

KR2RML: An Alternative Interpretation of R2RML for Heterogeneous Sources^{*}

Jason Slepicka, Chengye Yin, Pedro Szekely, and Craig A. Knoblock

University of Southern California
Information Sciences Institute and Department of Computer Science, USA
{knoblock,pszekely,slepicka}@isi.edu
{chengyey}@usc.edu

Abstract. Data sets are generated today at an ever increasing rate in a host of new formats and vocabularies, each with its own data quality issues and limited, if any, semantic annotations. Without semantic annotation and cleanup, integrating across these data sets is difficult. Approaches exist for integration by semantically mapping such data using R2RML and its extension for heterogeneous sources, RML, into RDF, but they are not easily extendable or scalable, nor do they provide facilities for cleaning. We present an alternative interpretation of R2RML paired with a source-agnostic R2RML processor that supports data cleaning and transformation. With this approach, it is easy to add new input and output formats without modifying the language or the processor, while supporting the efficient cleaning, transformation, and generation of billion triple datasets.

1 Introduction

Over the past decade, many strides have been made towards enabling content creators to publish Linked Open Data by building on what they're familiar with: RDFa, HTML5 Microdata, JSON-LD [11], and schema.org. There still exist countless messy data sets, both legacy and novel, that lack the semantic annotation necessary to be Linked Open Data. Numerous ETL suites attack the messy part of the problem, i.e. data cleaning, data transformation, and scale, but lack robust support for semantically mapping the data into RDF using ontologies. Other approaches provide support for semantically mapping data but cannot address messy data nor easily extend to support new formats. Our proposed interpretation of R2RML [12] and processor, implemented in the open source tool, Karma [6], addresses these concerns as captured by the following requirements.

Multiple inputs and Output Formats A language for semantically mapping data should extend beyond just mapping relational databases to RDF. Solutions

^{*} A KR2RML processor implementation is available for evaluation at <https://github.com/usc-isi-i2/Web-Karma>.

for relational databases can cleanly apply to tabular formats like CSV but break down or cannot fully capture the semantic relationships present in the data when applied to hierarchical sources like JSON and XML. A mapping should be able to capture these relationships. In terms of mapping output, there are many ways for a processor to serialize an RDF graph: RDF/XML, Turtle, N-Quads, RDF-HDT [4], and JSON-LD. A processor that can output nested JSON-LD is particularly valuable to developers and data scientists, because the same techniques required to serialize the graph as a collection of trees in JSON can be extended to any other hierarchical serialization format.

Extensibility As serialization formats propagate and evolve to fulfill new niches, it's important to be able to quickly support new ones. Consider JSON's offspring: BSON [9] and HOCON [13], or Interface Description Languages like Google Protocol Buffers [5] and Apache Avro [1]. Each format is JSON-like but requires its own libraries and query language, if one exists. A mapping language and its processor should reuse commonality between hierarchical formats and require few, if any, changes to be able to support them on ingestion and output.

Transformations Integrating structured data without consideration for cleaning and transformation as languages (both natural and artificial), vocabularies, standards, schemas, and formats proliferate is difficult. As such, design decisions for semantic modeling should not be made independently of data transformation. R2RML templates are insufficient for overcoming the complexities and messiness of hierarchical data. Instead, R2RML users rely on custom SQL queries or views to present a clean dataset for mapping. A better solution wouldn't require the user to know anything about the idiosyncrasies or domain-specific language of the source at all, e.g. DOM vs SAX parsing and XSLT for XML.

Scalability As the academic and commercial world regularly works with terabytes and even petabytes of data, any language's processor must carefully weigh the implications of its algorithms and feature sets in regards to scalability. While an extensive feature set may be attractive in terms of offering a comprehensive working environment, some features may be better supported by external solutions. Other features may not even be representative of those needed by the most common workloads and serve only as distractions.

