With half of the results each, both term types will be present in Lucene's cache. Figure 5.4 displays the measured response times from the Lucene experiments. The red line is response times from the query expansion search, and the blue line displays the baseline query response

times. Even though the measured response times is within the requirement for interactive applications, the experiment is not a good indicator for how it would have performed in a real world environment. More interesting is how many times slower query expansion is compared to the baseline query. When the result size contains 10 results, query expansion is more than 3 times slower compared to the baseline query, and it increases to more than 5 times slower when the result size is 200. This means that the response times are increasing faster with query expansion compared to the baseline query.

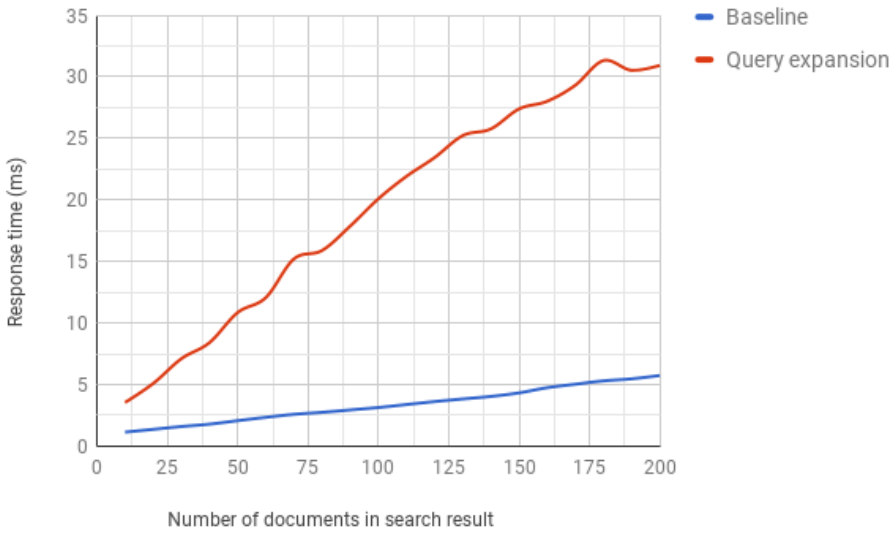

Figure 5.4: Response times from the Lucene implementation with varying result size.

5.2.2 Elasticsearch Experiment Results

As mentioned earlier, the Elasticsearch plugin is evaluated by measuring the response time from the client sends the request until the response arrives at the client. The response times were measured using a command line tool called ApacheBench (`ab`) [28], and the actual command used can be found in appendix A.8. `ab` collects min, average, median and the max response time for all the responses. The results also includes information about the different parts of the response time: connection time, processing time and waiting time.

To achieve reliable test results, each request with `ab` was executed 10,000 times. Two different tests were conducted: one with cache prewarming and one without. The prewarming query terms are `square` and `insta`. `square` is the most used tag, and `insta` is one of the least used terms. `sky` and `blue` are the actual terms used in the test query. Between each test, the cache was first flushed, and then warmed again. Each of the tests also tested the response time when the result size

increased. By default, Elasticsearch returns the top 10 results, and the tests varied the result size from 10 to 200 with a step size increase of 10. Figure 5.5 illustrates the response when the cache is prewarmed, and figure 5.6 illustrates the response times without prewarming the cache. The difference between the two graphs are so small that they barely are visible. Lucene requires a fair amount of requests before the caches are optimized. At a result size of 10, query expansion is 20 % slower compared to the base line query, and increases to about 250 % slower with a search result size of 120. After that point, the query expansion implementation decreases to 205 % with a search result size of 200.

