Pre-print of:
G. Kadilierakis, P. Fafalios, P. Papadakos, Y. Tzitzikas,
Keyword search over RDF using Document-Centric Information Retrieval Systems,
To appear at: Proceedings of the 17th Extended Semantic Web Conference (ESWC'2020),
June 2020, Heraklion, Crete

Keyword Search over RDF using Document-centric Information Retrieval Systems

Giorgos Kadilierakis[1,2], Pavlos Fafalios[2], Panagiotis Papadakos[1,2], and Yannis Tzitzikas[1,2]

[1] Information Systems Laboratory, FORTH-ICS, Heraklion, Greece,
[2] Computer Science Department, University of Crete, Heraklion, Greece
kadilier@csd.uoc.gr, {fafalios, papadako, tzitzik}@ics.forth.gr

Abstract. For ordinary users, the task of accessing knowledge graphs through structured query languages like SPARQL is rather demanding. As a result, various approaches exploit the simpler and widely used keyword-based search paradigm, either by translating keyword queries to structured queries, or by adopting classical information retrieval (IR) techniques. In this paper, we study and adapt `Elasticsearch`, an out-of-the-box document-centric IR system, for supporting keyword search over RDF datasets. Contrary to other works that mainly retrieve entities, we opt for retrieving triples, due to their expressiveness and informativeness. We specify the set of functional requirements and study the emerging questions related to the selection and weighting of the triple data to index, and the structuring and ranking of the retrieved results. Finally, we perform an extensive evaluation of the different factors that affect the IR performance for four different query types. The reported results are promising and offer useful insights on how different `Elasticsearch` configurations affect the retrieval effectiveness and efficiency.

1 Introduction

The Web of Data currently contains thousands of RDF datasets available online that includes cross-domain KBs like DBpedia and Wikidata, domain specific repositories like DrugBank and MarineTLO, as well as Markup data through schema.org (see [17] 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 system is another popular paradigm for interactive query formulation, however even such systems (see [24] for a survey) need a keyword search engine as an 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 out-of-the-box IR systems, like `Elasticsearch`, in different contexts. To this end in this paper we investigate how such existing document-centric Information Retrieval Systems (IRSs), can be used for enabling keyword search over arbitrary RDF datasets, and how they perform compared to dedicated keyword search systems for RDF. Towards this

aim, we study the following relevant questions: (a) how to index an RDF dataset, (b) what data we should rank and how, and (c) how the search results should be presented. In this work, we study and propose various methods for tackling the above questions over the popular IR system `Elasticsearch`, and report extensive evaluation results in terms of their effectiveness and efficiency.

The rest of the paper is organized as follows: §2 describes the related background, requirements and challenges, §3 the related work, and §4 the adaptation approach of `Elasticsearch` for RDF. Finally, §5 discusses the evaluation results, while §6 concludes the paper and identifies issues for further research.

2 Problem Statement and Requirements

§2.1 describes the background and the main objective, §2.2 discusses the requirements, and §2.3 identifies the rising questions and challenges.

2.1 Background and Objective

We first define the notions of *RDF triple* and *RDF dataset*. Consider an infinite set of URI references $\mathcal{U}$, an infinite set of blank nodes $\mathcal{B}$ (anonymous resources), and an infinite set of literals $\mathcal{L}$. A *triple* $\langle s, p, o\rangle \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{B})$ 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, etc. Fig. 1 depicts an example of a small RDF graph describing three albums of *The Beatles* band. It contains 16 triples, involving 4 entity URIs (black nodes), 2 class URIs (white nodes), and 8 literals (gray nodes). Among the 8 literals, 7 are strings (free text) and 1 is a number.

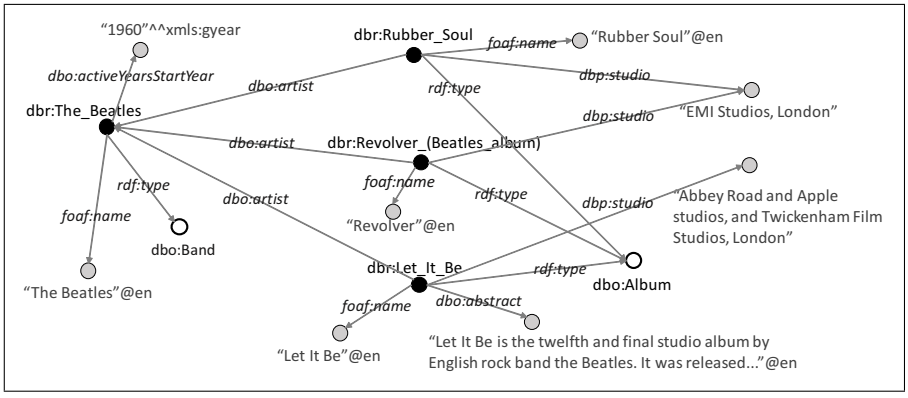

Fig. 1. An example of a small RDF graph.

