# Multi-Source Search Engine based on Elasticsearch with focus on relational Databases

Robin Buchta
Marius Luding

Hochschule Hannover
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science
Hannover, Germany
Email: buchta.robin@outlook.de
Email: mluding@gmail.com

Philip Ohm
Maximilian Senge

Hochschule Hannover
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science
Hannover, Germany
Email: philip.ohm@live.de
Email: maximilian.se@web.de

***Abstract*—This work discusses the development of a search engine for multiple sources. A subset of the Elastic stack is used for the searching and analyzing, while the data sources are in multiple different relational databases. By using ETL techniques, the data is being processed in Elasticsearch. The goal is that the results contain the path to the target application or file. Multiple options for indexing, visualization and loading data are discussed with their respective advantage and disadvantage, as well as a prototype which connects the Elastic stack multible sources like an MySQL and PostgreSQL database.**

***Index Terms*—Big Data, Elasticsearch, Kibana, Logstash, relational databases**

## I. Introduction

Today, more and more data is being stored in different forms and in target systems. The search for specific information often requires complex queries. These queries cannot be processed very efficiently by traditional relational databases. In addition, users need a lot of knowledge about the data schema and the query language. These databases were designed to preserve the ACID (atomicity, consistency, isolation, durability) properties. NoSQL databases, such as Elasticsearch, on the other hand, follow a BASE (basically available, soft state, eventual consistency) property. This property has many disadvantages, especially in terms of consistency properties, which are preserved by ACID, but the use cases where the disadvantages become a problem are no longer present, if we put them on top as a search engine. The NoSQL databases offer the following advantages: A simple horizontal scalability and schema freedom [1]. This makes them well suited to coexist with traditional databases, combining the advantages of both worlds. Elasticsearch, a NoSQL database, secializes in search and uses Lucene as a back end for indexing data to efficiently find data [2]. With this database it is possible to combine the advantages of both worlds (NoSQL and SQL Databases) and provide both consistent data on the operational side and a central efficient and comfortable search on the analytical side.

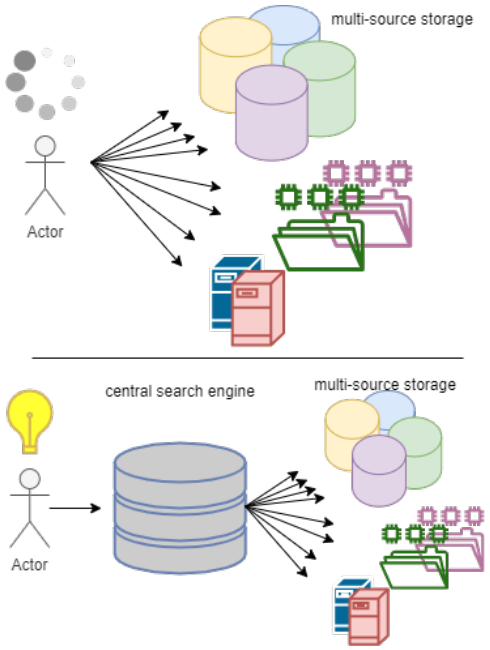


Fig. 1. The existing problem and the basic idea of implementation.

Figure 1 shows the existing problem with our proposed idea of implementation The figure shows that if a multi-source storage solution is given, the user needs to know all the sources for his information needs. In addition, he must know the query language and the data model. On the opposite is our solution, where the user only needs to know one source and also only one query language. In that way, the user will be able to do a full text search and get a better performance. The central search engine abstracts the different source systems and takes care of data acquisition, updating and processing by means of

suitable strategies.

The paper is structured as follows:

In the next chapter, *Related Work*, previous works are presented, which have dealt with similar issues and technologies. In the section, we will present paper that strives for coexistence, as we do, as well as paper that replaces legacy systems.

Chapter *Fundamentals* covers the basics, which are necessary to create a search engine for relational databases. It describes how data preparation must look like in order to be able to process the data in Elasticsearch. Different approaches are considered and discussed. In addition, the aspect of how data can be inserted is treated. Here, different methods from tiggers in the databases to continuous adding are considered. Further it goes with the indexing of the data, which possibilities has the user here and where are the pro and cons thereby. Then follows the topic of how the data can be searched and what optimizations can be made, and how Elasticsearch generally performs the search. The visualization of the data is also discussed. And an outlook is given on how the data can subsequently be retrieved in the source system.

The chapter, *Prototype*, covers our proof of concept. The implementation of a prototypical search engine. Here we used sample data from different source systems, which are written into an index. For this we used a MySql and a Postgresql database. The data was imported into Elasticsearch with Logstash and visualized using Kibana.

In chapter *Results and Analysis* we will discuss the problems encountered and then make a recommendation as to which methods we think are best suited for creating a search engine for relational databases.

In the last chapter *Conclusion*, we summarize the core aspects of the thesis and give a short outlook on possible further work.

