NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Nomothesi@: Migrating the Linked Data into Elasticsearch

Zacharias Polytseris

Supervisors: **Manolis Koubarakis,** Professor
Konstantinos Plas, Research Assistant

ATHENS

APRIL 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Nomothesi@: Μετάβαση των Διασυνδεδεμένων Δεδομένων στην Elasticsearch

Ζαχαρίας Πολυτσέρης

Επιβλέποντες: **Μανώλης Κουμπαράκης,** Καθηγητής
Κωνσταντίνος Πλας, Βοηθός Ερευνητής

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2024

BSc THESIS

Nomothesi@: Migrating the Linked Data into Elasticsearch

Zacharias Polytseris
S.N.: 1115201700129

SUPERVISORS: **Manolis Koubarakis,** Professor
Konstantinos Plas, Research Assistant

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Nomothesi@: Μετάβαση των Διασυνδεδεμένων Δεδομένων στην Elasticsearch

Ζαχαρίας Πολυτσέρης
Α.Μ.: 1115201700129

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Μανώλης Κουμπαράκης,** Καθηγητής
Κωνσταντίνος Πλας, Βοηθός Ερευνητής

ABSTRACT

The main objective of this thesis was the back-end redevelopment of the Greek Legislation Platform, Nomothesi@ and an evaluation performance test between the two implementations. The platform used a triple store for legislative RDF data which got replaced with an Elasticsearch document store in order to scale up, accelerate queries and provide new features and functionality that take advantage of Elasticsearches' engine. The most important step to achieve that was the transformation of the Turtle datasets into structured JSON documents along with the creation of the new indexes' context definition. The entirety of the code regarding indexing and searching data had to be reformatted by replacing the SPARQL queries with the elastic equivalents to maintain existing functionality, resulting in a cleaner Nomothesi@ back-end that supports full-text searches, aggregations and relevance scores.

SUBJECT AREA: Search Engines, Software Engineering, Web Applications

KEYWORDS: Elasticsearch, Indexing, RDF to ndJSON, Full-Text Search, Performance Testing, REST Services, Linked Open Data

ΠΕΡΙΛΗΨΗ

Ο κύριος στόχος της πτυχιακής εργασίας ήταν η επανάπτυξη του back-end της ελληνικής πλατφόρμας νομοθεσίας, Nomothesi@, καθώς και ένα τεστ εκτίμησης απόδοσης μεταξύ των δύο υλοποιήσεων. Η πλατφόρμα χρησιμοποιούσε ένα triple store για νομοθετικά RDF δεδομένα, το οποίο μεταφέρθηκε στην Elasticsearch για την επιτάχυνση των ερωτημάτων και την παροχή νέων δυνατοτήτων και λειτουργικότητας που εκμεταλλεύονται τη μηχανή της Elasticsearch. Το πιο σημαντικό βήμα για την επίτευξη αυτού ήταν η μετατροπή των συνόλων δεδομένων Turtle σε δομημένα JSON έγγραφα, μαζί με τη δημιουργία του πλαισίου ορισμού των νέων ευρετηρίων. Όλος ο κώδικας που αφορά την αναζήτηση των δεδομένων έπρεπε να αναδιαμορφωθεί με την αντικατάσταση των SPARQL ερωτημάτων με τα αντίστοιχα της Elasticsearch, προκειμένου να διατηρηθεί η υπάρχουσα λειτουργικότητα, με αποτέλεσμα να δημιουργηθεί ένα πιο καθαρό back-end για τη Nomothesi@ που υποστηρίζει αναζητήσεις πλήρους κειμένου, συσσώρευση δεδομένων και βαθμολογίες σχετικότητας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μηχανές Αναζήτησης, Αρχιτεκτονική Λογισμικού, Εφαρμογές Δικτύου

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Elasticsearch, Indexing, RDF σε ndJSON, Αναζήτηση Κειμένου, Τεστ Αποδόσεων, REST Υπηρεσίες, Συνδεδεμένα Ανοιχτά Δεδομένα

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Manolis Koubarakis for giving me this great opportunity to work on such a big project and for being patient with me. I'm also extremely grateful to Prof. Manolis Koubarakis' assistant, Konstantinos Plas who guided and helped me during the making of this thesis. The project is dedicated to anyone still unprotected by the existing discrimination laws.

CONTENTS

1. INTRODUCTION **12**

1.1 Problem Statement 12

1.2 Aim and Objectives 12

1.3 Expected Performance Improvement 12

2. RELATED WORK **13**

2.1 Summary 13

2.2 Legislation in RDF 13

2.2.1 Querying Legislation with SPARQL 13

2.2.2 Legislation and Linked Open Data 14

2.3 The Old Nomothesi@ 15

2.4 Introducing Elasticsearch 15

2.4.1 Elasticsearch with JSON 16

2.4.2 Legislation in JSON-LD 16

3. DATASET GENERATION **18**

3.1 Choosing a file format 18

3.2 Serializing to ndJSON 19

3.3 Dataset Deserialization 20

4. BACK-END REDEVELOPMENT **23**

4.1 Document Indexing 23

4.2 Elastic Query Builder 24

4.3 Hits Parser 25

4.4 SPARQL to Elastic Queries 25

4.5 Lucene Endpoint 27

5. NEW ELASTIC FEATURES **28**

5.1 Exact Phrase Match 28

5.2 Field Priority Match 29

6. QUERY PERFORMANCE TEST 30

6.1 Query Latency 30

6.2 Response size 30

7. CONCLUSIONS 32

ABBREVIATIONS - ACRONYMS 33

APPENDICES 33

A. Nomothesi@ Web Application 34

REFERENCES 35

LIST OF FIGURES

3.1 'RDF data conversion to JSON documents' 18

4.1 'Indexing time dependence on bulk size' 23
4.2 'SPARQL query conversion to Elasticsearch DSL' 24
4.3 'Elasticsearch Endpoint with Lucene Syntax example' 27

5.1 'Use case: retrieve legal documents with the exact word given' 28
5.2 'Use case: find the legal document with a given id or FEK code' 29

LIST OF TABLES

6.1 Time performance results with Elasticsearch implementation. 30
6.2 Response size results with Elasticsearch implementation. 31

1. INTRODUCTION

1.1 Problem Statement

With a fast paced data increase every year in the form of new legislation, search platforms need to maintain a standard of performance. Alongside lawyers, common citizens have started using tools for legislative research, increasing the expected amount of users and requiring new, complex search features. A platform for legislative search like Nomothesi@, needs to modernise and adapt to the current society's demands in order to educate and raise awareness of law enforcement.

