UNIVERSITY OF GOTHENBURG

Shard Selection in Distributed Collaborative Search Engines

A design, implementation and evaluation of shard selection in ElasticSearch

Master of Science Thesis in Computer Science

PER BERGLUND

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Shard Selection in Distributed Collaborative Search Engines
A design, implementation and evaluation of shard selection in ElasticSearch

Per Berglund

Examiner: Marina Papatriantafilou

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

To increase their scalability and reliability many search engines today are distributed systems. In a distributed search engine several nodes collaborate in handling the search operations. Usually each node is only responsible for one or a few parts of the index used for storing and searching. These smaller index parts are usually referred to as *shards*.

Lately ElasticSearch has emerged as a popular distributed search engine intended for medium- and large scale searching. An ElasticSearch cluster could potentially consist of a lot of nodes and shards. Sending a search query to all nodes and shards might result in high latency when the size of the cluster is large or when the nodes are far apart from each other. ElasticSearch provides some features for limiting the number of nodes which participate in each search query in special cases, but generally each query will be processed by all nodes and shards.

Shard selection is a method used to only forward queries to the shards which are estimated to be highly relevant to a query. In this thesis a shard selection plugin called SAFE has been developed for ElasticSearch. SAFE implements four state of the art shard selection algorithms and supports all current query types in ElasticSearch. The purpose of SAFE is to further increase the scalability of ElasticSearch by limiting the number of nodes which participate in each search query. The cost of using the plugin is that there might be a negative effect on the search results.

The purpose of this thesis has been to evaluate to which extent SAFE affects the search results in ElasticSearch. The four implemented algorithms have been compared in three different experiments using two different data sets. Two new metrics called Pk@N and Modified Recall have been developed for this thesis which measures the relative performance between exhaustive search and shard selection in a search engine like ElasticSearch.

The results indicate that three algorithms in SAFE perform very well when documents are distributed to shards depending on which linguistic topic they belong to. However if documents are randomly allocated to shards, which is the standard approach in ElasticSearch, then SAFE does not show any significant results and seems to be unusable.

This thesis shows that if a suitable document distribution policy is used and there is a tolerance for losing some relevant documents in the search results then a shard selection implementations like SAFE could be used to further increase the scalability of a distributed search engine, especially in a low resource environment.

Acknowledgements

This idea for this thesis came from Findwise AB, a search-oriented IT consultant company in Gothenburg. A special thanks to Karl Neyvaldt and Svetoslav Marinov at Findwise for technical and academic advice. I would also like to thank Nawar Alkurdi for her support and for helping me to get in touch with Findwise. Finally I would also like to thank my supervisor and examiner Marina Papatriantafilou at Chalmers/University of Göteborg for taking me on as a student and providing me with feedback and guidance.

Per Berglund, Gothenburg 2013-06-14

Contents

1

Introduction

Shard selection is an optimization for distributed search engines. The idea is that a query should only be processed by nodes which are likely to return relevant results. Shard selection is a well-researched problem to which many different solutions have emerged over the past fifteen years. This chapter is concerned with giving an understanding of why this problem exists and why it is important to research. It will also clarify the goal of this thesis give a summary of the results.

1.1 Historical background

When the Internet was introduced in the early 90s there were few who could predict the rapid growth of open information that would come. To enable users to find useful websites there was a server hosted by the European Organization for Nuclear Research (CERN)[1] which contained a list of available servers on the Internet [1]. Before long this centralized index became unfeasible to use for finding relevant servers for a user's need. The method of finding information on the Internet was revolutionized when the first search-engines appeared. By indexing web-pages from all over the world users could suddenly find relevant information by just providing names or other terms that they were interested in. As a result some of the most popular search-engines have grown into global multi-billion dollar companies.

Search engines have since become vital in many areas of information technology. They are used in all kinds of applications and organizations for navigating users, data-mining and statistics. Deploying a search engine for a specific purpose today is often quite easy

[1]CERN: Accelerating science, http://home.web.cern.ch/

and cheap since there are good open-source implementations and a growing knowledge-base. One server in the mid-range performance spectrum is now able to handle several gigabytes of data and hundreds of thousands of request without a problem.

Despite the increased capabilities of computers and networks there is still a need for increased scalability in search engines since information grows at a staggering rate. Many search engines today are distributed systems with several nodes collaborating in handling the work-load. A common approach is to split the index into smaller parts called *shards* and let each node be responsible for one or a few of them. By allowing new nodes to be added and new shards to be created the scalability of the search engine is secure. By replicating the shards to different nodes the reliability is also greatly enhanced since the system can handle node failures.

Although distributed search engines provide enhanced scalability and reliability they do not guarantee increased performance. The collaborating nodes might be far apart from each other resulting in network-latency. Another aspect is that the nodes might not have equal specifications, e.g. some nodes might have solid-state hard-drives while some may not. This sparked a new research-area in the mid-90s called *shard selection* with CORI [2] being the first successful algorithm. The idea is to identify a subset of the nodes which are most likely to have relevant information for each search request. By limiting the number of nodes which participate in each search request the scalability of distributed search engines can be further increased. Many algorithms have since emerged to tackle this problem depending on the architecture of the search engine and the data that they handle.

1.2 Research aim

1.2.1 Problem description

ElasticSearch [3] has recently emerged as a popular search engine. One of the main purposes of ElasticSearch is to provide an open source platform for large scale searching. To achieve this goal the search engine is distributed and allows several nodes to collaborate in handling the indexing and search operations. The search engines index is split into smaller parts called *shards* and each node is only responsible for one or a few shards.

When a user sends a query to an ElasticSearch cluster the results will not be returned before all targeted nodes have processed the query against their shards. In a best case scenario all nodes have equal performance, are put in the same location and are dedicated ElasticSearch servers, in which case the query-to-result time will not be affected by how many nodes process the search query. Usually this best case scenario cannot be achieved which result in higher latency when the number of nodes in the cluster increases.

To tackle this problem ElasticSearch provides some features to limit the number of nodes which process each search query. When a query is targeting a specific document ID or a specific document *type* only the nodes which contain these documents will be queried. A more detailed discussion about these features can be found in section 2.1.5. Since these features can only be used under special circumstances there are many cases in which all nodes will have to process a search query which sets a clear limitation to the scalability of ElasticSearch.

1.2.2 Main goals

The idea in this thesis is that a method called *shard selection* might solve the scalability issue in ElasticSearch described in the previous section. A plugin for ElasticSearch will provide the shard selection functionality and some of the most established shard selection algorithms will be implemented to the plugin.

The goal of this thesis is to evaluate what effect the shard selection plugin will have on the quality of the results from queries in ElasticSearch. This leads to the first research-question:

How is the quality of search-results affected by the different shard selection algorithms?

These results will indicate the usefulness of shard selection not only in ElasticSearch but also in other distributed and collaborative search engines. The main difference given the collaboration aspect is that some crucial statistics used in many algorithms are known by all servers and doesn't have to be estimated. This leads to the second research question:

Which shard selection algorithms are most suitable for shard selection in a collaborative distributed search engine?

1.2.3 Scope and limitation

Search engine is a commonly used name for *information retrieval (IR) systems* since such systems are searching for information rather than fetching information in a structured way as in a database. In order to give the reader a good understanding of the problem I will give an introduction to many concepts in information retrieval and some of the most established models for information retrieval systems. I will also give an overview of ElasticSearch but only the aspects which are relevant for shard selection. Except for these two subjects the report is only concerned with different aspects of shard selection.

To be able to focus on shard selection this thesis uses two assumptions. The first assumption is that the IR model used in ElasticSearch is sound and effective and that the implementation of the model in ElasticSearch is correct. This leads to the second assumption which states that all documents returned from ElasticSearch for a specific search query are relevant to that query. As discussed in section 2.1.2 this is usually not the case but by using this assumption the relative damage to the search results by shard selection can be evaluated.

Since the aim of this thesis is to evaluate shard selection in a cooperative distributed search engine (ElasticSearch) I will not go into much detail about the extra challenges facing shard selection in un-cooperative distributed search engines. Examples of un-cooperative distributed search engines are *meta search engines* which forwards a query to several different search engines which are unaware of each other.

2

Background

If you steal from one author it's plagiarism; if you steal from many it's research

Wilson Mizner

INFORMATION RETRIEVAL and search engines are both large areas for research. Everything from indexing time to the quality of the search results is important for the success of a search engine.

This chapter will start by introducing some on the key concepts that are related to shard selection to provide a basic understanding of how a search engine operates. The distributed search engine ElasticSearch will be used as a case study. Later the shard election problem will be discussed including detailed explanations of some of the most tested and established algorithms. Finally other aspects which could influence the performance of the algorithms will be discussed.

2.1 Introduction to Information Retrieval

The goal of information retrieval (IR) is to satisfy an information need from within a large collection of material [4]. There are many IR models which has been developed, and some of the most common ones include the *boolean model*, *vector space model* and *language model*. An *information retrieval* system is an implementation of an IR model. The following section will be general and is not dependent on which model is used.

Most people are probably more familiar with database systems when it comes to storing and finding information in computer science. To get started with IR it may be good to use database systems as a reference to some of the main concepts of IR systems. Some of the most common terms in database systems and their corresponding IR terms are listed in 2.1.

Even though most IR terms can be likened to database terms there are some major differences between the two concepts. A database consists of structured data. Queries to a database are also structured and the results from a query are data which are exact matches to the query.

In IR systems both the data and the queries are unstructured, usually consisting of natural language text. The retrieval method in IR systems is probabilistic meaning that data returned from a query are not exact matches. IR systems are said to be *searching* for their data and thus they are often referred to as *search engines* and this name will be used throughout this report.

Row, Tuple	Document
Column	Field
Table	Index
Database	Collection

Table 2.1: Common database terms and their corresponding IR terms

2.1.1 Data representation for efficient searching

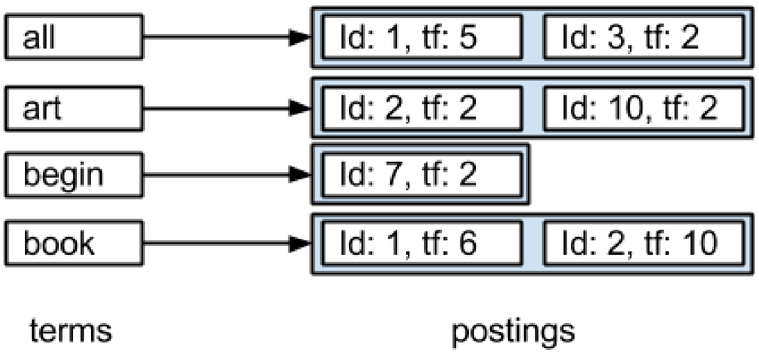

Figure 2.1: An inverted index with terms pointing to a list of positing containing document IDs and term-frequencies

In the basic boolean IR model a document is considered to be relevant to a query if they share at least one term. A naive approach to finding relevant documents for a query would then be to scan all documents to see if they contain at least one of the query-terms. This approach is obviously not scalable and would result in a very bad performance even when the collection of documents is relatively small.

Documents in a search engine are usually only scanned during the *indexing phase* [4]. When a document is added to the search engine for storage it is processed by a document *pipeline*, a series of steps which transforms the text of documents into indexable terms. The steps in the pipelines differ greatly depending on the structure and contents of the documents, but some variant of the following four steps are usually included:

1. Assign a unique ID to the document.
2. Split the text of the document into *tokens* by some rule, for example ”produce a new token each time a white space occurs in the text”. If this simple rules is applied on a document with the text $\boxed{\text{My name is Per}}$ it will produce the tokens $\boxed{\text{My}}$, $\boxed{\text{name}}$, $\boxed{\text{is}}$, $\boxed{\text{Per}}$.
3. Normalize the tokens into *indexing terms*. Normalization often includes applying a lower-case function on the letters in the tokens ($\boxed{\text{my}}$, $\boxed{\text{name}}$, $\boxed{\text{is}}$, $\boxed{\text{per}}$).
4. Remove *stop words* from the resulting terms. There is no formal definition of stop words but examples include ”this”, ”and”, ”or”. ($\boxed{\text{my}}$, $\boxed{\text{name}}$, $\boxed{\text{per}}$).

