xR2RML: Non-Relational Databases to RDF Mapping Language

Franck Michel, Loïc Djimenou, Catherine Faron Zucker, Johan Montagnat

▶ **To cite this version:**

Franck Michel, Loïc Djimenou, Catherine Faron Zucker, Johan Montagnat. xR2RML: Non-Relational Databases to RDF Mapping Language. 2014. hal-01066663v2

HAL Id: hal-01066663

https://hal.science/hal-01066663v2

Submitted on 8 Oct 2014 (v2), last revised 26 Oct 2017 (v5)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR7271

xR2RML: Non-Relational Databases to RDF Mapping Language

Franck Michel, Loïc Djimenou, Catherine Faron-Zucker, Johan Montagnat

Equipes Modalis/Wimmics

Rapport de Recherche
ISRN I3S/RR 2014-04-FR
Version 2

Oct. 2014 - 34 pages

Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S) - UMR7271 - UNS CNRS
2000, route des Lucioles - Les Algorithmes - bât. Euclide B 06900 Sophia Antipolis - France http://www.i3s.unice.fr

xR2RML: Non-Relational Databases to RDF Mapping Language

1 Introduction

This document describes xR2RML, a language for expressing customized mappings from various types of databases (XML, object-oriented, NoSQL) to RDF datasets.

xR2RML is an extension of the R2RML [1] mapping language. While R2RML addresses relational databases, xR2RML extends this scope to a wider range of non-relational databases. This document leverages the R2RML specification and mainly describes extensions. Consequently, the reader should have a good understanding of R2RML before reading this document.

xR2RML is backward compatible with R2RML.

1.1 Document Conventions

In this document, examples assume the following namespace prefix bindings unless otherwise stated:

Prefix	IRI
rr:	http://www.w3.org/ns/r2rml#
xrr:	http://www.i3s.unice.fr /ns/xr2rml#
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
xsd:	http://www.w3.org/2001/XMLSchema#
ex:	http://example.com/ns#

Vocabulary definitions are formatted in such grey boxes:

Definition

1.2 Query Languages and Data Models

R2RML is specifically focused on the translation of relational databases into RDF datasets. xR2RML extends this scope to non-relational databases. Although it is illusory to seek universal support of any database, our endeavor is to equally support most common relational and non-relational databases. In our approach, we more specifically analyze the requirements to support NoSQL and XML databases. Yet, xR2RML may not support all NoSQL databases, given the large variety of systems behind this word. Nevertheless, we argue that our work can be generalized to some other types of database, for instance object-oriented and directory (LDAP) databases.

Query languages:
Relational databases all support ANSI SQL (or at least a subset of it), and most XML databases support XPath and XQuery. By contrast, NoSQL is a catch-all term referring to very diverse systems [2]. They have heterogeneous access methods ranging from low-level APIs to expressive query languages. Despite several propositions of common query language (N1QL[1], UnQL[2], SQL++ [3], ArangoDB Query Language[3]), no consensus has emerged yet, that would fit most NoSQL databases. Therefore, until a standard eventually arises, xR2RML must be agile enough to cope with various query languages and protocols.

[1] http://www.couchbase.com/communities/n1ql
[2] http://unql.sqlite.org/index.html
[3] http://docs.arangodb.org/Aql/README.html

Remark**:** R2RML relies on the ability of relational databases to support a declarative query language. xR2RML does the same assumption with regards to other types of databases, although this may be limitative. Most NoSQL key-value stores provide simple key-based operations (such as put, get, delete) by means of APIs used in imperative programming languages, but they hardly provide a declarative query language. If such a system is to be mapped using xR2RML, an xR2RML processor designer should implement a mechanism to bridge this gap.

Data models:
Relational databases comply with a row-based model in which all rows have the same columns. NoSQL systems have heterogeneous data models (key-value store, document store, extensible column-store, graph store). Some of them also comply with the row-based model, such as extensible column-stores (also known as column family stores) with the difference that all rows do not necessarily share the same columns (BigTables, Cassandra…). Other databases in which data is formatted in JSON (document stores such as MongoDB, CouchDB…) or XML can hardly be reduced to a row-based model. JSON or XML documents consist of structured values representing collections or key-value associations:

- A JSON dictionary is an ordered association of keys and values, both keys and values may be of any type. A JSON array is an ordered collection of elements, it is a specific case of dictionary in which keys are implicit integer indexes: 0, 1, 2, etc.
- Similarly, a set of mixed XML elements having the same parent element can be seen as an ordered association of keys (element names) and values (element values). A set of XML elements of the same type, having the same parent element, can be seen as an ordered collection. Besides, attributes of an XML element can be seen as a specific type of key-value association.

The JSON and XML models of structured values described above can easily be applied to other databases. In an object-oriented model, an object can be approximated by as a key-value association: keys are attribute names while values are either a scalar, another object (composition or aggregation relationship), or a collection (depending on capabilities of the modelling language : list, map, etc). Similarly, an LDAP directory is organized as a tree. Each node has an identifier and a set of attributes represented as ''name=value'' that are nothing else than a key-value association. A set of attributes with the same name can be interpreted as either as a collection or a key-value association in which keys are not unique. Thus, xR2RML must be able to map data elements from rows as well as structured values (nested collections and key-value associations) to RDF terms.

Note: Below, we shall use the term "structured values" to refer to collections and key-value associations, whatever the representation syntax used.

1.3 xR2RML mapping graphs and mapping documents

An **xR2RML mapping** defines a mapping from a database to RDF.

An xR2RML mapping is represented as an RDF graph called an **xR2RML mapping graph**.

An **xR2RML mapping document** is any document written in the Turtle RDF syntax that encodes an xR2RML mapping graph.

Any R2RML mapping graph is a valid xR2RML mapping graph.

1.4 xR2RML processors

An xR2RML processor is a system that, given an xR2RML mapping and an input database, provides access to the output RDF dataset.

An xR2RML processor has access to an execution environment consisting of:

- a connection to the input database,
- a base IRI used in resolving relative IRIs produced by the R2RML mapping.

The connection is used by the xR2RML processor to evaluate queries against the input database. It must be established with sufficient privileges for read access to all database elements (tables, views, documents, objects…) that are referenced in the xR2RML mapping. How the connection is established, or how users are authenticated against the database, is outside the scope of this document.

2 xR2RML Overview and Examples

This section gives a brief overview of the xR2RML mapping language, followed by simple examples of mapping various types of database to RDF.

An xR2RML mapping refers to logical sources to retrieve data from the input database. A logical source can be either an *xR2RML base table (*for input databases where tables or views exist, such as SQL views), or an *xR2RML view* representing the result of executing a query against the input database. A logical source is assigned a format (property xrr:format) that specifies the format of data retrieved from the input database: xrr:Row, xrr:JSON or xrr:XML.

Each logical source is mapped to RDF using a triples map. As in R2RML, a triples map consists of a subject map that generates the subject of all RDF triples that will be generated from data elements, and multiple predicate-object maps that produce the predicate and object terms of triples.

In the examples below, the following heading declarations are assumed in each mapping graph:
@prefix xrr: <http://www.i3s.unice.fr/ns/xr2rml#>.
@prefix ex: <http://example.com/ns#>.

2.1 Mapping CSV data

The input database in the example below consists of one CSV document. It is assumed that the xR2RML processor is provided a connection to that file, e.g. by means of a descriptor to a file on the local file system.

As a CSV file simply consists of one unnamed table, the logical source is the most simple as can be: it only contains an xrr:format property to specify that data is accessed by rows (which is also the default).

Input data	title, year, director Manhattan, 1979, Woody Allen Annie Hall, 1979, Woody Allen 2046, 2004, Wong Kar-wai In the Mood for Love, 2000, Wong Kar-wai
Mapping graph	<#CSVTriplesMap> xrr:logicalSource [xrr:format xrr:Row;]; rr:subjectMap [rr:template "http://example.org/movie/{title}";]; rr:predicateObjectMap [rr:predicate ex:directedBy; rr:objectMap [xrr:reference "director";];].
RDF triples produced	<http://example.org/movie/Manhattan> ex:directedBy "Woody Allen". <http://example.org/movie/Annie%20Hall> ex:directedBy "Woody Allen". <http://example.org/movie/2046> ex:directedBy "Wong Kar-wai". <http://example.org/movie/In%20the%20Mood%20for%20Love> ex:directedBy "Wong Kar-wai".

