Translation of Relational and Non-Relational Databases into RDF with xR2RML

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

Translation of Relational and Non-Relational Databases into RDF with xR2RML

Franck Michel[1], Loïc Djimenou[1], Catherine Faron-Zucker and Johan Montagnat[1]
[1] *Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, France*

Keywords: Linked Data, RDF, R2RML, NoSQL

Abstract: With the growing amount of data being continuously produced, it is crucial to come up with solutions to expose data from ever more heterogeneous databases (e.g. NoSQL systems) as linked data. In this paper we present xR2RML, a language designed to describe the mapping of various types of databases to RDF. xR2RML flexibly adapts to heterogeneous query languages and data models while remaining free from any specific language or syntax. It extends R2RML, the W3C recommendation for the mapping of relational databases to RDF, and relies on RML for the handling of various data representation formats. We analyse data models of several modern databases as well as the format in which query results are returned, and we show that xR2RML can translate any data element within such results into RDF, relying on existing languages such as XPath and JSONPath if needed. We illustrate some features of xR2RML such as the generation of RDF collections and containers, and the ability to deal with mixed content.

1 INTRODUCTION

The web of data is now emerging through the publication and interlinking of various open data sets in RDF. Initiatives such as the W3C Data Activity[1] and the Linking Open Data (LOD) project[2] aim at Web-scale data integration and processing, assuming that making heterogenous data available in a common machine-readable format should create opportunities for novel applications and services. Their success largely depends on the ability to reach data from the deep web (He et al., 2007), a part of the web content consisting of documents and databases hardly linked with other data sources and hardly indexed by standard search engines. Furthermore, the integration of heterogeneous data sources is a major challenge in several domains (Field et al., 2013). As data semantics is often poorly captured in database schemas, or encoded in application logics, data integration techniques have to capture and expose database semantics in an explicit and machine-readable manner.

The deep web keeps on growing as data is continuously being accumulated in ever more heterogeneous databases. In particular, NoSQL systems have gained a remarkable success during recent years. Driven by major web companies, they have been developed to meet requirements of web 2.0 services, that relational databases (RDB) could not achieve (flexible schema, high throughput, high availability, horizontal elasticity on commodity hardware). Thus, NoSQL systems should be considered as potential big contributors of the linked open data. Other types of databases have been developed over time, either for generic purpose or specific domains, such as XML databases (notably used in edition and digital humanities), object-oriented databases or directory-based databases.

Significant efforts have been invested in the definition of methods to translate various kinds of data sources into RDF. R2RML (Das et al., 2012), for instance, is the W3C recommendation to describe RDB-to-RDF mappings. RML extends R2RML for the integration of heterogeneous data formats (Dimou et al., 2014a), but it does not address the constraints that arise when dealing with different types of databases and query languages. In particular, to our knowledge, no method has been proposed yet to tackle NoSQL-to-RDF translation.

In this paper, we present xR2RML, a mapping language designed as an extension of R2RML and RML. Besides relational databases, xR2RML addresses the mapping of a large and extensible scope of non-relational databases to RDF. It is designed to flexibly adapt to various data models and query languages. xR2RML can translate data with mixed formats and generate RDF collections and containers. Our primary focus includes some NoSQL and XML native databases but the approach can equally apply

[1] http://www.w3.org/2013/data/
[2] http://linkeddata.org/

to other types of database such as object-oriented and directory-based databases.

In the rest of this section we draw a picture of other works related to the translation of various data sources to RDF, and we scope the objectives of xR2RML. Section 2 explores in more details the capabilities required to reach these goals. In section 3 we summarize the main characteristics of R2RML and RML, and in section 4 we describe xR2RML specific extensions. Section 5 presents a working implementation of the language, finally sections 6 and 7 discuss xR2RML applicability in different contexts and concludes by outlining some perspectives.

1.1 Related Works

Wrapper-based data integration systems like Garlic (Roth and Schwartz, 1997) and SQL/MED (Melton et al., 2002) generally have similar architectures: a global data model is described using specific modelling languages (e.g. Garlic's GDL), a query federation engine handles user queries expressed in terms of a global data model and determines a query plan, a per-data source wrapper implements a specific wrapper interface and performs the mapping with the data source schema. No guideline is provided as to how a wrapper should describe and implement the mapping.

The same global architecture holds in data integration systems based on semantic web technologies. Existing works focus on efficient query planning and distribution, such as FedX (Schwarte et al., 2011), Anapsid (Acosta et al., 2011) and KGRAM-DQP (Gaignard, 2013). The global data model is expressed by domain ontologies using common languages, e.g. RDFS or OWL. User queries, expressed in terms of the domain ontologies, are written in SPARQL. SPARQL is also used as the wrapper interface. Each data source wrapper is a SPARQL endpoint that performs the schema mapping with the source schema. Our work, as well as most related works listed below, focuses on the mapping step: the rationale is to standardize the schema mapping description, so that a mapping description be written once and applied with different wrapper implementations.

RDB-to-RDF mapping has been an active field of research during the last ten years (Spanos et al., 2012; Sequeda et al., 2011; Michel et al., 2014b). Several mapping methods and languages have been proposed over time, based either on the materialization of RDF data sets or on the SPARQL-based access to relational data. Published in 2012, R2RML, the W3C RDB-to-RDF mapping language recommendation, has reached a notable consensus[3].

Similarly, various solutions exist to map XML data to RDF. The XSPARQL (Bischof et al., 2012) query language combines XQuery and SPARQL for bidirectional transformations between XML and RDF. Several other solutions are based on the XSLT technology such as XML Scissor-lift (Fennell, 2014) that describes mapping rules in Schematron XML validation language, and AstroGrid-D (Breitling, 2009). SPARQL2XQuery (Bikakis et al., 2013) applies XML Schema to RDF/OWL translation rules.

