SCARO: Scalable RDF Compression using Rule Subsumption Hierarchies

Klaus Lyko
Department of Computer Science
University of Leipzig
Augustusplatz 10, 04109
Leipzig, Germany
lyko@informatik.uni-leipzig.de

Marcus Nitzschke
Department of Computer Science
University of Leipzig
Augustusplatz 10, 04109
Leipzig, Germany
nitzschke@informatik.uni-leipzig.de

Axel-Cyrille Ngonga Ngomo
Department of Computer Science
University of Leipzig
Augustusplatz 10, 04109
Leipzig, Germany
ngonga@informatik.uni-leipzig.de

ABSTRACT

Over the last years, the Linked Data Cloud has been populated with ever larger data sets. The efficient storage of such large RDF data sets is central to ensure an effective management and archiving of Linked Data. While classical compression algorithms can be used on RDF data, novel approaches have shown that logical compression can further reduce the size of compressed RDF data sets. In this work, we present a logical compression approach based on rule subsumption hierarchies. Our approach begins by computing a hierarchy of sets of nodes. This hierarchy is used to detect compression rules that allow reducing the amount of information to compress logically. The compression rules and the graph hierarchy are finally compressed using the standard bzip2 compression library. We evaluate our approach against the state of the art on datasets of different sizes. Through our evaluation, we show that we achieve higher compression rates than related approaches. Moreover, we study the runtime of the approach both with respect to compression and decompression. Here, our results suggest that the runtime of our compression algorithm scales linearly with the size of the input data. Moreover, our compression ratio improves with the size of the input dataset. Finally, our compression approach is more time-efficient then the state of the art in logical compression.

General Terms

Keywords

Semantic web, Data compression, Data Management

1. INTRODUCTION

The Web of data has grown from less than a dozen of knowledge bases to more than 1000 knowledge bases[1] over less than a decade. While the number of knowledge bases increases steadily, the size of the knowledge bases published grows at a remarkable velocity as well. For example, recent additions to the Linked Data Cloud such as LinkedTCGA[2] resp. LinkedGeoData[3] contain over 20 resp. 30 billion facts. Devising approaches for managing RDF datasets of such size is crucial for manifold applications, including all applications that rely on data versioning, archiving, storage and transfer.

Several serialization formats have been devised to store RDF data, of which some already provide means for compressing the information displayed in the RDF documents:

- N-Triples[4] is a plain and native serialization format for RDF. Each RDF dataset can be regarded as a directed graph with labeled nodes and labeled edges. N-Triples represents such graphs as a set of triples `(s,p,o)`, where `s` and `o` are nodes while `p` is a property and stands for the labelled edge between `s` and `o`. Each triple is simply written by concatenating of s, p and o separated by a blank symbol and finalizing the triple with a point. Typically, every line of the serialization holds a single triple.
- RDF/XML[5] is a XML representation of RDF data. The main advantage of this serialization is that it can be easily parsed and edited with traditional XML parsers. The compression of redundant information is ensured by the use of prefixes for namespaces.
- N3[6] and Turtle[7] offer a more complex but still human-readable representation of RDF. Each line of N3 and Turtle can represent several triples, leading to a higher compression then with RDF/XML. For example, two triples `(s, p, o1)` and `(s, p, o2)` that share the same

[1] `http://stats.lod2.eu/`
[2] `http://aksw.org/Projects/LinkedTCGA`
[3] `http://linkedgeodata.org`
[4] `http://www.w3.org/2001/sw/RDFCore/ntriples/`
[5] `http://www.w3.org/TR/REC-rdf-syntax/`
[6] `http://www.w3.org/TeamSubmission/n3/`
[7] `http://www.w3.org/TeamSubmission/turtle/`

subject `s` and predicate `p` can be represented in one line by simply writing `s p o1; o2 .`. Moreover, resource and datatype names can be shortened by using prefixes (like in RDF/XML).

While some RDF serializations offer means to represent RDF data in a more compressed manner, they still demand a considerable amount of storage space. For example, the Turtle representation of the DBpedia datatypes (i.e., the list of RDF triples from DBpedia 2014 with `rdf:type` as predicate) requires more than 500MB hard disk space for storage. Obviously, classical data compression approaches such as bzip2 can be used to further reduce the size of the files at hand. However, such approaches work at character level and fail to make use of the semantics hidden in the RDF data. However, efficient and effective data compression techniques that make use of these semantics for compression have the potential of achieving better compression rates [10] and thus of easing tasks such as the archiving, transfer and storage of RDF data by reducing the amount of space necessary to store large datasets.

In this paper, we present an approach to compressing RDF data that makes use of the semantics of RDF to achieve better compression ratios that character-based approaches. Our approach is inspired by forward chaining and relies on the opposed principle: Instead of concatenating triples that share common subjects and object-type properties, we create hierarchies of rules which can generate such triples when given the subjects and properties. The facts that cannot be compressed by using such rules are stored in an *add graph*. The facts with datatype properties are stored in a *value graph*. Exceptions to the rules are stored in a set of *delete subjects* to the rules. By efficiently compressing (1) the add, value and delete graphs as well as (2) the rules and the mappings between these rules and the subjects for which they hold, our approach can generate small files that can be subsequently compressed using classical character-based approaches. The resulting files are smaller than when using solely previous compression approaches (see Section 4).

Our approach, dubbed SCARO, has several desirable features:

1. SCARO is *lossless* (in contrast to some image, video and sound compression approaches such as JPEG, MP4 and MP3) in the sense that the result of the decompression of a compressed file contains exactly the same set of triples that was contained in the input data. As we will show later, our approach can yet be used in a *lossy mode* to achieve even higher compression rates.

2. Our approach is *efficient* and *scalable*, as its runtime grows linearly with the size of the input data.

3. Moreover, we are *effective* as we achieve better compression ratios than state-of-the-art RDF compression algorithms.

4. Finally, our approach can be used in an *incremental manner*, meaning that once a set K of triples has been compressed, a supplementary set of triples Δ can be added to resp. removed from the compressed file without having to rerun the compression on the whole of $K \cup \Delta$ resp. $K \backslash \Delta$.