2.2 Mapping JSON data

The input database in the example below is a MongoDB database (document store). The query in the logical source retrieves one JSON document from collection "movies", that lists movie directors and movies they directed.

Without further instruction on how to parse the document, JSONPath expressions referring to data elements in the subject and object map will be evaluated against the whole document. For instance, a subject using expression "$.directors.name" will return two terms, while an object map using expression "$.directors.movies.*" will return four terms, one for each movie whatever its director. This will result in mixing up directors and movies. To avoid this, an xrr:iterator property is added to the logical source, specifying that the triples map iteration should occur on each element of the array of directors.

References to data elements (rr:template, xrr:reference), as well as the iterator pattern, are expressed in JSONPath.

Input data	{ "directors": [{ "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"] }, { "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"] }]}
Mapping graph	<#Directors> xrr:logicalSource [xrr:query "db.movies.find({ directors: { $exists: true} })"; xrr:iterator "$.directors.*"; xrr:format xrr:JSON;]; rr:subjectMap [rr:template "http://example.org/director/{$.name}";]; rr:predicateObjectMap [rr:predicate ex:directed; rr:objectMap [xrr:reference "$.directors.*.movies.*";];].
RDF triples produced	<http://example.org/director/Woody%20Allen> ex:directed "Manhattan". <http://example.org/director/Woody%20Allen> ex:directed "Annie Hall". <http://example.org/director/Wong%20Kar-wai> ex:directed "2046". <http://example.org/director/Wong%20Kar-wai> ex:directed "In the Mood for Love".

2.3 Mapping XML data

The example below is very similar to the previous one, using an XML database supporting XQuery. The query in the logical source retrieves "director" XML elements.

To avoid mixing up directors and movies, an xrr:iterator property is added to the logical source, specifying that the triples map iteration should occur on each "director" XML element.

References to data elements (rr:template, xrr:reference), as well as the iterator pattern, are expressed in XPath.

Input data	<directors> <director name="Wong Kar-wai"> <movies>

	<movie>2046</movie> <movie>In the Mood for Love</movie> </movies> </director> <director name="Woody Allen"> <movies> <movie>Manhattan</movie> <movie>Annie Hall</movie> </movies> </director> </directors>
Mapping graph	<#Directors> xrr:logicalSource [xrr:query "//directors/director"; xrr:iterator "//director"; xrr:format xrr:XML;]; rr:subjectMap [rr:template "http://example.org/director/{/director/@name}";]; rr:predicateObjectMap [rr:predicate ex:directed; rr:objectMap [xrr:reference "//movie";];].
RDF triples produced	<http://example.org/director/Woody%20Allen> ex:directed "Manhattan". <http://example.org/director/Woody%20Allen> ex:directed "Annie Hall". <http://example.org/director/Wong%20Kar-wai> ex:directed "2046". <http://example.org/director/Wong%20Kar-wai> ex:directed "In the Mood for Love".

2.4 Mapping data with mixed formats

The syntax of data retrieved from a logical source is specified using the xrr:format property. In some use cases though, it is common to store values in a format which is not the native database format. For instance, an application designer may choose to embed JSON, CSV, or XML values in the cells of a relational database, for performance concerns or application design constraints.

xR2RML allows to reference data elements within such mixed contents with *mixed-syntax paths*. A path with mixed-syntax consists of the concatenation of several path expressions separated by the slash '/' character. Each individual path is enclosed in a syntax path constructor: Column(column-name), CSV(column-name), TSV(column-name), JSONPath(JSONPath-expression), XPath(XPath-expression).

In the example below, the logical source is a relational table with two columns. The second column, Movies, holds values formatted as JSON arrays. The xrr:reference property of the object map uses a mixed-syntax path: *Column(Movies)/JSONPath($.*)*. This expression selects values from column "Movies" and evaluates JSONPath expression "$.*" against the values.

Input data	Table DIRECTORS:

<table>
<tr><td></td><td>
<table>
<tr><th>Name</th><th>Movies</th></tr>
<tr><td>Wong Kar-wai</td><td>["2046", "In the Mood for Love"]</td></tr>
<tr><td>Woody Allen</td><td>["Manhattan", "Annie Hall"]</td></tr>
</table>
</td></tr>
<tr><td>Mapping graph</td><td><#Directors>
rr:logicalTable [
rr:tableName "DIRECTORS";
];
rr:subjectMap [rr:template "http://example.org/director/{Name}";];
rr:predicateObjectMap [
rr:predicate ex:directed;
rr:objectMap [xrr:reference "Column(Movies)/JSONPath($.*)";];
].</td></tr>
<tr><td>RDF triples produced</td><td><http://example.org/director/Woody%20Allen> ex:directed "Manhattan".
<http://example.org/director/Woody%20Allen> ex:directed "Annie Hall".
<http://example.org/director/Wong%20Kar-wai> ex:directed "2046".
<http://example.org/director/Wong%20Kar-wai> ex:directed "In the Mood for Love".</td></tr>
</table>

2.5 Generating an RDF collection from a list of values

As illustrated by the previous example, several RDF terms can be generated by a term map during one triples map iteration step. While this can lead to the generation of several triples, this can also be used to generate structured values in the form of RDF collections or containers.

The example below was already presented in section 2.2. We add to the object map an rr:termType property with value xrr:RdfList. All RDF terms produced by the object map during one triples map iteration step are then grouped in one term of type rdf:List.

Finally, we want to add a language tag to the movie titles. The object map describes the generation of RDF lists, hence it would not make sense to add an rr:language property. To state properties about the members of the generated RDF list, we need a nested term map, introduced by the xrr:nestedTermMap property. A nested term map accepts the same properties as a term map, but it applies to members of RDF collection/container terms generated by its parent term map.

<table>
<tr><td>Input data</td><td>{ "directors": [
{ "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"] },
{ "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"] }
]}</td></tr>
<tr><td>Mapping graph</td><td><#Directors>
xrr:logicalSource [
xrr:query "db.movies.find({ directors: { $exists: true} })";
xrr:iterator "$.directors.*";
xrr:format xrr:JSON;
];
rr:subjectMap [rr:template "http://example.org/director/{$.name}";];
rr:predicateObjectMap [
rr:predicate ex:directed;</td></tr>
</table>

	rr:objectMap [xrr:reference "$.directors.*.movies.*"; rr:termType xrr:RdfList; xrr:nestedTermMap [rr:language "en";]];].
RDF triples produced	<http://example.org/director/Woody%20Allen> ex:directed [a rdf:List; rdf:_1 "Manhattan"@en; rdf:_2 "Annie Hall"@en;] <http://example.org/director/<Wong%20Kar-wai> ex:directed [a rdf:List; rdf:_1 "2046"@en; rdf:_2 "In the Mood for Love"@en;]

2.6 Generating an RDF container with a referencing object map

The example below uses a referencing object map to describe a reference relationship between two logical sources. In addition, it generates an RDF bag from the result of the join condition in the referencing object map.

Triples map <#Movies> generates IRIs for the movies. The referencing object map in triples map <#Directors> uses IRI generated in triples map <#Movies> as the members of an RDF bag (property rr:term:Type xrr:RdfBag).

The join condition in triples map <#Directors> produces a result if a movie title (rr:parent "$.title*") matches at least one movie in the list of movies associated with each director (rr:child "$.movies.*").