Much work has already been accomplished regarding the translation of CSV, TSV and spreadsheets to RDF. Tools have been developed such as XLWrap (Langegger and Wöss, 2009) and RDF Refine[4]. The Linked CSV[5] format is a proposition to embed metadata in a CSV file, that make it easy to link on the Web and eventually to translate to RDF or JSON. However this approach assumes that CSV data be made compliant with the format in the first place, before it can be translated to RDF. The CSV on the Web W3C Working Group[6], created in 2014, intends to propose a recommendation for the description of and access to CSV data on the Web. In this context, RDF is one of the formats envisaged either to represent metadata about CSV data, or as a format to translate CSV data into.

Several tools are designed as frameworks for the integration of sources with heterogeneous data formats. XSPARQL, cited above, provides an R2RML-compliant extension. Thus it can simultaneously translate relational, XML and RDF data to XML or RDF. TARQL[7] is a SPARQL-based mapping language that can convert from RDF, CSV/TSV and JSON formats to RDF, but it does not focus on how the data is retrieved from different types of databases. Datalift (Scharffe et al., 2012) provides an integrated set of tools for the publication in RDF of raw structured data (RDB, CSV, XML) and the interlinking of resulting data sets.

RML (Dimou et al., 2014b; Dimou et al., 2014a) is an extension of R2RML that tackles the mapping of data sources with heterogeneous data formats such as CSV/TSV, XML or JSON. Most approaches create links between data sets after they were translated to RDF, e.g. using properties `rdfs:seeAlso` or `owl:sameAs`. This is sometimes not adequate as logical resources having different identifiers in different data sets cannot easily be reconciled. RML creates linked data sets at mapping time by enabling the si-

[3] http://www.w3.org/2001/sw/rdb2rdf/wiki/Implementations

[4] http://refine.deri.ie/

[5] http://jenit.github.io/linked-csv/

[6] http://www.w3.org/2013/csvw/wiki

[7] https://github.com/cygri/tarql/wiki/TARQL-Mapping-Language

multaneous mapping of multiple data sources, thus allowing for cross-references between resources defined in various data sources. However, RML does not investigate the constraints that arise when dealing with different types of databases. It proposes a solution to reference data elements within query results using expressive languages such as XPath and JSONPath. But it does not clearly distinguish between such languages and the actual query language of a database. In some cases they might be the same, e.g. XPath can be used to query an XML native database, and later on to reference data elements from query results. But in the general case, the query language and the language used to reference elements within query results must be dissociated, e.g. NoSQL document stores proprietary query languages, while results are JSON documents that can be evaluated against JSONPath expressions. Furthermore, RML explicitly refers to known evaluation languages (ql:JSONPath, ql:XPath). In this context, supporting a new evaluation language requires to change the mapping language definition. To achieve more flexibility, we believe that such characteristics should be implementation-dependent, leaving the mapping language free from any explicit dependency.

1.2 Objectives of this Work

The works presented in section 1.1 address various types of data sources. Some of them could be extended to new data sources by developing ad-hoc extensions, although they are generally not designed to easily support new data models and query languages. Only RML comes with this flexibility as its design aims at adapting to new data models. Our goal with xR2RML is to define a generic mapping language able to equally apply to most common relational and non-relational databases. We make a specific focus on NoSQL and XML native databases, and we argue that our work can be generalized to some other types of database, for instance object-oriented and directory (LDAP) databases. In section 2 we explore the capabilities required by xR2RML to reach these goals.

2 XR2RML LANGUAGE REQUIREMENTS

Different kinds of databases typically differ in several aspects: the query language used to retrieve data, the data model that underlies the data structures retrieved and the cross-data referencing scheme, if any. Below we explore in further details the capabilities that we want xR2RML to provide.

Query languages. The landscape of modern database systems shows a vast diversity of query languages. Relational databases generally support ANSI SQL, and most native XML databases support XPath and XQuery. By contrast, NoSQL is a catch-all term referring to very diverse systems (Hecht and Jablonski, 2011; Gajendran, 2013). They have heterogeneous access methods ranging from low-level APIs to expressive query languages. Despite several propositions of common query language (N1QL[8], UnQL[9], SQL++ (Ong et al., 2014), ArangoDB QL[10], CloudMdsQL (Kolev et al., 2014)), 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 in a transparent manner.

Data models. Similarly to the case of query languages, we observe a large heterogeneity in data models of modern databases. To describe their translation to RDF, a mapping language must be able to reference any data element from their data models. Below we list most common data models, we shortly analyse formats in which data is retrieved and figure out how a mapping language can reference data elements within retrieved data.

Relational databases comply with a row-based model in which column names uniquely reference cells in a row. NoSQL extensible column stores[11] also comply with the row-based model, with the difference that all rows do not necessarily share the same columns. For such systems, referencing data elements is simply achieved using column names. Other non-relational systems, such as XML native databases, NoSQL key-value stores, document stores or graph stores, have heterogeneous data models that can hardly be reduced to a row-based model:

- In databases relying on a specific data representation format like JSON (notably in NoSQL document stores) and XML, data is stored and retrieved as documents consisting of tree-like compound values. Referencing data elements within such documents can be achieved thanks to languages such as JSONPath and XPath.
- Object-oriented databases conventionally provide methods to serialize objects, typically as key-value associations: keys are attribute names while values are objects (composition or aggregation relationship), or compound values (collection, map, etc). Serialization is typically done in XML or JSON, thus here again

[8] http://www.couchbase.com/communities/n1ql
[9] http://unql.sqlite.org/index.html
[10] http://docs.arangodb.org/Aql/README.html
[11] aka. column family store, column-oriented store, etc.

we can apply XPath or JSONPath expressions.

- A directory data model is organised as a tree: each node has an identifier and a set of attributes represented as `name=value`. Each entry retrieved from an LDAP request is named using an LDAP path expression, e.g. `cn=Franck Michel,ou=cnrs,o=fr`. Referencing data elements within such entries can be simply achieved using attribute names.