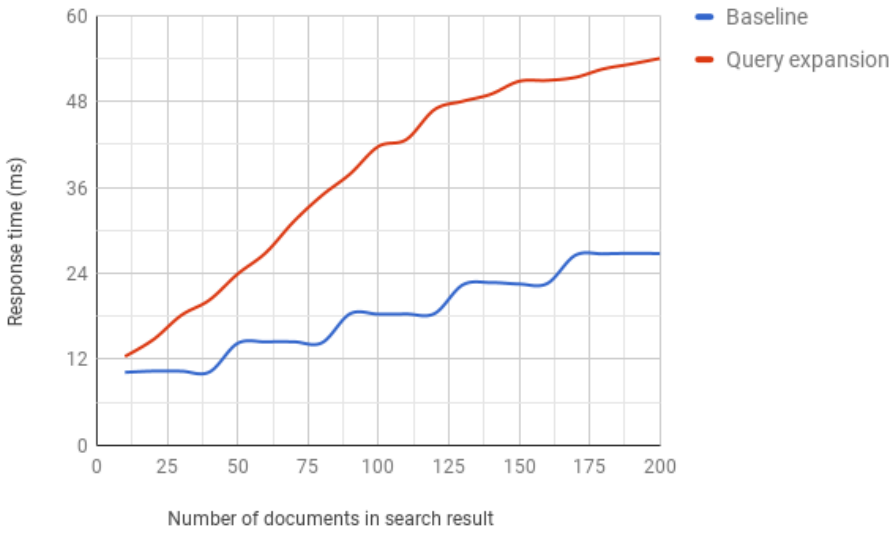

Figure 5.5: Response times using different result sizes with cache prewarming.

5.3 Discussion

This thesis' goal is to explore how advanced techniques can be used to give more relevant search results using an open source search engine, and at the same time deliver the results fast enough to be used with interactive applications. The results in the previous section show that both the Elasticsearch and the Lucene implementation have a linear increase in response time, compared to the baseline test. The difference between the baseline test and the query expansion in both implementations peaked when the search result had 120-130 documents. When the search result size increased even further, the difference decreased in both experiments. This shows that the implementation scales on both implementations. Even though the query expansion implementation is slower compared to the baseline, the results are well within the interactive requirement of 100 ms.

Elasticsearch was chosen as the platform has proven to scale to petabytes of

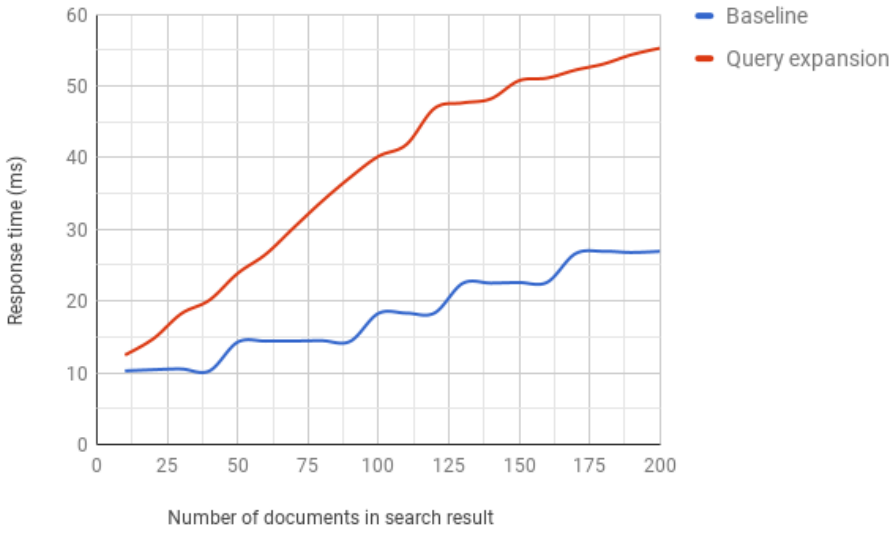

Figure 5.6: Response times using different result sizes without cache prewarming

data [1]. However, implementing techniques like query expansion and be fast enough at the same time, requires the implementation to be done within Elasticsearch source code or as standalone plugins. Elasticsearch's official documentation [12] for developing plugins is limited. The documentation only describes the required setup and links to other open source plugins to learn how plugins are developed. On Elastic's offical website there is a discussion about developing plugins for Elasticsearch. The quote below is an answer from the Elastic's Q&A site by Elastic developer Adrien Grand on wheter there is an offical guide on how to develop Elasticsearch plugins [20].