Input data	{ "directors": [{ "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"] }, { "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"] }]} { "movies": [{ "title": "Manhattan", "year": "1979" }, { "title": "Annie Hall", "year": "1977" }, { "title": "2046", "year": "2004" }, { "title": "In the Mood for Love", year: "2000"}]}
Mapping graph	<#Movies> xrr:logicalSource [xrr:query "db.movies.find({ movies: { $exists: true} })"; rr:iterator "$.movies.*";]; rr:subjectMap [rr:template "http://example.org/movies/{$.title}";]. <#Directors>

	xrr:logicalSource [xrr:query "db.movies.find({ directors: { $exists: true} })"; xrr:iterator "$.directors.*";]; rr:subjectMap [rr:template "http://example.org/director/{$.name}";]. rr:predicateObjectMap [rr:predicate ex:directed; rr:objectMap [rr:parentTriplesMap <# Movies >; rr:joinCondition [rr:child "$.movies.*"; rr:parent "$.title*";]; rr:termType xrr:RdfBag;];].
Generated RDF triples	<http://example.org/director/Woody%20Allen> ex:directed [a rdf:Bag; rdf:_1 <http://example.org/movie/Manhattan>"; rdf:_1 <http://example.org/movie/Annie%20Hall>.]. <http://example.org/director/<Wong%20Kar-wai> ex:directed [a rdf:Bag; rdf:_1 <http://example.org/movie/2046>"; rdf:_2 <http://example.org/movie/In%20the%20Mood%20for%20Love>.].

3 Language description

3.1 Mapping Logical Sources to RDF with Triples Maps

3.1.1 xR2RML Triples Map

A triples map specifies a rule for translating data elements of a logical source to zero or more RDF triples. The RDF triples generated from one data element (row, document, set of XML elements, etc) in the logical source all share the same subject.

An xR2RML triples map extends R2RML triples map by referencing a logical source (next section) instead of a logical table. An xR2RML triples map is represented by a resource that references the following resources:

- It must have exactly one **xrr:logicalSource** property. Its value is a logical source that specifies a query result to be mapped to triples.
- It must have exactly one subject map that specifies how to generate a subject for each data element of the logical source (row, document, set of XML elements, etc). A subject map may be specified in two ways:
  - using the rr:subjectMap property, whose value must be the subject map, or
  - using the constant shortcut property rr:subject.
- It may have zero or more rr:predicateObjectMap properties, whose values must be predicate-object maps. They specify pairs of predicate maps and object maps that, together with the subjects generated by the subject map, may form one or more RDF triples for each data element.

3.1.2 Defining a Logical Source

An R2RML logical table is a data set on which the triples map applies: this may be a relational table, SQL view, or the result of any valid SQL query. xR2RML, on the other hand, intends to cope with a wide, non-limited scope of data sources, both relational and non-relational.

While relational databases have clearly identified commonalities (row-based data model, ANSI SQL compatibility), non-relational databases show much more heterogeneity. NoSQL, in particular, is refers to a large variety of systems having specific data models (key-value, document, column, graph), and access methods (APIs, query languages…). Consequently, unless a standard NoSQL query language eventually arises, xR2RML must be agile enough to cope with various query languages and protocols in order to apply to a significant subset of non-relational players.

xR2RML defines several extensions to describe an input database:

A **logical source** (property **xrr:logicalSource**) extends the R2RML concept of logical table (property rr:logicalTable) in the case of non-relational databases. A logical source is the result of a query applied to the input database, to be mapped to RDF triples. A logical source is either an **xR2RML base table** or an **xR2RML view**.

An **xR2RML base table** is a logical source containing data from a table in the input database, in the context of databases where tables make sense (such as an extensible column store). A base table is represented by a resource that has exactly one **xrr:sourceName** property. Property **xrr:sourceName** extends R2RML property rr:tableName for non-relational sources.

An **xR2RML view** is a logical source whose content is the result of executing a query against the input database. It is represented by a resource that has exactly one

xrr:query property. The value of property xrr:query is a literal representing a valid expression with regards to the query language supported by the input database.

A logical source may have one property **xrr:format** that specifies the format of the data retrieved from the input database. Possible format values are: **xrr:Row**, **xrr:JSON**, and **xrr:XML**. If a logical source has no xrr:format property, its format defaults to xrr:Row (to ensure compatibility with R2RML).

A logical source may have one property **xrr:iterator** that specifies the iteration pattern to apply on data retrieved from the input database (see section 3.1.3).

Note: Format **xrr:Row** applies to any database returning data as sets of rows, each row consisting of a set of columns: relational database, CSV/TSV file, extensible column store, SPARQL result set (that can be seen as a table in which columns are named after the variables returned).

Remark: No property specifies the query language used to express queries. Defining a built-in set of query languages within xR2RML would be limitative with regards to NoSQL systems in which new query languages may come up. Therefore, an xR2RML processor should allow for the use of various query languages in a flexible manner. If needed though, similarly to the R2RML rr:sqlVersion property, a property may be considered in a future version of xR2RML to explicit the query language.

xR2RML logical source and R2RML logical table definitions may equally be used in the case of a relational database. Examples:

R2RML logical table	Equivalent xR2RML logical source

[] rr:logicalTable [
  rr:tableName "SOME_TABLE".
]
	[] xrr:logicalSource [
	  xrr:sourceName "SOME_TABLE";
	  xrr:format xrr:Row. # optional
	]

[] rr:logicalTable [
  rr:sqlQuery
  "SELECT NAME, DATE FROM MOVIES".
]
	[] rr:logicalTable [
	  xrr:query
	  "SELECT NAME, DATE FROM MOVIES".
	  xrr:format xrr:Row. # optional
	]

The table below shows examples of xR2RML logical source definition with different flavors of input databases.

Type of database	Logical source definition

Relational database	[] rr:logicalTable [
	  rr:sqlQuery """
	  SELECT TITLE FROM MOVIES
	  WHERE YEAR > 1980
	  ORDER BY YEAR
	  LIMIT 10""";
	];

XML file. The file URL is passed to the xR2RML processor, e.g. http://example.org/movies.xml.	[] xrr:logicalSource [xrr:format xrr:XML;

An iterator defines the pattern of XML elements to iterate on.	xrr:iterator "//movie";];
REST-based web service returning an XML stream based on parameters passed in the HTTP GET query string. The service URL is passed to the xR2RML processor e.g. http://example.org/service, while the query string is provided by the xrr:query property.	[] xrr:logicalSource [xrr:query "?minYear=1980&limit=10"; xrr:format xrr:XML; xrr:iterator "//movie";];
XML database supporting XQuery.	[] xrr:logicalSource [xrr:query """for $i in //movies/movie where $i/year gt 1980 order by $i/@title return $i/@title"""; xrr:format xrr:XML; xrr:iterator "//movie";];
JSON file. The file URL is passed to the xR2RML processor, e.g. http://example.org/movies.json. An iterator defines the pattern to iterate on.	[] xrr:logicalSource [xrr:format xrr:JSON; xrr:iterator "$.movies.*";];
MongoDB database (document store), search for documents in collection "movies".	[] xrr:logicalSource [xrr:query '"db.movies.find({"year":{$gt: 1980}})'"; xrr:format xrr:JSON;];
Cassandra (extensible column store) using Cassandra Query Language (CQL)	[] xrr:logicalSource [xrr:query """SELECT TITLE, YEAR FROM Movies WHERE YEAR > 1980 LIMIT 10"""; xrr:format xrr:Row;];
AllegroGraph (RDF graph store) using SPARQL. The xrr:Row format is applied to a SPARQL result set: the result set can be seen as a table in which columns are named after variable names.	[] xrr:logicalSource [xrr:query """select ?title ?year where { ?movie a ex:Movie; ex:name ? title; ex:year ?year. } filter (?year > "1980"^^xsd:integer) order by ?year limit 10"""; xrr:format xrr:Row;];

3.1.3 xR2RML Triples Map Iteration Model

In R2RML, the row-based iteration implicitly occurs on a set of rows read from a logical table. xR2RML applies this principle to other row-based systems such as CSV/TSV files and extensible column store, but also SPARQL result sets as already underlined. In the context of non row-based databases, the iteration implicitly occurs on each result of a result set, whatever the form of such result. Typically, a document-based iteration applies to a set of JSON documents retrieved from a NoSQL document store, or a set of XML documents retrieved from an XML database. In the case of data sources that do not provide iterators, for instance a REST web service returning an XML stream, then a single iteration occurs on the whole XML document retrieved at once.

