UPPSALA
UNIVERSITET

IT 19 009
Examensarbete 15 hp
April 2019

Using Elasticsearch for full-text searches on unstructured data

Jenny Olsson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Using Elasticsearch for full-text searches on unstructured data

Jenny Olsson

In order to perform effective searches on large amounts of data it is not viable to simply scan through all of said data. A well established solution for this problem is to generate an index based on the data. This report compares different libraries for establishing such an index and a prototype was implemented to enable full-text searches on an existing database. The libraries considered include Elasticsearch, Solr, Sphinx and Xapian. The database in question consists of audit logs generated by a software for online management of financial trade.

The author implemented a prototype using the open source search engine Elasticsearch. Besides performing searches in a reasonable time the implementation also allows for documents within the index to be fully removed without causing notable disturbances to the overall structure. The author defined a pattern analyzer for Elasticsearch to allow the use of the Swedish alphabet and accented letters.

The audit log database which this project concerns can contain personal information. For this reason the General Data Protection Regulation was considered during the project. This regulation is a EU-law regarding personal information. The implementation described in this report is meant to serve as a starting point to allow the finding and retrieval of personal information to run more smoothly. The author also made sure that the deletions performed can be made final to comply with the General Data Protection Regulation.

When testing the implementation a database of 708 megabyte containing unstructured data was used. Searching for double search terms, a first name and a last name, in the generated index resulted in an average return time of 11.5 ms when looking for exact matches and 59.3 ms when a small level of misspelling was allowed. The measurements suggest that a solution using Elasticsearch is suitable for the presented problem.

Handledare: Peter Falk
Ämnesgranskare: Sven-Olof Nyström
Examinator: Olle Gällmo
IT 19 009
Tryckt av: Reprocentralen ITC

Contents

1 Introduction **6**
- 1.1 Purpose . . . 6
- 1.2 Requirements . . . 6
- 1.3 Leanon and LISA . . . 6
- 1.4 System structure . . . 7

2 Background **8**
- 2.1 Information Retrieval . . . 8
- 2.2 Inverted index . . . 9
- 2.3 General Data Protection Regulation . . . 9

3 Existing Methods and Comparison **9**
- 3.1 Functionality through existing DBMS . . . 10
- 3.2 Open-source search engines . . . 10
- 3.3 Elasticsearch vs Solr . . . 11
- 3.4 Software decision . . . 12

4 Elasticsearch **13**
- 4.1 Index . . . 13
- 4.2 System architecture . . . 13
- 4.3 General behavior . . . 14
- 4.4 Elastic Stack . . . 15

5 Implementation **16**
- 5.1 Method . . . 16
- 5.2 Input and Reformatting . . . 17
- 5.3 Indexing . . . 18
- 5.4 Searching in Elasticsearch . . . 20
  - 5.4.1 Search query . . . 21
  - 5.4.2 Usage . . . 21
- 5.5 Deletion and GDPR compliancy . . . 22
- 5.6 Storage optimization . . . 23

6 Results **23**
- 6.1 Indexing . . . 24
- 6.2 Performance testing . . . 24

7 Future work **28**

8 Conclusions **29**

1 Introduction

1.1 Purpose

This thesis researches and compares different approaches on how to implement effective full-text searches on an unstructured relational database. A prototype is then implemented with the requirements specified in Section 1.2 to serve as a proof of concept.

A partial motive for this thesis was the General Data Protection Regulation (GDPR) [17]. Because the data related to this project can contain personal information the enforcement of the GDPR can lead to demands for retrieval and deletion of data.

1.2 Requirements

The implementation shall allow the user to make full-text searches as well as searches on specified fields. In addition to searching the implementation shall be able to delete sets of data.

The search component should be made as generic and re-usable as possible. For example the implementation should be able to handle a new type of field being introduced in the future. Furthermore the implementation shall be reasonably time effective. The implementation shall preferably be functional regardless of which database provider is being used.

1.3 Leanon and LISA

The projects derives from a company called Leanon [9]. Leanon develops software systems for companies in the financial industry. Leanon provides a number of systems which can be used as-is or tailored to the customer's need. One of these systems is LISA which is composed of a number of different services for online management of financial trade.

LISA provides a generic component for audit logging. An audit log is a file generated to keep track of what goes on within a system [6]. These logs are stored as unstructured text in a relational database, some parts of the text is in JSON format. This audit trail can contain large amount of data, the database for an internal environment of the system contains about 708 megabyte (MB) and the largest database currently used by a customer is approximately 30-100 times larger. Full-text searching is at the moment not

possible for performance reasons but searches can be made based on a few sets of metadata.

LISA is used by a number of different customers, which use different Database Management Systems (DBMS) for their respective databases. The company is interested in performing more versatile searches on these databases both for the customers using the system and for de-bugging purposes.

1.4 System structure

LISA is built in Java and as mentioned it is composed of several micro services; the functionality of which are not relevant to this project.

When using the platform data is sent to an audit log database in the form of audit *events*. Each event consists of an event type followed by an arbitrary number of key-value pairs. Depending on the configuration of LISA there can be around 20-80 different types of audit events being reported. Some of these pairs are fairly small, for example the key string 'userid' connected to a pointer of the userid of whoever performed the action which triggered the event. Other event types can contain values which are in turn string representations of Java objects or JSON objects with several sub-fields.

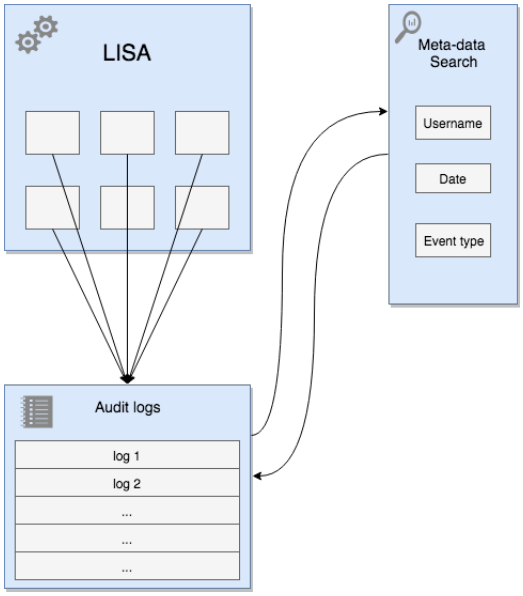


Figure 1: The structure of the existing system.