1.2 Aim and Objectives

The main objective of the thesis was the modernisation of Nomothesi@ by replacing Solr/Lucene as their search engines with Elasticsearch. More specifically, Elasticsearch would help with the scalability of the platform in order to handle even larger datasets with ease and it would provide new possibilities for more complex searches and features that take advantage of its fast text search options.

An additional objective was the practical comparison between the two engines and implementations for a variety of queries to observe if our hypothesis about elasticsearch performance is correct.

1.3 Expected Performance Improvement

Based on the documentations of each technology, we expect Elasticsearch to perform better with a larger dataset since it is designed for large scale applications. Even though they have similar use cases, Elasticsearch should be faster with complex full-text searches as well as statistical queries because of its automatic data sharding, replication and node discovery.

Most of the features of Nomothesi@ involve the retrieval of all information related to a specific Uniform Resource Identifier (URI), either representing a Legal Document or an Entity. RDF data work exceptionally well with such queries, therefore we expect the old implementation based on Lucene to be faster or equal to the Elasticsearch implementation. Elasticsearch is not as fast as triple stores for such queries since join operations are expensive and the use of multiple queries is not recommended. The solution for this problem that gives Elasticsearch a chance to compete is the deserialization of the JSON documents. Since we can afford having redundant data, we prefer to merge documents and fields to reduce the need for relations between them.

2. RELATED WORK

2.1 Summary

The thesis was based on a few previous works on the platform Nomothesi@, namely "Nomothesi@: Greek Legislation Platform" (2014), "Nomothesi@ API: Re-engineering the Electronic Platform" (2015) and most recently "Re-engineering Nomothesi@ API Web Application: Improvements and Support of new features" (2018). The platform provided support for various search filters, retrieval of most recent/viewed legal documents, statistical information in graph form, retrieval of child legislative documents(articles, paragraphs etc.), a customisable query feature for experienced users and more. The already existing SPARQL Endpoint was used as a blueprint to maintain any existing functionality with the Elasticsearch engine while no significant changes were needed for the Structure Model.

2.2 Legislation in RDF

RDF stands for Resource Description Framework. It's a data model for representing information on the web, particularly designed for describing resources and their relationships in a machine-readable format. RDF provides a way to structure and organize data using triples, which consist of three parts:

1. **Subject:** The resource being described.
2. **Predicate:** The property or attribute of the resource.
3. **Object:** The value of the property or attribute.

Legislative data are commonly distributed in a RDF format. The canonicalization of the data to a directed graph by the use of RDF allows for compact and encoded datasets that can be searched partially and in parallel. RDF also provides great flexibility which is important for legislation due to the vastly wide range of data types and relationships between entries. Finally, it is not uncommon for different countries to research each other's legal documents and therefore a universal and standardized way to represent data, like RDF, is invaluable.

The global need for legislative tools providing easy and fast search capabilities to citizens has lead to the birth of many platforms like legislation.gov.uk which use linked data formats to organise legislative data and metadata.

2.2.1 Querying Legislation with SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a semantic query language for databases, able to retrieve and manipulate data stored in Resource Description Framework (RDF) format. SPARQL allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns. Because of that, it is recognized as one of the key technologies of the semantic web. The main use cases for SPARQL are:

Semantic Web and Linked Data query and retrieve data from interconnected RDF graphs, enabling applications to navigate and explore linked data.

Metadata Management SPARQL enables efficient querying of this metadata, facilitating tasks such as resource discovery, classification, and aggregation.

Ontology-based Reasoning SPARQL can be used in conjunction with ontology reasoning engines to infer new knowledge from existing RDF data based on semantic rules defined in ontologies.

Despite the positives of SPARQL, as the datasets become more complex it shows drawbacks regarding:

Long-Term Maintenance It is hard to reach dynamic data that is updated daily (that is the nature of legislative data, new documents are uploaded daily).

User Accessibility It has a big learning curve and it is not easy for an average user to construct a custom query.

2.2.2 Legislation and Linked Open Data

Linked Data is a method of publishing structured data on the web in a way that allows it to be interlinked and interconnected with other data sources.

Linked Open Data (LOD) is a methodology for publishing and interlinking structured data on the web according to specific principles and standards. It builds upon the concept of Linked Data, which was proposed as a way to create a web of data that can be easily accessed and understood by both humans and machines.

Linked Open Data (LOD) is commonly used for legislation by platforms similar to Nomothesi@ due to several reasons:

Interoperability Legislation often spans multiple jurisdictions and legal systems. LOD provides a standardized way to represent and link data across different sources and formats, enabling interoperability between diverse legal datasets. This interoperability facilitates comparisons between different laws, regulations, and legal documents.

Semantic Web Principles LOD adheres to Semantic Web principles, which emphasize the use of standardized vocabularies (ontologies) and machine-readable formats (RDF, OWL) to represent and link data on the web. By applying these principles to legislation, legal documents become more accessible and understandable to both humans and machines.

Transparency and Accessibility LOD promotes transparency and accessibility by making legislative information available in a structured, machine-readable format. This enables citizens, researchers, policymakers, and developers to access, analyze, and reuse legal data more easily, fostering greater transparency, accountability, and civic engagement.

Overall, Linked Open Data offers a powerful framework for representing, publishing, and interlinking legislative information, thereby enhancing its accessibility, interoperability, and utility for various stakeholders in the legal domain. Therefore, Nomothesi@ benefits greatly from the maintenance of said principals for its data presentation.

2.3 The Old Nomothesi@

The previous version of Nomothesi@ used a number of datasets in Turtle format for its data. Nomothesi@ uses Semantic Web Technologies by developing and OWL ontology for Greek Legislation, namely Nomothesia ontology which follows all standards and adopts the ELI framework.It had implemented both a RDF4J and Lucene index to store the tuples which then were queried using SPARQL. The fact that both indexes were used, made the platform complex to maintain or distribute, therefore the need for a single index was created.

Most queries of the SPARQL Endpoing used by Nomothesi@'s REST API were saved as String literals which made the code hard to read and maintain. In a later chapter it is explained how a query builder for Elasticsearch helped with the construction of queries and the removal of repetitive code.

Nomothesi@ provided the user with search functionalities for greek legislation, eu legislation and entities which consisted of:

- Search legal documents or entities by keywords in their text fields.
- Search legal documents or entities by exact match in various fields.
- Search legal documents or entities by a date range for fields with a date data type.

The user could then view the full details of any result since the SPARQL Endpoint returned all tuples and relations of the given URI. For legal documents that means retrieving all child legislative data (such as articles, paragraphs and passages) in a constructed form which the user then could download in a variety of file formats such as PDF.