Each term will be added as a key to a multi-value *map* which is called an *inverted index* and can be seen in Figure 2.1. Each value (or *posting*) for a term in the index is the ID of a document which contains the term, together with statistics like the frequency of the term in that document. The inverted index has become the de-facto standard for representing documents in a search engine [4].

When a query is given to the search engine it is tokenized and normalized just like the documents. The terms in the query are then used as lookup-values in the inverted index. The posting-lists received from the look-ups are then used to sort the relevant documents depending on the scoring-function of the search engine.

A document structure is often more complex than the basic structure assumed here. A document is often split into *fields* with different attributes and data-types. Most documents include a ”text” field, but it’s also common to add fields for meta-data like an ”author” field and a ”date” field. A query may be applied to one or many of the fields. There are different approaches how to represent this extended inverted-index but one simple approach is to have a separate index for each document-field.

2.1.2 Document scoring

In the previous section a boolean relevance model was assumed, where a document is relevant if it contains one or more of the terms in a query. Reality is often more complicated than this. As an example, a document titled ”shard selection algorithm” contains the term ”algorithm”, but is it really relevant to the query ”Algorithm for path-finding”?

Determining if a document is relevant to a query is a fundamentally hard problem. The only way to really determine if a document is relevant to a query or not is for a user to judge it as relevant or not [4]. Most IR models assign a score to documents given a query, where a higher score is more likely to be judged as relevant by a user compared to a document with a lower score. Some of the most common scoring models will be described below.

The Tf-Idf scoring model assumes that a document/document-field with a high frequency of the query-terms is more relevant to the query [4]. Using only this criterion would however disproportionally discriminate against less common terms in a query. For example, in the query ”Computer Science Chalmers” the terms ”computer” and ”science” probably have a high term-frequency in many documents, but the term ”Chalmers” is probably the most important term since it is the most specific.

As a result, the tf-weights are combined with the *inverse document frequency* (idf) weights [4]. Df is calculated by counting the number of postings for a term in the inverted index, and the idf is calculated by dividing the total number of documents by this number as in equation 2.1.

$$Idf_t = log \times \frac{N}{Df_t} \tag{2.1}$$

The combination of tf-weight and idf-weights determines the score for a document given a query of terms [4] as is displayed in equation 2.2.

$$score(d,q) = \sum_{t \in q} Tf_{t,d} \times Idf_t \tag{2.2}$$

The vector-space model In the vector-space model documents and queries are represented as vectors of weighted terms [4]. The weights can be calculated in different manners, but a common approach is to use the Tf-idf weights described in the previous paragraph. Relevance between a document d and a query q is determined by the cosine of the angle between the vectors, often called *cosine similarity* and is calculated as in equation 2.3.

$$score(d,q) = \frac{\overrightarrow{V}_d \cdot \overrightarrow{V}_q}{|\overrightarrow{V}_d| \times |\overrightarrow{V}_q|} \tag{2.3}$$

2.1.3 Evaluating IR systems

Recall and *Precision* are two of the most common metrics used when evaluating IR systems [4]. Recall measures how many of all relevant documents in the collection that are returned from a query. More formally, let A be the set of documents that are relevant to a query and B be the set of documents that are retrieved. Then recall is calculated as in equation 2.4.

$$recall = \frac{|A \cap B|}{|A|} \tag{2.4}$$

Precision on the other hand measures how many of the returned documents from a query that are relevant. Precision is calculated as in equation 2.5.

$$precision = \frac{|A \cap B|}{|B|} \tag{2.5}$$

Note that achieving a high recall value is very easy; the system could return all documents in the collection for each query and it would result in a perfect score. However, this would result in a very low precision.

As mentioned in section 2.1.2 the only way to know if a document is relevant or not is to use relevance judgments from users. Both of these metrics require the data-sets used in evaluation to have such judgments.

2.1.4 Sharding of indices

When the number of indexed documents grows the posting lists for some terms might become quite extensive. Lookup-time for terms is a constant operation, but the time it takes to traverse the posting lists increases linearly, which eventually results in a lower throughput of the system. Performing various optimizations like defragmentation might help to maintain the performance in the short run. In the long run the best solution is to split the index into smaller parts. This process is called *sharding.* The *shards* may be distributed to different nodes which enables multiple computers to collaborate in the search-process.

In the context of databases, sharding refers to a horizontal partitioning of a table. In this scheme the *rows* of a table are split up rather than the *columns.* In this way each shard can stand on its own. One advantage of using this scheme is that the rows can be partitioned by one or more attributes, which mean that some of the shards can be filtered out in some queries.

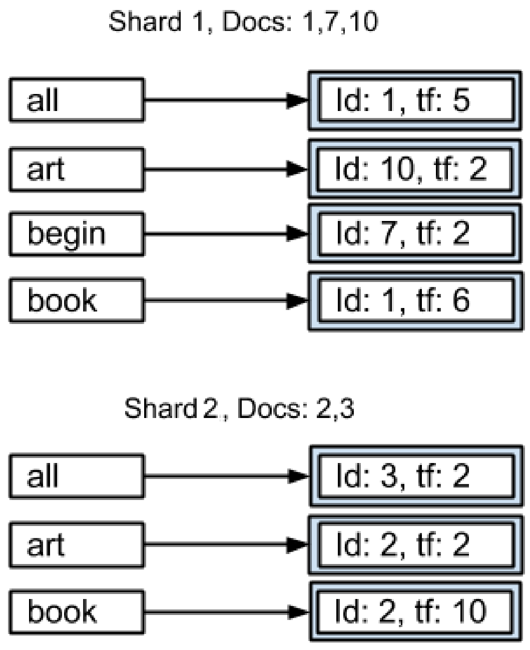

Figure 2.2: Illustration of the inverted index in Figure 2.1 split into two shards

When a search engine index is sharded the documents are partitioned rather than the terms (remember that documents may be likened to rows in a database). Since the scoring-function of some search engines depends on document-frequency of a term the results of a query may be affected.

2.1.5 Case study: ElasticSearch

EasticSearch [3] is an open-source distributed search engine under the Apache 2.0 license. It was built with big data in mind which has given it an emphasis on scalability and reliability. A running instance of ElasticSearch is called a *node* and together they form a *cluster* [5]. As the name implies it is very ”elastic” in that it automatically handles rebalancing of indices and shards when new nodes are added or removed from the cluster. As a result developers may add or remove nodes as a means to increase or reduce resources allocated for the cluster by demand.

Just like the popular search engine Solr [1] ElasticSearch uses Lucene [2] as a core library for indexing and scoring documents. Lucene supports several IR model but is shipped with an implementation of the vector-space model with tf-idf weights as discussed in earlier sections. Since Lucene has been used in and maintained by many different applications

[1]Solr: Ultra-fast Lucene-based Search Server, http://lucene.apache.org/solr/
[2]Lucene Core: Proven search capabilities, http://lucene.apache.org/core/

there is an implicit trust in the basic search capabilities of ElasticSearch. As a result the developers of ElasticSearch have been able to focus on usability, scalability and performance of the search engine.

Multiple indices ElasticSearch supports sharding of indices, but the number of shards of an index has to be set when the index is created and can thus not increase or decrease on demand [5]. To compensate for this ElasticSearch has support with multiple indices, a feature which distinguishes it from most other search engines.

Instead of increasing the number of shards when the number of documents grows it's recommended to construct a new index with the same *type*. A query may be forwarded to one or many indices which can be specified in the query. Thus, an index may also be referred to as a shard in ElasticSearch, if there are many indices with the same type.

From now on a *shard* in this report will refer to an *index* in ElasticSearch.

Query routing To enhance its scalability further ElasticSearch comes with a sophisticated system for routing queries to nodes. The simplest example is when ElasticSearch is used as a database, where the ID of documents to fetch is specified in a query. In this case only the nodes where the documents reside will process the query. Since ElasticSearch is mainly used for searching this feature only has a limited value.

There are other features which may be used for query-routing [5]. An index in ElasticSearch support different *types*. Each type may be assigned a specific routing-value. If a routing-value is assigned to a type the default policy is to cluster the documents of that type together in the same shard. If a type is specified in a query it will only be routed to the nodes which contain documents of that type.

In many cases the type of the documents requested are not known. As an example, assume we have an index containing documents representing *tweets* (twitter posts). Each user is assigned its own *type* with a unique routing-value and each tweet will be stores as the type of the user who posted it. As long as we are able to specify which user we want to search from in a query we can use ElasticSearch' built-in query-routing functionality. But in many cases we want to make a global search across all users, perhaps for a specific topic. As of the time this thesis is written all nodes containing at least one shard of the twitter-index will have to process the query for a global search. This is where shard-selection might come in handy for ElasticSearch.

2.2 Shard selection

Despite all the advantages of having a sharded index there are some drawbacks which have to be addressed. In a distributed search engine the bottleneck is often the slowest node, and this will determine the speed of delivering query-results to the users. Many factors could impact the performance of a node, like its hardware specification or location relative to the other nodes in the cluster. Even if the nodes are placed at the same location and are given the same hardware there might be network congestion when they receive many simultaneous requests. Especially in low-resource environment there is a need to limit the number of nodes which participate in each search query [6].

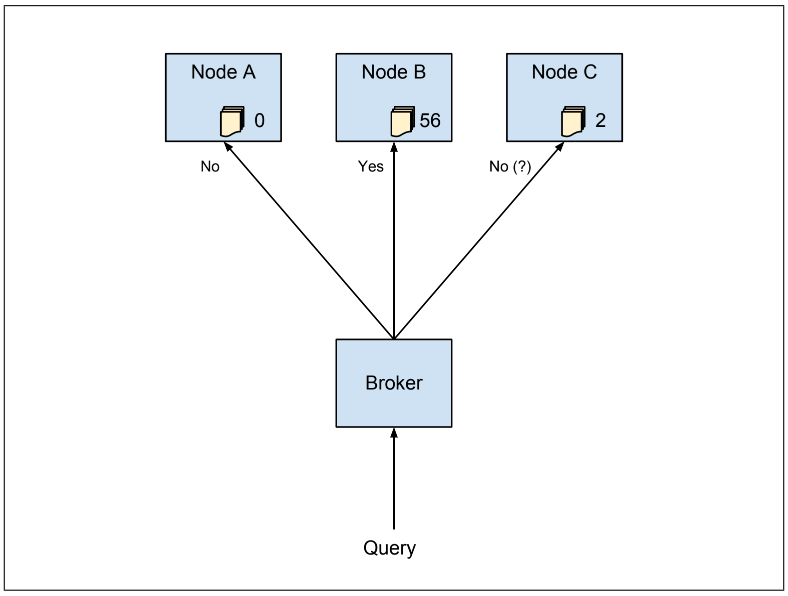

Figure 2.3: Illustration of the shard-selection problem: a query is sent to a broke node which routes it all data nodes, even nodes which do not contain any relevant documents.

In some cases only a few nodes contributes to the top-results for a query. An example is given in figure 2.3. A common approach is to have one shard on each data-node. A query is sent to a broker-node which will re-route the query to data-nodes. The data-nodes process the query and return documents which are determined as relevant to the query. The broker-node will combine the results and give it back to the user which sent the query.

In this example there are several relevant documents in node B but no relevant documents in node A. The broker-node could have ignored routing the query to node A to save resources without having any impact on search result. Since node C only holds a few relevant documents it could probably also be ignored but there is always a possibility that these documents would get a higher score than all documents in node B.

Predicting the impact a node will have on the result for a query is part of a problem called *shard selection*[3] [6]. A decision also has to be made if the predicted relevance

[3]Other common names for this problem are *collection-selection*, *resource-selection* and *server-selection*. In this report *shard selection* will be used since it makes most sense in a collaborative distributed search engine. .

is high enough for a query to be processed by the node. The goal is to save resources without having a big impact on the quality of the search-result.

