Keyword Search over RDF using Document-centric Information Retrieval Systems

Giorgos Kadilierakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Prof. *Yannis Tzitzikas*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Keyword Search over RDF using Document-centric Information Retrieval Systems

Thesis submitted by
Student G. Kadilierakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: Student G. Kadilierakis

Committee approvals: Yannis Tzitzikas
Associate Professor, Thesis Supervisor

Dimitris Plexousakis
Professor, Committee Member

Giorgos Flouris
Principal Researcher, Committee Member

Departmental approval: Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, January 2020

Keyword Search over RDF using Document-centric Information Retrieval Systems

Abstract

There are thousands of datasets published according to the principles of Linked Data and Semantic Web. Many of those datasets, organized in RDF, are maintained either in cross-domain Knowledge Bases (e.g. DBpedia, Wikidata) or domain specific repositories (e.g. DrugBank, MarineTLO), and are mainly used through navigation and structured query languages like SPARQL. However these techniques are complex, lack flexibility and possibly require a full knowledge of the underlying ontology. As a result, these datasets are exploited by expert users only.

On the other hand, keyword search is the most widely used method for searching. Keyword search is user friendly, offers instant content access, and keyword queries support a wide range of expression while being extremely flexible. Information Retrieval systems are designed for performing efficient keyword search in large data of information, usually organized as full text documents. There are various highly performant and effective state of the art search engines readily available. Such a search engine is Elasticsearch, a distributed full text search engine that provides scalable search over any kind of textual information.

In this thesis we introduce an approach for keyword-search over RDF datasets, by adapting traditional IR techniques for both indexing and retrieval. Specifically, we test how a dominant IR engine such as Elasticsearch, can be adapted for indexing RDF data and enable keyword search. We provide a systematic analysis of different approaches to cope with the challenges of indexing and retrieving structured information and exploiting the graph capabilities of RDF. The response of the system comprises ranked RDF triples. We also provide policies for ranking the different entities that are contained in these triples, in order to support the requirements of entity search.

We report evaluation results of the different approaches in terms of: (i) the efficiency of indexing and retrieval and (ii) the quality of retrieval. We test the effectiveness of our system by evaluating the relevance of the constructed entities against the DBpedia-Entity test collection, designed for entity search over the DBpedia KB and compare our results to various state of the art systems. Our results showcase the effectiveness of the proposed user friendly approach, that exploits the powerful features of scalable state of the art search engines, and can be applied in any RDF dataset, with no prior knowledge of the domain. Results show that Elasticsearch can effectively support keyword search over RDF data, offering effectiveness comparable to that of systems built from scratch for the task per se, that use entity-oriented and dataset-specic index structures.

Αναζήτηση μέσω Λέξεων-Κλειδιών επί RDF Δεδομένων Χρησιμοποιώντας Εγγραφο-κεντρικά Συστήματα Ανάκτησης Πληροφοριών

Περίληψη

Υπάρχουν χιλιάδες σύνολα δεδομένων που δημοσιεύονται σύμφωνα με τις αρχές των Συνδεδεμένων Δεδομένων (Linked Data) και του Σημασιολογικού Ιστού (Semantic Web). Πολλά από αυτά, όντας οργανωμένα σε RDF, βρίσκονται είτε σε Βάσεις Γνώσεων ανοικτού πεδίου (π.χ. DBpedia, Wikidata) είτε σε συλλογές κλειστού πεδίου (π.χ. DrugBank, MarineTLO) και η εξερεύνησή τους είναι εφικτή μόνο μέσω συστημάτων πλοήγησης και δομημένων γλωσσών όπως η SPARQL. Ωστόσο, οι τεχνικές αυτές είναι σύνθετες, στερούνται ευελιξίας και συνήθως απαιτούν από κάποιον γνώση της οντολογίας που περιγράφει τα δεδομένα. Αυτό έχει ως αποτέλεσμα να καταλήγουν να αξιοποιούνται μόνο από ειδικούς χρήστες.

Η αναζήτηση μέσω λέξεων-κλειδιών (keyword-search) είναι η πιο ευρέως χρησιμοποιούμενη μέθοδος αναζήτησης καθώς είναι φιλική προς το χρήστη και προσφέρει άμεση πρόσβαση στο περιεχόμενο, ενώ παράλληλα διατηρεί μεγάλη εκφραστικότητα. Τα Συστήματα Ανάκτησης Πληροφοριών (Information Retrieval Systems) είναι σχεδιασμένα για την αποτελεσματική αναζήτηση μέσω λέξεων-κλειδιών πάνω από μεγάλο όγκο εγγράφων κειμενικής πληροφορίας. Για το σκοπό αυτό, υπάρχουν διαθέσιμες διάφορες εξαιρετικά αποτελεσματικές και αποδοτικές μηχανές αναζήτησης. Ένα τέτοιο παράδειγμα είναι η Elasticsearch, μια κατανεμημένη μηχανή αναζήτησης κειμένου, η οποία παρέχει δυνατότητα κλιμακώσιμης αναζήτησης σε οποιοδήποτε είδος πληροφοριών κειμένου.

Σε αυτήν την εργασία σχεδιάσαμε μία μέθοδο για αναζήτηση μέσω λέξεων-κλειδιών πανω από RDF δεδομένα, προσαρμόζοντας τις παραδοσιακές τεχνικές ανάκτησης πληροφορίας (IR) για την ευρετηρίαση και την ανάκτηση. Συγκεκριμένα, δοκιμάζουμε τρόπους με τους οποίους μια κυρίαρχη στην αγορά μηχανή αναζήτησης, όπως η Elasticsearch, μπορεί να χρησιμοποιηθεί για την ευρετηρίαση RDF δεδομένων και την παροχή αναζήτησης μέσω λέξεων-κλειδιών σε αυτά. Παρέχουμε μια ανάλυση των διαφορετικών προσεγγίσεων που ακολουθήσαμε για να αντιμετωπίσουμε τις προκλήσεις της ευρετηρίασης και της ανάκτησης δομημένης πληροφορίας και την αξιοποίηση των δυνατοτήτων που μας δίνει ο RDF γράφος. Η απάντηση του συστήματος αποτελείται από μια κατάταξη συναφών RDF τριπλετών. Επίσης, παρέχουμε πολιτικές για την κατάταξη των διαφορετικών οντοτήτων που περιέχονται στις τριπλέτες προκειμένου να υποστηριχθεί και ο στόχος της αναζήτησης οντοτήτων (entity-search).

Τα αποτελέσματα της αξιολόγησης των διαφορετικών προσεγγίσεων μας αφορούν (α) την αποδοτικότητα της ευρετηρίασης και της ανάκτησης και (β) την ποιότητα της ανάκτησης. Δοκιμάζουμε την αποτελεσματικότητα του συστήματός μας αξιολογώντας τη συνάφεια των οντοτήτων που κατασκευάζουμε πάνω από την συλλογή DBpedia-Entity, σχεδιασμένη για την αναζήτηση οντοτήτων μέσω της βάσης γνώσης DBpedia και συγκρίνουμε τα αποτελέσματά μας με διάφορα συναφή συστήματα. Στα αποτελέσματά μας παρουσιάζουμε την αποδοτικότητα της προτεινόμενης προσέγγισης, η οποία εκμεταλλεύεται τα ισχυρά χαρακτηριστικά των κλιμακώσιμων μηχανών αναζήτησης, ενώ μπορεί να εφαρμοστεί πάνω από οποιοδήποτε σύνολο δεδομένων RDF χωρίς προηγούμενη γνώση του τομέα. Τα αποτελέσματα της αξιολόγησης καταδεικνύουν ότι η Elasticsearch μπορεί να υποστηρίξει αποτελεσματικά την αναζήτηση μέσω λέξεων-κλειδιών επί δεδομένων RDF , προσφέροντας αποτελεσματικότητα εφάμιλλη εκείνης των συστημάτων που έχουν δημιουργηθεί αποκλειστικά για RDF και χρησιμοποιούν οντο-κεντρικά ευρετήρια.

Ευχαριστίες

Πρωτίστως στον επόπτη καθηγητή μου Γιάννη Τζίτζικα για την υποστήριξη και εμπιστοσύνη που μου έδειξε σε όλη την διάρκεια της συνεργασίας μας καθώς και στο ΙΤΕ - Ινστιτούτο Πληροφορικής για την χορήγηση της υποτροφίας. Πιο συγκεκριμένα, θα ήθελα να ευχαριστήσω τον Παύλο Φαφαλιό και τον Παναγιώτη Παπαδάκο που με καθοδήγησαν, στήριξαν και συνεισέφεραν ουσιαστικά στην εργασία. Χωρίς αυτούς το thesis δεν θα είχε ολοκληρωθεί. Επίσης και στο υπόλοιπο εργαστήριο και συναδέλφους που μου στάθηκαν σαν φίλοι σε όλη την διάρκεια. Τέλος, και πιο σημαντικά, στην οικογένειά μου, που με στηρίζει και βρίσκεται πάντα στο πλευρό μου.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Web of Data currently contains thousands of RDF datasets available online that includes cross-domain Knowledge Bases (KBs) like DBpedia and Wikidata, domain specific repositories like DrugBank[1] and MarineTLO [38], as well as Markup data through schema.org[2] (see [30] for a recent survey). These datasets are queried through structured query languages (SPARQL), however this is quite complex for ordinary users. Ordinary users are acquainted with keyword search due to the widely used web search engines. Faceted search is another popular paradigm for interactive query formulation, however even such systems (see [39] for a survey) need a keyword search engine as a flexible entry point to the information space. We conclude that an effective method for keyword search over RDF is indispensable.

At the same time we observe a widespread use of classical IR systems (like Elasticsearch and Solr) in different contexts. To this end in this work we investigate how these, document-centric Information Retrieval Systems (IRSs), can be used for enabling keyword search over arbitrary RDF datasets. This endeavor raises various questions, including (a) how to index an RDF dataset, (b) what to rank and how, (c) how the search results should be presented.

To investigate the above, in this paper we select one popular IR system, namely `Elasticsearch`, over which we propose methods for tackling the aforementioned questions. We report extensive evaluation results of the different options in terms of quality of retrieval and efficiency of indexing and retrieval. This allows us to provide an answer to the general question: how good an existing document-centric IR system is for keyword search over RDF? Can it be configured in a way that yields a retrieval behavior comparable with those of dedicated keyword search systems for RDF?

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the background concepts and describes the related work. Chapter 3 discusses the problem analysis along with the requirements and challenges that arise. In

[1]`https://www.drugbank.ca`
[2]`https://schema.org`

Chapter 4 we describe our approach, that we call `ElaS4RDF`, and also give an overview of the considered IR system. After that, in Chapter 5, we review the results of our experiment evaluation that includes both the quality of retrieval and system's efficiency. Finally, Chapter 6 concludes the thesis and identifies issues that are worth further research.

Chapter 2

Background & Related Work

2.1 RDF, Linked Data and the Semantic Web

RDF is a framework for describing resources on the web, easily readable by computers [3]. It is the most common framework for representing data in knowledge bases. In RDF we describe entities (or resources) that may be either a person, a place, an institution, a concept or a relation between other resources. An entity is associated to a URI (Uniform Resource Identifier) which is a string of characters according to a particular syntax [4] that uniquely identifies an abstract or physical resource. The main component of RDF is a *triple*, which consists of a *subject(s)*, *object(o)*, *predicate(p)* and is the smallest representation of a relationship that is described in p, between s and o. Subjects and predicates are URIs while objects can either be a URI or a literal value. Anonymous (blank) nodes, or simply b-nodes, are also supported and can occur as subjects or objects of the triples. The information contained in any RDF collection, forms a graph where nodes are the subjects and objects of triples and edges correspond to the predicates. An edge p is created between two nodes (s and o) only if a triple (s,p,o) exists in the collection. More formally, let $\mathcal{U}$ be the set of all URIs, $\mathcal{L}$ the set of all literals and $\mathcal{B}$ an infinite set of blank nodes. A *triple* is any element of $\mathcal{T} = (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{B})$, while an *RDF graph* (or dataset) is any finite subset of $\mathcal{T}$. The RDF schema (RDFS) is an extension of the RDF vocabulary and intents to provide useful semantics to the triples by introducing a vocabulary for asserting user defined schemas within the RDF model. It also specifies standard URIs of specific types (classes, properties) to denote those relationships between URIs. The representation of both schema and instance information is in the form of RDF triples that can be expressed through various containers or formats.

In short, Linked Data use the Web to create links between data from different sources and their structure makes it easy to be accessed through semantic queries [6]. Those data are usually published on the Web in a machine-readable way and their links may be either diverse, e.g. connecting databases of different organizations, or more simple such as heterogeneous systems inside of a single organization.

Instead of using hyperlinks between HTML (Hyper Text Markup Language) documents, Linked Data rely on documents containing data in RDF. More than simply connecting those documents, they use the RDF statements to link arbitrary things in the Web. To enable this Linked Data combine two fundamental Web technologies that are: URIs & HTTP (Hyper Text Transfer Protocol). Entities, which are identified by URIs and use the *'http://'* schema, can be explored by dereferencing the URI over the HTTP protocol. This is how the HTTP protocol can be used to provide a simple yet universal mechanism for retrieving resources that are serialized as a stream of bytes. A set of rules for publishing Linked Data on the Web was proposed in [35]. The goal was that each newly published set of information would be become part of a single global dataset. This involves three basic steps (*i*) assign URIs to entities that are dereferencing over the HTTP protocol (*ii*) set RDF links to other data sources on the Web and (*iii*) provide metadata about the published data.

Semantic Web can be seen as an extension of the current Web, where all information contains a well-defined meaning, the semantics, enabling the development of mechanisms for the exchange and linking of the described data. Even though this idea has been expressed in various ways [5, 29] the common goal is to construct a Web of data that can be processed directly or indirectly by machines.

2.2 Information Retrieval & Semantic Search

Information Retrieval (IR) can be defined as the act of finding material (e.g. documents) with an unstructured nature (such as free text) that is relevant to an information need (e.g. expressed in a keyword query) from within large collections. There is a wide scope that includes many kinds of data and information needs that can be categorized as IR problems and a variety of applications related to search. Since the early days of the field, the primary focus of IR has been on documents containing text. Those documents have usually a structure such as the title and author of a book or the abstract, keywords of a scientific publication. However, most information inside a document is in the form of text hence relatively unstructured. Perhaps the most common IR task is when a user performs a web-search by issuing a keyword query searching the World Wide Web. The answer is typically a ranked list of links to web pages. The scale on which a web search engine operates is in that of billions of documents, stored and organized in a network of million computers. In [12] further categorizations of search in IR are stated such as *Vertical* where search is performed over a restricted domain topic *Enterprise* that concerns searching inside a corporate intranet and *Desktop* search where the information source are files stored on a personal computer.