2.4 Introducing Elasticsearch

Elasticsearch is an open-source, distributed search and analytics engine built on top of Apache Lucene. Elasticsearch is designed to handle large volumes of data and provide near real-time search and analytics capabilities.

Elasticsearch supports the scalability of a platform by:

Horizontal Scalability Users can add more nodes to an Elasticsearch cluster to distribute the workload and data across multiple servers. This approach allows the system to handle increased indexing and search demands by simply adding more hardware to the cluster.

Sharding Elasticsearch divides data into smaller units called shards. Each shard is a self-contained index that can be distributed across nodes in the cluster. Sharding enables parallel processing of data, improving both indexing and search performance.

Data Partitioning Elasticsearch employs a hash-based distribution mechanism to evenly distribute data across shards and nodes. This ensures that each node in the cluster carries a balanced portion of the data, preventing hotspots and optimizing resource utilization.

Elasticsearch is mostly used for text-based search cases. The greater advantage compared to RDF with SPARQL querying is the ability to perform full-text search with flexibility and speed. Legislative data is vast and mostly consist of text, therefore many use cases of the platform regarding text-based search can benefit greatly from Elasticsearch.

Elasticsearch can also be used for Analytics. It provides aggregations, which allow users to perform analytics on their data in real-time. Aggregations can be used to compute metrics such as counts, sums, averages, percentiles, and more across large datasets, facilitating data analysis and visualization. This is vital to maintain existing features of Nomothesi@ and simplify analytics for potential new user needs.

2.4.1 Elasticsearch with JSON

One of the file formats supported by Elasticsearch is JSON (JavaScript Object Notation). JSON is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. JSON is primarily used to transmit data between a server and a web application. It is commonly used for REST APIs for several reasons:

Simple Syntax JSON has a simple and intuitive syntax consisting of key-value pairs and arrays. This simplicity makes it easy to work with for both developers and machines, reducing the complexity of parsing and generating JSON data.

Native Support Most modern programming languages have built-in support for parsing and generating JSON data. This native support simplifies the process of working with JSON in applications and frameworks, reducing the need for external libraries or dependencies.

RESTful Principles JSON aligns well with the principles of Representational State Transfer (REST), which emphasize stateless communication, resource-based URLs, and uniform interfaces. JSON's simplicity and flexibility make it a natural choice for representing resources and their state in RESTful APIs.

2.4.2 Legislation in JSON-LD

Ideally, the new Nomothesi@ platform can benefit from both Elasticsearch and Linked Open Data principles while maintaining compatibility with the RDF formats that are used for distributing legislation. This can be achieved by the use of JSON-LD (JSON for Linking Data).

JSON-LD (JavaScript Object Notation for Linked Data) is a lightweight data interchange format for expressing Linked Data using JSON. It provides a way to serialize structured data in a manner that is both human-readable and machine-readable, while also supporting the principles of Linked Data, which aim to enable the publishing and interlinking of structured data on the web. JSON-LD extends JSON by adding context and semantics to the data, allowing it to be linked to other resources on the web. The key features of JSON-LD are:

Context JSON-LD documents include a context object that defines the meaning of terms used within the document. This context provides mappings between JSON keys

and their corresponding IRIs (Internationalized Resource Identifiers), as well as additional information such as data types and language tags.

Linked Data Principles JSON-LD follows the principles of Linked Data, including the use of dereferenceable URIs to identify resources, the inclusion of links to related resources, and the use of standard vocabularies and ontologies to describe data.

Interoperability JSON-LD promotes interoperability by providing a standardized way to represent and exchange Linked Data across different systems and platforms.

The compatibility it has with RDF means that the platform can easily convert existing RDF datasets to JSON-LD which are then available for insertion into Elasticsearch. Another significant advantage compared to RDF is the familiarity most users have with JSON and the readability it provides.

```
{
    "@context": {
        "owl": "http://www.w3.org/2002/07/owl#",
        "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
        "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
        "xsd": "http://www.w3.org/2001/XMLSchema#"
    },
    "@graph": [
        {
            "@id": "http://el.dbpedia.org/resource/resource_1",
            "rdfs:label": [
                {
                    "@language": "en",
                    "@value": "label of resource_1"
                }
            ]
        }
    ]
}
```

Listing 2.1: Example of the content of a JSON-LD file

3. DATASET GENERATION

3.1 Choosing a file format

ndJSON (Newline Delimited JSON) [5] is a simple and human-readable data interchange format that is used for storing structured data as a sequence of JSON objects, separated by newline characters. Each line in an ndJSON file represents a single, self-contained JSON object.

ndJSON is a commonly used format for indexing data into Elasticsearch for a couple of reasons:

- As a line-by-line format, ndJSON is well-suited for streaming large datasets because it allows the user to read and process data incrementally without loading the entire dataset into memory. This is especially important when dealing with large volumes of data that may not fit entirely in memory.
- Elasticsearch provides a Bulk API that allows us to index multiple documents in a single request. ndJSON is often used to create bulk requests by appending multiple JSON documents together in a single file. This minimizes the overhead of making individual HTTP requests for each document and improves indexing efficiency.