## II. Related Work

In research, the topic of search engines for relational databases is not frequently considered. There is a large amount of research that deals with the replacement of relational databases, whereas coexistence is less frequently addressed topic. Nevertheless, there are some interesting papers in the area. One of the first that was analysed was "Heterogeneous Database System for Faster Data Querying using Elasticsearch" by Umesh Taware and Nuzhat Shaikh [2]. Here, the authors describe the problem that one encounters when the amount of data exceeds the analysis capacity of relational databases. As a solution they propose heterogeneous database systems. SQL databases served as the primary data store and NoSQL databases, such as Elasticsearch, as the secondary data store. Thus, the advantages of both worlds are used. For example, the SQL database is utilised for the create, update and delete operations while the Elasticsearch database, is used for retrieving data. Furthermore, Elasticsearch optimizes redundancy by checking whether the data is already available in real time before it is created. Dequan Chen et. al. presents a real time persistence solution of health data in their paper [3]. For this, they created a Big Data system built on Hadoop and Elasticsearch. This allows both current and historical patient data to be analyzed in a timely manner. In addition, Elasticsearch makes it possible to use a full text search. The authors contrast the new solution against the old RDBMS solution and have been able to meet all their objectives. It has become a high-performance, scalable solution that enables full text search. Other researchers, Oleksii Kononenko et. al, have analyzed Elasticsarch as a solution to Big Data problems [4]. They describe the technology in detail and have identified its strengths and weaknesses. The strengths are scalability, agility and performance. The disadvantages are the security and the steep learning curve for the queries. It should be noted that the paper is from 2014 and the mentioned security disadvantages are no longer present today [5]. Another interesting contribution comes from He Yin and Deng Fengdong. Their goal is to create a new system for meteorological data to take advantage of Big Data. The previous solution was based on RDBMS systems and was not performant and scalable. They have two input sources. One is a data stream and the other being data that is regularly brought in from other systems. The new system uses Elasticsearch to search and Hadoop to store the data. A common API is provided, that searches for the data using Elasticsearch and returns accordingly from the HDFS [6]. Next, we would like to present a comparative study. Sheffi Gupta and Rikle Rani compared Elasticsearch with CouchDB [1]. The results show that Elasticsearch is slower in the operations: create, update, and delete. But much more performant in retrieving the data. What is also worth mentioning in this paper is the comparison of NoSQL databases and relational databases. Here the advantages of both worlds are considered. The NoSQL databases have a BASE property, whereas the relational databases track ACID. Furthermore, a comprehensive overview of Elasticsearch and CouchDB is given. Last, we would like to present a paper that deals specifically with the search topic. This paper is by Souabh S Badhya et. al. The authors create a system that uses natural language queries to request the SQL source systems to return the desired results. Here is also addressed the problem that to get knowledge from relational databases you need to know the syntax of the source system and data structure. The solution system uses a CSV import of the source database. Analyzers are used for full text search. The search is performed in several descriptive fields. The result returns the source and a score. A SQL query is created from the result, which is then used to query the source database. In this way, full text queries can be made to an SQL database by interposing Elasticsearch. The solution system uses a CSV import of the source database. Elasticsearch Analyzer ist

utilized for full text search [7].

## III. Fundamentals

### A. ETL - Extract, Transform, Load

The ETL process (Extract, Transform, Load) is about collecting data from different sources, transforming it and finally loading it into a target system [8].

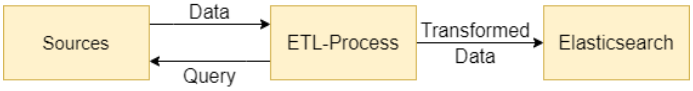


Fig. 2. Exemplary flow of an ETL process.

Since this work is dedicated to Elasticstack, Figure 2 shows a rough integration of the ETL process into the Elastic stack workflow. How the ETL process queries the data and transforms it into the appropriate format is explained below.

The SQL databases and events are considered as sources in this work.

*1) Relational Databases:* According to Objectrocket's whitepaper [9], there are many ways to process the data in the ETL process. The load on the database, the storage requirements of Elasticsearch, and the implementation/-configuration effort must be considered. Listed below are some examples of how the data model can be transformed into different Elasticsearch indexes using the ETL process. This process is called mapping [4].

The mapping is divided into two parts. One is the implicit mapping, which is automatically offered by Elasticsearch and is suitable for simple data structures. However, if the storage space is limited and attributes are to be omitted during indexing, this is the wrong approach, since Elasticsearch uses all attributes for the index. On the other hand, there is the explicit mapping. Here, the programmers determine how the mapping has to look, which attributes flow into the index and how the index looks in the end. There are different approaches. In this work three different approaches are discussed and the pro and cons examined [4]. All approaches are derived from [9]:

1) Denormalization of the data tables and replication: In this approach, the normalized database schema is denormalized and the information to be stored in Elasticsearch is extracted (see listing 1). For this purpose, a join is usually executed over all tables and the attributes of the interest are filtered using a selection. The records are then passed to Elasticsearch. If the source data have timestamps and headers as attributes, it is possible to create historical truth (the consideration of the records lying in the past for an entity) and thus also access the past. Each record is stored as a separate document. If an entity now has several records (employee working at two locations), two documents are created within Elasticsearch.

Pros:

- Simplicity,
- good for simple queries,
- good Kibana support (Elastic stack).

Cons:

- SQL is very expensive. Load on the database,
- high memory requirements, as each combination is stored redundantly.

```
{
    "_index": "employnorm",
    "_type": "doc",
    "_id": "someID",
    "_source": {
        "name": "Christian Koblink",
        "birthday": "12.12.1966",
        "department": "d004",
        "role": "Senior Engineer",
        "hired": "06.06.2012",
        "department.name": "Production"
    }
}
```

Listing 1. Query result of denormalization aproach [9].

2) Aggregation and filtering: Here, a different approach is taken to build the SQL statement. Each document belongs to an entity (for example employee). In addition, each entity contains the attributes of interest, which is then filled with the respective source data (see listing 2). If the example of the employee is considered further, the object would have the following attributes:

- Attributes of the employee (name, id, ...),
- roles (array, attribute of interest).

Pros:

- One document per entity,
- less overload when saving data.