- In graph databases, the abstract data model basically consists of nodes and edges. Query capabilities generally allow to retrieve either values matching certain patterns (like the SPARQL SELECT clause), or a set of nodes and edges representing a result graph (like the SPARQL CONSTRUCT clause). Whatever the type of result though, graph databases commonly provide APIs to manipulate query results. For instance a SPARQL SELECT result set has a row-based format: each row of a result set consists of columns typically named after query variable names. The Neo4J graph database provides a JDBC interface to process a query result, and its REST interface returns result graphs as JSON documents. Thus, although a graph may be a somehow complex data structure, query results can be fairly easy to manipulate using well-known formats: a row-column model, a serialization in JSON or some other representation syntax, etc.

Finally, the way a mapping language can reference data elements within query results depends more on the API capabilities than the data model itself. To be effective, xR2RML must transparently accept any type of data element reference expression. This includes a column name (applicable not only to row-based data models but also to any row-based query result), JSONPath, XPath or LDAP path expressions, etc. An xR2RML processing engine must be able to evaluate such expressions against query results, but the mapping language itself must remain free from any reference to specific expression syntaxes.

Collections. Many data models support the representation of collections: these can be sets, arrays or maps of all kinds (sorted or not sorted, with or without duplicates, etc.). Although the RDF data model supports such data structures, to the best of our knowledge, existing mapping languages do not allow for the production of RDF collections (`rdf:List`) nor RDF containers (`rdf:Bag`, `rdf:Seq`, `rdf:Alt`), except TARQL that is able to convert a JSON array into an `rdf:List`. In all other cases, structured values such as collections or key-value associations are flattened into multiple RDF triples. Listing 1 is an example XML collection consisting of two "movie" elements.

Its translation into two triples is illustrated in Listing 2. Assuming that the order of "movie" elements implicitly represents the chronological order in which movies were shot, triples in Listing 2 lose this information. Using an RDF sequence may be more appropriate in this case, as illustrated in Listing 3.

```
<director name="Woody Allen">
    <movie>Annie Hall</movie>
    <movie>Manhattan</movie>
</director>
```

Listing 1: Example of XML collection

```
<http://example.org/dir/Woody\%20Allen>
  ex:directed "Annie Hall".
<http://example.org/dir/Woody\%20Allen>
  ex:directed "Manhattan".
```

Listing 2: Translation to multiple RDF triples

```
<http://example.org/dir/Woody\%20Allen>
ex:movieList [ a rdf:Seq;
  rdf:_1 "Annie Hall";
  rdf:_2 "Manhattan" ].
```

Listing 3: Translation to an RDF sequence

Consequently, to map heterogeneous data to RDF while preserving concepts such as collections, bags, alternates or sequences, xR2RML must be able to map data elements to RDF collections and containers.

Cross-references. Cross-references are commonly implemented as foreign key constraints in relational data models, or aggregation and composition relationships in object-oriented models. Cross-referencing is even the primary goal of graph-based databases. More generally, it is possible to cross-reference logical entities in any type of database. For instance, a JSON document of a NoSQL document store may refer to another document by its identifier or any other field that identifies it uniquely, even if this is generally not recommended for the sake of performances.

A cross-referenced logical resource may be mapped alternatively as the subject or the object of triples. This may entail joint queries between tables or documents. Therefore, xR2RML must (i) allow a modular description so that the mapping of a logical resource can be written once and easily reused as a subject or an object, and (ii) allow the description of joint queries to retrieve cross-referenced logical resources.

Summary. Finally, we draw up the list of key capabilities expected from xR2RML as follows:

1. It enables to describe the mapping of various relational and non-relational databases to RDF.
2. It is flexible enough to allow for new databases, query languages and data models in an agile manner: supporting a new system, query language and/or data model only requires changes in the implementation (adaptor, plug-in, etc.), but no changes are required in

the mapping language itself.
3. It enables to generate RDF collections (`rdf:List`) or containers (`rdf:Seq`, `rdf:Bag`, `rdf:Alt`) from one-to-many relations modelled as compound values or as cross-references. RDF collections and containers can be nested.
4. It enables to perform joint queries following cross-references between logical resources, and it allows the modular reuse of mapping definitions.

Taken the other way round, data sources to be mapped to RDF using xR2RML need to fulfil some requirements entailed by xR2RML's capabilities:
1. The data source interface should provide a **declarative** query language. If not, it must be possible to fetch the whole data at once, like a CSV or XML file returned by a Web service.
2. There must exist technical means to parse query results and reference data elements: this ranges from simple column names to expressive languages like XMLPath.
3. In case of large data sets, the database interface should provide ways to iterate on query results, similarly to SQL cursors in RDBs.
Notice that the last two requirements are quite natural features of most decent database systems.

To help in the design of xR2RML we chose to leverage R2RML, a standard, well-adopted mapping language for relational databases. R2RML already provides some of the requirements listed above: modularity, management of cross-references, as well as rich features such as the ability to define target named graphs. To facilitate its understanding and adoption, xR2RML is designed as a backward compatible extension of R2RML. Besides, to address the mapping of heterogeneous data formats such as CSV/TSV, XML and JSON, we leverage propositions of RML that is itself an extension of R2RML.

3 R2RML AND RML

R2RML is a generic language meant to describe customized mappings that translate data from a relational database into an RDF data set. An R2RML mapping is expressed as an RDF graph written in Turtle syntax[12]. An R2RML mapping graph consists of *triples maps*, each one specifying how to map rows of a logical table to RDF triples. A triples map is composed of exactly one *logical table* (property `rr:logicalTable`), one *subject map* (property `rr:subjectMap`) and any number of *predicate-object maps* (property `rr:predicateObjectMap`). A logical table may be a table, an SQL view (property `rr:tableName`), or the result of a valid SQL query (property `rr:sqlQuery`). A predicate-object map consists of *predicate maps* (property `rr:predicateMap`) and *object maps* (property `rr:objectMap`). For each row of the logical table, the subject map generates a subject IRI, while each predicate-object map creates one or more predicate-object pairs. Triples are produced by combining the subject IRI with each predicate-object pair. Additionally, triples are generated either in the default graph or in a named graph specified using *graph maps* (property `rr:graphMap`).

