Data Analysis of Hierarchical Data for RDF Term Identification

Pieter Heyvaert, Anastasia Dimou, Ruben Verborgh, and Erik Mannens

In: Semantic Technology: 6th Joint International Conference, JIST 2016, Singapore, Singapore, November 2-4, 2016, Revised Selected Papers, 204–212, 2016.

http://dx.doi.org/10.1007/978-3-319-50112-3_15

Data Analysis of Hierarchical Data for RDF Term Identification

Pieter Heyvaert(✉), Anastasia Dimou, Ruben Verborgh, and Erik Mannens

iMinds – IDLab – Ghent University, Ghent, Belgium
pheyvaer.heyvaert@ugent.be

Abstract. Generating Linked Data based on existing data sources requires the modeling of their information structure. This modeling needs the identification of potential entities, their attributes and the relationships between them and among entities. For databases this identification is not required, because a data schema is always available. However, for other data formats, such as hierarchical data, this is not always the case. Therefore, analysis of the data is required to support RDF term and data type identification. We introduce a tool that performs such an analysis on hierarchical data. It implements the algorithms, Daro and S-Daro, proposed in this paper. Based on our evaluation, we conclude that S-Daro offers a more scalable solution regarding run time, with respect to the dataset size, and provides more complete results.

1 Introduction

Data often originally resides in (semi-)structured formats. Tools [1,2] and mapping languages [3,4] allow to describe how Linked Data, via RDF triples, is generated based on the original data. Information structure modeling [5] (henceforth referred to as 'modeling') is required during the creation of these descriptions. This modeling includes the following tasks: (1) identify the candidate entities, their attributes and the relationships among these entities; (2) generate IRIs for the entities; and (3) define the data type of each attribute, if needed. For RDF these tasks align with RDF term identification. However, they can be fulfilled in different ways, and not every way results in the desired RDF triples. Additionally, current tools come short in fulfilling these tasks (semi-)automatically or do not provide the users with the required information to fulfill them manually. This information includes the data model, keys and data types. Though, this information can be found in the data schema, for hierarchical data the schema is not always available, nor always complete, as opposed to databases. Tools, such as XmlGrid[1] and FreeFormatter[2] for XML data, exist to generate these schemas.

The described research activities were funded by Ghent University, iMinds, the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research Flanders (FWO Flanders), and the European Union.

[1] http://xmlgrid.net/xml2xsd.html.

[2] http://www.freeformatter.com/xsd-generator.html.

Y.-F. Li et al. (Eds.): JIST 2016, LNCS 10055, pp. 204–212, 2016.
DOI: 10.1007/978-3-319-50112-3_15

However, they do not give all the aforementioned information, such as keys, and the data type information is not fine-grained enough when working with dirty data. Additionally, manually extracting this information is error-prone and time consuming, as the complete data source needs to be analyzed. In this paper, we introduce a tool[3] to obtain the required information of hierarchical data to address the three tasks. The tool implements two algorithms, Daro and S-Daro, to conduct the data analysis in a scalable way, as the dataset can become large. Based on theoretical analysis, the key discovery of Daro is not always complete, while for S-Daro it is. From our evaluation, we conclude that S-Daro has a better run time when the dataset size increases. The remainder of the paper is structured as follows. In Sect. 2, we discuss the related work. In Sect. 3, we explain, using an example, how the data analysis information can be used to fulfill the modeling tasks. In Sect. 4, we explain the two algorithms. In Sect. 5, we elaborate on the evaluation of the two algorithms. Finally, in Sect. 6, we conclude the paper.

2 Related Work

For XML the data model, keys and data types can be described via the XML schema. However, not in all cases is the schema available, nor complete. Tools exist that allow to generate a schema based on an XML file, such as XmlGrid and FreeFormatter. The same is applicable for JSON and the JSON schema [6]. The tool at http://jsonschema.net can be used to generate a JSON schema given a JSON input. These tools provide data model and data type information. However, the latter lacks detail as a single data type is given when certain data fractions might have different data types. Furthermore, these tools lack key discovery.

3 Example: RDF Term Identification Using Data Analysis

In most cases Linked Data is interpreted as a graph structure, as done by RDF, where the nodes (representing the entities and their attributes) are linked using edges (representing the relationships). Using the XML example in Listing 1.1, we execute the three aforementioned tasks (see Sect. 1) of the modeling process to identify the RDF terms, taking into account which information from the data analysis is used to fulfill each task. We aim to give one possible set of declarative statements of how these terms are generated, using the mapping language RML [4], based on the data model, keys and data types. Subsequently, these statements are used to generate RDF triples.