Inside the IR field there are also cases that support browsing or filtering the documents. Clustering is a task in which a grouping of the documents is performed, based on their contents. Given a set of information needs (e.g. in the form of topics) classification is the task of deciding which class does a set of documents belongs to. Summarization systems can reduce documents to few paragraphs or phrases that describe the overall content. A such example is the text snippet that is presented next to a Web search result. Information extraction is a technique that enables IR systems to identify named entities (such as persons or places) and combine this information for creating structures that describe relationships between the entities, e.g. a list of all the songs recorded by the *The Beatles*. Systems that perform Question-Answering tasks usually integrate and extend multiple IR technologies (search, summarization, information extraction) for providing a specific answer to a question, e.g. see [18] for a recent survey. Works that address the historic progress of the Information Retrieval and its many applications include [40, 12] while a more recent look at the current state can be found in [9].

Semantic Search is a general term that embraces methods for retrieving relevant content from a search engine, by taking into account the semantics and the context of the user query and/or of the indexed content [13]. The approaches are divided into two basic categories, according to the target of retrieval: a) those that try to improve the relevance of classical search engine over documents by incorporating semantic information, and b) those that retrieve semi-structured data (e.g. entities or RDF triples) from a knowledge base, which is also the retrieval target of this work. Structured query languages are the method of choice when the data is inherently structured in Knowledge Bases. Complex information needs can be formulated without ambiguity. Specifically for this task, a complex query language, not suited for ordinary users, has been designed. SPARQL[1] is the standard query language for knowledge bases that are organized in RDF triples. It is based on the standard query language for databases, SQL. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. A complete survey on different Semantic Search paradigms can be found in [2].

2.3 Keyword-based Searching of RDF data

The concept of content search over linked data dates back to the first days of Semantic Web and Open Linked Data. Early works focus includes semantic search engines, often with the support of a web interface for simplicity to the end-user. Despite the early nature, those works apply a fundamental distinction between the potential implementations. As [25] states those approaches may include: (i) form-based systems that allow users to specify queries through sophisticated web forms, (ii) RDF-based querying languages that expect as input structured queries (e.g. SPARQL), (iii) semantic-based search that employs classical IR techniques for answering keyword queries, (iv) question-answering tools which exploit semantic mark-up for answering queries on natural language.
For the third category, our interest, further distinctions can be applied resulting in the following two categories: i) translating keyword queries to structured

[1]`https://www.w3.org/TR/rdf-sparql-query`

(SPARQL) queries, ii) building or extending an IR system that supports keyword search over RDF data.

2.3.1 Translating Keyword Queries to Structured Queries

Multiple works propose methods where the keyword input is translated to a formal structured language query, which is usually SPARQL. The main goal is to hide the complexity and peculiarities of formal querying from the user and providing a ranking of the final results, while at the same time retaining the preciseness of a structured query search.

One of the first works in that field is [25], where each keyword is matched to a semantic entity before translating it to the final formal query. A semantic entity index is build containing classes, properties and instances extracted from back-end data repositories while the search engine matches keywords to entities based on labels and literals indexed in Lucene. A similar approach is described in [36] where the the keyword query is translated into a Description Logic conjunctive query that exploits knowledge available in the knowledge base. Indices are again created using Lucene, extracting all the URIs and labels of entities and during searching, query terms are mapped to knowledge base entities. Then, using property member axioms, they explore connections and retrieve data values from neighboring individual entities up to a given range.

In [37] the user is presented with an extra step of choosing the appropriate query from a generated list of queries constructed based on the keywords. A list of top-k queries is computed by interpreting keywords as elements of structured queries and producing expressive formal queries. Using this approach, algorithms for subgraph exploration and summarization are described for computing and ranking the final top-k subgraphs. The work presented in [20] overviews ranked retrieval approaches of RDF data with keyword-augmented structured queries and provides an informativeness ranking method based on statistical language models for the structured, but schema-less setting of RDF triples and extended SPARQL queries.

Finally, the approach described in [28], distils the inter-entity relationship summary along with the complete schema information from the RDF data graph for composing SPARQL queries. They develop a new search prioritization scheme that combines the degree of a vertex with the distance from the original keyword element for controlling the expansion of the summary graph. In this way they are able to find the top-k subgraphs that are relevant to the conjunction of the entering keywords in a scalable way. A highly relevant effort, described in in [26], constructs inter-entity relationship summary from inter-entity relationships of RDF data graph, exploits the shortest property path and distance indexes, along with an r-neighborhoods index, and uses dynamic programming algorithm for translating keyword queries into SPARQL queries.

2.3.2 Keyword search over existing or specially created IRSs

Several approaches investigate the adaptation of classical IR methods and ranking techniques for processing the keyword queries over the RDF graph. These systems rely on constructing the structures (e.g. inverted index) either from scratch or by employing IR engines, and adapting the notion of a virtual document over the structured data, while the retrieval unit is usually a sub-graph.

Falcon [10, 11] was designed as a search engine that explored the Semantic Web by searching linked objects and exploiting the nature of structured data. In Falcon, each document corresponds to the textual description of an object. This description is obtained by the use of "RDF sentences", that consist of a maximum subset of b-connected RDF triples. Using Lucene, a combination of inverted indices serve both the mapping of keyword terms and classes into objects (documents). The ranking of the documents against a query is based on the cosine similarity scoring of the term-based similarity to the query and a boosting factor that corresponds to document's popularity.

In [33], the authors identify five basic categories of queries for adhoc object retrieval in the web of data according to data from real web logs: entity queries (i.e. find a specific entity), type queries (i.e. find entities of a particular type), attribute queries (i.e. find values of a particular attribute of an entity), relation queries (i.e. how two entities are related), and other keyword queries (i.e. queries that do not fit in the previous categories). The reported experimental results show that almost 40% of the queries are entity queries. The quality of the results of a simple TD-IDF baseline metric over literals are considered adequate. Additionally, in the the entity search track of SemSearch10 workshop[2], a number of related systems [16, 17, 27] were presented and evaluated. Most of those systems are based on variations of the TD-IDF approach, adapted for RDF data. The evaluation was conducted over the Billion Triple Challenge 2009 dataset [23]. L3S [17] uses Lucene for indexing tokens extracted from the URLs of resources.

One of the most referenced works in the field is [19], where the authors propose a retrieval model that returns a list of RDF sub-graphs using statistical language models ranking the results. The notion of a virtual document is that each document corresponds to an RDF triple and fields include sets of extracted keywords from the *s,p,o* parts along with the frequency of each term. An inverted index retrieves for each keyword a list of closely matching triples that are subsequently given as input to a backtracking sub-graph retrieval algorithm. This algorithm calculates a query likelihood estimation, that measures the probability of generating the query from each candidate subgraph.

A hybrid approach between free text and structured queries is presented in [15]. It uses inverted lists that are populated with terms that appear either as a predicate or an object in a triple. The document representation contains information about terms and the triples that these terms appear in, including term position and in which triple part each term was found. Keyword queries that match against those

[2]`http://km.aifb.kit.edu/ws/semsearch10`

terms return all entities from triples for different expressions e.g. return entities with triples containing *john* or *doe* in the object part for the query *john doe*. The above functionality is offered as an Solr[3] extension.

A distributed approach that is based on the Map-Reduce paradigm is illustrated in [14]. For the support of RDF data the system uses a path-based store as index, distributed across the cluster, while matching paths to query compose the final answer. The offered top-K answer has a monotonic behavior, that guarantees that in each step the system generates the optimum answer. A semantic similarity method for measuring scores for objects with the same predicate is described in [1]. The authors also investigate efficiency and scalability issues such as indexing massive datasets and the search depth when retrieving a subgraph. Similarity is calculated from two different aspects, first exploiting the structural relations of the data, such as distance, and then employing WordNet for taking into consideration the semantic strength between elements.

Another work in which the system returns subgraphs based on a keyword query is presented in [31]. Here, authors argue that since query keywords can not always be mapped to a graph element directly, external knowledge (in the form of patterns) can be used to explore relations between the keywords and the dataset components. First, an inverted index is built, where documents represent a graph element type, such as a literal or a resource. Extraction of the relevant fragments is done through a mapping function based on the textual information of the documents and through the use of patterns that measure the semantic distance between graph components. The final answer is a subgraph where each keyword of the query is mapped to a relevant fragment. The proposed method requires the detailed knowledge of the described domain while a TF-based function ranks the returned subgraphs.

Keyword-based search queries with spatial and temporal semantics on RDF data, which the authors call kSP queries, are studied in [41]. The objective of an kSP query is to find RDF subgraphs which contain the query keywords and are rooted at spatial entities close to the query location. Temporal semantics are added by considering the temporal differences between the keyword-matched vertices and the query timestamp and by using a temporal range to filter keyword-matched vertices.

The state of the art approaces in the literature, mainly employ the probabilistic model BM25 and various extensions of it for ranking the RDF data. Such an extension is the BM25F which takes into account the various fields of modeled document, and computes the normalized term-frequency using the length of a field instead of a document (see for e.g. [7] and [32]). Specifically, in [7], along with an adaptation of the BM25F ranking function for RDF data, the authors propose a set of index structures for entity search. Each document represents a resource organized either in a horizontal or in a vertical index. In the former, a single field contains all different properties, along with other fields for storing the extracted

[3]`https://lucene.apache.org/solr`

tokens (e.g. literals), while in the latter each property corresponds to a different field containing its values. Important properties, such as description, abstract, etc., are weighted more. Implementation is based on MG4J [8], by exploiting various options of this open-source IR engine such as the alignment operator that offers queries with term-position matching capabilities.

As already mentioned most works returned a ranking of subgraphs, while in our work we focus on ranking triples. The concept of triple ranking has also emerged in other works such as [21] or [34]. However those approaches are not directly targeting the task of keyword search over an RDF dataset. In [21] authors have presented the 'TripleRank' algorithm for authority ranking in the Semantic Web, much like how PageRank is for the World Wide Web. To achieve this, RDF graphs, along with their semantics, are represented based on 3 dimensional tensors. The algorithm manages to identify authoritative sources and groups of semantically coherent predicates and resources. Along with a set of browsing search strategies 'TripleRank' is targeted for use upon the Semantic Web. More recently, in [34] authors proposed a learning to rank framework with relation-independent features that aims at developing ranking models that measure triple significance. For a given relation type as input (e.g. *profession*) the computed score of each triple measures how well does the triple capture the relevance of the statement that it express, compared to other triples from the same relation.

2.4 The Placement of our Work in the Landscape

We focus on exploiting the functionalities of successful document-centric IR system for providing keyword search over RDF. We support the virtual document approach for dealing with the complexity of graph data using IR ranking techniques while maintaining the scalability that a document-centric system offers. In comparison to similar works, in our work (a) we focus on the retrieval of triples, which is the most flexible and informative retrieval unit (see Sec. 4.1), (b) we study methods for entities ranking on top of triples, since entities are the most studied retrieval unit in the bibliography and available datasets for evaluation do exist, and (c) we thoroughly study and evaluate different indexing and querying approaches, over the widely used Elasticsearch IR system and the commonly used entity search collection DBpedia-Entity as used in [24]. Our results show that we can configure common IR systems to offer comparable performance to adhoc keyword search systems for entity ranking, and take advantage of their out-of-the-box scalability features.

Chapter 3

Problem Analysis and Requirements

3.1 Problem Modeling

We first define the notions of *RDF triple* and *RDF dataset*. Consider an infinite set of URI reference $\mathcal{U}$, an infinite set of blank nodes $\mathcal{B}$ (anonymous resources), and an infinite set of literals $\mathcal{L}$. A *triple* $\langle s, p, o \rangle = (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{B})$ is called an RDF triple, where s is the subject, p the predicate, and o the object of the triple. An RDF dataset (or RDF graph) is a finite set of RDF triples. These triples usually describe information for a set of entities E (subject or object URIs), like persons, locations, organisations, etc.

Fig. 3.1 depicts an example of a small RDF graph describing information about three albums of *The Beatles* band. The dataset contains 16 triples and involves 4 entity URIs (blue nodes), 2 class URIs (red nodes), and 8 literals (gray nodes). Among the 8 literals, 7 are strings (free text) and 1 is a number representing the year 1960. The corresponding set of RDF triples is shown in Fig. 3.2. For a given set of RDF triples T, our objective is to offer a keyword-based search service over T, where a user can submit a free-text query q and get back the most relevant data from T.

3.2 Challenges

A series of challenges arise upon applying keyword search over structured data since the traditional notions of *document* and *index* are not applicable. In order to proceed, one has to to define those challenges.

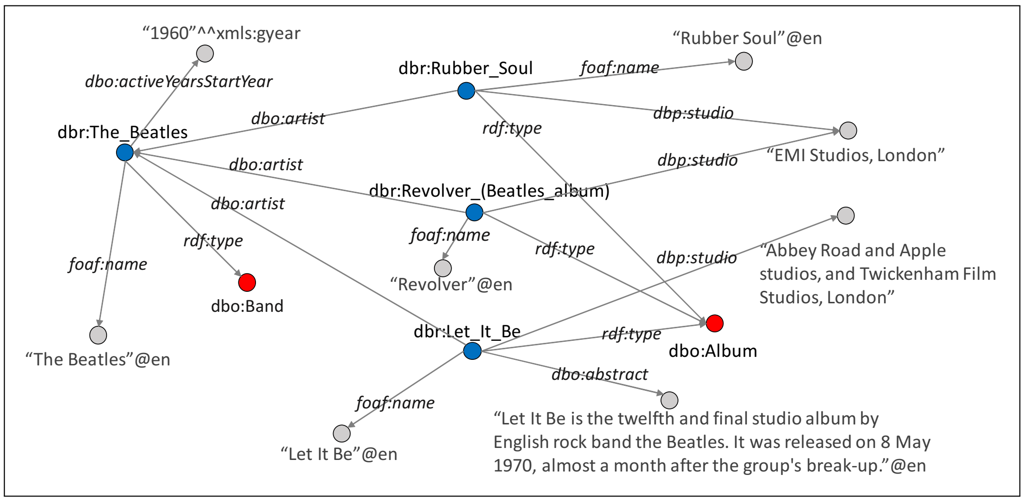

Figure 3.1: An example of a small RDF graph.

