Vrije Universiteit Amsterdam

Master thesis

Optimizing ElasticSearch for Texcavator

Author:

Eric de Kruijf

Supervisors:
Janneke van der Zwaan
Jason Maassen
Henri Bal

August 28, 2015

Abstract

This thesis will examine the ways in which ElasticSearch [11] (a full-text search engine) can be optimized for a text mining tool called Texcavator [63]. Texcavator is implemented on top of ElasticSearch to allow research on a medium sized (500GB) static dataset of digitized newspapers provided by the National Library of the Netherlands (Koninklijke Bibliotheek, KB [52]).

One purpose of this study was to assess the extend to which changing the number of shards or replicas within ElasticSearch improves the search performance for consecutive or concurrent queries. By deploying several ElasticSearch configurations and measuring their performances on actual user queries, insight is obtained in composing an optimal configuration. Using multiple shards on a single node does not improve search performance, but using multiple replicas does improve search performance under the load of concurrent queries.

Texcavator also allows its users to analyze search results in several ways. For example, by plotting the document distribution over time or by generating a word cloud over all returned documents. A second purpose of this study is to optimize word cloud generation while not degrading other users' experiences. MapReduce turns out to be a viable option compared to extracting document termvectors directly from ElasticSearch using ElasticSearch clients.

1 Introduction

Texcavator [63] was developed in collaboration with the Netherlands eScience Center [48] to analyze Big Data in the form of text mining. Texcavator is implemented on top of ElasticSearch [11], a full-text search engine which in turn relies on Apache Lucene [2] for document storage and querying. ElasticSearch provides ease of use in terms of a plug-and-play cluster setup procedure and well performing default settings.

Texcavator is set up to work on a specific static dataset, namely a collection of digitized newspapers provided by the National Library of the Netherlands (Koninklijke Bibliotheek, KB [52]). This research focuses on analyzing this dataset along with actual user queries in order to optimize ElasticSearch for this specific use case. The optimizations revolve around indexing speed, search request response times and word cloud generation speed.

This research covers the following research questions: (1) How should ElasticSearch be configured in order to obtain a high indexing throughput? (2) Which ElasticSearch settings improve search query response times for given user queries? (3) How do shards and replicas influence search query response times for consecutive as well as concurrent queries? (4) How can Texcavator's word cloud generation method be sped up without degrading other users' experiences?

ElasticSearch' architecture is outlined in Section 2, providing an understanding of the ElasticSearch software package without requiring prior knowledge. Related work is reviewed in Section 3 and the experimental setup is defined in Section 4. In Section 5 the dataset and acquired user queries are presented. Several data import methods are discussed in Section 6, optimizing the search response times is explained in Section 7 and word cloud generation is examined in Section 8. A final discussion in Section 9 concludes this research.

2 ElasticSearch' architecture

Before optimizing ElasticSearch it is important to understand its global architecture and to become familiar with the corresponding terminology. Figure 1 shows ElasticSearch' architecture, consisting of multiple components in a hierarchical configuration. In section the architecture is dissected into its separate components and the inner workings and importance of each component is explained.

2.1 Clusters and nodes

An ElasticSearch cluster consists of one or more nodes (e.g. node A and node B in Figure 1). ElasticSearch offers an easy plug-and-play experience when it comes to launching a cluster. Running a single instance of ElasicSearch essentially provides a single node cluster. New nodes can be added to a cluster by simply launching another instance of ElasticSearch and providing it the name of the cluster it should join. One node will be voted to be the master node, mainly to reduce ElasticSearch' communication complexity.

The cluster as a whole will store all the data by dividing it evenly over the available nodes. Data will be automatically reallocated when new nodes join the cluster. When the data is stored redundantly (see Section 2.4) it

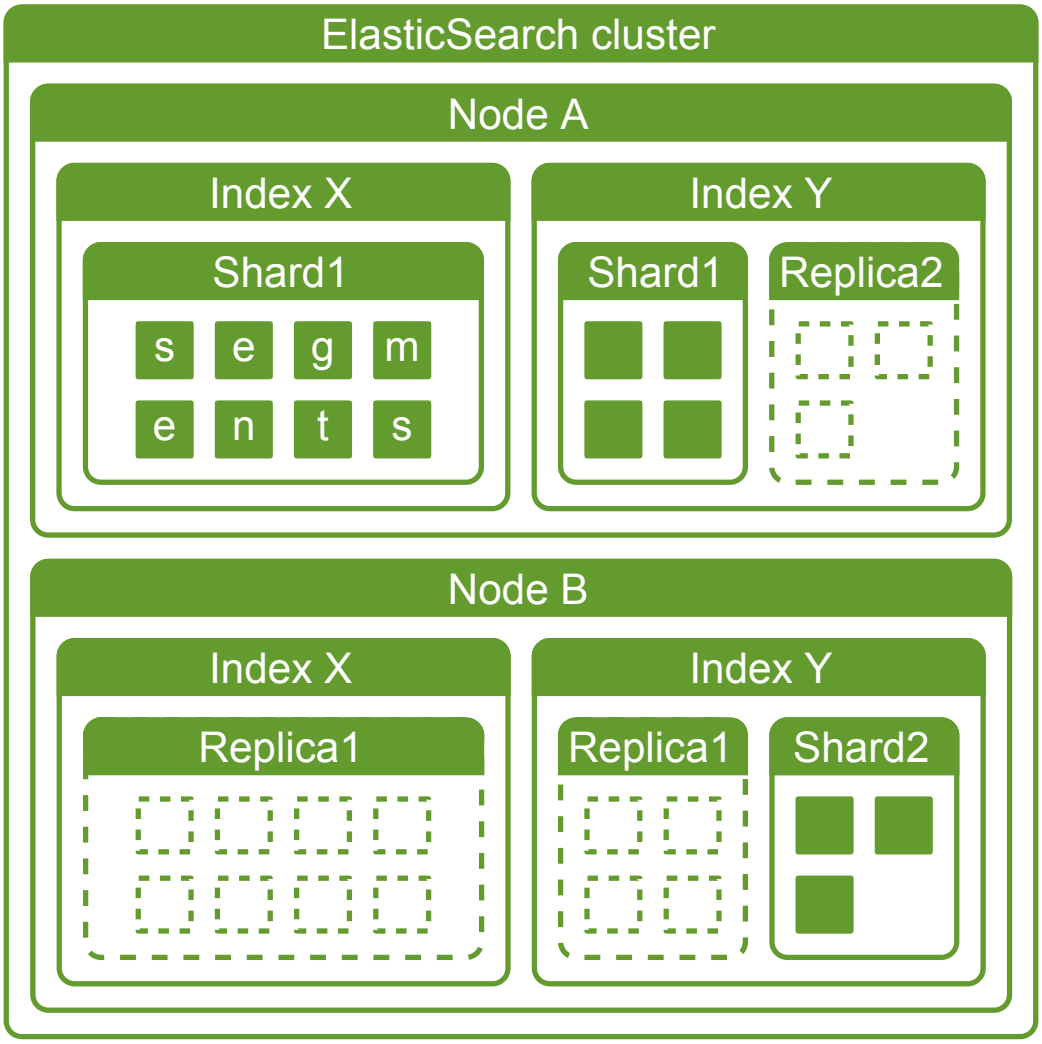

Figure 1: ElasticSearch' hierarchical architecture.

will even be reallocated when nodes leave or shutdown unexpectedly. Next to storing the data, the cluster offers a HTTP API[1] endpoint at each node. The API can be used to insert new data or to search the existing data. A search request made at an endpoint is distributed over the available nodes to ensure it hits each shard exactly once [32]. This can be either a primary shard *or* one of its replicas.

[1]API: Application programming interface

2.2 Indices, types and documents

A node can contain zero or more indices, which are used to store collections of documents. Figure 1 shows two indices, namely index X and index Y, available on each node. Documents within an index are single data entries, e.g. a tweet *or* a blog post, which can be assigned certain types.

Types are used to provide categorization of documents within an index by labeling a document with a certain tag (e.g. `tweet` or `blogpost`). Each type can be assigned a mapping which defines how the analyzer should handle the document data during indexing. For example, storing a tweet requires different fields compared to a blog post and each field may require different parsing or analysis (e.g. dates versus strings). Mappings and analyzers are explained in more detail in Section 2.3.

Figure 2 compares ElasticSearch to a relational database management system (RDBMS) such as MySQL to give more insight in the terminology. A MySQL server owns multiple databases as an ElasticSearch cluster owns multiple indices. Within a database multiple tables (types) can be defined and each table consists of rows (documents) which are single data entries. A single row (a document) complies with the tables column definitions (a mapping) in terms of which data is stored in what manner (e.g. `TIMESTAMP` versus `TEXT`).

Figure 2: Terminology comparison/overview of ElasticSearch and an DBMS (e.g. MySQL).

2.3 Mappings

Setting a mapping for a document type tells ElasticSearch what analyzer to use for which fields within a document. ElasticSearch offers tens of mapping settings and analyzers which are listed in the documentation [25]. To give an example of a mapping: `paper_dc_date:{type: date, format: dateOptionalTime}`. This specifies the `paper_dc_date` field to be parsed by the analyzer as a `date` using the `dateOptionalTime` format analyzer. Section 5.2 will elaborate on the settings relevant to this research.

If no mapping is specified or unspecified fields are encountered within a document, these fields are parsed as if they were strings requiring full text search and are added to the mapping as such for future reference. ElasticSearch does not specify this default behavior in its documentation [22]. However, processing fields for full text search logically requires processing power. Therefore, if fields should not be parsed as such, it is recommended to define them accordingly within the mapping. For example by excluding them from analysis entirely (`index: not_analyzed`).

2.4 Shards and replicas

ElasticSearch differs from an RDBMS in the way it stores its documents within the indices. An index is not a single datastore containing all documents, but it can consist of one or more smaller datastores named *shards*. A node can own one or more shards per index. This allows for parallel execution of queries by processing different parts of the index (i.e. the shards) at the same time. In Figure 1, index X is built up out of a single shard and index Y out of two shards.

When multiple shards are used within an index, the cluster can be scaled horizontally by adding more nodes to the cluster. As explained in Section 2.1, ElasticSearch will reallocate data to the new nodes in order to spread out the data distribution over the entire cluster. Shards are the smallest unit of data ElasticSearch can transfer between nodes. Therefore, using more shards results in a higher horizontal scalability. The number of shards an index holds is defined at index creation and cannot be changed. Using too many shards on a single node causes unnecessary overhead while parallelizing a search request using a single shard is impractical. Therefore the amount of shards should be chosen with care.

ElasticSearch offers the possibility to replicate an index to increase the scalability even further [38] as well as improving the data availability in case shards or nodes fail. An index can be replicated zero or more times. Replicating an index will replicate each of its shards. Each original shard, or *primary shard*, will then have zero or more replica shards, or *replicas*, assigned to it. Replicas are a read-only copies of primary shards and are not allowed to handle write operations (index requests), but are allowed to handle read operations (search requests). Each replica will be updated accordingly when its primary shard receives an update. Together both primary shards and replicas are evenly distributed over the available nodes. In Figure 1, each

index is replicated once, resulting in one replica per shard. When node B fails, replica 2 will be converted into a primary shard to allow index requests. This shows the increased data availability in action. Section 7.5 explores optimal settings for the number of shards and replicas for the KB dataset.

When a search request is issued at an endpoint that node will distribute the request to each shard [32], resulting in parallel query execution. When a shard is replicated, the request is only sent to the primary shard *or* one of its replicas, possibly residing on another node as each node is aware of the data distribution [21]. By using a round-robin procedure to determine which shard (primary or replica) to use [32] the load is also spread over the entire cluster. ElasticSearch advises to use the round-robin approach [21] also when accessing API endpoints on a multiple node cluster. Most of the clients developed by ElasticSearch automatically become aware of the nodes in the cluster and also use the round-robin approach to pick a new endpoint for each request [49].

2.5 Document distribution

To distribute the documents evenly over the available shards within an index, ElasticSearch uses the Murmur hashing function [37] [28] on the document identifier to route it to a certain primary shard. This document identifier can be defined by the user or can be automatically generated by ElasticSearch. This also allows fast retrieval of specific documents (by identifier) since such requests do not have to hit all shards, just the one the document has been routed to during indexing.

2.6 Lucene index

Although shards cannot be split up by ElasticSearch, they are yet another abstraction above the real data storage engine. As noted in Section 1, ElasticSearch is built on top of Apache Lucene, a high-performance, full-featured text search engine library [2]. Each shard is its own Lucene index which stores the document data fields along with possible meta data. As in any search engine, inverted indices are used to map terms to documents [4] [55]. This way it can easily be determined which documents match terms in search queries. Term dictionaries, or termvectors, can also be stored, which denote what terms are within a certain document, including the number of occurrences (frequency) within that document.

Since all this extra data is stored inside the Lucene index for each document, it becomes costly over time to add single documents. Therefore a Lucene index consists of sub-indices, or segments, which can be created when new documents are added to the index [3]. Figure 1 shows, Shard 1 of index X consists of 8 segments and index Y has 4 segments in its first and 3 in its second shard. Apache Lucene provides high level functions to incorporate parallel search on segments, but also allows low level functions to arrange custom scheduling [60]. Each segment on itself is an independent index and can be searched separately [3]. However, since searching hundreds of segments is not optimal either, Lucene can combine, or *merge*, these segments into fewer segments. This is analogous to a fragmented hard disk drive of which the performance can be increased by defragmenting the data, making sure the data can be read sequentially. Finding a good balance between the number of segments and occurring merges is important for the first research question (how to obtain a high indexing throughput), which is discussed in Section 6.

The ElasticSearch documentation states that optimizing an index by merging its segments allows for faster search operations [31]. It is important to know that optimizing an index does not mean that there cannot be any more documents added to it. When a new document is added to the index, a new segment is created for it. If the index becomes too segmented, it can be (automatically) optimized again.

3 Related work

There is little acadamic research published on ElasticSearch. Especially on optimizing ElasticSearch for a static dataset as which is the case with Texcavator. This section discusses several papers covering systems implemented on top of ElasticSearch, similar to Texcavator.

Both et al [1] built an architectural layer on top of several datastores amongst which ElasticSearch as well as Openlink Virtuoso Server and PostGIS. This research focuses on interpreting query semantics rather than optimizing ElasticSearch' search settings. They argue that users prefer knowledge-driven search over traditional search solutions. In their research they pick three different types of requirements users could request in a search for hotels. Logical properties (e.g. has wifi), geospatial properties (e.g. north of London) and properties driven by textual information (e.g. pool for children). Logical

properties are handled by Openlink Virtuoso Server, geospatial properties by PostGIS and textual information by ElasticSearch. They do not report optimizing ElasticSearch, but they do praise ElasticSearch in their conclusion as *"Providing high-performance federated large-scale full text search"*.