These logs can in turn be viewed using the platform and the existing, limited, search functionality. At the moment the user can make searches based upon certain time intervals, the user who generated the log or the type of event for a log.

2 Background

2.1 Information Retrieval

Information Retrieval (IR) is the process of finding and fetching something from a collection of data based upon a search request [4]. A piece of data in this collection can be referred to as a *document*. Full-text search is a form of IR where the data being kept is in text format and the search request can be based on any part of said data, not only specific parts of it. The collection of data is known as an index which is an optimized data structure based upon the original data. The index can also contain the entirety of the original documents.

The search request begins with a user specifying what they are looking for in the shape of one or more search *terms*. The search terms are then converted into a search *query* by the system [4]. When the actual search takes place the query is matched against the index and any suitable documents are returned to the user, most commonly in an order of relevancy. During the search a relevancy score can be calculated for each matching document. The relevancy score is an indicator of how well a document matched against a search query.

Both when generating the index and the query several steps can be taken to prepare and alter the given input, this process is known as *analysis* [13]. The analysis process starts with a tokenizer which turns the input into a stream of tokens, for example splitting a sentence upon blank steps to allow searches to match on each individual word rather than the entire sentence. These tokens then go through zero or more filters. The filtering can for example involve stemming; reducing a word to its base form, lowering capital letters and the removal of stop-words. Stop-words are words that will in all likelihood not be relevant when performing a search, such as 'for', 'in' or 'a'.

2.2 Inverted index

An index can be built in different ways and one of the commonly used techniques is the *inverted index*. This structure keeps a mapping of in which documents each of the terms within an index resides [13].

Forward index

Document ID	Term
1	Cat, Fish
2	Cat

Inverted index

Term	Document ID
Cat	1, 2
Fish	1

Figure 2: Simplified example of a forward index and an inverted index.

The mapping of an inverted index is term-centric, it goes from a term to a list of the documents which contain said term. A *forward index* on the other hand is document-centric [14], it maps from a document to a list of terms. During a search a forward index would need to scan through all of the terms for each document while an inverted index would simply look up the term and return the attached list of documents.

2.3 General Data Protection Regulation

In April 2016 a new regulation in EU law was adopted and as of May 2018 it became enforceable [17]. Through the General Data Protection Regulation individuals are meant to be given larger control of how their personal information is used. The aspects of GDPR which can be of interest regarding this project are rights to see what information is being kept about oneself as well as have that information removed, within certain limits. The GDPR was not enforceable at the beginning of this project and any consequences of it affecting Leanon are mere speculations. However as a precaution in case anyone requests to view or have their personal information removed Leanon wishes to be able to accommodate this person as smoothly as possible. This is the main reason behind the requirement of deletion mentioned previously.

3 Existing Methods and Comparison

The following alternatives were considered when planning the implementation.

- Use built-in search-functionality in the currently used database management systems.
- Use an open-source library to build a separate, independent, search component.

3.1 Functionality through existing DBMS

The Database Management Systems currently in use by different customers are *PostgreSQL* [5], *Oracle Database* [12] and *Microsoft SQL Server* [8]. All of these supports some form of indexing and full-text search capabilities. Despite this fact and without considering their effectiveness the problem of re-usability still remains. That is, if a prototype or complete implementation was made for one of the DMBS's, the ability to search in the others would not have improved. Also there is nothing stopping a user from changing their preferred DBMS or a new customer appearing with yet another type of DBMS. For these reasons the project will not utilize any built in search features from a DBMS, but rather focus on creating a separate, independent search component.

3.2 Open-source search engines

There are a lot of options available in terms of open-source libraries. To limit the comparison this thesis will focus on the top nine Search Engine (SE) options from DB-Engines's ranking [2] as well as a top five list from myTechLogy [11]. This results in eleven candidates.

The search engines Splunk, MarkLogic, Microsoft Azure Search, Algolia, Google Search Appliance and Amazon CloudSearch were all eliminated because they had a commercial license or only a limited free edition. The remaining five candidates are listed below.

- Apache Lucene
- Elasticsearch (ES)
- Solr
- Sphinx
- Xapian

Sphinx and Xapian are both written in C++ but have support for programming in Java. Apache Lucene, Elasticsearch and Solr are all written in Java.

Apache Lucene is an information retrieval software library which is considerably more low-level than the other remaining options. Both Elasticsearch and Solr are built on the Apache Lucene library. Using Apache Lucene directly would most likely not result in anything more efficient or as easy to use as Elasticsearch or Solr. For this reason Apache Lucene will be crossed from the list of candidates.

Looking at a comparison from 2017 [18] it can be noted that Xapian is slower at indexing and produces a notably larger index than the other candidates. The same behavior can be seen in another comparison from 2009 [15]. For these reasons Xapian will not be used for the implementation.

ES and Solr produce a slightly smaller index than Sphinx [18]. Sphinx has the fastest indexing speed followed by Elasticsearch and lastly Solr [18]. However none of the indexing times appear unreasonable and the initial indexing will not be a reoccurring time concern. Sphinx received the lowest value when calculating the accuracy of the relevancy scoring [15]. Since the relevancy score indicates how well a document matches against a search query a low level of accuracy means that it can be harder to find the desired information. Due to the index size and low level of accuracy on the relevancy scoring Sphinx will be taken of the list of considerations.

A closer comparison of the two remaining Lucene alternatives is given in Section 3.3.

3.3 Elasticsearch vs Solr

The current versions of Elasticsearch [13] and Solr [16] both deliver fairly similar attributes. For example stemming, result ranking, cross-platform support and 'more like this'-features exist in both options. As mentioned previously Elasticsearch was noted to have a faster indexing speed than Solr while Solr have a faster searching speed [18]. However another comparison from 2015 showed ES to have a faster searching speed [10]. Relative performance appears to depend on circumstances with no exact way of determining the results without making an actual implementation.

Because of the similarities in performance other factors also needed to be taken into consideration.

Documentation: Both Solr and Elasticsearch have fairly extensive documentation. However Solr has been around longer and thus have had a longer time to build up information such as user made tutorials and forum threads.

