Integrating Nested Data into Knowledge Graphs with RML Fields

Thomas Delva(✉)[0000−0001−9521−2185], Dylan Van Assche[0000−0002−7195−9935], Pieter Heyvaert[0000−0002−1583−5719], Ben De Meester[0000−0003−0248−0987], and Anastasia Dimou(✉)[0000−0003−2138−7972]

IDLab, Department of Electronics and Information Systems, Ghent University - imec
Technologiepark-Zwijnaarde 122, 9052 Ghent, Belgium
{firstname.lastname}@ugent.be

Abstract. To support business decisions or improve operational efficiency, heterogeneous data is often integrated into a knowledge graph. This integration can be achieved with one of the existing declarative mapping languages, which offer declarative data integration in the form of knowledge graphs. However, current mapping languages cannot always integrate data with nested structure, such as JSON or XML files or JSON documents stored in a database column. We designed a backwards-compatible extension of the RDF Mapping Language (RML) which empowers it to integrate nested data: RML fields. In this paper, we introduce RML fields, compare it with the state of the art in mapping languages, and validate it on mapping challenges formulated by the Knowledge Graph Construction W3C community group. Our extension allows addressing several of the challenges related to nested data that were previously not possible. RML fields can integrate even more datasets into knowledge graphs with all the advantages of using a language specially designed for that purpose. Our extension intends integrating multiple data sets independently, but some use cases require joins or other operations during knowledge graph generation, which we will investigate in the future.

1 Introduction

Graph structures recently became a popular way [10] to organize information: the so-called knowledge graphs [11]. Declarative mapping languages are often used to integrate non graph data into a knowledge graph [7]. A declarative mapping language allows describing schema and data transformations. R2RML, the W3C-recommended declarative mapping language creates knowledge graphs from tabular input data in databases [6]. R2RML was quickly extended to cover more input formats [8], but it also comes with added challenges.

References to common data formats like JSON or XML may return multiple values and these values can be composite: they may again contain multiple values. In contrast, a reference to tabular data typically returns exactly one, non-composite value: the value in a table cell. These two things, multiple and/or composite values, can occur independently of each other: a reference could return one value composed of different attributes, it could return multiple non-composite values, such as integers, or it could return any other combination of multiple and composite values. For instance, multiple objects are returned by applying the JSONPath reference `$.characters.[*]` on the JSON document in Listing 1a and each returned object is itself composed of several attributes: `firstname` and `items`. Declarative mapping languages that integrate such formats as JSON or XML use references that return multiple and composite values, but current mapping languages do not completely handle this. That is why several[1] of the challenges[2] the KGC W3C Community Group identified are related to handling references that can return multiple and/or composite values.

Integrating mixed-format data faces a similar challenge: what if data in one format contains multiple or composite values stored in another format? Examples are JSON objects stored inside a database column (Listing 1b) or multiple values stored as a delimiter-separated string. While certainly unnormalized (it violates the first normal form for relational databases [3]) such data is not unrealistic.

We extended the RDF Mapping Language (RML) [8], which already allows integration of heterogeneous data, with a nested iteration model. The nested iteration model empowers RML to write nested loops over input data. Nested iterations solve both previously mentioned problems: (i) references returning multiple or composite values can be treated as a deeper iteration level and (ii) every iteration level can iterate over data in a different format.

The rest of the paper is structured as follows. We give an overview of how current mapping languages integrate nested data in Section 2. Then, we introduce the "fields" extension to RML and show how nested fields allow integrating nested data in Section 3. We show how RML fields can handle the challenges related to integrating nested data formulated by the W3C Knowledge Graph Construction Community Group in Section 4. Finally we conclude in Section 5.

[1] access-fields-outside-iteration, generate-multiple-values, multivalue-references, process-multivalue-reference and rdf-collections

[2] `https://github.com/kg-construct/mapping-challenges/`

```
{ "characters": [
  { "firstname": "Ash",
    "items": [
      {"name":"gloves", "weight":340},
      {"name":"sword", "weight":4400} ]},
  { "firstname": "Misty",
    "items":[
      {"name":"gloves", "weight":340},
      {"name":"mittens", "weight":300},
      {"name":"hat", "weight":800} ]} ]}
```

(a) Example of tree-structured data in the JSON format.

```
firstname;|items
Ash;      |[{"name": "gloves",
          | "weight": 340 },
          |{"name": "sword",
          | "weight": 4400 }]
Misty;    |[{"name": "gloves",
          | "weight": 340 },
          |{"name": "mittens",
          | "weight": 300 },
          |{"name": "hat",
          | "weight": 800}]
```

(b) Example of mixed-format data: JSON object stored in a CSV column.