Kononenko et al developed DASH [53], a system which provides real-time search/analysis on Bugzilla entries. They argue that ElasticSearch is fit for the job mainly due to its (horizontal) *scalability*, *agility* in handling many incoming entries and its search *performance*. It is stated that ElasticSearch outperforms traditional databases in terms of search since it can search multiple shards concurrently. However, the main focus for DASH lies on real time data analysis, instead of analyzing a static dataset as which is the case in Texcavator. ElasticSearch' default configuration is praised in Section 2: *"The Elasticsearch server is easy to install, and the default configuration supplied with the server is sufficient for a standalone use without tweaking, although most users will eventually want to fine tune some of the parameters"*. This case study on Texcavator assesses whether making this is assumption is justified.

Most comparable to Texcavator is Quarry, by Linder et al [54]. Quarry provides three components: a bulk importer, a search service and interactive visualization methods. Quarry is built for enterprise data analysts and data scientists who 'play' with a dataset to test assumptions. Currently many of them write SQL queries in combination with small scripts and visualize exported data using tools such as Excel or R. However for none of Quarry's components the performance was measured nor discussed. The only measurement was a general usability evaluation by nine data scientists and two PhD students. They evaluated Quarry as a valuable tool for big data exploration.

The KB itself also offers a search engine, called Delpher [10]. This search engine can search all the documents the KB has digitized over the years (newspapers as well as books). Delpher allows similar possibilities as Texcavator does in terms of composing a text query in a rather extensive query language. It also allows defining date ranges, document types or geospatial distribution filters the query should comply with. Unlike Texcavator, Delpher does not allow for more advanced data analysis in terms of *visualizing* entire query result sets.

4 Experimental setup

An ElasticSearch cluster is currently set up for Texcavator at SURFsara [61] and Texcavator is already used for research. Since this is a production environment, it has not been used to conduct this research. The required experiments have been executed on the DAS-4 cluster [8], since the DAS-5 cluster [9] was not yet available at the start of this research. Table 1 compares the SURFsara nodes to two types of machines available at the DAS-4 cluster.

	SURFsara	**DAS-4 (fat)**	**DAS-4 (default)**
CPU model	Intel Xeon E5-2420	Intel Xeon E5620	Intel Xeon E5620
- clock speed	1.9GHz	2.4GHz	2.4GHz
- cores/threads	6/12	4/8	4/8
#CPUs	2	2	2
RAM	128GB	48G	24G
Local storage	4*3TB (RAID0)	2*2TB (RAID0)	2*1TB (RAID0)
Network	10GbE	IB and 1GbE	IB and 1GbE

Table 1: Texcavator's current production machines (at SURFsara) versus the DAS-4 machines used in this research.

The DAS-4 machines described in Table 1 reside at the Delft University of Technology [64]. These machines were the best ones available for this research in terms of RAM, even though the amount does not come close to the RAM available in the SURFsara machines. The VU site houses several machines yielding 64GB of RAM, but these machines already participate in a Hadoop cluster setup, and using them resulted in inconsistent benchmarks.

The ElasticSearch cluster at SURFsara consists of three physical machines each hosting two ElasticSearch nodes, resulting in a six node ElasticSearch cluster. Due to the amount of RAM available on the DAS-4 machines, each machine hosts a single ElasticSearch node. Most experiments were performed on a single node cluster hosted on a fat machine. Multiple node clusters were created by booting extra default machines hosting single ElasticSearch nodes set to join the available cluster.

Texcavator communicates with ElasticSearch using the official ElasticSearch client for Python, namely the elasticsearch-py library [46]. Code snippets from the Texcavator source are used to make sure the experiments access ElasticSearch in the same manner.

At the time of writing, ElasticSearch is already at version 1.7.1 [43], but at the start of this research its latest version was 1.5. Therefore ElasticSearch 1.5 [44] was used to perform all experiments to keep the results comparable.

Finally, Python 2.7.9 [58] is used to execute all the Python scripts, because some Texcavator scripts are incompatible with Python 3.

5 Data and query properties

Texcavator operates on 400 years of digitized newspaper entries, provided by the National Library of the Netherlands (Koninklijke Bibliotheek, KB [52]). The dataset is described in Section 5.1 and its ElasticSearch mapping is explained in Section 5.2.

ElasticSearch offers a rich query language and since Texcavator is already used as a research tool, real user queries are available. Six users volunteered their queries to be used in this research, resulting in a total of 280 queries (see Appendix A). The different user queries are discussed in Section 5.3, which shows how ElasticSearch' query language is put to use. Finally, Section 5.4 explains how Texcavator's query strings are converted to ElasticSearch queries.

5.1 Data description

The dataset is composed of 400 years of digitized newspaper articles, or *documents*. These documents are digitized by scanning them and running the scans through optical character recognition (OCR) software. Both the scanned images, as well as the text are available for use, but only the text data is used for search. The dataset consists of 3545 gzipped XML files (86GB compressed, 347GB decompressed), containing a little over 102 million documents. On average, each file contains about 30.000 documents.

The gzipped files were retrieved from the KB over a time period of four years and the files are named after the timestamp of when they were retrieved (e.g. `DDD_2009-11-11T15:03:26.000Z__didl_0.data.gz`). The retrieval dates are used in this research to create subsets of the data. The retrieval dates are chosen to divide on because they are readily available. Table 2 shows the number of documents and files per year of retrieval. It also shows the space it takes up when imported into ElasticSearch and the average document size.

Using the entire dataset for each benchmark slows down the research, because for several benchmarks the dataset has to be re-indexed or transferred between multiple nodes in the cluster. Therefore, from Section 7 (*"Search performance optimization"*) on, only the subset retrieved in 2012 is used. This subset is about half of the size (56%) of the entire corpus and is more manageable. Table 3 compares the documents (grouped per 50 years) in the entire dataset to the documents in the 2012 subset. This shows that the subset is representative of the entire dataset in terms of document distribution except for 1900-1945, where the proportion of documents in the 2012 dataset is smaller than half.

year	**#documents**	**gzip files**	**size**	**avg document size**
2009	3,934,686	262	20.7GB	5.5kb
2010	1,031,248	127	3.1GB	3.1kb
2011	25,787,544	667	84.5GB	3.4kb
2012	57,268,447	1835	194GB	3.5kb
2013	14,661,941	654	67.3GB	4.8kb

Table 2: Properties per subset (by year) where size denotes the size when imported into ElasticSearch.

date range	**#documents entire dataset**	**#documents 2012 subset**
1600-1649	10,335	10,312
1650-1699	53,191	52,729
1700-1749	184,593	154,414
1750-1799	531,679	298,597
1800-1849	2,237,015	904,875
1850-1899	9,663,308	4,361,439
1900-1949	54,407,187	23,719,850
1950-1999	35,628,402	27,766,184
2000-2049	87	47

Table 3: Document distribution by date comparison of the entire dataset versus the 2012 subset.

Four types of documents are available, namely: articles, adverts, illustrations and family notices. Similarly, four distribution regions exist: national within the Netherlands, regional within the Netherlands, the Netherlands Antilles, Surinam and Indonesia. Each document has a single type tag and distribution region tag assigned to it. Table 4 shows the document distribution based on type tags and Table 5 on distribution region tags.

type	#documents
Article	69233754
Advert	29592799
Illustration	1970921
Family notice	1918323

Table 4: Document type distribution of the entire dataset.

location	#documents
National (Netherlands)	46561226
Regional (Netherlands)	40941927
Netherlands Antilles	2030005
Surinam	1211460
Indonesia	11971179

Table 5: Distribution regions of the entire dataset.

The tables show that users can exclude large amounts of the dataset simply by filtering on tags. In Section 7.4 this is used to Texcavator's advantage by partitioning the dataset based on these tags.

5.2 Data mapping

Before importing documents into ElasticSearch, it is important to understand how it should process and store specific data fields, as explained in Section 2.3. The dataset has 31 fields for each document (see Appendix B), of which 8 are defined in ElasticSearch' mapping. Table 6 lists these 8 fields and their corresponding mapping parameters.

The defined fields contain five mapping attributes, namely: `format`, `index`, `include_in_all`, `term_vector` and `type`. All fields are defined as `type: string`, except for `paper_dc_date` which is of `type date` and its `format` is defined as a date with optional time (`dateOptionalTime`). This allows the user to search within a certain date range.

Several fields are set not to be analyzed (`index: not_analyzed`). This allows these fields to be used as tags. For example, `article_dc_subject` can assume the following values: *artikel* (article), *advertentie* (advert), *illustratie met onderschrift* (illustration) or *familiebericht* (family notice) as mentioned in the previous section.

field	mapping
article_dc_subject	type: string include_in_all: false index: not_analyzed
article_dc_title	type: string term_vector: with_positions_offsets_payloads
identifier	type: string include_in_all: false index: not_analyzed
paper_dc_date	type: date format: dateOptionalTime
paper_dc_title	type: string term_vector: with_positions_offsets_payloads
paper_dcterms_spatial	type: string include_in_all: false index: not_analyzed
paper_dcterms_temporal	type: string include_in_all: false index: not_analyzed
text_content	type: string term_vector: with_positions_offsets_payloads

Table 6: Texcavator's data mapping for the KB dataset

Finally, ElasticSearch can let Apache Lucene store entire `term_vector`s for a specific field. A term vector stores all the terms contained in a document along with the frequency at which it occurs. These vectors are useful for generating word clouds, which is discussed in Section 8. This mapping also makes sure that for each term the positions, offsets and payloads are stored, but this data is not used by Texcavator.

5.3 Query language

ElasticSearch offers a simple, but extensive query language [17]. This section explores possibilities of this query language and shows the queries composed by Texcavator's current users.

The query language allows the user to provide a single search term, but also allows defining a boolean relation between multiple terms using the `AND`, `OR` and `NOT` keywords. If a user specifies several terms without keywords (e.g. `pearl white`) these are interpreted by default as being in an `OR` relation (e.g. `pearl OR white`). Users can group certain terms together using brackets, creating more complex queries. The user can group terms together using double quotes to explicitly specify their order. Table 7 lists the usage of these options within the Texcavator queries and shows user query examples.

	Times used	**Example**
Single term	103 (37%)	`banaan`
Boolean operators	158 (56%)	`richter AND aardbeving`
Brackets	58 (21%)	`(film OR bioscoop) AND hollywood`
Double quotes	67 (24%)	`"pearl white"`

Table 7: Examples of user queries using a single term, boolean operators and grouping terms by brackets or double quotes.

Texcavator executes the query strings solely on the title and text content of the documents. However, the query language allows to override this behavior by defining the field the query should match. One user uses this in four queries to only search within article titles (e.g. `article_dc_title: Kunst`).

ElasticSearch also allows the usage of wildcards, where `*` denotes zero or more unknown characters and `?` a single unknown character. In addition, ElasticSearch allows fuzzy searches [34]. A fuzzy search searches for terms similar to the provided terms. For example, a query of *cinema~2* matches all terms similar to *cinema* by allowing a maximum Demerau-Levenshtein distance [7] of two changes. A change is defined by an insertion, deletion or substitution of a single character, or swapping two adjacent characters. As Demerau initially suggested, this can be used to allow for human spelling errors, but in the use-case of Texcavator it can be used to allow for OCR errors. Fuzzy search is used by two users in 12 queries, but there are some queries using wildcards which could be simplified using a fuzzy search (e.g. *f?lm OR r?lpr?nt* versus *film~1 OR rolprent~2*).

The query language offers one promising feature, namely regular expressions. If it is known which OCR errors are most likely to be made (e.g. an `i` is recognized as an `l`, a (semi)colon or a `1`), a regular expression can be used to match exactly these mistakes (e.g. */f[il\:\;1]lm/*). Unfortunately not a single user uses this feature.

5.4 Query filters

Aside from the query language, ElasticSearch supports a broad variety of query types which are listed in the documentation [17]. Texcavator only uses *'boolean filtered query string'* queries. *Query string* queries are composed of a query string, written in the query language as described in the previous section. This section explains the usage of *boolean filtered* queries.

In general, a *filtered* query consists of some query with a filter attached to it. ElasticSearch applies the filter before executing the query. The query is only executed on the subset of documents passing through the filter. If the filter returns a small subset, this is beneficial for the total query response time since the query logic only has to be tested against few documents. ElasticSearch automatically caches the filter results in case it can be reused for other queries [17].

In a boolean filtered query, the filter is defined by any (combination) of three attributes: `must`, `must_not` and `should`. An attribute is assigned simple logic, for example matching a specific fields value or having a field value within a certain range. For example, a filter limiting `paper_dc_date` between 1900 and 1950 is created as follows:

```
'filter': {
  'bool': {
    'must': [
      {
        'range': {
          'paper_dc_date': {
            'gte': 1900,
            'lte': 1950
          }
        }
      }
    ]
  }
}
```

Documents should comply with each logical expression in order to pass through the entire filter. For the `must` attribute holds that only documents matching the logic, can pass through the filter. Vice versa for the `must_not`

attribute. The `should` attributes allows for less strict logic, for example by requiring documents to satisfy at least 1 of multiple defined `should` attributes. Only the `must` and `must_not` attributes are used by Texcavator.

The boolean filter in Texcavator's queries consists of three fields, namely `paper_dc_date`, `article_dc_subject` and `paper_dcterms_spatial`. The two latter fields are not analyzed by ElasticSearch, as discussed in Section 5.2, and they therefore act as tags.

6 Data import optimization

Not all ElasticSearch settings can be changed after the data has been indexed (e.g. the number of shards within an index). In order to benchmark different configurations, the data should be imported multiple times, preferably as fast as possible. This section covers research question (1) How should ElasticSearch be configured in order to obtain a high indexing throughput?

The current import method is analyzed in Section 6.1. After this, an alternate import method is proposed and compared in Section 6.2.

6.1 Importing XML

Currently the XML data is imported using a Python script which reads directly from the gzipped XML files and loops over each document separately. For each document it extracts specific data fields to save in a (Python) dictionary, which is then fed into ElasticSearch-py [46]. ElasticSearch-py is connected to the ElasticSearch cluster and inserts the document using the API endpoint. Parsing XML files is an inherently CPU intensive task with a deterministic outcome. Repeatedly importing the same documents is required for this research and therefore repeatedly parsing the XML files seems a waste of time.

6.2 Importing JSON

ElasticSearch supports several methods for importing data, including a bulk API [13]. This API requires a JSON file containing the document data for multiple documents. Converting the XML files to such JSON files is feasible and only has to be performed once, since the JSON files can then be stored

on disk. These files can be reused after the cluster is reconfigured. Because this API call allows multiple documents to be sent at once, this will also improve performance in terms of communication overhead.

