KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

DE NAYER (SINT-KATELIJNE-WAVER) CAMPUS

Inverting knowledge graphs back to raw data

How can we leverage the rules we use to construct knowledge graphs to do the inverse?

Tijs VAN KAMPEN

Promotor: Prof dr. ir. Anastasia Dimou

Master thesis submitted to obtain the degree of Master of Science in the engineering technology: Electronics-ICT

academic year 2023 - 2024

Abstract

Knowledge graphs are gaining traction nowadays and more and more companies use them, such as Amazon, Bosch, IKEA, Facebook, Google, LinkedIn, SIEMENS, Zalando, etc. Most knowledge graphs are nowadays constructed from other heterogeneous data sources, such as tables in relational databases, data in XML files, or JSON format derived from a Web API. While the construction of knowledge graphs from heterogeneous data has been thoroughly investigated so far, the inverse, namely constructing raw data from knowledge graphs hasn't been explored in depth yet.

In this thesis, we propose a method to invert knowledge graphs back to raw data. We will use the same rules used to construct the knowledge graph to do the inverse. Our method is split into two parts. First, a template is created by inverting the value retrieval of the mappings. For each supported source file this requires a tailored implementation. Secondly, the data is retrieved from the knowledge graph by using the mappings to generate a query for each iterator in the mappings. In the end, both these parts are combined, putting the data into the template.

Access to the source can be given upon request at tijs.vankampen@student.kuleuven.be.

For this thesis, generative AI is used for assistance with the coding of the implementation, and rewording of sentences in the paper for readability.

Contents

Chapter 1

Introduction

The earliest academic definition of a knowledge graph can be found in a 1974 article as

> A mathematical structure with vertices as knowledge units connected by edges that represent the prerequisite relation (Marchi and Miguel, 1974; Bergman, 2019)

The idea of expressing knowledge in a graph structure predates even this definition, with the concept of semantic networks (Richens, 1956). However, the term knowledge graph only became well-known after Google announced they were using a knowledge graph to enhance their search engine in 2012 (Singhal, 2012). Knowledge graphs are used to make search engines, chatbots, question-answering systems, etc more intelligent by injecting knowledge into them (Ji et al., 2022).

A knowledge graph consists of many connected nodes, where each node is either an entity or a literal. These nodes are connected by edges, where each edge defines a relation between two nodes. RDF is a framework often used to represent knowledge graphs, it uses subject-predicate-object triples to represent the nodes and their edges. Every node is either an URI, a blank node, or a literal while the edges are URIs. This triple: `http://example.com/John_Doe http://schema.org/givenName "John" .` would represent the fact that the entity `John Doe` has the first name `John`. Often the predicates are chosen from an ontology/vocabulary, such as schema.org or FOAF. This allows for more interoperability between knowledge graphs, as the same predicates are used to represent the same concepts.

These knowledge graphs are constructed by extracting information from various sources, both unstructured sources such as text (using natural language processing) and (semi-)structured sources such as databases, CSV, XML, JSON, RDF (using mapping languages). Many mapping languages exist, differing in the way of defining the rules and the target source file format. Some mapping languages use the turtle syntax, while others provide their custom syntax, and others repurpose existing languages like SPARQL or ShEx. (Van Assche et al., 2023). Some languages are specific to a single source format, such as R2RML(turtle format) (Das et al., 2012) for relational databases, XSPARQL(XQuery format) (Bischof et al., 2012) for XML. Others can process multiple formats, such as RML (turtle) (Dimou et al., 2014), D-REPR (YAML) (Vu et al., 2019), xR2RML (turtle) (Michel et al., 2015), etc. These can handle mapping from multiple sources in different formats.

To achieve this these mapping languages use a declarative approach where the user specifies rules describing the desired output knowledge graph, the mapping rules. The implementation then takes care of the logic and transformations behind the mapping. Two ways of mapping exist, materialization and virtualization. Materialization constructs the knowledge graph as a file, which can be loaded into a triple store. Virtualization does not generate the knowledge graph as a file, but instead exposes a virtual knowledge graph, which can be queried as if it were a real knowledge graph. (Calvanese et al., 2017).

Creating these mapping rules is often done by hand. Some tools make creating these mappings easier, like RMLEditor (Heyvaert et al., 2018b) which exposes a visual editor, and YARRRML (Heyvaert et al., 2018a) which allows users to create rules in the user-friendly YAML which are then compiled to RML rules. Alternatively, tools are starting to be created for the automatic generation of mapping rules from e.g. SHACL.

Retrieving data from a knowledge graph, for consumption by other programs, is done by querying the knowledge graph using SPARQL (Seaborne and Prud'hommeaux, 2008) for tabular data and XSPARQL (Bischof et al., 2012) or XSLT for XML. XSPARQL is the only language that can both lift and lower, but the mappings for lifting and lowering differ.