Nevertheless, some specific iteration needs may not be fulfilled by this extension. For instance, a document store returns JSON documents. However it may be needed to iterate on multiple entries of the document. This is illustrated by the following example.

The following JSON document is retrieved by a request to a MongoDB document store, it describes movie directors and respective movies.

```
{ "directors": [
   { "name": "Wong Kar-wai",
     "movies": ["2046", "In the Mood for Love"] },
   { "name": "Woody Allen",
     "movies": ["Manhattan", "Annie Hall"] }
]}
```

The following mapping graph is used:

```
<#Directors>
   xrr:logicalSource [
        xrr:query "db.movies.find( { directors: { $exists: true} } )";
        xrr:format xrr:JSON;
   ];
   rr:subjectMap [
      rr:template "http://example.org/director/{$.name}";
   ];
   rr:predicateObjectMap [
        rr:predicate ex:directed;
        rr:objectMap [ xrr:reference "$.directors.*.movies.*"; ];
   ].
```

In this mapping, the subject map returns two terms (one per director) while the object map returns four terms (one per movie in the document). Consequently, triples are generated that mix up all directors and movies:

```
<http://example.org/director/Woody%20Allen> ex:directed "Manhattan".
<http://example.org/director/Woody%20Allen> ex:directed "Annie Hall".
<http://example.org/director/Woody%20Allen> ex:directed "2046".
<http://example.org/director/Woody%20Allen> ex:directed "In the Mood for Love".
<http://example.org/director/Wong%20Kar-wai> ex:directed "Manhattan".
<http://example.org/director/Wong%20Kar-wai> ex:directed "Annie Hall".
<http://example.org/director/Wong%20Kar-wai> ex:directed "2046".
<http://example.org/director/Wong%20Kar-wai> ex:directed "In the Mood for Love".
```

Therefore:

An **iterator** is defined within a logical source by means of the **xrr:iterator** property. It specifies the iteration pattern to apply to data retrieved from the input database.

The value of an xrr:iterator property is a valid path expression written using a syntax that depends on the logical source data format, as follows:

- Format xrr:Row: the iterator is either omitted or empty;
- Format xrr:XML: the iterator value is an XPath expression;

- Format xrr:JSON: the iterator value is an JSONPath expression.

With the xrr:iterator property, the previous example is modified as shown below:

```
<#Directors>
   xrr:logicalSource [
        xrr:query "db.movies.find( { directors: { $exists: true} } )";
        xrr:iterator "$.directors.*";
        xrr:format xrr:JSON;
   ];
   rr:subjectMap [
      rr:template "http://example.org/director/{$.name}";
   ];
   rr:predicateObjectMap [
        rr:predicate ex:directed;
        rr:objectMap [ xrr:reference "$.directors.*.movies.*"; ];
   ].
```

The xrr:iterator property indicates that, within the document retrieved, the triples map iteration should be performed on each director element rather than on the whole document, thus producing the expected results:

```
<http://example.org/director/Woody%20Allen> ex:directed "Manhattan".
<http://example.org/director/Woody%20Allen> ex:directed "Annie Hall".
<http://example.org/director/Wong%20Kar-wai> ex:directed "2046".
<http://example.org/director/Wong%20Kar-wai> ex:directed "In the Mood for Love".
```

3.2 Creating RDF terms with Term Maps

3.2.1 xR2RML Term Maps

R2RML defines a term map as a function that generates RDF terms from a logical table row.
A term map is either a subject map, predicate map, object map or graph map.
A term map must be exactly one of the following:

- a constant-valued term map (property rr:constant)
- a column-valued term map (property rr:column)
- a template-valued term map (property rr:template).

R2RML treats all values from the input database as literals expressed in built-in data types. To deal with structured values such as collections or key-value associations, xR2RML term maps extend R2RML term maps so that structured values can be parsed, and data elements within structured values can be selected to build RDF terms. xR2RML extensions are described in this section.

3.2.1.1 Constant-, Column-, Reference- and Template-valued Term Maps

R2RML properties rr:column and rr:template reference columns of a logical table. xR2RML not only references columns but also any data element within structured values. Whereas the rr:template property name is generic, the rr:column property name specifically refers to the relational column concept. To avoid confusion, xR2RML extends rr:column with property **xrr:reference** to allow for references to data elements in non-relational databases. This leads to the following definition of an xR2RML term map. xR2RML changes to R2RML are highlighted.

A constant-valued term map is represented by a resource that has exactly one rr:constant property. A constant-valued term map always generates the same RDF term.

A column-valued term map has exactly one rr:column property. The value of the rr:column property is a valid column name.

A reference-valued term map has exactly one xrr:reference property. The value of the xrr:reference property is a valid reference to a data element.

A template-valued term map has exactly one rr:template property. The value of the rr:template property is a valid string template. A string template is a format string used to build strings from multiple components. It uses valid references to data elements by enclosing them in curly braces ("{" and "}").

3.2.1.2 Path expression syntax for data element references

Properties xrr:reference and rr:template use **data element references** to select data elements from a logical source. Data element references are valid path expressions written in a syntax that depends on the logical source data format, as follows:

- Format xrr:Row: a reference is a column name in case of relational database, CSV data source or extensible column store, and a variable name in a SPARQL result set.
- Format xrr:XML: the reference is an XPath expression;
- Format xrr:JSON: the reference is a JSONPath expression.

3.2.1.3 Term Types of Term Maps

RDF terms generated by a term map have a term type (rr:termType) that may be one of the three R2RML term types: rr:Literal, rr:BlankNode or rr:IRI.

xR2RML extends the rr:termType property with four new values, hereafter referred to as **RDF collection or container term types**:

- A term map with **xrr:RdfList** as value of property rr:termType generates an RDF collection of type rdf:List;
- A term map with **xrr:RdfSeq**: as value of property rr:termType generates an RDF container of type rdf:Seq;
- A term map with **xrr:RdfBag**: as value of property rr:termType generates an RDF container of type rdf:Bag;
- A term map with **xrr:RdfAlt**: as value of property rr:termType generates an RDF container of type rdf:Alt.

3.2.1.4 Nested Term Maps

Structured data such as JSON or XML documents commonly have more than one level of nesting, resulting in tree-like values that may need to be parsed in depth to nest RDF collections and containers, e.g. to build an RDF collection of RDF collections.

An xR2RML term map may have an **xrr:nestedTermMap** property, whose range is the **xrr:NestedTermMap** class.
In a column-valued or reference-valued term map, the xrr:nestedTermMap property describes how to translate input values referenced by the rr:column or xrr:reference

property into RDF terms.

In a template-valued term map, the xrr:nestedTermMap property describes how to translate values produced by applying the template string to input values into RDF terms.

A **nested term map**, it may have the properties below:

- **xrr:reference** bears the same semantics as in the context of a term map. Its object is a valid path expression (possibly a mixed-syntax path). Evaluation of the path expression is performed against values retrieved by the parent term map or parent nested term map.
- **rr:template** bears the same semantics as in the context of a term map. References enclosed in capturing curly braces are valid path expressions (possibly mixed-syntax paths), they apply to values retrieved in the parent term map or parent nested term map.
- **xrr:nestedTermMap** is used to recursively parse any depth of nested structured values;
- **rr:termType** bears the same semantics as in the context of a term map;
- **rr:language** bears the same semantics as defined in R2RML;
- **rr:datatype** bears the same semantics as defined in R2RML.

A **simple nested term map** is a nested term map that has no xrr:reference nor rr:template property. A simple nested term map is used to qualify terms of an RDF collection or container, i.e. assign them an optional term type, data type or language tag.

A **reference-valued nested term map** is a nested term map that has exactly one xrr:reference property.