2 Approach

To meet these requirements, we rely on the Nested Relational Model (NRM)[7] as an intermediate form to represent data. The NRM allows us to abstract away the idiosyncrasies of the formats we support, so that once we translate the input data into this intermediate form, every downstream step, e.g. cleaning, transformation, RDF generation, is reusable. Adding a new source format is as simple as defining how to parse and translate it to this model. This frees us

from having to include references to input format specific considerations like XPath or XSLT in the language. Once the data is in the NRM, we can support and implement a set of common data transformations in the processor once, instead of for each input format. These transformations, e.g. split, fold/unfold, are common in the literature, but we extend their definitions as found in the Potter’s Wheel interactive data cleaning system [10] from a tabular data model to support the NRM. Our processor also provides an API for cleaning by example and transformations using User Defined Functions (UDFs) written in Python. After translating data to the NRM, cleaning it, and transforming it, we extend the R2RML column references to map to the NRM. This model allows us to uniformly apply our new interpretation of R2RML mapping for hierarchical data sources to generate RDF. Finally, for scalability, our processor does not support joins across logical tables, which precludes using KR2RML as a virtual SPARQL endpoint for relational databases. Instead we support materializing the RDF by processing in massively parallel frameworks in both batch via Hadoop and incrementally in a streaming manner like Storm.

3 Nested Relational Model

We map data into the Nested Relational Model by translating it into tables and rows where a column in a table can be either a scalar value or a nested table. Mapping tabular data like CSV, Excel, and relational databases is straightforward. The model will have a one to one mapping of tables, rows, and columns, unless we perform a transformation like split on a column, which will create a new column that contains a nested table. To illustrate how we map hierarchical data, we will describe our approach to JSON first and introduce an example¹. The example describes an organization, its employees and its locations.

```

1 {
2   "companyName": "Information Sciences Institute",
3   "tags": ["artificial intelligence", "nlp", "semantic web"]
4   "employees": [
5     {
6       "name": "Knoblock, Craig",
7       "title": "Director, Research Professor"
8     },
9     {
10      "name": "Slepicka, Jason",
11      "title": "Graduate Student, Research Assistant"
12    }
13  ],
14  "locationTable": {
15    "locationAddress" : [

```

¹ The example worked in this paper is available at <https://github.com/usc-isi-i2/iswc-2015-cold-example>

```

16         "4676 Admiralty Way Suite 1001,Marina Del Rey, CA 90292",
17         "3811 North Fairfax Drive Suite 200,Arlington, VA 22203"
18     ], "locationName" :["ISI - West", "ISI - East" ]
19     }
20 }

```

This document contains an object, which maps to a single row table in NRM with four columns for its four fields: `companyName`, `tags`, `employees` and `locationTable`. Each column is populated with the value of the appropriate field. Fields with scalar values, like `companyName`, need no translation, but fields like `tags`, `employees` and `locationTable`, which have array values, do.

The array values of `tags`, `employees` and `locationTables` are now mapped to their own nested tables. If the array contains scalar or object values, each array element becomes a row in the nested table. If the elements are scalar values like strings as in the `tags` field, a default column name “values” is provided, otherwise the objects in `employees` and `locationTable` and arrays in `locationAddress` and `locationName` are interpreted recursively using the rules just described. If a JSON document contains a JSON array at the top level, each element is treated like a row in a database table.

Once the data has been mapped to the NRM, we can refer to nested elements by creating a path to the appropriate column from the top level table. For example, the field referred to by the JSONPath `$.employees.name` in the source document is now the column [“employees”, “names”] in the NRM. An illustration of the NRM as displayed by the Karma GUI can be seen in Figure 1. Other hierarchical formats follow from this approach, sometimes directly; we use existing tools for translating XML and Avro to JSON and then import the sources as JSON. XML elements are treated like JSON objects, its attributes are modeled as a single row nested table where each attribute is a column. Repeated child elements become a nested table as well.