Ease of use: Though this is mostly a question of personal preference, in general Elasticsearch is considered easier and more intuitive to setup and get started with [14, 7]. However since it might be harder to find information about Elasticsearch, the work to improve on a base setup could be harder. Solr on the other hand might take more work setting up but once it is done there is a larger community and better documentation to help you along.

JSON: The database in question for this project is as mentioned partly in JSON format. Both Solr and Elasticsearch have support for JSON but ES utilizes JSON to a much larger extent, taking JSON objects as input and storing documents in JSON format [7, 13] . Having parts of the data already in the correct format might make the implementation easier.

Scalability: Elasticsearch was initially designed to be much more cloud-friendly and scalable. In later times though Solr holds up on this end as well through 'SolrCloud' which, through 'Zookeeper', creates an equivalence to Elasticsearche's scalability [14]. However it should still be kept in mind that this is an additional, separate application.

Analytics: Although it is not relevant at the moment it is worth mentioning that the support for analyzing and visualizing data is considerably stronger in Elasticsearch than in Solr [14].

3.4 Software decision

After considering the options a decision was made to use Elasticsearch for the implementation. ES was deemed more suitable for a couple of reasons. The project had a relatively short time-frame which fits well with the short learning curve of ES. The fact that the audit logs are partially in JSON form at the moment also pointed towards Elasticsearch. Lastly the scalability and analytics possibilities could prove helpful for someone continuing with this implementation past the scope of this project.

The following sections contain a more in-depth explanation of Elasticsearch and the method of implementation.

4 Elasticsearch

The first version of Elasticsearch was released in February 2010 [13] and has since then grown to be one of the most popular open-source search engines at the time of writing this, topping both of the lists which were used to find suitable search-engine candidates for this project. The version used in this project was Elasticsearch 6.2.3. It is as mentioned a search engine built in Java, utilizing the Apache Lucene library. Elasticsearch is supported by all operating systems through a Java VM.

4.1 Index

Elasticsearch, or Lucene as a whole, uses an inverted index (Section 2.2). Within Lucene the documents are all referred to by different integers, known as the document number. Each segment of an index is made up of a number of smaller files, some of which are optional. The following is some of the important information being kept in the index.

Segment info: General metadata about the segment, for example the number of documents it is based on.
Field names: The fields names which are used within the index.
Term dictionary: All of the terms which have occurred in the indexed documents along with how many documents contain each term and a pointer to the frequency and proximity data for said term.
Term Frequency data: The document number of every document which contains a term along with the number of times the term occurs in said document.
Term Proximity data: The position(s) at which a term occurs in a document.

4.2 System architecture

When an instance of Elasticsearch is started up it generates a node which either joins an existing cluster of nodes or, if none exists, creates a new cluster [13]. An index created in this node will then be spread out across a number of *shards*. A shard is a part of the main index which in reality is a separate Apache Lucene index. Shards can be either *primary*, containing parts of the main information of the index, or *replicas*, containing a copy of the information of a primary shard. If nothing else is specified an index will

be set to have five primary shards each accompanied by a replica shard. Thus by default ten shards will be generated. However the replica of one shard will never be kept on the same node as its corresponding primary shard and so if there only exists one node in a cluster no replicas will be generated.

This system can make effective use of multiple machines or servers (nodes) and parallel processing on a single machine (shards). The replicas function both as a backup in case a node fails and as a possible performance booster, since searches can be made on replicas as well as primaries. The data stored in each shard is automatically balanced when new data is added to or removed from an index.

When indexing the data is divided into several small write-once and read-many time segments [13]. This is done automatically by Elasticsearch and a segment which have been written to the disk cannot be changed. The entire segment can however be removed or merged with another segment. Upon indexing a field called the 'source'-field is normally kept [13]. This field is not searchable and contains the original JSON input. Storing this is in most cases convenient but it does however mean that additional storage space is needed.

4.3 General behavior

The results of a search is by default returned in order based on their relevancy score, to calculate this Elasticsearch uses the Term Frequency/Inverse Document Frequency (TF/IDF) scoring mechanism [13]. When indexing or building search queries this score can be manually increased or decreased to allow for more tailored results. Among other things the score can be set to behave differently if the search term occurs in a certain field. The term frequency part of the scoring can also be turned off, meaning a document containing the search term several times will not be given preference over one where it only occurs once.

Altering the ranking can be of interest if for example some fields are more important than others, say a field for the username compared to a field for side notes. When searching for 'Anna' it might be of more interest to find documents where 'Anna' is the user compared to documents where 'Anna' is mentioned briefly in a note. The ranking feature can also be turned off and the search is then referred to as a filtering. The result of a filtering is returned in an arbitrary order and the search can be quicker since the work of scoring has been removed.

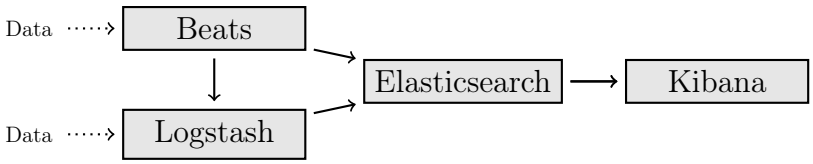
In later releases Elasticsearch no longer performs stop-word removal by default during indexing. This is because although smaller indexes will naturally take up less space and deliver faster results, the removal of stop-phrases can reduce the accuracy of the results. For example 'great' and 'not great' could be seen as the same thing and phrases like 'to be or not to be' could be entirely removed. This feature can however still be activated manually at an indexing-level, or on a searching-level through a 'common term'-query which gives lower or no relevancy score to such terms.

Normally a match will only occur when the exact same word used in the search query occurs in a document. This can be modified to allow *fuzzy* searches or to allow regular expressions to occur in the search term. A fuzzy search can allow matches to occur on phrases which vary slightly from the given search term [13]. The level of fuzziness, or changes allowed, can either be automatically chosen depending on the length of the search term or set to a specific number. One change can be deleting, adding or changing a single character, or switching the position of two adjacent characters. Allowing fuzzy searches can make the program more user friendly since it takes minor misspellings into account but it also means significantly more work may need to be done to check for potential matches.

After performing a search up to 10 000 of the matching documents can be retrieved in one go. Retrieving larger sets of documents will naturally take longer time and when dealing with large amount of documents the retrieval will need to be done in pages.