Our objective is to allow a user submit a free-text query q and get back the most relevant data, for a given set of RDF triples T.

2.2 Requirements

We consider the following three *functional requirements*:

- *Unrestricted RDF datasets.* A valid RDF dataset contains any set of valid RDF triples. We do not presuppose knowledge of the ontology/schema for describing the underlying data. Thus, triples describing the data schema may not exist. In addition, the dataset might not contain human-friendly URIs.
- *Unrestricted keyword-based/free-text queries.* The only input is a free-text query describing any type of information need (e.g. retrieving an entity, attributes of an entity, etc.). 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.
- *Exploitation of an existing IR system.* We do not aim at building a new IR system. Instead we want to use an existing widely-used system, exploit its functionalities, and tune it for retrieving RDF data. Whenever possible, we should use its default settings. 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 indexed dataset.

2.3 Challenges

We can identify four basic challenges of keyword search over RDF data:

Challenge 1: Deciding on the retrieval unit. Contrary to the classic IR task where the *retrieval unit* is 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*:

(i) **An *entity* corresponding to a single URI**. An RDF dataset usually describes information for a set of resources (e.g. persons or locations). Such a resource can be found either in the subject and/or the object of the triple, and satisfies *entity search* information needs, related to the retrieval of one or more entities, like the query "The Beatles albums", etc.

(ii) **A *triple* (subject-predicate-object)**. It provides more information than single URIs, satisfying information needs related to *attribute search*. In such tasks we want to find an attribute of an entity (e.g. "Beatles formation year", etc.), or general information of an entity as captured by string literals. The triple can also help verify the correctness of a result, e.g. `(dbr:The_Beatles, dbo:artist, dbr:Let_It_be)` for the query "Artist of Let It Be", instead of returning the URI of an entity like `dbr:The_Beatles`.

(iii) **A *subgraph* (of size *l* triples)**. It describes more complex information than a single triple. Consider the query "Beatles studios". The answer consists of the two literals ("EMI Studios...", "Abbey Road..."), connected to the Beatles' albums *Revolver* and *Rubber Soul* through the property `dbp:studio`, which in turn are connected to *The Beatles* entity through the property

`dbo:artist`. Thus, a correct candidate answer consists of a path or *subgraph* of two triples, e.g.: `<dbr:The_Beatles, dbo:artist, dbr:Rubber_Soul>` and `<dbr:Rubber_Soul, dbp:studio, "EMI Studios, London">`.

Challenge 2: Selecting the data to index. An RDF dataset contains elements of different types: i) resource identifiers (URIs/URLs), ii) string literals, iii) numerical and boolean literals, iv) date literals, and v) unnamed elements (blank nodes) that are used for connecting other elements. Types ii-iv are all literals, so there is no need for any special preprocessing, while blank nodes (type v) can be ignored. With respect to type i, the last part of a URI usually reveals the name of the corresponding entity or resource, and is rather useful after some pre-processing (e.g. replacing underscores with space). The domain of the URI usually reveals the knowledge base it belongs to, e.g., DBpedia, and its middle part can reveal the type of the resource (e.g., class, property, etc.), which can be useful for more experienced users. If the retrieval unit is an *entity*, one can index (parts of) its URI as well as all its outgoing properties that provide characteristics and more information about the entity. If the retrieval unit is a *triple*, one can just index all of its parts (subject, predicate, object), or choose to index additional data about the subject and object of the triple, e.g. literal properties like the `rdfs:label`. Finally, if we consider a *subgraph* as the retrieval unit, then the data to index depends on whether the subgraph has a constant size, independently of the query, or its size is selected dynamically. For the former, one storage-inefficient option is to index all possible subgraphs of size l. Thus, a more flexible approach is to index single triples and select the l triples that form a subgraph during the retrieval process.

Challenge 3: Weighting the indexed fields. Deciding on the importance of each indexed field may be another thing to consider. By assigning weights, important fields can affect more the final ranking. For example, we may assign higher weights to URI's containing certain properties (e.g. label, comment, etc.), or to literals over URIs. By allowing the adjustment of weights of the various fields at query time, we can fine-tune the IRS's query evaluator module at runtime, offering better results for easily identifiable query types (e.g. Q&A queries).

Challenge 4: Structuring the results. The final challenge is to decide on how to structure and show the results page. One option is to follow a classical IR approach and show a top-K ranked list of individual results (i.e., entities, triples or subgraphs), and its metadata (e.g., relevance score) through a faceted search UI. Another option is to show a top-K graph which depicts how the individual results (entities, triples, or subgraphs), are connected to each other.