companyName	tags	employees	locationTable																
Information Sciences Institute	<table border="1"> <thead> <tr> <th>values</th> </tr> </thead> <tbody> <tr> <td>artificial intelligence</td> </tr> <tr> <td>nlp</td> </tr> <tr> <td>semantic web</td> </tr> </tbody> </table>	values	artificial intelligence	nlp	semantic web	<table border="1"> <thead> <tr> <th>name</th> <th>title</th> </tr> </thead> <tbody> <tr> <td>Knoblock, Craig</td> <td>Director, Research Professor</td> </tr> <tr> <td>Slepicka, Jason</td> <td>Graduate Student, Research Assistant</td> </tr> </tbody> </table>	name	title	Knoblock, Craig	Director, Research Professor	Slepicka, Jason	Graduate Student, Research Assistant	<table border="1"> <thead> <tr> <th>locationAddress</th> <th>locationName</th> </tr> </thead> <tbody> <tr> <td>4676 Admiralty Way Suite 1001, Marina Del Rey, CA 90292</td> <td>ISI - West</td> </tr> <tr> <td>3811 North Fairfax Drive Suite 200, Arlington, VA</td> <td>ISI - East</td> </tr> </tbody> </table>	locationAddress	locationName	4676 Admiralty Way Suite 1001, Marina Del Rey, CA 90292	ISI - West	3811 North Fairfax Drive Suite 200, Arlington, VA	ISI - East
values																			
artificial intelligence																			
nlp																			
semantic web																			
name	title																		
Knoblock, Craig	Director, Research Professor																		
Slepicka, Jason	Graduate Student, Research Assistant																		
locationAddress	locationName																		
4676 Admiralty Way Suite 1001, Marina Del Rey, CA 90292	ISI - West																		
3811 North Fairfax Drive Suite 200, Arlington, VA	ISI - East																		

Fig. 1: Nested Relational Model example displayed in Karma GUI

4 KR2RML vs. R2RML

We propose the following modifications on how to interpret R2RML language elements to support large, hierarchical data sets.

rr:logicalTable The R2RML specification allows the processor to access the input database referenced by the `rr:logicalTable` through an actual database or some materialization. For legacy reasons, we support the former for relational databases with `rr:tableName` and `rr:sqlQuery`, but only the latter for non-relational database sources.

rr:column To support mapping nested columns in the NRM, we no longer limit column-valued term maps to SQL identifiers. Instead, we allow a JSON array to capture the column names that make up the path to a nested column from the document root. As described previously, the `$.employees.name` field is mapped in the NRM to `["employees", "name"]`. `$.companyName` is unchanged from how R2RML would map tabular data, `"companyName"`, since it is not nested. We chose to forgo languages like XPath or JSONPath for expressing complex paths to nested columns, because supporting their full feature sets complicates the processor implementation. A simple, format agnostic path preserves the ability to apply a mapping to any format as long as the column names match.

rr:template Often, applications require URIs that cannot be generated by concatenating constants and column values for `rr:templates` without preprocessing from `rr:sqlQuery`. Instead, we allow the template to include columns that do not exist in the original input but are the result of the transformations applied by the processor. For example, in Figure 2a, `["employees", "employeeURI"]` is derived from `"companyName"` and `["employees", "name"]`.

rr:joinCondition Joins are integral to how relational databases are structured, so they play an important role when mapping data across tables using R2RML. When dealing with either streaming data or data on the order of terabytes or greater, these join conditions across sources are impractical at best and require extensive planning and external support in the form of massively parallel processing engines like Spark or Impala to achieve reasonable performance. This is out of the scope of our processor, so we do not support them. We have rarely found a need for them in practice.

km-dev:hasWorksheetHistory This is a tag introduced to capture the cleaning, transformation and modeling steps taken by a Karma user to generate a KR2RML mapping. The steps are intended to be performed on the input before RDF generation, but its presence is optional. Without these steps, there is nothing in a KR2RML mapping that a normal R2RML processor would not understand, except for the nested field references.

5 Input Transformations

One additional benefit of using the Nested Relational Model is that we can use it to create a mathematical structure with transformation functions that are closed on the universe of nested relational tables. Here, we can reuse the transformations outlined in a system like Potter’s wheel, but, instead of forcing operations like Split to create a cartesian product of rows, a transformation function can create a new set of nested tables. Figure 2 shows the result of applying the transformations to the example data shown in the Karma GUI. The blue columns are present in the source data, while the brown columns contain the transformation results.