> No, there is no guide about writing plugins and the API is actually quite unstable. The plugin API is mainly a way for us to provide additional functionality through plugins so that we do not have to fold everything into a single release artifact that would be quite huge. Some community membors have writter plugins by taking inspiration of existing plugins but we do not want to commit on a stable API for plugins as this might prevent us from improving other areas of elasticsearch.

The answer from Grand indicates that Elastic will not develop an official support for Elasticsearch plugins in the near future. Recently, Elastic released a beta version of their machine learning plugin, but this plugin contains closed source code and requires a license to use. Without more support from Elastic, it would become hard to continue to develop more advanced techniques to increase the relevancy of search results.

My project report [21] implementation used aggregation to retrieve information. Aggregation has two important downsides, aggregations may consume a

great deal of memory, and aggregations are approximations and may not necessarily return the correct value. An additional remark decribed in the project report is that the aggregation might not retrieve all the terms mentioned in the top-k documents. With the implementation described in this thesis all terms in the top-k documents are guaranteed to be retrieved and considered in the query expansion. The aggregation query was needed to retrieve the number of times a term appeares in the complete collection. In the Elasticsearch plugin, this query is replaced by a single term query, which returns the exact number of hits. The downside of this approach being that the query has to be executed for each unique term in the top-k documents.

Both implementations have potential of increased speed, by optiziming the Java code. An example of an optimization would be looking at the loops. Some of the loops create new objects on each iteration instead of reusing one object. This may in many cases be inefficient as Java has a garbage collecter which has to clean up the old objects after each iteration. If the code was optimized with the garbage collector in mind, the code would most likely be significantly faster, especially when the search result size increases and thus the loop size in the algorithm. Java optimizations is outside the scope of this master thesis and thus never a focus during the implementation.

In Rudihagen's implementation of query expansion had a total of 4 round trips between the web server, the database and the search engine. In my project report, I was able to reduce the round trips to two times. Implementing an Elasticsearch plugin enabled me to reduce th number of round trips to one. Even though the same amount of work is done, all the work is done on the search engine itself. As a result, the response times are greatly improved by removing round trips and essentially removing unnecessary latency. Rudihagen measured response times between 150 - 600 ms which is well above the requirement for interactive tasks. The average measured response times in this thesis ranged from 12 ms to 54 ms depending on the result size. These results are within the requirement of 100 ms.

5.4 Research Question Evaluation

The following list contains a discussion of the research questions from chapter 1.

1. **How to make an improved search in terms of relevance compared to TF-IDF?** As mentioned in section 1.2 the main focus areas were scaling and latency. Therefore, the relevance was never measured. Based on the work by Rudihagen, this thesis assumes that query expansion returns more relevant results compared to the baseline search.

2. **How to develop an improved search which scales with an increasing amount of data?** The Elasticsearch plugin was tested with a index size of 2.3 GB, which is not an extreme data size in today's measurements. The query expansion plugin was about 20 % slower compared to the baseline search. As Elasticsearch has proven to scale to petabytes of data [1] together with good results described earlier in this chapter, we can assume that the plugin would be able to scale on a larger Elasticsearch cluster.

3. **How to delevop an improved search that fulfills the interactive latency requirements?** The results shows that the implementation is well within the limit for interactive tasks. Even when the search result size increases to 200 documents, the search is delivered within 54 ms on average. All the tests were conducted on a setup close to a real world web application, which indicates that the implementation also would work on an actual web application.

6 Conclusion & Further Work

This chapter discusses the query expansion plugin and compares the implemenation to the work by Rudihagen. Lastly, possible improvements are described to further improve the performance in terms of response time and in terms of relevance.

6.1 Conclusion

This thesis investigates how to implement algorithms to improve the relevance of search results compared to a search using TF-IDF. The final implementation is a proof of concept of whether techniques like query expansion can be implement on an open source search engine, and at the same time deliver search result instantaneously. The experiment setup is close to a real world web application, with both the web server and the search engine is hosted by a cloud provider. During the experiment a laptop were used to measure the response times. The experiment is close to what a user would expect using a website, but there are two things the experiment does not take into account. Firstly, web browser introduces some overhead to render the results. Secondly, an actual website would most likely have more than one user.