Cons:

- Query expensive,
- no out-of-the-box Kibana-Solution for nested attributes.

```
{
    "_index": "empnested",
    "_type": "doc",
    "_id": "someID",
    "_source": {
        "name": "Christian Koblink",
        "birthday": "12.12.1966",
        "hired": "06.06.2012",
        "roles": [
            {
                "department.name": "Production",
                "role": "Senior Engineer",
                "role.from": "06.06.2012",
                "role.to": "null",
                "department.number": "d004"
            },
            {
                "department.name": "Production",
                "role": "Junior Engineer",
                "role.from": "01.01.2008",
                "role.to": "06.06.2012",
                "department.number": "d004"
            }
        ]
    }
}
```

Listing 2. Query result of the aggregation and filtering aproach [9].

3) Parent and Child: This approach aims at a kind of normalization of the data. As in a normalized

database, documents refer to others in order to avoid redundancies (see listing 3). Therefore, this approach has the greatest complexity, since on the one hand two query types (parent and child) and also the Elasticsearch query (request query for Elasticsearch) must be defined.
Pros:

- Less redundancy,
- one document per entity,
- consistency is given, if of one documents is changed it effects the other documents.

Cons:

- No out-of-the-box solution available,
- ETL-tool needs to be well understood,
- Kibana support not available, much rework needed.

```
{
    "_index": "employnorm",
    "_type": "doc",
    "_id": "someID",
    "_source": {
    "name": "Christian Koblink",
    "birthday": "12.12.1966",
        "employeenumber": "1337",
        "hired": "06.06.2012",
        "department.name": "Production"
    }
},
{
    "role": {
        "hits": {
            "total": 1,
            "hits": [{
                "_type": "doc",
                "_id": "dsome id",
                "routing": "1337",
                "source": {
                    "role.from": "06.06.2012",
                    "role.to": "null",
                    "rname": "Senior Engineer",
                    "enumber": "1337",
                    "dnumber": "d004",
                    "dname": "Production"
                }
            }]
        }
    }
}
```

Listing 3. Query result of the parent and child aproach [9].

So far, approaches which only query the database at regular intervals have been considered. However, in order to enable that the data can be queried in real time, there are several other approaches that can be used. In the following, two approaches are described and their advantages and disadvantages are discussed:

- Polling at regular intervals [10]:
  The source data is stored with a timestamp. With each polling the data is then loaded from the delta (The time difference between the last load and this run). The use of a timestamp ensures that the entire database is not transferred/queried each time and thus a performance gain exists.
  Pros:
  - Only the delta is loaded,
  - easy to implement,
  - no logic within the database.

  Cons:
  - Source data needs timestamp as an attribute,
  - if polling interval is to big, its not real time anymore.
- Use of triggers [11]:
  When using triggers, logic must be moved to the database. The database takes over the ETL-process and transfers the updated data to Elasticsearch via HTTP call. As long as the database supports triggers and HTTP calls (supported by PL/SQL), it is possible to communicate directly with Elasticsearch's REST interface.
  Pros:
  - Real time,
  - no configuration of an ETL-tool needed.

  Cons:
  - Logic within the database,
  - debugging of triggers is difficult,
  - database needs support for HTTP-Calls.

It has already been suggested several times that the periodic loading of new data can significantly increase the load on the database. This is a classic data warehouse problem and can be minimized with the introduction of a staging area.

A staging area is used to create an image of the source data and thus decouples the ETL process from the sources. The ETL process then reads the data from the staging area, which is ideally located on another node. The staging area can be filled by the source database using triggers so that the load is minimized and the staging area is filled dynamically. The 3nf is preserved so that there are no large loads on the source database, when loading the data [12].

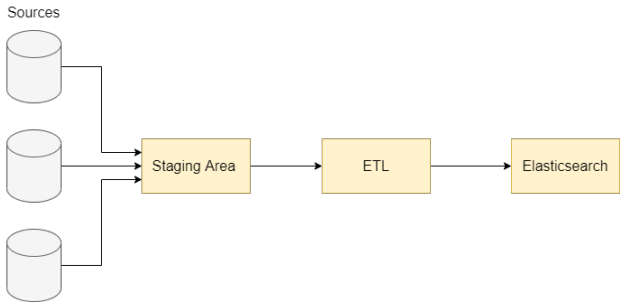


Fig. 3. Architecture of a staging area.

Figure 3 shows a classic solution with a staging area. These contain a 1 to 1 copy of the respective source tables and are updated dynamically. The ETL process (in the figure the core area) then accesses the staging area and not the original sources as before [12].

After explaining possible approaches on how to build an ETL process within Elasticsearch, we will now present tools that can be used to implement the ETL process:

- Logstash [13]:
  Logstash is an ETL-tool that comes from Elasticsearch and belongs to the Elastic stack. Therefore,

the integration of Logstash into the Elasticsearch workflow is easy and can be done without much effort. All basics regarding ETL, which were covered before, can be implemented using Logstash. Due to the possibility to connect plugins in Logstash, it supports a wide range of data sources. Each ETL process is represented by a pipe. The pipes can be scaled horizontal as well as vertical.
Pros:
  - Simplicity,
  - good documentation,
  - large community,
  - Open-Source.

- Apache NiFi: [14]
  Apache NiFi is an open-source tool suitable for developers who have a high-level knowledge of ETL-processes. NiFi supports SQL queries. Furthermore the ETL-process can be configured via a web-UI.
  Pros:
  - Web-UI for designing ETL,
  - easy to use,
  - able to use security policies,
  - scalability.

  Cons:
  - Consitencyproblem, when changing the main node.

- Apache Spark: [15]
  Spark is used to process streams and batches and to add them to Elasticsearch. Spark is particularly suitable for large data volumes, as the system is highly scalable. In addition, Apache Spark is open source. Since the focus is on streams and batches, Logstash or Apache NiFi should be used, since SQL support is available here.
  Pros:
  - Performance,
  - Analytics tools,
  - Good for big data.