3.2.1 Deciding on the retrieval unit

Contrary to the classical IR task where the *retrieval unit* is (usually) an unstructured or semi-structured textual document, an RDF dataset contains highly-structured data in the form of RDF triples, where each triple consists of three elements: *subject*, *predicate* and *object*. There are three main options to consider regarding the *retrieval unit*:

- An *entity* (URI of subject or object): an RDF dataset usually describes information for a set of resources (like persons or locations). In those datasets a resource can be found either in the subject and/or the object part of the triple. This retrieval unit satisfies information needs related to the retrieval of one or more entities (*entity search*), i.e., queries like "The Beatles albums", "Greek philosophers" etc.

- A *triple* (subject-predicate-object): provides more information to the user, than single URIs, and especially satisfies information needs related to *attribute search* where we want to find a characteristic or property of an entity, i.e., queries like "Beatles formation year", "birth date of George Harrison", "capital of Greece", "picture of Barack Obama". Presenting the full triple also provides an easy mean to verify the correctness of a returned result, e.g. `(dbr:Greece, dbo:capital, dbr:Athens)` vs the returning unit of an entity-based approach `(dbr:Athens)`.

- A *subgraph* (of size l triples): describes more complex information than a single triple. Consider, for example, the query "Beatles studios" and the RDF dataset of Fig. 3.1. Answer consists of two literals ("EMI Studios ..", "Abbey Road..") connected to the Beatles' albums (`dbr:Revolver`,

```
[frame=lines,numbers=left,numbersep=1pt]
<http://dbpedia.org/resource/The\_Beatles> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/ontology/Band> .
<http://dbpedia.org/resource/The\_Beatles> <http://xmlns.com/foaf/0.1/name>
"The Beatles"@en .
<http://dbpedia.org/resource/The\_Beatles> <http://dbpedia.org/ontology/activeYearsStartYear>
"1960"^^<http://www.w3.org/2001/XMLSchema#gYear> .
<http://dbpedia.org/resource/Rubber\_Soul> <http://dbpedia.org/property/studio>
"EMI Studios, London"^^<http://www.w3.org/1999/02/22-rdf-syntax-ns#langString> .
<http://dbpedia.org/resource/Rubber\_Soul> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/ontology/Album> .
<http://dbpedia.org/resource/Rubber\_Soul> <http://dbpedia.org/ontology/artist>
<http://dbpedia.org/resource/The\_Beatles> .
<http://dbpedia.org/resource/Rubber\_Soul>  <http://xmlns.com/foaf/0.1/name>
"Rubber Soul"@en .
<http://dbpedia.org/resource/Revolver\_(Beatle\_album)> <http://dbpedia.org/property/studio>
"EMI Studios, London"\^\^<http://www.w3.org/1999/02/22-rdf-syntax-ns#langString> .
<http://dbpedia.org/resource/Revolver\_(Beatles\_album)>  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/ontology/Album> .
<http://dbpedia.org/resource/Revolver\_(Beatles\_album)>  <http://dbpedia.org/ontology/artist>
<http://dbpedia.org/resource/The\_Beatles> .
<http://dbpedia.org/resource/Revolver\_(Beatles\_album)>  <http://xmlns.com/foaf/0.1/name>
"Revolver"@en .
<http://dbpedia.org/resource/Let\_It\_Be> <http://dbpedia.org/property/studio>
"Abbey Road and Apple studios, and Twickenham Film Studios, London"\^\^<http://www.w3.org
/1999/02/22-rdf-syntax-ns#langString> .
<http://dbpedia.org/resource/Let\_It\_Be>  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://dbpedia.org/ontology/Album> .
<http://dbpedia.org/resource/Let\_It\_Be>  <http://dbpedia.org/ontology/artist>
<http://dbpedia.org/resource/The\_Beatles> .
<http://dbpedia.org/resource/Let\_It\_Be> <http://xmlns.com/foaf/0.1/name>
"Let It Be"@en .
<http://dbpedia.org/resource/Let\_It\_Be> <http://dbpedia.org/ontology/abstract>
"Let It Be is the twelfth and final studio album by English rock band the Beatles. It
was released on 8 May 1970, almost a month after the group's break-up.?@en
```

Figure 3.2: The RDF triples of the RDF graph in Fig. 3.1.

`dbr:Rubber_Soul`) through the property `dbp:studio`. The two albums are connected to the Beatles entity `dbr:The_Beatles` through the property `dbo:artist`. Thus, a correct candidate answer consists of more than one triple, more specifically a path(or *subgraph*) of 2 triples: `<dbr:The_Beatles, dbo:artist, dbr:Rubber_Soul> <dbr:Rubber_Soul, dbp:studio, "EMI Studios, London"`.

3.2.2 Selecting which data to index

After deciding on the retrieval unit, one must choose which data to extract, i.e., what information parts are useful and thus a user should be able to query and retrieve. An RDF dataset contains elements of different types, namely: i) resource identifiers (URIs of subjects, predicates and objects), ii) free text (string literals), iii) numbers and boolean values (numerical and boolean literals), iv) dates (date literals), as well as v) elements with no external name (i.e. blank nodes) which are used mainly for connecting other named or unnamed elements. With respect to (i), in many cases the last part of a URI reveals the name of the corresponding entity or resource, and thus it might be useful to index (after some pre-processing, e.g. replacing underscores with space). Other hidden information inside a URI that might still be important to some users, are the domain name of the URI (it usually reveals the knowledge base it belongs to, e.g., DBpedia) as well as the middle part of the URI which can reveal the type of the resource (e.g., class, property, ontology, etc.). For instance, a more experienced user with expertise on Semantic Web technologies might want to retrieve schema-related data, like classes and properties of the underlying ontology. Moreover, some properties of RDF are used mainly for providing human readable and useful information, such as rdfs:label and rdfs: comment and thus may be considered for appropriate for indexing than other.

Deciding on which data to extract depends also on the chosen retrieval unit (*problem 1*). In the first case *entity*, all outgoing properties that provide characteristics and more information about the entity can be added. If the retrieval unit is a *triple*, one can just index all of triple's parts (subject, predicate, object), and/or choose to add additional information, for example some of subject's (or object's) common literal properties((e.g. `rdfs:label`) Finally, if we consider a *subgraph* as our retrieval unit, then the indexed data depend on whether the subgraph has a constant size (independently of the query) or its size is determined dynamically during the retrieval process. For the former, one option is to index all possible subgraphs of size l, however this might highly increase the size of the index. Thus, a more flexible approach is to index single triples and select the l triples that form a subgraph during the retrieval process.

3.2.3 Weighting indexed fields

Deciding on the importance of each indexed field may be another thing to consider. By assigning weights, important fields can be exploited during the retrieval and be beneficial to the final ranking process. One can argue that the name-space part of a URI is of less importance than it's keyword fields. Similarly, if retrieval unit is an entity, one may decide to assign a higher weight to URI's containing certain properties (e.g. label, comment, etc.), or that literals are more important than URIs. By allowing the adjustment of weights of the various fields at query time, we can fine-tune the IRS's query module evaluator at run-time and provide better results for easily identifiable query types (e.g. Q&A queries).

3.2.4 Results structuring

The final challenge is to decide on how to structure the results page and show the results. There various approaches that can be followed. One option is to follow a classical IR approach and show a top-K ranked list of individual results (i.e., a ranked list of entities, triples or subgraphs), along with any related metadata information (e.g. relevance score), through a faceted search UI. Another option is to show a top-K graph (i.e., a small subset of the RDF dataset) which depicts how the individual results are connected to each other (i.e., a top-K graph of entities, triples, or subgraphs).

Problems that are not directly related to the nature of the RDF data, like the ranking algorithm to use or how to index the data, are considered parameters of the considered IR system.

3.3 Requirements

In this work we consider three *functional requirements* that limit the scope of our approach but also widen its applicability:

- *Unrestricted RDF data.* We consider any set of valid RDF triples as a valid RDF dataset, without any prior knowledge of the ontology/schema used to describe the underlying data. Thus, input dataset may or may not contain triples describing the data schema. In addition, it is possible for a dataset to contain non-readable URIs.
- *Unrestricted keyword-based/free-text queries.* We consider as input a free-text query that can describe any type of information need (like retrieving an entity or list of entities, finding an attribute of an entity, etc.). Moreover, we do not consider query operators (like `AND/OR`), wildcards, the ability to search in specific indexed fields, phrasal queries, or any other input specified at query-time. The only input provided by the user is a free-text query.
- *Exploitation of an existing IR system.* We do not aim at building a new IR system from scratch, but instead we want to make use of an existing, widely-used IR system of general purpose, exploit its capabilities and functionalities, and tune it for the case of RDF data. Its default settings and parameters should be used when possible, i.e., any configuration should be made only if this is required by the nature of the RDF data, but without considering any information about the topic or domain of the dataset.

Apart from the above-mentioned functional requirements, we also consider the following *non-functional*:

- *Quality:* system should provide relevant and meaningful results for the majority of the submitted queries.
- *Efficiency:* system's response time on submitted queries should be low (e.g., less than 5 seconds).
- *Scalability:* system should scale well as the size of the data increases without sacrificing however any of the above 2 requirements.

Chapter 4

Approach

4.1 Overview

We first provide an overview of our approach and implementation. We essentially describe how we cope with challenges 1-4 discussed in section 3.2.

With respect to challenge 1 (*deciding on the retrieval unit*), we opt for high flexibility and thus consider *triple* as the retrieval unit. A triple is more informative than an entity, provides a means to verify the correctness of a piece of information (it is more close to question answering), and offers more flexibility on how to structure and present the final results to the users (for coping with *challenge 4*). For example, one can group a set of retrieved triples by their subject (entity) and display them as a single result, or show only the subjects if the information need is known to be entity search, or show graphs of connected entities. Moreover in *RDF*, a triple can be viewed as the simplest representation of a fact. This property is one of the major reasons we chose *triple* as our *virtual document.*

Regarding challenge 2 (*selecting the data to index*), we experiment and evaluate different approaches on what data to consider for each triple-document. Our first (*baseline*) approach, only considers data from the triple itself (i.e., text extracted from the subject, object and predicate). This simple approach, may appear problematic in a dataset where URI's are IDs, thus not descriptive of the underlying resource. Consequently, we also extend the *baseline* approach by exploiting information in the neighborhood of the triple's elements. For example, we consider important outgoing properties (such as *rdfs:label, rdfs:comment* , etc.). We evaluate how the different extension options affect both the index size and the quality of the results. We evaluate how the different extension options affect both the index size and quality of the results.

With respect to challenge 3 (*weighting the indexed fields*), we decide not to apply any predefined weights upon the indexed fields but instead, adjust the weights of the various fields at query evaluation time. In this way, we can fine-tune the IRS's query module evaluator at run-time and provide better results for specific query types (e.g. Q&A queries).

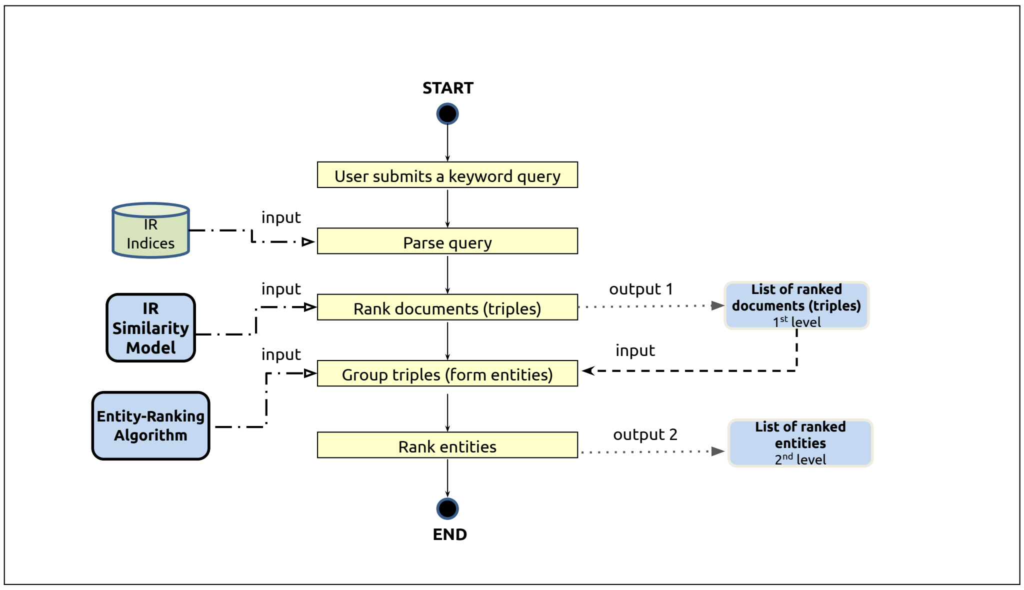

Figure 4.1: Overview of the proposed system's architecture

Finally, for Challenge 4 *(results structuring)*, we opt for a ranked-list of results since this is the way that traditionally IRS present the results to the user. On top of the ranked-list of triples, we propose three different methods for mapping the retrieved ranked list of triples into a ranked list of entities, based on the appearance of resources either in the subject or object. The first one, ranks the groups based on the ranking order of the triple, that the entity appears, i.e. the first group contains triples whose subject/object is the subject/object of the first retrieved triple and so on. The second method ranks the entities based on the number of triples that each entity appears in size (i.e. the first entity contains the entity that appears in the largest number of triples). The third method ranks the entities based on a weighted gain factor of the ranking order of the resources, similar to the discounted cumulative gain used in nDCG. The evaluation of different visualization methods (i.e. ranked list of resources, top-K graphs, etc.) and the corresponding user experience goes beyond the scope of this paper. An overview of system's architecture is depicted in Fig. 4.1.

Below, after an initial discussion of the considered IRS and its offered functionality and parameters, we detail the different approaches we experimented with for dealing with the challenges 2-4. Recall that for Challenge 1 we use a triple as our retrieval unit.