This thesis used the work from Rudihagen's master thesis [26] and my project report [21] as a starting point. Rudihagen focused on how instant searches may be more personal and relevant for the user. Rudihagen's implementation had a significant drawback in that the response time were above 100 ms. My project report looked at how the number of round trips in Rudihagen's implementation may be decreased to achieve a lower response time. By reducing the number of round trips to two rounds the latency were significantly descreased. In this thesis the query expansion algorithm were moved from the web server to the search engine, and ultimatly the round trips were changed to one. There is one important difference between the implementation described in this thesis compared to the work by Rudihagen. Rudighagen implemented a personalized search, which means that different users with the same search may recieve different searh results. In this thesis the same query by different users will result in the same search results.

The results shows that the increased response time increases linearly. From the results we can conclude that the query expansion plugin would most likely scale with increasing amounts of data. Before using the plugin in production it is recommended that the plugin implements Java specific optimizations. The com-

pany behind Elasticsearch have no plans to create official guide to develop plugins for Elasticsearch. This makes further developments of plugins demanding without proper documentation. Even though the code for Elasticsearch is open source, there are no official documentation on how the code is structured. This makes further development challenging. As a result the author of this thesis is uncertain whether Elasticsearch is a suggested platform to implement more advanced search techniques.

6.2 Further Work

The query expansion plugin for Elasticsearch described in this thesis is not generic and will only work on the photo data from Flickr. Further development of the plugin would require the plugin to be generic and to work with any type of data.

63 % of the photos did not have any tags. Using the query expansion implementation described in this thesis none of the photos were present in any of the search results. To achieve a better search result the query expansion should also include other fields like the title and the header. Two additional features to explore would be geolocation and when the photo is taken. In other words, images taken closer to the user would be ranked higher and photos taken closer to the current time would aslo be ranked higher. However, there is no guarantee that geolocation and the photo's capture time would increase the relevance of the search results

The query expansion plugin have room for improvements when it comes to Java optimizations. Java is a language were the Java Virtual Machine takes care of memory management for the programmer using a garbagde collector. Even though the programmer does not have to manually to memory management, the program can minimize the need for garbage collection.

The main focus in this thesis is speed. As a result, the search relevance performance was never measured. In further development the query expansion implementation should measure search relevance and compare it to the baseline search.

A | Appendix

A.1 Flickr Data Representation in Elasticsearch

Listing A.1: Internal photo data representation in elasticsearch

```
{
    id: id,
    title: "title",
    description: "description",
    tags: [
      "blue",
      "sky"
    ],
    dateuploaded: "1489805142",
    urls: [
      "url1",
      "url2"
    ]
}
```

A.2 Elasticsearch Static Mapping

Listing A.2: The static Elasticsearch mapping used on the photo index in the experiment setup.

```
{
  "mappings": {
     "photo": {
        "properties": {
           "dateuploaded": {
              "type": "text",
              "fields": {
                 "keyword": {
                    "type": "keyword",
```

```
                    "ignore_above": 256
                }
            }
        },
        "description": {
            "type": "text",
            "fields": {
                "keyword": {
                    "type": "keyword",
                    "ignore_above": 256
                }
            }
        },
        "id": {
            "type": "text",
            "fields": {
                "keyword": {
                    "type": "keyword",
                    "ignore_above": 256
                }
            }
        },
        "tags": {
            "type": "text",
            "fields": {
                "keyword": {
                    "type": "keyword",
                    "ignore_above": 256
                }
            }
        },
        "title": {
            "type": "text",
            "fields": {
                "keyword": {
                    "type": "keyword",
                    "ignore_above": 256
                }
            }
        },
        "urls": {
            "type": "text",
            "fields": {
                "keyword": {
                    "type": "keyword",
                    "ignore_above": 256
                }
```

```
57                     }
58                 }
59             }
60         }
61     }
62 }
```