The basic idea of all algorithms for shard-selection is that they collect information from the different nodes by investigating the information contained in the shards they hold. In an offline phase and use this information to make online relevance-judgments for shards given a query. Only one or some of the nodes collect this information (*broker-nodes*) and all queries from the user should handle to these nodes which will forward the query to the selected shards [7].

2.3 Lexicon algorithms for shard-selection

2.3.1 CORI

The *Collection Retrieval Inference Network* (CORI) was one of the first shard selection algorithms, and was part of the *InQuery* information-retrieval algorithm [8]. InQuery uses a probabilistic model for information retrieval, namely a Bayesian network. Although the algorithm has proven to be effective the field of Bayesian networks has evolved a lot since and the model used a lot of assumptions to estimate parameter values [4]. CORI can be seen as an extension of the InQuery algorithm but determines the similarity between a user query and shards, instead of individual documents. CORI calculates the similarity between a query q and a shard s as in equation 2.6.

$$CORI(q,s) = \frac{\sum_{t \in q \& s}(d_b + (1 - d_b) \times T_{s,t} \times I_{s,t})}{|q|} \tag{2.6}$$

$$T_{s,t} = d_t + (1 - d_t) \times \frac{log(f_{s,t} + 0.5)}{log(max_s + 1.0)} \tag{2.7}$$

$$I_{s,t} = \frac{log((N + 0.5)/f_t)}{log(N + 1.0)} \tag{2.8}$$

The value $T_{s,t}$ represents the weight of term t in shard s and $I_{s,t}$ is the inverse frequency of the term. The variables d_b and d_t are both set to 0.4 in many implementations, with the first value representing the *minimum belief component* and the latter represents the *minimum term frequency component.* The value max_s represents the number of documents in shard s which contain the most frequent term in the shard. Other parameters can be found in table 2.2.

CORI was long used as a benchmark for shard selection [9] [10] [11] but has since been demonstrated by D'Souza et al [12] to be problematic. CORI suffers from a lot of assumptions and hard coded values, just like InQuery. The variables d_b and d_t appears to be highly sensitive to variations in the data-sets [12] the optimal values of these variables are not easily obtained.

2.3.2 HighSim

D'Souza et al [9] investigated a range of lexicon algorithms for shard selection, out of which *HighSim* showed the best performance. The algorithm is based on an optimistic assumption that all terms from a query found in a shard can be found in a single document in that shard. The lexicon which is produced by the algorithm includes all indexed terms in all shards. For each term t, the total term frequency over all shard f_t and the frequency of the term in each shard $F_{c,t}$ is stored as statistics. The formula for scoring shard c given query q is as follows:

$$\text{HighSim}(q,c) = \frac{\sum_{t \in q\&c} w_{q,t} \times w_{c,t}}{W_c}$$

where w_t is the weight of term t across all shard, $w_{q,t}$ is the weight of term t in query q, $w_{c,t}$ is the weight of term t in shard c and W_c is the average number of terms in each document in shard c. A list of all the parameters can be found in Table 2.2.

N	Total number of shards
N_s	Nr of documents in shard s
f_t	Nr of occurrences of term t across all shards
$f_{q,t}$	Nr of occurrences of term t in query q
$F_{s,t}$	Nr of occurrences of term t in shard s
w_t	$log(N/f_t + 1)$
$w_{q,t}$	$w_t \times log(f_{q,t} + 1)$
$w_{s,t}$	$w_t \times log(F_{s,t} + 1)$
W_s	$\sqrt{(\sum_{t \in s} F_{s,t})/N_s)}$

Table 2.2: Parameters used in lexicon algorithms.

2.4 Surrogate algorithms for shard selection

Surrogate algorithms were originally developed to work in un-cooperative distributed search engines [9] [10]. The algorithms collects *documents surrogates* from all collections (uncooperative) or shards (cooperative). The surrogate documents can either be *partial* documents or *sample* documents. In the first case all documents are collected from all shards but only a subset of the terms are retained. In the latter case the documents are complete but only a subset of all documents is collected from each shard. The surrogate documents are then indexed together with the ID of the shard they were collected from in a central index at each broker node. When a user sends a query to a broker node it is first processed against this central index. The resulting documents scores are used to infer shard ranking [9] [10] [6].

2.4.1 Best-N algorithm

D'Souza et al [9] proposed a method called the *Best-N* algorithm. For each document in each shard, the goodness for each term is calculated as in equation 2.9. The n terms with the highest goodness in each document is stored together with the ID of the document and the ID of the shard the document is stored in. Global statistics should be used in the goodness formula for accuracy.

$$goodness(t) = log(1 + f_t) \times log(\frac{\sum_{s \in S} N_s}{f_t}) \tag{2.9}$$

The partial documents fetched from each shard are indexed at a centralized index at each broker node. When a query is sent from a user to the broker nodes the query is first processed on the centralized index. The result of the querying the centralized index is the top document scores together with the ID of the shard they belong to. D'Souza et al investigated several methods for converting the document scores into scores for shards. The most successful was the InvRank scoring method displayed in equation 2.10.

$$InvRank(q,c) = \sum_{d \in S_S} \frac{1}{r_d + K} \tag{2.10}$$

As for the value n the authors found that a value between 20 and 40 produced good results in which case the size of the centralized index is roughly the same size as the data structures used for lexicon methods, for example HighSim [9]. The value k in equation 2.10 is arbitrarily set to 10, and there appears to be no further research evaluating these values.

Although the best-n algorithms showed promising results in a variety of data-sets, there should be a problem if the data-sets contain many but small documents, for example twitter posts. In this case, the algorithm don not scale very well, since many twitter posts might do not contain more than 20 terms. If n is set to 40 we might end up with a centralized index which has roughly the same size as the combination of all the shards.

2.4.2 ReDDe

The *Relevant Document Distribution Estimation Method for Resource Selection* (ReDDE) [10] was developed for un-cooperative environments and has been used as a benchmark-algorithm in a wide range of studies [13] [14]. The algorithm uses a centralized index at the broker containing sampled documents from all of the available shards. The algorithms rank shards according to how many documents they are estimated to contain that are relevant to a query.

In case the centralized index is complete (containing all documents from all shards), the number of documents relevant to query q in a shards document-collection S_i is estimated as in equation 2.11. $P(Rel|d)$ is the estimated probability of relevance for document d to query q and $P(d|S_i)$ is the prior probability of document d in shard S_i [14].

$$Rel(S_i,q) = \sum_{d \in c_i} P(Rel|d) \times P(d|S_i) \times |S_i| \tag{2.11}$$

Since using a complete centralized index is unfeasible, even in a cooperative environment, ReDDE regards sampled documents as representative [10]. The above equation can therefore be approximated using equation 2.12. The value S_i_sampl is the set of sample documents from a shard S_i. The assumption is that for every relevant document in the sample from a shard, there are about $\frac{|S_i|}{|S_i\_sampl|}$ relevant documents in the complete shard.

$$Rel(S_i, q) \approx \sum_{d \in S_i\_sampl} P(Rel|d) \times \frac{|S_i|}{|S_i\_sampl|} \tag{2.12}$$

In ReDDE the probability of relevance for a document $P(Rel|d)$ is defined as the probability of relevance given the rank of document d in the centralized complete index (CCI). Since this CCI is not available, the central rank for a document has to be approximated using the rank of the document in the sampled index as in equation 2.13.

$$rank\_central(d_i) = \sum_{rank\_samp(d_j) < rank\_samp(d_i)} \frac{|S_j|}{|S_j\_sampl|} \tag{2.13}$$

After the centralized sample index has been queried and the centralized complete index rank has been approximated, the probability of relevance for document d is estimated by equation 2.14. Finally, the estimated relevance of shard S to the query q can be found with equation 2.15. After ranking shards by their goodness-score, ReDDE selects the top k shards, where k is a pre-defined value.

$$P(Rel|d) = \begin{cases} \alpha & \text{if } rank\_central(d) < \beta \times \sum |S_i| \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

$$goodness(S_i, q) = \frac{Rel(S_i, q)}{\sum_j R(S_j,q)} \tag{2.15}$$

2.4.3 Sushi

The *Scoring Scaled Samples for Server Selection* (Sushi) [13] algorithm is one of the most recent contribution to the surrogate family of algorithms. Just like ReDDe [10] it uses a centralized sample index to rank shards, but the ranking formula is a bit more complicated. Most shard selection algorithms try to achieve a high recall-value, but Sushi is mainly concerned with achieving a high precision-value. By focusing on precision the algorithm is able to automatically determine how many shards to select for each query, in contrast to most other algorithms.

The first step of the algorithm is to process the query on the centralized sample index. Only the top 50 documents returned from the index are retained for further evaluation. These documents are sorted into distinct sets according to their shard membership. The next two steps, *rank adjustment* and *curve fitting* are performed on each of these sets of documents.

Rank adjustment The ranks of all documents are adjusted according to the ratio between the size of the shard they belong to $|S|$ and the size of the sample from that shard $|S_sampl|$ as in equation 2.16. The goal is to estimate the rank each document would have in the non-existing centralized complete index [13]. As an example, assume we have a document d with rank 2 from the centralized sample index which is sampled from shard S_a and that the size of the sample is one tenth of the complete shard. Then the adjusted rank for document will be 20 since each sampled document represents 10 documents in the complete shard. If a very small number of documents get a score above zero from a shard (less than 5 in [13]) the ranks are not adjusted.

$$rank\_adjusted(d) = (rank\_sample(d) + 0.5) \times \frac{|c|}{|S_c|} \tag{2.16}$$

Curve-fitting To predict the scores for documents not present in the centralized sample index Sushi performs curve-fitting over the adjusted sample rankings using linear

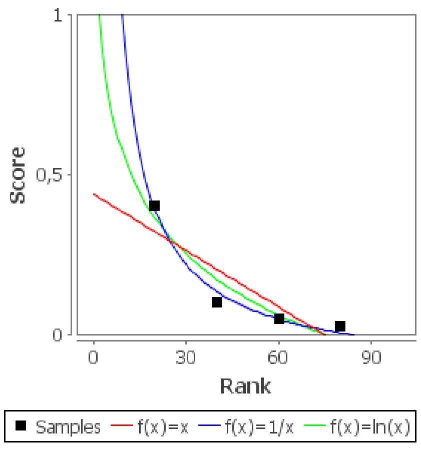

Figure 2.4: Curves produced by the three mapping functions in Sushi. In this example the exponential mapping function has produced the best fit.

regression as in equation 2.17. The mapping function $f()$ changes the distribution of the ranks of the samples [15].

$$Score(d) = k + f(rank(d)) \times m \tag{2.17}$$

The linear, logarithmic and exponential mapping function in table 2.3 are tried which produces three different curves as in figure 2.4. Sushi picks the curve with the best fit, which is measured by the highest *coefficient of determination* (R^2).

Curve	Mapping function
Linear	$f(x) = x$
Logarithmic	$f(x) = log(x)$
Exp	$f(x) = 1/x$

Table 2.3: Mapping functions for curve-fitting in Sushi

By using the selected curve the score for unseen documents in a shard can be predicted. The score for the top m documents from each shard are interpolated from the curve and are added to a sorted list. The top m documents from this merged list is then selected, and the query is only forwarded to shards which holds at least one of those documents. The value m was set to 10 in the original paper since users are rarely interested in documents with a lower rank [13].