Listing 1: Current mapping languages cannot successfully handle this nested data.

```
:people/Ash/items/gloves    :weight 340 .
:people/Ash/items/sword     :weight 4400 .
:people/Misty/items/gloves  :weight 340 .
:people/Misty/items/mittens :weight 300 .
:people/Misty/items/hat     :weight 800 .
```

Listing 2: This graph cannot be created from Listing 1a or Listing 1b with current languages, as it mixes data from multiple hierarchical levels (**bolded**).

2 Related work

With the increasing prevalence of RDF as a format for data on the web, W3C sought to standardize the RDF generation procedure. To this end, two recommendations were published related to generating RDF from relational databases: the Direct Mapping [1] and R2RML [6] recommendations. Direct Mapping is a transformation that generates an RDF graph with the same structure and contains exactly the same information as a relational database. R2RML is a declarative mapping language that can be used to define customized mappings from a relational database to RDF. With R2RML, information in a database can be used to generate RDF graphs with different structures than the database itself.

R2RML was soon generalized by RML [8] to be extended towards other input data formats, e.g., CSV, JSON and XML, than relational databases. To achieve this, RML introduces, among others, the concept of reference formulation. A reference formulation specifies for each integrated data set how data elements in that data set should be referred to. For example, RML considers (i) XPath expressions to refer to data in XML format, (ii) column names to refer to data in CSV/TSV format or relational databases, and (iii) JSONPath expressions to refer to data in JSON format. However, in going beyond relational databases, references to non-relational data that return multiple values are not considered, even though they may be needed for e.g., XML and JSON.

Other mapping languages, such as xR2RML [13] and ShExML [9], were proposed to cover some of RML's limitations but none offers a complete solution. xR2RML [13] extends both R2RML and RML and was the first to handle challenges that come with nested input data. For this reason, xR2RML introduced the nested term map and mixed-syntax paths.

Nested term maps can be used to generate triples from hierarchical data, where one of the triples' terms is generated from a deeper level of the input data's hierarchy. However, it becomes difficult to refer to data stored in different hierarchical levels in the input data.

xR2RML introduced the `xrr:pushDown` term to address the aforementioned issue, which allows "pushing down" values from a higher hierarchical level into a lower hierarchical level: those values are "remembered" during iteration over the lower level. For example, in Listing 3, a nested term map is used together with `xrr:pushDown` to generate URIs from data on different hierarchical levels: `:people/Ash/items/gloves` is created by pushing `Ash` from level one in

```
<#Characters>
  a rr:TriplesMap;
  xrr:logicalSource [
    xrr:query """db.characters.find()""" ;
    rml:iterator "$.characters[*]" ] ;
  rr:subjectMap [
    rr:template
      ":people/{$.firstname}/" ] ;
  rr:predicateObjectMap [
    rr:predicate ex:hasItem;
    rr:objectMap [
      xrr:reference "$.items[*]" ;
      xrr:pushDown [
        xrr:reference "$.firstname" ;
        xrr:as "firstname" ] ;
      xrr:nestedTermMap [
        rr:template
          ":people/{$.firstname}/items/{$.name}"
      ] ] ] .
```

```
:people/Ash   :hasItem
    :people/Ash/items/gloves ,
    :people/Ash/items/sword .
:people/Misty :hasItem
    :people/Misty/items/gloves ,
    :people/Misty/items/mittens ,
    :people/Misty/items/hat .
```

Listing 3: This xR2RML mapping (left) partially handles the example in Listings 1 and 2: xR2RML can generate *single terms* from data from across the input hierarchy (shown **bolded** on the right), but not full triples, as is needed in Listing 2.

Listing 1a down to level two (inside the nested `items` array), where it can be used together with `gloves` to generate the needed URI. `xrr:pushDown` can be used to solve many practical cases, but the nested term map, being a term map, by definition generates individual terms in a triple, e.g., the subject *or* the object terms. Therefore, cases where data from different hierarchical levels is used to generate *more than one term* in a triple, e.g., the subject *and* the object terms, are not accounted for with `xrr:pushDown`. This is the case in the graph in Listing 2: there, `"Ash"`, `"gloves"`, and `340` come from more than one hierarchical level and are used in the subject and in the object. Therefore it is impossible to generate this graph using xR2RML.

Independently of nested term maps, xR2RML introduced mixed-syntax paths. These paths generalize the use of a single reference formulation and can be used to refer to data stored in mixed formats by using mixed reference formulations (syntaxes). An example explains the idea best: if JSON objects are stored in a database column, fields of such a JSON object can be referred to with an expression like `Column(.)/JSONPath(.)`.