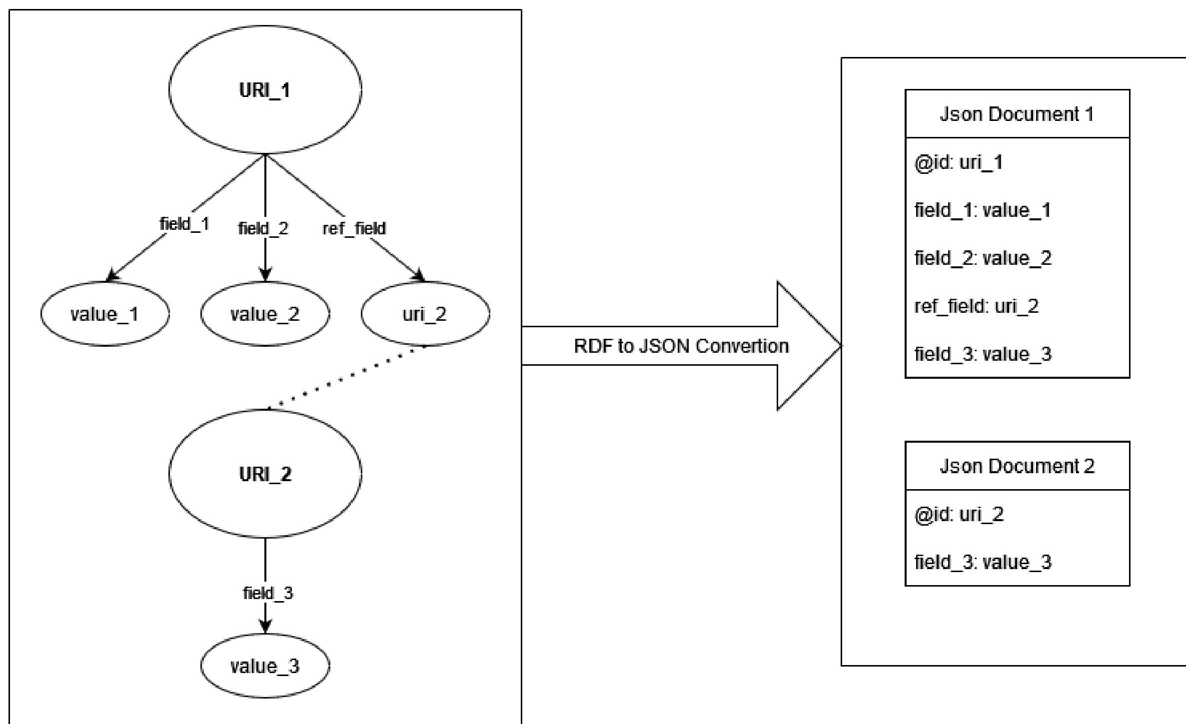


Figure 3.1: 'RDF data conversion to JSON documents'

3.2 Serializing to ndJSON

An RDF Graph consists of a set of RDF triples, each triple consisting of a subject, a predicate and an object. A JSON-LD document serializes such a set of RDF triples as a series of nested data structures. A conforming JSON-LD document consists of a single JSON object called the root object. Each unique subject in the set of triples is represented as a key in the root object. No key may appear more than once in the root object. The value of each root object key is a further JSON object whose keys are the URIs of the predicates occurring in triples with the given subject. These keys are known as predicate keys. No predicate key may appear more than once within a single object. The value of each predicate key is an array of JSON objects representing the object of each serialized triple.

```
1  from rdflib import Graph
2  import json
3  from io import StringIO
4  import os
5
6  file_handle = os.open(datasetName + 'ND.json', flags)
7  g = Graph()
8
9  with os.fdopen(file_handle, 'w') as file_obj:
10     g.parse(datasetName + ".ttl")
11
12     ld = g.serialize(format='json-ld')
13     in_json = StringIO(ld)
14     result = [json.dumps(record) for record in json.load(in_json)]
15
16     file_obj.write('\n'.join(result))
```

Listing 3.1: python serialization script for tuple data

The code creates a Graph out of the tuples and then serializes it in JSON-LD (JSON for Linking Data) format [4] to merge related tuples into one document and generate the identifier of each one, namely the URI. Then, each document is printed as one line to populate the ndJSON file.

The entirety of Nomothesi@'s data consisted of the following datasets in Turtle format: 'dbpedia.ttl', 'entities.ttl', 'eu_legislation.ttl', 'gr_legislation.ttl', 'kallikratis.ttl' Each dataset contained unique data in grouped triplets. For example the entry:

```
1 <http://legislation.di.uoa.gr/eli/gazette/EUR/2017/070>
2     <http://data.europa.eu/eli/ontology#date_publication> "2017-03-15"^^<http
    ://www.w3.org/2001/XMLSchema#date> ;
3   <http://data.europa.eu/eli/ontology#title> "EUR/2017/070"@el ;
4   <http://legislation.di.uoa.gr/ontology#eurlex_link> <http://publications.
    europa.eu/resource/cellar/55f8ca07-0949-11e7-8a35-01aa75ed71a1> ;
5   a <http://legislation.di.uoa.gr/ontology#EUGazette> .
```

holds the relations of the URI "<http://legislation.di.uoa.gr/eli/gazette/EUR/2017/070>" and more specifically the type (EU Gazette), its publication date, its title and its eurlex link. The main information for this gazette is grouped together nicely as it is, so it is easy to imagine how the finalized JSON document would look like.

After the serialization to JSON a document with the following fields and values should be created:

```
{
    "@id": "http://legislation.di.uoa.gr/eli/gazette/EUR/2017/070",
    "@type": ["http://legislation.di.uoa.gr/ontology#EUGazette"],
    "http://data.europa.eu/eli/ontology#date_publication": [
        {
            "@type": "http://www.w3.org/2001/XMLSchema#date",
            "@value": "2017-03-15"
        }
    ],
    "http://data.europa.eu/eli/ontology#title": [
        {
            "@language": "el",
            "@value": "EUR/2017/070"
        }
    ],
    "http://legislation.di.uoa.gr/ontology#eurlex_link": [
        {
            "@id": "http://publications.europa.eu/resource/cellar
            /55f8ca07-0949-11e7-8a35-01aa75ed71a1"
        }
    ]
}
```

3.3 Dataset Deserialization

The following is a different example of tuples, containing basic information about a landmark, turned into documents:

```
<http://legislation.di.uoa.gr/eli/pd/1996/295/
    article/1/paragraph/1/linea/1/reference/landmark/1>
<http://legislation.di.uoa.gr/ontology#relevant_for>
<http://legislation.di.uoa.gr/entity/landmark/1794> .

<http://legislation.di.uoa.gr/entity/landmark/1794>
a <http://legislation.di.uoa.gr/ontology#Area> .

<http://legislation.di.uoa.gr/eli/pd/1996/295/
    article/1/paragraph/1/linea/1/reference/landmark/1>
<http://www.w3.org/2000/01/rdf-schema#label>
"ΘΕΣΗ ΒΟΥΤΣΑΡΑΣ"@el , "AREA OF VOUTSARAS"@en .
```

As of now, the entirety of the information needed for an entity is not gathered under one URI. Fields like the label were found by using the entity's URI to find its reference tuple and finally get the value from there.

```
SELECT ?label
    WHERE{
        ?ref leg:relevant_for <"uri">.
        ?ref rdfs:label ?label.
    }LIMIT 1
```

Listing 3.2: SPARQL Query Sample: getting the label of an entity by its URI

With the current ndJSON documents, since join operations are not supported, the implementation must either:

1. perform two different queries
2. declare a parent/child relationship between documents.
3. declare nested fields

With RDF data, the query is perfectly fast, but the solutions that Elasticsearch provides are time consuming and costly. More specifically:

1. Large numbers of queries mean the multiplication of time needed for large requests.
2. Parent/child mappings have extra memory overhead, since ES maintains a "join" list in memory
3. Updating a single field in a nested document (parent or nested children) forces ES to re-index the entire nested document. This can be very expensive for large nested docs