  Cons:
  - Fewer algorithms,
  - Not good for small files.

*2) Events:* Another way to get data into Elasticsearch is to communicate directly with the REST API of Elasticsearch, as it was briefly mentioned earlier. All functions offered by Elasticsearch are accessible through the REST API [16]. So the data, which is inserted into the database, will also be inserted in Elasticsearch at the same time, so there is less load on the relational database. This reuqires calling the aforementioned REST API. In the process, certain optimizations are accessible. It strongly depends on the data load and the application environment. The advantage of this variant is that a push mechanism is used and thus no network overhead is generated. The pull mechanisms, as they are used by Logstash, also have requests, which do not lead to new entries, or only to a few and therefore the indexing is less performant. For example, the authors of [7] use Elasticsearch's Bulk API so they don't have to index each record individually. Thus, significant performance improvements can be made. This also only succeeds if the data holds all the information. The authors of [6], are in the situation that the data are not yet complete. They use a Kafka [17] queue, so that they can transport the data in real time with data can reliably transport the data in real time. At the other end of the queue is Storm. Storm is a distributed, reliable and fault-tolerant data real-time streaming processing system [18]. And then they integrate their data also with Elasticsearch Bolt API, Apache Storm offers support of this feature [19].

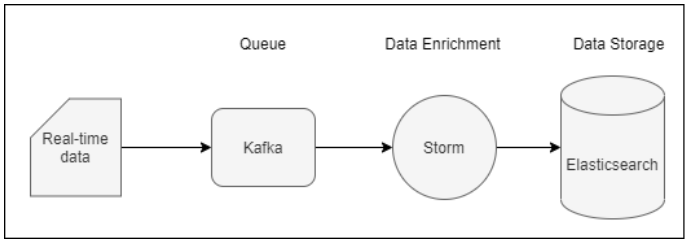


Fig. 4. Example structure for event data.

Figure 4 shows an examplary architecture for real-time event data. Here a Kafka queue was used for Data reliability and Storm for Data enrichment. Of course, it depends on the use case whether a queue and/or the use of a stream processing technology is required.

### *B. The Text Analyzer*

When searching, and by extension analysing, all data types need to be different, depending on multiple factors. Contrary to numbers or other datatypes that can be ordered and then searched, this does not apply to text, where a high relevance is not just defined as a clear match for one of the input search queries, but might also depend on the semantics of it. When ingesting text with Elasticsearch, the input is being broken up into terms by the analyzer set for that specific field. In figure 5 the three main stages are displayed.

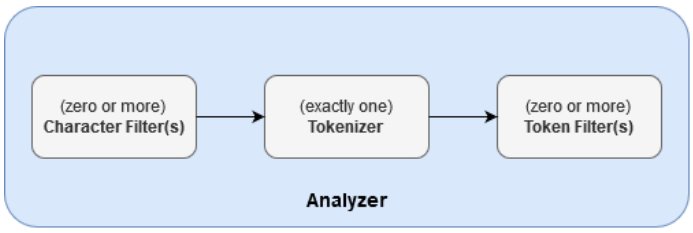


Fig. 5. Components of an analyzer [20].

In the character filter(s) the input can be cleaned of any symbols that may be contained in the original dataset, but are not needed or might even interfere with further analysis. An example would be formatting data such as

markup or html-code or converting abbreviations to their full-text variants. The next step is the tokenizer, which defines how the given string is broken up into terms and how those terms are mapped. Tokenizers fit into three categories:

- Word-oriented-tokenizers: These divide a text-stream into words upon finding whitespaces or other delimiters and rules. Additional functionality such as removal of punctuation symbols or lowercasing may also be included.
- Partial-word-tokenizers: When encountering a specified character, these break up the words into small fragments, e.g. for usage in partial word matching.
- Stuctured-text-tokenizers: For text containing structured data, these can be used to map individual parts of the input e.g. splitting an address into a numerical type for the zip or text-type for the city-name by using regular expressions.

The output of the tokenizers can then further be filtered. Examples for commonly used token-filters are:

- Stop-word-filter: Words such as "the", "it" and "and" do not always add to the information content of a text and can be removed to save storage capacity.
- Snowball-token-filter: Breaks down words into their word-stem. This is useful in combination with a fuzzy query.
- Keep-types-token-filter: This filter reads the type set by the tokenizer and can be configured to discard all numeric-type tokens.

If no specific analyzer is defined, the standard analyzer will be used. It consists of only a tokenizer that splits the input on whitespace characters and passes it to a lowercase-token-filter which simply converts all input tokens to lowercase[20].

### C. Indexing and Storage

In the Elasticsearch world an index can be one of two closely related concepts. The Elasticsearch indices are the largest unit of data inside of Elasticsearch (we will refer to this as 'index' from now on). The other, Lucene indices, are also known as shards inside the Elasticsearch ecosystem. By default an index is split into five shards which may be distributed over multiple nodes. A node is an instance of Elasticsearch, one or more connected nodes form a cluster. Each shard can be replicated on a different node, the default set is one replica for each primary shard. Replicas not only serve as a copy in case of failure of a node, but also help to further distribute load, as each replica shard can be queried simultaneously to the primary shard. In Fig. 6 the arrangement of a whole cluster that serves a single index can be seen. [21]

On the other end of the scale, an index is made up of documents of a single type. Inside of the documents are fields which contain the information that needs to be searched for. The fields are stored as Elasticsearch datatypes in for those dedicated, optimized data structures,