(a) URI Template generated from Python Transformation

(b) Split Transformation

(c) Glue Transformation

(d) Python Transformations

(c) Glue Transformation (d) Python Transformations

Fig. 2: Karma Transformations

In Figure 2b, the split transformation breaks apart comma separated values in a column so the job role titles can be mapped into individual triples. The NRM allows us to insert a new column with a nested table where each row has a single column for the title. The other transformations, Unfold, Fold, and Group By follow similarly.

One unique challenge faced by the Nested Relational Model is transforming data from across nested tables. This can be accomplished in two ways. The first is demonstrated in Figure 2c. The locationAddress and locationName fields are mapped from the data as two nested tables but each row should be joined to its pair with the same index in the other table. This is accomplished by a Glue Transformation. Alternatively, it could be accomplished by executing a Python Transformation.

Python Transformations allow the user to write Python functions to reference, combine, and transform any data in the model and add the result as a new

column in the NRM. This is akin to User Defined Functions (UDFs) in Cloudera Morphlines, Pig, and Hive, in the sense that the processor can take as input libraries of UDFs written in Python. In Figure 2d, the Python Transformations are used to extract and format the components of the locationAddress field for mapping. Python Transformations are most commonly used to augment templates in R2RML because templates have limited utility as illustrated in Figure 2a. Finally, the processor supports using these same Python APIs to select, or filter, data from the model, by evaluating a UDF that returns a boolean for each column value indicating whether to ignore it during RDF generation.

6 RDF Generation

The final step in this process is to apply the R2RML mapping to the Nested Relational Model and generate RDF. Without careful consideration of how to apply the mapping, the process could be computationally very expensive or be artificially limited in the relationships its allowed to express in the hierarchy as the order in which the input data is traversed to generate the RDF is very important. We approach this in three steps: translate the TriplesMaps of an R2RML mapping into an execution plan, evaluate the TriplesMaps by traversing the NRM, and finally serializing the RDF.

An example R2RML mapping for this source is illustrated in Figure 3 and available online. The JSON document has been semantically modeled according to the schema.org ontology. The dark gray bubbles in Figure 3 correspond to TriplesMaps. Their labels are their SubjectMap's class. The arcs between the gray bubbles are PredicateObjectMaps with RefObjectMaps, while the arcs from the gray bubbles to the worksheet are PredicateObjectMaps with ObjectMaps with column references. The light grey bubbles on the arcs are the Predicate.

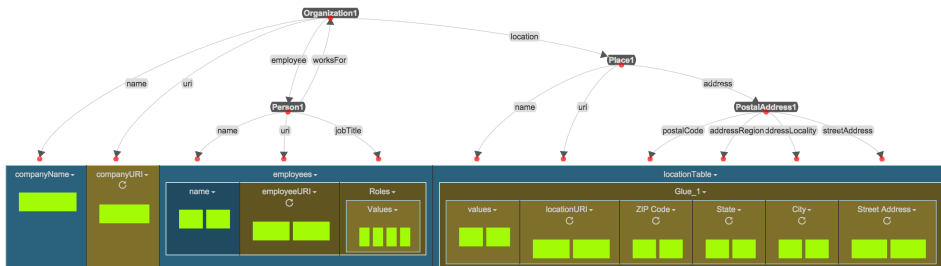


Fig. 3: KR2RML Mapping Graph mapped to NRM in Karma

R2RML Execution Planning To create an execution plan, we first consider the R2RML mapping as a directed, cyclic multigraph where TriplesMaps are the vertices, V , and the PredicateObjectMaps with RefObjectMaps as edges, E . From this directed, cyclic multigraph, we break cycles by flipping the direction of the edges, until we can create a topological ordering over the TriplesMaps. This ordering then become an execution plan, starting from the TriplesMaps with no outgoing edges. The algorithm is presented as follows.