6.2.1 Generating JSON files

The JSON files are easily generated by modifying the existing Python script, forcing it to write all the parsed contents from the XML file, containing multiple documents, into a JSON file instead of sending it to ElasticSearch. JSON files for bulk requests require to be composed in a specific manner. Each document requires two consecutive lines in the JSON file. The first line denotes metadata about the document (i.e. id, type and index) and the second line should contain all the document data itself. Since the bulk API can be called on a specific index and document type [13], only the document id has to be defined in the metadata line. The JSON files are therefore formatted as follows:

```
{"index":{"_id":"ddd:010042315:mpeg21:a0001"}}
{"paper_dc_date": "1831-08-12", "paper_dcx_recordRights": ...}
{"index":{"_id":"ddd:010042315:mpeg21:a0002"}}
{"paper_dc_date": "1831-08-12", "paper_dcx_recordRights": ...}
{"index":{"_id":"ddd:010042315:mpeg21:a0003"}}
{"paper_dc_date": "1831-08-12", "paper_dcx_recordRights": ...}
```

As mentioned in Section 5.1, the dataset consists of 3545 gzipped XML files. Table 8 shows the conversion times from XML to JSON for the entire dataset using different numbers of threads. Since the machine only consists of 16 logical cores, no further speed up is achieved when using more than 16 threads. The conversion task is CPU bound since only a small amount of data is read from disk (and written to it) considering the tasks total execution time.

# threads	**time**
8	8h45m
16	7h03m
32	7h02m

Table 8: Conversion time converting the entire dataset from XML to JSON for several numbers of threads.

6.2.2 Importing JSON files

The bulk API can be accessed from both ElasticSearch-py [47] as well as directly using `curl` [6]. Since using `curl` saves an extra step of processing the data, this method is preferred above ElasticSearch-py. `parallel` [62] is used to parallelize the `curl` requests.

Several ElasticSearch configurations have been tested in order to achieve a high indexing throughput. The final settings can be found in Table 9. These settings are partially based on a gist by reyjrar [59] and will be discussed in detail in the following paragraphs. Along with these settings ElasticSearch is given a heap size of 32GB and `parallel` is using three threads to push the data from one fat machine to the fat machine hosting the ElasticSearch single node cluster. This dual machine set up is used to provide a fair comparison when comparing it to XML. In the case of XML, running everything on a single node would be extra disadvantageous. This is the case because the processing power used for conversion, would negatively impact ElasticSearch' performance.

key	**value**	**default**
indices.store.throttle.type	merge	merge
indices.store.throttle.max_bytes_per_sec	200mb	20mb
index.merge.policy.segments_per_tier	60	10
index.merge.scheduler.max_thread_count	1	3
indices.memory.index_buffer_size	30%	10%
indices.memory.min_shard_index_buffer_size	12mb	4mb
indices.memory.min_index_buffer_size	96mb	48mb
index.refresh_interval	30s	1s
bootstrap.mlockall	true	false

Table 9: ElasticSearch settings achieving a high indexing throughput.

ElasticSearch suggests that setting `indices.store.throttle.type` to `none` obtains a higher indexing throughput [41]. This makes sense, since not throttling the disk writes results in high throughput. However, it also clogs up the node when indexing a lot of data with the amount of merges occurring. Therefore `indices.store.throttle.type` is kept at `merge` and the throttling speed is set high, but it will throttle.

Setting `index.merge.scheduler.max_thread_count` to 1 ensures there will not be any merge operations running in parallel, fighting over IO band-

width. Setting `index.merge.policy.segments_per_tier` higher than default (10) makes sure more segments are allowed inside a shard, resulting in less (but larger) merges.

Because the cluster does not have to serve any search query requests while indexing, larger buffers than normal are assigned to the indexers. The value of 30% for `indices.memory.index_buffer_size` seems arbitrary, but setting it to 50% degraded the performance. The `refresh_interval` has been increased, since newly indexed documents do not have to be available for search instantaneously. Finally, `bootstrap.mlockall` will make sure ElasticSearch' memory will not be swapped out.

6.2.3 Import performance

The performance is measured over three different collections of documents, namely: the 2009 subset plus the 2010 subset, resulting in about 5 million documents, the 2011 subset containing about 26 million documents and the entire dataset which consists of about 102 million documents. The years denote the year of retrieval, as described in Section 5.1. Each test is executed in threefold and the average results can be found in Table 10.

These settings clearly improve performance on data import, as shown in the first two rows of Table 10. This significantly speeds up deploying new cluster configurations for benchmarking. Further improvements on these settings may be possible.

The final two rows in Table 10 compare the time required to import JSON or XML on ElasticSearch' default settings. The maximum HTTP request size has been increased to 2GB in order to support the JSON files in the bulk API. It must be noted that the XML import using default settings started generating HTTP read timeouts before it even completed the entire import. Nevertheless, the entire dataset was imported faster using JSON files than the partial import accomplished by importing XML files.

	2009+2010 (5M)	**2011 (26M)**	**2009-2013 (102M)**
JSON	26m5s	3h01m44s	19h51m30s
XML	1h41m56s	9h05m48s	45h35m16s
JSON default	1h48m56s	n/a	n/a
XML default	2h29m21s+	n/a	n/a

Table 10: Comparison between the average import times over three runs for JSON and XML over three different datasets. Each benchmark used optimized settings and default settings as comparison.

Table 10 shows that importing the JSON files is more than 50% faster than the XML import method. For a fair comparison, the time required to generate the JSON files must also be taken into account. Table 11 shows the times per document collection and Figure 3 plots a time comparison plot, including the conversion times. Even when conversion times are added, the JSON import outperforms the XML import. Without considering the conversion times, importing JSON files is more than 50% faster than the XML method. Taking the conversion counts into account still results in a 40% speed up in favor of importing the JSON files.

dataset	**time**
2009+2010	25m26s
2011	1h26m54s
2009-2013	7h03m

Table 11: XML to JSON conversion times for several subsets using 16 cores.

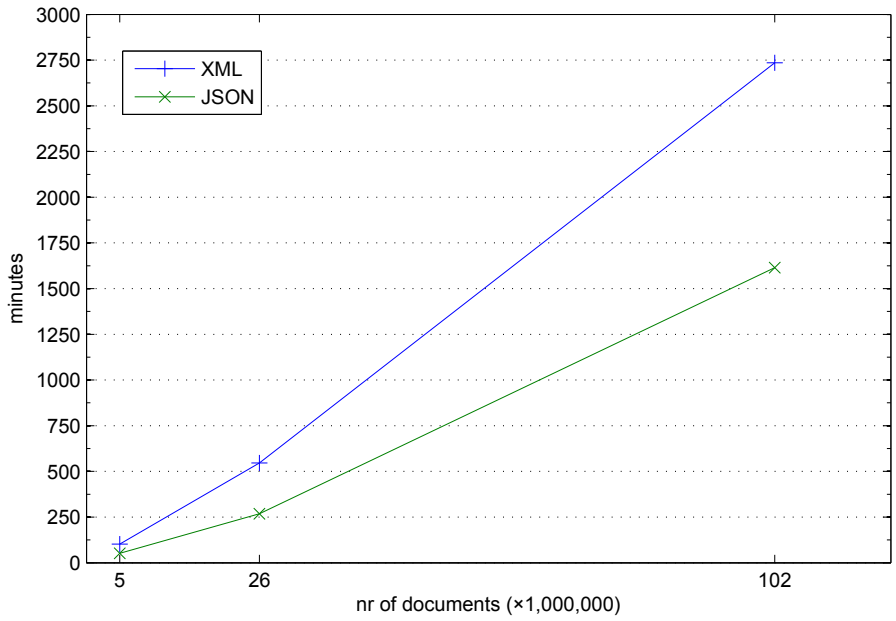

Figure 3: Import comparison between XML and JSON, including conversion times (XML to JSON) on three different datasets averaged over three runs.

7 Search performance optimization

Texcavator allows data exploration by letting its users define search queries. Improving ElasticSearch' response time for those queries will result in a better user experience for Texcavator. This section assesses whether and how to improve ElasticSearch' query response times.

ElasticSearch' count API [16] is used to execute all the queries. This call returns how many documents match the specified query and how long it took to perform this search. Texcavator also uses this API to determine how large the result set of a specific search query is.

This section covers several options on optimization offered by ElasticSearch and answers the research questions (2) Which ElasticSearch settings improve search query response times for given user queries? and (3) How do shards and replicas influence search query response times for consecutive as well as concurrent queries?

Section 7.1 assesses whether the documents are equally distributed over the available shards. In section 7.2, difficulties revolving around caching are explained. The optimize API (Section 7.3), data partitioning (Section 7.4) and changing the number of shards or replicas (Section 7.5) are evaluated on how they affect search performance.

7.1 Document identifiers

As explained in Section 2.4, a unique identifier is used to route a document to a single shard. In the KB dataset each document already has a unique id assigned to it (e.g. `ddd:010042315:mpeg21:a0001`).

Whether the data is equally distributed over all shards depends on the hashing algorithm, the provided identifiers, as well as the number of shards used. To make sure the data is equally distributed, all identifiers for the 2012 dataset are passed through the Murmur hashing algorithm [28] as ElasticSearch would during indexing.

Figure 4 shows the document distribution over time for a 16 shard index by plotting the size of the biggest as well as the smallest shard. Both values are expressed as the percentage of imported documents within said shards. Note, that it can change over time which shard contains the most or least documents.

The final difference between these two shards is less than 0.01%, or about 5 thousand documents on 57 million documents. This difference is achieved

for each possible configuration of 2 to 32 shards. There are no anomalies in terms of, for example, an odd number of shards resulting in unevenly distributed documents. Therefore document identifiers do not require additional optimization.

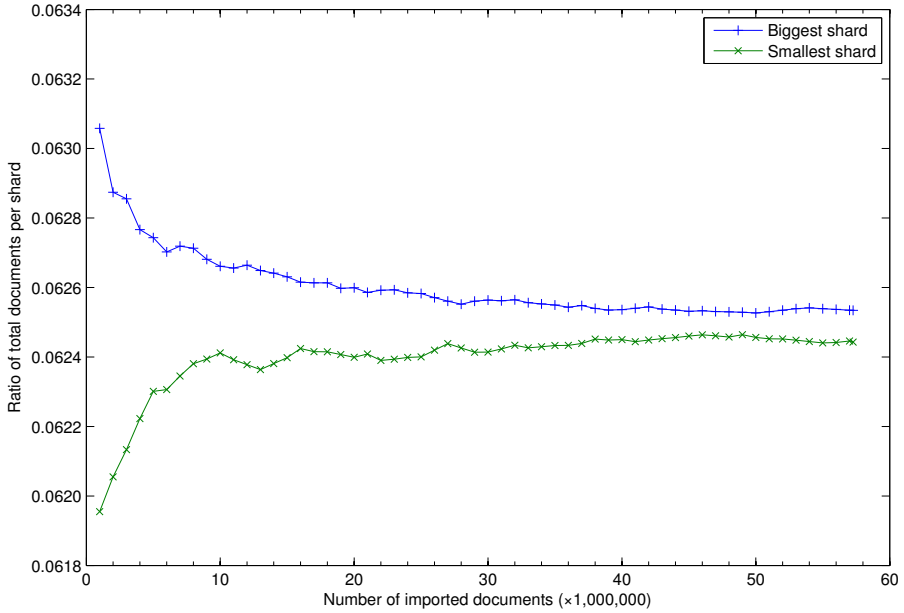

Figure 4: Document distribution during indexing for the biggest shard versus the smallest shard on a total of 16 shards.

7.2 Caching

Any system built to handle large datasets needs to cache data wisely since spinning disks are slow and RAM is limited. However, when benchmarking such a system the caching behavior can become difficult to take into account. Repeatedly executing the same query in a tight loop on an ElasticSearch cluster will yield faster responses due to caching. This makes it harder to obtain a sensible estimate for the performance in a production environment.

ElasticSearch offers an API call to clear all caches [14] and several settings to control caching behavior. Table 12 lists these settings along with a short description. Using the API call to clear the caches after executing a query

and then executing the same query again resulted in a faster execution. The same behavior occurred when configuring the cluster not to cache. This suggests that there is more to caching than the user can control.

attribute	description
indices.cache.filter.size	Space allocated for cached query filters
index.cache.filter.max_size	Space allocated for cached query filters per index
index.cache.filter.expire	Time a cached query filter will be kept
indices.fielddata.cache.size	Space allocated for the fielddata cache
indices.fielddata.cache.expire	Time the fielddata cache will be kept
index.cache.query.enable	Whether a complete query can be cached

Table 12: ElasticSearch cache settings, where each attribute is set to zero or false.

Besides ElasticSearch' internal caching there is Linux' IO caching. ElasticSearch suggests not to use more than half of the RAM available on the machine [20], since ElasticSearch and Lucene heavily depend on IO caching. To reduce extreme performance gains by IO caching each index will be triplicated and the queries will be executed on this in a round-robin fashion. It would be impossible for Linux to cache *all* of it, due to the sheer size of the indices.

To reduce caching based on the query itself, all the queries are shuffled before they are executed. This way similar consecutive queries will not always be executed consecutively. The total query set is executed 8 times and the median execution time is used as the representative execution time for each query. Median times are used since the disabled caching causes outliers more frequently.

7.3 Optimizing an index

ElasticSearch offers an API call to optimize an index [31]. Optimizing an index merges all segments within each shard (the Lucene index) into a single segment as explained in Section 2.6. Since the KB dataset is a static dataset it is highly recommended to optimize the index [36].

Optimizing a dataset of this size requires time and because the import settings have been tweaked to do less merging during indexing, this process takes even longer. Table 13 shows for several configurations the time required to optimize all segments within each shard into a single segment per shard. For a single shard this results in a single large segment, containing all

document (meta)data, which takes a lot of time to compose. Using multiple shards (4 or more) takes considerably less time to optimize. However, the difference between using 4 or 16 shards is negligible. This is presumably due to the available IO bandwidth, making it more difficult to optimize all the shards in parallel.

dataset	**#shards**	**time**
2012	1	5h40m
2012	4	4h36m
2012	8	4h47m
2012	16	4h21m
all	16	12h01m

Table 13: Time required to perform the optimization of merging all segments within each shard into a single segment.

Each segment consists out of several data and meta-data files and reducing the number of segments therefore reduces the total number of files Lucene has to read. This also speeds up restarting an ElasticSearch instance where shards are loaded from disk almost instantaneously instead of taking several minutes.

Figure 5 shows the execution time per query on the 2012 dataset with 8 shards before and after optimization. The optimization results in a median query execution time that is 40 times faster than before optimization. This resulted in each query executing in under a second, instead of almost all queries taking far over a second to complete. The median ratio between the median time and standard deviation stayed the same after optimization. The query peaking at almost 15 seconds after optimization, is the query `*gif`. Such a query cannot be solved quickly by simply looking at the inverted index. For each term in the inverted index, it has to be checked whether it ends on *gif* or not.