The best solution in this case is to deserialize the ndJSON documents since storage cost is much cheaper than the computing cost. For this reason a deserializer project was created. Based on selected fields representing a reference to other documents, the deserializer follows the below algorithm:

```
Data: A set of ndJSON files whose rows contain a document
Result: A Deserialized ndJSON file with all documents containing referenced info
/* Iterate through the datasets and map documents based on their URI(@id
   field)                                                              */
for ds in datasets do
    create MAP (key = URI, value = JSON document in ds);
    for ds in datasets do
        for document in ds do
            if document has reference fields then
                get reference fields' ids;
                search for ids in MAP;
                print result of current document merged with mapped documents;
            end
        end
    end
end
```

Algorithm 1: Nomothesi@ ndJSON datasets Denormalizer

The following constants were created for the implementation of the denormalizer in order to minimize possible errors and operations on the queries' hits:

```
// List containing the fields we merge based on.
// An example of a field that is not included is eli:has_part to avoid
static final List<String> referenceFields = Arrays.asList(
            "http://www.w3.org/2002/07/owl#sameAs",
            "http://legislation.di.uoa.gr/ontology#relevant_for",
            "http://data.europa.eu/eli/ontology#published_in",
            "http://data.europa.eu/eli/ontology#transposed_by"
);

// Specifically for the case of field eli:transposed_by
// Only add the following information to generated document
private static final List<String> wantedTransByFields = Arrays.asList(
            "@type",
            "http://data.europa.eu/eli/ontology#date_publication",
            "http://data.europa.eu/eli/ontology#id_local",
            "http://legislation.di.uoa.gr/ontology#published_by"
);


// Fields and values not merged into the generated document
private static final List<String> excludedFromMergeFields = Arrays.asList(
            "@id",
            "http://geo.linkedopendata.gr/gag/ontology/has_geometry",
            "http://data.europa.eu/eli/ontology#date_publication",
            "http://data.europa.eu/eli/ontology#transposed_by"
);

private static final List<String> excludedFromMergeValues = Arrays.asList(
            "http://legislation.di.uoa.gr/ontology#GovernmentGazette"
);
```

Listing 3.3: Denormalizer constants

After the denormalizer project processes all the datasets, the output is a unified dataset consisting of merged documents ready to be indexed into Elasticsearch. The parent documents that are referenced are also kept inside the final dataset since not all information is merged. For example, the field containing the geometry of a geographical area is excluded from merge and only kept in the original document since passing it along a relationship tree would be too much of a memory burden.

4. BACK-END REDEVELOPMENT

4.1 Document Indexing

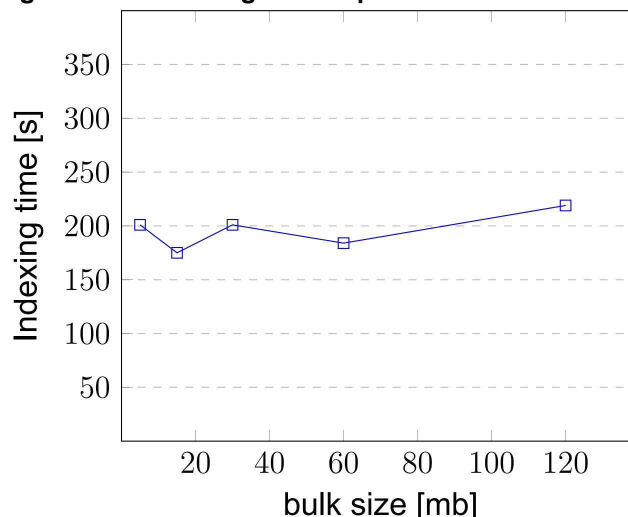
Mapping is the process of defining how a document, and the fields it contains, are stored and indexed. Each index can be defined by a mapping definition, which contains the document fields' data types, metadata and customization. In Elasticsearch, mappings can be either dynamic, meaning created while indexing documents, or explicit for better optimization. Nomothesi@'s index is created with an explicit mapping which helps with the support of different data formats and the exclusion of unused fields when calculating a document's score in order to improve query latency.

```
{
  "mappings": {
    "properties": {
      "@id": {
        "type": "keyword"
      },
      "@type": {
        "type": "keyword"
      },
      "http://dbpedia.org/ontology/birthPlace": {
        "type": "object",
        "enabled": false
      },
      "http://data.europa.eu/eli/ontology#date_signature": {
        "properties": {
          "@type": { "type": "keyword" },
          "@value":  { "type": "text" }
        }
      },
```

Listing 4.1: Snipet of Nomothesi@'s index mapping

The platform uses the Bulk API of Elasticsearch for indexing. It enables the platform to make multiple indexing operations in a single API call.

Figure 4.1: 'Indexing time dependence on bulk size'



Using the Bulk API speeds up the indexing process significantly since the number of requests is divided and the batch of documents is indexed simultaneously thanks to Elasticsearches' parallel processing. Multiple threads were also used to increase concurrency.

Elasticsearch limits the maximum size of a HTTP request to 100mb by default so it is important for the document batches to not exceed that limit. The next logical step was to construct the batches based on byte size rather than a static size since they can differ greatly. Documents with geographical data or large text would force the batch size to shrink and therefore slow down indexing of small documents. Finally, a dynamic approach is the best safety measure for possible future documents that may break indexing due to abnormally large sizes.

Elasticsearch suggests to start with a bulk size around 5–15 MB and slowly increase it until there are no performance gains anymore. Indeed, 15mb gave the best time in the local environment and no big differences were present even with a larger bulk size.