A.3 Single Term Query

Listing A.3: Java code used to search for a single term.

```
1    SearchResponse searchResponse = client.prepareSearch(
         INDEX_NAME)
2       .setQuery(QueryBuilders.termQuery("tags", term))
3       .setSize(0)
4       .get();
```

A.4 Multiple Term Query

Listing A.4: Java code used to search for multiple terms.

```
1  SearchResponse searchResponse = client
2       .prepareSearch(INDEX_NAME)
3       .setQuery(QueryBuilders.termsQuery("tags",
            termsQuery))
4       .setSize(searchResultSize)
5       .get();
6
7  return searcchResponse;
```

A.5 Query to Retrieve the Number of Occurences for a Given Term

Listing A.5: Java code used retrieve the number of occurences for a given term in the collection.

```
1  SearchResponse searchResponse = client.prepareSearch(
       INDEX_NAME)
2       .setQuery(QueryBuilders.termQuery("tags", term))
3       .setSize(0)
4       .get();
5
6  long numberOfTimesInCollection = searchResponse
```

```
7       .getHits()
8       .getTotalHits();
9
10  return numberOfTimesInCollection;
```

A.6 Field Stats Query

Listing A.6: Java code used retrieve the total number of terms in a field in the collection.

```
1   FieldStatsResponse fieldStatsResponse = client
2       .prepareFieldStats()
3       .setFields(FieldNames.TAGS_FIELD_NAME)
4       .get();
5
6   long numberOfTermsInCollection = fieldStatsResponse
7       .getAllFieldStats()
8       .get(FieldNames.TAGS_FIELD_NAME)
9       .getSumTotalTermFreq();
10
11  return numberOfTermsInCollection;
```

A.7 Pseudocode

Algorithm 1 Psudocode for the query expansion with pseudo-relevance algorithm.

Require: Search terms from the user is defined as *searchTerms*
return Query expanded search terms

initialSearchResult ← termsSearch(*searchTerms*)
// TermData is an object which hold term statistics such as the number of times in top-k documents
// Dictionary with the term as key and TermData as object
termsData ← []

// Array of sorted TermData object
sortedTerms ← []

numberOfTermsInTopKDocuments
for *photo* in *initialSearchResult* **do**
  for *term* in *photo* **do**
    if *term* **then** exists in *termsData*
      currentTermData ← getTermFromDictionary(*term*)
      incrementNumberOfTimesInTopKDocuments(currentTermData)
    else
      numberOfTimesInCollection ← getNumberOfTimesInCol(*term*)
      insertTermsData(*termsData*, *numberOfTimesInCollection*)
    end if
    numberOfTermsInTopKDocuments+ = 1
  end for
end for

numberOfTermsInCollection ← getNumberOfTermsInCollection()
for *termData* in *termsData* **do**
  // Kl score of each term is stored within the termData object
  calulateKlScore(termData, numberOfTermsInCollection, numberOfTermsInTopKDocuments)
  insertTerm(*sortedTerms*, *termData*)
end for
sort(sortedTerms)
// Retrieve the top n search terms
queryExpandedSearchTerms ← getTopTerms(*sortedTerms*)

A.8 ApacheBench

ApacheBench [28] were used to measure response times. The bullet list below contains a description of the command at the end of this section.

- `-n` defines the number of times to send a request.
- `-c` defines the number of concurrent request.
- `-l` is used to allow responses of different sizes
- `localhost` have to be changed to the IP or domain name of the web server.
- The url contains two parameters, `q` and `size`. `q` is the acutal search query and `size` defines the size of the search result.

```
ab -n 10000 -c 10 -l http://localhost/expansion?
q=sky+blue&size=10
```

References

[1] Alquiza J. Field notes - ElasticSearch at petabyte scale on AWS. https://grey-boundary.io/field-notes-elasticsearch-at-petabyte-scale-on-aws/, 2016. Accessed: 07.12.2016.

[2] J. Pedersen B. Jansen, A. Spink. A temporal comparison of altavista web searching. pages 559–570, 2005.

[3] R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval the concepts and technology behind search. CRC Press, 2011.

[4] Jake D. Brutlag, Hilary Hutchinson, and Maria Stone. User preference and search engine latency. In JSM Proceedings, Qualtiy and Productivity Research Section., Alexandria, VA, 2008.