Subject, predicate, object and graph maps are all R2RML *term maps*. A term map is a function that generates RDF terms (either a literal, an IRI or a blank node) from elements of a logical table row. A term map must be exactly one of the following: a *constant-valued term map* (property `rr:constant`) always generates the same value; a *column-valued term map* (property `rr:column`) produces the value of a given column in the current row; a *template-valued term* map (property `rr:template`) builds a value from a template string that references columns of the current row.

When a logical resource is cross-referenced, typically by means of a foreign key relationship, it may be used as the subject of some triples and the object of some others. In such cases, a *referencing object map* uses IRIs produced by the subject map of a (parent) triples map as the objects of triples produced by another (child) triples map. In case both triples maps do not share the same logical table, a joint query must be performed. A join condition (property `rr:joinCondition`) names the columns from the parent and child triples maps, that must be joined (properties `rr:parent` and `rr:child`).

Below we provide a short illustrative example. Triples map `<#R2RML_Directors>` uses table `DIRECTORS` to create triples linking movie directors (whose IRIs are built from column `NAME`) with their birth date (column `BIRTH_DATE`).

```
<#R2RML_Directors>
 rr:logicalTable [
   rr:tableName "DIRECTORS"; ];
 rr:subjectMap [ rr:template
  "http://example.org/dir/{NAME}"; ];
 rr:predicateObjectMap [
   rr:predicate ex:bithdate;
   rr:objectMap [
     rr:column "BIRTH_DATE";
     rr:datatype xsd:date; ]; ].
```

RML is an extension of R2RML that targets the simultaneous mapping of heterogeneous data sources with various data formats, in particular hierarchical data formats. An RML logical source

[12] http://www.w3.org/TR/turtle/

```
Collection "directors":
{"name": "Woody Allen", "directed": ["Manhattan", "Interiors"]},
{"name": "Wong Kar-wai", "directed": ["2046", "In the Mood for Love"]}

Collection "movies":
{ "decade": "2000s", "movies": [
    {"name": "2046", "code": "m2046", "actors": ["T. Leung", "G. Li"]},
    {"name": "In the Mood for Love", "code": "Mood", "actors": ["M. Cheung"]} ] }
{ "decade": "1970s":, "movies": [
    {"name": "Manhattan", "code": "Manh", "actors": ["Woody Allen", "Diane Keaton"]}
    {"name": "Interiors", "code": "Int01", "actors": ["D. Keaton", "G. Page"]} ] }
```

Listing 4: Example Database

(property `rml:logicalSource`) extends R2RML logical table and points to the data source (property `rml:source`): this may be a file on the local file system, or data returned from a web service for instance. A reference formulation (property `rml:referenceFormulation`) names the syntax used to reference data elements within the logical source. As of today, possible values are `ql:JSONPath` for JSON data, `ql:XPath` for XML data, and `rr:SQL2008` for relational databases. Data elements are referenced with property `rml:reference` that extends `rr:column`. Its object is an expression whose syntax matches the reference formulation. Similarly, the definition of property `rr:template` is extended to allow such reference expressions to be enclosed within curly braces ('{' and '}'). Below we provide an RML example. It is very similar to the R2RML example above, with the difference that data now comes from a JSON file "directors.json".

```
<#RML_Directors>
 rml:logicalSource [
   rml:source "directors.json";
   rml:referenceFormulation ql:JSONPath;
   rml:iterator "$.*";  ];
 rr:subjectMap [ rr:template
   "http://example.org/dir/{$.*.name}";
 ];
 rr:predicateObjectMap [
   rr:predicate ex:bithdate;
   rr:objectMap [
     rml:reference "$.*.bithdate";
     rr:datatype xsd:date; ]; ].
```

4 THE XR2RML MAPPING LANGUAGE

In this section we breifly describe the elements of the xR2RML language. A complete specification is provided in (Michel et al., 2014a). We illustrate the descriptions with a running example: Listing 4 shows JSON documents stored in a MongoDB database, in two collections: a "directors" collection with documents on movie directors, and a "movies" collection in which movies are grouped in per-decade documents. Listing 5 shows an xR2RML mapping graph to translate those documents into RDF. Director IRIs are built using director names, while IRIs of resources representing movies use movie codes. We assume the following namespace prefix definitions (the `@predfix` key word is not displayed for readability):

```
xrr:  <http://www.i3s.unice.fr/ns/xr2rml#>.
rr:  <http://www.w3.org/ns/r2rml#>.
rml:  <http://semweb.mmlab.be/ns/rml#>.
xsd:  <http://www.w3.org/2001/XMLSchema#>.
ex:  <http://example.com/ns#>.
```

4.1 Describing A Logical Source

To reach its genericity objective, xR2RML must avoid explicitly referring to specific query languages or data models. Keeping this in mind, we define logical sources as a mean to represent a data set from any kind of database. In conformance with R2RML principles, we keep database connection details out of the scope of the mapping language. In RML on the other hand, a logical source points to the data to be mapped typically using a file URL (property `rml:source`). This difference makes it difficult for xR2RML to extend RML's logical source concept. Instead, xR2RML extends the R2RML logical source while commonalities are addressed by using or extending some RML properties (`rml:referenceFormulation`, `rml:query`, `rml:iterator`).

xR2RML triples maps extend R2RML triples maps by referencing a *logical source* (property `xrr:logicalSource`) which is the result of a request applied to the input database. It is either an *xR2RML base table* or an *xR2RML view*. The xR2RML base table extends the concept of *R2RML table or view* to tabular databases beyond relational databases (exten-

```
<#Movies>
   xrr:logicalSource [
      xrr:query "db.movies.find({decade:{$exists:true}})";
      rml:iterator "$.movies.*";
   ];
   rr:subjectMap [ rr:template "http://example.org/movie/{$.code}"; ];
   rr:predicateObjectMap [
      rr:predicate ex:starring;
      rr:objectMap [
         rr:termType xrr:RdfBag;
         xrr:reference "$.actors.*";
         xrr:nestedTermMap [ rr:datatype xsd:string; ]; ]; ].

<#Directors>
   xrr:logicalSource [ xrr:query "db.directors.find()"; ];
   rr:subjectMap [ rr:template "http://example.org/dir/{$.name}"; ];
   rr:predicateObjectMap [
      rr:predicate ex:directed;
      rr:objectMap [
         rr:parentTriplesMap <#Movies>;
         rr:join [
            rr:child "$.directed.*";
            rr:parent "$.name";
         ]; ]; ].
```

Listing 5: xR2RML Example Mapping Graph

sible column store, CSV/TSV, etc.). It refers to a table by its name (property `rr:tableName`). An xR2RML view represents the result of executing a query against the input database. It has exactly one `xrr:query` property that extends RML property `rml:query` (which itself extends `rr:sqlQuery`[13]). Its value is a valid expression with regards to the query language supported by the input database. No assumption is made whatsoever as to the query language used.