ShExML [9] uses ShEx shapes [14] to define the structure of RDF generated from other sources. To extract information from input data, ShExML uses iterators and fields. Iterators give a name to collections in the input data, and fields give a name to individual values. Iterators can be defined nestedly to handle nested input data. Names of fields and iterators are used in ShEx shape templates to specify how the extracted information is written to RDF. For referring to data in different hierarchical levels ShExML introduces "pushed" and "popped" fields which can push down information during nested iteration, similar to xR2RML's `xrr:pushdown`. As such, ShExML is missing little to generate the graph in Listing 2, yet ShExML can only generate URIs from one attribute, while the desired URI `:people/Ash/items/gloves` is generated from

```
ITERATOR chars_it <jsonpath: $.characters[*]> {
  PUSHED_FIELD firstname <firstname>
  ITERATOR items <items[*]> {
    FIELD name <name>
    FIELD weight <weight>
    POPPED_FIELD firstname <firstname> }}

EXPRESSION chars <chars_file.chars_it>

:Item :[chars.items.name] {
  :hasweight [chars.items.weight] ;
  :ownedBy :[chars.items.firstname] }
```

```
:gloves  :hasWeight 340 ;
         :ownedBy   :Ash .
:sword   :hasWeight 4400 ;
         :ownedBy   :Ash .
:gloves  :hasWeight 340 ;
         :ownedBy   :Misty .
:mittens :hasWeight 300 ;
         :ownedBy   :Misty .
:hat     :hasWeight 800 ;
         :ownedBy   :Misty .
```

Listing 4: This ShExML mapping (left) partially handles the example in Listing 1: ShExML can access all the attributes required to generate the triples in Listing 2, but can only make terms from exactly one attribute (shown **bolded** on the right).

Task Language	**Referring to mixed-format data**	**Referring to tree-structured data**	**Writing nested data to graph**
xR2RML	Mixed syntax paths	Nested term map	
ShExML	–	Nested iterator	Linked shapes
RML fields	Reference formulation	Nested fields	(Nested) term map

Table 1: Overview of how different mapping languages handle different tasks related to generating graphs from nested data.

two attributes. In Listing 4 we give a partial solution in ShExML for the input data and desired graph in Listing 1. ShExML does also not provide solutions for input data in mixed formats.

In the next section, we will build on xR2RML's concepts of nested term map and mixed-syntax paths and on ShExML's concepts of fields and nested iterators. Our main contribution on top of these two mapping languages is a method to preserve the relation between related values from different hierarchical levels without explicitly pushing down those values. The relation between xR2RML, ShExML and our contribution, RML fields, is shown in Table 1.

3 RML fields

In this section, we introduce the fields extension of RML. The main contribution of RML fields lies in introducing a greater the separation of concerns between (i) extracting information from data sources and (ii) writing that information to RDF. We first explain how to extract information from nested data using fields (Section 3.1). Then we show an algorithmic representation of the extracted information (Section 3.2) and how that information can be written to RDF (Section 3.4). We close by showing how RML with fields is backwards compatible with RML (Section 3.4).

3.1 Fields

A field gives a name to a reference, as in ShExML. References are the concept in RML for referring to parts of data so those parts can be extracted by an RML engine and used to generate RDF. This extraction involves two other concepts, besides the reference concept: iterators and records. The extraction process as it is defined in RML can be summarized as follows. Given a data source, the iterator extracts a list of records from it. From each record in this list, a reference extracts values to create RDF terms with. For example the iterator `$.characters[*]` returns a list containing the objects in the `characters` JSON array in Listing 1a. The reference `$.firstname` extracts the `firstname` attribute of each record: in consecutive iterations this reference will return `"Ash"` and `"Misty"`.

An RML field can be used to give a human-readable name to the extracted attribute, the field declaration for this is shown on lines 5-7 of Listing 5a. This name can then be used in term maps and URI templates to generate RDF triples from the extracted information. By introducing RML fields, we separate (i) extracting information from data sources from (ii) writing the extracted information to RDF. The former is done with fields, i.e., named references, while the latter is done with term maps, e.g., URI templates.

This separation of concerns is also the reason we do not consider fields which use URI templates to generate values[3]. We only consider URI templates to write extracted information to RDF. Additionally, URI templates can themselves contain multiple references, and it is not trivial how to combine those references' values. In Section 3.2 we explain this combination of values for *fields defined with references*, using a full outer join.

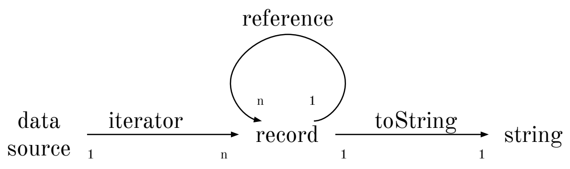

$$\begin{tikzcd}
\text{data source} \arrow[r, "\text{iterator}", "1\quad\quad n"'] & \text{record} \arrow[loop above, "\text{reference}", "n\quad 1"'] \arrow[r, "\text{toString}", "1\quad\quad 1"'] & \text{string}
\end{tikzcd}$$

Fig. 1: Extracting information from data sources with RML fields. An iterator extracts n records from a data source. One or more references extract n further records from each record. A record has a string value.