A **template-valued nested term map** is a nested term map that has exactly one rr:template property.

xrr:nestedTermMap vs. rr:termType:

A nested term map *N* describes how to translate into RDF terms values produced by its parent *P*, *P* may be a term map or a nested term map.
If *P* has no rr:termType property, it simply returns values produced by *N*, therefore its term type is that of *N*.
If *P* has an rr:termType property with an RDF collection or container term type as value, then values produced by *N* will be assembled in an RDF collection or container.
Lastly, *P* should not have an rr:termType property with an R2RML term type (literal, blank node, IRI). Thus:

If a term map or nested term map has an xrr:nestedTermMap property, then it should have either no rr:termType property or an rr:termType property with an RDF collection or container term type. Formally:

?P is an rr:TermMap or xrr:NestedTermMap.
?P xrr:nestedTermMap ?N.
?P rr:termType ?tt.
⇒ ?tt is one of xrr:RdfList, xrr:RdfSeq, xrr:RdfBag or xrr:RdfAlt

A term map or nested term map with an RDF collection or container term type and no xrr:nestedTermMap property is assumed to have a default xrr:nestedTermMap property defined as follows:
- If the parent term map or nested term map is reference-valued:

```
        xrr:nestedTermMap [ rr:termType rr:Literal ];
- If the parent term map or nested term map is template-valued:
        xrr:nestedTermMap [ rr:termType rr:IRI ];
```

Finally, as defined in R2RML, properties rr:language and rr:datatype apply when generating literals only:

> A term map or nested term map may have an rr:language or rr:datatype property only if its term type is rr:Literal (either stated by property rr:termType or inferred as a default value).

Nested term maps are exemplified in section 3.2.3.

3.2.2 Referencing data elements

3.2.2.1 Referencing simple data elements

The table below exemplifies the use of properties rr:column, xrr:reference and rr:template to reference simple data elements (i.e. non-structured values) from the logical source.

Logical source	Term map
Relational database: either rr:column or xrr:reference can be used to name a column.	[] rr:column "NAME". [] xrr:reference "NAME". [] rr:template "{NAME}".
Extensible column store: properties xrr:reference and rr:template reference data elements by column names.	[] xrr:reference "NAME". [] rr:template "{NAME}".
XML database supporting: properties xrr:reference and rr:template reference data elements by XPath expressions.	[] xrr:reference "//name". [] rr:template "{//name }".
NoSQL document store: xrr:reference and rr:template reference data elements using a valid JSONPath expression.	[] xrr:reference "$.name". [] rr:template "{$.name }".
RDF graph store: xrr:reference and rr:template reference data elements by name of variable returned in the SPARQL result set.	[] xrr:reference "?name". [] rr:template "{?name }".

Remark: If a term map references a structured value but does not parse it using a nested term map, then generated RDF terms are string literals containing a simple serialization of structured values. Example:

Input data	{ "person": { "FirstName":"John", "LastName":"Smith" } }
Term map	[] rr:objectMap [xrr:reference "$.person";];
Generated RDF term	The structured value matching the JSONPath expression "$.person" is returned as a string literal in quotes:

	'{ "FirstName":"John", "LastName":"Smith" }'

3.2.2.2 Referencing data elements with mixed data formats

The syntax of data retrieved from a logical source is specified using the xrr:format property of a logical source. In some use cases though, database are commonly used to store values written in a data format that they cannot interpret. For instance, an application designer may choose to embed JSON, CSV, or XML values in the cells of a relational table, for performance concerns or application design constraints.

To reference data elements within such mixed contents, xR2RML allows a term map to reference data elements with **mixed-syntax paths**:

> Properties xrr:reference and rr:template may use **mixed-syntax paths** to reference data elements across data in different formats. A mixed-syntax path consists of the concatenation of several path expressions separated by the slash '/' character. Each individual path is enclosed in a **syntax path constructor** naming the path syntax explicitly. Existing constructors are:
> - **Column**(column-name): applies to row/column databases such as relational database and extensible column-store.
> - **CSV**(column-name), **TSV**(column-name): applies to data formatted as comma-separated or tab-separated values. Column-name may be a 0-based column index, or an actual column name if a head line provides column names.
> - **JSONPath**(JSONPath-expression): applies to any data formatted in JSON.
> - **XPath**(XPath-expression): applies to any data formatted in XML.

Example:

Input data	Relational table with one column: Name { "FirstName":"John", "LastName":"Smith" }
Logical source definition and Term map	[] xrr:logicalSource [xrr:format xrr:Row; …]; … rr:objectMap [xrr:reference "Column(Name)/JSONPath($.FirstName)"; rr:language "en";];
Generated RDF term	"John"@en

From the example above, it can be noticed that (i) the leftmost syntax path constructor (Column) is consistent with the logical source data format (xrr:Row), and (ii) data elements referenced by mixed-syntax path "Column(Name)/JSONPath($.FirstName)" are formatted in JSON, corresponding to the rightmost syntax path constructor. More generally:

> The leftmost syntax path constructor of a mixed-syntax path must be consistent with the logical source data format.
> - Constructors Column() CSV() and TSV() apply with format xrr:Row,
> - Constructor XPath() applies with format xrr:XML,
> - Constructor JSONPath() applies with format xrr:JSON.

> The format of data retrieved by a mixed-syntax path is the format of the rightmost syntax path constructor.

3.2.2.3 Production of multiple RDF terms

In a row-based logical source, a column reference returns exactly one scalar value per triples map iteration step: the value of the cell identified by "column name" in the current row. Thus, an R2RML term map generates zero or one RDF term during each iteration step, ultimately a triples map generates zero or one triple during each iteration step.

Due to the tree-like nature or JSON and XML data formats, JSONPath and XPath expressions allow addressing not only literals but also structured values. Thus, using the xrr:reference or rr:template properties with a JSONPath or XPath expression may return multiple values during each triples map iteration step. Hence the introduction of the term map iteration.

> A **term map iteration** is a process that occurs in a term map during each triples-map iteration step. Thus a reference-valued or template-valued term map generates zero to any number of RDF terms during each triples-map iteration step.

Examples:

Input data retrieved in one triples-map iteration step	{
	"FirstNames":
	["John", "Albert"],
	"LastName": "Smith"
	}
		<person>
		<FirstNames>
		<item>John</item>
		<item>Albert</item>
		</FirstNames>
		<LastName>Smith</LastName>
		</person>

Term map	[] rr:objectMap [xrr:reference "$.FirstNames.*";];	[] rr:objectMap [xrr:reference "/person/FirstNames/item";];
Generated RDF terms	"John" "Albert"	"John" "Albert"

The term map iteration applies identically in the context of mixed-syntax paths. Example:

Input data	<person> <name>John Smith</name> <items>[1,2,3]</items> </person> XML element "items" contains a value expressed as a JSON array.
Term map	[] xrr:logicalSource [xrr:format xrr:XML; ...] … rr:objectMap [xrr:reference "XPath(/person/items)/JSONPath($.*)"; rr:datatype xsd:integer;]

	The last expression of the mixed-syntax path, "JSONPath($.*)", indicates that (i) value "[1,2,3]" is formatted in JSON syntax, and (ii) it must be parsed as such using the "$.*" JSONPath expression.
Generated RDF terms	1^^xsd:integer 2^^xsd:integer 3^^xsd:integer

A template-valued term map may reference several data elements from the logical source, captured by curly braces ('{' and '}'). If at least one of the data elements referenced in a template string produces several terms, then the following applies:

> A template-valued term map produces RDF terms by performing a Cartesian product between all values produced by all data elements referenced in the template.

Example:

Input data	{
	"FirstNames": '["John", "Albert"]',
	"LastName": "Smith"
	}

Term map	[] xrr:logicalSource [xrr:format xrr:JSON; …]; rr:subjectMap [rr:template "http://example.org/{$.FirstNames.*}/{$.LastName}";] ;
Generated RDF terms	The template performs a Cartesian product between "Smith" and ["John", "Albert"], resulting in two terms: <http://example.org/John/Smith> <http://example.org/Albert/Smith>

Finally, below we define the behavior of a triples map in which one or several term maps generate multiple RDF terms during a single triples map iteration step:

> During each iteration of an xR2RML triples map, triples are generated as the Cartesian product between RDF terms produced by the subject map and each predicate-object map. Predicate-object couples result of the Cartesian product between RDF terms produced by each predicate and object map.
> Note that a graph map may also produce multiple terms, in which case triples are produced simultaneously in several target graphs.