Reference formulation. Retrieving values from a query result set requires evaluating data element references against the query result, as discussed in section 1.2. Relational database APIs (such as JDBC drivers) natively support the evaluation of a column name against the current row of a result set. Conversely, some databases come with simple APIs that provide low-level evaluation features. For instance, APIs of most NoSQL document stores return JSON documents but hardly support JSONPath. Therefore, the responsibility of evaluating data element references may fall back on the xR2RML processing engine. To do so, it needs to know which syntax is being used. To this end, RML introduced the reference formulation concept (property `rml:referenceFormulation` of a logical source) to name the syntax of data element references. As underlined above, xR2RML adheres to R2RML's principle that database-specific details are out of the scope of the mapping language. Moreover, we want the mapping language to remain free from explicit reference to specific syntaxes. As a result, we amend the R2RML processor definition as follows: an xR2RML processor must be provided with a database connection and the reference formulation applicable to results of queries run against the connection. If the reference formulation is not provided, it defaults to column name, in order to ensure backward compatibility with R2RML.

Iteration model. In R2RML, the row-based iteration occurs on a set of rows read from a logical table. xR2RML applies this principle to other systems returning row-based result sets: CSV/TSV files, extensible column stores, but also some graph databases as underlined in 1.2, e.g. a SPARQL SELECT result set is a table in which columns are named after the variables in the SELECT clause. In the context of non row-based result sets, the model is implicitly extended to a *document-based iteration model*: a document is basically one entry of a result set returned by the database, e.g. a JSON document retrieved from a NoSQL document store, or an XML document retrieved from an XML native database. In the case of data sources whose access interface does not provide built-in iterators, e.g. a web service returning an XML response at once, then a single iteration occurs on the

[13] rml:query also subsumes rml:xmlQuery and rml:queryLanguage, although none of those properties are described or exemplified in the RML language specification and articles at the time of writing.

whole retrieved document.

Yet, some specific needs may not be fulfilled. For instance, it may be needed to iterate on explicitly specified entries of a JSON document or elements of an XML tree. To this end, we leverage the concept of iterator introduced in RML. An iterator (property `rml:iterator`) specifies the iteration pattern to apply to data read from the input database. Its value is a valid expression written using the syntax specified in the reference formulation. The iterator can be either omitted or empty when the reference formulation is a column name.

Listing 5 presents two logical source definition examples. Both consist of a MongoDB query (property `xrr:query`). We assume that the JSONPath reference formulation is provided along with the database connection. In collection "directors" (Listing 4), each document describes exactly one director. By contrast, in collection "movies" each document refers to several movies grouped by decade. To avoid mixing up multiple movies of a single document, an iterator with JSONPath expression `$.movies.*` is associated with triples map `<#Movies>`: thus, the triples map applies separately on each movie of each document.

4.2 Referencing Data Elements

In section 3 we have seen that RML properties `rml:reference` and `rr:template` both allow data element references expressed according to the reference formulation (column name, XPath, JSONPath). xR2RML uses these RML definitions as a starting point to a broader set of use cases.

In real world use cases, databases commonly store values written in a data format that they cannot interpret. For instance, in key-value stores and in most extensible column stores, values are stored as binary objects whose content is opaque to the system. A developer may choose to embed JSON, CSV or XML values in the column of a relational table, for performance issues or due to application design constraints. We call such cases *mixed content*.

xR2RML proposes to apply the principle of data element references defined in RML, and extend it to allow referencing data elements within mixed content. An xR2RML *mixed-syntax path* consists of the concatenation of several path expressions, each path being enclosed in a *syntax path constructor* that explicits the path syntax. Existing constructors are: Column(), CSV(), TSV(), JSONPath() and XPath(). For example, in a relational table, a text column `NAME` stores JSON-formatted values containing people's first and last names, e.g.: `{"First":"John", "Last":"Smith"}`. Field `FirstName` can be referenced with the following mixed-syntax path: `Column(NAME)/JSONPath($.First)`. An xR2RML processing engine evaluates a mixed-syntax path from left to right, passing the result of each path constructor on to the next one. In this example, the first path retrieves the value associated with column `NAME`. Then the value is passed on to the next path constructor that evaluates JSONPath expression "$.First" against the value. The resulting value is finally translated into an RDF term according to the current term map definition.

xR2RML defines property `xrr:reference` as an extension of RML property `rml:reference`, and extends the definition of property `rr:template`. Both properties accept either simple references (illustrated in Listing 5) or mixed-syntax path expressions.

4.3 Producing RDF Terms and (Nested) RDF Collections/Containers

In a row-based logical source, a valid column name reference returns zero or one value during each triples map iteration. In turn an R2RML term map generates zero or one RDF term per iteration. By contrast, JSONPath and XPath expressions used with properties `xrr:reference` and `rr:template` allow addressing multiple values. For instance, XPath expression `//movie/name` returns all `<name>` elements of all `<movie>` elements. Therefore, reference-valued and template-valued term maps can return multiple RDF terms at once. This change entails the definition of two strategies with regards to how triples maps combine RDF terms to build triples: the Cartesian product strategy, and the collection/container strategy.

Cartesian product strategy. 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 pair. Predicate-object pairs result of the Cartesian product between RDF terms produced by the predicate maps and object maps of each predicate-object map. Like any other term map, a graph map may also produce multiple terms. The Cartesian product strategy equally applies in that case, therefore triples are produced simultaneously in all target graphs corresponding to the multiple RDF terms produced by the graph map.