2.4.4 Sampling-Based Hierarchical Relevance Estimation (SHiRE)

In a recent article Kulkarni et al [6] published three new algorithms which also utilizes a centralized sample index (CSI). Like SUSHI [13] the algorithms use a dynamic cutoff-

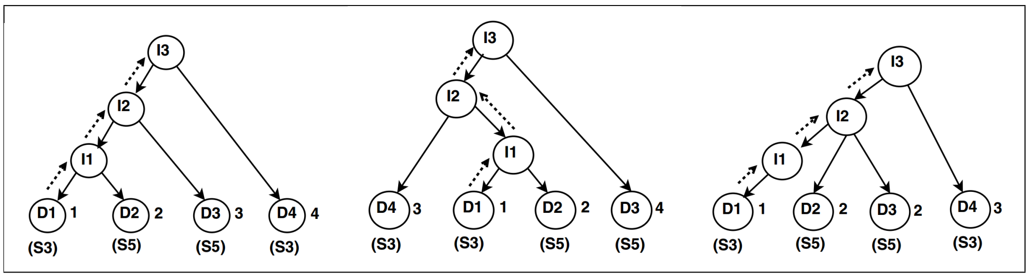

Figure 2.5: Toy examples of the SHiRE hierarchies. From left to right; Ranked, Lexicon and Connected. Figures taken from [6]

value and are able to automatically determine how many shards should be selected for a query.

As described in the previous sections, a query is first given to the CSI which returns a ranked list of the relevant documents together with references to the shards they belong to. The authors conclude that since the CSI is typically very small compared to the combined size of the original shards, more information besides this ranking might be needed to make accurate decision about which collections are most relevant to a query. By transforming the flat document-ranking received when querying the CSI into tree-like hierarchies, relationships between the documents can be found which might result in better shard-ranking [6].

The authors present three such hierarchies, displayed in figure 2.5. Shard ranking is inferred by traversing a hierarchy bottom-up starting at the highest-ranking document. When a document is found in a hierarchy, it may cast a vote for the shard it was fetched from.

$$Vote(d) = S \times B^{-U} \tag{2.18}$$

The value of a vote from a document to its shard is given in equation 2.18 where S is the score of the document from the CSI ranking, B is an exponential base and U is the level at which the document was found in the hierarchy. A range of values for B was investigated, and stable results are seen when the value is set between 20 and 50.

The final score for a collection is the sum of all the votes it received while traversing the tree-hierarchy. A collection is cut-off from the search if its final score converges to zero, which interpreted as ≤ 0.0001 by the authors [6].

Lexicon SHiRE (lex-s) uses the lexical similarity between sample documents to construct a hierarchy. The similarity between documents is determined by the manhattan distance of the documents tf-idf vectors. Each document is placed in each own cluster, and the clusters are bound together into a hierarchy by an agglomerative clustering algorithm.

Connected SHiRE (conn-s) uses the shard-membership of the documents to construct the hierarchy. The documents are added to the hierarchy bottom-up, starting at the top-ranked document. As long as documents are added with the same shard-membership as the previous document, it is placed at the same level as the previous document. If the new document and the previous document belong to different shards the new document is added at a new level in the hierarchy.

Ranked SHiRE (rank-s) is perhaps the simplest hierarchy in the SHiRE family. The hierarchy is a left-branching binary tree built bottom-up starting with the highest-ranked document. Each document is added at its own level in the hierarchy. The voting system on the other hand is a little bit more complicated compared to lex-s and conn-s. Since the document with the highest rank votes first, and the value of a vote is exponentially decaying, one of two criteria has to be met for the top document to cast its vote [6]. The first criteria is that one of the first m documents have to vote for the same collection as the top document. The second is that at least 10% of the documents at the 30 first levels has to give their vote to the same collection as the top document. As a side note, the value m is not determined in the paper.

Out of the three algorithms presented by Kalkurni et al, lex-s performed slightly better than the other algorithms, but rank-s was more efficient (selecting fewer shards) while still having a comparable performance to ReDDE [6].

2.5 Other approaches for shard selection

2.5.1 Shard and query clustering

Puppin et al [16] proposed a new design for web-based search engines which centers around shard selection, instead of using it as a possible extension. The design uses a shard-selection friendly document allocation policy by default, minimizing the need to re-design the index structure if shard selection is used. The authors utilize the fact that there are many query-logs publicly available for web-bases search engines. The authors construct a matrix with rows representing queries and columns representing documents. Each entry in the matrix is the score that documents received from sending a query to a reference search engine. When all the queries have been sent to the reference search engine they use a co-clustering algorithms on the matrix which re-orders the matrix so that relevant documents are close together and relevant queries are close together. This PCAP matrix [16] is used both when documents are allocated to shards and to perform shard-selection when a query is sent to the search engine.

2.5.2 Highly discriminative keys

Booking & Heimstra [7] tested a method which is aimed for search engines with a high level of collaboration. The main idea is that terms and phrases which are *highly discriminative* are most suitable to use for shard selection. The first step in the algorithm is called *peer-indexing* which is an iterative process performed on each shard. A highly discriminative key (HDK) is either a term or a phrase which occurs less than n times in the shard.

The HKD's produced from each shard in the peer-indexing step are then sent to the broker nodes in the cluster. The broker nodes will only store the HDKs which are infrequent in at most m shards. When a query is sent to a broker it will be matched against the stored HDKs, and query will be forwarded to the shards which reported the matching HDKs. The results from using HDK's for shard selection were comparable to using the language based algorithm *Indri* [7].

2.6 Document allocation policies for shard-selection

There are many policies for distributing documents between shards. Allocation policies has a big impact on the usefulness of the shard selection algorithms, since they all depend on some shards being more relevant to a query than others. The *clustering hypothesis* states that "'documents with similar content are likely to be relevant to the same information need"'[17] [18] which implies that shards should contain documents which are similar by some aspect.

2.6.1 Random allocation

The most common document-allocation policy is to randomly distribute the documents to shards. It can be implemented by taking a hash on the documents IDs and use the result as the ID of the shards they should be allocated to. This policy guarantees a balanced size between shards and is used by many of the leading search providers like Google [17].

If the random distribution policy is employed the usefulness of the shard selection algorithms will be minimized. This has been observed in many articles [9][13][10][18].

2.6.2 Attribute-based allocation

A simple yet effective policy is to distribute the documents depending on some attribute available in their meta-data. Callan & Kulkarni investigated a source-based allocation policy in [18] where documents from a web-based data set were grouped and sorted based on their URL. Each shard was allocated a group if N/M documents where N is the total number of documents and M is the number of shards available which guarantees the shards to be balanced in size. A similar policy was investigated in [7] but instead of grouping by the documents URL the IP of the server the documents were fetched from was used. Source-based allocation has proven to one of the most effective policies for shard selection while still being relatively easy to implement [18].

2.6.3 Topic-based allocation

This policy states that documents which belong to the same topic should be allocated to the same shard. Determining a topic for a document is a hard problem, especially when a topic is not provided in the documents meta-data. When topics are not provided, different methods may be used to group documents into topics. In [17] documents were grouped by their lexical similarity.

Instead of relying on the provided topics in documents, documents are divided into topics by a clustering-algorithm, for example the K-means algorithm. The lexical similarity between documents is used as the clustering similarity-metric [17].

2.6.4 Time-based data flow

The previous policies discussed assume that much of the data that should be indexed are available when the index is created. This is not always the case as many search engines continually adds documents to their index, sometimes with very few documents to start with.

This approach could be seen as both a data design-pattern and a policy. The data design-pattern states that new documents are continually flowing in for indexing as they are being created. The policy states that a new document should be added to the most recently created shard (or index in the case of ElasticSearch). It's assumed that new shards may be created on demand. This could be after a fixed period of time or when the size of the most recent shard has reached some threshold [19].

In many cases the content of documents are highly dependent on the date that they were created. Twitter is a good example where interest in trending topics usually fades within a couple of days or even hours. In such cases the shards will be *naturally clustered* eliminating the need to re-allocate documents to enhance the clustering attribute of the shards. At the time of writing this report, this design-pattern and policy has not been tested for shard selection.

2.7 Evaluating shard selection

The Precision and Recall metrics mentioned in section 2.1.3 are also frequently used to evaluate shard selection [20]. A variant of precision called P@N has almost become a standard metric in recent articles [6] [13] [7]. In this variant only the first N documents are considered when measuring precision. The justification of this metric is that users are often mostly concerned with the top results from a query [4].

Rk(n) is a special recall-metric which has been used in some articles about shard selection [13] [10]. Rk only measures the effectiveness of the shard selection algorithms without being concerned about the effectiveness of the underlying IR system. Rk is defined as

$$Rk = \frac{\sum_{i=1}^{k} \Omega_i}{\sum_{i=1}^{k} O_i}$$

where Ω_i is the number of relevant documents in shard i selected by the algorithm and O_i is the number of relevant document in shard i selected from an optimal baseline.

A conjuncture of P@N and one of the recall measurements are often enough to indicate the performance of the algorithms.

3

Shard Selection Algorithms Extension for ElasticSearch

This chapter presents SAFE, a shard selection plugin for ElasticSearch which has been developed for this thesis. This chapter starts out with presenting the plugin and the motivation behind its design. Four shard selection algorithms were implemented to the plugin with some minor modifications to make them compatible with ElasticSearch.

3.1 Work-flow

ElasticSearch [3] provides quite a good API-guide on its website, but since there are so many different configurations and query-types available it can be hard to get a good overview of the search engine. I took the *learning by doing* approach and started to experiment with setting up clusters and examining the results from different queries.

When I had developed a good understanding of the problem and how the various algorithms worked I had to figure out how to implement them in ElasticSearch. It should be said that ElasticSearch is a huge piece of software and most of the classes and their methods are undocumented. Luckily there are other plugins for the search-engine available online, some which require functionality at the same level as the shard-selection plug-in. I investigated some of these plug-ins as a way to bootstrap my own work.

Along the way there have been many problems while implementing the plugin. I wanted the plugin to be dynamic and work with the most common configurations of ElasticSearch. I had to avoid imposing special requirements like storing the original documents on the nodes. As it turns out I was able to provide workarounds to most problems, in some cases by investigating the source-code and discussing what modifications could be acceptable with my supervisor at Findwise. I was also lucky that a new version of ElasticSearch was released during the project which made accessing some of the statistics needed by the algorithms multiple times faster.

3.2 SAFE

The *Shard Selection Algorithms Extension for ElasticSearch* (SAFE) was developed for this thesis. SAFE is a plugin for ElasticSeach [3] which gives the search engine shard selection functionality. ElasticSearch has been architecture with Google Guice[1], a dependency injection framework. The framework has allowed SAFE to override and extend many core modules in ElasticSearch without having to modify the source code of the search engine. The following sections will give an overview SAFE and how it operates. A more technical reference manual which assumes a basic knowledge of ElasticSearch has been included in appendix B.

3.2.1 SAFE Requirements

There were many decision which had to be made regarding the design of SAFE. The list of core requirements below was used to guide the development. The list may be studied to get a quick overview of the problems which faced the development of SAFE and how it is supposed to be used.

1. SAFE shall support algorithms from the lexicon and surrogate paradigms.
2. SAFE shall support all current queries in ElasticSearch.
3. SAFE shall use the same scoring model for the centralized sample index as is used for normal indices in ElasticSearch.
4. SAFE shall support query-requests from both http clients and java clients.
5. SAFE should require minimal modifications to work with future versions of ElasticSearch.
6. The centralized sample index in SAFE shall use the same scoring model as normal indices in ElasticSearch.

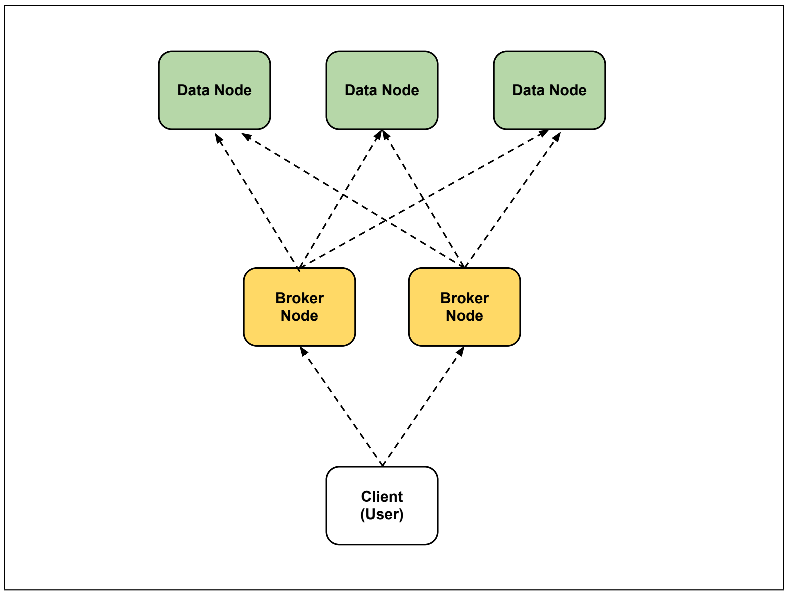

Figure 3.1: The suggested cluster design when SAFE used; A search request is sent to a broker nodes which forwards the query to data nodes.