Relational DB	=>	Databases	=>	Tables	=>	Rows	=>	Columns
Elasticsearch	=>	Indices	=>	Types	=>	Docs	=>	Fields

Figure 4.2: Comparison between Elasticsearch and a traditional relational DB, as depicted in [22].

4.2 Considered IR system: Elasticsearch

`Elasticsearch` [1] is a highly scalable open-source full text search engine that allows to store and search big volumes of data. It is built on top of Apache's Lucene[2] and uses, along with the distributed inverted indexes architecture, all of Lucene's powerful capabilities. All data in Elasticsearch are stored in *indices* containing different types of *documents* that Elasticsearch can *store, update* and *search.*

4.2.1 Basic Concepts

Document & Field
In most IR systems, including `Elasticsearch` *document* is the unit of search and index. Each document is a JSON object and is stored in an index, with a unique id. It contains a set of fields each being a key-value pair of any datatype including a primitive (*string*, *numeric*, *boolean*) or a more complex structure (*JSON objects*). Fields can be seen as document's named attributes. Differently than in a common IR system, a document has a *type* associated with it however since Elasticsearch is schema-free, documents with the same type can have different sets of fields. In our implementation we experiment with different types of documents (sets of fields) with the aim to find out which type of a document suits best to a *triple.*

Index
An index consists of one or multiple documents, and can be seen like a classic database. It has a mapping which defines multiple types and can operate as a data organization mechanism, allowing users to partition data in various ways. A rough comparison with a classic relational database can be seen in Fig. 4.2. An important remark is that a single `Elasticsearch` index does not correspond to a single Lucene index. In a working cluster, we can define multiple elastic indexes, but in all of our cases an index corresponds to a particular approach.

Shard
A shard is a Lucene index and usually multiple shards may form an Elasticsearch index. The number of shards that each index has must be defined before the index

[1]https://www.elastic.co
[2]https://lucene.apache.org

is created. It is a fixed value, and when a new documents is inserted to an index, Elasticsearch defines which shard will be responsible for storing and indexing that document. This is done for balancing the load between available shards and affects the overall performance, since all shards can be used simultaneously. This so called *'automatic sharding'* behavior is one of the key parts of the distributed nature of Elasticsearch. As Elasticsearch operates in a network (or cloud) and environment failures are expected, it maintains a rescuing mechanism in case a shard goes offline or fails. This mechanism allows making copies of index's shards, called *replicas*.

Node
A node is a single server that is part of a cluster, stores data and participates in the cluster's indexing and search operations. A cluster usually has multiple nodes where all of the data are spread across. In the same manner as in 'automatic sharding', automatic distribution of shards among the nodes in cluster maintains the distributed nature of `Elasticsearch`.

Cluster
A cluster is a collection of one or more nodes(servers) that holds the entire data and provides federated indexing and search capabilities across all nodes. Even though Elasticsearch can work as a standalone server, it is designed to run on many cooperating servers, each one of them being a single node. Again, Elasticsearch server handles all nodes in the cluster and makes sure that are loaded equally.

To summarize, each Elasticsearch index can be split into multiple shards. Each shard may also be replicated with the use of *replicas*. A node contains multiple shards (and replicas) and if the number of nodes is greater than one, Elasticsearch balances the load equally. A single cluster may contain one or more nodes, that run in parallel and serve multiple requests. As a simple example of a configuration, imagine we have an index that consists of 3 shards(and 3 replicas) but our cluster has only 1 node. In this case, all shards will be on the same node, but as soon as another node is added, Elasticsearch will automatically balance all shards and replicas equally. No matter how many shards are in a node, or the number of nodes, an index is always seen to a client as a single entity. Figure 4.3 depicts this simple example.

4.2.2 Mapping

Mapping is the process of defining how a document, and all the fields it contains, are stored and indexed. Each index has a mapping type which determines how the document will be indexed. Even though it is not necessary for the mapping of each index to be static, it is a common practice to finalize its structure before indexing. In such cases it is equivalent to a schema definition in common SQL databases.

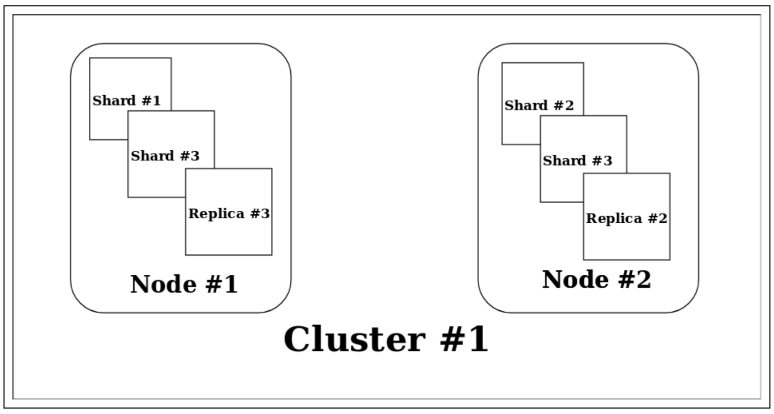

Figure 4.3: Overview of a cluster's components in Elasticsearch

Elasticsearch indexes documents, using either a dynamic (implicit) or a static predefined (explicit) mapping. In the first case, Elasticsearch applies an automatic detection and addition of new fields, without prior knowledge of the type or the analysis of a previously unseen field. This is particular useful in applications where the exact structure of a document is unknown, or in cases where different types of fields are discovered along the indexing process. Moreover in our case, an explicit mapping may define index-wide settings like: *(i)* which fields contain full text *(ii)* which fields contain URIs *(iii)* which custom analyzer (including stemmer, tokenizer) should be applied on which field. It can also allow the disabling of indexing for some fields in a document, an action that results in reducing the amount of the disk space needed while increasing the speed of new document insertion. This is useful in RDF datasets that contain non-readable information (e.g, resource identifiers), since these fields are not indexed but stored and thus can be retrieved. Additionally, during the mapping phase a number of different parameters can be set, including settings that correspond to the cluster configuration of the `Elasticsearch` instance such as the number of *shards* and *replicas*.

In our case, different mappings are created depending on the approach we follow. For example, a snippet of our *baseline* model's mapping is in Fig. 4.4 where values inside *properties* are the explicitly defined fields of our *baseline* index. For each of the document fields, we also specify a *type*, a custom *analyzer* and an additional *copy_to* field. Our type corresponds to *text* while our custom analyzer (*m_analyzer*) specifies which *tokenizer*, *stemmer* and *stopword-list* will be used. Lastly, the *copy_to* parameter copies the values of multiple fields into a group field (*sub_pre_obj*). This allows to query all 3 different fields as a single.

4.2.3 Query DSL & APIs

Elasticsearch has a powerful query DSL (Domain Specific Language) which supports advanced search features while it enables all of Lucene's query syntax to be

```
"mappings": {
    "_doc": {
        "properties": {
            "subject_keywords" : {
                "type": "text",
                "analyzer": "m_analyzer",
                "copy_to": "sub_pre_obj"
            },
            "predicate_keywords" : {
                "type": "text",
                "analyzer": "m_analyzer",
                "copy_to": "sub_pre_obj"
            },
            "object_keywords" : {
            "type": "text",
                "analyzer": "m_analyzer",
                "copy_to": "sub_pre_obj"
            },
            ...
        }
    }
}
```

Figure 4.4: A simplified version of the *baseline* index mapping

easy accessible by using a JSON syntax. Two main type of query clauses exist: (i) *filter-context* which answers whether a query matches a document much like in a boolean-way and (ii) *query-context* which answers how well does a document match a query by providing a relevance score. Since our interest lies on full text searches and the relevance of each result in the search is important, we will mostly be using clauses of type *(ii)*. *Query-context* clauses be further categorized in (i) *leaf-query* type that look for a specific value in a particular field and in (ii) *compound-query* type that wrap other leaf (or compound queries) and are used for building strong combinations of multiple queries, e.g. in a logical fashion.

Term query
Even though we are mostly using full-text queries, a very useful for our case *filter-context* query type is called, *term-query*. As stated by its name, it returns documents that contain an exact term in a provided field. This is commonly used when we are dealing with structured data, e,g, searching for a precise number (date), but in our case it can be used for an exact matching of a resource name. By using a *term-query* we can retrieve all triples (documents) that contain a certain entity appearing either as a subject or as an object or retrieve all triples with a specified property (e.g. *rdfs:label*).

Full text queries
Full text queries enables us to search on any analyzed text field by using the same analyzer that was applied to the field during indexing, a task that is automatically handled by `Elasticsearch`. Members of this family include (i) *match-query*, (ii) *multi-match-query*, (iii) *match-phrase query* and (iv) *query-string query*. The first two types are the standard practice for searching free-text upon specific fields and since they lie on the core of our implementation we discuss them in detail. Queries of type (iii) are designed for searching an exact phrase across a field by taking into consideration the order in which the query-terms appear. A such case in our implementation, where loose-matching is not desirable, is property matching. For example, when searching for *rdfs:label* properties of the *"The_Beatles"* resource, we probably are not interested in loose-match results such as *<dbr:The_Beatles><dbo:recordLabel><dbr:Apple_Records>*. In `Elasticsearch` we can achieve this by employing a *match-phrase* type query. Transforming a phrase match into a proximity search is also possible, by defining a more relaxed search window through the use of the *slop* parameter. Lastly, full text queries of type (iv) further exploit Lucene's powerful query parsing capabilities. A *query-string* type retrieves documents using a syntax that expects boolean operators and wildcards. Depending on the underlying data one can construct really complex search patterns, using boolean operators, wildcards, regular expression along with a number of tuning parameters. Even though we are not using explicitly this type, we regard it as potential useful function for searching over URIs.

Full text queries: *match* & *multi-match*
A *match-query* is executed over a single specified field, which internally is translated into one or multiple Lucene's term queries, that look for each query-term in the field. These are then executed and *match_query* combines their individual scores into the final score of the document. The way these scores are combined for producing the final score can be affected by a number of parameters. A such parameter is *operator*, that if is set to *'and'* will require all query terms to match. Multi-match is an extension type of *match-query* for searching upon multiple fields. Along with all the options that it inherits, further builds on defining 3 additional types depending on the way a multi-match query is executed internally. More specifically it includes:

- *best-fields*: returns documents that match any of the specified fields but it uses the score from the best field for ranking the results. This is useful when a multi-keyword query is best found on the same field.

- *most-fields*: best suited when multiple fields in our index describe the same entity. In this case, we expect same query keywords to be found in more than one field. The score is calculated by adding all individual clauses together and then divided by the number of those match clauses.

- *cross-fields*: is suited when we expect our query keywords to be scattered across multiple fields. This query type attempts to do a search across disjoint fields that we might consider parts of a whole. This is done at query-time by treating different fields as they were one big field by looking for each query-keyword in any field.

Full text queries: *field-centric* & *term-centric* approach
Both type (i) *best-fields* and type (ii) *most-fields* follow a field-centric approach. This essentially means, that they evaluate all query keywords on each field before combining scores from each field. Type (i) assigns as document score the score of the best-matched field while type (ii) calculates final score by adding all field scores together and dividing them by their number. A field-centric approach appears to be problematic in cases where we expect our query terms to be scattered across multiple fields, e.g. in our case where each triple part (s,p,o) corresponds to a specific document field.
A term-centric approach addresses this issue by searching a query term-by-term on each individual field. This is implemented in type (iii) where *cross-fields* searches for each term in any field. The result is that it returns the best overall document. However, `Elasticsearch` index statistics are organized per field and *cross-fields* operates during query-time. Subsequently when fields are combined, the (global-across all fields) document-frequency of each term is unknown. An approximate solution is to consider as term's global DF, the maximum field DF of the term. In that manner, type *cross-fields* modifies each term's document frequency, before

evaluating a document score. This most often results in non-accurate calculations that may affect the overall performance.

Scoring & tie-breaker
Figure 4.5 shows a comparison between the aforementioned approaches for the query *"Beatles albums"* across the fields: *sub, pre, obj.*

The *field-centric* approach consists of 3 separate term queries corresponding to each of our fields: *sub*, *pre*, *obj.* The score of each term query is calculated by summing the score of all distinct clauses. In our example, the score of term query *q1* would be $s_{q1} = s_{sub:Beatles} + s_{sub:Albums}$.
The default behavior of `Elasticsearch` on field-centric approach is *best-fields.* Document's score is the score of the highest-scoring field as evaluated by each distinct term query: $s_{doc} = max(s_{q1}, s_{q2}, s_{q3})$.
In the *term-centric* approach, we have two (blended) term queries corresponding to each query keyword: *Beatles, albums.* By default when using *cross-fields* type, each blended query will use the best score returned by any field, e.g. $s_{q1} = max(s_{sub:Beatles}, s_{pre:Beatles}, s_{obj:Beatles})$ and consequently document's score would be calculated as $s_{doc} = s_{q1} + s_{q2}$.
A very useful tuning option that allows each blended score to be influenced by all fields, instead of the highest-scoring alone, is the *tie-breaker* parameter that accepts values between 0 (default) and 1. This parameter, as stated by its name, tries to break ties that have been formed by documents in which their best fields against a term return the same score. With tie-breaker enabled, and assuming that $s_{sub:Beatles}$ is the highest-scoring field in *q1*, the score of *q1* is calculated as: $s_{q1} = s_{sub:Beatles} + tie\_breaker * (s_{pre:Beatles} + s_{obj:Beatles})$. In the same manner, the score of *q2* is calculated. Next, all distinct blended query term scores are added together for the document score ($s_{doc} = s_{q1} + s_{q2}$).

field-centric

```
Q = "beatles albums"
fields = ["sub","pre","obj"]

q1: (sub:beatles   sub:albums) |
q2: (pre:beatles   pre:albums) |
q3: (obj:beatles   obj:albums)
```

term-centric

```
Q = "beatles albums"
fields = ["sub", "pre", "obj"]

q1:   (sub:beatles  pre:beatles  obj:beatles)|
q2:   (sub:albums   pre:albums  obj:albums)
```

Figure 4.5: Comparison of *field-centric* vs *term-centric* approach using Lucene's syntax of term queries.