Collection/container strategy. Multiple values returned by properties `xrr:reference` and `rr:template` are combined into an RDF collection or container. This is achieved using new xR2RML values of the `rr:termType` property: a term map with term type `xrr:RdfList` generates an RDF term of type `rdf:List`, term type `xrr:RdfSeq` corresponds to

rdf:Seq, xrr:RdfBag to rdf:Bag and xrr:RdfAlt to rdf:Alt. Listing 5 illustrates this use case. Instead of generating multiple triples relating each movie to one actor, triples map <#Movies> relates each movie to a bag of actors starring in that movie. For instance:

```
<http://example.org/movie/m2046> ex:starring [
   a rdf:Bag;
   rdf:_1 "Tony Leung"; rdf:_2 "Gong Li" ].
```

At this point, two important needs must still be addressed in the collection/container strategy: (i) like in a regular term map, it must be possible to assign a term type, language tag or data type to the members of an RDF collection or container; and (ii) it must be possible to nest any number of RDF collections and containers inside each-other. Both needs are fulfilled using *xR2RML Nested Term Maps*. A nested term map (property xrr:nestedTermMap) very much resembles a regular term map, with the exception that it can be defined only in the context of a term map that produces RDF collections or containers. In a column-valued or reference-valued term map, a nested term map describes how to translate values read from the logical source into RDF terms, by specifying optional properties rr:termType, rr:language and rr:datatype. Similarly, in a template-valued term map, a nested term map applies to values produced by applying the template string to input values. Listing 5 illustrates the usage of nested term maps by the production of bags of literals representing movie names: the nested term map assigns each movie name an xsd:string datatype. For instance:

```
<http://example.org/movie/m2046> ex:starring [
   a rdf:Bag;
   rdf:_1 "Tony Leung"^^xsd:string;
   rdf:_2 "Gong Li"^^xsd:string ].
```

Finally, properties xrr:reference and rr:template can be used within a nested term map to recursively parse structured values while producing nested RDF collections and containers.

4.4 Reference Relationships Between Logical Sources

A cross-referenced logical resource usually serves as the subject of some triples and the object of other triples. In R2RML, this is achieved using a referencing object map. xR2RML extends R2RML referencing object maps in two ways. Firstly, when a joint query is needed (i.e. the parent and chlid triples map do not share the same logical source), properties rr:child and rr:parent of the join condition contain data element references (4.2), possibly including mixed-syntax paths. As underlined in section 4.3, such data element references may produce multiple terms. Consequently, the equivalent joint query of a referencing object map must deal with multi-valued child and parent references. More precisely, a join condition between two multi-valued references should be satisfied if at least one data element of the child reference matches one data element of the parent reference. This is described in Definition 1 using an SQL-like syntax and first order logic for the description of WHERE conditions.

Definition 1: If a referencing object map has at least one join condition, then its equivalent joint query is:

$$\begin{gathered}\textit{SELECT * FROM (child-query) AS child,}\\ \textit{(parent-query) AS parent WHERE}\\ \exists c1 \in \textit{eval(child, \{child-ref1\}),}\\ \exists p1 \in \textit{eval(parent, \{parent-ref1\}), } c1 = p1\\ \textit{AND}\\ \exists c2 \in \textit{eval(child, \{child-ref2\}),}\\ \exists p2 \in \textit{eval(parent, \{parent-ref2\}), } c2 = p2\\ \textit{AND ...}\end{gathered}$$

where "{child-refi}" and "{parent-refi}" are the child and parent references of the i^{th} join condition, and "eval(child, {ref})" and "eval(parent, {ref})" are the result of evaluating data element reference "{ref}" on the result of the child and parent queries.

Listing 5 depicts a simple example: in triples map <#Directors>, the object map uses movie IRIs generated by parent triples map <#Movies>. When processing director "Wong Kar-wai", the child reference ($.directed.*) returns values "2046" and "In the Mood for Love", while the parent reference ($.name) returns a single movie name. The condition is satisfied if the parent reference returns one of "2046" or "In the Mood for Love". Generated triples use movie codes to build movie IRIs, such as:

```
<http://example.org/dir/Wong%20Kar-wai>
ex:directed <http://example.org/movie/m2046>.
```

Secondly, the objects produced by a referencing object map can be grouped in an RDF collection or container, instead of being the objects of multiple triples. To do so, an xR2RML referencing object map may have a rr:termType property with value xrr:RdfList, xrr:RdfSeq, xrr:RdfBag or xrr:RdfAlt. Results of the joint query are grouped by child value, i.e. objects generated by the parent triples map, referring to the same child value, are grouped as members of an RDF collection or container. An interesting consequence of this use case is the ability, in the case of a regular relational database, to build an RDF collection or container reflecting a one-to-many relation.

5 IMPLEMENTATION AND EVALUATION

To evaluate the effectiveness of xR2RML, we have developed an open source prototype implementation available on Github[14]. It is developed in Scala and based on Morph-RDB (Priyatna et al., 2014), an R2RML implementation that we have extended to support xR2RML specificities.

In a first step, we upgraded Morph-RDB to support xR2RML features in the context of relational databases. This included the support of logical sources, mixed contents (JSON, XML, CSV or TSV data embedded in cells) and RDF collections/containers. In a second step, we developed a connector to the MongoDB document store, to translate MongoDB JSON documents into RDF. A MongoDB shell query string is specified in each triples map logical source (property `xrr:query`). The connector executes the query and iterates over result documents returned by the database. Subsequently, results are passed to the xR2RML processor that applies the optional iterator (`rml:iterator`) and evaluates JSONPath expressions in each `xrr:reference` and `rr:template` property of all term maps. The support of RDF collections and containers was validated, in particular in the case of cross-references (referencing object map) that entail a joint query between two JSON documents.