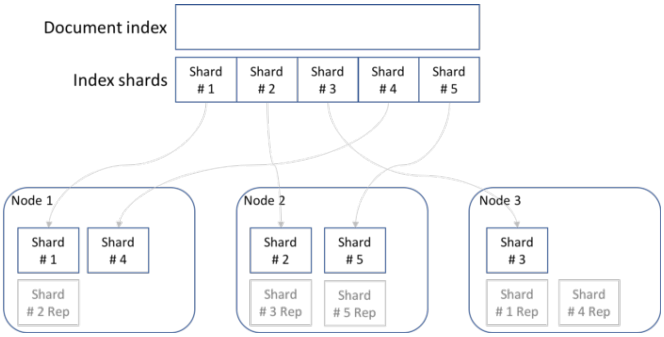


Fig. 6. Components of an Elasticsearch cluster [21].

such as an inverted index for text, or BKD-trees for storing numerical data. An inverted index is similar to the indices found at the end of books. In Elasticsearch the number of occurences in the document is also tracked. As the index points back to the document including the search term, the rest of the data can easily be retrieved, once its relevance has been confirmed (see section III-D1 on how the relevance scoring works). In figure 7 on the bottom, an example for an inverted index, built from the input 'gods & heroes: rome rising' using the standard analyzer can be seen. When ingesting data, the fields can be mapped to datatypes either implicit (i.e. when Elasticsearch automatically maps incoming fields to datatypes) or explicit mapping, where precise rules can be defined about which fields should be treated as text fields or the format for date values, or if the field should be discarded. [22] During this, it can also be decided to keep (the default setting) or discard the inital non-analyzed input, which is stored in the '_source' field of a document.

### D. Search

A search request contains the information which index is to be searched, and a json string with the search terms and search options, such as which fields are to be searched and which analyzer to use. Depending on the chosen analyser for the query, the search terms are tokenized, mapped and then searched for in the specified document fields.

*1) Queries:* One of the most basic queries is to find if a given document should be included in the response, e.g. in our example from figure 7 only when the document has a title field ('exists query'). This however would simply return a (presumably long) list of all documents with the title, but not give any indication of how important each of the results would be. For numerical values and other structured data there are the following query types:

- Range query: Return all documents whose values are inside the defined search-range. This may also be used for date-ranges, and also includes the lower and upper boundary.
- Exists query: Return all documents from the index, where the defined field is present. This could be used to prevent a document from being displayed, that

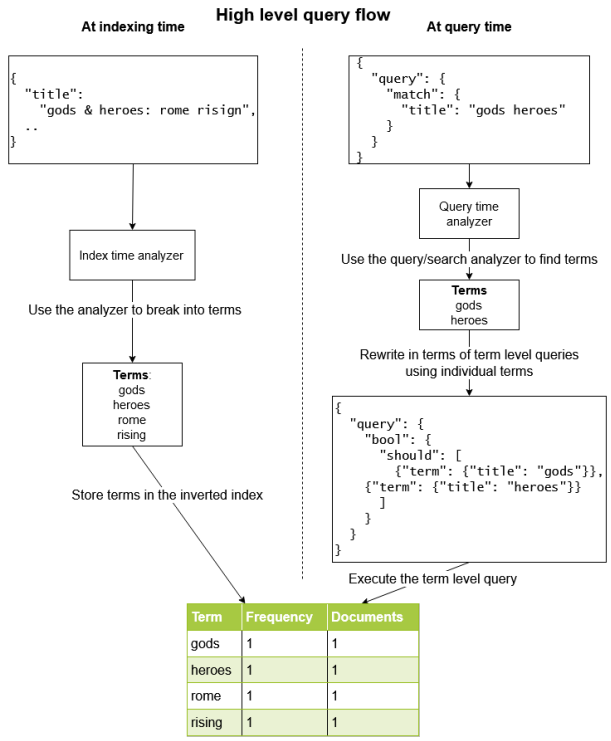


Fig. 7. Example of a high level query flow [20].

has no information in the field the user is actually interested in. (e.g. when searching for all documents with an optional 'comments' field.)

- Term query: Only returns results with an exact match, e.g. where a boolean for a field is set to 'true'.

A more advanced text search for the example input in figure 7, 'gods heroes' would be tokenised by the standard analyzer to 'gods' and 'heroes' and searching for those tokens in the field of 'title' would return the following entry:

```
{
 ...,
 "hits": {
 "total": 1,
 "max_score": 2,
    "hits": [
         {
         "_index": "library_content",
         "_type": "books",
         "_id": "AV5rBfasNI_2eZGciIbg",
         "_score": 2,
         "_source": {
          "title": "gods & heroes: rome rising",
          ...
         }
         }
    ]
 }
}
```

Listing 4. Text query using the standard analyzer.

Here the term 'gods' is only found with a frequency of one in the document 1, with the same applying to 'heroes'. This overall results in a score of two. If a request were to be 'hero and god' and the expected result was to also display the above results, a query with a higher 'fuzziness' might be used. In this case, the word 'god' is one edit away from 'gods' but 'hero' would be 2 edits from 'heroes' thus a fuzziness of 2 would be needed. A higher fuzziness however also generates more false-positives (e.g. 'hell' is also 2 edits away) and it should be noted, that every additional 'fuzzy' change also results in another search being run behind the scenes. 'Heroine' would not be found, as the maximum supported amount of changes to match a term is 2. The opposite of 'fuzzy' queries, are 'match phrase queries' where an exact sequence of words is being searched for. This can be useful, if a phrase is from the searched document is already known, with an additional parameter being the slop, which controls the amount of words in the phrase that might be skipped and still match the request.