The rest of this paper is organized as follows: After presenting the preliminaries necessary to understand this paper, we present our approach on compression for RDF data and illustrate it with a running example. The correspoding decompression is presented in the subsequent section. We subsequently evaluate our approach on seven datasets with respect to runtime and compression rate. After an introduction and a comparison with the state of the art, we conclude the paper by presenting some future work.

The implementation of our approach is open-source and can be found at `https://github.com/KLyko/RDFCompressor` and includes a graphical user interface as well as a command-line interface for compressing and decompressing RDF data.

2. RULES AND RULE HIERARCHIES

In the following section, we present the basic concepts upon which SCARO relies, i.e., the concepts of rules and rule hierarchies.

2.1 Rules

Let K be an RDF knowledge base that is to be compressed. K can be regarded as a set of triples $(s, p, o) \in (\mathcal{R} \cup \mathcal{B}) \times \mathcal{P} \times (\mathcal{R} \cup \mathcal{L} \cup \mathcal{B})$, where $\mathcal{R}$ is the set of all resources, $\mathcal{B}$ is the set of all blank nodes, $\mathcal{P}$ the set of all predicates and $\mathcal{L}$ the set of all literals. The basic idea of modular compression is to exploit patterns in those triples. For large and comprehensive knowledge bases there will be a number of different subjects who share the same property-object relation. We use the term *profile* to designate property-resource pairs (p_i, o_j) (denoted $p_{(i,j)}$) that are such that $\exists s \in \mathcal{R} : (s, p, o) \in K$. We denote the set of all profiles with P:

$$P = \{(p_i, o_j) | (s, p_i, o_j) \in K\}. \quad (1)$$

Each profile $p_{(i,j)} \in P$ is associated with a *rule* $r_{(i,j)}$. Rules are used to derive sets of subjects for which certain information does not need to be stored. To this end, each rule is associated with the set $S(r_{(i,j)})$ of all subjects s which participate in the profile $p_{(i,j)}$. We call $S(r_{(i,j)})$ the *add set* of $r_{(i,j)}$. Moreover, each rule is assigned a set $D(r_{(i,j)})$ of subjects which do not participate in $p_{(i,j)}$. We dub the set $D(r_{(i,j)})$ the *delete set* of a $(r_{(i,j)})$. Formally, the following holds:

$$s \in D(r_{(i,j)}) \rightarrow (s, p_i, o_j) \notin K. \quad (2)$$

We will often use the following notation for rules: $r = (S(r), D(r))$.

Given a rule r, we can generate triples by first generating the triples that result from the add set and then deleting triples by using the delete set. Note that this order is necessary to ensure that the deletion is always effective.

It is now easy to prove that any set of triples can be described by a set of such rules. This is important as it means that for any given finite RDF dataset K, we can find a set of rules that allow reconstructing K exactly.

Example 1 Running example RDF data about Persons and their occupation containing 11 statements. Note, the property `a` is a standard abbreviation for `rdf:type` in N3 and Turtle serialization. For sake of compactness we omit any base and prefix definition.

```
:john a :ComputerScientist .
:doe a :ComputerScientist .

:john a :Person .
:doe a :Person .
:mary a :Person .
:kate a :Person .

:john a foaf:Agent .
:doe a foaf:Agent .
:kate a foaf:Agent .
:christin a foaf:Agent .

:christin  foaf:age "41"^^xs:int .
```

For the sake of illustration, consider the sample RDF data about Persons and their occupation in Example 1, which is also presented as a graph in figure 1. Creating rules for the example would result in the following rules:

1. $r_{(a,:ComputerScientist)} = (\{:john, :doe\}, \emptyset)$
2. $r_{(a,:Person)} = (\{:john, :doe, :mary, :kate\}, \emptyset)$
3. $r_{(a,foaf:Agent)} = (\{:john, :doe, :kate, :christin\}, \emptyset)$
4. $r_{(foaf:age,\text{"41"}^^\text{xs:int})} = (\{:christin\}, \emptyset)$

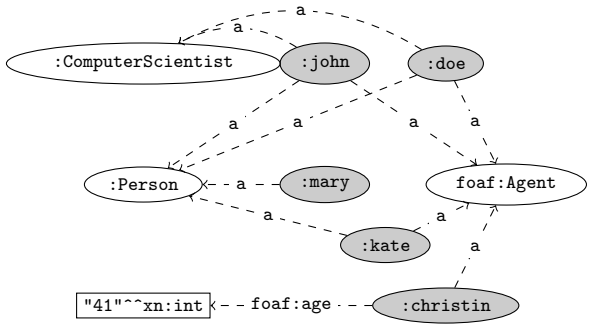

Figure 1: RDF graph of the running example. Ellipse shaped nodes represent RDF resources, literals are rectangular nodes and dashed arcs RDF properties.

2.2 Super rules

With rules, we can already compress some of the information contained in RDF datasets. We can yet go exploit the logical features in data sets further by considering subjects shared by different rules. To this end, we introduce the notion of *super rules*. Given a rule $r_{(i,j)}$ another rule $r_{(k,l)}$ is called a *super rule* if the following equation holds:

$$S(r_{(i,j)}) \subseteq S(r_{(k,l)}) \cup D(r_{(k,l)} \tag{3}$$

Super rules r' basically generalize existing rules r and are akin to stating that if r holds for resource, then r' does as well. We exploit this notion to further compress rules by computing those super rules as described in the subsequent section. Upon serialization, rule r possibly point to a set of super rules $Sup_r = \{r_1, ..., r_n\}$.

3. APPROACH

In the following, we begin by giving an overview of SCARO. Thereafter, we present the compression and decompression algorithms implemented by our approach in detail.

3.1 Basic Considerations

3.1.1 Consideration 1

The basic idea behind SCARO is to make use of logical implications contained in the data to minimize the number of triples that need to be stored so as to be able to reconstruct the input data completely. Consider for example the knowledge base K that describe persons and their occupations shown in Example 1. Our first insight is that instead of storing triples of the form `?x a :Person` , one can simply (1) create a rule that generates triples of that form and (2) store all `s` with `(s, a, :Person)` $\in K$. To this end, we can simply use a rule as presented in the section above. Now, if one can infer from the data that having the occupation `ComputerScientist` means that one is a human (formally: `x a :ComputerScientist` $\rightarrow$ `x a :Person`), then one can simply store this superrule and does not need to compress triples of the form `x a :Person` when `x` is a computer scientist.