Software architecture. The prototype architecture derives from the initial Morph-RDB architecture. To deal with the heterogeneity of databases, Morph follows the object factory design pattern. An abstract *runner factory* class provides abstract methods to build a *runner*, the core object that performs the translation of an input database with regards to an xR2RML mapping graph. A concrete runner factory class copes with database specificities through a set of objects: (i) a generic connection wraps a database connection; (ii) a query unfolder builds a concrete query object reflecting each defined triples map; (iii) a data translator runs the query against the database connection and generates triples according to the triples map definitions; (iv) finally, a data materializer writes created triples into a target file according to the chosen RDF serialization.

In the current status we provide two factory implementations: the RDB implementation extends the original Morph-RDB code, while the new MongoDB implementation relies on the MongoDB API and the Jongo API for the management of MongoDB shell queries. In the RDB context, the unfolder builds an SQL query from the table name (logical table definition), named columns (properties`rr:column` and `rr:template`) and the optional join conditions (referencing object maps). In the MongoDB case, the query string is provided in the mapping. Furthermore, since MongoDB does not support joint queries, the xR2RML processing engine has to perform two queries and join results afterwards. As a result, the unfolder is fairly simple, it checks the query string correctness and returns an appropriate API object.

Evaluation. We evaluated the prototype using two simple databases: a MySQL relational database and a MongoDB database with two collections. In both cases, the data and associated xR2RML mappings were written to cover most mapping situations addressed by xR2RML: strategies for handling multiple RDF terms, JSONPath and XPath expressions, mixed-syntax paths with mixed contents (relational, JSON, XML, CSV/TSV), cross-references, production of RDF collection/containers, management of UTF-8 characters. A dump of both databases as well as the example mappings are available on the same GitHub repository. The current status of the prototype applies the data materialization approach, i.e. RDF data is generated by sequentially applying all triples maps. The query rewriting approach (SPARQL to database specific query rewriting) may be considered in future works as suggested in section 7. At the time of writing the prototype has two limitations: (i) only one level of RDF collections and containers can be generated (no nested collections/containers), and (ii) the result of a joint query in a relational database cannot be translated into an RDF collection or container.

6 DISCUSSION

xR2RML relies on the assumption that databases to translate into RDF provide a declarative query language, such that queries can be expressed directly in a mapping description. This complies with the equivalent assumption of R2RML that all RDBs support ANSI SQL. However this is somehow restrictive. Some NoSQL key-value stores, like DynamoDB and Riak, have no declarative query language, instead they provide APIs for usual programming languages to describe queries in an imperative manner. For xR2RML to work with those systems, a query language should be figured out along with a compiler that transforms queries into imperative code. Interestingly, this is already the case of some systems supporting the MapReduce programming model. MapReduce is conventionally supported through APIs for programming languages, however more and more systems now propose an SQL or SQL-

[14]https://github.com/frmichel/morph-xr2rml/

like query language on top of a MapReduce framework (e.g. Apache Hive). Queries are compiled into MapReduce jobs. This approach is often referred to as SQL-on-Hadoop (Floratou et al., 2014).

To achieve the targeted flexibility, xR2RML comes with features that are applicable independently of the type of database used. Nevertheless, all features should probably not be applied with all kinds of database. For instance, join conditions entail joint queries. Whereas RDBs are optimized to support joins very efficiently, it is not recommended to make cross-references within NoSQL document or extensible column stores, as this may lead to poor performances. Similarly, translating a JSON element into an RDF collection is quite straightforward, but translating the result of an SQL joint query into an RDF collection is likely to be quite inefficient. In other words, because the language makes a mapping possible does not mean that it should be applied regardless of the context (database type, data model, query capabilities). Consequently, mapping designers should be aware of how databases work in order to write efficient mappings of big databases to RDF.

Like R2RML, xR2RML assumes that well-defined domain ontologies exist beforehand, whereof classes and properties will be used to translate a data source into RDF triples. In the context of RDBs, an alternative approach, the Direct Mapping, translates relational data into RDF in a straightforward manner, by converting tables to classes and columns to properties (Sequeda et al., 2011; Arenas et al., 2012). The direct mapping comes up with an ad-hoc ontology that reflects the relational schema. R2RML implementations often provide a tool to automatically generate an R2RML direct mapping from the relational schema (e.g. Morph-RDB (de Medeiros et al., 2015)). The same principles could be extended to automatically generate an xR2RML mapping for other types of data source, as long as they comply with a schema: column names in CSV/TSV files and extensible column stores, XSD or DTD for XML data, JSON schema[15] or a JSON-LD[16] description for JSON data. Nevertheless, such schemas do not necessarily exist, and some databases like the DynamoDB key-value store are schemaless. In such cases, automatically generating an xR2RML direct mapping should involve different methods aimed at learning the database schema from the data itself.

More generally, how to automate the generation of xR2RML mappings may become a concern to map large and/or complex schemas. There exists significant work related to schema mapping and matching (Shvaiko and Euzenat, 2005). For instance, Clio (Fagin et al., 2009) generates a schema mapping based on the discovery of queries over the source and target schemas and a specification of their relationships. Karma (Knoblock et al., 2012) semi-automatic maps structured data sources to existing domain ontologies. It produces a Global-and-Local-As-View mapping that can be used to translate the data into RDF. xR2RML does not directly address the question of how mappings are written, but can be complementary of approaches like Clio and Karma. In particular, Karma authors suggest that their tool could easily export mapping rules as an R2RML mapping graph. A similar approach could be applied to discover mappings between a non-relational database and domain ontologies, and export the result as an xR2RML mapping graph.

[15]http://json-schema.org/

[16]http://www.w3.org/TR/json-ld/

7 CONCLUSION AND PERSPECTIVES

In this paper we have presented xR2RML, a language designed to describe the mapping of various types of databases to RDF, by flexibly adapting to heterogeneous query languages and data models. We have analysed data models of several modern databases as well as the format in which query results are returned, and we have shown that xR2RML can translate any data element within such results into RDF, relying when necessary on existing languages such as XPath and JSONPath. We have illustrated some features of xR2RML such as the generation of RDF collections and containers, and the ability to deal with mixed content, e.g. when a relational table stores data formatted in another syntax like XML, JSON or CSV.