In the example below, during one triples map iteration step, the subject map produces two RDF terms <http://example.org/company/Dell> and <http://example.org/company/Asus>, while the object map produces two literals "Laptop" and "Desktop". A Cartesian product between the two subjects and the two objects results in the production of four triples:

Input data: one row retrieved from a relational table, values are formatted in	

cos	products
["Dell", "Asus"]	<list> <product>Laptop</product>

JSON in column "cos", and XML in column "products"	<product>Desktop</product> </list>
Mapping graph	[] xrr:logicalSource [...]; rr:subjectMap [rr:template "http://example.org/{Column(cos)/JSONPath($.*)}";]; rr:predicateObjectMap [rr:predicate ex:produces; rr:objectMap [xrr:reference "Column(products)/XPath(/list/*)";];];
Generated triples	<http://example.org/Dell> ex:produces "Laptop". <http://example.org/Dell> ex:produces "Desktop". <http://example.org/Asus> ex:produces "Laptop". <http://example.org/Asus> ex:produces "Desktop".

3.2.2.4 Production of RDF collections or containers

A term map with an RDF collection or container as term type generates one RDF term during each triples map iteration step. The elements of the collection or container are the RDF terms produced by the term map, whether using property rr:column, xrr:reference or rr:template.
In the example below, the triples map generates one triple per iteration step. The object of this triple is an RDF bag:

Input data: JSON document retrieved in a single iteration step	<company "name"="Dell"> <products> <product>Laptop</product> <product>Desktop</product> </products> </company>
Mapping graph	[] xrr:logicalSource [xrr:format xrr:XML; ...]; rr:subjectMap [rr:template "http://example.org/{/company/@name}";]; rr:predicateObjectMap [rr:predicate ex:builds; rr:objectMap [xrr:reference "//company/products/*"; rr:termType xrr:RdfBag;];];
Generated triples	<http://example.org/Dell> ex:builds [a rdf:Bag; rdf:_1 "Laptop"; rdf:_2 "Desktop" .] .

Unlike RDF terms of type IRI or blank node, RDF terms of type RDF collection or container cannot be used as subject or predicate of an RDF triple, nor as a graph IRI. Consequently:

> A term map with term type xrr:RdfList, xrr:RdfSeq, xrr:RdfBag or xrr:RdfAlt is an object map (hence it cannot be a subject map, predicate map nor graph map).
> Formally:
> ?X an rr:TermMap.
> ?X rr:termType ?tt.
> ?tt is one of xrr:RdfList, xrr:RdfSeq, xrr:RdfBag or xrr:RdfAlt
> ⇒ ?X an rr:ObjectMap.

A nested term map (property xrr:nestedTermMap) can be used to specify a term type, language tag or data type of members of an RDF collection or container. The example below illustrates the usage of a nested term map to generate an RDF collection of elements typed as IRIs (first example), and an RDF sequence of data-typed literals (second example):

Input data	{ "key1": ["url1", "url2"] }	{ "key1": [10, 20] }

Term map	[] rr:objectMap [
	  xrr:reference "$.key1.*";
	  rr:termType xrr:RdfList;
	  xrr:nestedTermMap [
	  rr:termType rr:IRI;
	  ];
	];
		[] rr:objectMap [
		  xrr:reference "$.key1.*";
		  rr:termType xrr:RdfSeq;
		  xrr:nestedTermMap [
		  rr:termType rr:Literal;
		  rr:datatype xsd:integer;
		  ];
		];

Generated RDF terms	In Turtle abbreviated notation:
	(<url1> <url2>)
		[a rdf:Seq;
		  rdf:_1 10^^xsd:integer;
		  rdf:_2 20^^xsd:integer.
		];

In a template-valued term map, the xrr:nestedTermMap property applies to values resulting from the application of the template string to the input values. In the first example below, term type rr:IRI applies to the result of the template string. The same principle applies in the second example with term type rr:Literal and datatype xsd:string.

Input data	{
	"FirstNames": '["John", "Albert"]',
	"LastName": "Smith"
	}
		{
		"FirstNames": '["John", "Albert"]',
		"LastName": "Smith"
		}

Term map	[] rr:objectMap [rr:template "http://example.org/ {$.FirstNames.*}/{$.LastName}"; rr:termType xrr:RdfList; xrr:nestedTermMap [rr:termType rr:IRI;];];	[] rr:objectMap [rr:template "{$.FirstNames.*} {$.LastName}"; rr:termType xrr:RdfList; xrr:nestedTermMap [rr:termType rr:Literal; rr:datatype xsd:string;];];
Generated RDF terms	(<http://example.org/John/Smith> <http://example.org/Albert/Smith>)	("John Smith"^^xsd:string "Albert Smith"^^xsd:string)

3.2.3 Parsing nested structured values

The example below illustrates the use of a nested term map to (i) parse nested structured values ("teams" are collections of "team" elements, which are collections of "member" elements) and (ii) translate those nested structured values into RDF terms of class rdf:List.

Input data	<teams> <team> <member>John</member> <member>Paul</member> </team> <team> <member>Cathy</member> <member>Ed</member> </team> </teams>
Term map	[] rr:objectMap [xrr:reference "/teams/team"; xrr:nestedTermMap [xrr:reference "/member"; rr:termType xrr:RdfList;];]; The first xrr:reference property ("/teams/team") selects "team" elements from the XML input, each "team" element being the root of an XML tree whose descendants are "member" elements. The second xrr:reference property ("/member"), within the xrr:nestedTermMap property, applies to results of the parent reference expression. Thus, the xrr:RdfList term type successively applies to "member" elements of the first team, then to "member" elements of the second team. Finally the term map generates two RDF collections, one per team element.
Generated RDF terms	("John" "Paul") ("Cathy" "Ed")

The subsequent example generates one RDF sequence of nested RDF collections. Elements of the inner RDF collections are typed as rr:Literal and assigned a language tag using a second nested xrr:nestedTermMap property.

Input data	{ "teams": [["John", "Paul"] , ["Cathy", "Ed"]] }
Term map	[] rr:objectMap [xrr:reference "$.teams.*"; rr:termType xrr:RdfSeq; # represent "teams" as an rdf:Seq # Describe the elements of the RDF sequence xrr:nestedTermMap [rr:template "Player {$.*}"; rr:termType xrr:RdfList; # represent each team as an rdf:List # Type members of each team as literals with language "en"

	xrr:nestedTermMap [rr:termType rr:Literal; rr:language "en";];];];
Generated RDF terms	[a rdf:Seq; rdf:_1 ("Player John"@en "Player Paul"@en); rdf:_2 ("Player Cathy"@en "Player Ed"@en);]

As already mentioned, in a template-valued term map, property xrr:nestedTermMap applies to values resulting from the application of the template string to input values. Thus, defining a nested term map in a template-valued term map suggests that the template produces a valid expression with regards to the current data format, that, in turn, is interpreted against a path expression provided by an xrr:reference or rr:template property.

For instance, applying the template string:
'\{ "first": "{FirstNames}", "last": "{LastName}" \}'
would produce a string formatted as a JSON dictionary, like:
{ "first": "John", "last": "Smith" }

This use case is illustrated in the example below:

Input data	{
	"FirstNames": '["John", "Albert"]',
	"LastName": "Smith"
	}

Term map	[] rr:objectMap [rr:template '\{ "first": "{$.FirstNames.*}", "last": "{$.LastName}" \}'; xrr:nestedTermMap [xrr:reference "$.*"; rr:termType xrr:RdfList;];]
Generated RDF terms	("John" "Smith") ("Albert" "Smith") Two values are generated by applying the template string, those values are formatted as JSON arrays: { "first": "John", "last": "Smith" } { "first": "Albert", "last": "Smith" } The xrr:nestedTermMap property instructs to parse those values using the JSONPath expression "$.*" (property xrr:reference), and generates an RDF collection (rdf:List) for each of them.