*2) Relevance Scoring:* An important criterion for sorting the results gathered by a search is the relevance returned inside the result set. As an example we will take a range query, where if the condition matches, a relevance value of 1 is always returned. This may be 'boosted' by a factor, to affect the overall resulting score. This could be used to search for a name primarily inside a date-range, but to also include possible name-matches that are outside the range with a lower score. The scoring for text however is more complex and can be described by the function:

$score(q,d) = coord(q,d) * queryNorm(q) * \sum(tf(t\ in\ d) * idf(t)^2 * (t.getBoost()) * norm(t,d))$

Where we want to find the relevance score of the document d for a given query q.

- A higher *coord(q,d)* means that more of the search terms/tokens have been found in the document.
- The *queryNorm(q)* ensures that a comparision between queries can be made.
- *tf(t in d)* is the frequency, in which the term appears in the document.
- *idf(t)* correlates to the inverse of the occurrences of the term t in all documents in the index.
- *t.getBoost()* can be considered the other side of relevance-boosting of an individual field, because it can be set at query time.
- *norm(t,d)* encapsulates the field-boost and lengthNorm, which are both added to the document at indexing. LenghtNorm ensures that even short fields which have exhausted their terms matching a query, will have a similar score to long fields doing the same.

[7][23]

### *E. Visualization*

For visualizing the result of the search query, Elasticsearch outputs the data as a json-object through a REST API. Due to the wide support of the json file format, implementing a front-end is possible in many different ways. To make the results „clickable“, the path to the application/website has to be stored in the data and be specificially visualized by the frontend. Calling the URL Elasticsearch provides from the frontend itself poses security issues in a non controlled environment, since

being able to access the URL means, having the ability to perform CRUD operations if no security measures were taken. In many sources it is adviced to put a proxy between Elasticsearch and the frontend [4]. Since the newer versions cover security features, like authorization and authentification for free, this is not necessary anymore and might be more error prone [24]. Having autocompletion in a search engine is common today. This can be circumvented by using the X-Pack extension and it's security features, which is explained later. For that, Elasticsearch provides the completion suggester, which simplifies the implementation of auto-completion. To take advantage of that feature, the mapping has to be specified first as well as indexing additional data to determine the suggestions [25].

Building a custom front-end can take a lot of time and resources. For analyzing and visualizing data, Kibana was devloped. Kibana is a part of the Elastic stack and provides a web-based interface for searching, analyzing, managing and viewing data by using the RESTful Elasticsearch API. The main target of it are advanced users such as analyists, administrators or business users [26]. Kibana provides multiple tools for different needs. The following are the ones not limited to the X-Pack extension [27]:

- Discover: Enables the user to search through an index of data with different filters. Additionally to that, it also supports the Kibana Query Language (KQL), which offers a simplified syntax for scripted fields inside the documents.
- Visualize: Provides multiple ways to visualize data, such as displaying different kind of charts, tables of data, etc.
- Dashboard: A composition of multiple visualizations as well as results of Discover in one view. The dashboard in configurable in size and form, so the user has the ability to customize it for their individual needs.

The X-Pack extension provides additional features, such as machine learning, canvas, maps and security features. To access some of these, a paid subscription is necessary. Especially for security, Kibana doesn't provide a user-management feature by itself. In version 7.1. the security features were included in the BASIC liscense and are thus free of charge [28]. The solution to this can be restricting the access to the website like described in [29]. Kibana is extensible by using plugins to suit one particular needs.

### F. Referencing back to the Source System

After finishing searching for the right result, the user might want to get to the application or file where the result originated. To realize that, the result has to contain a data-field which contains the path, or the means to get to field in the target system. This can be an URL, SQL-Query, a filepath or other options, which has to be considered, when transfering the data to Elasticsearch from the relational databases. The additional data field has to be processed by either the frontend or backend, for example to visualize that the result is clickable and leads to the target application.

## IV. Prototype

This section presents a multi-source search engine based on the concepts presented earlier. Each source system must be considered individually. The requirements for searching and displaying the data depend heavily on the use case. Therefore, for a proof of concept, a genetic design was chosen that can be adapted as desired and considers different aspects. All sources and an instruction manual are published under the Open-Source MIT-Licence [30].

### A. Data Sourcing

First of all, we had to obtain data, for which we selected different data sets. As a first exemplary dataset we used a data model, that deals with books [10]. The next dataset we used was a very generic dataset. It consists of different attributes that reflect the different data types and is created randomly. In this way we want to give the user of our prototype a picture of a typical and abstract use.

### B. Technologies

The following technology stack was used for the implementation. Docker-compose was used for the implementation so that the prototype can be easily tried out by anyone [31]. Docker serves as the basis for this [32]. The following technologies were all used in docker containers. Which was managed by a docker-compose. As relational databases, we have used both a PostgresSQL [33] and a MySQL database [34]. For auxiliary work, such as creating tables, automatic data generation and an event-based solution, Python was used [35]. In addition, Elasticsearch [36], Kibana [26] and Logstash [37] were used.

### C. Architecture

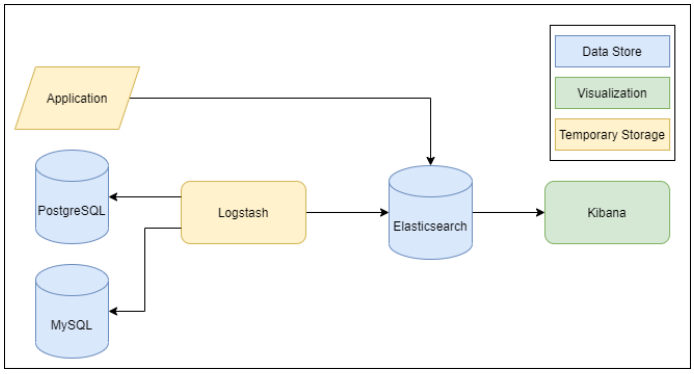


Fig. 8. Architecture of the protoype environment.