4.2 Elastic Query Builder

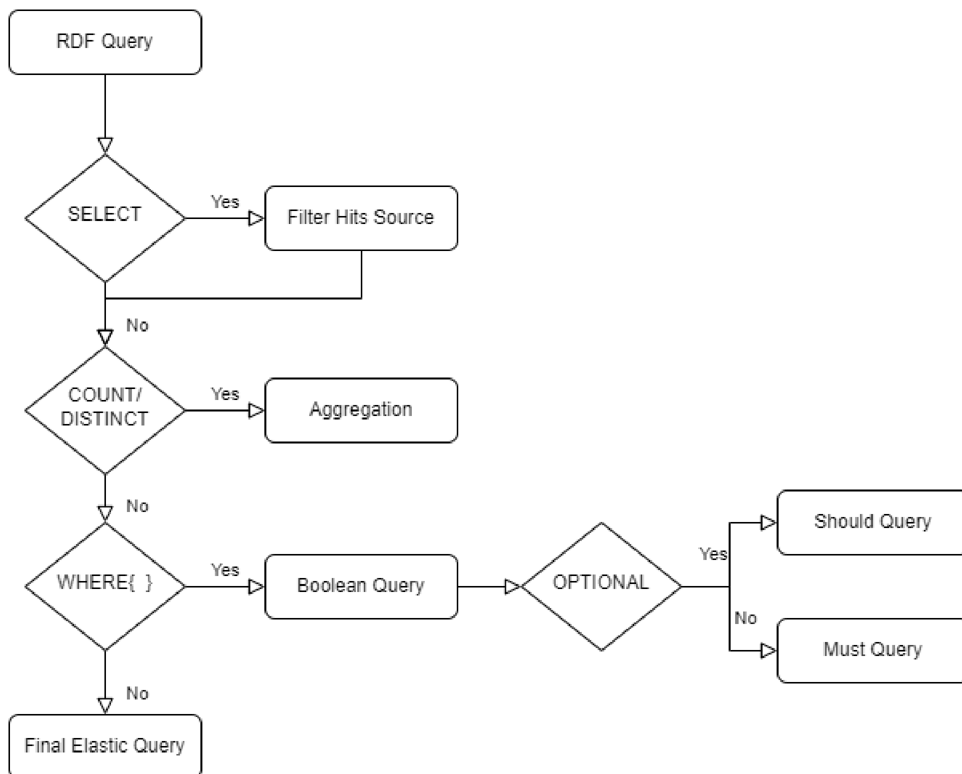


Figure 4.2: 'SPARQL query conversion to Elasticsearch DSL'

The Elasticsearch Java REST Client is a powerful tool that provides flexible and easy to use solutions for programmatic request generation. In order to reduce redundant code and increase readability, we created a builder class that takes care of the details of creating queries or aggregations, the ElasticQueryBuilder.

```
1 public Query matchQuery(String field, String value) {
2     value = replaceDebugUri(value);
3     String finalValue = value;
4 
5     return MatchQuery.of(q -> q
6             .field(field)
```

```
            .query(finalValue)
    )._toQuery();
}

public Query booleanQuery(List<Query> queries, ELASTIC_OPERATOR operator) {
    switch (operator) {
        case FILTER:
            return BoolQuery.of(q -> q
                    .filter(queries)
            )._toQuery();
        case SHOULD:
            return BoolQuery.of(q -> q
                    .should(queries)
            )._toQuery();
        case MUST_NOT:
            return BoolQuery.of(q -> q
                    .mustNot(queries)
            )._toQuery();
        case MUST:
        default:
            return BoolQuery.of(q -> q
                    .must(queries)
            )._toQuery();
    }
}
```

Listing 4.2: Sample of methods returning different Elastic queries

After any complex query is built, the application can finally create a Search Request with a variety of configurations like: hits number. field selection for more compact hits. hit order based on a selected field. skip hits after a selected identifier for batch requests that retrieve a lot of data.

- Hits number.
- Field selection for more compact hits.
- Hit order based on a selected field.
- Skip hits after a selected identifier for batch requests that retrieve a lot of data.

4.3 Hits Parser

The results of a search request contain a list of hits. Each hit has the information of a document in JSON form. To create the appropriate model entity from a JSON, the need for a parser was created. The documents can be mostly categorized as Entities or Legal Documents and therefore different Parser classes were created.

The Parsers resulted in a fast and straight-forward way of handling data, without the hustle of iterating through large amounts of tuples.

4.4 SPARQL to Elastic Queries

Converting the queries into Elasticsearch Query DSL (Domain-Specific Language) [3] required an in-depth knowledge of the possible information within a single document to min-

imize the amount of requests needed.
We will now proceed to construct some sample queries to explain the changes made.

```
1 SELECT *
2 WHERE {
3     OPTIONAL{ <" + uri + "> rdf:type ?type. }
4     OPTIONAL{ ?ref leg:relevant_for <" + uri + ">. ?ref rdfs:label ?reflabel.
   FILTER(langMatches(lang(?reflabel), \"el\"))}
5 }
```

Listing 4.3: Retrieve type/label of entity RDF query

The above query should retrieve all types of the unique identifier given as well as the label of the one referencing it. After the deserialization, both of these fields should be in a single document, more specifically one that references the given uri and therefore a simple Match query should give us exactly that.

```
1 Query searchQuery = queryBuilder.matchQuery(leg + "relevant_for.@id", uri);
```

Listing 4.4: Retrieve type/label of entity Elastic DSL query

After creating the query, we can easily pass it into a search request mentioning the specific fields that need to be retrieved to achieve faster completion times.

Now what if we want to know the type of a Legal Document's parts? The RDF query would be as follows:

```
1 SELECT ?part
2 WHERE {
3     <" + uri + "> eli:has_part+ ?part.
4     ?part rdf:type ?parttype.
5 }
```

Listing 4.5: Retrieve type of Legal Document's parts RDF query

In this case, we will not find a document that contains all the info we need since merging all Legal Documents with their parts would create massive documents and an over-complicated index context. Therefore, we need to first retrieve the URIs of the Legal Document's parts and then, with a second request, get their types. The Terms query is used to tell Elasticsearch that any of the values provided are equally fitted for a match.

```
1  Query searchQuery =
2          queryBuilder.booleanQuery(Arrays.asList(
3                      queryBuilder.matchQuery("@id", uri),
4                      queryBuilder.existsQuery(eli + "has_part")
5          ), ELASTIC_OPERATOR.MUST);
6
7  /* Get request hit and parse "has_part" field. Save the values (URIs) in a
    list */
8
9  Query searchQuery =
10         queryBuilder.booleanQuery(Arrays.asList(
11                     queryBuilder.termsQuery("@id", uriList),
12                     queryBuilder.existsQuery("@type")
13         ), ELASTIC_OPERATOR.MUST);
```

Listing 4.6: Retrieve type of Legal Document's parts Elastic DSL query

With the above example we can clearly see Elasticsearch'es disadvantage with RDF Data since multiple queries are needed for a multitude of requests, most noticeably here with multi-dimensional tree structures like Legal Documents.

Regarding statistics queries, Aggregations provided a great way to find numbers of documents with specific traits. The triple store's aggregations with the `COUNT` or `GROUP_CONCAT` functions were replaced with the equivalent Elastic Aggregations for Terms or Date Histograms. This allowed to maintain fast results when counting documents matching given terms or within a wanted date range.

4.5 Lucene Endpoint

Nomothesi@ gave the ability to experienced users to write and run custom SPARQL queries and retrieve results. An equivalent feature is possible with Elasticsearch since it can parse lucene queries passed to the **query_string** parameter. Users can write lucene syntax in a text field and get the relevant documents in JSON format. The limitations of the lucene syntax are the inability to search nested objects or scripted fields.