A term-centric approach across multiple fields can be expensive (when number of terms is large) and also introduce inaccuracy. Another choice would be to create a static "super-field", containing the concatenated fields (s,p,o) at index time. This is easily done in `Elasticsearch` by defining a *copy_to* parameter via mapping and

then execute our queries over this field using the more simple *match_query*. This approach is also term-centric and contains accurate statistics since all calculations are stored at index time. However if we only create a single 'super-field' we can no longer provide custom weights for distinguishing the important parts (e.g. object over predicate) and in cases where we are dealing with large datasets it may not be viable of maintaining both concepts.
Summarizing, a *multi_match* of *cross-field* offers flexibility in blending fields, lowers the storage requirements (by not having to store a "super-field") and also comes with some valuable tuning options such as *field-boosting* and the *tie-breaker* parameter.Of course, by calculating and altering at query-time the DF of each term-field it becomes expensive and introduces inaccuracy but it is usually a trade-off between the slight inaccuracy at scoring and the storage-overhead of using a "super-field". In our evaluation tests, we experiment with both approaches.

APIs
Another strength of Elasticsearch lies on is it's extensive list of APIs which allow to manage and query the indexed data. They are exposed using JSON over HTTP, fully asynchronous using REST that enables different platform clients to attach. A comprehensive list of APIs that distinguished them includes: *index*, *document*, *search* etc. Client support includes implementantions in a broad spectrum of programming languages such as: *Java*, *Python*, *Ruby*, *Go* etc.

4.3 Indexing

Our indexing approaches rely on the main idea in which each triple is represented by a single Elasticsearch document, thus constructing a *triple-doc* as our virtual document. The vital part in these approaches is the index structure and more specifically which *fields*, representing a distinct triple component, each *document* will contain.

4.3.1 Constructing a baseline index

Our first index (which we call *baseline-index*) only considers components from the triple itself. The included components are both the extracted text and the namespaces of each triple meaning that each document has 3 *keyword-based* fields, containing the analyzed keywords of the extracted text of each triple part (s,p,o), and 3 *uri-based* fields containing the corresponding namespaces. Additionally one "super-field", using the *copy_to* parameter, concatenates all 3 *keyword-based* fields into one. These 7 fields form a document inside our baseline index and are all typed as *text* fields. We include them in our mapping configuration along with a definition of two analyzers. For the set of *keyword-based* fields we include a special tokenizer for extracting keywords at the end of a URI (tokenize '/', '#' etc) while for the *uri-based* fields we use an `Elasticsearch` custom tokenizer called *letter*.

This tokenizer breaks text into terms whenever it encounters a character which is not a letter and therefore does a good job when applied upon the namespace of a URI. Stemming and stop-word removal is again configurable, e.g. `Elasticsearch` provides a list of available languages while it can also be custom defined (e.g. removing 'http' from namespace fields). All of these settings are highly configurable and can be changed based on the needs of our dataset. A such example is when namespaces do not contain useful information and thus we are not interested in performing a keyword-search on *uri-based* fields. In that case, we may only store this information without indexing, by using the *enable* parameter on those fields. Those fields can be still retrieved but will not be part of the inverted index. This has benefits both in storage and in indexing time.

4.3.2 Extending the baseline index

In our second index approach (which we call *extended-index*) we experiment with adding more information for each document-triple. Here, we rely on *baseline-index* for extract new information by exploring useful properties that will be added to each triple-document. We essentially use *baseline* index as a triple-store for performing custom keyword or filter query operations for creating the *extended* index. Although there are many options for enriching a document, we summarize the following:

(i) *domain-dependent properties*: domain or schema specific properties such as *rdfs:label, rdfs:comment, dbpedia:birthPlace* etc.

(ii) *domain-independent properties*: close relevant properties based on a keyword-query.

(iii) *outgoing properties*: include all properties of either subject, object or both.

Option *(i)* is ideal when the underlying ontology is known since these properties can be configured statically. In cases where we are not aware of the exact schema of the described data, option *(ii)*, we may extract useful properties by performing a keyword search to find relevant properties to a specific query. For example searching for *label* in a dataset similar to the one in Fig.3.1, we may retrieve the following useful property: *<dbr:The_Beatles><dbo:recordLabel><dbr:Apple_Records>*. Those properties are then concatenated in a single field, much like the 'super-field' used in *baseline*. Another option that bridges the gap between *(i)* and *(ii)*, is to perform an analysis on the data and extract the most used properties. Those properties can afterwards be used as the fields for the enriched triple-document. Lastly in option *(iii)*, an extreme approach would be including all outgoing properties of each triple element. This means, that we enrich each document-triple by adding a field of all subject's (or/and object's) outgoing properties. This is applicable in small collections, where the number of triples and corresponding properties is not very large. Again, all *outgoing* properties are stored as single fields, distinguishing them between those of subject's and object's.

Table 4.1 shows an example of the two indexing approaches (baseline-index, extended-index) for the triple `(dbr:The_Beatles, dbo:artist, dbr:Happiness-_Is_a_Warm_Gun)`, where the extended index contains the value of the `rdfs:comment` property of each resource.

Table 4.1: Comparison between *baseline-index* and *extended-baseline* indexing the document: `(dbr:The_Beatles, dbo:artist, dbr:Happiness_Is_a_Warm_Gun)`

baseline index	extended index
	sub_keys: "beatles"
sub_keys: "beatles"	**pre_keys**: "artist of"
pre_keys: "artist of"	**obj_keys**: "happiness warm gun"
obj_keys: "happiness warm gun"	**sub_nspace**: "dbpedia resource"
sub_nspace: "dbpedia resource"	**pre_nspace**: "dbpedia ontology"
pre_nspace: "dbpedia ontology"	**obj_nspace**: "dbpedia resource"
obj_nspace: "dbpedia resource"	**rdfs_comment_sub**: "beatles greatest rock band .."
	rdfs_comment_obj: "happiness warm gun song beatles featured .."

4.4 Retrieval

In this section we describe all the different query methods that we experimented with for retrieving and ranking the indexed documents.

4.4.1 Querying and ranking triples

Exploring different query types
Since our indexes contain different sets of fields we can use multiple types of `Elasticsearch` queries. Regardless of the underlying index structure, we always want to perform a full-text search on analyzed text fields. For this reason we study the following query-approaches:

(i) single *match*: upon a single field, corresponding to each triple-element but also includes the 'super-field' *(spo)*.

(ii) *multi-match*: upon multiple fields, using the type of *cross-fields.* e.g. search over subject and object fields *(s), (p).*

(iii) *boolean*: constructing complex queries by combining clauses of *(i)* and *(ii).*

Examples of the above are presented in Fig. 4.6. The first two approaches explore single *match* and *multi-match* queries upon fields *(spo)* and *(s),(p),(o)* respectively, while the third one combines distinct fields (s),(p),(o) in a boolean

Query 1: single *match*

```
"match": {
  "query": "Beatles studio",
  "field": "sub_pre_obj_keys",
}
```

Query 2: *multi_match*

```
"multi_match": {
  "query": "Beatles studio",
  "fields": ["sub_keys",
  "pre_keys", "obj_keys"],
  "type": "cross_fields",
}
```

Query 3: *boolean* combined

```
MUST
  "multi_match": {
    "query": "Beatles studio",
    "fields": ["sub_keys",
      "obj_keys"],
    "type": "cross_fields",
  },
SHOULD
  "match": {
    "query": "Beatles studio",
    "field": "pre_keys",
  }
```

Figure 4.6: `Elasticsearch` query syntax

logical fashion. A number of additional parameters for further tuning the results may yield better results.
Query 1 from Fig. 4.6 consists of a simple *match* query on a single field that includes all extracted keywords of *subject, predicate* and *object.* Each keyword-query corresponds to a Lucene's term query that is executed internally. The scores of these term queries are added for the final document score. *Query 2* executes the same keyword-query on three distinct fields in the type of *cross-fields.* Since, *cross-fields* follow a term-centric approach, the overall score of the combined term queries is calculated as was described in 4.2.3. *Query 3* behaves in a boolean fashion by combining two different query clauses. It first returns only documents based on the *multi_match* query (on subject and object fields – *sub_keys*, *obj_keys*) that is included inside the MUST clause. It then boosts those documents that also contain a match on their *predicate* field, by including a *match_query* (on *pre_keys*) inside of a SHOULD clause. The final score will be a combination between those two clauses, in a *more-is-better* approach by adding query scores of each clause together.

Weighting fields

Another factor for improving relevance at query-time is applying weights on the various fields. In each of the above query approaches we may additionally boost particular fields, increasing their impact upon relevance score. Boosting fields only makes sense upon multi-field queries, specifying the importance of a field over another. For example in Query 2 from Fig. 4.6 we may define *object* keywords to be twice as important than the other two (i.e. subject, predicate) by modifying the *fields* value to the following: ["sub_keys", "pre_keys", "obj_keysˆ2"]. Documents matching a query on *obj_keys* field will benefit. For deciding those fields along with the boosting factor, we perform evaluation tests on each triple element discussed in section 5.

Exploring different similarity modules
In `Elasticsearch` a similarity model defines how matching documents are scored and it can be configured before indexing via the mapping. Default model is Okapi BM25, a tf/idf based similarity ideal for short fields. Contrary to the classic tf/idf however, BM25 has an upper limit in boosting terms with a high tf, meaning that it follows a nonlinear term frequency saturation. Parameter *k1* can control how quickly this saturation will happen based on the increase in term frequency. Its default value is *1.2* and higher values result in slower saturation. In our case, since the text in our fields is generally short *k1* will probably perform better towards lower values. The other tuning option of BM25 is the field-length normalization, that can be controlled with parameter *b* that defaults in *0.75*. Shorter-fields gain more weight than longer fields by increasing *b*, in our case this can be used to boost a short descriptive resource over a long literal inside an object field. Another available similarity module is *DFR*, a probabilistic model based on measuring the divergence from randomness. Parameters include a basic model definition, e.g. the tf-idf of randomness, and a two-level normalization. *DFI* implements the divergence from independence model, a non-parametric term weighting method based on information theory that expects as input the measure of independence(e.g. *standarized* or χ^2). Language models include the *LM-Dirichlet* similarity, a bayesian smoothing that accepts the μ parameter and the *LM-Jelinek Mercer* similarity which can be parameterized with λ.
In `Elasticsearch` configuring similarity modules is the last step when improving relevance. Even though RDF data differ from typical IR corpus and may not follow the same distributions, the performance of a similarity module is also heavily dependent on the dataset. For this reason we only will test different modules in the end, upon the best query-index combination that has resulted from our experiments.

4.4.2 Grouping and final ranking

At this point we have performed a keyword query upon our documents and have retrieved a ranked list of triples (1st-level results). Our goal is now to return a final ranked list of entities (2nd-level results), by grouping those triples based on the same *subject* and *object*, if the latter is also a resource. As an example if our system returns as 1st-level results the following triples: ***t1*** *<s1,p1,o1>*, ***t2*** *<s2,p2,o2>*, ***t3*** *<s1,p3,o3>* and ***t4*** *<s2,p4,o4>* we end up with the following set of 2 entities: ***e1*** *<t1,t3>*, ***e2*** *<t2,t4>*. Our last step now is to rank and produce the final entity-list that will be the final answer of our system. For the ranking those entities we have developed 3 different ranking alternatives:

1. *r1* where the ranking order follows the order of the subjects of the retrieved

triples, meaning that the first entity corresponds to the triples whose subject is the subject of the first retrieved triple.

2. *r2* where ranking order depends on group size, meaning that the first entity is the one with the largest number of triples.

3. *r3* where ranking order is based on a weighted factor of the ranking order of the triples.

We expect *r3* to work better, since the gain that each entity accumulates works in a logarithmic reduction manner, as in the widely used Discounted Cumulative Gain metric. Each entity collects the discounted gain of each triple based on the ranking position that it appeared on the *1st-level results* ranking. The final score of an entity-result e for a keyword-query q is given by the formula:

$$score(e) = sum_{t_i}^{t_n} \frac{2^{(n\_score_i)} - 1}{\log_2(i+1)} \tag{4.1}$$

where t is the list of triples that form the entity e based on the ranking position of the list (*1st-level* results) that was returned by `Elasticsearch` and n_score_i is the normalized score of triple i in that list for query q. Since `Elasticsearch` deliberately scores documents with any number > 0, we normalize each result in list t based on the *max* and *min* `Elasticsearch` score.

Chapter 5

Evaluation

In Section 5.1 we describe the setup and the dataset used for performing evaluation tests. In 5.2.1 we report results related to the retrieval effectiveness. More specifically, we test the retrieval methods discussed in 4.4 applied on the different indexing approaches discussed in 4.3. In 5.2.2 we report results related to efficiency, including index time and size for each of our approaches.

5.1 Evaluation Setup

5.1.1 Datasets

For our experiments we have used 'DBpedia-Entity' test collection [24], based on DBpedia's dump of 2015-10. The collection contains a set of heterogenous entity-based queries along with relevance judgment that was obtained using crowdsourcing. There exist 4 different query categories (described in Table 5.1) and in total, over 49K query-entity pairs are labeled using a three-point scale (0: irrelevant, 1: relevant, and 2: highly relevant). We followed the instructions for selecting the required files and generating the comparative runs. After removing duplicates we end up with a collection of approximately 400 million triples. In addition to this *full-collection*, we also generated a subset of 15 million triples that forms our *mini-collection* by *(a)* extracting all judged entity-based triples ($\approx$ 6m) and *(b)* randomly adding extra 9m triples.

Table 5.1: 'DBpedia Entity Challenge' collection's different query-categories as depicted on https://github.com/iai-group/DBpedia-Entity

category	**description**	**example**	**queries**
SemSearch	Named entity queries	"brooklyn bridge", "08 toyota tundra"	113
INEX-LD	IR-style keyword queries	"jazz music genres"	99
QALD2	Natural language questions	"Who is the mayor of Berlin?"	115
ListSearch	Queries that seek particular list of entities	"Professional sports teams in Philadelphia"	140

As mentioned in 4.3, an analysis on the dataset can be performed for finding the most used properties. This will be useful for deciding which properties should an *extended* index include. Using `Elasticsearch` powerful 'Terms-Aggregation' we can sort each distinct property based on the number of triples that it has been found. Table 5.2 shows the top-10 most used properties for our collection. Properties such as *dbo:wikiPageWikiLink*, that have multiple occurrences per unique resource, can result in non-efficient structures and also introduce noise in our documents. Other properties such as *foaf:name*, *dbo:name* give us mostly information we have already acquired by handling subject's keywords while we consider *rdfs:type* not descriptive enough to be included in each document. A very useful property, is *rdfs:comment* or *rdfs:abstract* that uniquely describes each resource inside the collection. Since *comment* appears as a further summary of the *abstract* property, we choose it as an *extended* field.