Our architecture is shown in 8 and is structured as follows. Three source systems are available. One being a MySQL database and the second a PostgreSQL. In addition, there is a Python application that creates randomized data at different time intervals. The data from the relational databases is retrieved using Logstash. The

data is written from Logstash directly to the Elasticsearch database. Elasticsearch was set up as a single node in this scenario. The Python application writes its data directly to Elasticsearch, using a push mechanism via Elasticserach's Rest API. Kibana serves as a visualization UI of the data that has subsequently been collected in Elasticsearch. Kibana is specialized for Elasticsearch, also accesses via the REST API. As mentioned before, the data from the relational databases is being written to Elasticsearch using Logstash. Three pipelines are provided for this purpose:

1) To initialize the MySQL database.
2) To propagate changes to the MySQL database.
3) To initialize the PostgresSQL database.

Tracking the changes was solved in our example via Tiggers and by means of a continuous polling interval. All data fields were indexed using the default analyzer, which was then also used for the search.

### D. Possible Extensions

The prototype environment can be extended as follows. Add more sources. Set up Elasticsearch as a cluster. Apply specific analyzers to meet different search needs. Use a more complex data model. Use different data sourcing technologies. Use other methods to keep the data up to date.

### E. Conclusion on the Prototype

The prototype shows how different source systems, each with its own data model and query language, can be bundled into a central search engine. Any source system can be used. In the same way the search can be adapted to special user needs, as well as how up-to-date the data is. All these things are very application specific and therefore the prototype serves as a general proof-of-concept.

## V. Results and Analysis

Whenever possible an event based solution should be used, but this is often not the case because the operational business should not be influenced. Also an on top solution is required, because the data has to be taken from the data stores of the individual applications. Here, the quality requirements of the target systems must be taken into account so that won't be influenced. This solution provides higher latency for indexing, since it goes through a data store. Both methods were considered in the prototype. They each target different requirements, but do not affect the later steps.

Most of the effort is in the continuous procurement of the data. Not only must a possible solution be found, but the solution must not strongly influence the source system. Different optimization variants for data collection from source systemsem are possible. We recommend a staging area, as this can also be used by other systems (for example a DWH) and not only for the search engine. If it is clear that only the search engine will use the staging area, it is recommended to take only the attributes from the source that will be indexed later to reduce the overhead. The default settings that were used in the prototype gave good results, but the possibility of optimization is there for many languages and is a positive thing to note.

Visualization is in many cases possible out-of-the-box using Kibana and also allows many features such as the creation of interactive dashboards. Another option is the development of an own visualization using the built-in Rest-API. The connection to the source systems is possible in different ways and can range from a path to the source that is indexed as a field, to the integration of the search in the application itself and is an individual decision which should be made based on the requirements.

It can be seen that there is no lack of possibilities here, but that different requirements are the challenge. Both the operational side should be influenced as little as possible and the search engine wants to be as performant as possible. Here it is necessary to find a compromise, if the entire infrastructure is not to be changed, to an event-based solution or similar.

The components used are designed to be horizontally scalable so that any performance requirement can be met. The components can also be operated in the cloud, and there are multiple providers who offer these services. However, the scalability is also limited to the source systems and their requirements if the solution is used as an on-top variant or hybrid.

Security requirements can be implemented using Elasticsearch; there is an authorization service with access control. It should be noted that this results in a double maintenance effort. The rights control can be adopted one-to-one from the source systems in the indexes, but it is not recommended. We recommend that only non-critical data is used for the search and only if it is not otherwise possible to exhaust the rights management possibilities.

A widely accepted opinion is that open source software (OSS) systems are more secure and of higher quality than proprietary competitors. This contrasts with support, which is often perceived to be less available with OSS [38]. We do not see this disadvantage with Elasticsearch, as Elastic (the software's publisher) offers a wide range of support options [39].

## VI. Conclusion

In this paper we have worked out the basics which are necessary to create a multi-source search engine based on elasticsearch. An essential part of the work that has to be done is the data acquisition, which is similar to a data warehouse. Furthermore, the indexing of the data has to be dealt with, as well as, if necessary, suitable analyzers, if an optimized search is to be made possible. The basics were completed with the visualization of the data and possibilities for the integration of the source systems were shown. Furthermore, a prototype was developed, which will serve as a proof of concept. This prototype includes

some but not all of the concepts presented here and is often based on the default settings.

## VII. References


[1] S. Gupta and R. Rani. "A comparative study of elasticsearch and CouchDB document oriented databases". In: *2016 International Conference on Inventive Computation Technologies (ICICT)*. Vol. 1. 2016, pp. 1–4. DOI: 10.1109/INVENTIVE.2016.7823252.

[2] U. Taware and N. Shaikh. "Heterogeneous Database System for Faster Data Querying Using Elasticsearch". In: *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. 2018, pp. 1–4. DOI: 10.1109/ICCUBEA.2018.8697437.

[3] D. Chen et al. "Real-Time or Near Real-Time Persisting Daily Healthcare Data Into HDFS and ElasticSearch Index Inside a Big Data Platform". In: *IEEE Transactions on Industrial Informatics* 13.2 (2017), pp. 595–606. DOI: 10.1109/TII.2016.2645606.

[4] Oleksii Kononenko et al. "Mining Modern Repositories with Elasticsearch". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 328–331. ISBN: 9781450328630. DOI: 10.1145/2597073.2597091. URL: https://doi.org/10.1145/2597073.2597091.

[5] *Secure a clusteredit*. Accessed on 12.02.2021. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/secure-cluster.html.