[3] https://github.com/RMLio/data-analysis-cli; available under the MIT license.

```
<person>
  <firstName>John</firstName>
  <lastName>Doe</lastName>
   <car id="0695-77968-33844">
     <brand>Peugeot</brand>
     <purchDate>12-01-2015</purchDate>
   </car>
</person>
<person>
  <firstName>Jane</firstName>
  <lastName>Doe</lastName>
  <car id="0695-77968-33897">
    <brand>Peugeot</brand>
    <purchDate>16-01-2015</purchDate>
  </car>
 </person>
</persons>
```

Listing 1.1. XML example with person metadata (`http://ex.com/persons.xml`)

Task 1: Identify Entities, Attributes and Relationships Using Data Model. RDF term identification is required to find the appropriate IRIs, blank nodes, and literals. It is supported by using the *data model*. The tree structure of these data sources allows determining possible entities, their literals and relationships by looking at the XML elements and XML attributes: parent elements (i.e., elements with child elements) are identified as entities (IRIs or blank nodes), and leaf elements (i.e., elements with no child elements) and attributes as the entities corresponding IRIs' or blank nodes' literals. Additionally, if a parent element has a parent element as a child, there exists a relationship between the corresponding entities. In the example, the parent elements are `<person>` and `<car>`. This leads to:

```
@prefix rr: <http://www.w3.org/ns/r2rml#> . @prefix rml:
<http://semweb.mmlab.be/ns/rml#> . @prefix xsd:
<http://www.w3.org/2001/XMLSchema#> .

<#PersonMapping>
  rml:logicalSource [
    rml:source "http://ex.com/persons.xml";
    rml:referenceFormulation ql:XPath;
    rml:iterator "/persons/person" ] .
<#CarMapping>
  rml:logicalSource [
    rml:source "http://ex.com/persons.xml";
    rml:referenceFormulation ql:XPath;
    rml:iterator "/persons/person/car" ] .
```

For each parent elements there is a triples map (lines 5 and 10). Each map requires a logical source, which includes the path to the parent element (lines 9 and 14). The leaf elements of `<person>` are `<firstName>` and `<lastName>`. Consequently, they can be identified as literals of the parent element's IRI or blank node, resulting in:

```
<#PersonMapping> rr:predicateObjectMap <#PreObjMapFirstName> .
<#PreObjMapFirstName> rr:objectMap [ rml:reference "firstName" ] .
<#PersonMapping> rr:predicateObjectMap <#PreObjMapLastName> .
<#PreObjMapLastName> rr:objectMap [ rml:reference "lastName" ] .
```

A predicate object map, with an object map, is added to the triples map for the `<firstName>` (lines 1 and 2) and `<lastName>` (lines 3 and 4). The same is the case for the parent element `<car>` and its leaf elements `<brand>`, `<purchDate>`, and the attribute `@id`, resulting in:

```
<#CarMapping> rr:predicateObjectMap <#PreObjMapBrand> .
<#PreObjMapBrand> rr:objectMap [ rml:reference "brand" ] .
<#CarMapping> rr:predicateObjectMap <#PreObjMapID> .
<#PreObjMapID> rr:objectMap [ rml:reference "@id" ] .
<#CarMapping> rr:predicateObjectMap <#PreObjMapPurchaseDate> .
<#PreObjMapPurchaseDate> rr:objectMap <#ObjMapPurchaseDate> .
<#ObjMapPurchaseDate> rml:reference "purchDate" .
```

Furthermore, we conclude that there is a relationship between these two entities, because `<car>` is a child element of `<person>`. This is done by adding a new predicate object map to the triples map for `<person>`, together with a parent triples map that refers to the triples map for `<car>`. This results in:

```
<#PersonMapping> rr:predicateObjectMap <#PreObjMapCar> .
<#PreObjMapCar> rr:objectMap [ rr:parentTriplesMap <#CarMapping> ] ] .
```

Task 2: Generate IRIs Using Keys. In most cases the IRIs have a specific structure, and certain elements of this structure are depended on the data. Additionally, each IRI has to represent at most one entity. This can be accomplished by using *keys* as part of the IRIs. Keys are data fractions that have a unique value for each entity in the original data. In the example, a key identified for the persons is `firstName`. A key identified for the cars is `@id`. This results in:

```
<#PersonMapping> rr:subjectMap [ rr:template "http://ex.com/person/{firstName}] .
<#CarMapping> rr:subjectMap [ rr:template"http://ex.com/car/{@id} ] .
```

A subject map is added to the triples map of each element together with a possible template to generate IRIs using the specified keys.