Table 5.2: Top-10 used properties inside the 'DBpedia-dump' dataset.

rank	**property**	**counts**
1	dbo:wikiPageWikiLink	167,064,876
2	rdf:type	32,695,025
3	dbo:wikiPageWikiLinkText	23,209,738
4	dcterms:subject	21,625,633
5	rdfs:label	11,983,732
6	dbo:wikiPageRedirects	7,135,896
7	foaf:name	5,514,517
8	dbo:abstract	4,641,890
9	rdfs:comment	4,641,890
10	dbo:name	4,200,676

5.1.2 Environment

We deploy Elasticsearch 6.4 as a single node with max heap size set at 32GB, with 6 physical cores running on Debian 9.6. Additionally, using Python's multiprocessing pool we initiate *12* indexing instances with a bulk-size of *3500* documents each. Those number are assigned empirically defining a configuration that suits our hardware and collection setup. The number of shards is also assigned empirically and it alters between *baseline* and *extended* index. For the *baseline* index we select 2 shards while depending on the *extended* approach we alter between 3 and 4. Source code and detailed description for the setup can be found in our repositories [1] [2].

5.2 Evaluation Results

5.2.1 Effectiveness (Quality of Retrieval)

As regards the effectiveness, in this section we measure the quality of search results. Following the same evaluation method as in [24] we report NDCG results both at @10 and @100. First we experiment on index models over *mini-collection*, then test if our best model upon the *full-collection* and finally compare those results against the ones using 'DBpedia-Entity v2' in [24]. Our plan is to use the triple as our initial building block in order to gain an insight on how to extend further a document. For this reason, our initial experiments (Exp.1-3) are first executed over the *baseline* index, while Exp. 4, 5 & 6 deal with the *extended* indexes. In Exp. 7 we try different `Elasticsearch` similarity module configurations on the best models from Exp.1-6, and finally on Exp. 8 we apply our best model on the *full-collection*.

5.2.1.1 Exp. 1-3 : baseline index

A better understanding of each model's behaviour, before expanding to more complex structures, can be obtained by distinguishing the following field-methods:

1. (s) - consists of one field that corresponds to *subject's* keywords.
2. (p) - same as (1) but for *predicate's* keywords.
3. (o) - same as (1) but for *object's* keywords.
4. (spo) - contains the concatenation of *subject's*, *predicate's* and *object's* keywords as one field.
5. (s)(p)(o) - same as (4) but organized as three distinct fields.

[1]`https://github.com/SemanticAccessAndRetrieval/Elas4RDF-index`
[2]`https://github.com/SemanticAccessAndRetrieval/Elas4RDF-search`

Exp. 1: examining field separation
Table 5.3 contains results from our first experiment based on field separation. It allows us to review how important each triple element is. More specifically, we investigate the importance of each triple-element (*subject* vs *object*) and the use of a 'super-field' (*spo*) vs distinct fields (*s,p,o*). Single field queries (1.1-1.4) are executed through the *match_query* while multi-fields (1.5),(1.6) use the *multi_match* of type *cross-fields* query.

A first remark is the important role that *object* field holds, since method (1.3) performs way better than the other two parts (1.1 & 1.2). The role of an object in a triple is to describe a resource that is defined in the subject, and in our dataset resources are often marked by descriptive literal values. As a result, `Elasticsearch` ranks those triples high before our grouping-method boost the resources that are contained in the triple's subjects.
As expected, better results are obtained when all fields are involved. The use of a 'super-field' (spo) in method (1.4) seems to perform slightly better in average, mostly for the query types of SemSearch & INEX_LD, than the distinct fields in method (1.5). However, the other two types are favored from a structure that distinguishes fields, as results from method (1.5) show. In particular, the information need of ListSearch queries is satisfied almost entirely by the object part as results of method (1.3) show.
In method (1.6) we extend method (1.5) of distinct fields, by inserting the tie-breaker parameter. This parameter, as discussed in Sec. 4.2, attempts to dissolve scoring ties of documents by taking into consideration all individual field scores for each query term. Queries of type SemSearch are mostly favored since the gap between methods (1.4) & (1.5) is now smaller.

Table 5.3: Comparing *baseline* field separation methods over *mini-collection*, reporting ndcg@100 (@10).

#	**field method**	**SemSearch _ES**	**INEX _LD**	**QALD 2**	**List Search**	**avg**
(1.1)	**(s)**	0.48 (0.46)	0.28 (0.26)	0.30 (0.20)	0.30 (0.30)	0.340 (0.270)
(1.2)	**(p)**	0.02 (0.00)	0.04 (0.01)	0.06 (0.03)	0.07 (0.03)	0.04 (0.01)
(1.3)	**(o)**	0.63 (0.50)	0.43 (0.30)	0.42 (0.26)	**0.47** (0.26)	0.485 (0.330)
(1.4)	**(spo)**	**0.70** (**0.61**)	**0.45** (**0.33**)	0.43 (0.30)	0.44 (0.26)	**0.505** (**0.372**)
(1.5)	**(s),(p),(o)**	0.65 (0.55)	0.44 (0.32)	**0.45** (**0.31**)	0.46 **(0.28)**	0.500 (0.358)

(1.6)	**(s),(p),(o)** tie_breaker = 0.1	0.68 (0.58)	0.44 (**0.33**)	**0.45** (**0.31**)	0.45 (0.27)	**0.505** (**0.372**)

Exp. 2: examining field weighting
Multiple-field queries allow specifying custom weights. This allows us to boost specific fields in order to distinguish the most important and further improve our methods. Following Exp.1 observations, we consider the object part twice as important than the other two parts by assigning a weight of 2 on field (o) in method (2.3). Those results are depicted in Table 5.4.

Our object weighting method (2.3) improves the performance of distinct fields (s),(p),(o) from (1.5). It also closes the gap between the 'super-field' method (1.4) since it performs equally or better in all query types except SemSearch. With the use of the *tie-breaker* parameter we are able to improve this, obtaining our best results so far (on average) in method (2.4).

Table 5.4: Comparing *baseline* field weighting methods over *mini-collection*, reporting ndcg@100 (@10).

#	**field method**	**SemSearch _ES**	**INEX _LD**	**QALD 2**	**List Search**	**avg**
(1.4)	***(spo)***	**0.70** (**0.61**)	0.45 (**0.33**)	0.43 (0.30)	0.44 (0.26)	0.505 (0.372)
(1.5)	***(s),(p),(o)***	0.65 (0.54)	0.44 (0.32)	**0.45** (**0.30**)	0.46 (0.27)	0.500 (0.358)
(2.1)	**$(s)^2$,(p),$(o)^2$**	0.64 (0.54)	0.44 (0.32)	0.44 (0.29)	0.46 (0.26)	0.495 (0.355)
(2.2)	**$(s)^2$,(p),(o)**	0.54 (0.50)	0.36 (0.31)	0.36 (0.28)	0.36 (0.23)	0.405 (0.330)
(2.3)	**(s),(p),$(o)^2$**	0.67 (0.55)	0.45 (0.31)	0.44 (0.28)	**0.48** (**0.28**)	0.510 (0.357)
(2.4)	**(s),(p),$(o)^2$** tie_breaker = 0.1	0.69 (0.58)	**0.46** (0.32)	0.44 (0.28)	**0.48** (0.27)	**0.517** (**0.360**)

Exp. 3: combining methods with boolean logic
Here we rely on the previous experiments remarks, by investigating whether we obtain better results if we combine the best methods of Exp. 1-2 using a boolean logic. These methods can be expressed in `Elasticsearch` through the use of the *boolean* query with the clauses MUST/SHOULD, as was shown in Fig. 4.6 (Query 3).

Table 5.5: Comparing combinations of the methods from Exp. 1-2 using `Elasticsearch` boolean clauses. Results over *mini-collection*, reporting ndcg@100(@10)

#	**field method**	**SemSearch _ES**	**INEX _LD**	**QALD 2**	**List Search**	**avg**
(3.1)	SHOULD **(o)** SHOULD **(spo)**	**0.71** (**0.61**)	**0.46** (**0.33**)	0.44 (0.30)	0.45 (0.26)	0.515 (**0.375**)
(3.2)	SHOULD **(o)** SHOULD **(s),(p),(o)**	0.68 (0.56)	0.45 (0.31)	**0.45** **(0.31)**	**0.47** (**0.28**)	0.512 (0.365)
(3.3)	SHOULD **(spo)** SHOULD **(s),(p),(o)**	0.70 (0.60)	0.45 (**0.33**)	0.44 (0.30)	0.45 (0.27)	0.510 **(0.375)**
(3.4)	SHOULD **(spo)** SHOULD **(s),(p),(o)**^{2}	**0.71** (**0.61**)	**0.46** (**0.33**)	**0.45** (**0.31**)	0.46 (0.27)	**0.520** (0.362)

Method (3.1) combines (1.4) & (1.5) from Exp. 1 and manages to perform better than each one separately. We manage to acquire an improvement in the overall performance when we combine the best methods of Exp.1-2: (1.4) & (2.3) in method (3.2).

Table 5.6: Comparing the best methods from *baseline* index. Results over *mini-collection*, reporting ndcg@100 (@10).

#	**field method**	**SemSearch _ES**	**INEX _LD**	**QALD 2**	**List Search**	**avg**
(2.4)	**(s),(p),(o)**^{2} tie_breaker = 0.1	0.69 (0.58)	**0.46** (0.32)	0.44 (0.28)	**0.48** (**0.27**)	0.517 (0.360)
(3.4)	SHOULD **(spo)** SHOULD **(s),(p),(o)**^{2}	**0.71** (**0.61**)	**0.46** (**0.33**)	**0.45** (**0.31**)	0.46 (**0.27**)	**0.520** (**0.362**)

Our best results from the *baseline* index included in methods (2.4) & (3.4) and are depicted together in Fig. 5.6. We have managed to reduce our methods down to an approach that contains a distinct fields method with a weighted object and another that includes this along with a 'super-field' of all triple parts. The latter seems as a very good workaround since it performs equally good across all query types, while the former is an approximate solution that performs slightly worse. However, since the difference between those approaches is tiny and because maintaining the same information both as a 'super-field' (spo) and as distinct fields (s),(p),(o) requires extra storage and indexing time, we decide to omit from our *extending* index the use of a 'super-field'. Results have shown that a single method with the distinct fields (s),(p),(o) contains enough expressiveness and we believe that with proper tuning, on field-weighting and on the *tie-breaker* parameter for each different category type, method (2.4) can perform close or even better than (3.4).

5.2.1.2 Exp. 4-6 : extended index

In Exp.1-3 we've studied the role of distinct fields, assigning custom weights on certain fields (e.g. the object) and how this affects the performance of each query category type. We now want to introduce additional information for each triple-document by extending the *baseline* as discussed in Sec. 4.3.2. For this reason, we will use three different approaches for extending a triple-document discussed in Exp. 4-6.

Exp. 4: extending the baseline model - *Case 1*
In this experiment, we insert *rdfs:comment* as additional information. Since this is an extension approach, all methods containing distinct fields (s),(p),(o) that were referenced in Exp. (1-3) are also included here. Along with those *baseline* field methods we have also the following:

1. (comment_sub) - consists of one field that includes *subject's* rdfs:comment value.
2. (comment_obj) - consists of one field that includes *object's* rdfs:comment value.
3. (comment_sub_obj) - consists of one field that includes the concatenated values of 1&2.

Our results with the *extended* index are shown in Table 5.7. Single field queries (4.1, 4.2) are again executed through the *match_query* in `Elasticsearch` while the multi-fields (4.3-4.6) correspond to *multi_match* of type *cross-fields*.

Table 5.7: Comparing **extended** index methods, with *rdfs:comment* as additional information. Results over *mini-collection*, reporting ndcg@100(@10)

#	field method	SemSearch _ES	INEX _LD	QALD 2	List Search	avg
Additional information (*rdfs:comment*)						
(4.1)	**(comment_sub)**	0.46 (0.44)	0.32 (0.29)	0.27 (0.22)	0.31 (0.27)	0.340 (0.305)
(4.2)	**(comment_obj)**	0.57 (0.46)	0.43 (0.32)	0.37 (0.24)	0.44 (0.28)	0.453 (0.325)
(4.3)	**(comment_sub_obj)**	0.55 (0.49)	0.41 (0.34)	0.35 (0.28)	0.40 (0.32)	0.427 (0.358)
(4.4)	**(comment_sub) (comment_obj)**	0.50 (0.46)	0.42 (0.33)	0.37 (0.28)	0.43 (0.31)	0.430 (0.345)
Combined fields						
(4.5)	**(s),(p),(o), (comment_sub)**	0.56 (0.51)	0.40 (0.35)	0.39 (0.29)	0.41 (0.31)	0.440 (0.365)
(4.6)	**(s),(p),(o), (comment_obj)**	0.68 (0.57)	0.49 (0.36)	0.45 (0.32)	0.50 (0.33)	0.530 (0.395)
(4.7)	**(s),(p),(o), (comment_sub_obj)**	0.59 (0.52)	0.43 (0.36)	0.39 (0.30)	0.43 (0.34)	0.460 (0.380)
(4.8)	**(s),(p),(o), (comment_sub) (comment_obj)**	0.57 (0.51)	0.44 (0.36)	0.42 (0.31)	0.45 (0.33)	0.470 (0.378)
Weighted fields						
(4.9)	**(s),(p),(o^2), (comment_sub)**	0.68 (0.56)	0.52 (0.36)	**0.50** (**0.34**)	0.53 (0.33)	0.557 (0.398)
(4.10)	**(s),(p),(o^2), (comment_obj)**	0.68 (0.55)	0.48 (0.33)	0.46 (0.30)	0.49 (0.29)	0.527 (0.367)
(4.11)	**(s),(p),(o^2) (comment_sub) (comment_obj)**	0.68 (0.56)	0.53 (**0.37**)	**0.50** (**0.34**)	**0.54** (0.34)	0.562 (0.403)
(4.12)	**(s),(p),(o^2) (comment_sub) (comment_obj)** tie_breaker = 0.1	**0.70** (**0.57**)	**0.54** (**0.37**)	**0.50** (**0.34**)	**0.54** (**0.35**)	**0.572** (**0.410**)

Object-related fields are still the most important as we acquire better results on (4.2) vs (4.1). Even though those two methods have a similar behavior as (1.1) & (1.3) from Exp.1, when combined with the use of a multi-field (4.4) we do not acquire any improvement. Using a 'super-field' for both subject's and object's comment values in (4.3) does not also improve results, since it performs worse than (4.2). Similarly in (4.7) combined with the distinct fields (s),(p),(o) it performs worse than other methods. Our best results come when we combine all three triple fields with object's comment field(4.6). This can be seen as a further boost of the object field, which we already know that it performs better. However, this changes in 'Weighted Fields' where we introduce a custom weight for the object field. In method (4.10) subject's comment seems to be more valuable that object's (4.11). This mean that subject's comment inserts new information by introducing new relative entities that weren't present from the object's comment.