4.4 Elastic Stack

Alongside Elasticsearch the Elastic company has developed a set of open source products which together form what is known as the Elastic Stack [3]. The Elastic Stack allows one to manage, search and visualize data using a collection of open source products which are easily integrated with one another.



The core product Elasticsearch is of course the search component of the Elastic Stack. Kibana can be used to visualize and analyze the data stored in ES. It has an easy to use interface which shows changes to the data in real time.

For gathering data Logstash and/or Beats can be used. Logstash is a data collection pipeline which can input data from- and output data to a number of different sources. Through various filtering the given data can also be modified before forwarding. Beats is a more lightweight product which does not have the ability to alter data. The two can be used in combination, letting Beats forward its data to Logstash.

Logstash uses a Java database connectivity (JDBC) plugin to communicate with SQL databases. Different databases might require the JDBC-plugin to use different drivers, these must be stated and passed into the JDBC-plugin separately. In addition to reading an entire database a schedule can be established to continuously read any changes to the database.

For the purpose of this project Logstash can be thought of as a device to turn data into proper JSON-format before forwarding it to ES. Kibana was used during the implementation as a means of simple, undocumented, testing.

5 Implementation

As a foundation for the implementation a postgres-server, of roughly 708 MB was used. The server consisted of log entries retrieved from an internal test-environment of the LISA-platform. The logs could be reached through two separate databases, one where each row contains an event-id, event-date and event-type (111 MB). The other database contains an event-id, name and value (597 MB) where the event-id in the first database corresponds to one or more event-ids in the second.

5.1 Method

Firstly a connection was made to the postgres-server containing the database, both to take in the existing data and to continuously handle incoming data. The data was then filtered to the appropriate JSON format. An Elasticsearch index was created and a suitable analyzer was defined.

A Java application was implemented and connected to Elasticsearch through a REST API. In the application queries were defined to handle searching and deletion. A simple terminal user interface was created to demonstrate the application.

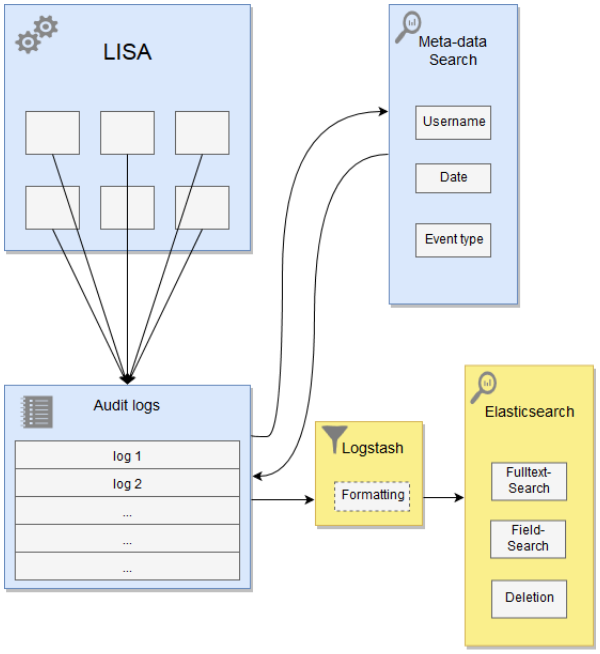


Figure 3: The structure of the prototype system.

5.2 Input and Reformatting

Logstash was used to format the input before it was indexed by Elasticsearch. In order to reach the server the JDBC-plugin was used. By modifying the setup this plugin can handle data from any of the three supported data storage facilities currently used in combination with the software. To allow for larger sets of data paging was enabled with a page size of 100 000 fields. An input schedule was also established to continuously look for new entries in the server.

A SQL-query was setup to join the two databases and the data was then filtered so that each document contained all of the information regarding a certain event-id. A field for the date of indexing was added to each document. Some field renaming and deletion of redundant fields which are automatically generated by Logstash also took place in this step. The reformed data was

then sent onwards to Elasticsearch.

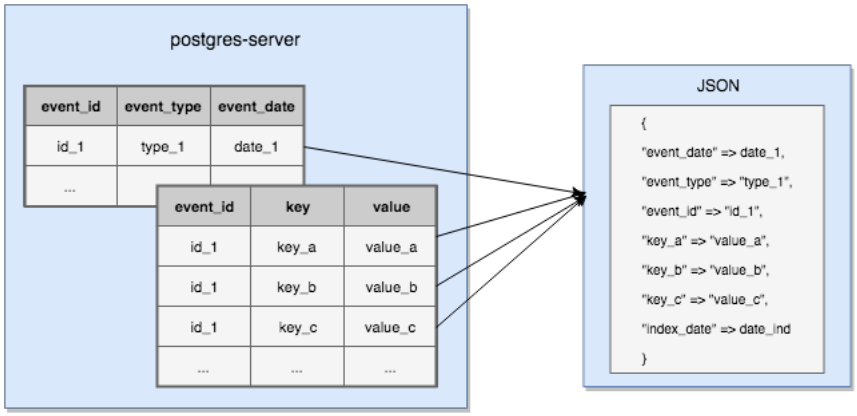


Figure 4: A general description of the reformatting input and output.

5.3 Indexing

Information in the audit logs regarding users often contains names in the form of usernames or email addresses. Since searches performed on names could be a common use case it would be preferable if this could be done smoothly and intuitively. As an example we will assume that 'john.doe@place.com' exists in a document. One would expect that the email above should be matched when using the search term 'john doe', without the dot or other surrounding elements. However by default Elasticsearch uses an analyzer which does not separate words by the dot-symbol and similar tokens. This means that 'john doe' will not match the document containing 'john.doe@place.com'.

To produce the desired search behavior the implementation initially used a regular expression in the search query. Each word of the search term was surrounded with the star symbol which means that regardless of any prefix or suffix of the input a match can still occur. Using regular expressions when searching can have a large impact on the overall performance. The index can not be used as intended since it has to look for words which might not have been indexed to begin with. When using regular expressions as mentioned above entire documents could need to be scanned from start to finish. Although using regular expressions in the implementation resulted in the desired search behavior and the return time of searches was reasonable, it was clearly not the best of solutions.