Note: this use case may seem rather awkward and probably of little use, but insofar as it is consistent with the xR2RML language definition, we think it should be considered as valid.

3.2.4 Multiple Mapping Strategies

The flexibility offered by nested term maps allows the same mapping to be written using various strategies: path expressions of properties xrr:reference and rr:template can be split in several levels of term map and nested term map.

For instance, both term maps below produce equivalent results. In the first case (left), the JSONPath expression ($.teams.*.*) retrieves team members at once. In the second case (right), teams are retrieved first ($.teams.*), then the xrr:nestedTermMap property runs a second JSONPath evaluation to retrieve and datatype team members.

Input data	{ "teams": [["John", "Paul"] , ["Cathy", "Ed"]] }	
Term maps	[] xrr:logicalSource [xrr:format xrr:XML; …]; rr:objectMap [xrr:reference "$.teams.*.*"; rr:datatype xsd:string;];	[] xrr:logicalSource [xrr:format xrr:XML; …]; rr:objectMap [xrr:reference "$.teams"; xrr:nestedTermMap [xrr:reference "$.*"; rr:datatype xsd:string;];];
Generated RDF terms	"John"^^xsd:string "Paul"^^xsd:string "Cathy"^^xsd:string "Ed"^^xsd:string	

It is likely that the first case will be more efficient as only one XPath evaluation is performed, whereas in the second case two XPath evaluations are performed in sequence.

Similarly, the example below shows how a mixed-syntax path can be split into a term map and a nested term map:

```
[] xrr:logicalSource [ xrr:format xrr:Row; … ];
  rr:objectMap [
    xrr:reference
      "Column(col)/XPath(/person/name)";
    rr:datatype xsd:string;
  ];
```

```
[] xrr:logicalSource
     [ xrr:format xrr:Row; … ];
  rr:objectMap [
    rr:column "col";
    xrr:nestedTermMap [
      xrr:reference "XPath(/person/name)";
      rr:datatype xsd:string;
    ];
  ];
```

Both mappings are likely to be equally efficient, as both evaluations (column selection and XPath expression evaluation) need to be done anyway.

3.2.5 Default Term Types

This section is an adaptation of section 7.4 (http://www.w3.org/TR/r2rml/#termtype) of the R2RML specification. xR2RML additions to R2RML are highlighted.

If the term map has an optional rr:termType property then its term type is the value of that property. The value MUST be one of the following options:

- If the term map is a subject map: rr:IRI or rr:BlankNode
- If the term map is a predicate map: rr:IRI
- If the term map is an object map: rr:IRI, rr:BlankNode, rr:Literal, rdf:List, rdf:Seq, rdf:Bag, rdf:Alt.
- If the term map is a graph map: rr:IRI.

If the term map does not have an rr:termType property, then its term type is:

- rr:Literal, if it is an object map and at least one of the following conditions is true:
  - It is a column-based term map.
  - It has an rr:language property (and thus a specified language tag).
  - It has an rr:datatype property (and thus a specified datatype).
  - It does not have an rr:language property and it has a nested term map that has an rr:language property.
  - It does not have an rr:datatype property and it has a nested term map that has an rr:datatype property.
- the term type of the value of its nested term map.
- rr:IRI, otherwise.

A corollary of this definition is that the xrr:nestedTermMap property may be used in a subject map, predicate map or graph map only if it produces IRIs. Consequently:

> A term map with an xrr:nestedTermMap property may be a subject map or graph map only if (i) it does not have an rr:termType property and (ii) its nested term map has an rr:termType property with value rr:IRI or rr:BlankNode.
> A term map with an xrr:nestedTermMap property may be a predicate map only if (i) it does not have an rr:termType property and (ii) its nested term map property has an rr:termType property with value rr:IRI.

3.3 Reference relationships between logical sources

The following definitions are an adaptation of R2RML specification section 8 (http://www.w3.org/TR/r2rml/#foreign-key). xR2RML additions to R2RML are highlighted.

> A referencing object map allows using the subjects of another triples map as the objects generated by a predicate-object map. Since both triples maps may be based on different logical sources, this may require a join between the logical sources.
>
> A referencing object map resource has exactly one rr:parentTriplesMap property (its value is a triples map), and optional rr:joinCondition properties. A join condition has exactly one rr:child property and one rr:parent property. The rr:child property references the join condition's child data element, the rr:parent property references the join condition's parent data element. Data element references are valid path expressions with regards to the logical source data format, possibly using mixed-syntax paths.
>
> A referencing object map may have an rr:termType property with an RDF collection or container term type (see further details in §3.3.2).
>
> The child query of a referencing object map is the query or source name of the logical source of the triples map containing the referencing object map.
>
> The parent query of a referencing object map is the query or source name of the logical source of the referencing object map's parent triples map.

Properties rr:child and rr:parent use valid path expressions to reference data elements. As described in §3.2.2.3, such path expressions may produce multiple terms. Consequently, the equivalent joint query of a referencing object map must take into account the fact that child and parent references be multi-valued. More precisely, a join between two multi-valued references should be satisfied if at least one data element of the first reference matches one data element of the second reference.

The joint query of a referencing object map is defined below using SQL syntax (SELECT... FROM... AS... WHERE) and first order logic for the description of WHERE conditions:

> If a referencing object map has no join condition, its joint query is:
>
> SELECT * FROM (*{child-query}*) AS tmp
>
> If a referencing object map has at least one join condition, its joint query is:
>
> SELECT * FROM ({child-query}) AS child, ({parent-query}) AS parent
> WHERE ∃c ∈ child.{child-ref1}, ∃p ∈ parent.{parent-ref1}, c = p
> AND ∃c ∈ child.{child-ref2}, ∃p ∈ parent.{parent-ref2}, c = p
> AND ...
>
> where {child-ref1} and {parent-ref1} are the child reference and parent reference of the first join condition, and so on.

Note: when applied to a relational database, in which child and parent references are single-valued, this definition can be simplified into the R2RML joint query definition:

```
SELECT * FROM ({child-query}) AS child, ({parent-query}) AS parent
WHERE child.{child-ref1} = parent.{parent-ref1}
AND   child.{child-ref2} = parent.{parent-ref2}
AND …
```

3.3.1 Reference relationship with structured values

The relational database example below models the relation between medical doctors and the studies for which they are investigators. Column "Doctor.studies" contains JSON arrays of which elements are references (similar to foreign keys) to column "Study.study_id".

<table>
<tr><td>Input data</td><td>Table Study
<table><tr><th>study_id</th><th>study_name</th></tr><tr><td>1</td><td>study1</td></tr><tr><td>2</td><td>study2</td></tr><tr><td>3</td><td>study3</td></tr></table>
Table Doctor
<table><tr><th>doc_id</th><th>doc_name</th><th>studies</th></tr><tr><td>1</td><td>D1</td><td>[1,2]</td></tr><tr><td>2</td><td>D2</td><td>[3]</td></tr></table></td></tr>
<tr><td>Mapping graph</td><td><#Study>
rr:logicalTable [rr:tableName "Study"];
rr:subjectMap [
rr:template "http://example.org/study/{study_name}";</td></tr>
</table>

<table>
<tr><td></td><td>

```
   ].

<#Doctor>
   rr:logicalTable [ rr:tableName "Doctor" ];
   rr:subjectMap [
      rr:template "http://example.org/doc/{doc_name}";
   ];
   rr:predicateObjectMap [
      rr:predicate ex:investigator;
      rr:objectMap [
         rr:parentTriplesMap <#Study>;
         rr:joinCondition [
            rr:parent "study_id";
            rr:child "Column(studies)/JSONPath($.*)";
         ];
      ];
   ].
```

The rr:child property uses a mixed-syntax path specifying that the data retrieved is formatted in JSON, and that each element of this structured value is considered in the join operation.
</td></tr>
<tr><td>Generated triples</td><td>

```
<http://example.org/doc/D1>
      ex:investigator <http://example.org/study/study1> .
<http://example.org/doc/D1>
      ex:investigator <http://example.org/study/study2> .
<http://example.org/doc/D2>
      ex:investigator <http://example.org/study/study3> .
```

According to the equivalent joint query definition, the joint query is as follows ("child" and "parent" notations have been removed for readability):

```
SELECT * FROM Doctor, Study
WHERE ∃c ∈ eval(Doctor, Column(studies)/JSONPath($.*)),
      ∃p ∈ Study.study_id,
      c = p
```

where *eval(Doctor, Column(studies)/JSONPath($.*))* represents the evaluation of mixed-syntax path "Column(studies)/JSONPath($.*)" on table Doctor.