Generate Topological Ordering `GenerateTopologicalOrdering` is applied to each graph in the multigraph of the R2RML mapping. It returns a topologically sorted list of `TriplesMaps` for processing, along with edges that were flipped to break cycles or simplify processing. The resulting lists for a multigraph can be processed independently and in parallel. The root will be the last `TriplesMap` processed. Its selection has important implications for generating JSON-LD which will be discussed later.

```

GenerateTopologicalOrdering(V,E, root)
  (spilled, flipped, V, E) := spill(V,E, root, {},{ })
  if(|V| > 0)
    flipped := DFSBreakCycles(V, E, root, {}, root, flipped)
    (spilled, flipped, V, E) = spill(V,E, root, spilled, flipped)
  return spilled, flipped

```

Spill TriplesMaps `Spill` iteratively removes `TriplesMaps` for processing from the R2RML mapping until all nodes have outgoing edges or a cycle is found. The earlier a `TriplesMap` is spilled, the earlier it is processed.

```

Spill(V, E, root, spilled, flipped)
  modifications := true
  while(|V| > 0 && modifications)
    modifications := false
    for( v in V)
      //v has only one edge or all edges are incoming
      if(deg(v) == 1 || outDeg(v) == 0)
        if(v == root && deg(v) > 0)//don't spill root
          continue
        //flip edges because spilled nodes
        //are "lower" in hierarchy
        for(e in edges(v))
          if(source(e) == v && v != root)
            flipped = flipped + e
          E = E - e
        if(e in edges(v) == 0)//remove edgeless vertexes
          V = V - v
          spilled = V + v
        modifications := true
  return spilled, flipped, V, E

```

Break Cycles `DFSBreakCycles` arbitrarily break cycles in the graph by performing a Depth First Search of the `TriplesMaps` graph, starting at the root, flipping incoming edges if the target node is visited before the source

```

DFSBreakCycles(V, E, root, visited, current, flipped)
  toVisit := {}

```



```

visited := visited + current
//sort edges so edges so outgoing edges are first
sortedEdges := sort(edges(current))
for( e in sortedEdges)
  if(source(e) = v)
    toVisit := toVisit + target(e)
  else
    if(!visited(source(e)))
      flipped := flipped + e
      toVisit := toVisit + source(e)
for( v in toVisit)
  if(!visited(v))
    flipped, visited := DFSBreakCycles(V, E, root, visited, v,
    flipped)
return flipped, visited

```

If we apply `GenerateTopologicalOrdering` to the mapping illustrated in Figure 3, it takes the following steps.

- Spill removes `PostalAddress`, then `Place`, then stops because of a cycle
- `DFSBreakCycles` starts at `Organization`, and sorts its edges, outgoing first
- Root is the source for `schema:employee`, so `Person` is added to `toVisit` set
- Root is not the source for `schema:worksFor`, so it is added to `flipped` set
- `Person` is the target for `schema:worksFor`, but it already a part of `toVisit`
- `DFSBreakCycles` visits the next node in the `toVisit` set, `Person`
- `DFSBreakCycles` halts since `Person`'s edges point to `root`, which is in `visited`
- The Spill algorithm is applied again and it removes `Person`
- This leaves the root, `Organization`, without any links, so Spill removes it

This results in a topological ordering of the `TriplesMaps` for the processor: `PostalAddress`, `Place`, `Person`, `Organization`.

R2RML TriplesMap Evaluation After the transformations, the NRM is immutable, so topologically sorted `TriplesMaps` can be evaluated according to their partial order on massive data in parallel without interference. As the processor evaluates each `TriplesMap`, it first populates the `SubjectMap` templates. The processor populates templates by generating the n-tuple cartesian product of their column values, which it derives by finding the table reachable by the shortest common subpath from the root between the column paths and then navigating the NRM to the columns from there. For a tabular format, the cardinality of the n-tuple set is always 1, just like R2RML applied to a SQL table. For our JSON example, we consider each JSON object in the `employee` array individually, so the cardinality for the `schema:Person` in the first `Organization` is 2.