Task 3: Define Data Types. The *data types* of all values are string with exception of the purchase date (`<purchDate>`; lines 7 and 15), which is a date. This results in the following statement, where date data type is added to the object map corresponding with `<purchDate>`.

```
<#ObjMapPurchaseDate> rr:datatype xsd:date .
```

Subsequently, these statements can be used directly or via a tool, e.g., the RMLEditor [1], to provide the predicates to generate the desired triples.

4 Algorithms

Preliminaries. We structure hierarchical data using a *tree*, in which each node has a set of *properties*, regardless of the data format, e.g., XML or JSON. Each property points to one or more children or data values. For the example in Listing 1.1, the properties of `<person>` are given by the paths `firstName`, `lastName` and `car`. N is the set of all nodes in the tree. $\mathcal{P}$ is the set of all multi-level properties of a node. *Multi-level properties* are the properties of a node including all properties of that node's childnode trees. For the example in Listing 1.1, the multi-level properties of `<person>` are given by the paths `firstName`, `lastName`, `car/brand`, `car/id`, `car/purchaseDate`. P is used for a set of properties where $P \subseteq \mathcal{P}$. The

value v of a node n for a certain (multi-level) property p is defined as $(n, p, v) \in N \times P \times V$, where V represents all values. Two nodes are distinguishable from each other given a set of properties if for at least one property the values of both nodes are not the same. This is formally given in Eq. 1.

$$dist(n, n', P) = \exists p \in P \ \wedge \ \exists (n, p, v) \in N \times P \times V \ \wedge \\ \exists (n', p, v') \in N \times P \times V : v \neq v' \tag{1}$$

Daro. The first algorithm is based on the ROCKER algorithm, which uses a refinement operator for the discovery of keys, proposed by Soru et al. [7]. The operator refines which keys are worth checking, opposed to checking all possible keys. Originally, it was applied for key discovery on RDF datasets. Our version supports hierarchical data sources, and is called 'Data Analysis using the ROCKER Operator' (Daro). It uses a *scoring function* that gives the ratio of the number of nodes that is distinguishable given a set of properties over the total number of nodes ($score(P)$ in Eq. 2). P is a key if $score(P) = 1$, because that means that all nodes are uniquely identifiable using P. Additionally, the function $sortByScore(P)$ returns the properties of P ascendantly ordered using their score, i.e., $\forall p_i, p_j \in P : i \leq j \implies score(p_i) \geq score(p_j)$.

$$score(P) = \frac{|\{n \in N \mid \forall n' \in N : n \neq n' \Rightarrow dist(n, n', P)\}|}{|N|} \tag{2}$$

The refinement operator ($\rho(P)$ in Eq. 3) defines which sets of properties need to be checked next given a set of (previously checked) properties. It requires the properties of $\mathcal{P}$ to be ordered using $sortByScore(\mathcal{P})$.

$$\rho(P) = \begin{cases} \mathcal{P} \text{ if } P = \emptyset, \\ \{P \cup \{p_1\}, \ldots, P \cup \{p_i\}\} : & p'_0 \in sortByScore(P) \ \wedge \\ & (\exists p_j \in \mathcal{P} : p'_0 = p_j) \ \wedge (p_i \in \mathcal{P} : i < j) \end{cases} \tag{3}$$

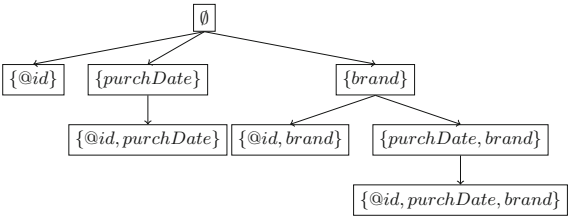


Fig. 1. Complete refinement operator tree for `<car>` of Listing 1.1

We explain the operator using `<car>` from Listing 1.1. In Fig. 1 you can see the complete refinement tree for the child elements and attributes of `<car>`, i.e.,

$\mathcal{P} = \{@id, purchDate, brand\}$. First, we start with an empty set of sets ($= \emptyset$), because we do not have a set of properties. Applying the operator on the empty set ($= \rho(\emptyset)$) results in the following sets of properties: $\{@id\}$, $\{purchDate\}$ and $\{brand\}$. This is visualized in the second level of the tree. The sets on the third and fourth level of the tree are generated by applying refinement operator on each element on the second and third level, respectively. The theorems and proofs regarding the operator are given in the original work by Soru et al. [7]. Additionally, we created a function to generate the data model and a method that analyzes the values of the properties in order to provide the data types.