[5] Miles Efron. Hashtag retrieval in a microblogging environment. 2010.

[6] Elastic. Elasticsearch controlling relevance. https://www.elastic.co/guide/en/elasticsearch/guide/current/controlling-relevance.html. Accessed: 16.11.2016.

[7] Elastic. Elasticsearch plugins and integrations. https://www.elastic.co/guide/en/elasticsearch/plugins/5.2/index.html. Accessed: 08.06.2017.

[8] Elastic. Fuzziness. https://www.elastic.co/guide/en/elasticsearch/guide/current/fuzziness.html. Accessed: 08.06.2017.

[9] Elastic. Hardware. https://www.elastic.co/guide/en/elasticsearch/guide/current/hardware.html. Accessed: 08.06.2017.

[10] Elastic. Heap: Sizing and swapping. https://www.elastic.co/guide/en/elasticsearch/guide/current/heap-sizing.html. Accessed: 08.06.2017.

[11] Elastic. Lucene's practical scoring function. https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html. Accessed: 08.06.2017.

[12] Elastic. Elasticsearch plugins and integrations. https://www.elastic.co/guide/en/elasticsearch/plugins/5.2/index.html, 2017. Accessed: 30.05.2017.

[13] Facebook. News feed fyi. https://newsroom.fb.com/news/category/news-feed-fyi/. Accessed: 08.06.2017.

[14] The Apache Software Foundation. Apache lucene 6.4.0 documentation. https://lucene.apache.org/core/6_4_0/index.html. Accessed: 18.05.2017.

[15] The Apache Software Foundation. Lucene stringfield documentation. https://github.com/apache/lucene-solr/blob/master/lucene/core/src/java/org/apache/lucene/document/StringField.java. Accessed: 08.06.2017.

[16] The Apache Software Foundation. Lucene textfield documentation. https://github.com/apache/lucene-solr/blob/master/lucene/core/src/java/org/apache/lucene/document/TextField.java. Accessed: 08.06.2017.

[17] The Apache Software Foundation. Apache lucene core. http://lucene.apache.org/core/, 2016. Accessed: 08.06.2017.

[18] J. Gao G. Cao, J.-Y. Nie and S. Robertson. Selecting good expansion terms for pseudo-relevance. pages 243–250, 2008.

[19] Carlos A. Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. ACM Trans. Manage. Inf. Syst., 6(4):13:1–13:19, December 2015.

[20] Adrien Grand. Discussion thread on elastic's offical website about developing elasticsearch plugins. https://discuss.elastic.co/t/how-to-write-elasticsearch-plugin-for-2-2-version/55419/4. Accessed: 22.05.2017.

[21] Lund J. Instant search using query expansion with pseudo-relevance, 2016.

[22] Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. Efficient interactive fuzzy keyword search. In Proceedings of the 18th international conference on World wide web, pages 371–380. ACM, 2009.

[23] Mayer M. In Search of... A better, faster stronger Web. http://assets.en.oreilly.com/1/event/29/Keynote%20Presentation%202.pdf, 2006. Accessed: 02.06.2017.

[24] Gilad Mishne, Jeff Dalton, Zhenghua Li, Aneesh Sharma, and Jimmy Lin. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pages 1147–1158, New York, NY, USA, 2013. ACM.

[25] NetMarketShare. Desktop search engine market share. https://www.netmarketshare.com/search-engine-market-share.aspx?qprid=4&qpcustomd=0&qpstick=0&qpsp=2016&qpnp=1&qptimeframe=Y, 2016. Accessed: 08.06.2017.

[26] Rudihagen J. A. R. Instant, personalized search recommandation, 2015.

[27] Spotify. On track: Discover weekly. https://community.spotify.com/t5/Community-Blog/On-Track-Discover-Weekly/ba-p/1456790, 2016. Accessed: 08.06.2017.

[28] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool, 2016.

[29] Bin Wang Yang Xu, Gareth J. F. Jones. Query dependent pseudo-relevance feedback based on wikipedia. 2009.