Multiple values. We should clarify that each reference is treated as if it could return multiple values, as is common for tree-structured or document-structured data formats. Therefore we need to make this clarification about references: a reference extracts *a list of records* from a record. In particular we treat references that return only one value, such as when extracting a unique attribute like

[3] In regular RML, references and URI templates are two types of term maps.

```
1  :source1 a rml:LogicalSource ;
2    rml:source :file ;
3    rml:referenceFormulation
4      ql:JSONPath ;
5    rml:iterator "$.characters[*]" ;
6    rml:field [
7      rml:name "name" ;
8      rml:reference "$.firstname" ; ] ;
9    rml:field [
10     rml:name "item" ;
11     rml:reference "$.items[*]" ;
12     rml:field [
13       rml:name "name" ;
14       rml:reference "$.name" ; ] ;
15     rml:field [
16       rml:name "weight" ;
17       rml:reference "$.weight" ; ] ;
18   ].
```

(a) RML fields snippet to extract information from the nested JSON collections in Listing 1a.

```
:source1 a rml:LogicalSource ;                  1
  rml:source :file ;                            2
  rml:referenceFormulation ql:CSV ;             3
  rml:field [                                   4
    rml:name "name" ;                           5
    rml:reference "firstname" ; ] ;             6
  rml:field [                                   7
    rml:name "items" ;                          8
    rml:reference "items" ;                     9
  rml:field [                                   10
    rml:referenceFormulation                    11
      ql:JSONPath ;                             12
    rml:name "item" ;                           13
    rml:reference "$.[*]" ;                     14
    rml:field [                                 15
      rml:name "name" ;                         16
      rml:reference "$.name" ; ] ;              17
    rml:field [                                 18
      rml:name "weight" ;                       19
      rml:reference "$.weight" ;                20
    ] ; ] .                                     21
```

(b) RML fields snippet to extract information from the mixed-format data (JSON in CSV) in Listing 1b

Listing 5: RML field declarations for extracting information from the files in Listing 1.

`firstname`, as returning a list of length 1. Further, we treat extracted non-composite values, such as the string `"Ash"` or number `340`, as one particularly simple type of record. The relation between data sources, iterators, records and references is pictured in Fig. 1.

Chaining references. The fact that references extract records from records is crucial, as it allows to define references that extract information from the output of other references, since that output is also just records. For example, applying the JSONPath expression `$.items.[*]` on the document in Listing 1a will return each value in the `items` array and each of these returned values is themselves an object composed of attributes `name` and `weight`. With an additional reference `$.name`, applied on the output of the `$.items.[*]` reference, it is possible to extract the name information from that output. We introduce nested fields for such "chaining" of references and an example of a field with two nested fields inside is shown on lines 9–18 of Listing 5a.

Mixed format. To solve the mixed data format problem introduced earlier, we extend the reference formulation concept of RML to specify the formulation for each field separately. An example is shown on line 11 of Listing 5b: the `items` column can be referenced as CSV column, but the JSON objects stored inside can be processed using JSONPath. This process is similar to xR2RML's mixed-syntax paths.

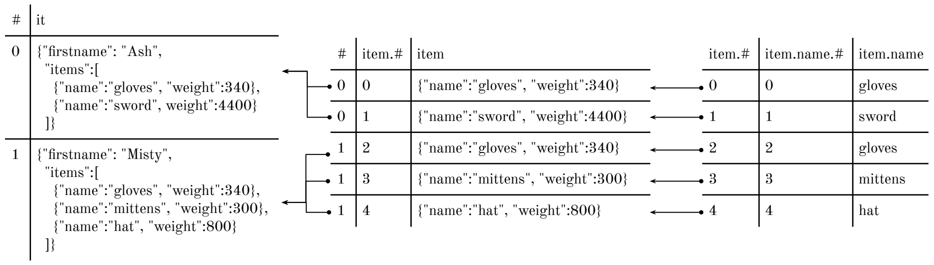
3.2 Algorithmic representation

The information extracted from the input data can be represented as a sequence of iterations. We only introduce this representation as a tool for defining the semantics of RML fields, RML engines are not required to instantiate this sequence and can take shortcuts where needed.

Since each reference can in theory return multiple values, we give each field a separate iteration sequence. Each iteration in the sequence has three attributes containing records, as well as provenance of which previous records created a record. Concretely, for each field we have these attributes (shown in Fig. 2, right-to-left):

- An attribute containing the field's records, one per iteration, as returned by the field's reference. This attribute has the same name as the field[4].
- An attribute containing an index of every record produced by this reference. This index makes it possible for records from this field to be referred to by records from this field's subfields. The name of this attribute is the name of the field concatenated with `.#`.
- An attribute containing references to the index attribute of the field's parent field (or to the iterator's index attribute for fields without a parent field). These references are used to keep track of which iteration in the parent field's iteration sequence the given iteration is based on: which record from the parent field was used as input to create the record in this iteration.

There is also one additional iteration sequence for the iterator. Iterations in this sequence have two attributes: one with the iteration index (attribute name `#`) and one with the iteration record (attribute name `it`). In Fig. 2 the extracted sequences for the iterator and for the fields `item` and `item.name` (all defined in Listing 5a) are shown as tables.