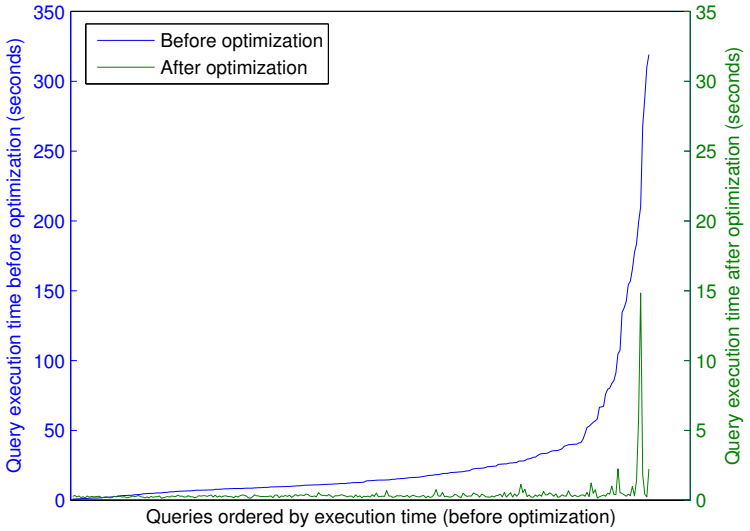

Figure 5: Execution time comparison before and after optimization. Note the two separate vertical axis. The queries are ordered based on their execution time before optimization. The query still peaking after optimization (15 seconds) is the **gif* query.

7.4 Data partitioning

A common method [57] [56] to increase performance in relational databases is to partition the data by disjoint time intervals (e.g. dates) or unique/distinct values (e.g. document types). By sorting the data beforehand, a query only has to be executed on a subset of the data, increasing the overall performance. This is comparable to ElasticSearch' filters. ElasticSearch suggests using an index per time frame (e.g. a year or month) when using ElasticSearch for logging. This is mainly suggested to keep the indices at a reasonable size [42].

As discussed in Section 5.4, there are multiple fields available within each document and the three most evident fields to partition on are: `paper_dc_date`, `paper_dcterms_spatial` or `article_dc_subject`. Resulting in partitions by date range, geospatial distribution or document type respectively.

The 2012 dataset on 16 shards is partitioned on `article_dc_subject` to test the hypothesis that data partitioning improves query response times. Table 14 shows the benchmark configuration in terms of shards and documents per index/partition. Each set of indices is optimized as described in Section 7.3 and tripled to reduce caching speedups as explained in Section 7.2.

Table 15 reports the performance for queries **excluding** specific article types. Queries which do not exclude any type perform worse than without partitioning (a speedup of 0.90). This in contradicts ElasticSearch' remark: *"Searching 1 index of 50 shards is exactly equivalent to searching 50 indices with 1 shard each: both search requests hit 50 shards."* [27]. However, as expected, queries excluding certain types do gain performance (a speedup of 1.24 to 1.88).

Partition	**#shards**	**#documents**
Article	8	37,634,553 (66%)
Advert	4	17,306,938 (30%)
Illustration	2	1,188,303 (2%)
Family	2	1,138,653 (2%)

Table 14: Subject partitioning configuration in terms of shards and documents.

Excludes	**#queries**	**Speedup**
None	85	0.90
Article	27	1.88
Advert	146	1.24
Illustration	165	1.33
Family	177	1.31

Table 15: Speedups on partitioned data (over 4 indices) versus a single index (16 shards). The speedup is calculated per individual query and the average of these speedups is reported and categorized on which partition is **excluded** from the search.

7.5 Shards and replicas

As claimed in Section 2.4, using multiple shards improves query performance and allows the cluster to scale horizontally. These claims are evaluated for a single node cluster as well as for a multiple node cluster. Section 7.5.1 covers the case of a single node cluster and Section 7.5.2 the case of a multiple node cluster consisting of three machines.

7.5.1 Single node cluster

Figure 6 compares several configurations namely consisting of 1, 4, 8, 16 or 32 shards. The collection of all queries is executed 8 times per configuration

in order to calculate a median execution time. These median times are used to calculate the speedup of using more shards compared to using a single shard. Table 16 shows the median of these speedups per shard configuration. This shows that using more shards increases performance until the number of shards exceeds the number of available cores. Exceeding the number of available cores degrades the query performance drastically in case of a single node cluster.

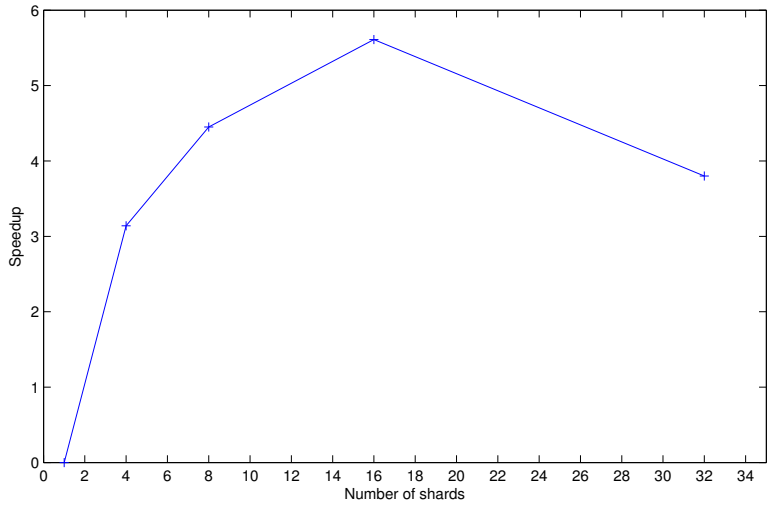

Figure 6: Speedup comparison between multiple shards and a single shard on a single node cluster.

#shards	**speedup**	**median time**
1	0.00	1.26s
4	3.14	0.42s
8	4.45	0.29s
16	5.61	0.22s
32	3.80	0.35s

Table 16: Speedup comparison between multiple shards and a single shard on a single node cluster.

7.5.2 Multiple node cluster

A multiple node cluster is set up as described in Section 4. Importing the data is done on a single node on a fat machine. Once the data is imported, two additional ElasticSearch instances are deployed (each on a default machine). ElasticSearch then automatically starts distributing the data from the fat machine to the default machines, as explained in Section 2.4. ElasticSearch' heap size is decreased from 32GB to 12GB for both the fat and default machine, since the default machines only consist of 24GB of RAM.

Even when transferring the data over InfiniBand, the reallocation takes time. Table 17 shows cluster settings which improved the transfer speeds from the default of 80MB/s to 120MB/s. There is a lot of bandwidth available to each machine, due to the available InfiniBand interface. Therefore the files can be sent in large chunks (`indices.recovery.file_chunk_size`) and do not have to be compressed (`indices.recovery.compress`). Multiple file transfers may also happen concurrently. ElasticSearch offers separate stream pools for small files (smaller than 5mb [40]) and other files. The excessive number of small file streams ensures all small files are sent instantaneously on reallocation. This relieves the disks from having to do small reads for small files while transferring the larger files.

key	**value**	**default**
indices.recovery.file_chunk_size	200mb	512kb
indices.recovery.compress	false	true
indices.recovery.concurrent_streams	8	3
indices.recovery.concurrent_small_file_streams	16	2
indices.recovery.max_bytes_per_sec	200mb	40mb
cluster.routing.allocation.node_concurrent_recoveries	8	2
cluster.routing.allocation.cluster_concurrent_rebalance	8	2

Table 17: ElasticSearch settings achieving reasonable data transfer.

On a multiple node cluster the number of replicas can be increased. To make the results for a multiple node cluster comparable to the single node cluster, the queries are executed in the same manner as described in Section 7.2.

Figure 7 compares the single node performance to the multiple node performance for several replica settings. The figure is sorted on the speedup obtained by using two replicas. This shows that using multiple replicas does *not* improve the performance for consecutively executing single queries, be-

cause higher speedups are achieved using a single replica or no replicas at all. This result contradicts ElasticSearch' claim that *"you can increase search performance by increasing the number of replicas"* [35]. However, this claim is made in context of a *'search heavy'* index. This setup could be argued not to be *'search heavy'* since only a single user is executing queries in a consecutive fashion. It is unknown how ElasticSearch defines a *'search heavy'* index.

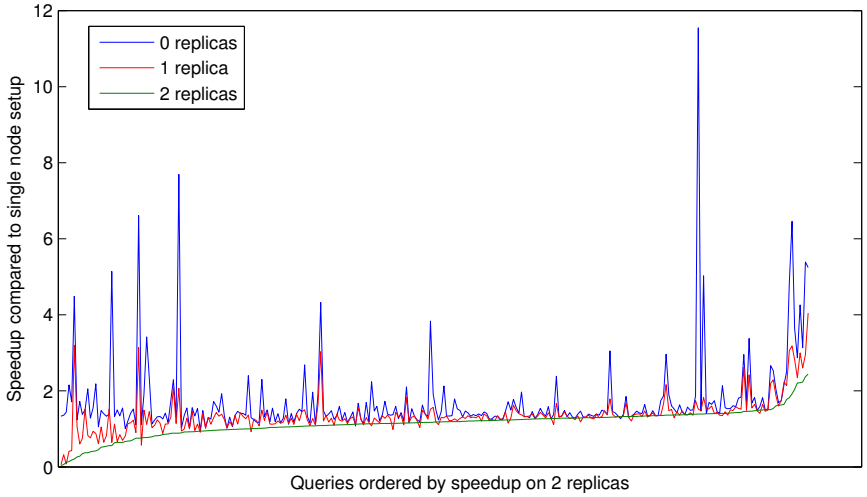

Figure 7: Median speedup per query on a three node cluster using 0, 1 or 2 replicas, compared to a single node setup. The queries are sorted by their speedup on 2 replicas. This shows that higher speedups are achieved using less replicas.

7.5.3 Multiple node cluster - Load tests

The previous section showed that using replicas does *not* increase query response times when single queries are executed consecutively on the cluster. However, in a real-life scenario multiple users will try to use the system at the same time. Several load test are performed to check whether increasing the number of replicas influences the query response times when the cluster is under heavy load.

A load test simulates multiple users by starting a thread per 'user'. Each user loops through the 280 provided (shuffled) queries and sleeps randomly between 3 and 10 seconds before executing each query. In addition, each user randomly mutates its query before it is sent, to make sure ElasticSearch does not use a fancy form of caching when identical queries are sent by different users. One user is sending unmodified queries, these execution times are used to calculate the performance for an average user.

Queries are mutated by changing lower case letters within the query string into different lower case letters. Only lower case letters are changed in order to avoid changing brackets, boolean operators or other special tokens used by the query language. Each query will have 10% of its characters changed. For example, changing `Amerika OR New York` to `Amerika OR Ngw York`. More examples of mutated queries are listed in Appendix C.

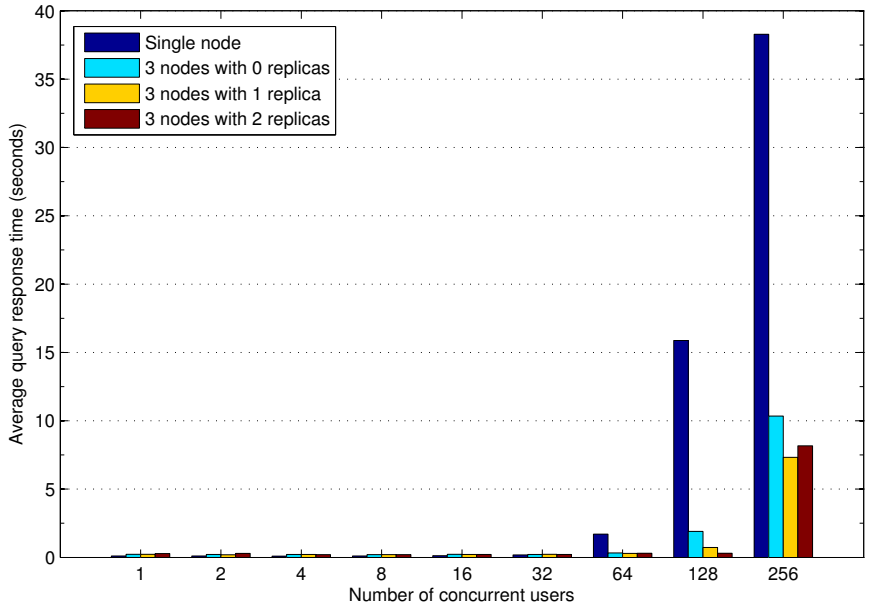

Figure 8: Load test on a single node cluster versus a three node cluster with all the possible number of replicas, showing a definite performance gain when using replicas.

Figure 8 presents the results from several performed load tests. Two important observations can be made from this figure. First, the search response

times for a single node cluster increase drastically at a load of 64 users, while a multiple node cluster can handle this load with ease. Second, at 128 users it shows that it is beneficial to increase the number of replicas on a multiple node cluster. Since not using replicas degrades the performance drastically while the cluster using two replicas is unaffected by the load of 128 users.

At a load of 256 users, the search response times also drastically increase on the multiple node cluster, regardless of the number of replicas. This increase is not due to the fact that a single machine is simulating all 256 users. Most of the time the user is idle by either waiting to receive a response or to send a new request. This is confirmed by spreading the 256 users over two machines, each simulating 128, resulting in similar increases in response times.

8 Optimizing word cloud generation

This section covers Texcavator's functionality of generating a word cloud over an entire query result set. Generating a word cloud is achieved by adding up the frequencies of all words in all documents in the result set. If the result set is large, generating a word cloud becomes resource intensive and takes longer than the user is willing to wait. The word clouds are capped to solely contain the 100 most frequent words, giving researchers insight in potentially related terms.

Section 8.1 explains the method currently used by Texcavator to generate word clouds. Since this method is said to be resource intensive and can overload the system, this method limits the total number of users being able to do research at the same time. Two alternative methods are proposed, both written in Java. One uses the Java ElasticSearch client (Section 8.2) and the other uses the ElasticSearch Hadoop library (Section 8.3).

The performance for each method is measured for all queries having a result set of less than 50.000 documents. This is a reasonable threshold since on average the queries return 40.000 documents on the 2012 dataset, with a median of just 2.000 documents.

8.1 Python implementation

The current word cloud implementation is written in Python like the rest of Texcavator. Once a user submits a request to generate a word cloud for a certain query, this request is posted to a Celery server [5]. The Celery server executes the appropriate Python function and keeps the user updated on the word cloud generation progress. The word cloud generation consists of two (interleaved) steps. First a (partial) list of document identifiers is composed (Section 8.1.1) which are then fed into ElasticSearch' multi termvectors API [26] (Section 8.1.2). This repeats until all documents from the result set are included in the word cloud.