Exp. 5: extending the baseline model - *Case 2*
In Exp. 4 we added *rdfs:comment* as additional information for creating an *extended* index. A such property uniquely describes each resource meaning that it results in a single extra value for each new field in the *extended* index. For the next experiments (Exp. 5 & 6), we create an *extended* index with properties that may have multiple occurrences per unique resource and thus insert more than one value for each new field.

A such property is *rdfs:label* that provides a human-readable label of a resource's name. Following our typical approach, we distinguish the field-methods below:

- (label_sub) - consists of one field that includes *subject's rdfs:label* values
- (label_obj) - consists of one field that includes *object's rdfs:label* values

Table 5.8: Comparing **extended** index methods, with *rdfs:label* as additional information. Results over *mini-collection*, reporting ndcg@100(@10)

#	field method	SemSearch _ES	INEX _LD	QALD 2	List Search	avg
Additional information (*rdfs:label*)						
(5.1)	**(label_sub)**	0.50 (0.46)	0.28 (0.25)	0.29 (0.21)	0.30 (0.18)	0.343 (0.275)
(5.2)	**(label_obj)**	0.56 (0.45)	0.37 (0.26)	0.36 (0.22)	0.40 (0.23)	0.422 (0.290)
(5.3)	**(label_sub) (label_obj)**	0.55 (0.50)	0.37 (0.30)	0.36 (0.25)	0.38 (0.24)	0.415 (0.323)
Combined Fields						
(5.4)	**(s),(p),(o), (label_sub)**	0.64 (0.54)	0.44 (**0.32**)	**0.45** (**0.30**)	0.45 (0.26)	0.495 (0.355)
(5.5)	**(s),(p),(o), (label_obj)**	0.65 (0.55)	0.44 (**0.32**)	**0.45** (**0.30**)	0.46 (0.27)	0.500 (**0.360**)
(5.6)	**(s),(p),(o), (label_sub) (label_obj)**	0.64 (0.54)	0.44 (**0.32**)	**0.45** (**0.30**)	0.45 (0.26)	0.495 (0.355)
Weighted Fields						

(5.7)	**(s),(p),(o^2), (label_sub)**	**0.67** (**0.56**)	**0.45** (0.31)	0.44 (0.28)	**0.48** (**0.28**)	**0.510** (0.358)
(5.8)	**(s),(p),(o^2), (label_obj)**	**0.67** (**0.56**)	**0.45** (0.31)	0.44 (0.28)	**0.48** (**0.28**)	**0.510** (0.358)
(5.9)	**(s),(p),(o^2), (label_sub) (label_obj)**	**0.67** (**0.56**)	**0.45** (0.31)	0.44 (0.28)	**0.48** (**0.28**)	**0.510** (0.358)

Labels properties seem to work best for the object part, however when compared to distinct field methods of Exp. 1 (1.1) & (1.3) we observe that we do not acquire any improvement from their use. Subject's label (5.1) performs close to (1.1) while object's label (5.2) performs worst than its corresponding object field (1.3). This is expected since most of the information that label properties insert is already acquired from the extracted keywords of the URIs. In cases where an object is a literal, or an object in the collection does not contain an *rdfs:label*, the new field becomes useless. This can also be confirmed by comparing the methods in (5.4) & (5,6) with Exp. 1 (1.5) baseline method. In this extension model we do not acquire any improvement from the insertion of the *rdfs:label* values.

Exp. 6: extending the baseline model - *Case 3*
For our final case in building an *extended* index, we deal with including all outgoing properties as additional information in each triple. We regard this as an extreme case, that helps us though investigate how important the neighborhood of each triple element is. We distinguish the following field methods:

- (*outgoing-sub*) - contains all subject's outgoing properties values)
- (*outgoing-obj*) - contains all object's outgoing properties values)

Table 5.9: Comparing *extended* index methods, with all *outgoing* properties as additional information. Results over *mini-collection*, reporting ndcg@100(@10)

#	**field method**	**SemSearch _ES**	**INEX _LD**	**QALD 2**	**List Search**	**avg**
Additional information (*outgoing-properties*)						
(6.1)	**(outgoing_sub)**	0.49 (0.47)	0.33 (0.31)	0.32 (0.30)	0.37 (0.32)	0.378 (0.350)
(6.2)	**(outgoing_obj)**	0.62 (0.49)	0.50 (0.38)	0.46 (0.31)	**0.54** (0.36)	0.530 (0.385)
(6.3)	**(outgoing_sub) (outgoing_obj)**	0.50 (0.47)	0.36 (0.33)	0.33 (0.30)	0.38 (0.33)	0.393 (0.357)
Combined fields						

(6.4)	**(s),(p),(o), (outgoing_sub)**	0.49 (0.48)	0.33 (0.32)	0.33 (0.30)	0.37 (0.32)	0.380 (0.355)
(6.5)	**(s),(p),(o), (outgoing_obj)**	0.64 (0.53)	0.51 (0.36)	0.47 (0.32)	**0.54** (**0.37**)	0.540 (0.400)
(6.6)	**(s),(p),(o), (outgoing_sub) (outgoing_obj)**	0.50 (0.47)	0.36 (0.33)	0.34 (0.30)	0.38 (0.33)	0.395 (0.357)
Weighted Fields						
(6.7)	**(s),(p),(o^2), (outgoing_sub)**	0.60 (0.52)	0.44 (0.34)	0.42 (0.32)	0.47 (0.33)	0.482 (0.378)
(6.8)	**(s),(p),(o^2), (outgoing_obj)**	0.66 (0.52)	0.49 (0.34)	0.49 (0.32)	0.52 (0.34)	0.540 (0.380)
(6.9)	**(s),(p),(o^2) (outgoing_sub) (outgoing_obj)**	0.61 (0.52)	0.45 (0.34)	0.43 (0.32)	0.49 (0.33)	0.495 (0.378)
Boolean: Weighted + Combined						
(6.10)	SHOULD **(s),(p),(o^2)** SHOULD **(outgoing_sub)**	**0.72** (**0.60**)	**0.55** (**0.37**)	**0.51** (**0.33**)	**0.54** (0.34)	**0.580** (0.402)
(6.11)	SHOULD **(s),(p),(o^2)** SHOULD **(outgoing_obj)**	0.65 (0.53)	0.47 (0.32)	0.45 (0.29)	0.49 (0.30)	0.515 (0.360)
(6.12)	SHOULD **(s),(p),(o^2)** SHOULD **(outgoing_sub) (outgoing_obj)**	**0.72** (**0.60**)	**0.55** (**0.37**)	0.50 (**0.33**)	**0.54** (0.34)	0.578 (**0.407**)

In Table 5.9 method (6.2) of object's outgoing properties outperforms subject's in (6.1). When we combine this information with triple's elements in method (6.5) we acquire a slight improvement. However, in 'Weighted Fields' when it is combined with the weighted object in a *multi_match* query in method (6.8), we fail to improve our results. This is like due to the nature of query's *term-centric* approach when the combined fields have large differences in length. Combining all fields in a single query drops performance and introduces noise.
Based on the above hypothesis, in 'Boolean combined', we combine the same field methods as in 'Weighted Fields' but we evaluate them using a boolean clause with a SHOULD operator that results in two distinct queries. Indeed, this helps us in extracting new information that is contained in the *outgoing* fields without distorting the (s),(p),(o) fields. This improvement is depicted in methods (6.10) & (6.12).

Those two methods give us the highest performance of all *extended* methods in all of our experiments. At a close second come the methods (4.9) & (4.11) from the *rdfs:comment* extended index. Those methods have a slight difference in performance (around 2%) and given the extreme storage requirements of *extended (outgoing)* we opt for *extended - (comment)* as our final model of *extended* methods.

5.2.1.3 Exp. 7 : similarity modules

In this experiment, we investigate how different tweaks inside the similarity modules of `Elasticsearch` affect the performance of a) *baseline* and b) *extended* indexes performance. We consider only best methods of each index as were explored in previous experiments. This means that we use a *multi-match* of type *cross-fields* (*tie_breaker* = 0.1.) on:

- *Baseline* with field method: **(s),(p),(o^2)**
- *Extended* with field method: **(s),(p),(o^2),(comment_sub), (comment_obj)**

First we gather an overview of each module's performance for our two indexes, using `Elasticsearch` default options. Note however that module's parameters are heavily dependent on the dataset, index structure and the queries, meaning that optimal values do not exist. Comparisons for the *baseline* and the *extended* indexes are in Tables 5.10 & 5.11 respectively.

Table 5.10: Comparative results for *baseline* index best method, between different `Elasticsearch` modules with **default** configurations. Results over *mini-collection*, reporting ndcg@100 (@10).

Baseline							
#	**module**	**config**	**Sem Search _ES**	**INEX _LD**	**QALD 2**	**List Search**	**avg**
(7.1.1)	BM25	k1 = 1.2 b = 0.75	0.68 (0.56)	0.46 (0.33)	**0.44** (**0.28**)	0.49 (0.29)	0.517 (0.365)
(7.1.2)	DFR	model: g after_e: l norm: z	**0.72** (**0.61**)	**0.52** (0.35)	**0.44** (0.27)	0.50 (0.28)	**0.545** (0.378)
(7.1.3)	LM Dirichlet	$\mu = 2000$	0.62 (0.47)	0.39 (0.25)	0.36 (0.20)	0.43 (0.23)	0.450 (0.287)
(7.1.4)	LM Jel.-Mer.	$\lambda = 0.1$	0.70 (0.59)	**0.52** (**0.36**)	**0.44** (**0.28**)	**0.51** (**0.29**)	0.542 (**0.380**)

Table 5.11: Comparative results for *extended* index best method, between different `Elasticsearch` modules with **default** configurations. Results over *mini-collection*, reporting ndcg@100 (@10).

Extended							
#	**module**	**config**	**Sem Search _ES**	**INEX _LD**	**QALD 2**	**List Search**	**avg**
(7.2.1)	BM25	k1 = 1.2 b = 0.75	0.70 (0.57)	0.54 (0.37)	**0.50** (**0.34**)	0.54 (**0.35**)	0.572 (0.410)
(7.2.2)	DFR	model: g after_e: l norm: z	**0.72** (**0.61**)	**0.55** (0.38)	**0.50** (0.33)	0.53 (0.33)	0.575 (0.412)
(7.2.3)	LM Dirichlet	$\mu = 2000$	0.42 (0.38)	0.31 (0.26)	0.29 (0.23)	0.31 (0.23)	0.333 (0.275)
(7.2.4)	LM Jel.-Mer.	$\lambda = 0.1$	0.71 (0.59)	**0.55** (**0.39**)	**0.50** (**0.34**)	**0.55** (**0.35**)	**0.578** (**0.417**)

In methods (7.1.1), (7.2.1) we report results using the Okapi-BM25 module. Since this is `Elasticsearch` default similarity module, the reported numbers are the same numbers that were extracted in the previous experiments. Next we use DFR, a probabilistic model based on measuring the divergence from randomness. To the best of our knowledge, this module has not been used in Entity-Retrieval tasks and thus no default configuration exists. For this reason, we tested a number of different combinations and in methods (7.1.2, 7.2.2) we report the best. Finally, we use two available Language Models: LM-Dirichlet & LM-Jelinek-Mercer. We were unable to achieve high NDCG values using the former while the latter achieves its best results with λ value set to default (0.1).
We notice that for the *extended* model, three of the similarity modules (BM25, DFR, and LM Jelinek-Mercer) have a very similar performance, with LM Jelinek-Mercer slightly outperforming the other two in all query categories apart from SemSearch_ES, the simplest category, for which DFR provides the best results. However, DFR is a complicated module that requires a lot of training time internally and in Tables 5.10 & 5.11 we report the best configuration we encountered. For these reasons, in Exp. 8 we apply our best models using LM Jelinek-Mercer (*best*) and BM25 (*default*).

5.2.1.4 Exp. 8 : applying best models on *full-collection*

We now examine the performance of our approach, which we call `ElaS4RDF`, on the full collection and compare it to a set of previous approaches that focus on entity search in DBpedia. Specifically, we consider the best performing methods for baseline and extended approaches as used in Exp. 7 with both BM25 and LM

Jelinek-Mercer similarity models.
Since the proposed *Elas4RDF* methods do not require training, we compare them with the *unsupervised* methods of [24] (BM25, PRMS, MLM-all, LM, SDM). Note also that all the methods in [24] have been particularly designed for *entity search in DBpedia* and, as described in the dataset github repository[3], a set of more than 25 DBpedia-specific properties was collected for representing an entity and creating the index. Additionally, they do not deal with efficiency issues meaning that it is possible that some of these models are impractical. On the contrary, we provide general methods that consider an existing IRS (using triple as the retrieval unit), that do not require special dataset-specific information for building the indexes, apart from the use of a very common property, like `rdfs:comment` and also, as shown in Sec. 5.2.2, can be considered efficient.

Table 5.12: nDCG@100 (nDCG@10) results on full collection.