Algorithm 1. Daro

```
1: nodes ← xml.query(nodePath)
2: foundKeys ← [ ]
3: if ¬nodes.isEmpty() then
4:     paths ← getPaths(nodes)
5:     buildTreeAndIndex(nodes,paths)
6:     model ← getModel(paths)
7:     paths ← sortByScore(paths)
8:     if score(paths) = 1 then
9:         q ← new PriorityQueue()
10:        q.add(∅, 0)
11:        while ¬q.isEmpty() do
12:            P ← q.pop()
13:            P' ← ρ(P)
14:            for p in P' do
15:                s ← score(p)
16:                if s = 1 then
17:                    foundKeys.add(p)
18:                else
19:                    q.add(p, s)
20:                end if
21:            end for
22:        end while
23:    end if
24: end if
```

Algorithm 2. S-Daro

```
1: nodes ← xml.query(nodePath)
2: trees ← [ ]
3: if ¬nodes.isEmpty() then
4:     paths ← getPaths(nodes)
5:     model ← getModel(paths)
6:     possibleKeys ← generateKeys(paths)
7:     for node in nodes do
8:         for key in possibleKeys do
9:             if ¬key.parent.valid() then
10:                groups ← [ ]
11:                for path in key do
12:                    value ← node.query(path)
13:                    analyze(value)
14:                    tree ← trees.search(path)
15:                    group ← tree.search(value)
16:                    groups.add(group)
17:                end for
18:                if groups.hasDupNode() then
19:                    key.valid(false)
20:                end if
21:            end if
22:        end for
23:    end for
24: end if
```

The pseudo-code[4] of the algorithm can be found in Algorithm 1. The properties and the nodes are used to build a search tree of the nodes and an index over the values, during which also the data types are determined (line 5). The data model is generated using the properties (line 6). If the score of the set of all properties is 1, then all nodes are unique when taking into account all properties (line 8). Only when this is true, we continue the key discovery.

Keys that are supersets of already found keys will not be returned, because the algorithm only adds set of properties to the queue again when they are not keys. Therefore, the number of found keys might be smaller than the total number of keys. It depends on the data which keys will be found and which keys not, as the refinement operator is based on the scores of the properties. These scores are based on the actual values of the data. However, the algorithm always returns all keys consisting of one property, together with all the keys that contain

[4] For brevity, we did not include the code that allows users to determine the data model, keys, and data types separately.

a property that on itself is not a key. The reason is that the empty set (added on line 10) results in checking all possible keys consisting of one property, and properties that are not a key are used to generate new possible keys using the operator (line 19) until a key is found or none can be found.

Key discovery is the most expensive part of the analysis, because the different elements of the data have to be compared. The other elements of the data analysis only require a single pass over the data. However, they are done during the key discovery, because it is needed to iterate over the data in any case.

S-Daro. The second algorithm is called 'Scalable Data Analysis using the ROCKER Operator' (S-Daro). While building upon the ROCKER algorithm, it builds up an index for each property containing all possible values present in that dataset together with the nodes that have this value. Additionally, it does not use the scoring function to lower the run time. Algorithm 2 contains the pseudo code. Using the refinement operator of the previous algorithm, we determine all the possible sets of properties (line 6). They are all possible keys. Additionally, for each set we remember on which other set it was based, if applicable. In the refinement operator tree, this is the set on the lower level to which it connects. We call this set the parent set. A set is only evaluated if the parent set is not valid (i.e., not a key; line 9). If the parent set is valid than the current set stays valid, because the properties of the current set are a superset of the properties of the parent set [8]. If for all properties with those values there is one node (besides the current node) that is present (line 18), than the set is not a key. The current node and that specific node are indistinguishable using these properties.