8.1.1 Document identifier retrieval

The Python method, `generate_tv_cloud`, takes the user query and adds the `from` and `size` attributes to it. These attributes allow to loop over the entire search result, by defining a start position (`from`) and how many documents should be returned (`size`) [19]. Texcavator loops over the result set per 1000 documents and ElasticSearch is set to only return the document identifier (and type) per result [18].

An alternative method to loop over a document collection, is to define `scroll` [39]. This attribute allows to scroll over the entire search result, without having to recalculate a `from` attribute. Because it is unimportant in which order the termvectors are retrieved, the `search_type` attribute can be set to `scan`. This disables sorting the result set on relevance, speeding up the process. The only relevant note for Texcavator on this method is that the `size` attribute is now *per shard* instead of the total number of documents it should return, due to the optimizations used by ElasticSearch. A cluster set up of 16 shards is used and 60 documents are requested per shard, resulting in 960 documents per request.

Figure 9 shows the time required to loop through the entire result set per query. This shows that scrolling over a scanning result set is beneficial, since it is on average almost three times faster.

8.1.2 Multi termvectors API

ElasticSearch offers a multi termvectors API [26], capable of retrieving multiple termvectors given a list of identifiers. Texcavator retrieves the termvectors for the `text_content` and `article_dc_title` fields, based on the (partial)

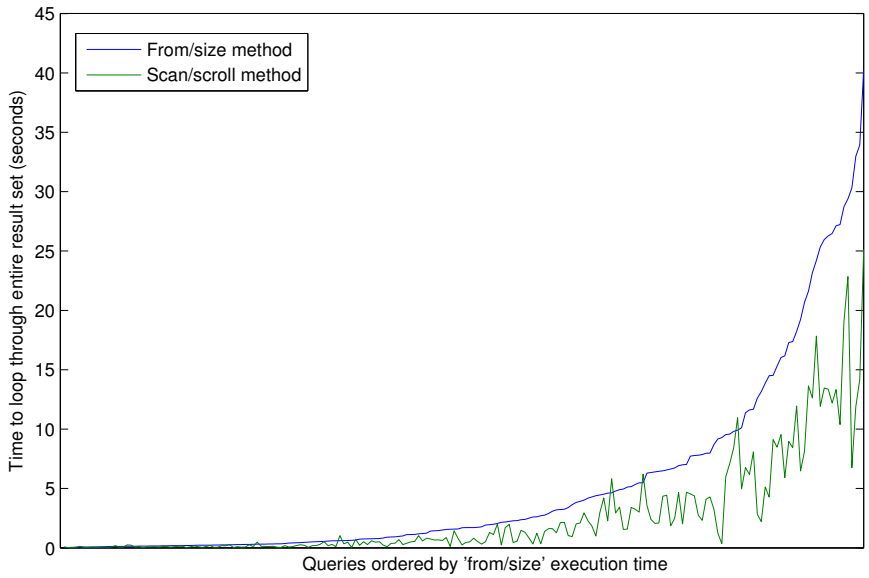

Figure 9: Comparison using `from` and `size` versus `scan` and `scroll` over all queries. The queries are ordered by execution time using the `from` method.

list of document identifiers. These terms and frequencies are stored and the terms and frequencies from each consecutive multi termvectors request are merged with the existing values. Then the top 100 is extracted and returned to the user to visually generate the word cloud.

Additionally, Texcavator allows its users to define a list of stop words, which should be excluded from the word cloud. For example, articles (*de*, *het* and *een*) do not provide useful research statistics when processing this many documents, since it makes sense that articles occur many times in almost every document. Excluding these words from the word cloud leaves room for other words to reach the top 100. This filtering is achieved during the merge step, but is not executed during the performance measurements.

Figure 10 shows the trend line based on the data points acquired by executing the user queries. This shows a correlation between the number of documents in the result set and the time it takes to generate the word cloud. With a median time of 16ms per document, it takes a long time to actually retrieve the termvectors from ElasticSearch.

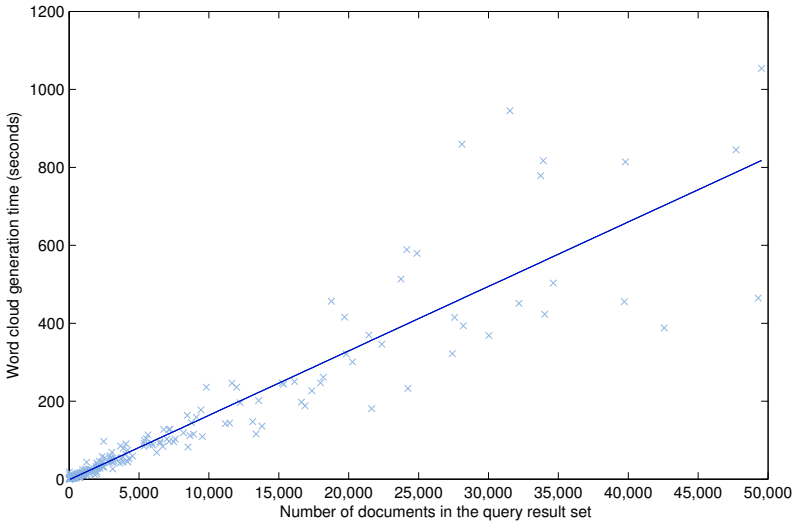

Figure 10: Correlation between word cloud generation times for Python method and the number of documents in the result set, visualized by a trend line.

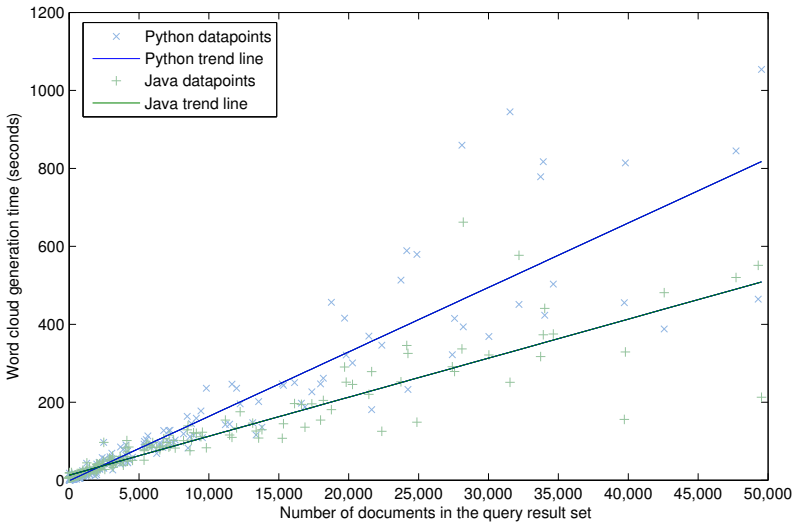

Figure 11: Comparison between Python and Java implementation, showing both datapoints and trend lines.

8.2 Java implementation

The ElasticSearch-py client uses the API endpoints, which are accessible over HTTP. This requires converting the Python objects into JSON strings when sending requests or vice versa when receiving replies. The ElasticSearch Java client [29] however, actually joins the internal cluster communication protocol, since ElasticSearch itself is also written in Java. This takes away inefficiencies when it comes to transferring data back and forth.

The same methodology is used as discussed in Section 8.1. Document identifiers are retrieved using a `scan` and `scroll` method and the termvectors are retrieved using a `MultiTermVectorsRequest` object. These termvectors are combined and the top 100 is returned.

Figure 11 compares the Python implementation to the Java implementation and shows that using the native protocol improves performance. However, it also shows that it takes too much time to join the cluster for less than 2,000 documents. Below that threshold it is faster to use the API.

8.3 Mapreduce implementation

Generating a word cloud is essentially doing a word count task, which is Hadoop MapReduce's [50] classic example. ElasticSearch offers an ElasticSearch Hadoop library (ElasticSearch-hadoop) [30] which assists users in directly accessing ElasticSearch data from within Hadoop. For example, a MapReduce task can be started based on a query where each query result is fed into a mapper.

A Hadoop cluster is launched on the same (default) machines as the triple node ElasticSearch cluster is running. This is preferred over using separate machines due to data locality. Because the ElasticSearch-Hadoop library cannot access the termvectors, the MapReduce implementation uses the entire documents.

The MapReduce implementation is based upon Andrea Iacono's top-n terms examples [51], resulting in an optimized MapReduce application using a Mapper, Combiner and a single Reducer. Andrea has an inefficiency in his Reducer, requiring more memory than necessary. This is solved by prematurely throwing terms out of the Reducer's memory when they surely will not make the top-n.

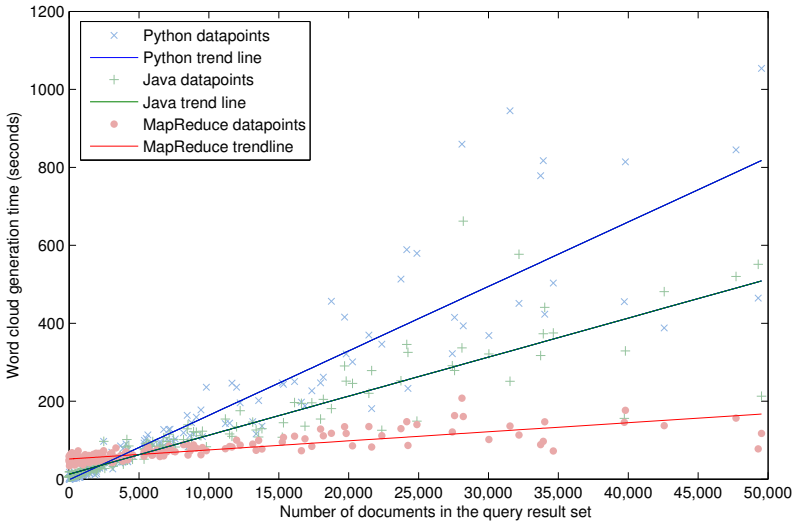

Figure 12: MapReduce implementation compared to Python and Java implementations, showing both datapoints and trend lines.

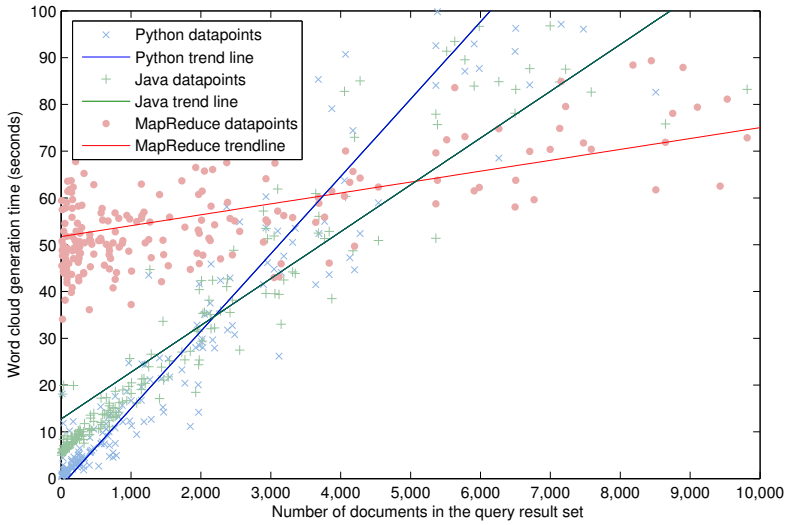

Figure 13: MapReduce comparison zoomed in on the area of interest.

Figure 12 shows that MapReduce requires a lot of time initializing and is therefore not beneficial for result sets of less than 5000 documents. However, if the document result set contains more than 5000 documents, the MapReduce implementation is cleary faster. Surprisingly, MapReduce's execution time barely increases when there are many more documents in the query result set. This means that data locality is handled well by ElasticSearch-Hadoop. A zoomed version of Figure 12 is available in Figure 13, clearly showing the thresholds of 2,000 documents on Python versus Java and 5,000 documents on Java versus MapReduce.

The most important note here is that essentially creating termvectors from scratch outperforms retrieving the existing termvector data. Since that is essentially what the MapReduce implementation does, compared to the Python and Java implementations.

9 Conclusions

This research answered many questions, amongst which the research questions: (1) How should ElasticSearch be configured in order to obtain a high indexing throughput? (2) Which ElasticSearch settings improve search query response times for given user queries? (3) How do shards and replicas influence search query response times for consecutive as well as concurrent queries? (4) How can Texcavator's word cloud generation method be sped up without degrading other users' experiences? This section discusses each of the questions by summarizing their answers and assesses whether future work is required to obtain more insight in ElasticSearch' performance.

In order to obtain a high indexing throughput, Section 6 assessed the current data import method. The current import method converted XML on-the-fly to Python dictionaries and inserted them one-by-one into ElasicSearch. The conversion task was CPU intensive, with a deterministic outcome and therefore seemed a waste of CPU cycles. Using JSON files along with ElasticSearch' bulk API and custom cluster settings resulted into a speedup of 40%-50%, depending on whether conversion times were taken into account or not. With an import time of almost two days, the achieved speedup is beneficial for this research. However, the production cluster can also benefit from this, since completely re-indexing can now be achieved with less downtime.

Search query response times for given user queries were mainly improved by using the optimize API call, as discussed in Section 7.3. After using the optimize API, queries executed 40 times faster than before optimization. This is partially due to the fact that less merges happen during indexing to speed up the data import, resulting in search overhead. Data partitioning was evaluated in Section 7.4. Partitioning the data provided a 1.24 to 1.88 speedup for queries excluding partitions from search. However, queries on the entire dataset would execute slower than normal, at a speedup of 0.90. The document identifiers (Section 7.1) required no further optimization since the Murmur hashing algorithm distributed the documents evenly over the available shards using the existing identifiers.

Changing the number of shards or replicas influenced the overall query response times in several ways. Section 7.5.1 showed that using as many shards as there are cores available on the machine is beneficial. Section 7.5.2 showed unexpected behavior for the number of used replicas. Using more replicas in the case of consecutively executing single queries resulted in higher query response times. However, in Section 7.5.3, the load tests showed that using multiple replicas allows the cluster to handle more concurrent users. Future research can confirm whether using multiple machines with less RAM and processing power is beneficial in the case of handling many concurrent users.

Finally, Section 8 answered the question on how to speed up word cloud generation. The Java implementation, using the ElasticSearch client for Java, outperformed the Python implementation. This was due to the fact that the Python library for ElasticSearch uses the HTTP API and the Java client actually joins the internal cluster communication protocol, which saves on generating JSON requests/responses. Using MapReduce resulted in an immense speedup (4 times) compared to the Python implementation. This is worrying since the MapReduce implementation created all the termvectors from scratch in order to compose the word cloud. Whereas Python tried to retrieve existing termvectors from ElasticSearch.