[6] H. Yin and D. Fengdong. "Design and Implementation of Meteorological Big Data Platform Based on Hadoop and Elasticsearch". In: *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. 2019, pp. 705–710. DOI: 10.1109/ICCCBDA.2019.8725660.

[7] S. S. Badhya et al. "Natural Language to Structured Query Language using Elasticsearch for descriptive columns". In: *2019 4th International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS)*. Vol. 4. 2019, pp. 1–5. DOI: 10.1109/CSITSS47250.2019.9031030.

[8] H. AliEl-Sappagh. *A proposed model for data warehouse ETL processes - ScienceDirect*. https://www.sciencedirect.com/science/article/pii/S131915781100019X/?imgSel=Y. (Accessed on 02/17/2021). July 2011.

[9] Rackspace. *Untitled*. https://www.objectrocket.com/wp-content/uploads/2018/10/OR_Whitepaper_MySQL_to_Elasticsearch.pdf. (Accessed on 02/11/2021). Oct. 2018.

[10] Redouane Achouri. *How to synchronize Elasticsearch with MySQL*. Accessed on 17.02.2021. Nov. 2020. URL: https://towardsdatascience.com/how-to-synchronize-elasticsearch-with-mysql-ed32fc57b339.

[11] Dan Moore. *Posting to REST APIs from mysql triggers | Dan Moore!* https://www.mooreds.com/wordpress/archives/1497. (Accessed on 02/16/2021). July 2014.

[12] btelligent. *Staging-Area: Potentiale gegenüber Quellsystemen - b.telligent*. https://www.btelligent.com/blog/staging-area-potentiale-gegenueber-quellsystemen/. (Accessed on 02/17/2021).

[13] Stefan Luber and Nico Litzel. *Was ist Logstash?* https://www.bigdata-insider.de/was-ist-logstash-a-939698/. (Accessed on 02/14/2021). June 2017.

[14] Tarun Manrai. *Things you should know about Apache NiFi | by Tarun Manrai | Medium*. https://medium.com/@manrai.tarun/things-you-should-know-about-apache-nifi-9dd7160baf8. (Accessed on 02/16/2021). Feb. 2020.

[15] KnowledgeHut. *What are the Advantages & Disadvantages of Apache Spark?* https://www.knowledgehut.com/blog/big-data/apache-spark-advantages-disadvantages. (Accessed on 02/19/2021). Jan. 2020.

[16] *Elasticsearch API*. Accessed on 17.02.2021. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/rest-apis.html.

[17] *Kafka*. Accessed on 17.02.2021. URL: https://kafka.apache.org/intro.

[18] *Storm*. Accessed on 17.02.2021. URL: https://storm.apache.org/.

[19] *Storm Elasticsearch Ingetration*. Accessed on 17.02.2021. URL: https://storm.apache.org/releases/current/storm-elasticsearch.html/.

[20] S. Shukla and s. Kumar. *Learning Elastic Stack 6.0*. Birmingham, UK: Packt Publishing, 2017.

[21] Paolo Ragone. *Scaling Elasticsearch*. Accessed on 18.02.2021. Jan. 2017. URL: https://medium.com/hipages-engineering/scaling-elasticsearch-b63fa400ee9e.

[22] *Mapping*. Accessed on 19.02.2021. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html.

[23] *Lucene Practical Scoring Function*. Accessed on 19.02.2021. URL: https://lucene.apache.org/core/4_10_1/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

[24] *Tips to secure Elasticsearch clusters for free with encryption, users, and more*. Accessed on 24.02.2021. URL: https://www.elastic.co/blog/tips-to-secure-elasticsearch-clusters-for-free-with-encryption-users-and-more.

[25] *Multi-field Partial Word Autocomplete in Elasticsearch Using nGrams*. Accessed on 17.02.2021. URL: https://qbox.io/blog/multi-field-partial-word-autocomplete-in-elasticsearch-using-ngrams/.

[26] *Kibana*. Accessed on 17.02.2021. URL: https://www.elastic.co/de/kibana.

[27] *Elastic Stack subscriptions*. Accessed on 22.02.2021. URL: https://www.elastic.co/subscriptions.

[28] *Security for Elasticsearch is now free*. Accessed on 22.02.2021. URL: https://www.elastic.co/de/blog/security-for-elasticsearch-is-now-free.

[29] M. Bajer. "Building an IoT Data Hub with Elasticsearch, Logstash and Kibana". In: *2017 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. 2017, pp. 63–68. DOI: 10.1109/FiCloudW.2017.101.

[30] *elasticsearch multisouce searchengine*. Accessed on 17.02.2021. URL: https://github.com/PanzerknackerR/elasticsearch_multisouce_searchengine.

[31] *Docker-Compose*. Accessed on 17.02.2021. URL: https://docs.docker.com/compose/.

[32] *Docker*. Accessed on 17.02.2021. URL: https://www.docker.com/.

[33] *PostgreSQL*. Accessed on 17.02.2021. URL: https://www.postgresql.org/.

[34] *MySQL*. Accessed on 17.02.2021. URL: https://www.mysql.com/de/.

[35] *Python*. Accessed on 17.02.2021. URL: https://www.python.org/.

[36] *Elasticsearch*. Accessed on 17.02.2021. URL: https://www.elastic.co/de/elasticsearch.

[37] *Logstash*. Accessed on 17.02.2021. URL: https://www.elastic.co/de/logstash.

[38] Gordon Haff Et al. *The State of Enterprise Open Source*. Zugriff am: 08.03.2021. 2021. URL: https://www.redhat.com/rhdc/managed-files/rh-enterprise-open-source-report-f27565-202101-en.pdf.

[39] *Elastic Cloud managed service features*. Zugriff am: 08.03.2021. 2021. URL: https://www.elastic.co/subscriptions/cloud.