#	it
0	{"firstname": "Ash", "items":[{"name":"gloves", "weight":340}, {"name":"sword", weight":4400}]}
1	{"firstname": "Misty", "items":[{"name":"gloves", "weight":340}, {"name":"mittens", "weight":300}, {"name":"hat", "weight":800}]}

#	item.#	item
0	0	{"name":"gloves", "weight":340}
0	1	{"name":"sword", "weight":4400}
1	2	{"name":"gloves", "weight":340}
1	3	{"name":"mittens", "weight":300}
1	4	{"name":"hat", "weight":800}

item.#	item.name.#	item.name
0	0	gloves
1	1	sword
2	2	gloves
3	3	mittens
4	4	hat

Fig. 2: The tree structure of the data in Listing 1a, here explicitly shown with arrows, is preserved by the index attributes `#` and `item.#`.

The iteration sequences as defined for each field and the iterator can be joined to create one denormalized iteration sequence for the logical source, containing

[4] For nested fields, we consider <`parentname`>.<`declaredname`> as their name, with <`parentname`> the name of the parent field and <`declaredname`> the object of the field's `rml:name` property. For example: `item.name`.

all attributes of all the source's fields' iterations. The RDF generation is defined from this denormalized iteration sequence. The logical source's denormalized iteration sequence is defined as the natural, full outer join [4] of the iterator's iteration sequence and all the fields' iteration sequences. By this definition records from different hierarchical levels in the input data that "belong together", i.e., records along the same root-to-leaf path in the input data's tree structure, will end up in the same iteration in the denormalized iteration sequence. Therefore, RDF can be generated that mixes data from different levels of the input data's hierarchy without losing information about which data belongs together.

This solution avoids concepts as "pushing down" values in a nested iteration (as in ShExML and xR2RML), but achieves the same goal of mixing input data from different hierarchical levels, while preserving the relation between related data. The join by which the denormalized iteration sequence is defined should be an order-preserving join, as the order in the data source might need to be preserved in the generated RDF. The denormalized iteration sequence for the running example is shown in Table 2[5]. Again, this iteration sequence is intended as a tool to specify the semantics of RML fields, an RML engine should not necessarily instantiate this iteration sequence, especially as it might contain duplicate values in attributes higher in the field hierarchy, as can for example be seen in the leftmost four columns in Table 2.

#	it	name.#	name	item.#	item	item.name.#	item.name	item.weight.#	item.weight
0	{...}	0	Ash	0	{...}	0	gloves	0	340
0	{...}	0	Ash	1	{...}	1	sword	1	44400
1	{...}	1	Misty	2	{...}	2	gloves	1	340
1	{...}	1	Misty	3	{...}	3	mittens	1	300
1	{...}	1	Misty	4	{...}	4	hat	1	800

Table 2: Data along the same root-to-leaf path in Listing 1a ends up in the same iteration of this denormalized iteration sequence, after being extracted by the fields in Listing 5a.

3.3 Writing to RDF

Writing information extracted from a source to RDF is done using RML triples maps. A triples map is a combination of term maps. Each term map is defined by a position (subject, predicate or object) and by a way to generate an RDF term from field values. A typical way a term map creates RDF terms from field values is the URI template: this is a URI with "gaps" which are filled in by field values. For example the URI template `:person/{name}` creates URIs from the value of the `name` field. The semantics of a triples map can be defined using

[5] The values of attributes `it` and `item` are omitted for brevity in Table 2, but should be the same as those in Fig. 2.

the denormalized iteration sequence from the previous paragraph: for each iteration in the denormalized iteration sequence, fill in the term maps with the values of the corresponding attributes and return the thus created triples. One caveat is that the denormalized iteration sequence might contain NULL values introduced by the outer join. If a term map would be filled in with a NULL value, the triples created from this NULL value are omitted from the output.

In general, order and duplicates do not matter when generating RDF, since the RDF model itself is unordered and duplicate-free [5], so the duplicates introduced by the full outer join do not matter. However, some RDF constructs such as collections *are* affected by order and duplicates. Further, removing duplicates has been shown to positively affect performance of knowledge graph generation [12]. For these two reasons we introduce a duplicate-removal mechanism to select relevant duplicate-free segments of a logical source's denormalized iteration sequence.

The duplicate removal works as follows: if a combination of subject, predicate and object map uses fields f_1, ... , f_n, then select from the denormalized iteration sequence the distinct values of those attributes (f_1,...,f_n) and their index attributes (f_1.#,...,f_n.#), while preserving order. Triples for this subject, predicate and object map are then generated from the thus selected iteration sequence, which contains all the required information, but no duplicates. An example of such a selected duplicate-free iteration sequence is shown in Table 3, for the three term maps `:person/{name} a :Person`, which together only contain a reference to one field: `name`. It should be clear that in many cases, the duplicate-free iteration sequence needed for a triples map can be instantiated without first instantiating the denormalized iteration sequence from which it is defined.

name.#	name
0	Ash
1	Misty

Table 3: Duplicate-free iteration sequence selected from Table 2 for field `name`.