Next to the research questions, other interesting findings came to light. Section 5.3 showed that some users are unaware of the existence of fuzzy search and the possibility to use regular expression to (partially) overcome the OCR errors. This suggests that Texcavator's (future) users should be informed better about the powerful possibilities of the rich query language.

Many issues were encountered during this research. The issues revolve around two key points, namely: the dataset (configuration) and Elastic-Search. Appendix D lists all encountered issues worth mentioning regarding these two key points.

References

[1] Andreas Both, Axel-Cyrille Ngonga Ngomo, Ricardo Usbeck, Denis Lukovnikov, Christiane Lemke, and Maximilian Speicher. 2014. A service-oriented search framework for full text, geospatial and semantic search. In *Proceedings of the 10th International Conference on Semantic Systems* (SEM '14), Harald Sack, Agata Filipowska, Jens Lehmann, and Sebastian Hellmann (Eds.). ACM, New York, NY, USA, 65-72. DOI=10.1145/2660517.2660528 `http://doi.acm.org/10.1145/2660517.2660528`

[2] Apache Lucene, `https://lucene.apache.org/core/`

[3] Apache Lucene - Index File Formats: Segments, `https://lucene.apache.org/core/3_0_3/fileformats.html#Segments`

[4] Apache Lucene - Index File Formats: Inverted Indexing, `https://lucene.apache.org/core/3_0_3/fileformats.html#Inverted%20Indexing`

[5] Celery: Distributed Task Queue, `http://www.celeryproject.org/`

[6] `curl`, `http://curl.haxx.se/download.html`

[7] Fred J. Damerau. 1964. A technique for computer detection and correction of spelling errors. Commun. ACM 7, 3 (March 1964), 171-176. DOI=10.1145/363958.363994 `http://doi.acm.org/10.1145/363958.363994`

[8] The Distributed ASCI Supercomputer 4 (DAS-4), `http://www.cs.vu.nl/das4/`

[9] The Distributed ASCI Supercomputer 5 (DAS-5), `http://www.cs.vu.nl/das5/`

[10] Delpher, http://www.delpher.nl/

[11] ElasticSearch, https://www.elastic.co/

[12] ElasticSearch - Bool Filter, https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-bool-filter.html

[13] ElasticSearch - Bulk Import API, http://www.elastic.co/guide/en/elasticsearch/reference/1.3/docs-bulk.html

[14] ElasticSearch - Clear caches, http://elasticsearch-py.readthedocs.org/en/latest/api.html#elasticsearch.client.IndicesClient.clear_cache

[15] ElasticSearch - Core Types, https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-core-types.html

[16] ElasticSearch - Count API, https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-search-type.html#count

[17] ElasticSearch - DSL Queries, https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-queries.html

[18] ElasticSearch - Fields, https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-fields.html

[19] ElasticSearch - From / Size, https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-from-size.html

[20] ElasticSearch - Heap: Sizing and Swapping, https://www.elastic.co/guide/en/elasticsearch/guide/current/heap-sizing.html

[21] ElasticSearch - How Primary and Replica Shards Interact, https://www.elastic.co/guide/en/elasticsearch/guide/current/_how_primary_and_replica_shards_interact.html

[22] ElasticSearch - Mapping, https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html

[23] ElasticSearch - Mapping: _all field, https://www.elastic.co/guide/en/elasticsearch/reference/1.6/mapping-all-field.html

[24] ElasticSearch - Mapping: Date Format ,https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-date-format.html

[25] ElasticSearch - Mapping: types, https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-core-types.html

[26] ElasticSearch - Multi termvectors API, https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-multi-termvectors.html

[27] ElasticSearch - Multiple Indices, https://www.elastic.co/guide/en/elasticsearch/guide/current/multiple-indices.html

[28] ElasticSearch - Murmur algorithm, https://github.com/elastic/elasticsearch/blob/bfbee383bd2b2e4a928006d1703c22c0a2aae155/core/src/main/java/org/elasticsearch/cluster/metadata/IndexMetaData.java#L250

[29] ElasticSearch - Official Java client, https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/index.html

[30] ElasticSearch - Official Hadoop client, https://www.elastic.co/products/hadoop

[31] ElasticSearch - Optimize, https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-optimize.html

[32] ElasticSearch - Query Phase, https://www.elastic.co/guide/en/elasticsearch/guide/current/_query_phase.html

[33] ElasticSearch - Query String Query, https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html

[34] ElasticSearch - Query String Query - Fuzziness, https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html#_fuzziness

[35] ElasticSearch - Replica Shards, https://www.elastic.co/guide/en/elasticsearch/guide/current/replica-shards.html

[36] ElasticSearch - Retiring Data, https://www.elastic.co/guide/en/elasticsearch/guide/current/retiring-data.html

[37] ElasticSearch - Routing a Document to a Shard, https://www.elastic.co/guide/en/elasticsearch/guide/current/routing-value.html

[38] ElasticSearch - Scale horizontally, https://www.elastic.co/guide/en/elasticsearch/guide/current/_scale_horizontally.html

[39] ElasticSearch - Scroll, https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-scroll.html

[40] ElasticSearch - SMALL_FILE_CUTOFF_BYTES, https://github.com/elastic/elasticsearch/blob/065275443d9954220a004d4dcc5138a5f91444f5/core/src/main/java/org/elasticsearch/indices/recovery/RecoverySettings.java#L76

[41] ElasticSearch - Throttle, https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-store.html

[42] ElasticSearch - Time-Based Data, https://www.elastic.co/guide/en/elasticsearch/guide/current/time-based.html

[43] ElasticSearch 1.7.1 release notes, https://www.elastic.co/downloads/past-releases/elasticsearch-1-7-1

[44] ElasticSearch 1.5.0 release notes, https://www.elastic.co/downloads/past-releases/elasticsearch-1-5-0

[45] ElasticSearch 0.10.0 release notes, https://www.elastic.co/downloads/past-releases/elasticsearch-0-10-0

[46] ElasticSearch-py, ElasticSearch' official Python client, http://www.elastic.co/guide/en/elasticsearch/client/python-api/current/

[47] ElasticSearch-py - Bulk Import, http://elasticsearch-py.readthedocs.org/en/latest/api.html

[48] Netherlands eScience Center, https://www.esciencecenter.nl/

[49] Gormley, C., "Release the clients! Ruby, Python, PHP, Perl", September 24, 2013, https://www.elastic.co/blog/unleash-the-clients-ruby-python-php-perl

[50] Hadoop MapReduce, http://hadoop.apache.org/

[51] Andrea Iacono's MapReduce examples for top N items, http://andreaiacono.blogspot.nl/2014/03/mapreduce-for-top-n-items.html

[52] Koninklijke Bibliotheek, Den Haag, Nederland, https://www.kb.nl/

[53] Oleksii Kononenko, Olga Baysal, Reid Holmes, and Michael W. Godfrey. 2014. Mining modern repositories with elasticsearch. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (MSR 2014). ACM, New York, NY, USA, 328-331. DOI=10.1145/2597073.2597091 http://doi.acm.org/10.1145/2597073.2597091

[54] Rhema Linder and Eunyee Koh. 2015. Quarry: Picking From Examples to Explore Big Data. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems* (CHI EA '15). ACM, New York, NY, USA, 1869-1874. DOI=10.1145/2702613.2732933 http://doi.acm.org/10.1145/2702613.2732933

[55] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze, *Introduction to Information Retrieval*, Cambridge University Press. 2008. http://nlp.stanford.edu/IR-book/html/htmledition/a-first-take-at-building-an-inverted-index-1.html

[56] Microsoft - Partitioning, https://technet.microsoft.com/en-us/library/ms178148(v=sql.105).aspx

[57] Oracle - Partitioning Concepts, http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm

[58] Python, https://www.python.org/downloads/

[59] Reyjrar's ElasticSearch settings, https://gist.github.com/reyjrar/4364063

[60] Seecr: Multi-threaded expert search with Lucene, http://blog.seecr.nl/2014/10/02/multi-threaded-expert-search-with-lucene/

[61] SURFsara, https://www.surf.nl/en/about-surf/subsidiaries/surfsara/

[62] O. Tange (2011): GNU Parallel - The Command-Line Power Tool, The USENIX Magazine, February 2011:42-47, http://www.gnu.org/software/parallel/

[63] Texcavator project page, https://www.esciencecenter.nl/project/texcavator

[64] TU Delft - DAS-4 page, http://www.asci.tudelft.nl/pages/about-asci/das.php

A Queries

Table 18 shows all available user queries. Column **ex_type** shows excluded article types in shorthand (**AR**ticles, **AD**verts, **I**llustrations and **F**amily notices) and column **ex_distr** shows excluded geospatial distributions (**N**ational, **R**egional, Netherlands **A**ntilles, **S**urinam and **I**ndonesia).

date_lower	date_upper	ex_type	ex_distr	query
1960-01-01	1990-12-31			"Amerika" AND "Cola"
1970-01-01	1990-12-31			"Amerika" AND "Cola"
1850-01-01	1990-12-31			"Mainzer beobachter"
1886-01-01	1886-12-31		A, S, I	Kuyper NOT kerk
1850-01-01	1990-12-31		A, S, I	(Richter or Mercalli) AND aardbeving
1850-01-01	1990-12-31		A, S, I	Mercalli AND aardbeving
1850-01-01	1990-12-31		A, S, I	Richter AND aardbeving
1945-01-01	1955-12-31			"Amerika"
1850-01-01	1939-12-31			vetzucht OR zwaarlijvigheid OR vetlijvigheid
1880-01-01	1990-12-31			verslaafd*
1950-01-01	1969-12-31	AD, I, F	A, S, I	firato
1880-01-01	1990-12-31			morphinist OR morfinist
1880-01-01	1990-12-31	I, F	A, S, I	Verslaafd AND Amerika
1850-01-01	1939-12-31	AR, I, F	A, S, I	vetzucht
1950-01-01	1969-12-31	AD, I, F	A, S, I	firato AND duitsland
1880-01-01	1940-12-31	I, F	A, S, I	Verslaafd AND Amerika

date_lower	date_upper	ex_type	ex_distr	query
1950-01-01	1959-12-31	AD, I, F	A, S, I	(draagbare OR transistor OR zakradio OR portable) AND radio
1900-01-01	1910-12-31	I, F	A, S, I	drooglegging NOT Zuiderzee
1890-01-01	1910-12-31	AR, I, F	A, S, I	corpulentie
1920-01-01	1930-12-31			acteur OR tone?lspeler
1850-01-01	1939-12-31	AR, I, F	A, S, I	corpulentie AND NOT mariënbader
1960-01-01	1969-12-31	AD, I, F	A, S, I	(draagbare OR transistor OR zakradio OR portable) AND radio
1890-01-01	1910-12-31			corpulentie
1880-01-01	1990-12-31	I, F	A, S, I	drug
1880-01-01	1900-12-31	AD, I, F	A, S, I	Morphine OR morfine
1850-01-01	1939-12-31	AR, I, F	A, S, I	vet
1960-01-01	1969-12-31	AD, I, F	A, S, I	firato
1920-01-01	1939-12-31	AD, I, F	A, S, I	(draagbare OR portable) AND radio
1920-01-01	1930-12-31			banaan
1950-01-01	1959-12-31	AD, I, F	A, S, I	(transistor OR draagbare OR portable OR transportable) AND radio
1945-01-01	1965-12-31	AD	A, S, I	amerik* OR “verenigde staten”
1945-01-01	1965-12-31	AD	A, S, I	doolaar* AND amerika*
1945-01-01	1965-12-31	AD	A, S, I	eenmanskijk AND amerika
1945-01-01	1965-12-31	AD	A, S, I	paradijs AND cellophaan
1945-01-01	1965-12-31	AD	A, S, I	impressies AND emigrante
1945-01-01	1965-12-31	AD	A, S, I	kris-kras AND amerika
1945-01-01	1965-12-31	AD	A, S, I	geheimen AND amerika
1950-01-01	1959-12-31	AR	A, S, I	(transistor OR draagbare OR portable OR transportable) AND radio
1880-01-01	1939-12-31	AD, I, F	A, S, I	(morphine-inspuitingen OR geneesmiddel OR genotmiddel OR verdoovend OR dope OR drug OR verdoovingsmiddelen OR morphine OR heroine OR narcotica OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR narcoticum OR cocaïne OR vergift OR opium OR coca OR oxycodon OR middelen OR chloral OR prikje OR zenuwen kalmeren OR verdooven OR verdoven OR verdovingsmiddel OR verdovingsmiddelen OR naald OR gif OR bedwelmende) AND (morphinisme OR cocainisme OR heroinisme OR morphinist OR morfinist OR heroinist OR cocainist OR verslaafd OR verslaafden OR neurasthenie OR morfinomanie OR morphinomanie OR cocainomanie OR verslaving OR opiophagen OR misbruiker)

date_lower	date_upper	ex_type	ex_distr	query
1960-01-01	1969-12-31	AR, I, F	A, S, I	(transistor OR draagbare OR portable OR transportable) AND radio
1956-01-01	1960-12-31			"gast van de amerikaanse regering"
1887-01-01	1892-12-31			doleantie
1850-01-01	1990-12-31			chl?ra?l*
1850-01-01	1990-12-31			coca?n*
1850-01-01	1990-12-31			dope NOT doping
1850-01-01	1990-12-31			gene?smiddel*
1850-01-01	1990-12-31			gen?tm* NOT voeding? NOT koffie NOT thee
1850-01-01	1990-12-31			*gif
1850-01-01	1990-12-31			hero?n*
1850-01-01	1990-12-31			morf?n*
1850-01-01	1990-12-31			morp?n*
1850-01-01	1990-12-31			narcot*
1850-01-01	1990-12-31			neura* NOT neurath
1850-01-01	1990-12-31			opi?m*
1850-01-01	1990-12-31			oxyco*
1850-01-01	1990-12-31			code?n*
1850-01-01	1990-12-31			laudanum
1850-01-01	1990-12-31			misbruik AND middel*
1850-01-01	1990-12-31			zenuw* AND kalmer*
1870-01-01	1914-12-31	AD, I, F	R, A, S, I	staat OR maatschappij OR arbeid OR vrijheid OR eeuw OR ware OR zelven OR zooveel OR eigen OR kleine
1950-01-01	1970-12-31	AD, I, F	A, S, I	televisie AND publiek
1970-01-01	1980-12-31	AD, I, F	A, S, I	valium
1950-01-01	1969-12-31	AD, I, F	A, S, I	((draagbare AND radio) OR (transportable AND radio) OR (portable) OR (transistor) AND verbod)
1870-01-01	1914-12-31	AD, I, F	R, A, S, I	wij AND Nederland
1960-01-01	1960-12-31			"huwelijk"
1945-01-01	1959-12-31	AR, I, F	A, S, I	overgewicht OR vetzucht OR zwaarlijvigheid OR corpulentie OR slank
1950-01-01	1950-12-31			applaus or boegeroep or ovaties
1900-01-01	1990-12-31			voetbal
1950-01-01	1960-12-31	AD, I, F	R, A, S, I	betaald voetbal
1952-01-01	1956-12-31	AD, I, F	A, S, I	betaald voetbal
1950-01-01	1960-12-31	AD, I, F	A, S, I	(betaald voetbal) AND invoering
1950-01-01	1960-12-31	AD, I, F	A, S, I	voetbal AND publiek
1950-01-01	1960-12-31	AD, I, F	A, S, I	(betaald voetbal) AND invoering
1850-01-01	1940-12-31			coc??n*
1900-01-01	1940-12-31			chl?r?f?r*
1900-01-01	1990-12-31			laud?n?m