3.2.2 Cluster configuration

Nodes which have have least one index shard in ElasticSearch are called *data nodes*. When there are a lot of data-nodes collaborating in a cluster a common design is to have a couple of *client nodes* in front of the data nodes [5]. The purpose of the client nodes is to alleviate the data nodes from handling computations except for indexing and searching. Client nodes handle all the http traffic, query parsing, query routing and merging of results.

Client nodes should make an excellent candidate for acting as *broker nodes*, a concept suggested in many previous articles about shard selection [16][15][7]. By adding shard selection functionality to the client nodes and transforming them into broker nodes the broker nodes serves two purposes. This also enables SAFE can work with many present configurations of ElasticSearch since there is no need to make a big re-design if there are already client nodes in the cluster.

The plugin has to be installed on all nodes, no matter if they are configured to be broker nodes or data nodes. This is because the broker nodes and data nodes collaborate in constructing the data used for shard-selection. Installation and configuration of the plugin has to be performed before the cluster is started.

The two main operations of the plugin are the *refresh* and *select* operations. The first is

[1]Google Guice: lightweight dependency injection framework for Java, https://code.google.com/p/google-guice/.

a collaboration between all the client nodes and the broker nodes, but the latter is only handled by the broker nodes.

3.2.3 Implemented algorithms

Four algorithms was implemented in SAFE; HighSim, ReDDe, Rank-S and Sushi. Out of these only HighSim belongs to the lexicon family.

Selecting which algorithms to implement proved to be a tough problem. As discussed in the section 2.2 there are many different approaches for shard-selection and many algorithms have proven to be effective in different studies. The decision came down to the results the algorithms has shown in the most recent studies, and also how suitable they would be for implementation in ElasticSearch. Some of the most promising results have been shown by the lexicon-type algorithms, but there has been very little research on this family of algorithms in the past decade. One reason is probably that most of the recent papers assume an un-cooperative environment where surrogate algorithms are clearly more suitable to use.

The most recent lexicon-type algorithms were the ones presented in [9] and out of these HighSim proved to be the most effective which led to the decision to implement it. The same paper also presented the only surrogate-algorithm which assumes a fully cooperative environment to date, the N-best algorithm. It would be interesting to compare this surrogate approach to the sample type algorithms like ReDDe. The main reason for not implementing N-best was that it do not scale as well as the other surrogate algorithms, especially with the data set used in the experiment which consists of tweets which are usually very short documents. The centralized index for N-best would be almost as large as if the index shards were merged into one big index.

The surrogate algorithms all share a common centralized sample index. The CSI is implemented as a Lucene index to mimic the implementation of the shards in ElasticSearch. By using a lucene-index the scoring of documents in the CSI will reflect how they would've been scored by ElasticSearch. Though the fraction of documents to sample from each shard can be set dynamically in the plugin this value was set to 4% in the experiments.

3.2.4 Refresh operation

A broker node cannot perform shard selection before the refresh operation has been called. In fact, no data associated with shard selection will be allocated if this operation is never called, meaning that the plugin can safely be installed on a cluster without taking any resources if the operation is never called.

The refresh operation is visualized in Figure 3.2. The operation starts at a broker node, which broadcasts the request to all the data-nodes. The data nodes collect the needed

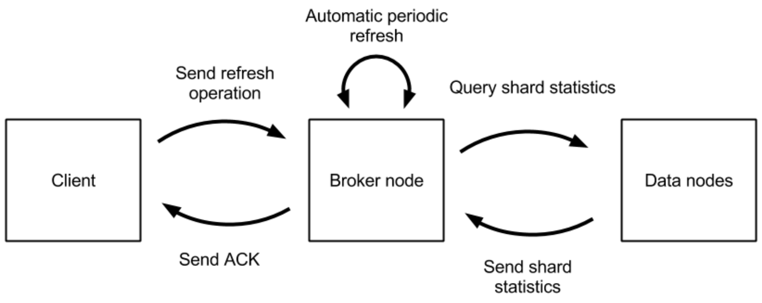

Figure 3.2: Visualization of the refresh operation in SAFE.

data and return it to the broker node. The broker node will collect the data and construct the needed data-structures for shard selection. Note that the refresh operation has to be called on all broker nodes which are intended to utilize shard selection since the shard selection data is stored locally and in-memory on a broker node.

The refresh operation is an offline operation since it only has to be called once in order for shard selection to work. The operation may take some time depending on the number of shards and the number of documents they contain. Since the operation is asynchronous other operations like searching can be performed while the refresh operation is running.

The refresh operation gathers different kinds of data depending on which algorithms are intended to use for shard selection. For the HighSim lexicon-algorithm terms and terms-statistics are gathered from the data-nodes. For the surrogate-algorithms sample documents are gathered. If all algorithms are to be used, the refresh operation has to be called twice, once with ”lexicon” as the type-parameter and once with ”surrogate” as a type-parameter. A more detailed description can be found in appendix B.

There are two ways to trigger a refresh-operation. The first way is to manually send a refresh request to a broker node, though this is mainly intended for testing. The other way is to configure the broker nodes to automatically call the refresh operation. If a broker node is configured to automatically trigger the refresh operation first call will be when the node joins the cluster. When a new node enters or leaves the cluster the data will be refreshed again. After this initial refresh a new refresh operation will be triggered when a configured time interval has passed, which is set to ten minutes by default. This may be useful when new documents are continually being added to the shards, as when the time-based data-flow document allocation policy is used as described in section 2.6.4.

3.2.5 Shard selection operation

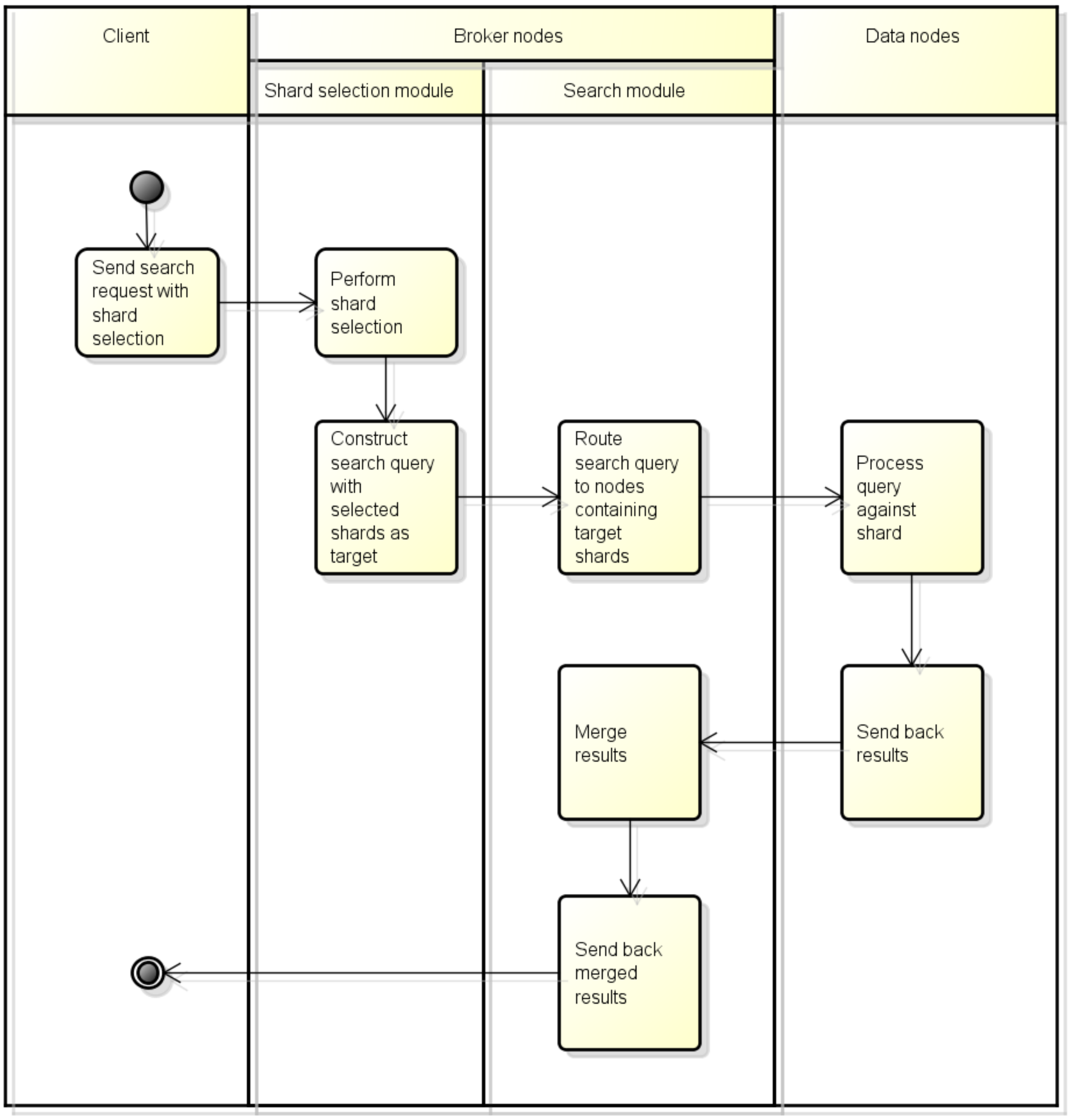
Client
Broker nodes
Shard selection module
Search module
Data nodes
Send search request with shard selection
Perform shard selection
Construct search query with selected shards as target
Route search query to nodes containing target shards
Process query against shard
Send back results
Merge results
Send back merged results

Figure 3.3: Activity diagram visualizing how a search operation is performed when SAFE is used.

When a broker node has performed the refresh operation it is ready to handle shard selection operations. A search query with shard selection includes both information needed to perform shard selection and the search query which will be performed on the shards which were selected. A detailed example of how a shard selection search query looks can be found in Appendix B.

An overview of how such an operation is handled can be seen in figure 3.3. When a shard selection query is sent to a broker node (which has performed the refresh operation) it will first be handled by the shard selection module implemented by SAFE. Here the actual shard selection will be performed. A regular search query will then be constructed using the information in the shard selection query. The selected shards will be set as the target for the search query, which is then forwarded to the standard search module in ElasticSearch. The search module keeps a routing-table for all shards in the system so ElasticSearch will automatically only forward the query to the nodes which contain the targeted shards. The data nodes which receive the query will then process it against its shards and send back the relevant documents. The search module received these documents, merges them and sends them back to the shard selection module which will add shard selection information to the results (how many shards were selected, how long time the shard selection operation took etc.).

In other words, SAFE can be seen as a ”man in the middle” between the user which sends the query and the search module which handles the search operation. There are many benefits to this approach; when a new search query type is added to ElasticSearch none or minimal modification has to be made to SAFE since the actual query will be handled by ElasticSearch itself. Also SAFE does not have to bother with keeping its own routing table to all shards and nodes, which would be prone to errors.

When the algorithms show low confidence in their results all shards will be selected. This fallback option is probably more appreciated than sending a query to random shards which could otherwise be the case. For the surrogate algorithms the fallback option will be used when there are less than five hits in the centralized sample index. For HighSim the fallback option will be used when no shards get a score above zero, but this should happen very rarely since the complete lexical information from all shards is available to it.