3.1.2 Consideration 2

A further consideration of importance is that compression is always time-consuming for large datasets. Thus, compression approaches that scale to large datasets must provide means to add or delete triples from a compressed knowledge base K without having to reprocess the whole of K. The delete and add set of each rule allow dealing with addition and deletions at runtime. Every time a triple (s, p, o) is updated or deleted from the underlying RDF graph, we simply delete s from the set of subjects of the corresponding rule $r_{(p,o)}$ and add s to its set of delete subjects. Subjects in the set of delete subjects should not produce any triple upon decompression and should not be forwarded to the set of super rules. Thus, we additionally copy s to the set of subject of any precalculated super rule $sr \in Sup(r_{(p,o)})$. Through this operation, we can ensure that any previous calculated super rule relation is still valid. For any new statement (s, p, o) we simply add s to any existing rule $r_{(p,o)}$ and add s to the set of delete subjects of every super rule of $r_{(p,o)}$ iff o is a RDF resource.

3.1.3 Consideration 3

Literal values are less commonly shared than object property values. This has to do with the infinite space of values that a datatype property can have. In contrast, an object type property can only adopt a finite (and usually considerably smaller) number of property values. Thus, SCARO differentiate between datatype and object type properties and compresses RDF data based mainly on object type properties. To store triples with datatype properties, we use an RDF graph, which we dub value graph.

3.1.4 Consideration 4

The URIs of RDF resources can be quite a long string. We can thus use a dictionary-based approach to reference any RDF resource. To build such dictionaries, all we need to do is to assign an index to each subject, property and object. Here, we opt for building two dictionaries: One for subjects and objects and a second one for all properties. We store both subjects and objects in the same dictionary, to take advantage of any overlaps across those two sets. We treat predicates differently because of their commonly low number and low overlapping with the sets of subjects and objects.

Given these considerations, we can now formulate a general approach for SCARO: First, index the dataset. Then, generate rules and profiles operate on these indices. In an subsequent step, compute all super rules. In a third and final step, write all data (dictionaries and rules) into a single file and compress it using character-based compression (in our case bzip2). In the following, we describe these steps in details (see Algorithm 1).

3.2 Data Compression Algorithm

Algorithm 1 describes the compression carried out by SCARO. Given a knowledge base K, we first construct the set P of all profiles. To this end, we begin by iterating over all properties $p : \exists(s, p, o) \in K$. For each given property p we retrieve all particpating objects o for which there exists a triple $(s, p, o) \in K$ (lines 6-7 of Algorithm 1). Thus, given any property in K, we are able to query the RDF model of K for the set of all subjects which participate in the (p, o) profile: $S(p, o) := \{s|\exists(s, p, o) \in K\}$ (line 10).

For all s, p, and o, we use a dictionary to compute and store the index $i(s)$, $i(p)$ and $i(p)$ of s, p and o. Remember that all subjects and objects are stored in the dictionary $Dict_s$ and all properties in a separate dictionary $Dict_p$. All subsequent computation for the rules is done using the indexes from the corresponding dictionaries. For a given profile (p, o) and its participating set of subjects $S(p, o)$ we differentiate the type of o. First, in case o is a RDF resource (line 12-19) we create a dictionary entry and get its index $i(o)$. The corresponding rule $r_{i(p),i(o)}$ is created using these indices. For all participating subjects $s \in S(p, o)$ we also create dictionary entries and add the indices to the rule $r_{i(p),i(o)}$. Additionally we save a mapping for each subject $i(s)$ pointing to the set of rules this subject is part of (line 18). This mapping is later used to efficiently compute the super rules. Now, if the object o of the relation (p, o) is an RDF literal we do not create a dictionary entry for it, but we add all triples $\{(i(s), i(p), o|s \in S(p, o)\}$ to the RDF model of the value graph V (line 21-24). Thus, after iterating over all object property relations, the set of rules for K holds all flat rules while V contains any addtional statements where the object is a RDF literal.

For the sake of illustration, consider the example 1 displayed as RDF graph in Figure 1. Running algorithm 1 until step 27 without computing the super rules results into the set R of all rules holding 3 rules:

1. $r_{(a,:ComputerScientist)} = \{: john, : doe\}$
2. $r_{(a,:Person)} = \{: john, : doe, : mary, : kate\}$
3. $r_{(a,foaf:Agent)} = \{: john, : doe, : kate, : christin\}$

Algorithm 1 Compression

```
1: procedure COMPRESS(File f)                    ▷ Compress the file f
2:     K := loadModel(f)                         ▷ Load RDF Graph
3:     R := newSet < Rule >
4:     V := newRDFModel()   ▷ Initialize RDF Model for value graph
5:     Dict_s, Dict_p := newDictionary()         ▷ Initialize Dictionaries
6:     Map < Integer, Rule > sTr := newMap()
7:     for all p : ∃ (s, p, o) ∈ K do
8:         for all o : ∃ (s, p, o) ∈ K do
9:             i(p) := getIndex(s, D_s)
10:            S(p, o) := {s|∃(s, p, o) ∈ K}
11:            i(s) := getIndex(s, Dict_s)
12:            if o is RDF resource then
13:                i(o) := getIndex(o, Dict_s)
14:                r_{i(p),i(o)} := newRule()
15:                for all s ∈ S(p, o) do
16:                    i(s) := getIndex(s, Dict_s)
17:                    r.addSubject(i(s))
18:                    sTr.add(i(s), r)
19:                end for
20:                R := R ∪ r
21:            else
22:                for all s ∈ S(p, o) do
23:                    V := V∪(getIndex(s, Dict_s), i(p), o)
24:                end for
25:            end if
26:        end for
27:    end for
28:    for all r ∈ R do
29:        superRules := computeSuperRules(r, sTr, 0)
30:        for all sr ∈ superRules do
31:            if !sr.hasSuperRule(r) then
32:                r.addSuperRule(sr)
33:                remove transitive redundancies (r, sr)
34:            end if
35:        end for
36:    end for
37:    write Dict_s, Dict_p, R, V to file and apply bzip2
38:    return R                                  ▷ Return the all rules
39: end procedure
```