date_lower	date_upper	ex_type	ex_distr	query
1880-01-01	1940-12-31	I, F	A, S, I	opium
1890-01-01	1940-12-31	AD, I, F	A, S, I	verdo*ven* AND middel*
1880-01-01	1990-12-31	I, F	A, S, I	opium
1880-01-01	1990-12-31	I, F	A, S, I	Verslaafd AND Engeland
1920-01-01	1939-12-31	AD, I, F	A, S, I	alcoholverbod OR prohibitie OR drooglegging
1920-01-01	1939-12-31	AD, I, F	A, S, I	Amerikaansch OR Amerika OR Amerikaansche OR New-York OR New york OR Brooklyn OR Chicago OR nieuw-york OR Noord-Amerika OR Yankee OR "Vereenigde Staten" OR "Verenigde Staten" OR Noord-Amerika OR America
1920-01-01	1939-12-31	AD, I, F	A, S, I	(Amerikaansch OR Amerika OR Amerikaansche OR New-York OR New york OR Brooklyn OR Chicago OR nieuw-york OR Noord-Amerika OR Yankee OR "Vereenigde Staten" OR "Verenigde Staten" OR Noord-Amerika OR America) AND (alcoholverbod OR prohibitie OR drooglegging)
1920-01-01	1939-12-31	AD, I, F	A, S, I	drooglegging AND Amerika
1920-01-01	1939-12-31	AD, I, F	A, S, I	(drooglegging OR prohibitie OR alcoholverbod) AND Amerika
1920-01-01	1939-12-31	AD, I, F	A, S, I	(drooglegging OR prohibitie OR alcoholverbod) AND (Amerika OR "Vereenigde Staten" OR "Verenigde Staten")
1910-01-01	1939-12-31	AD, I, F	A, S, I	"verdoovende middelen"
1900-01-01	1939-12-31	AD, I, F	A, S, I	morphinisme OR cocainisme OR heroinisme OR morphinist OR morfinist OR heroinist OR cocainist OR verslaafd OR verslaafden OR neurasthenie OR morfinomanie OR morphinomanie OR cocainomanie OR verslaving OR opiophagen OR misbruiker
1920-01-01	1944-12-31	AD, I, F	A, S, I	alcoholverbod OR prohibitie OR drooglegging
1900-01-01	1939-12-31	AD, I, F	A, S, I	morphinisme OR cocainisme OR heroinisme OR morphinist OR morfinist OR heroinist OR cocainist OR verslaafd
1950-01-01	1960-12-31	AD, I, F	R, A, S, I	Frankrijk
1945-01-01	1955-12-31	AD, I, F	A, S, I	Europese AND integratie
1880-01-01	1940-12-31	AD, I, F	A, S, I	"verdoovende middelen" OR "verdovende middelen" OR "bedwelmende middelen" OR narcotica OR narcoticum OR morphine OR cocaine OR heroine OR opium

date_lower	date_upper	ex_type	ex_distr	query
1880-01-01	1890-12-31	AD, I, F	A, S, I	genotmiddel OR dope OR drug OR verdoovingsmiddelen OR morphine OR heroine OR narcotica OR narcoticum OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR cocaïne OR opium OR coca OR chloral OR "verdoovende middelen" OR "verdovende middelen" OR "bedwelmende middelen"
1880-01-01	1940-12-31	AD, I, F	A, S, I	(genotmiddel OR dope OR drug OR verdoovingsmiddelen OR morphine OR heroine OR narcotica OR narcoticum OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR cocaïne OR opium OR coca OR chloral OR "verdoovende middelen" OR "verdovende middelen" OR "bedwelmende middelen") AND (Amerikaansch OR Amerika OR Amerikaansche OR New-York OR New york OR Brooklyn OR Chicago OR nieuw-york OR Noord-Amerika OR Yankee OR "Vereenigde Staten" OR "Verenigde Staten" OR Noord-Amerika OR America)
1850-01-01	1990-12-31			"Atlanta"
1945-01-01	1990-12-31			"Atlanta"
1945-01-01	1990-12-31			cowboy* AND Cola
1961-01-01	1990-12-31	AD, I, F	A, S, I	(kwis OR quiz OR quiz* OR kwisprogr*) AND (tv OR televisie)
1923-01-01	1926-12-31			"Greta Garbo"
1946-01-01	1950-12-31	AD, I, F	A, S, I	cola
1951-01-01	1955-12-31	AD, I, F	A, S, I	cola
1955-01-01	1960-12-31	AD, I, F	A, S, I	cola
1960-01-01	1965-12-31	AD, I, F	A, S, I	cola
1950-01-01	1959-12-31	AD, I, F	A, S, I	64000 question
1945-01-01	1984-12-31	AR, I, F	A, S, I	cola AND (amerika* OR USA OR "Verenigde Staten")
1945-01-01	1984-12-31	AR, I, F	A, S, I	coca-cola
1945-01-01	1959-12-31	AR, I, F	A, S, I	coca-cola
1945-01-01	1959-12-31	AR, I, F	A, S, I	coca-cola
1945-01-01	1984-12-31	AR, I, F	A, S, I	"coca-cola" or "Coca-Cola" AND (amerika* OR USA OR "Verenigde Staten")
1945-01-01	1989-12-31	AR, I, F	A, S, I	cola AND olympis*
1945-01-01	1989-12-31	AR, I, F	A, S, I	cola AND "long drink"
1850-01-01	1990-12-31			"jumbo" AND ("Amerika" OR "Verenigde Staten")

date_lower	date_upper	ex_type	ex_distr	query
1850-01-01	1939-12-31		A, S, I	“miss blanche virginia”
1880-01-01	1940-12-31	AD, I, F	A, S, I	(“verdoovende middelen” OR “verdovende middelen” OR “bedwelmende middelen” OR narcotica OR narcoticum OR morphine OR cocaine OR heroine OR opium) AND (Amerika* OR “Vereenigde Staten” OR “Verenigde Staten”)
1900-01-01	1940-01-01	F	A, S, I	pict?r?
1900-01-01	1940-01-01	F	A, S, I	f?lm or r?lpr?nt
1900-01-01	1940-01-01	F	A, S, I	b?osc??p or holl?w*d
1900-01-01	1940-01-01	F	A, S, I	“motion picture”
1900-01-01	1940-01-01	F	A, S, I	c?nema
1900-01-01	1940-01-01	F	A, S, I	f?lmst?r
1900-01-01	1940-01-01	F	A, S, I	f?lm*
1900-01-01	1940-01-01	F	A, S, I	cinem*
1900-01-01	1941-01-01	F	A, S, I	*bioscoop*
1945-01-01	1955-12-31	AD, I, F	R, A, S, I	Europa
1950-01-01	1969-12-31			massaconsumptie AND (amerika OR verenigde staten)
1950-01-01	1969-12-31	AD, I, F	A, S, I	massaconsumptie
1918-01-01	1928-12-31	AD, I, F	A, S, I	“verdoovende middelen” OR “verdovende middelen” OR “drugs”
1950-01-01	1969-12-31	I, F	A, S, I	firato
1950-01-01	1969-12-31	AR, I, F	A, S, I	firato
1900-01-01	1914-07-27	AD, I, F	A, S, I	(v?rd??v?nde AND m?dd?l??) OR (v?rd??v?nd AND m?dd?l??)
1918-11-11	1940-05-01	AD, I, F	A, S, I	(v?rd??v?nde AND m?dd?l??) OR (v?rd??v?nd AND m?dd?l??)
1900-01-01	1914-07-27	AD, I, F	A, S, I	(v?rd??v?nd AND m?dd?l) OR (v?rd??v?nd AND m?dd?l)
1918-11-11	1940-05-01	AD, I, F	A, S, I	(v?rd??v?nd AND m?dd?l) OR (v?rd??v?nd AND m?dd?l)
1918-11-11	1940-05-01	AD, I, F	A, S, I	verdovend~1 AND middel~1
1900-01-01	1914-07-27	AD, I, F	A, S, I	verdovend~1 AND middel~1
1900-01-01	1940-12-31	AD, I, F	A, S, I	verdoovingsmiddelen OR narcotica OR narcoticum OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR “verdoovende middelen” OR “verdovende middelen” OR “bedwelmende middelen”
1900-01-01	1940-12-31	AD, F	A, S, I	bioscoop~2
1900-01-01	1920-12-31	AD, F	A, S, I	film~2
1921-01-01	1940-12-31	AD, F	A, S, I	film~2
1900-01-01	1940-12-31	AD, F	A, S, I	cinema~2
1900-01-01	1940-12-31	AD, F	A, S, I	filmster~2

date_lower	date_upper	ex_type	ex_distr	query
1900-01-01	1940-12-31	AD, F	A, S, I	picture~2
1900-01-01	1940-12-31	AD, F	A, S, I	motion~2
1918-01-01	1928-12-31	AD, I, F	R, A, S, I	verdoovingsmiddelen OR narcotica OR narcoticum OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR "verdoovende middelen" OR "verdovende middelen" OR "bedwelmende middelen"
1918-01-01	1928-12-31	AD, I, F	R, A, S, I	verdoovingsmiddelen OR narcotic* OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR "verdoovende middelen" OR "verdovende middelen" OR "bedwelmende middelen"
1918-01-01	1990-12-31	AD, I	A, S, I	Frankrijk
1918-01-01	1990-12-31	AD, I	A, S, I	Franse
1945-01-01	1980-12-31	AD, I	A, S, I	Frankrijk
1945-01-01	1980-12-31	AD, I	A, S, I	Franse
1945-01-01	1980-12-31	AD, I	A, S, I	existentialisme
1945-01-01	1980-12-31	AD, I	A, S, I	Sartre
1890-01-01	1990-12-31		A, S, I	"reagan"
1890-01-01	1990-12-31		A, S, I	"roosevelt"
1890-01-01	1990-12-31		A, S, I	"coolidge"
1890-01-01	1990-12-31		A, S, I	"kennedy"
1890-01-01	1990-12-31		A, S, I	"nixon"
1890-01-01	1990-12-31		A, S, I	"carter"
1890-01-01	1990-12-31		A, S, I	"wilson"
1890-01-01	1990-12-31		A, S, I	"harding"
1890-01-01	1990-12-31		A, S, I	"hoover"
1890-01-01	1990-12-31		A, S, I	"truman"
1890-01-01	1990-12-31		A, S, I	"eisenhower"
1890-01-01	1990-12-31		A, S, I	"bush"
1890-01-01	1924-12-31		A, S, I	"roosevelt"
1924-01-01	1972-12-31		A, S, I	"roosevelt"
1890-01-01	1949-12-31		A, S, I	"hoover"
1924-01-01	1972-12-31		A, S, I	"roosevelt"
1890-01-01	1989-12-31			commercialisme
1900-01-01	1940-05-01	AD	A, S, I	article_dc_title:(kunsten en wetenschappen)
1900-01-01	1940-05-01	AD	A, S, I	article_dc_title:(Uitgaan)
1900-01-01	1940-05-01	AD	A, S, I	article_dc_title:(In Carré)
1900-01-01	1940-05-01	AD	A, S, I	article_dc_title:(Kunst)
1919-01-01	1939-12-31	I, F	A, S, I	"tom mix"
1919-01-01	1939-12-31	AD, I, F	A, S, I	"tom mix"
1945-01-01	1990-12-31	AD, I, F	I, S, A	Banning
1945-01-01	1990-12-31	AD, I, F	A, S, I	Schillebeeckx
1870-01-01	1914-12-31	AD, I, F	R, A, S, I	provincie

date_lower	date_upper	ex_type	ex_distr	query
1870-01-01	1914-12-31	AD, I, F	N, A, S, I	provincie
1945-01-01	1965-12-31	AD, I, F	A, S, I	cola
1945-01-01	1950-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1951-01-01	1955-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1956-01-01	1960-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1961-01-01	1965-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1966-01-01	1970-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1971-01-01	1975-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1976-01-01	1980-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1981-01-01	1985-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1986-01-01	1990-12-31	AD, I, F	A, S, I	Frankrijk AND Franse
1945-01-01	1965-12-31		A, S, I	“cola” AND “modern*”
1900-01-01	1914-07-28		A, S, I	verd*v?nd?
1919-01-01	1939-12-31	I, F	A, S, I	bioscooppaleis OR bioscooppaleizen
1900-01-01	1914-07-28		A, S, I	verd*v?nd? mid?el?n
1966-01-01	1970-12-31	AD, I, F	A, S, I	cola
1970-01-01	1975-12-31	AD, I, F	A, S, I	cola
1976-01-01	1980-12-31	AD, I, F	A, S, I	cola
1981-01-01	1985-12-31	AD, I, F	A, S, I	cola
1986-01-01	1990-12-31	AD, I, F	A, S, I	cola
1919-01-01	1939-12-31	AD, I, F	A, S, I	film AND ster
1947-01-01	1962-12-31	AD, I, F	A, S, I	drugs
1920-01-01	1947-12-31	AD, I, F	A, S, I	drugs
1947-01-01	1962-12-31	AD, I, F	A, S, I	morfine OR morphine
1947-01-01	1962-12-31	AD, I, F	A, S, I	verdoovingsmiddelen OR narcotica OR narcoticum OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR “verdoovende middelen” OR “verdovende middelen” OR “bedwelmende middelen” OR morphine OR morfine
1918-11-11	1939-09-01	AD, I, F	A, S, I	greta garbo
1918-11-11	1939-09-01	AD, I, F	A, S, I	greta garbo
1918-11-11	1940-05-01	F	A, S, I	Greta~AND Garbo~
1918-11-11	1940-05-01	F	A, S, I	Clark~AND Gable~
1920-01-01	1929-12-31	AD, I, F	A, S, I	mary pickford
1920-01-01	1929-12-31	AD, I, F	A, S, I	mary pickford AND groot
1920-01-01	1929-12-31	AD, I, F	A, S, I	mary pickford AND groot AND succes
1920-01-01	1929-12-31	AR, I, F	A, S, I	mary pickford
1930-01-01	1939-12-31	AD, I, F	A, S, I	mary pickford
1920-01-01	1929-12-31	AD, I, F	A, S, I	clara bow
1850-01-01	1940-05-01			klankfilm
1920-01-01	1929-12-31	AD, I, F	A, S, I	clara bow AND popula*
1920-01-01	1939-12-31	AD, I, F	A, S, I	mary pickford AND popula*
1920-01-01	1939-12-31	AD, I, F	A, S, I	clara bow AND popula*