We study all these challenges as parameters of `Elasticsearch` (see §4).

3 Related Work

Keyword search over RDF data can be supported either by translating keyword queries to structured (SPARQL) queries (like in [22,23,8,15]), or by building or adapting a dedicated IRS using classical IR methods for indexing and retrieval.

Since our work falls under the second direction, below we report related works and showcase the difference of our approach. Such systems construct the required indexing structures either from scratch or by employing existing IR engines (e.g., Lucene and Solr), and adapt the notion of a virtual document for the structured RDF data. Usually, they rank the results (entities, or subgraphs) according to commonly used IR ranking functions. One of the first such systems was Falcon [2], where each document corresponds to the textual description of the maximum subset of connected RDF triples, while the ranking of the documents is done by mapping keyword terms to documents through cosine similarity and the popularity of each document. In the entity search track of SemSearch10 workshop[3], a number of related systems were presented and evaluated [4,5,16]. Most of those systems are based on variations of the TD-IDF weighting adapted for RDF data, and return a ranked list of entities (i.e., URIs). An approach that uses inverted lists over terms that appear as predicates or objects of triples is described in [3], where the keyword query is translated to a logical expression that returns the ids of the matching entities. Another direction in the bibliography is to return ranked subgraphs instead of relevant entities URIs. For example, in [18] documents represent a literal or a resource, and external knowledge is used to explore relations between the keywords and the dataset components, while the returned subgraphs are ranked using a TF-based function. In [7] the returned subgraphs are computed using statistical language models based on the likelihood estimation of generating the query from each subgraph.

In current state-of-the-art approaches though, RDF data are ranked based on extensions of the BM25 model. For example, BM25F [1,19], takes into account the various fields of a virtual document and computes the normalized term-frequency using the field's length instead of the document's. Further, the recent work described in [6] introduces the TSA+VDP keyword search system, where initially, the system builds offline an index of documents over a set of subgraphs via a breadth-first search method, while at query-time, it returns a ranked list of these documents based on a BM25 model.

Regarding the retrieval unit, we have seen that most works return either URIs or subgraphs. However, the concept of triple ranking has also emerged in works that do not directly target the task of keyword search over RDF data. For example, the TripleRank algorithm presented in [9] ranks authorities in the Semantic Web, in the same manner as PageRank for the WWW. [20] 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 expresses, compared to other triples from the same relation.

With respect to works that make use of `Elasticsearch`, LOTUS [11,12] is a text-based entry point to the Linked Data cloud. It makes use of `Elasticsearch` for supporting keyword search, offering various approaches for matching and ranking the relevant information. Its focus is on scalability and does not study

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

how the different matching and ranking methods affect the retrieval performance. `Elasticsearch` has been also used for indexing and querying Linked Bibliographic Data in JSON-LD format [14], while the *ElasticSearch RDF River Plugin*[4] uses it as a way to index URIs from various endpoints and enrich the indexed documents with RDF data.[5]

Positioning. In our work, we make use of `Elasticsearch` for supporting schema-agnostic keyword search over a set of RDF triples, in order to return a ranked list of triples. We also provide ways of constructing a ranked list of entities over this list of triples. Complementary to the approach followed by LOTUS [11,12], which focuses on the scalability and efficiency of query evaluation using `Elasticsearch`, we study in detail how the various configuration options affect the retrieval accuracy. We aim at gaining a better understanding on how `Elasticsearch` performs over RDF, so that anyone can use it out-of-the-box over any RDF dataset. Our experimental evaluation (Sect. 5) showed that, a proper (schema-agnostic) configuration in `Elasticsearch` provides a retrieval accuracy similar to that of dataset-specific approaches built from scratch for the task per se. To our knowledge, our work is the first that studies how the different indexing and retrieval options in `Elasticsearch` affect the retrieval accuracy.

4 Adapting a Document-centric IRS for RDF

Here, we describe the selected IRS (§4.1) and provide an overview of our approach (§4.2). Then, we detail the various options we experimented with, regarding indexing (§4.3), retrieval (§4.4) and ranking (§. 4.5).

4.1 Considered IRS: `Elasticsearch`

`Elasticsearch` is a highly-scalable, open-source, full text search engine that allows to store and search big volumes of data. It is based on top of Apache Lucene and offers a distributed inverted indexes architecture.

Basic concepts. All data in `Elasticsearch` are stored in *indices* containing different types of *documents* (units of search and index) that `Elasticsearch` can store, update and search. Each document is a JSON object stored with a unique ID that contains a set of *fields*. Each field is a key-value pair of various datatypes (e.g. strings, JSON objects, etc.), organized by a *mapping* type for each index. In our work, we create different mappings depending on the approach we follow. For each field, we need to specify a *type* (e.g., *text*) and an *analyzer*, defining the used *tokenizer*, *stemmer* and *stopword-list*. Each *index* can be split into multiple *shards*, and each shard can be replicated using *replicas*. A *node* contains multiple shards/replicas and if the number of nodes is greater than one, `Elasticsearch` balances the load equally. Finally, a single *cluster* may contain one or more nodes that run in parallel and serve multiple requests.