The map sTR that points from the subjects to the rules they particpate in would hold the following 5 entries:

1. (:john,$\{r_{(a,:ComputerScientist)}, r_{(a,:Person)}, r_{(a,foaf:Agent)}\}$),
2. (:doe,$\{r_{(a,:ComputerScientist)}, r_{(a,:Person)}, r_{(a,foaf:Agent)}\}$),
3. (:mary,$\{r_{(a,:Person)}\}$),
4. (:kate,$\{, r_{(a,:Person)}, r_{(a,foaf:Agent)}\}$),
5. (:christin,$\{r_{(a,foaf:Agent)}\}$).

Finally the model of the value RDF graph V consists of only one statement:

1. `:christin foaf:age "41"^^ xs:int .`

Note that we omit the use of dictionaries in the example for the sake of readability. In the algorithm as explained above, we would actually have integers in the set of rules and the mapping between subjects and sets of rules.

The following step of the algorithm exploits our notion of rule subsumption hierachies. We compute a set of super rules for each rule $r \in R$ as described in Algorithm 2. The efficient computation of the super rules is ensured by making use of the map sTr between subjects and the set of rules they participate in.

Algorithm 2 Computation of super rules for a given rule r and it set of subjects $S(r) = \{s_1, ..., s_n\}$

```
1: procedure computeSuperRules(Rule r,
   Map<Integer, Rule> subToRule, Integer δ)
2:     superRules := ⋃ subToRule.get(s_i ∈ S(r))
3:     superRules := superRules\r
4:     for all s ∈ S(r) do
5:         for all sr = (S(sr), D(sr)) ∈ superRules do
6:             if s ∉ S(sr) then
7:                 D(sr) := D(sr) ∪ s
8:                 if |D(sr)| > δ then
9:                     superRules := superRules\sr
10:                end if
11:            end if
12:        end for
13:    end for
14:    return superrules
15: end procedure
```

Algorithm 2 describes our method to compute the set of super rules for a given rule r. It also requires the map from subjects to rules they participate in and a parameter δ stating the maximum amount of subjects added to a delete list of a super rule. Note, per default we do not compute super rules with delete subjects (i.e., $\delta = 0$). The basic approach is to compute the union all rules the subjects of the given rule r participates in (line 2 in algorithm 2). From this set of all possible super rules of r, we then remove the rule r itself. For all subjects $s \in S(r)$, we then compute those super rules sr for which $s \notin S(sr)$ and add r to $D(sr)$ (line 4-7). If $|D(sr)| > \delta$, then sr is removed from the set of super rules for r (line 8-9).

For our example 1 and $\delta = 0$, the following super rules are computed:

1. $Sup(r_{(a,:ComputerScientist)}) = \{r_{(a,:Person)}, r_{(a,foaf:Agent)}\}$
2. $Sup(r_{(a,:Person)}) = \emptyset$ and $Sup(r_{(a,:Agent)}) = \emptyset$.

In contrast, if we set $\delta = 1$, then algorithm 2 returns $\{r_{(a,:Agent)}\}$ as the set of super rules for the rule $r_{(a,:Person)}$ with $D_{r_{(a,:Agent)}} = \{: mary\}$. The other sets of super rules remain unchanged.

The hierarchy computed by Algorithm 2 leads to a redundant hierarchy. This is simply due to the rule subsumption being transitive. Specifically, The super rule relation

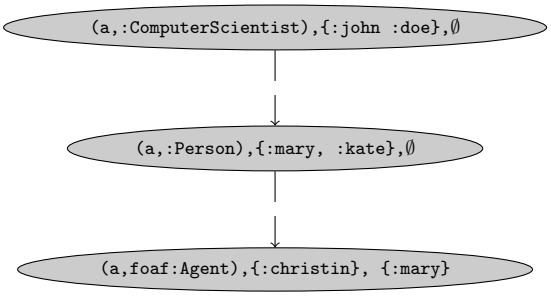

Figure 2: Graphical representation of rules computed for the example 1 allowing delete set to contain 1 subject. Nodes represent a rule $r_{p,o}$ as (p, o), S_r, $D :_r$. Edges represent super relations.

is clearly transitive when $\delta = 0$ and remains quasi transitive for small values of δ. This behavior can be seen in our example: The rule $r_1 = r_{(a,:Person)}$ points to its super rule $r_2 = r_{(a,foaf:Agent)}$ while the rule $r_3 = r_{(a,:ComputerScientist)}$ points to both as super rules. By removing $r_{(a,foaf:Agent)}$ from the set of superrules of $r_{(a,:ComputerScientist)}$ we are still able to produce any triple produced by rule r_3 with subjects in r_1. By these means, we omit the redundant storage of subjects. We implement this insight by postprocessing our hierarchy and removing the subjects $S(r)$ of any rule r from the set of subjects of any of its super-rules r'. In our example, this is equivalent to limiting the subjects in rule r_2 to $S(r_2) = S(r_2) \backslash S(r_1)$. Hence, following the compression of example 1 and the later super rule computation described above the final set of rules looks as follows:

1. $r_{(a,:ComputerScientist)} = (\{: john, : doe\}, D(r_{(a,:ComputerScientist)}) = \emptyset, Sup(r_{(a,:ComputerScientist)}) = \{r_{(a,:Person)}\})$
2. $r_{(a,:Person)} = (\{: mary, : kate\}, D(r_{(a,:Person)}) = \emptyset, Sup(r_{(a,:Person)}) = \{r_{(a,foaf:Agent)}\})$
3. $r_{(a,foaf:Agent)} = (\{: christin\}, D(r_{(a,foaf:Agent)}) = \{: mary\}, Sup(r_{(a,foaf:Agent)}) = \emptyset$

Figure 2 illustrates our forrest of rules and their super rule hierachy.