date_lower	date_upper	ex_type	ex_distr	query
1946-01-01	1989-12-31		A, S, I	“cola” AND “amerika”
1946-01-01	1989-12-31		A, S, I	“king size” OR “kingsize” OR “king-size”
1919-01-01	1929-12-31	AD, I, F	A, S, I	pearl white
1919-01-01	1939-12-31	AD, I, F	A, S, I	“pearl white”
1919-01-01	1939-12-31			film AND hollywood
1919-01-01	1939-12-31		A, S, I	(film OR bioscoop) AND hollywood
1945-01-01	1959-12-31	AR, I, F	A, S, I	“coca-cola” OR (“Cola” AND “Cola”)
1919-01-01	1939-12-31	AD, I, F	A, S, I	(re*lame OR propaganda) AND (film OR movie OR rolprent)
1918-01-01	1927-12-31	AD, I, F	A, S, I	(verdoovingsmiddelen OR narcotica OR narcoticum OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR “verdoovende middelen” OR “verdovende middelen” OR “bedwelmende middelen” OR morphine OR morfine OR heroïne OR cocaïne OR opium) AND (binnenland OR nederland)
1918-01-01	1940-12-31	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR “verdo*ven* middel*” OR “bedwelmen* middel*” OR mor*ine OR heroïne OR cocaïne OR opium) AND (binnenland*~OR nederland*~)
1945-01-01	1965-12-31			(“Amerika”) (“Verenigde” AND “Staten”)
1945-01-01	1965-12-31		A, S, I	(“Amerika”) (“Verenigde” AND “Staten”)
1918-01-01	1940-12-31	AD, I, F	A, S, I	verdoovingsmiddelen OR narcotica OR narcoticum OR verslavingsvergift OR opiumderivaten OR bedwelmingsmiddelen OR “verdoovende middelen” OR “verdovende middelen” OR “bedwelmende middelen” OR morphine OR morfine OR heroïne OR cocaïne OR opium
1920-01-01	1929-12-31	AD, I, F	A, S, I	(re*lame OR propaganda OR publicit*) AND (film OR movie OR rolprent) and (amerika OR verenigde staten OR hollywood)
1919-01-01	1939-12-31	AD, I, F	A, S, I	(film OR movie OR rolprent) AND (publiciteit OR publicity OR reclame OR reklame)
1918-11-01	1928-04-21	AD, I, F	A, S, I	verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR “verdo*ven* middel*” OR “bedwelmen* middel*” OR mor*ine OR heroïne OR cocaïne

date_lower	date_upper	ex_type	ex_distr	query
1918-11-01	1931-04-21	AD, I, F	A, S, I	verslaving*
1918-11-01	1931-04-21	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR "verdo*ven* middel*" OR "bedwelmen* middel*" OR mor*ine OR heroïne OR cocaïne) AND (amerik* OR "Ver*nigde staten")
1918-11-01	1931-04-21	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR "verdo*ven* middel*" OR "bedwelmen* middel*" OR mor*ine OR heroïne OR cocaïne) AND (Chin*)
1918-11-01	1931-04-21	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR "verdo*ven* middel*" OR "bedwelmen* middel*" OR mor*ine OR heroïne OR cocaïne) AND (Parijs)
1918-11-01	1928-04-21	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR "verdo*ven* middel*" OR "bedwelmen* middel*" OR mor*ine OR heroïne OR cocaïne) AND (Frans* OR Frankrijk)
1918-11-01	1928-04-21	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR "verdo*ven* middel*" OR "bedwelmen* middel*" OR mor*ine OR heroïne OR cocaïne) AND (Duits*)
1918-11-01	1928-04-21	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR "verdo*ven* middel*" OR "bedwelmen* middel*" OR mor*ine OR heroïne OR cocaïne) AND (Berlijn)
1918-11-01	1928-04-21	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR "verdo*ven* middel*" OR "bedwelmen* middel*" OR mor*ine OR heroïne OR cocaïne) AND (Japan*)
1918-11-01	1940-05-10	AD, I, F	A, S, I	verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR "verdo*ven* middel*" OR "bedwelmen* middel*" OR mor*ine OR heroïne OR cocaïne
1850-01-01	1990-12-31	I, F	A, S, I	"verfrissend*"

date_lower	date_upper	ex_type	ex_distr	query
1919-01-01	1939-12-31	AD, I, F	A, S, I	filmster OR filmsterren OR filmstar OR filmstars
1965-01-01	1973-12-31			“cyclamaten” AND “cyclamaat”
1965-01-01	1973-12-31			“cyclamaten” OR “cyclamaat”
1919-01-01	1939-12-31	AD, I, F	A, S, I	bioscooppaleis OR bioscooppaleizen OR filmpaleis OR filmpaleizen
1945-01-01	1989-12-31	AD, I, F	A, S, I	“cola” OR “frisdrank” OR “hilo” OR “fanta” OR “seven-up” OR “Pepsi”
1945-01-01	1989-12-31			“Amerikanisme” OR “anti-amerikanisme”
1919-01-01	1939-12-31	AD, I, F	A, S, I	bios*ooppaleis OR bios*ooppaleizen OR cinemapaleis OR cinemapaleizen OR filmpaleis OR filmpaleizen OR filmkathedraal
1919-01-01	1939-12-31	AD, I, F	A, S, I	filmpaleis OR filmpaleizen OR bios*ooppaleis OR bios*ooppaleizen OR filmkathedraal OR cinemapaleis OR cinemapaleizen OR kinopalast OR filmkathedraal OR filmkathedralen
1945-01-01	1989-12-31	AD, I, F	A, S, I	“Cola” AND (“Amerika” OR “Verenigde Staten”)
1945-01-01	1989-12-31	AD, I, F	A, S, I	“Cola” AND (“Amerika*” OR “Verenigde Staten” OR “USA” OR “New York”)
1945-01-01	1989-12-31	AR, I, F	A, S, I	“Cola” AND (“Amerika*” OR “Verenigde Staten” OR “USA” OR “New York”)
1945-01-01	1989-12-31	AR, I, F	A, S, I	“Cola”
1945-01-01	1989-12-31	AD, I, F	A, S, I	(“Amerika” OR “Verenigde Staten”)AND “plastic”
1850-01-01	1990-12-31			“mainzer beobachter”
1950-01-01	1970-12-31	AD, I, F	A, S, I	verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR “verdo*ven* middel*” OR “bedwelmen* middel*” OR mor*ine OR heroïne OR cocaïne OR drugs
1966-01-01	1989-12-31		A, S, I	(“Amerika”) (“Verenigde” AND “Staten”)
1918-11-01	1940-04-21	AD, I, F	R, A, S, I	“plaatselijke keuze”
1850-01-01	1990-12-31	AR, I, F	A, S, I	cola AND (Amerika OR “Verenigde Staten” OR USA)
1945-01-01	1990-12-31	AR, I, F	A, S, I	cola OR coca-cola
1850-01-01	1990-12-31	AR, I, F	A, S, I	cola AND (Amerika* OR “Verenigde Staten” OR USA)
1918-11-01	1941-05-10	AD, I, F	A, S, I	(verd*vingsmiddel OR narcotic* OR verslaving* OR bedwelmingsmiddel* OR “verdo*ven* middel*” OR “bedwelmen* middel*” OR mor*ine OR heroïne OR cocaïne) AND alcohol*

date_lower	date_upper	ex_type	ex_distr	query
1945-01-01	1989-12-31	AR, I, F	A, S, I	cola AND (modern* OR man* OR vrouw* OR sport OR fris* OR verfris* OR gast*)
1945-01-01	1989-12-31			verfriss*
1880-01-01	1914-07-28	AD, I, F	A, S, I	morphine OR morfine
1919-01-01	1939-12-31	AD, I, F	A, S, I	(film OR film*) AND (afgod OR afgod*)
1850-01-01	1990-12-31			“Atlanta” AND “cola”
1850-01-01	1990-12-31		A, S, I	“Atlanta” AND “cola”
1850-01-01	1990-12-31		A, S, I	“Atlanta”
1945-12-31	1990-10-01		A, S, I	“Atlanta”
1850-01-01	1990-12-31			“dollar-imperialisme” OR “dollar-imperialisten” OR “dollar imperialisme” OR “amerikaanse imperialisten” OR “amerikaans imperialisme” OR “amerikaanse imperialisten”
1945-01-01	1989-12-31		A, S, I	“dollar-imperialisme” OR “dollar-imperialisten” OR “dollar imperialisme” OR “amerikaanse imperialisten” OR “amerikaans imperialisme” OR “amerikaanse imperialisten”
1880-01-01	1914-07-28	AD, I, F	A, S, I	verd*vingsmiddel OR narcotic* OR bedwelmingsmiddel* OR “verdo*ven* middel*” OR “bedwelmen* middel*” OR mor*ine OR heroïne OR cocaïne
1945-01-01	1990-12-31	AD, I, F	A, S, I	“relax*” OR “Ontspannen” AND (“Amerika8” OR “Verenigde Staten” OR “USA”)
1945-01-01	1960-12-31	AR, I, F	A, S, I	cola AND (modern* OR man* OR vrouw* OR sport OR fris* OR verfris* OR gast*)
1945-01-01	1960-12-31	AR, I, F	A, S, I	cola
1945-01-01	1956-12-31	AR, I, F	A, S, I	cola
1945-01-01	1989-12-31	AR, I, F	A, S, I	cola AND (modern* OR man* OR vrouw* OR sport OR gast*)

Table 18: All available user queries and their properties. Date range borders are inclusive. Excluded article types (**ex_types**) are **AR**ticle, **AD**vert, **I**llustration and **F**amily notice. Excluded geospatial distributions (**ex_distr**) are **N**ational, **R**egional, Netherlands **A**ntilles, **S**urinam and **I**ndonesia

B Data fields

field	description
identifier	A unique identifier for the document
text_content	The body of the document
article_dc_subject	The document type
article_dc_title	The title of the document
article_dc_identifier_resolver	URI reference to the document online
paper_dc_date	The publish date of the entire newspaper
paper_dc_identifier	An identifier defining the entire newspaper
paper_dc_identifier_resolver	URI reference to the entire newspaper online
paper_dc_language	The language the newspaper is written in
paper_dc_publisher	The publisher for the newspaper
paper_dc_source	Identifier denoting the KB
paper_dc_title	The title of the newspaper
article_dcterms_accessRights	Denotes whether the article is available online
paper_dcterms_alternative	Alternative name(s) for the newspaper
paper_dcterms_isPartOf	Denotes the project (Databank of Digital Daily newspapers)
paper_dcterms_temporal	Whether the newspaper was a daily, weekly, .. edition
paper_dcterms_isVersionOf	A more specific newspaper categorization type
paper_dcterms_issued	When these newspapers got distributed
paper_dcterms_spatial	Where these newspapers were distributed
paper_dcterms_spatial_creation	Where the newspapers were distributed from
article_dcx_recordIdentifier	A unique identifier for the article
paper_dcx_issuenumber	The issuenumber of the newspaper
paper_dcx_recordIdentifier	A unique identifier defining the newspaper
paper_dcx_recordRights	Denotes who owns the rights on the document
paper_dcx_volume	The volume number of the newspaper
paper_ddd_yearsDigitized	When the newspaper was digitized
zipfilename	The zipfile which contains this document

Table 19: Document fields and their description.

C Mutated queries

Original query:

```
"Cola" AND ("Amerika*" OR "Verenigde Staten" OR "USA" OR "New York")
```

Possible mutated queries:

```
"Colh" AND ("Amjrika*" OR "Verenigde Staten" OR "USA" OR "New Yark")
"Colw" AND ("Amerika*" OR "Verenighe Stateh" OR "USA" OR "New York")
"Cfla" AND ("Amerika*" OR "Verenibde Stawen" OR "USA" OR "New York")
"Corm" AND ("Amerika*" OR "Verenigde Staten" OR "USA" OR "New York")
```

```
"Cola" AND ("Amerika*" OR "Vernnigde Stateh" OR "USA" OR "New Yyrk")
"Cola" AND ("Amerika*" OR "Verynigde Sxaten" OR "USA" OR "Nbw York")
"Coca" AND ("Amarika*" OR "Verenigde Staten" OR "USA" OR "Nxw York")
"Cola" AND ("Amerika*" OR "Vegoaigde Staten" OR "USA" OR "New York")
"Coma" AND ("Acerika*" OR "Verenigde Staten" OR "USA" OR "New Yorj")
"Cola" AND ("Amecika*" OR "Verenigde Stattn" OR "USA" OR "Nfw York")
```

D Encountered issues

The most important lesson about datasets is that they should not be assumed to contain only certain information. The Texcavator queries only cover a data range between 1850-1990, as defined by the Texcavator source, but documents range from 1618-2011. As shown in Section 5.1, many documents could be ignored during import if only the data from 1850-1990 has to be imported. Furthermore, the dataset consisted of millions of duplicate documents. The 2012 subset, for example, contained about 18 million duplicate entries. Ignoring these during import, instead of letting ElasticSearch analyze the same data *again*, would improve the overall effective throughput.

The mapping, as discussed in Section 5.2, misses out on 23 fields within each document. These are parsed by ElasticSearch to require full text search. However, nobody ever searches on these fields. Therefore it is a waste of CPU power and disk space to parse these fields as such. Since each field is also automatically included in an `_all` [23] field. This field is created to be used when no fields are specified during search. However, due to the way Texcavator uses ElasticSearch, there are always fields defined. Therefore, disabling the `_all` field or removing unnecessary fields from the `_all` field should reduce the amount of wasted resources.

Several problems were encountered with ElasticSearch. First of all, ElasticSearch' documentation seems to be present. However, when the documentation is actually needed, it does not suffice in the explanations it provides, if any. For example, most settings are explained by repeating the words used in the setting name. Next, ElasticSearch had some technical issues. During shard relocation IO errors can occur (at the sending or receiving node), resulting in the shard not being transferred. However, ElasticSearch will retry to reallocate the shard indefinitely since the shard distribution amongst the

nodes is imbalanced. The cluster state will stay on 'yellow', meaning that the cluster is fine, but not done reallocating. This meant automating benchmarks was not always possible.

Finally, the ElasticSearch-Hadoop library only allows to start Hadoop on a query, but there are multiple API endpoints available resulting in lists of data. For example, the multi termvectors API. Incorporating these API endpoints into the library would greatly extend the power of this library. Especially once the multi termvectors API is fixed, since that could speed up word cloud generation even further.