3.2.6 Handling of constraints presented by ElasticSearch and Lucene

Most algorithms for shard selections assume a very simple underlying IR system where all queries are targeting one big field containing all the information in the document. Most IR systems use a more sophisticated indexing structure with many fields. A query may target one or multiple of these fields. Here I will explain how some of the constraints presented by ElasticSearch and lucene have been handled for the algorithms to fit with these constraints and the needs of the users.

Lexicon algorithms The tricky statistic for the lexicon algorithms is terms, term-frequencies and the number of documents which contain at least one occurrence of each term. Since these statistics are also used by Lucene internally for its vector space model they can easily be obtained from each shard in ElasticSearch. Most of the lexicon algorithms (like HighSim) assumes that there is only one field which is searchable, but in Lucene a document is represented as a set of separate *fields* which can be included or excluded from a search operation. Another problem is that the fields can be indexed using different *analyzers*, which also means that a query should use different analyzers depending on which field a search is performed on. The terms which be retrieved from the Lucene index are already analyzed.

With these constraints in mind, the following information has to be considered when performing a refresh operation for the lexicon algorithms:

- The fields which the statistics should be gathered from has to be specified in the request-request.
- These fields should use the same analyzer when indexing.

Surrogate algorithms For each field in a Lucene-index there is an option to store the original content of the documents. This option defaults to *false* and most of the time this is an appropriate option since the only data needed to be stored is the identifier of the documents which can later be used to fetch the original document from a separate database.

- The fields which should be used in shard selection has to be set to *stored=true* in the ElasticSearch mapping.
- The fields which should be used in shard selection have to be included in the request.

4

Experiment

THE THREE experiments conducted in this thesis will be presented in this chapter. Two data sets were constructed using twitter data and both data sets were used in all three experiments.

4.1 Data sets

Many previous articles on shard selection has used one or more of the TREC-databases as data-sets for evaluation [13][10][6][12]. In order to get any significant results from shard selection the data-sets have to be clustered into separate collections by different factors so the collections are somewhat coherent. A range of factors has been used for clustering like source [7], date of creation [13] and topic [6].

Today there is lots of public information available from many sources. Some of the big player like Wikipedia[1], Twitter[2] and Facebook[3] offers access to their data for free with varying limitations. Twitter allows its users to gather newly posted tweets using their public stream-API[4]. There are several limitations when the free version is used, but it is still possible to gather millions of tweets each day. Twitter's streaming API has an option to use one or several *filter-terms* to filter out the content of the tweets that should be gathered. If a tweet doesn't contain at least one of the filter-terms it won't be received through the streaming API.

[1]www.wikipedia.org
[2] www.twitter.com
[3]www.facebook.com
[4]https://dev.twitter.com/docs/streaming-apis

One of the requirements for shard selection to be successful is that the documents are clustered as discussed in section 2.6. A topical clustering should be achievable by gathering collections of tweets using different term-filters. To test this out ten different collections of tweets were gathered, each with its own term-filter. Each term-filter contained common terms used in a specific topic, for example ”health”, ”business”, and ”politics”. The full list of topics and their corresponding term-list can be found in appendix A.

A common feature in Twitter is the ability to send copies of other tweets, so called *retweets*. Turns out that a large portion of the tweets gather initially were retweets. This could possibly distort the results for the surrogate algorithms since the samples would be less descriptive. In the end, retweets were filtered out from the gathering process.

Tweets from languages other than English were also filtered out though there are limitations to twitter ability to perform this filtering. A twitter user might set English as their default language but occasionally send tweets in other languages in which case the language filtering fails. Inspections on the tweets indicate that tweets from another language were quite rare in the final collections.

The collection of tweets was used to construct two data-sets. In the first data-set one shard was constructed from each collection of tweets. This data-set will be referred to as the *topical* data-set. The other data-set was constructed by indexing the tweets randomly to one of ten shards. This data-set will be referred to as the *random* data-set.

A set of queries was also constructed for the experiments, three queries especially aimed at each topic resulting in a total of 30 queries.

4.2 Experimental setup

The cluster used in the experiment consisted of ten data nodes and one broker node. Each of the data nodes held one shard each. All communication between the test application and the cluster was handled through the broker node which was also responsible for the handling shard selection.

Each experiment was run five times so the results in section 5.1 are the average results. The reason for running each experiment five times is that the centralized sample index consists of random documents from each shards, and the results may vary depending on which documents were used as samples.

Each run for an experiment consisted of nine iterations. All queries were sent to the broker node on each iteration for all algorithms. On each iteration the number of nodes the algorithms was allowed to select increased by one.

4.3 Metrics

As discussed in section 2.1.2 determining if a document is relevant or not to a query is a hard problem. Most data-sets used in previous articles about shard selection provide relevance judgments which have been collected from test involving humans judging the relevance of the results for a query. The data-sets used in this thesis has no such relevance judgments, which renders most of the standard metrics for shard selection mentioned in section 2.7 unusable, like recall and P@n.

Remember that the aim of this thesis is not to evaluate how well the underlying search engine performs; the aim is to aim evaluate what effect shard selection has on the search results compared to not utilizing shard selection. Therefore the assumption that *all documents returned from the search engine* (ElasticSearch) are relevant to a specific query can been used. Using this assumption a new set of metrics can be used, which perhaps could more relevant to search engine administrators than the standard metrics. These metrics will show the negative effects of shard selection, which could be used to evaluate if it is worth utilizing shard selection in a project or not (accuracy vs. performance).

4.3.1 Modified recall

One standard metric which can be used with this assumption is the *R(k)* metric. Rk(K) gives a good indication of how close the algorithms are to being optimal since it measures the algorithms performance compared to a *relevance based ranking* (RBR) baseline. But since it would also be interesting to see what effect the algorithms have even if they are optimal, I've but the RBR baseline in comparison to the results from an exhaustive search. This metric is called *recall* in the experiment chapter.

4.3.2 Pk@N

The recall metric will give a good indication how many documents are lost when shard selection is used. But since users are mostly interested in the top results from a query [8] a new metric has been constructed for this thesis, called *Pk@N*. Pk@N measures the fraction of the top N documents in an exhaustive search that is present in the top ten results retrieved when shard selection is used

$$Pk@N(K) = \frac{\sum_{j=1}^{K} |\{top\_N\_docs(\Omega_j)\} \in \{top\_tN\_docs\_exhaustive\}|}{10}$$

In other words Pk@N will indicate how many of the top-scored documents are lost when shard selection is used. This metric is a bit tricky, since it is dependent on the scoring model of the search engine. ElasticSearch uses the vector-space model with tf-idf weights described in section 2.1.2. The idf-values may be calculated locally on each shard or globally across all shards in ElasticSearch. If local idf values are used the document ranking will be distorted, since shards with a lot of relevant documents will provide a lower score to its documents compared to shards with few relevant documents. As a result global idf-values were used for this metric.

4.3.3 Average number of shards selected

The last metric is the average shard-cutoff value for the algorithms with a dynamic cutoff-value. HighSim and ReDDe were omitted in this metric since they always pick as many shards as was specified in the query.

5

Results and discussion

Negative results are still results. Even 20 000 of them.

Big Bang Theory

THE results from the experiments are presented in this chapter. Three of the algorithms have shown good performance compared to the baselines, but Sushi fails to impress in all experiments. These findings are discussed in detail in the last section including the impact the data distribution policy seems to have on the usefulness of the plugin.

5.1 Experimental result

The follow sections will present the results from the experiments described in chapter 4. The domain value K in the charts below represents the maximum number of shards the algorithms and baselines are allowed to select (shard-cutoff value). Note that the cutoff value is only an upper-bound for the algorithms and the algorithms may select fewer shards if they judge it unnecessarily costly to query K shards.

5.1.1 Modified recall

The results from the recall experiment are displayed in Figure 5.1. On the random data-set the algorithms are not able to produce any significant results for any shard-cutoff values, since for example 10% of all relevant documents are returned when 10% of all shards are selection. This is expected since relevant documents are uniformly

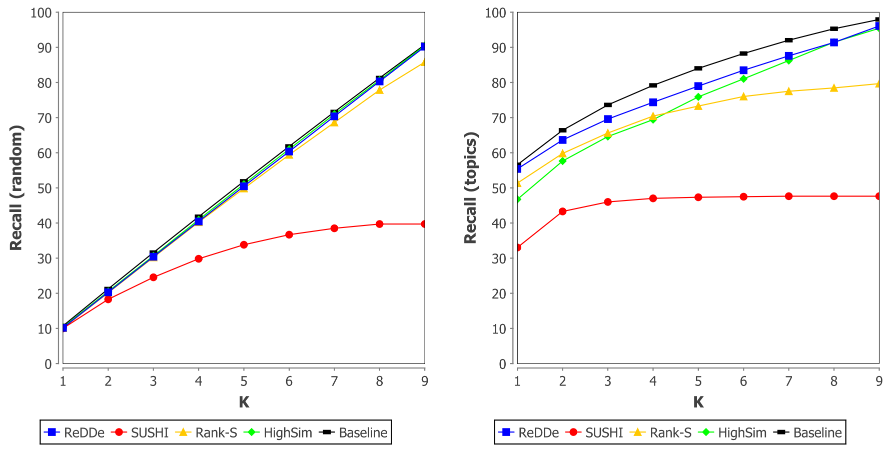

Figure 5.1: Results from recall experiment (random- and topics data sets). The X-axis represents the maximum number of shards the algorithms are allowed to select. The Y-axis represents percentage of all relevant documents returned (on average).

s

distributed among the shards. Further evidence for this is that the RBR baseline is barely able to produce more significant results than the shard-selection algorithms. For the random data-set SUSHI manages to perform worse than what would have been achieved by randomly picking k shards, suggesting its shard cutoff-mechanism may be too aggressive.

The results from the topics data-set are more encouraging. The RBR baseline is able to select as many as 55% of all documents which means that the clustering attribute of the data-set is good. The most interesting aspect to look for here is the gap between the baseline and the algorithms. ReDDe performs very well on all variations of k since it returns about 90% as many documents as the baseline. HighSim trails the performance of ReDDe on low values of k but seems to catch up with higher cutoff values. Rank-s has similar performance with ReDDe on lower cutoff-value but the gap between the algorithms widens a bit with high cutoff-values. The performance of Sushi once again gets worse compared to the other algorithms with increasing value of k, but it performs significantly worse than the other algorithms even with low values of k. A more detailed discussion about the low performance of Sushi can be found in section 5.2

5.1.2 Pk@N

The results from the Pk@N experiments can be seen in figure 5.2, 5.3 and 5.4. Using the random data-set the algorithms are not able to perform better than picking a random subset of shards as in the recall experiment. The baseline is able to produce a significant result even on this data-set. This is not very surprising, especially in the Pk@5 experiments since the top five results can always be found by querying five shards.

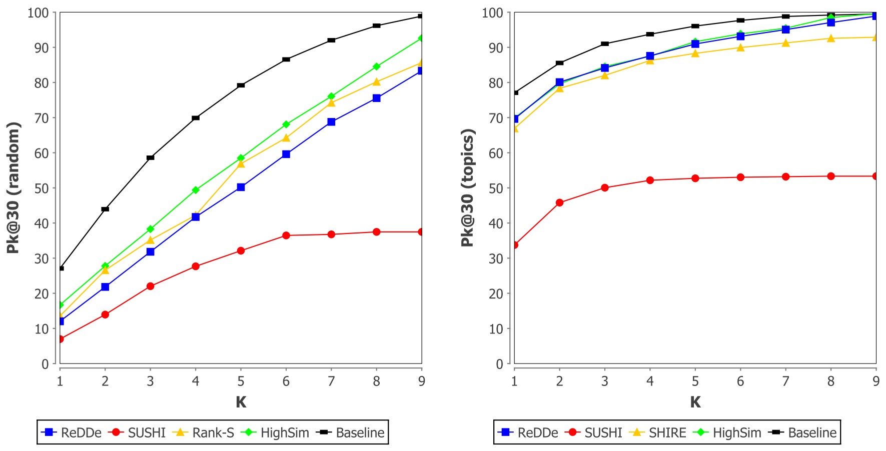

Figure 5.2: Results from the Pk@30 experiment (random- and topics data sets). The X-axis represents the maximum number of shards the algorithms are allowed to select. The Y-axis represents percentage of top 30 documents in an exhaustive search that are found in the top 30 documents from using shard selection (on average).