3.3 Serialization

With the computation of super rules and the creation of a hierarchy, we complete the logical compression of the input knowledge base K. The output of our algorithm is serialized into 2 files which are packed into a tar-archive and subsequently compressed using the standard bzip2 algorithm. The first file holds the serialized dictionaries for subjects/objects $Dict_s$, the dictionary for properties $Dict_p$ and the serialized set of rules R. The second file holds the serialized Turtle representation of our value-based RDF graph V.

All entries in the dictionaries are given subsequent indices. Those indices are not serialized. Rather every dictionary entry is written as a single line. Thus, indices can be reconstructed via line numbers.

To serialize our set of rules R we also have to assign an index for every rule in R. To do this we first sort all rules by their profile (first by the index of the property $i(p)$ then by the index of the object $i(o)$). Thus, every rule is given a number in this order. All rules are stored in this order, once again the index is given by their line number. For a rule $r_{(p,o)}$ all references to a super rule in $sr \in sup_{r_{(p,o)}}$ are stored using the index of sr in R. Another feature of sorting the rules by their profile is that we can omit to explicitly serializing the index of the property $i(p)$ for adjacent rules over profiles with the same property p.

Rules $r_{(p,o)}$ over the profile (p, o) are serialized as shown in Listing 4.

$$i(p) - i(o) - S(r_{(p,o)}) - D(r_{(p,o)}) - Sup(r_{(p,o)}) \quad (4)$$

All subject lists $S(r_{(p,o)})$ and $D(r_{(p,o)})$ are ordered internally by their dictionary index, thus the numbers are serialized based on offsets. Equally the list of super rules $Sup(r_{(p,o)})$ is ordered by the indices of the super rules and also stored offset based.

3.4 Decompression

The compression implemented by SCARO is lossless. In the following we present how to reconstruct the input knowledge base given to our compression approach (see Algorithm 3). First, we decompress the given file using bzip2., thus generating the two files that were computed by SCARO. We then begin by processing the first of the two files. First, we parse the dictionaries and compute the index of each resource based on line numbers in the files. By these means, we map each resource to an index. Subsequently, we parse the set of rules R. For every rule the number of its line is its index in the ordered list of R done upon serialization (c.f. section 3.3). We reconstruct the sets $S(r)$ and $D(r)$ for each rule r based on the indexes of the dictionaries computed in the precedent step and the rule storage syntax shown in Listing 4. Moreover, we reconstruct the hierarchy of rules by mapping the rule indices to rules in R. To reconstruct all triples of the knowledge base we apply the recursive function described in algorithm 4 to build all statements that can be computed for each rule $r \in R$. The approach first checks whether the call is recursive by checking whether the submitted set of subjects is empty. If our approach is not being called recursively on a rule r, i.e., if the subject set it is called with is empty, it computes all triples using the set of subjects $S(r)$ of this rule and calls all super rules of r recursively using the set of subjects $S(r)$.

Finally we load value model V using the standard deserialization features provided by RDF and expand statements within it, because all subjects and properties of statements in V are compressed with the dictionaries $Dict_s$ and $Dict_p$.

Consider the rules generated for Example 1 as depicted in Figure 2 and suppose we already parsed the dictionaries. Suppose our algorithm is called first for $r_{(a,:ComputerScientist)}$. Then the triples `:john a :ComputerScientist` and `:doe a :ComputerScientist` are generated. Subsequently, a recursive call to its parent $r_{(a,:Person)}$ is carried out with these two subjects as input. This call produces the triples `:john a :Person` and `:doe a :Person` because the set of subjects

Algorithm 3 Decompression

```
1: procedure DECOMPRESS(File file)
2:     Unzip file using bzip2
3:     Parse Dictionaries Dict_s, Dict_p and rule set R from
   tar entry
4:     RDFModel model := new RDFModel()
5:     for all Rules r ∈ R do
6:         model := model∪buildTriples(r, ∅)
7:     end for
8:     Load and expand value model V from value model
   tar entry
9: end procedure
```

Algorithm 4 Building triples for Rule r

```
1: procedure BUILDTRIPLES(Rule r_(p,o) = (S(r_(p,o)),
   D(r_(p,o)), Sup(r_(p,o))), Set<Integer> childSubjects)
2:     Set<Statements> stmt
3:     if |childSubjects| = 0 then ▷ If this is no recursive
   call on a super rule
4:         stmt := stmt ⋃_{s_i ∈ S(r_(p,o))} < s_i ∈ S(r_(p,o)), p, o >
5:     end if
6:     for all s ∈ childSubjects\D_r do
7:         stmt := stmt∪ < s, p, o >
8:     end for
9:     for all sr ∈ Sup(r_(p,o)) do    ▷ super rule recursion
10:        if |childSubjects| = 0 then
11:            stmt := stmt∪buildTriples(sr, Sr)
12:        else
13:            stmt:=stmt∪ buildTriples(sr, subjects\D(r_(p,o)))
14:        end if
15:    end for
16:    return stmt
17: end procedure
```

given as parameter isn't empty. Subsequently, another recursive call is carried out on the parent $r_{(a,foaf:Agent)}$ with the same set of subjects. This call produces the two triples `:john a foaf:Agent` and `:doe a foaf:Agent`. Following the same approach, the second call of algorithm 4 produces first `:mary a :Person` and `:kate a :Person` and emits an addtional recursive call to its super rule $r_{(a,foaf:Agent)}$ with `:mary, :kate` as parameter. This call produces only the triple `:kate a foaf:Agent` because `:mary` is part of its delete set: $D(r_{(a,foaf:Agent)})$.
The last call on rule $r_{(a,foaf:Agent)}$ eventually generates the remaining triple `:christin a foaf:Agent`. Thus, we generate all 10 triples of the example which were the base for creating the set of rules. Finally, the remaining statement of the example `:christin foaf:age "41"8s:int` is part of the addtional value based RDF graph V and thus is reconstructed in the last step of algorithm 3.