This example lacks templates that cover multiple nested tables, but, for a mapping that does, the shortest common subpath between columns is captured by a concept called column affinity. Every column has an affinity with every other column. The closest affinity is with columns in the same row, then columns

descending from the same parent table row, then columns descending from a common row in an ancestor table, then columns with no affinity.

The processor populates PredicateObjectMaps with ObjectMaps that have column and template values by navigating the NRM in relation to the elements that make up the SubjectMap templates, bound by the tightest affinity between the columns of the SubjectMap Templates and the columns of the ObjectMap templates. Values common to multiple subjects can be cached.

The processor evaluates PredicateObjectMaps with RefObjectMaps by populating the templates in a similar fashion to regular ObjectMaps. A predicateMap with a template, however, requires calculating an affinity for the template population algorithm between the template and both of the Subject and Object templates to ensure only the right cartesian product is considered. Edges that have been flipped while generating the topological ordering are evaluated at the TriplesMap indicated by the RefObjectMap. Populating the flipped Subject and Object templates is the same as if the RefObjectMap's TriplesMap was actually the Subject. The subject and object are just swapped on output.

R2RML RDF Generation The processor supports RDF output in N-Triples (or N-Quads) as a straightforward output of the TriplesMaps evaluation. What is more interesting is that this evaluation order enables specifying a hierarchical structure to the output. It is worth noting that JSON-LD is often serialized as flat objects and requires a special framing algorithm to create nested JSON-LD. Instead, Karma automatically generates a frame from the KR2RML Mapping to output nested JSON-LD. It can also generate Avro schemas and data and is easily extendable to any other hierarchical format like Protocol Buffers or Turtle.

Evaluating PredicateObjectMaps for each TriplesMap results in a JSON object for each generated subject and a scalar or an array depending on the cardinality of the template evaluation of the ObjectMaps. Because of the output order, when generating RDF for any TriplesMap with a RefObjectMap that has not been flipped, the objects' JSON objects will have already been outputted, so they can be nested and denormalized into the new JSON object. This recursively allows nesting all the way up until the root, which is shown in the output below.

In this example, the user indicates to the processor that the root is Organization. Any cycles are prevented by the flipped RefObjectMaps, so instead of nesting JSON objects, the URI is added. To finish formatting, a JSON-LD context can also be inferred from the R2RML mapping prefixes or can be provided.

```

1 {
2   "@context": "http://ex.com/contexts/iswc2015_json-context.json",
3   "location": [
4     {"address": {
5       "streetAddress": "4676 Admiralty Way Suite 1001",
6       "addressLocality": " Marina Del Rey", "postalCode": "90292",
7       "addressRegion": "CA", "a": "PostalAddress"
8     }},

```

```

9     "name": "ISI - West", "a": "Place", "uri":
      "isi-location:ISI-West"},
10    {"address": {
11      "streetAddress": "3811 North Fairfax Drive Suite 200",
12      "addressLocality": " Arlington", "postalCode": "22203",
13      "addressRegion": "VA", "a": "PostalAddress"
14    }},
15    "name": "ISI - East", "a": "Place", "uri":
      "isi-location:/ISI-East"}
16  ],
17  "name": "Information Sciences Institute", "a": "Organization",
18  "employee": [
19    {"name": "Knoblock, Craig", "a": "Person",
20     "uri": "isi-employee:Knoblock/Craig",
21     "jobTitle": ["Research Professor", "Director"],
22     "worksFor": "isi:company/InformationSciencesInstitute"},
23    {"name": "Slepicka, Jason", "a": "Person",
24     "uri": "isi-employee:Slepicka/Jason",
25     "jobTitle": ["Graduate Student", " Research Assistant"],
26     "worksFor": "isi:company/InformationSciencesInstitute"}
27  ],
28  "uri": "isi:company/InformationSciencesInstitute"
29  }