Finally, to create RDF collections we reuse the nested term map from xR2RML. A nested term map generates collections by grouping several generated terms into one list or set or other RDF collection. By default, terms are grouped into a collection based on the fields used in the other terms of the triples map, but other groupings could be explicitly declared by the user. For example, in Listing 6 we see that the items generated by the nested term map in the object position are grouped on the `name` field since that field is used by the subject term map.

3.4 Backwards-compatibility

As long as no nested fields are used, the fields extension of RML is exactly as expressive as "regular" RML. In fact, there is quite a simple equivalency

```
:triplesMap a rr:TriplesMap ;
  rr:subjectMap [
    rr:template ":person/{name}" ] ;
  rr:predicateObjectMap [
    rr:predicate :hasItems ;
    rr:objectMap [
      rr:termType xrr:List ;
      xrr:nestedTermMap [
        rr:template ":person/{name}/
               ↪ item/{item.name}"
      ] ] ] .
```

(a) Nested term map used with RML fields to create lists of persons' items.

```
:person/Ash :hasItems
  ( :person/Ash/item/gloves
    :person/Ash/item/sword ) .
:person/Misty :hasItems
  ( :person/Misty/item/gloves
    :person/Misty/item/mittens
    :person/Misty/item/hat ) .
```

(b) Graph with RDF lists in the turtle syntax [2]. Created by the triples map in Listing 6a from the logical source in Listing 5a.

Listing 6: Generating lists of `items` grouped per `person` with RML fields.

between regular RML on the one hand and RML with non-nested fields on the other. Regular RML references can be seen as "syntactic sugar" for RML fields in the following way. Given an RML mapping, each reference in a term map can be replaced by adding a field with the same reference and replacing the use of the reference by the name of the field. The obtained RML mapping *with* fields has exactly the same semantics as the original RML mapping. Any regular RML mapping, i.e., without fields or other extensions, can be expressed this way using RML fields.

Similarly, we introduce a syntax for nested fields in the same style as regular RML. The idea is that this syntax is less verbose and more similar in style and spirit to regular RML than the syntax for nested fields from the start of this section. In short, if an expression like `{ref1}.{ref2}` is used in a term map, it should be interpreted as if there is a field with reference `ref1` with a nested field inside with reference `ref2`. A more extended example of this and of the previous syntactic sugar can be seen in Listing 7. This shorter syntax for nested fields is almost as expressive as writing nested fields the long way, except that (i) there is no short syntax for switching reference formulations and (ii) fields with the same reference but different names cannot be written in this shorter syntax [6]. It should also be clear that this shorter syntax is more expressive than regular RML references, as the latter have less support for declaring nested iterations.

The shorter syntaxes proposed in this section overload the RML language, especially when the full syntax and the shorter syntax are used together in one document. For example, the expression `person.name` could be interpreted both (i) as the name of a nested field and (ii) as a JSONPath expression. To deal with this potential ambiguity, we propose a simple precedence rule: IF a term map contains an expression that is the name of a field declared on the term map's logical source, THEN interpret the expression as the name of that field and ELSE, interpret it as a reference in the source's reference formulation.

[6] Such a construction could be needed if a cartesian product of the reference's values is required, e.g. in Listing 8

```
:source1 a rml:LogicalSource ;
  rml:source :file ;
  rml:referenceFormulation ql:JSONPath ;
  rml:iterator "$.characters[*]" ;

:triplesMapPeople a rr:TriplesMap ;
  rr:subjectMap [ rr:template ":person/{$.name}" ] ;
  rr:predicateObjectMap [
    rr:predicate :hasName ;
    rr:objectMap [ rml:reference "$.firstname" ] ; ] ;
  rr:predicateObjectMap [
    rr:predicate :hasItem ;
    rr:objectMap [ rr:template
      ":person/{$.name}/item/{{$.items.[*]}.{$.name}}"]].

:triplesMapItems a rr:TriplesMap ;
  rr:subjectMap [ rr:template
    ":person/{$.name}/item/{{$.items.[*]}.{$.name}}" ] ;
  rr:predicateObjectMap [
    rr:predicate :hasName ;
    rr:objectMap [ rml:reference "{$.items.[*]}.{$.name}" ] ; ] ;
  rr:predicateObjectMap [
    rr:predicate :hasWeight ;
    rr:objectMap [ rml:reference "{$.items.[*]}.{$.weight}" ] ; ] .
```

Listing 7: Shorter syntax for the logical source in Listing 5a. References replacing fields in Listing 5a are **bolded**. References replacing *nested* fields are also ***italicized***.