4. EVALUATION

4.1 Experimental Setup

We evaluated our approach on both real and synthetic data. The real-world data sets varied in size and domain and are shown in Table 1. We also used synthetic data generated using the Leigh University Benchmark (LUBM) [8]. LUBM provides methods to generate synthetic data of varying size for a domain ontology pertaining to university and affiliated

persons. We generated two datasets, which described 50 resp. 100 universities[8]. To remain comparable with related work such as [10], we set both the index and seed parameter to 0.

DataSet	Size (MB)	Triples (K)	OTriples (K)	Domain
AH	71	431	281	Research
DBT	2,160	15,894	15,984	Multi-domain
JM	84	1,048	796	Music
LMDB	428	3,580	1,668	Movies
DBLP	564	8,424	3,201	Publications
LUBM 50	1,092	6,657	4,448	Universities
LUBM 100	2002	13,409	8,959	Universities

Table 1: Datasets used in our experiments. The top portion shows the real-world datasets while the bottom portion shows the synthetic datasets. OTriples stands for the number of triples with object properties. AH stands for Archive Hub. DBT stands for DBbpedia rdf:types. JM stands for Jamendo. LMDB stands for Linked Movie Database.

The goal of our evaluation was to assess the effectiveness of the SCARO on datasets of varying sizes. We thus measured both the *compression ratio* and the *runtime* achieved by our approach and compared them with those achieved by the state-of-the-art approaches RB [10] and HDT [5, 6] on the datasets mentioned above.[9] Throughout our evaluation, we compared SCARO with the best performing version of RB, i.e., RB in combination with the inter-property compression approach. We set the δ parameter to 0 throughout our experiments. We chose this setup because preliminary experiments suggested that setting δ to higher values did not lead to an improvement of the compression ratio achieved by SCARO. Moreover, detecting the right δ automatically can be time-intensive, which can lead to unpracticable runtimes on large datasets.

To compute the compression ratio achieved by the approaches, we used the approach followed in [10] and divided the size of the output of SCARO, HDT and RB with that of the bzip2-compressed input dataset in the N-triples format. Note that smaller compression ratios thus point to a better compression. As time measurement we used the clock time provided by the computer clock of the evaluation hardware. In addition to measuring the compression ratio achieved by SCARO, we also aimed to measure its scalability. We thus studied the runtime well our compression and decompression approaches on fragments of growing sizes generated from the DBLP and LUBM 100 datasets.

All experiments were carried out as a single allocated 2GB of RAM of a server with an AMD Opteron Quad-core Processor (2GHz) running JDK 1.7. Each experiment was ran at least three times. We report the minimal runtime achieved by the different approaches.

4.2 Results

4.2.1 Compression Ratio

We began our evaluation by studying the sizes of the compressed files generated by the algorithms RB, HDT, SCARO and bzip2. Our results are shown in Table 2. Overall, the file sizes generated suggest that SCARO clearly outperforms the native version of all algorithms on all datasets. In particular, SCARO performed particularly well on the DBT dataset, where it putperforms RB by more than 38%. As pointed out in [10], RB and HDT can be combined to achieve even better compression ratios. We thus computed the compression ratios of HDT and compared it with (1) the compression ratios reported in [10] for RB and RB+HDT and (2) the compression ration achieved by SCARO in its native implementation as well as with an extension SCARO+HDT, where the graph V is compressed by using HDT before being stored.

DataSet	bzip2	SCARO	HDT	RB
AH	3.71	**2.18**	6.02	2.82
DBT	87.71	**25.35**	89.64	35.08
JM	7.95	**6.86**	17.22	7.39
LMDB	19.67	**12.10**	31.45	20.26
DBLP	76.95	**45.38**	129.42	69.26
LUBM 50	29.82	**13.68**	36.76	22.75
LUBM 100	60.89	**28.12**	75.48	46.09

Table 2: Sizes of the datasets in MB. The values for RB were computed by using the compression ratios presented in [10].

Figure 3 presents the achieved compression ratios. Note that we only present the results of RB (and not RB+HDT) on the synthetic datasets because the compression ration achieved by RB+HDT were not reported in the reference work. The combination of RB and HDT leads to significantly better compression ratio, especially on knowledge bases with complex ontologies. Thus, RB+HDT outperforms SCARO on two of the real datasets (JM: 0.77 (RB+HDT) vs. 0.86 (SCARO); DBT: 0.22 (RB+HDT) vs. 0.29 (SCARO)). However, our approach clearly outperforms the other on 3 of the real datasets as well as on all synthetic datasets. These results suggest that SCARO is well suited for the largest fraction of the knowledge bases on the Web of Data, which do not have complex ontologies. Interestingly, combining HDT with SCARO leads to better compression ratios in all cases. Thus, it is now the default implementation of SCARO as

[8]Datasets are generated for the `http://swat.cse.lehigh.edu/onto/univ-bench.owl` OWL ontology.

[9]Note that there were slight variations between the sizes of the datasets we downloaded from the Web and used and the sizes reported in [10] except for AH, LUBM 50 and LUBM100, where the dataset sizes were exactly the same. We unsuccessfully requested the exact same datasets and the code from the authors of the paper. However, the authors suggest that the compression ratio of their approach remains close to constant for datasets of the same type but of larger sizes. We thus assume that the compression ratio results still do hold even for the datasets that did not have exactly the same size.

found at the project page.[10]

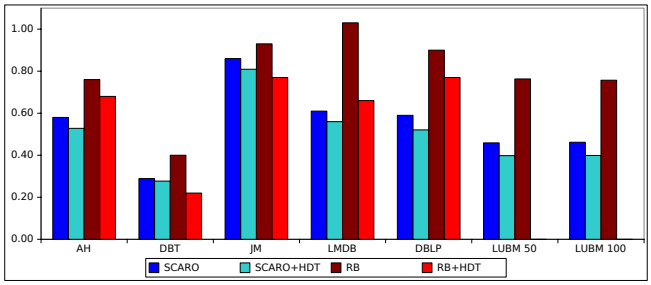

Figure 3: Compression ratios of the different approaches on several data sets. Smaller compression ratio values are better.