```

7 Related Work

Approaches exist for semantically mapping relational databases like D2R [2] but cannot address messy data nor easily extend to support new formats. RML [3] was proposed as an extension to R2RML to support heterogeneous, and often hierarchical, sources, e.g. XML and JSON, but its support for additional data types requires a custom processor for extracting values from each data type. RML tries to deal with the ambiguity that surrounds identifying subjects in hierarchical sources by introducing an `rml:iterator` pattern. Iterators benefit from their precision but are an additional step that can often be inferred by using column affinities. When we cannot infer them, however, we either require a transformation or end up generating a subject for each tuple in the cartesian product of the corresponding TriplesMap's PredicateObjectMaps that map to columns. XR2RML [8] is an extension of both RML and R2RML that supports NoSQL document stores. It agrees with our approach to leave format specific query language specification out of the language but still adopts the iterator pattern and also lacks supports for transformations. Mashroom [14], on the other hand, takes a similar approach with the Nested Relational Model for integrating and transforming hierarchical data, but it does have support for semantics nor a standard for the mapping.

8 Conclusions and Future Direction

Developing scalable solutions for data cleaning, transformation, and semantic modeling becomes more important every day as the amount of data available grows and its sources diversify. This alternative interpretation of R2RML and its processor are available under the Apache 2.0 license and have been embedded in Apache Hadoop and Apache Storm to generate billions of triples and billions of JSON documents in both a batch and streaming fashion and can be extended to consume any hierarchical format. It will soon be available for Spark and has well defined interfaces for adding new input and output formats.

References

1. APACHE SOFTWARE FOUNDATION. Avro. Version 1.7.6, 2014-01-22, <https://avro.apache.org>.
2. BIZER, C., AND CYGANIAK, R. D2r server-publishing relational databases on the semantic web. In *Poster at the 5th International Semantic Web Conference (2006)*, pp. 294–309.
3. DIMOU, A., SANDE, M. V., COLPAERT, P., MANNENS, E., AND DE WALLE, R. V. Extending r2rml to a source-independent mapping language for rdf. In *International Semantic Web Conference (Posters) (2013)*, vol. 1035, pp. 237–240.
4. FERNÁNDEZ, J. D., MARTÍNEZ-PRIETO, M. A., GUTIÉRREZ, C., POLLERES, A., AND ARIAS, M. Binary rdf representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web 19* (2013), 22–41.
5. GOOGLE INC. Protocol buffers. Version 2.6.1, 2014-10-20, <https://developers.google.com/protocol-buffers/>.
6. KNOBLOCK, C. A., SZEKELY, P., AMBITE, J. L., GUPTA, S., GOEL, A., MUSLEA, M., LERMAN, K., TAHERIYAN, M., AND MALLICK, P. Semi-automatically mapping structured sources into the semantic web. In *Proceedings of the Extended Semantic Web Conference (Crete, Greece, 2012)*.
7. MAKINOCHI, A. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of the 3rd VLDB Conference (1977)*, Citeseer, pp. 445–453.
8. MICHEL, F., DJIMENOU, L., FARON-ZUCKER, C., AND MONTAGNAT, J. Translation of relational and non-relational databases into rdf with xr2rml. In *11th Web Information Systems and Technologies (WEBIST) (2015)*.
9. MONGODB INC. Bson. Version 1.0.0, 2014-11-18, <http://bsonspec.org/>.
10. RAMAN, V., AND HELLERSTEIN, J. M. Potter’s wheel: An interactive data cleaning system. In *VLDB (2001)*, vol. 1, pp. 381–390.
11. SPORNY, M., KELLOGG, G., LANTHALER, M., GROUP, W. R. W., ET AL. Json-ld 1.0 json-based serialization for linked data. *W3C Working Draft* (2014). <http://www.w3.org/TR/json-ld/>.
12. SUNDARA, S., CYGANIAK, R., AND DAS, S. R2RML: RDB to RDF mapping language. W3C recommendation, W3C, Sept. 2012.
13. TYPESAFE INC. Hocon. Version 1.3.0, 2015-05-08, <https://github.com/typesafehub/config/blob/v1.3.0/HOCON.md>.
14. WANG, G., YANG, S., AND HAN, Y. Mashroom: End-user mashup programming using nested tables. In *Proceedings of the 18th International Conference on World Wide Web (New York, NY, USA, 2009)*, WWW ’09, ACM, pp. 861–870.