Method	**SemSearch _ES**	**INEX _LD**	**QALD2**	**List Search**	**AVG**
Elas4RDF$_{BL}$ BM25	0.67 (0.57)	0.45 (0.34)	0.32 (0.23)	0.37 (0.27)	0.455 (0.352)
Elas4RDF$_{EXT}$ BM25	**0.68** (**0.59**)	**0.48** (**0.38**)	**0.41** (**0.29**)	**0.43** (**0.30**)	**0.500** (**0.390**)
Elas4RDF$_{BL}$ LM Jelinek-Mercer	0.67 (0.56)	0.44 (0.32)	0.37 (0.25)	0.37 (0.25)	0.463 (0.345)
Elas4RDF$_{EXT}$ LM Jelinek-Mercer	**0.68** (**0.59**)	0.46 (0.36)	**0.41** (**0.29**)	0.41 (0.29)	0.490 (0.382)
DBpedia-Entity-v2 BM25	0.41 (0.24)	0.36 (0.27)	0.33 (0.27)	0.33 (0.21)	0.358 (0.255)
DBpedia-Entity-v2 PRMS	0.61 (0.53)	0.43 (0.36)	0.40 (0.32)	0.44 (0.37)	0.469 (0.391)
DBpedia-Entity-v2 MLM-all	0.62 (0.55)	0.45 (0.38)	0.42 (0.32)	0.46 (0.37)	0.485 (0.402)
DBpedia-Entity-v2 LM	0.65 (**0.56**)	0.47 (**0.40**)	**0.43** (**0.34**)	0.47 (0.39)	0.504 (0.418)
DBpedia-Entity-v2 SDM	**0.67** (0.55)	**0.49** (**0.40**)	**0.43** (**0.34**)	**0.49** (**0.40**)	**0.514** (**0.419**)

Table 5.12 shows the results. We see that, on average, our `ElaS4RDF` method achieves the highest performance when using the extended index and the BM25 model. Compared to the DBpedia-Entity-v2 methods, we notice that the performance of our approach is very close to the top-performing SDM method (the difference is 0.014 for nDCG@100 and 0.029 for nDCG@10). This is a rather promising result, given that the DBpedia-Entity-v2 methods are tailored to the

[3]`https://iai-group.github.io/DBpedia-Entity/index_details.html`

DBpedia dataset. The SDM method is slightly better than our method on average because of its very high performance on the ListSearch query type.

*On tuning **Okapi-BM25***
As we already mentioned in Sec. 4.4, BM25 contains a set of parameters that consist of (1) tf saturation: $k1 \geqslant 0$ and (2) field normalization: $0 \leqslant b \leqslant 1$. It has been noted in [24] that the default settings may be unfitting for Entity Retrieval tasks. Similarly, based on our experimental results we've observed that in our case the default values of parameters b & $k1$ are not the optimal. However those values change based on the query category type. Our focus in this work has been an out-of-the-box approach for `Elasticsearch` meaning that we want to introduce a model that uses as much as possible the default configuration. For this reason, and because BM25 seems like a good overall workaround across the different query types, we do not apply any specific tuning.

5.2.2 Efficiency

In this section we report the storage requirements and the query execution time of our best models for *baseline* and *extended* indexes considering the full DBpedia collection (57GB uncompressed).

The number of virtual documents in both cases is 395,569,688. The size of the baseline index is around 36 GB and that of the extended (with *rdfs:comment*) around 145 GB. We see that, as expected, the extended index requires more than 2 times the size of the baseline index while the index time in both cases may be considered relatively small for such a large collection, 2 and 12 hours respectively.

The average query execution time is around 0.5 sec for the baseline method and 1.4 sec for the extended and depends on the query type. We see that extending the index improves performance, however it affects both the space requirements and the query execution time. Results are depicted in Tables 5.13, 5.14.

Table 5.13: Indexing statistics for the final *baseline* and *extended* models.

model	**# docs**	**size (gb)**	**time(mins)**
Elas4RDF$_{BL}$	395,569,688	36.3	128.4
Elas4RDF$_{EXT}$	395,569,688	145.1	751

Table 5.14: Average query execution time (ms) across different models.

model	**SemSearch _ES**	**INEX _LD**	**QALD 2**	**List Search**	**avg**
Elas4RDF$_{BL}$	227	505	725	478	483.7
Elas4RDF$_{EXT}$	684	1479	1992	1502	1414.2

5.3 Executive Summary

We have studied two different approaches concerning the indexing of our data. The *baseline* method is very space-efficient since it does not require any additional information about a triple (but it also requires readable URIs), while the *extended-comment* method enrich the keywords describing the subject and object URIs with their rdfs:comment property (i.e. it does not require readable URIs). The key results from the aforementioned results are: i) all triple components contribute on achieving high performance; ii) object keywords seem to be more important than subject keywords (performance difference between them is *15%*), thus giving higher weight to the object fields can improve performance (e.g. a weight of 2); iii) extending the index with additional (descriptive) information about the triple URIs improves performance (up to 6%); however, including all available information about the URIs (e.g. all outgoing properties) may improve performance but is not a viable solution in terms of efficiency; iv) system's average response time remains efficient between *baseline* and *extended-comment* models, *0.5* & *1.4* seconds respectively. v) the default similarity model of `Elasticsearch` (BM25) achieves high performance - and may achieve further after a basic tuning; vi) efficiency is also preserved in the storage requirements since *baseline* index only needs 37gb & *extended* 145gb for a 57gb text collection vii) the use of `Elasticsearch` for keyword-based search on RDF data can provide high performance, very close to that of task-and dataset-specific systems that were built from scratch. Moreover, our extended model performs really close to the best untrained model that is however designed for running over the DBpedia dataset (the difference is 0.019 for nDCG@100 and 0.029 for nDCG@10). Our agnostic model, adjusts to a previous unseen dataset simply by including the *rdfs:comment* property in each triple document.

Chapter 6

Conclusion & Future Work

The objective of this work was to investigate the use of a classic document-centric IR system, for enabling keyword search over arbitrary RDF datasets. For this study, we decided to use one of the most widely used IR systems, namely `Elasticsearch`. To this end, we specified the requirements and identified the main rising questions and issues, related to the selection of the retrieval unit and the data to index. We selected triple as our retrieval unit due to its expressiveness and informativeness, and developed a mapping of a ranked list of triples to a ranked list of entities. Then we experimented with a large number of implementation approaches, including different indexing structures, query types, field-weighting methods and similarity models offered by `Elasticsearch`. We essentially investigated how (a) the different indexing approaches (*baseline* and *extended* indexes), (b) the different field methods (field separation & weighting) (c) the different query types (d) the different configurations of `Elasticsearch` similarity modules affect (i) the quality of the results (ii) the efficiency of query execution time and (iii) the index size.

We evaluated the performance of the approaches against the *DBpedia-Entity v2* test collection. The results show that `Elasticsearch` can effectively support keyword search over RDF data, performing similarly to systems built from scratch for the task per se, that use entity-oriented and dataset-specific index structures. The difference from the best unsupervised model, specifically designed over the DBpedia collection, is *0.014* for nDCG@100 and *0.029* for nDCG@10. Additionally, we have shown that after a basic tuning of the BM25 parameters, depending on the query type, those numbers can be further improved. Efficiency in storage requirements is also preserved since our *57* gigabytes collection is indexed and stored using *36* and *145* gigabytes for the *baseline* and *extended* model respectively, while the average query execution time is *0.4* and *1.5* seconds. Scalability can be also supported by `Elasticsearch` capabilities meaning that indexes can grow horizontally in a cluster environment. Our approach is flexible and schema-agnostic, in the sense that it can be applied over any RDF dataset that is organized in the form of triples.

An interesting direction for future work is the automatic detection of the query category and the application of a different configuration parameters for each case. Additionally, our model can be applied over a larger dataset in a cluster, for further exploiting the scalable nature of `Elasticsearch`. We also plan to apply our approach in other RDF collections of different nature, like in domain specific RDF repositories.

Bibliography

[1] Minho Bae, Sanggil Kang, and Sangyoon Oh. Semantic similarity method for keyword query system on rdf. *Neurocomputing*, 146:264–275, 2014.

[2] Hannah Bast, Björn Buchhold, Elmar Haussmann, et al. Semantic search on text and knowledge bases. *Foundations and Trends® in Information Retrieval*, 10(2-3):119–271, 2016.

[3] Dave Beckett and Brian McBride. Rdf/xml syntax specification (revised). *W3C recommendation*, 10(2.3), 2004.

[4] Tim Berners-Lee, Roy Fielding, Larry Masinter, et al. Uniform resource identifiers (uri): Generic syntax, 1998.

[5] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[6] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global, 2011.

[7] Roi Blanco, Peter Mika, and Sebastiano Vigna. Effective and efficient entity search in rdf data. pages 83–97, 01 2011.

[8] Paolo Boldi and Sebastiano Vigna. Mg4j at trec 2005. 01 2005.

[9] Stefan Büttcher, Charles LA Clarke, and Gordon V Cormack. *Information retrieval: Implementing and evaluating search engines*. Mit Press, 2016.

[10] Gong Cheng, Weiyi Ge, and Yuzhong Qu. Falcons: searching and browsing entities on the semantic web. In *Proceedings of the 17th international conference on World Wide Web*, pages 1101–1102. ACM, 2008.

[11] Gong Cheng and Yuzhong Qu. Searching linked objects with falcons: Approach, implementation and evaluation. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):49–70, 2009.

[12] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*, volume 520. Addison-Wesley Reading, 2010.

[13] Philippe Cudre-Mauroux. *Semantic Search*, pages 1–6. Springer International Publishing, Cham, 2018.

[14] Roberto De Virgilio and Antonio Maccioni. Distributed keyword search over rdf via mapreduce. In *European Semantic Web Conference*, pages 208–223. Springer, 2014.

[15] Renaud Delbru, Stephane Campinas, and Giovanni Tummarello. Searching web data: An entity retrieval and high-performance indexing model. *Journal of Web Semantics*, 10:33 – 58, 2012. Web-Scale Semantic Information Processing.

[16] Renaud Delbru, Nur Aini Rakhmawati, and Giovanni Tummarello. Sindice at semsearch 2010. In *Proceedings of the 19th International World Wide Web Conference, Raleigh, North Carolina, USA*. Citeseer, 2010.

[17] Gianluca Demartini, Philipp Kärger, George Papadakis, and Peter Fankhauser. L3s research center at the sem-search 2010 evaluation for entity search track. In *Proc. of the 3rd Intl. Semantic Search Workshop*, 2010.

[18] Eleftherios Dimitrakis, Konstantinos Sgontzos, and Yannis Tzitzikas. A survey on question answering systems over linked data and documents. *Journal of Intelligent Information Systems*, pages 1–27, 2019.

[19] Shady Elbassuoni and Roi Blanco. Keyword search over rdf graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 237–242. ACM, 2011.

[20] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, and Gerhard Weikum. Searching rdf graphs with sparql and keywords. *IEEE Data Eng. Bull.*, 33(1):16–24, 2010.

[21] Thomas Franz, Antje Schultz, Sergej Sizov, and Steffen Staab. Triplerank: Ranking semantic web data by tensor decomposition. In *International semantic web conference*, pages 213–228. Springer, 2009.

[22] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[23] Andreas Harth. Billion Triples Challenge data set. Downloaded from http://km.aifb.kit.edu/projects/btc-2009/, 2009.

[24] Faegheh Hasibi, Fedor Nikolaev, Chenyan Xiong, Krisztian Balog, Svein Erik Bratsberg, Alexander Kotov, and Jamie Callan. Dbpedia-entity v2: A test collection for entity search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 1265–1268. ACM, 2017.

[25] Yuangui Lei, Victoria Uren, and Enrico Motta. Semsearch: A search engine for the semantic web. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 238–245. Springer, 2006.

[26] Xiaoqing Lin, Fu Zhang, and Danling Wang. Rdf keyword search using multiple indexes. *Filomat*, 32(5), 2018.

[27] Xitong Liu and Hui Fang. A study of entity search in semantic search workshop. In *Proc. of the 3rd Intl. Semantic Search Workshop*, 2010.

[28] Zongmin Ma, Xiaoqing Lin, Li Yan, and Zhen Zhao. Rdf keyword search by query computation. *Journal of Database Management (JDM)*, 29(4):1–27, 2018.

[29] Catherine C Marshall and Frank M Shipman. Which semantic web? In *Proceedings of the fourteenth ACM conference on Hypertext and hypermedia*, pages 57–66. ACM, 2003.

[30] Michalis Mountantonakis and Yannis Tzitzikas. Large-scale semantic integration of linked data: A survey. *ACM Computing Surveys (CSUR)*, 52(5):103, 2019.

[31] Hanane Ouksili, Zoubida Kedad, Stéphane Lopes, and Sylvaine Nugier. Using patterns for keyword search in rdf graphs. In *EDBT/ICDT Workshops*, 2017.

[32] José R Pérez-Agüera, Javier Arroyo, Jane Greenberg, Joaquin Perez Iglesias, and Victor Fresno. Using bm25f for semantic search. In *Proceedings of the 3rd international semantic search workshop*, page 2. ACM, 2010.

[33] Jeffrey Pound, Peter Mika, and Hugo Zaragoza. Ad-hoc object retrieval in the web of data. In *Proceedings of the 19th international conference on World wide web*, pages 771–780. ACM, 2010.

[34] Mahsa S Shahshahani, Faegheh Hasibi, Hamed Zamani, and Azadeh Shakery. Towards a unified supervised approach for ranking triples of type-like relations. In *European Conference on Information Retrieval*, pages 707–714. Springer, 2018.

[35] Berners-Lee Tim. Linked data–design issues, 2006.

[36] Thanh Tran, Philipp Cimiano, Sebastian Rudolph, and Rudi Studer. Ontology-based interpretation of keywords for semantic search. In *The Semantic Web*, pages 523–536. Springer, 2007.

[37] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 405–416, Washington, DC, USA, 2009. IEEE Computer Society.

[38] Y. Tzitzikas, C. Alloca, C. Bekiari, Y. Marketakis, P. Fafalios, M. Doerr, N. Minadakis, T. Patkos, and L. Candela. Integrating heterogeneous and distributed information about marine species through a top level ontology. In *Proceedings of the 7th Metadata and Semantic Research Conference (MTSR'13)*, Thessaloniki, Greece, November 2013.

[39] Yannis Tzitzikas, Nikos Manolis, and Panagiotis Papadakos. Faceted exploration of rdf/s datasets: a survey. *Journal of Intelligent Information Systems*, 48(2):329–364, 2017.

[40] Olga Vechtomova. Introduction to information retrieval christopher d, 2009.

[41] Dingming Wu, Hao Zhou, Jieming Shi, and Nikos Mamoulis. Top-k relevant semantic place retrieval on spatiotemporal rdf data. *The VLDB Journal*, pages 1–25, 2019.