To improve on the implementation and avoid the regular expression causing performance issues further down the road, the default analyzer of Elasticsearch was changed. ES provides an analyzer called 'pattern' which uses a pattern, on the form of Java regular expressions, to split up words. A simple setup script was written to use the pattern analyzer when generating the index. This change allowed for the desired search behavior regarding dots and removed the downsides of using regular expressions in the searches. Using regular expressions at this stage, compared to in the search query, means a one time event instead of a reoccurring one. When creating the index the database will need to be read in either case and changing the analyzer simply changes when a new word is added to the index.

The implementation allows both fuzzy and exact searches. Comparing these two one would expect the exact search to return less or the same amount of hits as the fuzzy one. However, during the first round of testing it was noted that some search terms returned more hits when performed in the exact form. It turned out that the search engine had problems with names that contain accent symbols.

As an example let us use the name 'Linnéa Larsson'. It was found that indexing the first name as-is would result in it being split up into two words, 'Linn' and 'a'. Analyzing the name as a search term showed that when used in an exact-search context it was split up into the same two words. However when using the fuzzy-search it was seen as one continuous word, 'Linnéa'. When two terms are entered together without any boolean operator between them the default behavior creates an 'OR' connection. What this implies for the noted odd case is that the searches resulted in the following queries being used.

exact	(Linn OR a) AND Larsson
fuzzy	Linnéa~1 AND Larsson~1

To understand this behavior we take a look at the default pattern used for analyzing, which is '[\\W]'. This means that any symbol other then a-z, 0-9 or '_' will break up the input. Thus this pattern will split on any accented-characters, such as á or é, as well as the Swedish characters 'åäö'. This explains why the exact search behaves the way it does, but it does not account for the difference between the two search types. Looking closer at the documentation it was found that analysis during searches is only performed when the query type used require exact terms, meaning that no analysis takes

place when a fuzzy search is performed. The difference between the searches is then to be expected and the problem lies solely within the analyzer. The pattern used was therefore modified to allow for the Swedish and accented characters, resulting in the following final pattern:

$$[\backslash\backslash W\&\&[\hat{}\text{À - ÿ}]]$$

It was decided not to split on the underscore symbol since it is present in several field names. The majority of event types, a field present in all logs, also contains this symbol. For example, the search string 'USER_UPDATED' has a quite distinct meaning and when searching for it one would not wish for the same results as when searching for the two words separately.

5.4 Searching in Elasticsearch

For the searching component a small Java application was built, utilizing Elasticsearch's high-level REST-API. Through this a connection was made to ES and some queries were set up to demonstrate the searching possibilities. This in turn could be used through a simple terminal interface.

The application allows a user to perform a search on all fields or a specified one. Several searches can also be combined into one, for example searching for one term in a specified field and another on all fields. The default result is that of a fuzzy search, but the user can specify to have an exact search instead. The search can also be modified to perform a more light-weight filtering rather than a scored search. This returns the same documents as the scored search would but in a possibly different order.

When starting the application a rest client is created which connects to ES and the user can then enter the desired search string. The input is first modified slightly to be on the correct format. Any extra symbols, such as '.' or '@', are trimmed away and if it is to be a fuzzy search the tilde-symbol is added to each of the search terms. Multiple search terms are connected with an 'AND' meaning all of the terms must be present for a match to occur. The modified string is then turned into a *string query*, more on this in Section 5.4.1, and specified as either a regular search or a filtering.

5.4.1 Search query

Elasticsearch provides a number of different query types and to decide which to use we first looked for one where a search could be done on all available fields. In earlier versions of ES there was an '_all' field for this purpose, however this was deprecated in version 6.0.0. To search on all available fields the two main options now are the query_string query and the multi_match query. The main difference between these two can be seen when more than one search term is given. The multi_match query requires all given search terms to exist in the same field for a match to occur, while the query_string query will return a match as long as every term appears in some field.

For the purpose of this project the latter one was deemed more suitable. The query_string query appears more user friendly since it does not expect the user to know if certain content resides within the same field or not before making a search. However, along with the pure full-text searches, the implementation also allows for searches to be made on specified fields and so a user with more insight to the log structure can still produce a relatively narrow search.

5.4.2 Usage

The terminal user interface allows input on the form 'header:search_term' where the header is optional. The search term can consist of multiple search terms separated by a comma. Terms can also be specified not to exist in the result. This is done by simply prepending a minus to the search term, '-search_term'.

These terms can in turn be specified to be searched for on a single field or on all fields. To limit the search to a specified field the search term takes the form of 'field_name = search_term'. The header can contain one or both of the words 'exact' and 'filter'. The word 'exact' means that the default fuzziness of the search is removed and only exact matches are returned. 'filter' means that the scored search is replaced by a simpler filtering.

Although it is not recommended as standard usage it should be mentioned that some regular expressions can be used when entering a search request through the implementation's user interface. This is not the intended use but rather a side effect of the chosen query type which can be exploited. It does however not work in all cases and can be costly, further work will need to be done if it is desired to use more complex regular expressions during searching.

5.5 Deletion and GDPR compliancy

A concern at the start of the project was whether removing parts of an index would affect the search performance. If this was the case the deletions would be handled by removing the relevant parts in the underlying database on a scheduled basis and then re-indexing all of the remaining data. However the following information shows that this will not be a problem and deletions will not require re-indexing.

Since ES stores its documents in segments, a search is performed on each segment in turn and the results are combined. Search times increase as the number of segments increase. To keep the number of segments at a reasonable number Elasticsearch tries to merge smaller segments. A user can also demand force-merges if desired.

Sending a command to delete a document in Elasticsearch essentially means that the document will be marked for deletion. The documents marked for deletion will be skipped whenever a search is performed and for the user appear to be non-existing. The actual removal of a document marked for deletion happens when the segment in which it resides is merged with another. What this implies in terms of performance is that if there are a large number of documents marked for deletion still in existence it can have a negative effect on the response time of searches. However this effect is reversed when the documents are actually removed. To make sure that the relevant documents have truly been removed, say if it is done in response to a personal information removal complying with the GDPR, one would need to ensure that a merge has taken place afterwards. This can be done by simply sending a command to perform a force merge on an index.

Marking an indexed documents for removal can be done easily through the API. To demonstrate this a small delete-feature was implemented. To delete a document the user simply specifies the identifier of what they which to remove. This feature can be extended to take in lists of identifiers or remove everything in a search hit for example.