[4] `https://github.com/eea/eea.elasticsearch.river.rdf#main-features`
[5] `https://www.opensemanticsearch.org/connector/rdf`

Query Domain Specific Language (DSL). Elasticsearch has a powerful Query DSL which supports advanced search features on top of Lucene's query syntax. There are two main types of query clauses: (a) *filter-context* which answers whether a query matches a document (exact-match), and (b) *query-context* which answers how well does a document matches a query using a relevance score (best-match). Since we are interested in free-text search that provides a ranked-list of results, we will solely be using *query-context* clauses. Queries can be further categorised in *match queries* and *multi-match queries*. A *match query* is executed over a single field, while a *multi-match query* allows searching upon multiple fields. Depending on the way it is executed internally, a *multi-match query* is categorized in: (i) *best-fields*, (ii) *most-fields* and (iii) *cross-fields*. Types *(i)* and *(ii)* follow a field-centric approach, evaluating 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 the final score by adding all field scores together and dividing them by their number. Field-centric approaches appear to be problematic in cases where the query terms are scattered across multiple fields (e.g., across the triple's subject, predicate and object). A term-centric approach addresses this issue by searching a query term-by-term on each field. This is implemented in type (iii) where *cross-fields* searches each term across fields, favoring queries whose answer is scattered across multiple fields.

4.2 Overview of the Approach

In this section we describe how we cope with the challenges discussed in §2.3, and provide an overview of our approach and implementation.

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, since it is closer to Q&A, and offers flexibility on how to structure and present the final results (Challenge 4). For example, one can use various aggregation methods over a ranked list of retrieved triples, for providing a ranked list of entities for entity search or showing 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 virtual document. Our *baseline* approach, considers only 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, and 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* and *rdfs:comment*, and evaluate how different extensions affect both the quality of the results and the index size.

With respect to Challenge 3 (*weighting the indexed fields*), we do not apply any predefined weights in the indexed fields, but instead, adjust the weights of

the various fields at query time. In this way, the IRS's query module evaluator can be fine-tuned at run-time for specific query types (e.g. Q&A queries).

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 a method for mapping the retrieved triples into a ranked list of entities, based on the appearance of URIs either in the subject/object. Then, the entities are ranked based on a weighted gain factor of the ranking order of the triples in which they appear, similar to the discounted cumulative gain used in the nDCG metric [13]. The evaluation of different visualization methods (e.g., list of resources, top-K graphs, etc.) and the corresponding user experience goes beyond the scope of this paper.

Below, we provide details of the different approaches we experimented with for indexing, retrieval and ranking of RDF triples in `Elasticsearch`.

4.3 Indexing

We try variations of two different indexing approaches, the *baseline index* that considers only the triple itself, and the *extended index* that extends the *baseline index* with additional descriptive information about the triple components.

Baseline index. This index uses only information that exists in the triple's three components (subject, predicate, object). In case the value of one of the components is a URI, the URI is tokenized into keywords, based on a special tokenizer that extracts the last part of the URI (i.e., the text after the last '/' or '#') that usually describes the underlying resource, and its namespace parts.

Extended index. The *extended index*, includes additional information when one of the triple components is a resource (URI). This is particularly useful when the last part of the URIs are not descriptive of the corresponding resources, and thus not useful for querying. We experiment with three different variations that include the value(s) of: i) the `rdfs:label` property, that usually contains the name of the corresponding resource, ii) the `rdfs:comment` property, which is very descriptive of the underlying resource, and provides much more information than the `rdfs:label`, and iii) all the outgoing properties of the resource. The first two approaches are useful when we are aware of the schema(s) used to describe the data. The latter one, includes all the information that describes the resource. However it can highly increase the size of the index and introduce noise.

4.4 Retrieval

We experimented with various *query types*, *weighting methods* and *similarity models* offered by `Elasticsearch`.

Query types. Since our indexes contain different sets of fields, we can use multiple types of `Elasticsearch` queries. We study the following two approaches: (i) *single-field*: a single field is created containing the index data, e.g., a super-field containing all keywords describing the subject, the predicate and the object, and

(ii) *multi-field*: multiple fields are created, each one containing a specific piece of information, e.g., one field for the subject keywords, one for the predicate keywords and one for the object keywords.