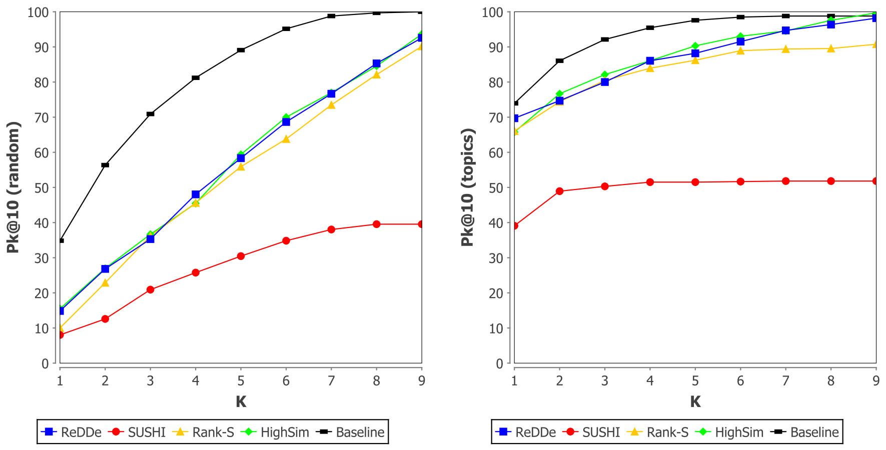

Figure 5.3: Results from the Pk@10 experiment (random- and topics data sets). The X-axis represents the maximum number of shards the algorithms are allowed to select. The Y-axis represents percentage of top 10 documents in an exhaustive search that are found in the top 10 documents from using shard selection (on average).

The results from the topics data-set are encouraging since three of the algorithms are able to retain about 70% of the top results with a cutoff-value of 1. It's somewhat surprising that SUSHI has a low performance in this test, since the algorithm is mainly concerned with picking shards which are likely to contain top-ranked documents. Overall the results from the Pk@N experiments are similar with N set to 5, 10 or 30.

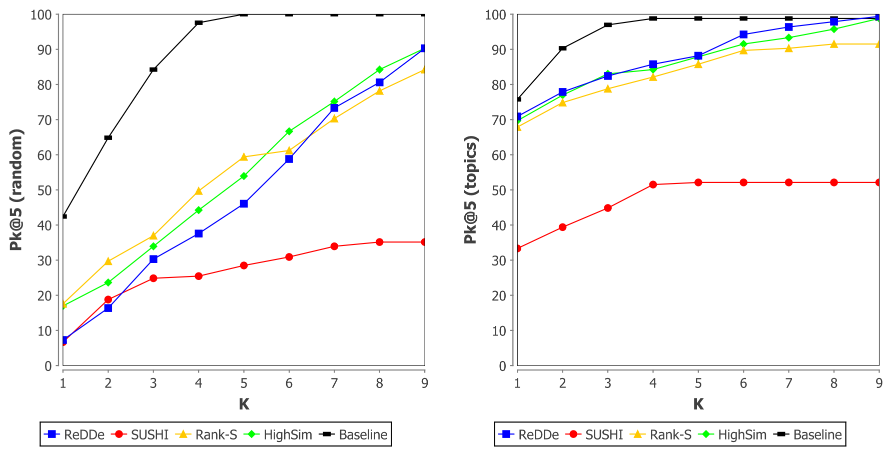

Figure 5.4: Results from the Pk@5 experiment (random- and topics data sets). The X-axis represents the maximum number of shards the algorithms are allowed to select. The Y-axis represents percentage of top 5 documents in an exhaustive search that are found in the top 5 documents from using shard selection (on average).

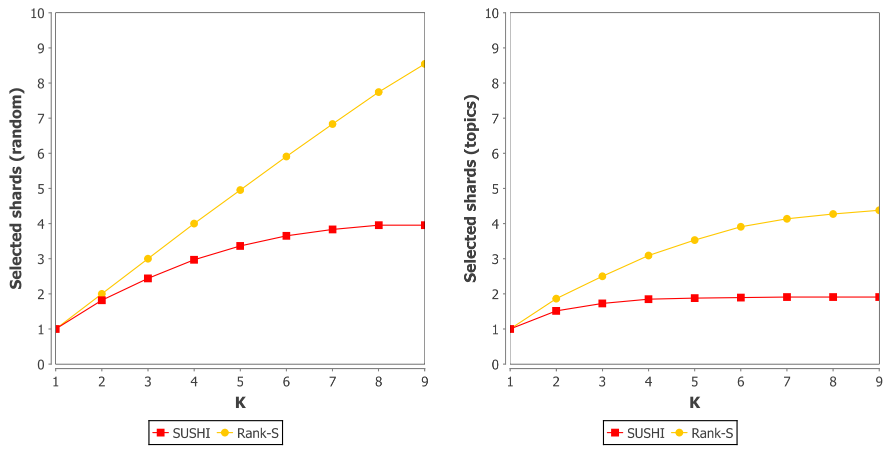

Figure 5.5: Results from the shard cutoff experiment (random- and topics data sets). The X-axis represents the maximum number of shards the algorithms are allowed to select. The Y-axis represents the actual number of shards that were selected by the algorithms (on average).

5.1.3 Shard cutoff

To investigate the efficiency of the algorithms the average number of shards selected was measured. ReDDe and HighSim were left out from this experiment since they always select as many shards as they are allowed to. The results can be seen in Figure 5.5. On the random data-set Rank-s almost consistently selects k shards, which indicates that the algorithm is sound since all shards are just as likely to contain relevant documents. Sushi on the other hand seems to be very aggressive and never selects more than four shards on average.

Using the topics data-set the shard-cutoff mechanism of Rank-S is much more evident. On average Rank-S never selects more than 50% of all shards. Sushi is even more aggressive on this data-set compared to the random data-set. and never selects more than two shard on average.

5.2 Discussion

The results show that the usefulness of shard selection is highly dependent on the clustering attribute of the shards. If documents are randomly distributed to shards, as one of the most popular document distribution policies states, then there is no point in using shard selection since a better method would be to query k shards in a round-robin fashion. If documents are distributed to shards depending on their topic then shard selection could be a viable option to use for saving resources since three of the algorithms in this experiments appear to be sound and perform close to the optimal baselines.

Rank-s is able to perform almost as well as ReDDe and HighSim in both the recall and the Pk@N experiments, but at a much higher efficiency. For each query ReDDe investigates the top 300 documents from the CSI while Rank-S only investigates the top 100 documents. Furthermore Rank-S selects on average 50% as many shards as ReDDE and HighSim when the upper-bound cutoff-value K is high ($\geq$ 80% of all shards). Proper cutoff values for HighSim and ReDDe may be found by experiment on a particular data-set, but since the results are highly dependent on the clustering attribute of the shards this probably these experiment cannot give any general indications. Rank-S eliminated this problem and promises a more *out-of-the-box* solution.

Sushi is consistently outperformed by the other algorithms. This is not coherent with the findings of the original authors [15] although Kalkurni et. al [17] found it to be outperformed by ReDDe and the SHiRE algorithms, but to a lesser extent than in this experiment. Since there is no reference implementation of the algorithm there is a possibility that there is a problem with the implementation in SAFE. Assuming the implementation is correct, informal observations after the experiments indicates that the curve-fitting approach results in the over-aggressive shard-cutoff behavior. Often Sushi believes that the top 50-100 documents can be found in a single shard. Tuning how many of the top documents that should be used to determine which shards to select were tuned from 10 to 30 without any significant increase in the results.

The two data-sets used in this experiment were constructed specifically for this thesis. The two potential problem with this approach was that at least one of the data-sets should be clustered by topics and that there were no relevance judgments available. The results show that the clustering attribute seems to be good in the topics data-set and that using term-filters to fetch twitter posts is a viable method for constructing such a data-set. Since the metrics used in the experiments were able to distinguish the performance between the algorithms and show what effect shard selection has on the search results the relevance judgments problem has also been circumvented. However, using a standard data-set like the TREC micro-blog data-set would probably have given even more confidence in the results since it would make the experiments more replicable, although a clustering algorithm like k-means would have to be used to achieve the clustering attribute.

6

Conclusion

THIS CHAPTER starts out with presenting some suggested improvements to both the algorithms in SAFE and the plugin itself based on the results presented in the previous section and reflections from using and analyzing ElasticSearch. Finally the conclusion to this report will be presented which states that a shard selection plugin like SAFE could be useful in large scale searching if a suitable document distribution policy is used and there is a tolerance for losing some relevant documents in the search results.

6.1 Future work

There are many parameters in the algorithms which, to some extent, seem arbitrarily pre-determined. ReDDe considers the first 300 documents from the centralized sample index, while SUSHI considers 50 and the SHiRE algorithms considers 100 documents. There seems to be no logical reasoning behind these values, other than perhaps that SUSHI and SHiRE aims to be more efficient than ReDDe. In this thesis there has been no formal investigation of these values and how they impact the results from the algorithms. Without further research these parameters probably has to be tested and tweaked for each project the plugin is used in, which hinders the usability of the plugin.

There has been very limited research on the resource-saving aspects of shard selection, even though saving resources is arguably the main reason for using shard selection. Some interesting measurements would be the impact shard selection has on query-to-result time and network traffic. Arguably such an investigation could also look at the parameters mentioned earlier, since the number of documents considered by the parameters may not have much impact on the performance compared to other aspects in the plugin.

Some of the algorithms could be extended to consider more parameters than relevance when shards are ranked. In ElasticSearch it is possible to construct *zones* within the cluster [5]. One usage of this feature could be to put all high-end nodes in one zone and the other nodes in another zone. If the time-flow data pattern is used (see section 2.6.4) then newly created shards could be allocated to the high-end zone while older shards could be migrated to the low-end zone. Shards which belong to the low-end zone should perhaps have a higher burden of proving their relevance since they are more costly to query.

Shard selection is often seen as a possible extension to distributed search engine to increase the scalability of the system, as is the case in this thesis. But perhaps shard selection should be one of the central aspects to consider when designing a search engine when for extremely large-scale search purposes like web-searching. This is the approach taken by Puppin et al [16] which is described briefly in section 2.5.1. The problem is that this approach is not suitable when the scale of the search engine is expected to be medium-sized, for example in enterprise search where the shards can be structured by some logic which do not require shard selection to limit the number of nodes which should be queried.

6.2 Conclusion

This thesis has evaluated methods for shard selection and their effect on the search results in distributed collaborative search engines. One such search engine is ElasticSearch [5] which was used a case study because of its widespread use, modern architecture and open-source license. To answer the thesis question the shard selection plugin SAFE was developed for ElasticSearch. Four algorithms were implemented to the plugin; ReDDE [10], HighSim [9], Sushi [13] and the state-of-the-art algorithm Rank-S [6].

The first research question stated *How is the quality of search-results affected by the different shard selection algorithms?*. The experimental results in section 5.1 showed that even in a best-case scenario shard selection has a negative effect on the search results but that the extent of this effect is highly dependent on how the data is allocated to the shards. When documents are added to shards based on their *topic* as many as 50% of all relevant documents can be retrieved on average by just querying 10% of all shards.

The second research question stated *which shard selection algorithms are most suitable in a collaborative distributed search engine?*. The results showed that the performance of *Sushi* is consistently lower compared to the other implemented algorithms due to its very aggressive shard-cutoff mechanism. The performance of *ReDDe* is slightly higher than *HighSim* and *Rank-s*, but Rank-s is more *efficient* since it selects fewer shards on average. All three algorithms should be suitable to use in a distributed collaborative search engine, but Rank-s and ReDDe should be more scalable than HighSim since the sample size from each shard can be lowered when the number shards grow, whereas the data needed for HighSim cannot be scaled down by such a simple modification. More research is needed to give a definitive answer to this question in form of two other measurements; throughput of search queries and the size of the data needed at the broker nodes.