Principles of the xR2RML mapping language have been validated in a prototype implementation supporting several RDBs and the MongoDB NoSQL document store. The development of connectors to other types of database shall be considered based on concrete use cases. Depending on the target system, different optimizations shall be studied, notably regarding the computation of joint queries. Furthermore, the data materialization approach we implemented is effective but it does not scale to big data sets. Dealing with big data sets requires the data to remain in legacy databases, and that translation to RDF be performed on demand through the xR2RML-based rewriting of SPARQL queries into the source database query language. In this regard, existing works related to RDBs should be leveraged (Priyatna et al., 2014; Sequeda and Miranker, 2013).

REFERENCES

Acosta, M., Vidal, M., Lampo, T., Castillo, J., and Ruckhaus, E. (2011). ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *Proc. of ISWC'11*, pages 18–34.

Arenas, M., Bertails, A., Prud'hommeaux, E., and Sequeda, J. (2012). A direct mapping of relational data to RDF.

Bikakis, N., Tsinaraki, C., Stavrakantonakis, I., Gioldasis, N., and Christodoulakis, S. (2013). The SPARQL2XQuery interoperability framework. *CoRR*, abs/1311.0536.

Bischof, S., Decker, S., Krennwallner, T., Lopes, N., and Polleres, A. (2012). Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics*, 1(3):147–185.

Breitling, F. (2009). A standard transformation from XML to RDF via XSLT. *Astronomical Notes*, 330:755.

Das, S., Sundara, S., and Cyganiak, R. (2012). R2RML: RDB to RDF mapping language.

de Medeiros, L. F., Priyatna, F., and Corcho, O. (2015). MIRROR: Automatic R2RML mapping generation from relational databases. In *Submission to ICWE 2015*.

Dimou, A., Sande, M. V., Slepicka, J., Szekely, P., Mannens, E., Knoblock, C., and Walle, R. V. d. (2014a). Mapping hierarchical sources into RDF using the RML mapping language. In *Proc. of ICSC'2014*, pages 151–158. IEEE.

Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., and Van de Walle, R. (2014b). RML: A generic language for integrated RDF mappings of heterogeneous data. In *Proc. of the 7th LDOW workshop*.

Fagin, R., Haas, L. M., Hernndez, M., Miller, R. J., Popa, L., and Velegrakis, Y. (2009). Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236. Springer.

Fennell, P. (2014). Schematron - more useful than you'd thought. In *Proc. of the XML London 2014 Conference*, pages 103–112.

Field, L., Suhr, S., Ison, J., Wittenburg, P., Los, W., Broeder, D., Hardisty, A., Repo, S., and Jenkinson, A. (2013). Realising the full potential of research data: common challenges in data management, sharing and integration across scientific disciplines.

Floratou, A., Minhas, U. F., and Ozcan, F. (2014). Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proc. of the VLDB Endowment*, 7(12).

Gaignard, A. (2013). Distributed knowledge sharing and production through collaborative e-science platforms. PhD thesis.

Gajendran, S. K. (2013). A survey on NoSQL databases (technical report).

He, B., Patel, M., Zhang, Z., and Chang, K. C.-C. (2007). Accessing the deep web. *Communications of the ACM*, 50(5):94–101.

Hecht, R. and Jablonski, S. (2011). NoSQL evaluation: A use case oriented survey. In *Proc. of CSC'2011*, pages 336–341. IEEE Computer Society.

Knoblock, C. A., Szekely, P., Ambite, J. L., Goel, A., Gupta, S., Lerman, K., Muslea, M., Taheriyan, M., and Mallick, P. (2012). Semi-automatically mapping structured sources into the semantic web. In *Proc. of ESWC'2012*, pages 375–390. Springer.

Kolev, B., Valduriez, P., Jimenez-Peris, R., Martìnez-Bazan, N., and Pereira, J. (2014). CloudMdsQL: Querying heterogeneous cloud data stores with a common language. In *Proc. of the BDA'2014 Conference*.

Langegger, A. and Wöss, W. (2009). XLWrap - querying and integrating arbitrary spreadsheets with SPARQL. In *Proc. of ISWC'2009*.

Melton, J., Michels, J. E., Josifovski, V., Kulkarni, K., and Schwarz, P. (2002). SQL/MED: a status report. *ACM SIGMOD Record*, 31(3):81–89.

Michel, F., Djimenou, L., Faron-Zucker, C., and Montagnat, J. (2014a). xR2RML: Relational and non-relational databases to RDF mapping language. Research report. ISRN I3S/RR 2014-04-FR v3.

Michel, F., Montagnat, J., and Faron-Zucker, C. (2014b). A survey of RDB to RDF translation approaches and tools. Research report. ISRN I3S/RR 2013-04-FR.

Ong, K. W., Papakonstantinou, Y., and Vernoux, R. (2014). The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases (submitted). *CoRR*, abs/1405.3631.

Priyatna, F., Corcho, O., and Sequeda, J. (2014). Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph. In *Proc. of WWW'2014*.

Roth, M. T. and Schwartz, P. (1997). Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of VLDB'1997*, pages 266–275.

Scharffe, F., Atemezing, G., Troncy, R., Gandon, F., Villata, S., Bucher, B., Hamdi, F., Bihanic, L., Képéklian, G., Cotton, F., and others (2012). Enabling linked data publication with the Datalift platform. In *Proc. of the AAAI workshop on semantic cities*.

Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). FedX: Optimization techniques for federated query processing on linked data. In *Proc. of ISWC'11*, pages 601–616.

Sequeda, J., Tirmizi, S. H., Corcho, s., and Miranker, D. P. (2011). Survey of directly mapping SQL databases to the semantic web. *Knowledge Eng. Review*, 26(4):445–486.

Sequeda, J. F. and Miranker, D. P. (2013). Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the WWW*, 22:19–39.

Shvaiko, P. and Euzenat, J. (2005). A survey of schema-based matching approaches. In *Journal on Data Semantics IV*, pages 146–171. Springer.

Spanos, D.-E., Stavrou, P., and Mitrou, N. (2012). Bringing relational databases into the semantic web: A survey. *Semantic Web Journal*, 3(2):169–209.