Weighting. Another factor for improving relevance at query retrieval time is applying weights on the various fields. Boosting fields only makes sense upon *multi-field* queries, for specifying the importance of a field over another. For example, we may define that the field containing the object keywords is twice more important than the fields containing the subject and predicate keywords. We experimented with different weighting approaches: i) giving more to weight to the subject keywords; ii) giving more weight to the object keywords; iii) giving more weight to both the subject and the object keywords.

Similarity models (and parameterization). A similarity model defines how matching documents are scored. In Elasticsearch the default model is Okapi BM25, which is a TF/IDF based similarity measure. 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 TF. 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 which has default value *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 in Elasticsearch is *DFR*, a probabilistic model that measures the divergence from randomness. Parameters include a basic randomness model definition, using inverse term frequency, and a two-level normalization. Language models supported by Elasticsearch 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. We experimented with all the above-mentioned similarity models.

4.5 Grouping and Final Ranking

At this point we have performed a keyword query and have retrieved a ranked list of triples (1st-level results). Now, we need to decide on how we will present the results to the user. One approach is to group the retrieved triples based on entities (i.e. subject and object URIs), and return a ranked list of entities (2nd-level results) to the user, where each entity is associated with a ranked list of triples. Such an approach offers flexibility on how to display the results to the user and allows evaluating the different variations we experiment with, using existing ground truth datasets for entity search [10] (more below).

For ranking the derived entities, we exploit the ranking order of the triples based on a weighted factor. Thereby, the gain that each entity accumulates works in a logarithmic reduction manner, as in the widely used Discounted Cumulative Gain (DCG) metric. Specifically, each entity collects the discounted gain of each

Table 1. Query categories in the 'DBpedia Entity' test collection for entity search.

Category	Description	Example	# queries
SemSearch_ES	Named entity queries	"brooklyn bridge"	113
INEX-LD	IR-style keyword queries	"electronic music genres"	99
QALD2	Natural language questions	"Who is the mayor of Berlin?"	115
ListSearch	Entity-list queries	"Professional sports teams in New York"	140

triple based on the ranking position that it appeared on the *1st-level results* ranking. The final score of an entity e for a keyword-query q is given by the formula:

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

where t is the ranked list of triples that the entity e appears in, and n_score_i is the normalized score of triple i in that list for the query q. Since `Elasticsearch` deliberately scores documents with any number > 0, we normalize each result in list t based on the *max* and *min* scores.

5 Evaluation

In §5.1 we describe the setup and the dataset of the evaluation, while §5.2 and §5.3 report retrieval effectiveness and space and time efficiency, respectively. Finally, §5.4 summarizes the key findings.

5.1 Test Collection & Setup

For our experiments we used the DBpedia-Entity test collection for entity search [10], which is based on a DBpedia dump of 2015-10. The collection contains a set of heterogenous keyword queries along with relevance judgments obtained using crowdsourcing. There are four categories of queries: i) named-entity queries, ii) IR-style keyword queries, iii) natural language questions, and iv) entity-list queries. Table 1 provides an example and the total number of queries for each query category. In total, over 49K query-entity pairs are labeled using a three-point scale (0: irrelevant, 1: relevant, and 2: highly relevant).

After following the instructions in [10] for building the RDF dataset and removing duplicates, we end up with a collection of approximately 400M triples. In addition to this *full-collection*, we also generated a subset of 15 million triples that forms our *mini-collection* by extracting all judged entity-based triples (≈6M) and randomly adding an extra of 9M unjudged triples. The mini-collection allows us to run a large number of experiments and study how the different factors discussed in the previous section affect the quality of the retrieved results.

We deployed `Elasticsearch` 6.4 as a single node with max heap size set at 32GB and 6 physical cores running on Debian 9.6. Using Python's multiprocessing pool we initiate 12 indexing instances with a bulk-size of 3,500 documents each. These numbers were assigned empirically based on the collection and our

hardware. The number of shards is also assigned empirically and it alters between the *baseline* and the *extended* index. For the *baseline* we select 2 shards while depending on the *extended* approach we alter between 3 and 4 shards.

The source code of our implementation is available on GitHub (both the indexing service[6] and the search API[7]).

5.2 Quality of Retrieval

Our objective is to measure how the various decisions regarding the indexed triple data, the used Elasticsearch query type, the weighting of the fields, the additional indexed data for each triple, and the similarity models supported by Elasticsearch, affect the quality of search results. We first study the effect of all these factors using the *mini-collection* and then evaluate the best performing methods on the *full-collection*. For measuring the quality, we make use of the evaluation metric nDCG in positions 100 and 10, as in [10].