4 Validation

The W3C community group for Knowledge Graph Construction identified nine challenges for declarative mapping languages. Each challenge gives a high level problem statement of a feature current mapping languages cannot always successfully handle. There is also a set of input files and the expected output graph for each challenge. Of these nine challenges, five are related to nested iteration and references returning multiple values, namely: access-fields-outside-iteration, generate-multiple-values, multivalue-references, process-multivalue-reference and rdf-collections. We go over these five challenges, briefly describe each of them, and explain how RML fields allows to handle the challenge. We published our solutions to these challenges in a public github repository[7].

Access fields outside iteration. The challenge access-fields-outside-iteration relates to accessing data not directly present in the current iteration element. For example, sometimes during iteration over a lower hierarchical level, data in a higher hierarchical level is needed to mint a URI. RML fields solve this challenge by allowing to use field names from different hierarchical levels to create one term. The template `:person/{name}/item/{item.name}` from our running example (Listing 6a), creates URIs from person names (from level one of the input data hierarchy) and item names (from level two of the input data hierarchy). We could handle this challenge the same as our running example, since the input

[7] `https://github.com/RMLio/mapping-challenges-rml-fields/`

```
rml:field [
  rml:name "id" ;
  rml:reference "$.id" ] ;
rml:field [
  rml:name "friendId" ;
  rml:reference "$.id" ] ] ;
rr:subjectMap [
  rr:template "http://example.com/{id}" ] ;
rr:predicateObjectMap [
  rr:predicate ex:hasFriend ;
  rr:objectMap [
    rr:template "http://example.com/{friendId}" ] ] .
```

Listing 8: By declaring a field with reference `$.id` twice with different names, all combinations of the reference's values can be linked by predicate `ex:hasFriend`.

file for this challenge has a very similar structure to the file in our running example.

The challenge has an extension related to referring to data on the same iteration level but in different records, so-called "sibling" data. This challenge can be *almost* tackled by creating two fields with identical references, but different names: `id` and `friendId`. In different iterations in the denormalized iteration sequence, `id` and `friendId` will contain every combination of ids that occur in the data, since the content of these fields is defined by a full outer join. Saying a person is friends with all their siblings in the input data is then as simple as generating `ex:hasFriend` triples connecting people identified by `id` and `friendId`, as shown in Listing 8. This solution would also generate "self-loops": in the produced output, each person would be a friend of themselves, which is not the case in the expected output. To eliminate these self-loops, a filter operation would be needed, but operators are not included in this paper and left as future work.

The challenge has a second extension about joining data in different data sets. While the RML fields extension as it is described in this document does not cover joins, we think the algorithmic representation we introduced gives a solid foundation to add joins and other relational operators.

Generate multiple values. The generate-multiple-values challenge relates to generating multiple literals, each with its own language tag. This feature is typically needed when the input data stores strings together with the strings' languages.

RML fields handles this challenge by iterating over the multiple stored strings and their languages together. Then, using the standard RML feature of the language map, literals can be generated having the language tag stored in the input data, as shown in Listing 9.

This challenge has an extension related to default values for language tags. As it is more related to default behaviour of language maps and not to the iteration model, we did not focus on handling this extension when designing RML fields and this challenge is therefore not covered by this proposal.

Multivalue references. The multivalue-references challenge relates to using references returning multiple values. This typically occurs when referring to tree-

```
rr:objectMap [
  rr:reference "firstname.label" ;
  rml:languageMap "firstname.lang" ]
```

Listing 9: The label and language of `firstname` are iterated over together, so both can be used to create a single RDF literal.

structured data. RML fields handles this challenge because it uses an iteration model where every reference can return multiple values.

The challenge has an extension related to generating triples that "skip" one or more iteration levels: for example, generating triples from data in levels one and three of the input hierarchy (skipping level two). This extension of the challenge is handled by RML fields since field names from any levels of the iteration can be used in triples maps. This includes using field names that skip a level in the same triples map, as shown in Listing 10.

```
rr:subjectMap [
  rr:template "http://example.com/lab/{name}" ] ;
rr:predicateObjectMap [
  rr:predicate ex:hasMember ;
  rr:objectMap [
    rr:template "http://example.com/author/{article.author.name}" ] .
```

Listing 10: This triples map generates triples connecting data from the first level of the input hierarchy (`name`) to data in the third level (`article.author.name`), skipping the second level.

Process multivalue references. The process-multivalue-references challenge, like the previous one, relates to creating RDF from references returning multiple values, with an additional focus on extracting multiple values from strings and on generating different types of RDF collections.

RML fields handle extracting multiple values from strings by treating the strings as comma-separated value records and using the appropriate reference formulation for such records, as shown in Listing 11. Further, RML fields handle the generation of RDF collections by using xR2RML's nested term map, for which an RDF collection type can be specified.

RDF collections. The rdf-collections challenge focuses on the generation of RDF collections. As such, it overlaps with the previous challenge, but this challenge combines generating RDF collections with different structures for the input data.