4.2.2 *Runtime Comparison*

In addition to comparing the compression ratio achieved by our approach, we measured the runtime it achieves on the datasets at hand. Table 3 summarizes our results. HDT is clearly the more time-efficient approach. However, it achieves poor compression as shown in Table 2. We clearly outperform RB w.r.t. to runtime and are more than 3 times faster on LUBM and more than 15 times faster on AH.

DataSet	SCARO	HDT	RB
AH	19	6	300
DBT	1100	119	800
JM	65	14	800*
LMDB	251	31	1200*
DBLP	754	88	4200*
LUBM 50	241	73	715
LUBM 100	520	152	1485

Table 3: Showing the runtimes of the approaches on all datasets. Note the values for RB on all realworld datasets are estimated due to those values were only available in a diagram in [10].

One might assume that difference in runtime are due to RB finding more rules than SCARO and thus requiring more time to apply them. We thus comared the number of rules resp. transactions computed by SCARO resp. RB (see Table 4). Our results suggest that the number of rules does not have the highest impact on runtimes (compare DBT vs. LUBM 100). In fact, over 50% of the runtime of our approach is spent on I/O.

4.2.3 *Scalability*

We also wanted to know how well our approach scales on similar datasets of growing size. To measure this feature, we evaluated SCARO's runtime on fragments of growing sizes of the LUBM100 and DBLP datasets. In both cases, we use fragments containing between 10% and 100% of the triples in the datasets with a 10% increment. Figure 4 shows the results for the LUBM100 benchmark while Figure 5 shows the results for the DBLP dataset. In both cases, the runtime for the compression and decompression scales linearly with the growth of the input size. Interestingly, the compression

[10] `http://aksw.org/projects/SCARO`

DataSet	SCARO (K)	RB (K)
AH	114	51
DBPT	0.434	9,237
JM	411	336
LMDB	1,600	694
DBLP	458	2,840
LUBM 50	420	1,082
LUBM 100	842	2,179

Table 4: Number of rules resp. transactions computed by SCARO resp. RB in thousands. The

ratio of SCARO improves with the larger amounts of data. We assume that this is simply due to larger datasets leading to more regularities in the data and thus to rules that can compress more resources into the leaves of the rule hierarchy.

Overall, our evaluation suggests that SCARO outperforms the state of the art on typical Linked Data RDF datasets w.r.t. the compression ratio it achieves. Moreover, our approach requires less time to achieve these compression ratios. The compression and decompression times of our approach scale linearly with the size of the datasets, which is a desirable feature for compression approaches. The only drawback of our approach seems to be that it can be outperformed by frequent rule-mining based approaches such as RB when the ontology of the dataset is large. Here, using higher values of δ might lead to an improvement of the compression ratio. The detection of the appropriate δ for a given dataset can yet be very time-consuming. We will thus address this particular problem in future work.

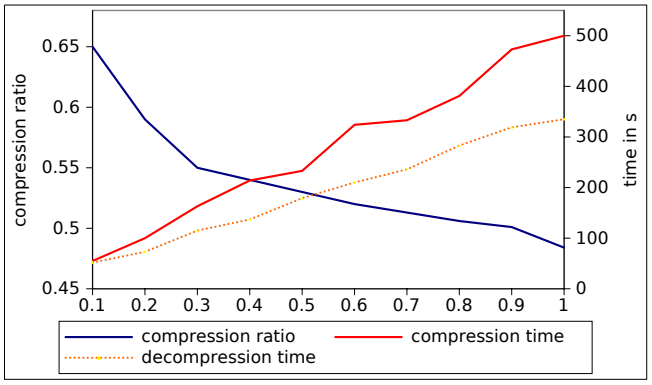

Figure 4: Compression ratio and runtime of SCARO on fragments of LUBM100

5. RELATED WORK

The goal of RDF/XML [3] was to enrich documents or web pages with machine-readable descriptions (metadata). Although, other representations like N3[11], Turtle[12] and RDF/JSON [1] improved the original format both with respect to readability and efficiency, there is still need for more efficient compressing techniques. As the trend to publish large scale

[11] http://www.w3.org/DesignIssues/Notation3
[12] http://www.w3.org/TeamSubmission/turtle/

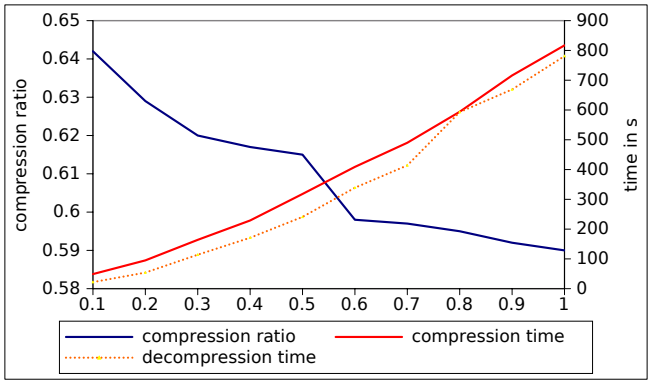

Figure 5: Compression ratio and runtime of SCARO on fragments of DBLP

RDF knowledge bases still holds on, a more data-centric view on RDF is needed.

Fernández, Gutierrez and Martínez-Prieto evaluated several basic approaches to compress RDF data [4]. First, direct compression of the file using traditional compression techniques such as LZ77 or bzip2 were evaluated. Second, adjacency lists, techniques that focus data repeatability and approaches that work on the graph structure of RDF and combine dictionary compression with graph based compression techniques on the triples. While, classic techniques efficiently compress RDF URIs, dedicated data structures and specific graph compression methods are needed to further facilitate compression.

[2, 5] studied generating a compact RDF representation to use support in memory RDF engines. HDT introduced by [5, 6] is a compact format for RDF data. It combines dictionary representation of RDF data with a separation of the triples graph structure. Furthermore it holds header information such as provenance, editorial information and data set statistics. Those three components are stored serparatly in a Header, Dictionary and Triple component, hence the name HDT. Via dictionary encoding HDT takes advantage of the powerlaw distribution in term-frequencies and RDF resources. Compression is not achieved by minimizing the number of triples but rather a compact representation Plain HDT combined with specific compressors for the dictionary and triples component is able to outperform universal compressors such as bzip2, gzip and ppmdi [5]. While providing online RDF management, it allows a up to 15 times more compact representation than N3. All these approaches take advantage of the syntatic verbosity and redundancy of RDF data. Like HDT we also apply a dictionary compression on the RDF resources.