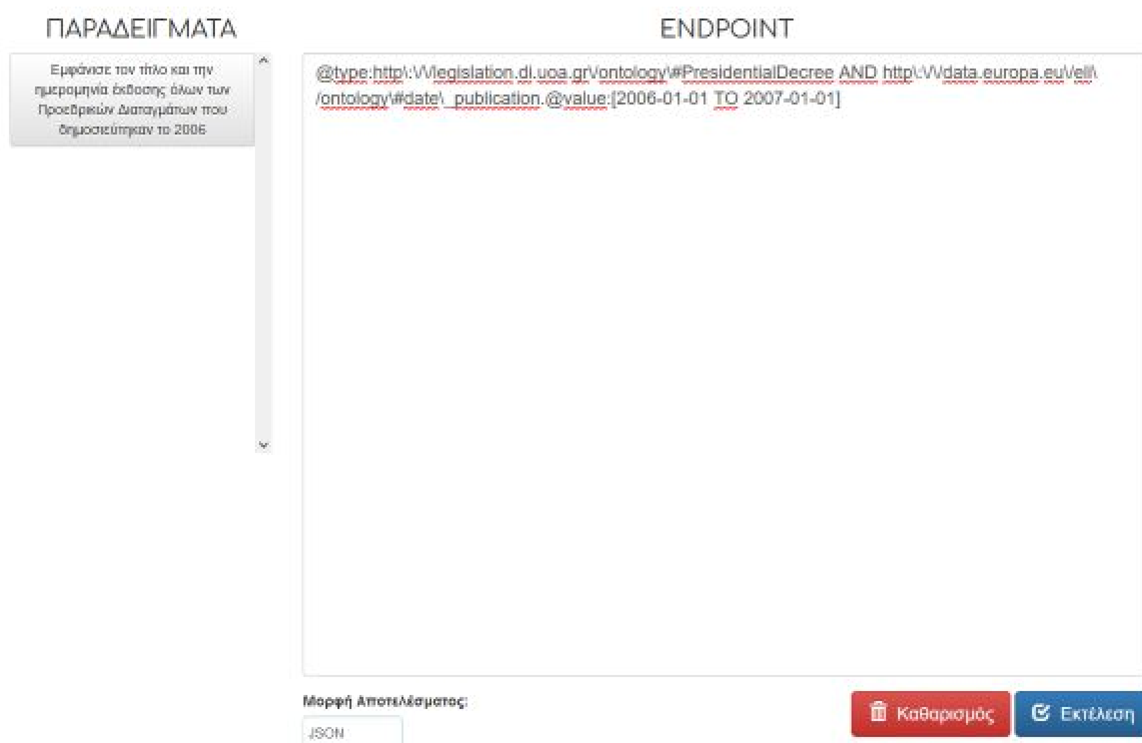


Figure 4.3: 'Elasticsearch Endpoint with Lucene Syntax example'

5. NEW ELASTIC FEATURES

5.1 Exact Phrase Match

In Elasticsearch, the MatchPhrase query is a type of full-text query that is used to search for exact phrases within text fields. It is designed to match documents where the specified phrase appears in the text exactly as it is provided in the query from the user. Using the MatchPhrase Query, the user can now search for hits with the exact phrase in their label or equivalent main text field. Like other text-based queries in Elasticsearch, the text in the specified field is tokenized during indexing and searching. However, the match_phrase query operates on the tokens to maintain the word order.

MatchPhrase is particularly useful when the exact sequence of words matters, such as in the case of a quote or a specific phrase. Some use cases expected to be covered by this feature are:

Precision Searching Legal documents often contain precise terminology and language. An exact phrase search allows users to find specific clauses, definitions, or provisions without sorting through irrelevant results.

Interpretation of Legal Precedents When researching case law or precedent, lawyers may need to locate instances where a particular phrase or wording has been used in previous judgments.

Research and Academia Legal scholars and researchers often need to conduct precise searches to support their academic work or to explore specific legal concepts.

Legal Writing and Drafting When drafting legal documents, lawyers may want to ensure consistency in language usage or adhere to specific wording used in precedents.

In each of these scenarios, the ability to perform precise searches for exact phrases within legal documents is crucial for saving time, ensuring accuracy, and supporting informed decision-making in the legal field.

Figure 5.1: 'Use case: retrieve legal documents with the exact word given'

5.2 Field Priority Match

If the user wants to search a phrase in a document, Elaticsearch also provides the MultiMatch Query. With this functionality, it is now possible to search the same phrase in multiple valid fields. The MultiMatch Query is particularly useful when searching for a specific term or phrase across various fields in an Elasticsearch index. This query type is commonly used in scenarios where a broad search across different fields is needed, such as titles, descriptions, and content, to find relevant documents.

The MultiMatch query is flexible and supports various options to customize how the search is conducted. Some of the key features and options of the MultiMatch query include:

- Specifying which fields in the documents will be searched across. This can be done by providing a list of field names or by using wildcard expressions to match multiple fields.
- Different weights (boosts) can be assigned to each field to influence the relevance scoring of the documents. Fields with higher boosts contribute more to the overall relevance score of a document.
- Elasticsearch supports different match types for the MultiMatch query, including ”best_fields”, ”most_fields”, ”cross_fields”, ”phrase”, and ”phrase_prefix”.
- Skip hits after a selected identifier for batch requests that retrieve a lot of data.

Currently, the platform enables the user to search for legislative data based on the FEK code or the local id. It is possible that more use cases that can take advantage of MultiMatch may exist in the future.

ΑΠΟΤΕΛΕΣΜΑΤΑ ΑΝΑΖΗΤΗΣΗΣ
Λέξεις Κλειδιά:
Εισάγετε επιθυμητή έκφραση...
Ακριβής Αντιστοιχία Κειμένου:
Εισάγετε επιθυμητή έκφραση...
Αναζήτηση ΦΕΚ ή τοπικού id:
2690
Τύπος Νομοθεσίας:
Συμφωνία (ΣΥΜ.)
Ανακοίνωση (ΑΝΑΚ.)
Τίτλος
Κύρωση του Κώδικα Διοικητικής Διαδικασίας και άλλες διατάξεις.
Εμφανίζει από 1 μέχρι 1 των 1 εγγραφών

Figure 5.2: ’Use case: find the legal document with a given id or FEK code’

6. QUERY PERFORMANCE TEST

6.1 Query Latency

The following table shows the **Query Latency** of the new platform when using Elasticsearch as its search engine. This metric measures the time it takes for a query to be processed and return results.

page	case	time processing (sec.)
index	-	1.088
legislation details	eli/pd/2014/69	4.791
	constitution (eli/con/2008/1)	16.470
	penal code (eli/penalcode/1985/1)	29.406
search	all	0.348
	type: decision	1.924
	date from 06/2016 to 12/2023	1.868
	exact phrase: ”Τεχνολογική Συνεργασία”	0.890
	keywords: ”Τεχνολογική Συνεργασία”	2.186
entities	all	1.233
	type: landmark	0.519
gazette	-	24.175
statistics	-	7.683

Table 6.1: Time performance results with Elasticsearch implementation.