Again, the generation of RDF collections is handled with RML fields by using xR2RML's nested term map. We find that using the nested term map allows to handle all cases in this more detailed challenge as well, one excerpt from a solution creating RDF lists is shown in Listing 12.

```
rml:field [
  rml:name "author" ;
  rml:referenceFormulation ql:JSONPath ;
  rml:reference "$.authors.[*].name"
  rml:field [
    rml:name "firstname" ;
    rml:referenceFormulation ql:CSV ;
    rml:reference "1" ] ;
  rml:field [
    rml:name "lastname" ;
    rml:referenceFormulation ql:CSV ;
    rml:reference "0" ] ] .
```

Listing 11: By changing reference formulation in nested fields, comma-separated values can be extracted from a JSON string.

```
rr:objectMap [
  rr:termType xrr:RdfList ;
  xrr:nestedTermMap [
    rr:template "http://example.com/author/{article.author}" ] ;
] ;
```

Listing 12: This object map creates RDF lists of authors.

5 Conclusion

We introduced the "fields" extension of RML that allows RML to generate knowledge graphs from nested data. We introduced this approach using a running example that existing mapping languages cannot handle, but RML fields *can* handle. We validated our approach by showing how it can now handle the mapping challenges formulated by the W3C Knowledge Graph Construction Community Group related to nested data. Currently no declarative mapping language can cover all the challenges our proposed RML extension would cover.

So far, we considered generating RDF from one data set with nested structure. However, in some use cases, information from multiple data sets needs to be combined during knowledge graph generation. This includes the "basic" case of joining two data sets on an attribute, but it also includes more advanced cases, such as generating RDF from data set A only if data set B contains no relevant information about an entity. This is exactly what we will investigate next. The herein introduced iteration model provides a strong basis for such combination of data sets, since many operators defined for the tabular format, such as join, leftjoin, etc., carry over to our iteration model.

References

1. Arenas, M., Bertails, A., Prud'hommeaux, E., Sequeda, J.: A Direct Mapping of Relational Data to RDF. Recommendation, World Wide Web Consortium (W3C) (Sep 2012), `http://www.w3.org/TR/rdb-direct-mapping/`
2. Beckett, D., Berners-Lee, T., Prud'hommeaux, E., Carothers, G.: RDF 1.1 Turtle – Terse RDF Triple Language. Recommendation, World Wide Web Consortium (W3C) (Feb 2014), `http://www.w3.org/TR/turtle/`

3. Codd, E.F.: Further normalization of the data base relational model. Data base systems pp. 33–64 (1972)
4. Codd, E.F.: A relational model of data for large shared data banks. In: Software pioneers, pp. 263–294. Springer (2002)
5. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. Recommendation, World Wide Web Consortium (W3C) (Feb 2014), `http://www.w3.org/TR/rdf11-concepts/`
6. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF Mapping Language. Working group recommendation, World Wide Web Consortium (W3C) (Sep 2012), `http://www.w3.org/TR/r2rml/`
7. Dimou, A.: High-Quality Knowledge Graphs Generation: R2RML and RML Comparison, Rules Validation and Inconsistency Resolution. In: Cota, G., Daquino, M., Pozzato, G. (eds.) Applications and Practices in Ontology Design, Extraction, and Reasoning, vol. 49, pp. 55–72. IOS Press (2020)
8. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In: Proceedings of the 7th Workshop on Linked Data on the Web. vol. 1184 (2014)
9. García-González, H., Boneva, I., Staworko, S., Labra-Gayo, J.E., Lovelle, J.M.C.: ShExML: improving the usability of heterogeneous data mapping languages for first-time users. PeerJ Computer Science **6**, e318 (2020)
10. Gutierrez, C., Sequeda, J.F.: Knowledge graphs. Commun. ACM **64**(3), 96104 (Feb 2021). https://doi.org/10.1145/3418294
11. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Gayo, J.E.L., Kirrane, S., Neumaier, S., Polleres, A., Navigli, R., Ngomo, A.C.N., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., Zimmermann, A.: Knowledge Graphs (Mar 2020), preprint
12. Jozashoori, S., Chaves-Fraga, D., Iglesias, E., Vidal, M.E., Corcho, O.: Funmap: Efficient execution of functional mappings for knowledge graph creation. In: International Semantic Web Conference. pp. 276–293 (2020)
13. Michel, F., Djimenou, L., Faron-Zucker, C., Montagnat, J.: xR2RML: Relational and Non-Relational Databases to RDF Mapping Language. Rapport de recherche, Laboratoire d'Informatique, Signaux et Systmes de Sophia-Antipolis (I3S) (Oct 2017), `https://hal.archives-ouvertes.fr/hal-01066663/document/`
14. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.: Shape expressions: an RDF validation and transformation language. In: Proceedings of the 10th International Conference on Semantic Systems. pp. 32–40. New York, NY, United States (2014)