[10] present a semantic rule-based compression (RB Compression) of RDF data based also on logical features of RDF. They take advantage of semantic features in a RDF graph by generating rules and removing triples which can be inferred using these rules. Based upon frequent pattern mining technique parallel FP-Growth, RB Compression is able to infer patterns both within a property (intra property) as well as inter-property. The RDF graph is decomposed into a set of rules, an active graph holding triples these rules can be applied to and a dormant graph holding uncompressed triples. Thereby, RB Compression can be combined with HDT. Furthermore, it allows delta compression by adding new triples based upon a subject to either the active or dormant graph dependent on the rules computed before. We adopted this technique as we partition the RDF graph into two subsets: the one holding statements about RDF resources, we apply our rule detection technique upon and the so called value graph with statements $< s, p, o >$ where o is no RDF resource but a RDF-Literal. So, in theory we also offer online insertion on the compressed graph, by simply adding these statements to the value graph and periodically compress the complete graph again. Furthermore online updates are also possible by simply puting affected subjects to the delete set of the affected rule. [9] introduced with RDFCore another approaches that lower the number of triples. [11, 12] analysed the problem of redundancy detection and elimination on RDF graphs in the presence of rules, constraints and with respect to queries. Thus the deleted data could be reconstructed using these rules. But, as both approaches are application dependent and require human input, their applicability for the ever growing RDF datasets is limited.

Fokoue et al. present summary of ABox (*summary ABox*) of Description Logic ontologies [7]. An ABox consits of role and membership assumptions over individuals. By summarizing similar individuals and assertions they are able to extremely minimize the size of the summary ABox while still supporting consistency detection. If a portion in the summary abox is inconsistent the image of that portion in the original ABox is also inconsistent. Thereby they are able to dramatically scale down consistency checks. While this is no compression technique on RDF data, the basic idea to summarize statements about similar individuals is related to our approach creating rules on similar subjects.

6. DISCUSSION AND FUTURE WORK

We presented SCARO, a logical lossless compression approach for RDF data based on rule subsumption hierachies. Our evaluation of SCARO suggest that it is effective. In particular, we outperform the state-of-the-art approaches RB and HDT w.r.t. the compression ratio we achieve on typical Linked Data sets. Our evaluation yet also points out that frequent item mining is better suited for datasets with large ontologies. One of the main advantages of our approach is that it is scalable. In particular, our scalability evaluation on synthetic and real-world data suggests that SCARO's runtime grows linearly with the input size. In addition, by relying on a separate value graph and delete sets for rules we are able to support delta compression. Finally, SCARO can be used in a parameter-free manner on arbitrary RDF data and offers both a free command line application as well as an graphical user interface.

As SCARO is able to detect rule subsumption hierachies, one can use the core of the compression algorithm to discover implicit ontology knowledge in RDF data. For example, the super-rule between computer scientists and agents in our running example can allow deriving that the class `:ComputerScientist` must be a subclass of `:Person` as well as means to identify errors by computing rules with delete subject sets.

In future work, we will devise an approach to efficiently discover the right value for δ. Moreover, we will improve the runtime of our approach by providing a parallel implementation of the compression and decompression algorithms.

7. REFERENCES

[1] K. Alexander. Rdf/json: A specification for serialising rdf in json. In *Scripting for the Semantic Web*, CEUR Workshop Proceedings, 2008.

[2] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto. Compressed k2-triples for full-in-memory RDF engines. *CoRR*, abs/1105.4004, 2011.

[3] D. Beckett. RDF/XML Syntax Specification (Revised), February 2004.

[4] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto. Rdf compression: Basic approaches. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1091–1092, New York, NY, USA, 2010. ACM.

[5] J. Fernández, M. Martínez-Prieto, and C. Gutierrez. Compact representation of large rdf data sets for publishing and exchange. In P. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Pan, I. Horrocks, and B. Glimm, editors, *The Semantic Web – ISWC 2010*, volume 6496 of *Lecture Notes in Computer Science*, pages 193–208. Springer Berlin Heidelberg, 2010.

[6] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary rdf representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19(0), 2013.

[7] A. Fokoue, A. Kershenbaum, L. Ma, E. Schonberg, and K. Srinivas. The summary abox: Cutting ontologies down to size. *The Semantic Web - ISWC 2006*, pages 343–356, 2006.

[8] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for {OWL} knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2–3):158 – 182, 2005. Selcted Papers from the International Semantic Web Conference, 2004 ISWC, 2004 3rd. International Semantic Web Conference, 2004.

[9] L. Iannone, I. Palmisano, and D. Redavid. Optimizing rdf storage removing redundancies: An algorithm. In M. Ali and F. Esposito, editors, *Innovations in Applied Artificial Intelligence*, volume 3533 of *Lecture Notes in Computer Science*, pages 732–742. Springer Berlin Heidelberg, 2005.

[10] A. Joshi, P. Hitzler, and G. Dong. Logical linked data compression. In P. Cimiano, O. Corcho, V. Presutti, L. Hollink, and S. Rudolph, editors, *The Semantic Web: Semantics and Big Data*, volume 7882 of *Lecture Notes in Computer Science*, pages 170–184. Springer Berlin Heidelberg, 2013.

[11] M. Meier. Towards rule-based minimization of rdf graphs under constraints. In D. Calvanese and G. Lausen, editors, *Web Reasoning and Rule Systems*, volume 5341 of *Lecture Notes in Computer Science*, pages 89–103. Springer Berlin Heidelberg, 2008.

[12] R. Pichler, A. Polleres, S. Skritek, and S. Woltran. Redundancy elimination on rdf graphs in the presence of rules, constraints, and queries. In P. Hitzler and T. Lukasiewicz, editors, *Web Reasoning and Rule Systems*, volume 6333 of *Lecture Notes in Computer Science*, pages 133–148. Springer Berlin Heidelberg, 2010.