Since Study.study_id is single-valued, we can rewrite the query as:

```
SELECT * FROM Doctor, Study
WHERE ∃ c ∈ Doctor.Column(studies)/JSONPath($.*),
      c = Studies.study_id
```

The join query results in this table:

doc_id	doc_name	studies	study_id	study_name
1	D1	[1,2]	1	study1
1	D1	[1,2]	2	study2
2	D2	[3]	3	study3
</td></tr>
</table>

3.3.2 Generating RDF collection/container with a referencing object map

In R2RML, referencing object maps do not have an rr:termType property as they should only produce RDF terms of type rr:IRI. In xR2RML however, the result of a joint query may be translated into an RDF collection or container using property rr:termType. The rr:termType has a specific semantics here: it groups joint query results by subjects of the generated triples, i.e. by child reference, and renders all objects in the same grouping as an RDF collection or container.

> If a referencing object map has no rr:termType property, then its term type is rr:IRI (compliant with the R2RML definition).
>
> A referencing object map may have an rr:termType property with an RDF collection or container term type (xrr:RdfList, xrr:RdfSeq, xrr:RdfBag or xrr:RdfAlt). In that case, elements of the collection or container are necessarily of type rr:IRI.
>
> In a referencing object map with an RDF collection or container term type, results of the joint query are grouped by child value, i.e. by subjects of the triples map containing the referencing object map. The parent values of such formed groups (the objects of the triples map) are grouped in a single object of type RDF collection or container, as instructed by the rr:termType property.

In the example below the referencing object map has an rr:termType property with value xrr:RdfList:

Input data	JSON documents retrieved by the query in the <#Study> triples map: { "study_id":1, "study_name":"study1" } { "study_id":2, "study_name":"study2"} { "study_id":3, "study_name":"study3"} JSON documents retrieved by the query in the <#Doctor> triples map: { "doc_name":"D1", "studies": [1,2] } { "doc_name":"D2", "studies": [2,3] }
Mapping graph	Below, queries to retrieve Studies and Doctors are referred to as <*Study query*> and <*Doctor query*>. <#Doctor> xrr:logicalSource [xrr:format xrr:JSON; xrr:query "<*Study query*>";]; rr:subjectMap [rr:template "http://example.org/doc/{$.doc_name}";]. <#Study> xrr:logicalSource [xrr:format xrr:JSON; xrr:query "<*Doctor query*>";]; rr:subjectMap [rr:template "http://example.org/study/{$.study_name}";]; rr:predicateObjectMap [rr:predicate ex:hasInvestigator; rr:objectMap [

	rr:parentTriplesMap <#Doctor>; rr:joinCondition [rr:child "$.study_id"; rr:parent "$.studies.*";]; rr:termType xrr:RdfList;];].
Generated RDF triples	<http://example.org/study/study1> ex:hasInvestigator (<http://example.org/doc/D1>). <http://example.org/study/study2> ex:hasInvestigator (<http://example.org/doc/D1> <http://example.org/doc/D2>). <http://example.org/study/study3> ex:hasInvestigator (<http://example.org/doc/D2>). Explanation: according to the equivalent joint query definition, the joint query is as follows: SELECT * FROM (<*Study query*>) as child, (<*Doctor query*>) as parent WHERE ∃ p ∈ eval(parent, $.studies.*), p = eval(child, $.study_id) where *eval(parent, $.studies.*)* represents the evaluation of path "$.studies.*" on the result of the parent query, and *eval(child, $.study_id)* represents the evaluation of path "$.study_id" on the result of the child query. The equivalent joint query results in the following documents: { "study_id":1, "study_name":"study1", "doc_name":"D1", "studies": [1,2] } { "study_id":2, "study_name":"study2", "doc_name":"D1", "studies": [1,2] } { "study_id":2, "study_name":"study2", "doc_name":"D2", "studies": [2,3] } { "study_id":3, "study_name":"study3", "doc_name":"D2", "studies": [2,3] } Then, term type xrr:RdfList groups results by child reference, i.e. by "study_id".

3.3.3 Generating RDF collection/container with a referencing object map in the relational case

An interesting consequence of using the rr:termType in a referencing object map is the ability, in the case of a relational database with standard SQL values, to build an RDF collection or container reflecting a one-to-many relation. In the example below, foreign key Study.doctor relates each study to its investigator in a many-to-one relation (several studies may have the same investigator). Considered the other way round, it can be seen as a one-to-many relation (one doctor investigates several studies). The mapping graph describes the generation of each doctor along with the list of studies he/she investigates.

<table>
<tr><td>Input data</td><td>
Table Study
<table>
<tr><th>study_id</th><th>study_name</th><th>doctor</th></tr>
<tr><td>1</td><td>study1</td><td>1</td></tr>
<tr><td>2</td><td>study2</td><td>1</td></tr>
<tr><td>3</td><td>study3</td><td>2</td></tr>
</table>
Table Doctor
<table>
<tr><th>doc_id</th><th>doc_name</th></tr>
<tr><td>1</td><td>D1</td></tr>
<tr><td>2</td><td>D2</td></tr>
</table>
</td></tr>
<tr><td>Mapping graph</td><td>
<#Study>
rr:logicalTable [rr:tableName "Study"];
rr:subjectMap [
rr:template "http://example.org/study/{study_name}";
].

<#Doctor>
rr:logicalTable [rr:tableName "Doctor"];
rr:subjectMap [
rr:template "http://example.org/doc/{doc_name}";
].
rr:predicateObjectMap [
rr:predicate ex:investigator;
rr:objectMap [
rr:parentTriplesMap <#Study>;
rr:joinCondition [
rr:child "doc_id";
rr:parent "doctor;
];
rr:termType xrr:RdfList;
];
].
</td></tr>
<tr><td>Generated RDF triples</td><td>
<http://example.org/doc/D1> ex:investigator
(<http://example.org/study/study1>
<http://example.org/study/study2>) .
<http://example.org/doc/D2> ex:investigator
(<http://example.org/study/study3>) .

The equivalent joint query results in this table:
<table>
<tr><th>doc_id</th><th>doc_name</th><th>study_id</th><th>study_name</th><th>doctor</th></tr>
<tr><td>1</td><td>D1</td><td>1</td><td>study1</td><td>1</td></tr>
<tr><td>1</td><td>D1</td><td>2</td><td>study2</td><td>1</td></tr>
<tr><td>2</td><td>D2</td><td>3</td><td>study3</td><td>2</td></tr>
</table>
Results are grouped by child reference, i.e. by column "doc_id".
</td></tr>
</table>

4 References

[1] S. Das, S. Sundara, R. Cyganiak, R2RML: RDB to RDF Mapping Language, (2012).
[2] R. Hecht, S. Jablonski, NoSQL Evaluation: A Use Case Oriented Survey, in: Proceedings of the 2011 International Conference on Cloud and Service Computing, IEEE Computer Society, Washington, DC, USA, 2011: pp. 336–341.
[3] K.W. Ong, Y. Papakonstantinou, R. Vernoux, The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases (submitted), CoRR. abs/1405.3631 (2014).