Examining field separation and query type (baseline-index). We start by examining how each part of the triple (subject, predicate, object) and the different query types (single field, multi-field) affect the quality of the retrieved results using the baseline index, i.e. without considering additional information about the triple subject, predicate or object. Specifically, we examine the following cases: i) *baseline (s)*: only the keywords of the subject are indexed, ii) *baseline (p)*: only the keywords of the predicate are indexed, iii) *baseline (o)*: only the keywords of the object are indexed, iv) *baseline (spo)*: the keywords of all triple's elements are indexed as a single field, v) *baseline (s)(p)(o)*: the keywords of all triple's elements are indexed as different fields. Single-field queries are executed using the *match query* retrieval method while multi-fields using *multi-match query* and *cross-fields*.

Table 2 shows the results. As expected, better results are obtained when all triple elements are indexed. The use of a super-field (spo) seems to perform slightly better in average than using distinct fields, mostly for the query types of SemSearch & INEX-LD. However, the other two types are favored from a structure that distinguishes the fields. This means that the answer of these queries comes from more than one field. Recall that, as described in §4.1, the cross-fields query evaluation type of Elasticsearch favors queries whose answer is scattered across multiple fields. With respect to the distinct triple elements, we see that considering only the object provides the best results, outperforming the case where we consider only the subject, by more than 14%. This means that the answer usually exists in the object part of the triple. It is interesting also that considering only the object provides better results than considering all the triple elements for the ListSearch queries. Finally, as expected, considering only the predicate provides a very poor performance.

Examining field weighting (baseline-index). Multiple-field queries allow specifying custom weights, enabling us to boost the importance of specific fields.

[6] Indexing: https://github.com/SemanticAccessAndRetrieval/Elas4RDF-index
[7] Searching: https://github.com/SemanticAccessAndRetrieval/Elas4RDF-search

Table 2. nDCG@100 (@10) for different field separation and query type approaches.

Method	SemSearch_ES	INEX-LD	QALD2	ListSearch	AVG
baseline (s)	0.48 (0.46)	0.28 (0.26)	0.30 (0.20)	0.30 (**0.30**)	0.340 (0.270)
baseline (p)	0.02 (0.00)	0.04 (0.01)	0.06 (0.03)	0.07 (0.03)	0.04 (0.01)
baseline (o)	0.63 (0.50)	0.43 (0.30)	0.42 (0.26)	**0.47** (0.26)	0.485 (0.330)
baseline (spo)	**0.70** (**0.61**)	**0.45** (**0.33**)	0.43 (0.30)	0.44 (0.26)	**0.505** (**0.372**)
baseline (s)(p)(o)	0.65 (0.55)	0.44 (0.32)	**0.45** (**0.31**)	0.46 (0.28)	0.500 (0.358)

Table 3. nDCG@100 (@10) for different *field weighting* approaches.

Method	SemSearch_ES	INEX-LD	QALD2	ListSearch	AVG
baseline $(s)^2(p)(o)$	0.54 (0.50)	0.36 (0.31)	0.36 (0.28)	0.36 (0.23)	0.405 (0.330)
baseline $(s)(p)(o)^2$	**0.67** (**0.55**)	**0.45** (0.31)	**0.44** (0.28)	**0.48** (**0.28**)	**0.509** (**0.355**)
baseline $(s)^2(p)(o)^2$	0.64 (0.54)	0.44 (**0.32**)	**0.44** (**0.29**)	0.46 (0.26)	0.495 (**0.355**)

Table 4. nDCG@100 (@10) for different approaches to *extend* the index.

Method	SemSearch_ES	INEX-LD	QALD2	ListSearch	AVG
extended-label $(s)(p)(o)^2$	0.67 (**0.56**)	0.45 (0.31)	0.44 (0.28)	0.48 (0.28)	0.510 (0.358)
extended-comment $(s)(p)(o)^2$	**0.68** (**0.56**)	**0.53** (**0.37**)	**0.50** (**0.34**)	**0.54** (**0.34**)	**0.562** (**0.403**)
extended-outgoing $(s)(p)(o)^2$	0.61 (0.52)	0.45 (0.34)	0.43 (0.32)	0.49 (0.33)	0.495 (0.378)

Table 5. nDCG@100 (@10) for different similarity models.

Module	SemSearch_ES	INEX-LD	QALD2	ListSearch	AVG
BM25	0.68 (0.56)	0.53 (0.37)	**0.50** (**0.34**)	0.54 (0.34)	0.562 (0.403)
DFR	**0.72** (**0.61**)	**0.55** (0.38)	**0.50** (0.33)	0.53 (0.33)	0.575 (0.412)
LM Dirichlet	0.42 (0.38)	0.31 (0.26)	0.29 (0.23)	0.31 (0.23)	0.333 (0.275)
LM Jelinek-Mercer	0.71 (0.59)	**0.55** (**0.39**)	**0.50** (**0.34**)	**0.55** (**0.35**)	**0.578** (**0.417**)

We examine the following cases: i) *baseline $(s)^2(p)(o)$*: doubling the weight of subject, ii) *baseline $(s)(p)(o)^2$*: doubling the weight of object, and iii) *baseline $(s)^2(p)(o)^2$*: doubling the weight of both subject and object.