Observing the query latency of Nomothesi@ with Elasticsearch shows that the time searching legislation has a slight deviation based on the filter used. For example, a date range query is more expensive than a match query based on legislation type. Requests for legislation and entity details are noticeably slower due to the multiple search requests sent to Elasticsearch in order to retrieve all child documents in full, such as articles of a legal document.

6.2 Response size

Another useful metric for scalable search engines is the memory needed for the response. Transmitting larger amounts of data over the network can incur higher costs. Understanding the size of responses is crucial when considering the scalability and cost implications of an Elasticsearch deployment. Since Nomothesi@ needs to be scalable, a high response size might lead to the platform being too inefficient after too much data get indexed. With Elasticsearch, the respose size can be optimized with the **fields** parameter since it can filter the source to contain only the fields with needed information.

page	case	response size (bits)
index	-	128.400
legislation details	eli/pd/2014/69 constitution (eli/con/2008/1) penal code (eli/penalcode/1985/1)	1.993.912 3.568.832 20.129.216
search	all type: decision date from 06/2016 to 12/2023 exact phrase: ”Τεχνολογική Συνεργασία” keywords: ”Τεχνολογική Συνεργασία”	2.844.992 2.999.584 1.666.408 181.328 1.436.584
entities	all type: landmark	1.494.784 1.324.768
gazette	-	3.648.000
statistics	-	32.031.832

Table 6.2: Response size results with Elasticsearch implementation.

By observing the above table, it can be seen that the size of search is mostly consistent due to the set cap of 1000 legal documents/entities in the results, unless the query is relatively specific. The legal document details vary greatly since some are larger than others. Another interesting observation is that the gazette query doesnt return a large result size even if it is slow in the query latency test. This is due to the small but many queries done in order to get child document information. Queries that need multiple searches in Elasticsearch show the weaknesses of the search engine. Finally, since the statistics screen needs to process lots of documents, it retrieves a large amount of data and is probably a good candidate for optimization after the platform scales significantly.

7. CONCLUSIONS

Elasticsearch is a powerful tool, capable of hosting large amounts of data while providing a variety of optimisation options for the improvement of query performance, such as field exclusion and data sharding. Although legislative data are linked and not an ideal fit for Elasticsearch, they are ever-expanding and need a scalable search engine that can support the increasing demand while also providing the possibility to cover new user cases that were not possible before with the RDF store.

It is expected that the benefits of migrating to Elasticsearch will be more visible in the future, since with the current dataset size, no significant improvements can be observed. A larger and more demanding dataset may also create the need for better indexing and querying strategies, mostly focused on the documents' deserialization and interconnection methods.

Finally, the Elasticsearch JAVA API was also a great tool that helped with the development of clean, readable and reusable code that will most likely make the implementation of future features or optimizations easier.

Nomothesi@ has the important role of educating and providing professionals and normal citizens alike with easy access to legislation and keeping up with the ever-expanding demand is crucial. The Elasticsearch back-end will definitely assist with this goal for the foreseeable years since it was designed to make such future improvements easily implementable.

ABBREVIATIONS - ACRONYMS

RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
OWL	Web Ontology Language
LOD	Linked Open Data
JSON	JavaScript Object Notation
JSON-LD	JSON for Linking Data
NDJSON	Newline Delimited JSON
DSL	Domain-Specific Language
TTL	Terse RDF Triple Language
URI	Uniform Resource Identifier

APPENDIX A. NOMOTHESI@ WEB APPLICATION

Nomothesi@ is implemented as a Spring web MVC Project. For the purpose of this thesis a few dependency versions were upgraded. Additionally, RDF related dependencies were removed as they were no longer used.

The Elasticsearch version installed for the thesis was 8.3.3. The installation guide for linux can be found here:
`https://www.elastic.co/guide/en/elasticsearch/reference/current/targz.html`
To install the needed version, simply change the number used in the relevant commands.

A tool that was used extensively during development was Kibana. Among many things, it is used to view, query and edit indexes and documents. It is recommended to install the same version number with Elasticsearch. To install Kibana on linux follow the relevant guide here:
`https://www.elastic.co/guide/en/kibana/current/targz.html`

In order to create the ndjson dataset from .ttl files and deploy Nomothesi@ locally, the repository's READMY.md steps should be followed. Most importantly, the local folder containing the finalized dataset batch files should be changed in the application properties.

BIBLIOGRAPHY

[1] Apache lucene - query parser syntax. URL: https://lucene.apache.org/core/2_9_4/queryparsersyntax.html.

[2] Elasticsearch java api client. URL: https://www.elastic.co/guide/en/elasticsearch/client/java-api-client/current/index.html.

[3] Elasticsearch query dsl (domain-specific language). URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html.

[4] Json for linking data. URL: https://json-ld.org/.

[5] ndjson format (new line delimeter json). URL: http://ndjson.org/.

[6] Resource description framework (rdf). URL: https://www.w3.org/RDF/.

[7] Sparql 1.1 query language. URL: https://www.w3.org/TR/sparql11-query/.

[8] Apostolopoulos G. Re-engineering nomothesi@ api web application: Improvements and support of new features. 2018.

[9] Chalkidis I. Nomothesi@: Greek legislation platform. b.sc thesis. national and kapodistrian university of athens. 2014.

[10] Panagiotis Soursos Manolis Koubarakis Ilias Chalkidis, Charalampos Nikolaou. Modeling and querying greek legislation using semantic web technologies. 2017.

[11] Soursos P. Nomothesi@ api: Re-engineering the electronic platform. b.sc thesis. national and kapodistrian university of athens. 2015.

[12] Manolis Koubarakis/Eleni Tsalapati. M164 - cs2: Knowledge technologies. In *Knowledge Representation and Reasoning for the World Wide Web, i.e., on the research areas of Semantic Web and Linked Data*, 2022-2023. URL: https://cgi.di.uoa.gr/~pms509/lectures.htm.