The conclusion is that there seems to be several algorithms for shard selection which perform very well compared to the baselines used in this report, but even the baselines has a low performance if the document distribution policy do not cluster the documents by some attribute. To make shard selection usable in collaborative distributed search engines both high-performing shard selection algorithms and a suitable document distribution policy has to be used. If these requirements are met and there is a tolerance for losing some relevant documents in the search results then a shard selection plugin like SAFE should be useful in large scale searching.

Bibliography

[1] T. Seymour, D. Frantsvog, S. Kumar, History of search engines, International Journal of Management & Information Systems (IJMIS) 15 (4) (2011) 47–58.

[2] J. P. Callan, Z. Lu, W. B. Croft, Searching distributed collections with inference networks, in: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 1995, pp. 21–28.

[3] S. Banon, Elasticsearch [software], Online, [Online; accessed 17-Jun-2013] (2010-). URL `http://www.elasticsearch.org`

[4] C. D. Manning, P. Raghavan, H. Schütze, Introduction to information retrieval, Vol. 1, Cambridge University Press Cambridge, 2008.

[5] R. Kuc, M. Rogozinski, ElasticSearch Server, Vol. 1, Packt Publishing Ltd. Birmingham, UK, 2013.

[6] A. Kulkarni, A. S. Tigelaar, D. Hiemstra, J. Callan, Shard ranking and cutoff estimation for topically partitioned collections, in: Proceedings of the 21st ACM international conference on Information and knowledge management, ACM, 2012, pp. 555–564.

[7] S. Bockting, D. Hiemstra, Collection selection with highly discriminative keys.

[8] M. Shokouhi, L. Si, Federated search, Foundations and Trends® in Information Retrieval 7 (2011) 1–102.

[9] D. D'Souza, J. A. Thom, J. Zobel, Collection selection for managed distributed document databases, Information processing & management 40 (3) (2004) 527–546.

[10] L. Si, J. Callan, Relevant document distribution estimation method for resource selection, in: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval, ACM, 2003, pp. 298–305.

[11] J. Xu, W. B. Croft, Cluster-based language models for distributed retrieval, in: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 1999, pp. 254–261.

[12] D. D'Souza, J. Zobel, J. Thom, Is cori effective for collection selection? an exploration of parameters, queries, and data, in: Proc. Australian Document Computing Symposium, 2004, pp. 41–46.

[13] P. Thomas, M. Shokouhi, Sushi: Scoring scaled samples for server selection, in: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, ACM, 2009, pp. 419–426.

[14] P. Thomas, D. Hawking, Server selection methods in personal metasearch: a comparative empirical study, Information retrieval 12 (5) (2009) 581–604.

[15] M. Shokouhi, J. Zobel, Robust result merging using sample-based score estimates, ACM Transactions on Information Systems (TOIS) 27 (3) (2009) 14.

[16] D. Puppin, F. Silvestri, R. Perego, R. Baeza-Yates, Load-balancing and caching for collection selection architectures, in: Proceedings of the 2nd international conference on Scalable information systems, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, p. 2.

[17] A. Kulkarni, Efficient and effective large-scale search, Ph.D. thesis, Carnegie Mellon University (2013).

[18] A. Kulkarni, J. Callan, Document allocation policies for selective searching of distributed indexes, in: Proceedings of the 19th ACM international conference on Information and knowledge management, ACM, 2010, pp. 449–458.

[19] S. Banon, Big data, search and analytics (video lecture), [Online; accessed 17-Jun-2013] (Jun. 2012).
URL `http://www.elasticsearch.org/videos/big-data-search-and-analytics/`

[20] P. Thomas, M. Shokouhi, Evaluating server selection for federated search, Advances in Information Retrieval (2010) 607–610.

A

Data set

A.1 Filters applied to Twitter API

The collection of tweets that was used for both the topics- and random data sets were fetched using Twitter streaming API. The term-filters used to construct the ten collections of tweets can be seen in table A.1.

Topic	Date	Filter
Funny	10/04/20	funny, fun, hilarious, joke, aprils fool, laugh
Music	12/04/20	music, song, band, Justin Bieber, album, concert
Politics	12/04/20	politics, Obama, gun control, BarackObama, immigration, gop, Democrats, democrat, republican, congress, sequester
Business	13/04/21	job application, hiring, market, business, stock holders, revnue, sales
Environment	14/04/21	climate, climatechange, environment, co2, global warming, globalwarming, green technology, green tech, emission, emissions, coal, oil, gas
College	15/04/21	university, college, degree, exam, dorm, party, school, thesis, grades, professor, course, courses, semester, study
Health	16/04/21	doctor, sick, ill, pain, medicine, nurse, illness, depression, depressed, medication, health, workout, healty
Family	17/04/21	family, love you, tomyfuturepartner, girlfriend, boyfriend, mom, dad
Sport	18/04/21	Football, Soccer, Hockey, NBA, ESPN, Player, La liga, Bundersliga, League, Cup, Sport,
MobileTech	18/04/21	android, iphone, smartphone, tablet, iphonegames, androidgames, android game, iphone game, ipad game, app

Table A.1: The ten topics used in the data set and their corresponding term-filters

B

Project reference manual

B.1 General information

At the publication date of this thesis the full source code of the shard selection plugin for ElasticSearch is hosted at Github[1]. The plugin will most likely be updated in the future to meet new requirements and fix potential bugs. If the plugin is still hosted on Github at the time of reading I would recommend taking a look at the provided README file instead of this reference manual.

The plugin is named ”Index Selection Plugin for ElasticSearch” on GitHub. In the report the plugin is referred to as ”Shard Selection Extension for ElasticSearch” (SAFE). The reason is simple; for users of ElasticSearch *index selection* makes more sense than *shard selection* since shards referrers to an even smaller entity in ElasticSearch than what is dealt with in the plugin. A more detailed explanation can be found in section 2.1.5

B.2 README.md from Github

[1]Github repository for Index Select Plugin: https://github.com/bergetp/es-index-selection.

Index Selection Plugin for ElasticSearch

This plugin estimates which indices are most likely to contain relevant documents for a search query.

THIS PLUGIN IN IS STILL IN EARLY BETA.

```
-----------------------------------------
| Plugin              | ElasticSearch    |
-----------------------------------------
| master              | 0.90.0           |
-----------------------------------------
```

When is this plugin useful?

ElasticSearch' built-in routing functionality is very useful in many cases. If used correctly a query is only forwarded to one index / one index shard. However, in some cases we cannot rely on the routing-functionality, either because of how the indices are mapped or that we cannot determine what routing value to use in the query.

Let's say that we have indices containing "tweets". We may construct a new `type` with a unique routing value for each user. As long as a query is targeting tweets from a specific user we can use the routing functionality. But if the query is supposed to be a global search across all user we may have a problem if we have a lot of indices and nodes.

This plugin can estimate which indices are most likely to contain relevant documents for a query. Each query will only be routed to the top k indices.

For the plugin to be useful, documents should be clustered by some attribute. A common way to make ElasticSearch more scalable is the "time" data-flow pattern. In this pattern a new index is created on a regular interval, e.g every month. In case the documents indexed are for example tweets, we should get a natural topical clustering of documents, since some topics are more common December (Christmas, gifts) compared to for example February (super-bowl).

How does it work?

This problem is often called "resource-selection", "collection-selection" or "shard-selection". By collecting information from each index their relevance to a query can be estimated. Note that the information fetched from the indexes is an offline operation, and only has to be performed once for index selection to work since the same information can be reused for each query (although it's common to update the information on a regular interval since new documents and indices are usually added to ES in real time).

Surrogate algorithms

Surrogate algorithms collects sample-documents and construct a centralized sample index at the broker-nodes (CSI). The algorithms then use the documents retrieved from querying the CSI to estimate how many relevant documents there are in each index (or which indices contain the most highly relevant documents). There are three surrogate-algorithms implemented at the moment (`redde` , `sushi` and `s-rank`).

Lexicon algorithms

Lexicon algorithms collects tf, df and other statistics from the different indices. There is only one lexicon-algorithm implemented at the moment (`highsim`).

Installation

Installing the plugin

The plugin has to be installed on every node in the cluster.

To build the plugin, type `mvn package` in the IndexSelect directory.

To install the plugin on a node, type

```
$ES_HOME/bin/plugin -url <url to target (zip)> -install index-selection
```

To remove the plugin on a node, type

```
$ES_HOME/bin/plugin -remove index-selection
```

Configuring the cluster

When the ES cluster consists of a lot of nodes it is common to place a couple of client nodes in front of the data nodes. The client nodes don't hold any index data but will handle all the http-trafic, query-routing, gathering of results etc.

This plugin assumes that such a cluster configuration is used. The client nodes will perform the index selection locally before the query is routed to the data nodes.

Configuring the indices

For all fields which should be used to perform index-selection the following should be configured:

- `data.store` should be set to `true`
- `analyzer`: should be the same for all fields

Configuring the client nodes

All client nodes have to have their own copy of the information used to perform index selection. Therefore the they all have to use the same index select configuration.

Set

```
indexselect.refresh_enabled: true
```

to enable index selection on a client node.

The data used for index selection is not automatically updated when a new index is created or a new document is added. Instead it will be refreshed according to a pre-defined time interval:

```
indexselect.refresh_interval: 10m
```

In this case the data will be updated every 10 minutes. A refresh can also be triggered manually using the REST interface (example in next section).

Since the data needed for `highsim' is different from the other algorithms there are two different flags which can be set to define what data should be updated:

```
indexselect.refresh_surrogate: true
indexselect.refresh_lexicon: true
```

In this case, data will be refreshed for all algorithms. This could be useful for testing, but otherwise it is recommended to only refresh one type of data.

Its likely that not all fields will be used for index select. The algorithms currently only support fields with string values..

```
indexselect:
  fields: [message, reply]
```

In this case data for index selection will only be fetched from the "message" and "reply" fields in the indices.

You may also want to specify which analyzer to use with index selection. If not set, the standard analyzer will be used

```
indexselect:
  analyzer: snowball
```

Force refresh of index selection data

Data used for index selection is automatically refreshed in a specific time interval. In some cases it can be useful to force the broker nodes to refresh their data instantly.

```
Curl -XPOST localhost:9200/_indexSelectRefresh -d
'{
   "method" : "surrogate",
   "fields" : ["message, reply"],
   "max_age" : "1m",
   "analyzer" : "snowball"
}'
```

In the example above, the data will only be refreshed if it was less than one minute since the data was last refreshed ("max_age" option). Note that the refresh request has to be sent to all broker nodes for a full refresh to be performed.

Searching using index selection

The following fields are used in the index-select request:

- `indsel_algorithm` : Specified which index selection algorithm to use [`redde` | `rank-s` | `highsim` | `sushi`]
- `indsel_max_indices` : Maximum indices to select and perform query on (if results are confident)
- `indsel_fields` : Fields which should be used for index selection
- `indsel_query_string` : The search query which will be performed on the selected indices
- `search query` : The actual search query which will be performed on the selected indices

Here is an example of a complete index select request:

```
Curl -XGET localhost:9200/_indexSelect-d
'{
   "indsel_algorithm" : "rank-s",
   "indsel_max_indices" : 2,
   "indsel_fields" : "twitter_message",
   "indsel_query_string" : "Justin Bieber",
   "search_query" : {
     "term" : { "twitter_message" : "Bieber" }
   }
}'
```

The example will first perform index selection using the `rank-s` algorithm on the "twitter_message" field in the sample documents with the query-string "Justin Bieber". The term query will be performed on the two highest ranked indices.

Acknowledgements

This plugin is part of my master thesis in Computer Science at University of Göteborg / Chalmers University.

The project has been a collaboration with Findwise AB.

I would like to thank spinscale. Your suggest-plugin very helpful for figuring out how this could be implemented.