Table 3 shows the results. We see that doubling the weight of the object keywords provides the best results on average, slightly outperforming both *baseline (spo)* and *baseline (s)(p)(o)* @100 (cf. Table 2). On the contrary, we notice that doubling the importance of the subject keywords drops the performance by around 10%. Thus, we can conclude that for this collection, object keywords are more useful for keyword searching than subject keywords.

Extending the index. We now study the case where we extend the index with additional information about the triple's elements. We consider the best performing weighting method, i.e. $(s)(p)(o)^2$, and examine the cases described in

§4.3): i) *extended-label*, that includes the `rdfs:label` property value of the subject and object URIs as two different fields, ii) *extended-comment*, that includes the `rdfs:comment` property value of the subject and object URIs as two different fields, and iii) *extended-outgoing*, that includes the values of all the outgoing properties of the subject and object URIs as two different fields. The object is enriched only if it is a URI. We do not enrich the predicate because the used collection does not include triples that describe the property URIs.

Table 4 shows the results. We see that including the comment property improves performance by more than 5%. On the contrary, including all outgoing properties drops the performance from 0.510 to 0.495, which means that this extension method introduces noise. With respect to the label property, we see that performance is almost the same. This is an expected result given that, in the DBpedia collection, for the majority of resources the last part of the URI is similar to the value of the `rdfs:label` property.

Examining different similarity models. We now study the effect of the different similarity models offered by `Elasticsearch` (on their default setting), as described in §4.4: BM25 ($k1 = 1.2$, $b = 0.75$), DFR (basic model: g, after effect: l, normalization: z), LM Dirichlet ($\mu = 2,000$), and LM Jelinek-Mercer ($\lambda = 0.1$). Since the performance of a similarity model is highly affected by the indexed data, we consider the best performing *extended* method of our previous experiments, i.e., *extended-comment (s)(p)(o)*[2].

Table 5 shows the results. We notice that three of the models (BM25, DFR, and LM Jelinek-Mercer) have a very similar performance, with LM Jelinek-Mercer outperforming the other two in all query categories apart from Sem-Search_ES, the simplest category, for which DFR provides the best results.

Comparative results on the 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: *baseline (s)(p)(o)*[2] and *extended-comment (s)(p)(o)*[2] respectively, 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 [10] (BM25, PRMS, MLM-all, LM, SDM). Note also that all the methods in [10] have been particularly designed for *entity search in DBpedia* and, as described in the dataset's github repository[8], a set of more than 25 DBpedia-specific properties was collected for representing an entity and creating the index. 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`.

Table 6 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 per-

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

Table 6. nDCG@100 (nDCG@10) results on full collection.

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

formance 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 DBpedia dataset and the task per se (entity search). The SDM method is slightly better than our method on average (the difference is 0.014 for nDCG@100 and 0.029 for nDCG@10), mainly because of its high performance on the ListSearch query type.

5.3 Space and Efficiency

We report the space requirements and the average 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 72 GB and that of the extended (with rdfs:comment) around 160 GB. We see that, as expected, the extended index requires more than 2 times the size of the baseline index. The average query execution time is around 0.7 sec for the baseline method and 1.6 sec for the extended and depends on the query type. We see that extending the index improves performance, however it affects the space requirements.

5.4 Executive Summary

The key results from the aforementioned results are: i) all triple components contribute on achieving the highest performance; ii) object keywords seem to be more important than subject keywords, thus giving higher weight to the object fields can improve performance; iii) extending the index with additional (descriptive) information about the triple URIs improves performance; however, including all available information about the URIs (e.g., all outgoing properties) can introduce noise and drop performance; iv) the default similarity

model of Elasticsearch (BM25) achieves a satisfactory performance; v) using Elasticsearch for keyword-based search on RDF data can provide a performance very close to that of task- and dataset-specific systems built from scratch.

6 Conclusion

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 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 if configured properly. Specifically, our Elas4RDF approach, which considers triple as the retrieval unit and provides a higher weight to the object part of the triple, performs similarly to systems built from scratch for the task per se, that use entity-oriented and dataset-specific index structures.

One direction that is worth investigating, is the provision of good answers for *entity-relation* queries, i.e., queries that involve entities that are not directly connected in the indexed RDF graph but they are connected through a long path(s) of triples. In that case, different sets of unconnected triples might be retrieved, each one corresponding to an entity appearing in the query. Thus, in future we plan to study how our approach can be extended for providing answers to such type of queries. Another interesting direction for future work is the automatic detection of the query category and the application of different configuration parameters for each case. Finally, we plan to apply and evaluate our approach in domain-specific RDF datasets, e.g., ClaimsKG [21].