5.6 Storage optimization

The simple user interface of the prototype displays the returned documents directly. This design was chosen to help debugging. However, if this application leads to an actual full-scale implementation it will clearly not use this interface and thus storing the entire _source field might be redundant and costly in terms of memory usage.

The index used for testing (Section 6) is 797 MB which is 13% larger than the original database. In an attempt to reduce the size of the index a second index was setup where the '_source' field had been all but removed, keeping only the 'event_id' field. This does not effect the searching behavior, but it does mean that the content of the retrieved documents cannot be viewed directly. However since the LISA platform can access and display the logs directly from the underlying database, it seems reasonable to simply return the 'event_id's of the relevant documents and then use the event_id's to extract the logs from the database.

The resulting simplified index had a size of 373 MB which is 53% of the original database. When removing the source one also loses the possibility to, based on nothing but the index itself, update documents or perform re-indexing. However after the audit logs have been generated they should not be altered and so the indexed documents should not need to be updated. In fact there is some work being put in at the moment to ensure that this does not happen. In regards to the re-indexing possibilities this can be done as easily using the actual starting database, which the company has access to and thus if there occurred a need to re-index, the '_source' field would still not be necessary.

6 Results

The following machine was used during testing.

Operating system MacOs High Sierra *version 10.13.4*

Processor 2.7 GHz Intel Core i7

Memory 16GB 2133 MHz LPDDR3

The following tests were performed mainly on an index of 797 MB consisting of 541 281 documents. In some tests, where explicitly stated, a simplified

index of 373 MB with the same number of documents was also used. Figure 7 includes measurements from both indices and Figure 8 was made using the simplified index.

6.1 Indexing

Indexing a database takes approximately two minutes to initiate and thereafter chunks of 100 000 fields are indexed, taking between 30 seconds and one minute for each chunk, 42.5 seconds on average. The test database contains 20 such chunks, resulting in an index time of roughly 16 minutes. There is no notable time difference when indexing the optimized index compared to the original one. Limiting the input to only parts of the database has no notable impact on the initial time before the indexing starts, indicating that this time is fairly fixed. The time, in minutes, to index a database containing X number of fields can thus be estimated with the following formula:

$$T = 2 + 7.08e^{-6} * X$$

The size of the databases used in combination with the system can vary greatly, the largest one at the moment is estimated to be about 30-100 times the size of the test database. This would result in an index time of between 7 and 23 hours. Although a time close to one full day sounds rather long this is just the initial setup and most likely a one-time occurrence.

6.2 Performance testing

For the tests two lists of search terms were used; *event_type* (single term) and *name* (double term). The first one contains the 71 types of events which are present in the used database and the other contains 450 names. The second list was generated by creating all possible permutations of the top 30 most common Swedish forenames, 15 male and 15 female, combined with the top 15 most common Swedish surnames [1]. These lists were then used to make searches on all fields or the following existing fields *event_type*, *event_id* and *USER_ACCOUNT*.

The field *event_id* contains numbers, while *event_type* contains words in snake-cased format. Snake-casing means two or more words are joined together using the underscore symbol. The content of an *event_type* field will always be one of the strings in the *event_type* list of search terms. Thus searching for an event type in the *event_id* field should not yield any results and searching for it in the *event_type* field should always yield some results. The field *USER_ACCOUNT* exists in several of the documents and contains

information about the user, often in the form of first and last names or email addresses. After a search is performed the matching documents need to be retrieved in order for the user to view them. If nothing else is specified the number of documents to retrieve is set to zero.

Search type	**Search term**	**Field**	**Average** (ms)
Fuzzy	single term	event_type	1.5
		event_id	40.9
		all	87.9
	double term	USER_ACCOUNT	0.2
		all	59.3
Exact	single term	event_type	0.1
		event_id	0.0
		all	4.3
	double term	USER_ACCOUNT	0.0
		all	11.5

Table 1: Average return times for single and double term searches on different fields.

Table 1 shows that as expected exact searches are quicker then fuzzy ones and searching in a specified field is faster then searching on all fields. Comparing the searches made on all fields one can see that the single term search is faster than the double term search when performing exact searches and slower otherwise. This can be traced down to the fact that the event types in general contain much longer strings, 21 symbols on average, and the length of a search term of course correlates strongly with the search times in the case of fuzzy searches. The names have an average length of six symbols in the forenames and eight in the surnames.

Figure 5 shows that there is no clear correlation between return time and the number of hits. Also note that in Figure 5 no documents were retrieved. For more information regarding the impacts of retrieving different number of documents see Figure 8.

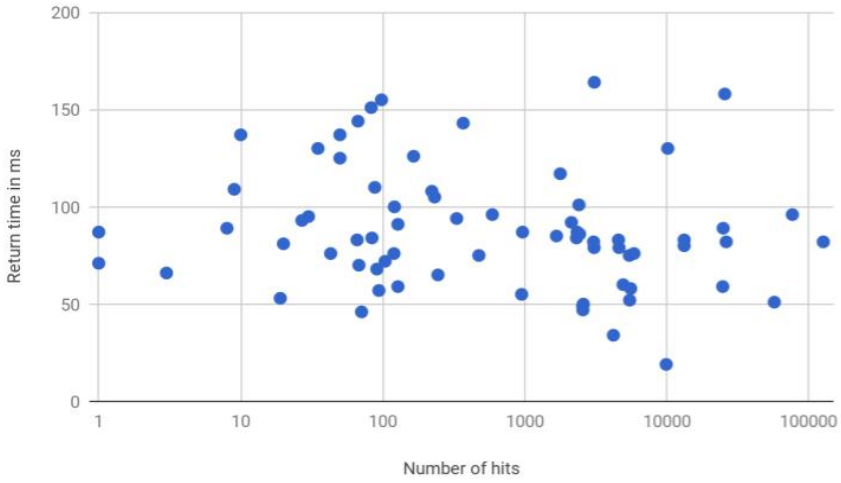


Figure 5: Single term search (event_type).

Figure 6 shows the search times when performing the simpler filtering compared to normal scored searching. On fuzzy searches there is a slight improvement in the average return time while on the exact searches the average search time is cut nearly in half. The fuzzy search times are very close to each other because when doing a fuzzy search the time needed to try variations of the given search term(s) is significantly bigger than the time needed to rank the result.