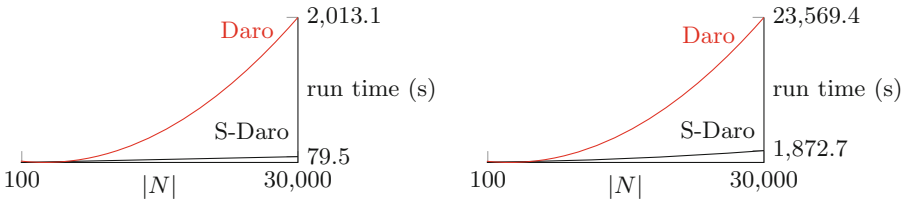
As opposed to Daro, this algorithm returns all keys, because, besides the keys that were marked valid during checking, also the keys that have a valid parent key are valid keys. Like for Daro, key discovery is the most expensive part of the analysis, because the different elements of the data have to be compared.

5 Evaluation

In this section, we elaborate on the evaluation conducted on Daro and S-Daro. The criterion of the evaluation is the run time, because the algorithms are only useful for practical purposes if they finish within a reasonable amount of time. We have evaluated[5] both algorithms using 4 sets of 240 artificially generated files[6]. These files have between 100 and 30,000 nodes, and have between 6 and 13 properties. Their data is about people and their jobs. In Fig. 2a and b plots of the fitted functions of the run times for both algorithms can be found for 6 and 13 properties, respectively. We see that S-Daro outperforms Daro, when the number of nodes becomes larger. The functions are polynomial of the second

[5] All experiments were conducted on a 64-bit Ubuntu 14.04 machine with 128 GB of RAM and a 24-core 2.40 GHz CPU. Each algorithm was run in a Docker container and was able to use at any moment a maximum of 8 GB of RAM and 1 CPU core.

[6] http://rml.io/data/ISWC16/ph/files.



(a) 6 properties/node (b) 13 properties/node

Fig. 2. Daro vs S-Daro

degree for both algorithms. Nevertheless, the function for S-Daro rises slower than for Daro, because the coefficient of the quadratic number of nodes of S-Daro remains small when compared to the coefficient for Daro. However, the coefficient for S-Daro can still be fitted to an exponential function. The reason for this is the exponential growth of possible keys in function of the total number of properties [7]. Therefore, when the number of properties becomes too large even S-Daro might not be able to provide a result within a desired time frame.

6 Conclusion

Our tool implements the two algorithms Daro and S-Daro with support for XML data sources. However, they are applicable to other formats of hierarchical data, such as JSON. Although both algorithms benefit from the refinement operator regarding their run times, the evaluation showed that S-Daro outperforms Daro when the number of nodes becomes larger. Furthermore, the incompleteness of the key discovery of Daro drives the choice towards S-Daro when all keys are required. However, certain use cases might find the results of Daro sufficient.

References

1. Heyvaert, P., Dimou, A., Herregodts, A.-L., Verborgh, R., Schuurman, D., Mannens, E., Walle, R.: RMLEditor: a graph-based mapping editor for linked data mappings. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 709–723. Springer, Heidelberg (2016). doi:10.1007/978-3-319-34129-3_43
2. Pinkel, C., Schwarte, A., Trame, J., Nikolov, A., Bastinos, A.S., Zeuch, T.: DataOps: seamless end-to-end anything-to-RDF data integration. In: Gandon, F., Guéret, C., Villata, S., Breslin, J., Faron-Zucker, C., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9341, pp. 123–127. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25639-9_24
3. Das, S., Sundara, S., Cyganiak, R., R2RML: RDB to RDF mapping language. Working group recommendation, W3C. http://www.w3.org/TR/r2rml/
4. Dimou, A., Sande, M.V., Colpaert, P., Verborgh, R., Mannens, E., Rik Van de Walle, R.M.L.: A generic language for integrated rdf mappings of heterogeneous data. In: Workshop on Linked Data on the Web (2014)

5. Chen, P.P.-S.: The entity-relationship model - toward a unified view of data. ACM Trans. Database Syst. (TODS) **1**(1), 9–36 (1976)
6. Galiegue, F., Zyp, K., Json schema: core definitions and terminology. In: Internet Engineering Task Force (IETF) (2013)
7. Soru, T., Marx, E., Ngonga Ngomo, A.-C.: ROCKER - a refinement operator for key discovery. In: Proceedings of the 24th International Conference on World Wide Web, pp. 1025–1033. International World Wide Web Conferences Steering Committee (2015)
8. Pernelle, N., Saïs, F., Symeonidou, D.: An automatic key discovery approach for data linking. Web Semant. Sci. Serv. Agents WWW **23**, 16–30 (2013)