References

1. Blanco, R., Mika, P., Vigna, S.: Effective and efficient entity search in RDF data. In: International Semantic Web Conference. pp. 83–97. Springer (2011)
2. Cheng, G., Qu, Y.: Searching linked objects with Falcons: Approach, implementation and evaluation. International Journal on Semantic Web and Information Systems (IJSWIS) **5**(3), 49–70 (2009)
3. Delbru, R., Campinas, S., Tummarello, G.: Searching web data: An entity retrieval and high-performance indexing model. Journal of Web Semantics **10**, 33 – 58 (2012), web-Scale Semantic Information Processing
4. Delbru, R., Rakhmawati, N.A., Tummarello, G.: Sindice at SemSearch 2010. In: WWW. Citeseer (2010)

5. Demartini, G., Kärger, P., Papadakis, G., Fankhauser, P.: L3S research center at the SemSearch 2010 evaluation for entity search track. In: Proc. of the 3rd Intl. Semantic Search Workshop (2010)
6. Dosso, D., Silvello, G.: A scalable virtual document-based keyword search system for RDF datasets. In: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 965–968 (2019)
7. Elbassuoni, S., Blanco, R.: Keyword search over RDF graphs. In: Proceedings of the 20th ACM international conference on Information and knowledge management. pp. 237–242. ACM (2011)
8. Elbassuoni, S., Ramanath, M., Schenkel, R., Weikum, G.: Searching RDF graphs with SPARQL and keywords. IEEE Data Eng. Bull. **33**(1), 16–24 (2010)
9. Franz, T., Schultz, A., Sizov, S., Staab, S.: Triplerank: Ranking semantic web data by tensor decomposition. In: International semantic web conference. pp. 213–228. Springer (2009)
10. Hasibi, F., Nikolaev, F., Xiong, C., Balog, K., Bratsberg, S.E., Kotov, A., Callan, J.: DBpedia-Entity V2: A test collection for entity search. In: SIGIR. pp. 1265–1268. ACM (2017)
11. Ilievski, F., Beek, W., van Erp, M., Rietveld, L., Schlobach, S.: LOTUS: Adaptive text search for big linked data. In: European Semantic Web Conference. pp. 470–485. Springer (2016)
12. Ilievski, F., Beek, W., Van Erp, M., Rietveld, L., Schlobach, S.: LOTUS: Linked open text unleashed. In: COLD (2015)
13. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of IR techniques. ACM Transactions on Information Systems (TOIS) **20**(4), 422–446 (2002)
14. Johnson, T.: Indexing linked bibliographic data with JSON-LD, BibJSON and Elasticsearch. Code4lib Journal **19**, 1–11 (2013)
15. Lin, X., Zhang, F., Wang, D.: RDF keyword search using multiple indexes. Filomat **32**(5) (2018)
16. Liu, X., Fang, H.: A study of entity search in semantic search workshop. In: Proc. of the 3rd Intl. Semantic Search Workshop (2010)
17. Mountantonakis, M., Tzitzikas, Y.: Large-scale semantic integration of linked data: A survey. ACM Computing Surveys (CSUR) **52**(5), 103 (2019)
18. Ouksili, H., Kedad, Z., Lopes, S., Nugier, S.: Using patterns for keyword search in RDF graphs. In: EDBT/ICDT Workshops (2017)
19. Pérez-Agüera, J.R., Arroyo, J., Greenberg, J., Iglesias, J.P., Fresno, V.: Using BM25F for semantic search. In: Proceedings of the 3rd international semantic search workshop. p. 2. ACM (2010)
20. Shahshahani, M.S., Hasibi, F., Zamani, H., Shakery, A.: Towards a unified supervised approach for ranking triples of type-like relations. In: European Conference on Information Retrieval. pp. 707–714. Springer (2018)
21. Tchechmedjiev, A., Fafalios, P., Boland, K., Gasquet, M., Zloch, M., Zapilko, B., Dietze, S., Todorov, K.: ClaimsKG: A knowledge graph of fact-checked claims. In: International Semantic Web Conference. pp. 309–324. Springer (2019)
22. Tran, T., Cimiano, P., Rudolph, S., Studer, R.: Ontology-based interpretation of keywords for semantic search. In: The Semantic Web, pp. 523–536. Springer (2007)
23. Tran, T., Wang, H., Rudolph, S., Cimiano, P.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In: 2009 IEEE International Conference on Data Engineering. pp. 405–416. ICDE '09, IEEE (2009)
24. Tzitzikas, Y., Manolis, N., Papadakos, P.: Faceted exploration of RDF/S datasets: a survey. Journal of Intelligent Information Systems **48**(2), 329–364 (2017)