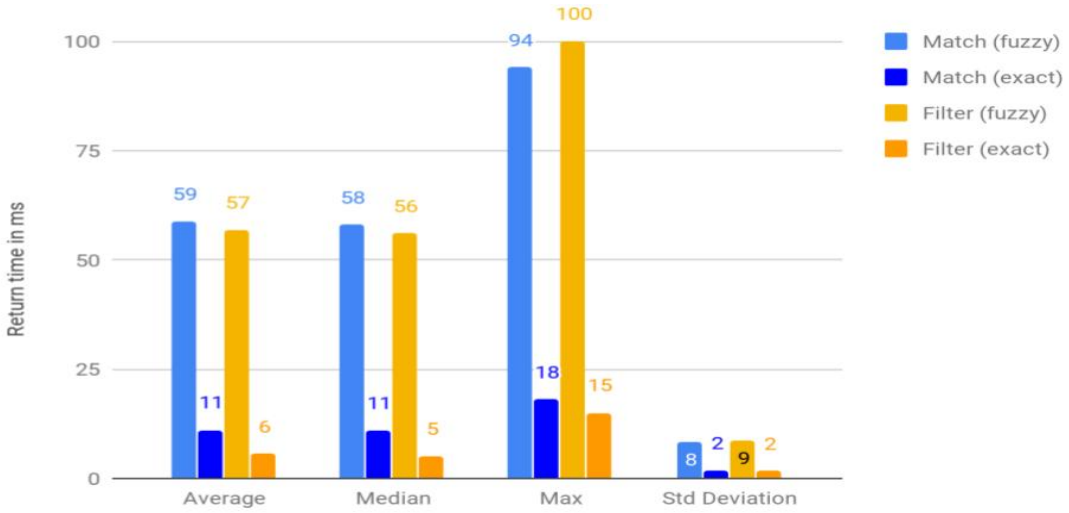


Figure 6: Time for double term search (name) when performing filtering compared to ranked searches (matches).

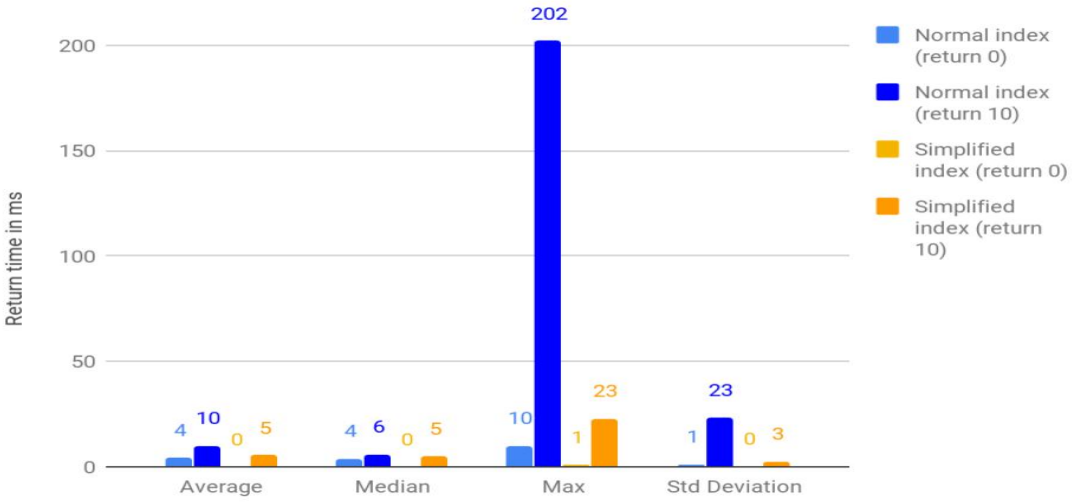


Figure 7: Time for exact single term search (event_type) performed on all fields in both the original index and the simplified one.

From Figure 7 one can see that the average return times are notably lower when using the smaller index. The maximum return time for searching in the larger index when documents are being returned is however quite unreasonably large compared to its average value. The search term in question is a specific event_type. A closer look at the contents of the returned documents showed that they each contained a field value consisting mainly of a large list of integers which could explain the long return time.

In Figure 8 the simplified index was used to compare the return time when allowing retrieval of five different numbers of documents. From Figure 8 one can see that as long as the number of search hits stay below a few thousands it does not make a noticeable difference whether all or a few of the hits are returned. When the number of hits becomes too high however the return time suffers. This behavior is to be expected and when dealing with large returns it is recommended practice to use the scroll API [13] to manage the returned documents.

To ensure that the program behaved properly the amount of hits for each search was noted. A number of expected behaviors could be seen, such that: when using the same search term exact searches returned less or the same number of hits as a fuzzy one, searching on all fields returned more or the same number of hits as searching on a specified field and performing a filtered search returned the same number of documents as an unfiltered one.

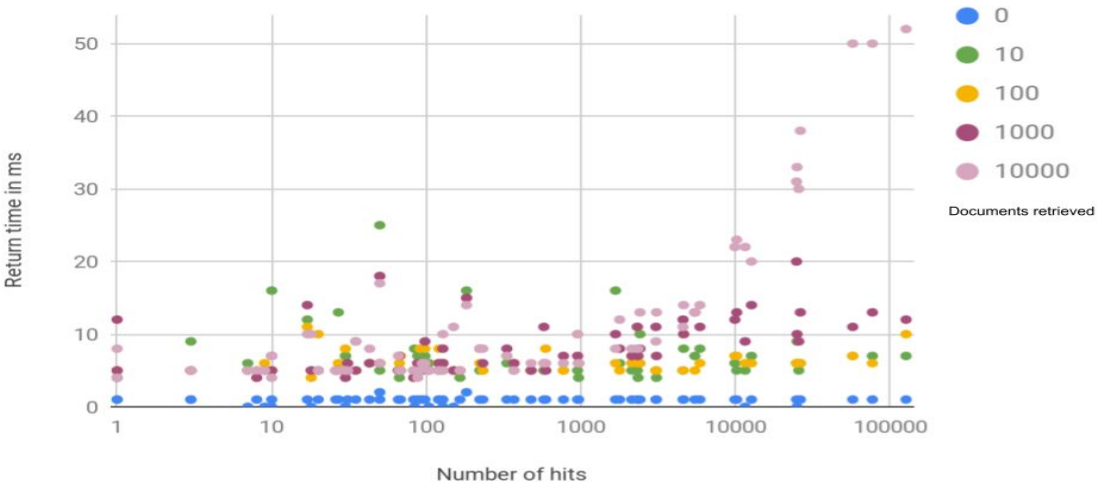


Figure 8: Return time for single term search (event_type) on simplified index using different return sizes.

7 Future work

Since the purpose of this report was only to create a proof of concept there are naturally a lot more aspects which can be further developed and improved upon. Some of these are mentioned here.

When indexing the original database the content of a field can still sometimes contain fairly large sets of data. More complex filters can be set up to further divide these fields into several subfields, which could have a positive effect on the performance.

The scale of implementation and testing during the project have been fairly small, all done on one single machine. Setting up a system which functions over a larger number of machines and users will naturally require further work.

The scoring of documents have not been modified and uses the default Elasticsearch behavior. Someone with more insight could tweak this behavior to easier retrieve more relevant documents. For example some fields might contain more interesting information than others, in such a case the score of a match in that field could be boosted to result in a higher ranking for the document. In contrast to this the order of the result could also be changed to depend on something other then the ranking. For example the event date

could be used, or the alphabetic order of the event types. The number of retrieved documents from the search hits can as seen in the testing have an impact on the overall performance and so implementing some form of paging will probably be necessary.

The minimized index described in Section 5.6 would most likely be the better approach if creating an actual implementation. With the database used during the project a decrease of about 50% in the index size could be seen, as well as a boost of the performance. Another approach to minimizing the storage needed for the index could be to allow the system used for generating audit logs to send its data directly to the index, completely circumventing the current database structure.

As mentioned in Section 1.4 it is possible to perform a few forms of metadata searching in the existing software. If this project leads to an implementation using ES it might be worth considering replacing these search options with solutions implemented using ES, to keep things more uniform.

8 Conclusions

A concern with using Elasticsearch was the lack of documentation and the relatively small user community. This did however not prove to be much of an obstacle; in the author's opinion there were some parts where the documentation was a bit scarce but what existed was sufficient.

One reason for choosing ES over Solr was the presence of JSON format in some of the audit logs. At the moment the JSON parts still lie embedded as-is within the documents. A more detailed implementation could allow the sub fields of the logs to be treated as sub fields by the index as well. Thus the existence of JSON format in the logs could still be taken advantage of. Such behavior could be accomplished by defining more advanced filters in the Logstash configuration.

In Section 1.2 it was specified that the prototype should aim to be generic, time effective and preferably not bound to a single database provider. The prototype appears to fulfill these requirements. The implementation will have no problem if new field names or other changes are introduced in future log content. The time measurements (Section 6) does not indicate any alarmingly high return times. Logstash can be modified to take input from a number of different sources and the plugin used only needs a minor change

in the setup to handle data from any of the currently supported DBMSs.

The application does not match related words or synonyms, for example 'fruit' will not match 'apple' and 'autumn' will not match 'fall'. Though such behavior might be desired in more general types of search engines the data in the audit logs relevant to this project are on a fairly strict format and does in general not contain the type of language which uses synonyms. However ES does have a feature for suggesting similar search terms which could be used if desired.

Overall the results suggest that the required search behavior can be achieved with the approach presented in this report. The database used in the testing is smaller than the ones used by most customers but since Elasticsearch is foundationally meant to handle changes and scaling of its indices it should not have any difficulty handling larger databases.

References

[1] Statistiska centralbyrån. Most common Swedish forenames and surnames, spelling variations taken into account, 2017.

[2] DB-Engines. Ranking of search engines, June 2018. https://db-engines.com/en/ranking/search+engine.

[3] Elasticsearch. Elastic stack home page, 2018. https://www.elastic.co/products.

[4] Ayse Goker, John Davies, and Damon D. Ridley. *Information Retrieval: Searching in the 21st Century*. John Wiley & Sons, Incorporated, Hoboken New Jersey, USA, 2009.

[5] The PostgreSQL Global Development Group. Full text search capabilities in PostgreSQL, 2018. https://www.postgresql.org/docs/9.5/static/textsearch-intro.html.

[6] Darrel Ince. *A Dictionary of the Internet*. Oxford University Press, 2009. http://www.oxfordreference.com/view/10.1093/acref/9780199571444.001.0001/acref-9780199571444.

[7] Rafal Kuć. Solr vs elasticsearch differences, 2017. (Blog posting). https://sematext.com/blog/solr-vs-elasticsearch-differences/.

[8] Douglas Laudenschlager, Gene Milener, Saisang Cai, Craig Guyer, and Steve Stein. Full text search capabilities in Microsoft SQL Server, 2018. https://docs.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-2017.

[9] Leanon. Home page, 2018. https://leanon.se/.

[10] Tom Mortimer. Elasticsearch vs. solr performance: round 2, 2015. (Blog posting). http://www.flax.co.uk/blog/2015/12/02/elasticsearch-vs-solr-performance-round-2/.

[11] myTechlogy. Top 5 open source search engines, 2018. (Blog posting.) https://www.mytechlogy.com/IT-blogs/8685/top-5-open-source-search-engines/#.Wvv0jNOFMuR.

[12] Oracle. Full text search capabilities in Oracle Database, 2018. http://www.oracle.com/technetwork/documentation/index-098492.html.

[13] Marek Rogozinski and Rafal Kuc. *ElasticSearch Server*. Packt Publishing, Birmingham B3 2PB, UK, 2014.

[14] Dikshant Shahi. *Apache Solr: A Practical Approach to Enterprise Search*. Apress, Berkeley, CA, 2015.

[15] Vik Singh. A comparison of open source searchengines and indexing, 2009. (Blog posting.) https://partyondata.com/2009/07/06/a-comparison-of-open-source-search-engines-and-indexing-twitter/.

[16] David Smiley, Eric Pugh, Kranti Parisa, and Matt Mitchell. *Apache Solr Enterprise Search Server Third Edition*. Packt Publishing, Birmingham B3 2PB, UK, 2015.

[17] The European parliament and the council of the European union. General data protection regulation. *Official Journal of the European Union*, 2018. http://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN.

[18] Aleksei Voit, Aleksei Stankus, Shamil Magomedov, and Irina Ivanova. Big data processing for full-text search and visualization with elasticsearch. *International Journal of Advanced Computer Science and Applications, Vol. 8, No. 12*, 2017.