A knowledge graph cannot be converted back to the original data format using the same rules we created it with. As a result any changes we make to the data are hard to propagate back. We can not update, expand, or improve the original data using e.g. knowledge graph refining nor can we apply changes to a virtual knowledge graph to change the original data.

In this work, we seek to answer the question: *How can we extend an existing system like RML or create a new system to construct raw data from knowledge graphs?* We choose to extend the Morph-KGC implementation (Arenas-Guerrero et al., 2022) of RML (Dimou et al., 2014) as RML's end-to-end (from file to knowledge graph) characteristics make it a good candidate for this task. To answer the main research question we need to answer the following sub-questions:

RQ1 How can we construct the schema of the original data from the mapping rules?

- We will study each type of source format, as each format has its challenges.

RQ2 How can we populate the schema with data from the knowledge graph?

- We will study how we can best retrieve the data, trying different approaches.

1.1 Thesis outline

The aim of this thesis is to explore the possibility of inverting knowledge graphs back to their original data format using RML mapping rules. To achieve this we will take a closer look at the technologies used like RDF, SPARQL, and RML in chapter 2. In chapter 3 a closer look will be taken at our implementation of the inversion algorithm. We will look at the algorithm itself, and the implementation details. In chapter 4 an evaluation our implementation using various benchmarks will be done. For basic testing, we use a subset of the RML test cases, which are designed to test the conformance of tools to the RML specification. For more advanced testing we will use various benchmarks simulating real-life use cases like LUBM4OBDA, GTFS-Madrid-Bench, and SDM-Genomic-dataset.

Finally in chapter 5 we will conclude this thesis, and look at possible future work.

Chapter 2

Related work

In this chapter, we discuss the various technologies related to this thesis. We begin by discussing the semantic web and build from there to technologies used within its ecosystem like RDF, SPARQL, and mapping languages. We finish by discussing the current state of the art in updating or creating the original data source from a knowledge graph.

2.1 Semantic Web

Tim Berneers-Lee envisioned a version of the web that would also be understandable by machines, and thus the semantic web was born. It is not designed as a separate entity to the web, but instead as an extension, mostly hidden for normal humans. It is designed mostly with existing technologies like XML(including HTML, being a superset of it), URI and RDF. Even ontologies, a key component of the semantic web, are not a new concept but rather co-opted from the field of philosophy. (Berners-Lee et al., 2001)

2.2 RDF

RDF was originally designed as a data model for metadata but has since been extended to be a general-purpose framework for graph data. RDF is a directed graph, where the nodes represent entities, and the edges represent relations between these entities. This graph is built up from triples, which connect a subject and an object using a predicate as shown in figure 2.1.

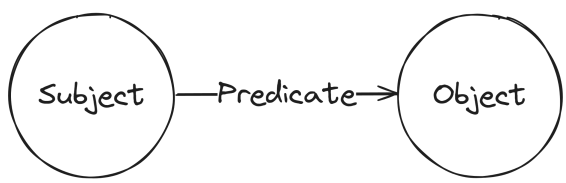

Figure 2.1: An RDF triple

Subject	Predicate	Object
http://example.com/De_Nayer	https://schema.org/location	http://example.com/Sint_Katelijne_Waver
http://example.com/r0785695	https://schema.org/givenName	"Tijs"

Figure 2.2: Example of RDF triples

Subject	Predicate	Object
ex:student/r0785695	schema:address	_:addrr0785695
_:addrr0785695	schema:postalCode	"2800"^^xsd:integer
_:addrr0785695	schema:streetAddress	"Gentsesteenweg XXXX"@nl

Figure 2.3: Example of a blank node and prefix notation

The subject must always be an entity, which can either be represented by an URI or be a blank node. The predicate must be an URI, and the object can be either an URI, a blank node, or a literal. (Manola and Miller, 2004)

An URI is a unique identifier for a resource on the web. Unlike a normal URL, it does not have to point to a network location, but can also be used to identify a person, a location, a concept, etc. (Manola and Miller, 2004). In RDF the URI is purely used for identifying resources. As such, unlike in HTML where certain conventions are expected, there are no conventions for URIs in RDF. An example of this can be seen in figure 2.2, the example subjects share the same domain but this does not imply that they are closely related, or even related at all. URIs are extended to IRIs to allow for a wider range of characters. Except for allowing Unicode characters, IRIs are identical to URIs so little distinction is made between the two in this thesis.

A blank node is a node that is not identified by a URI. It is used to represent an anonymous resource that can't be or has no reason to be uniquely identified. For example, the address of student r0785695 in figure 2.3 is only pertinent to the student and thus does not need to be uniquely identified. A blank node is serialized as `_:name`, where name is a unique identifier for the blank node. This identifier is only unique within the document, and thus can't be used to refer to the blank node from outside the document. (Manola and Miller, 2004)

A literal is a value, e.g. a string, integer, or date. This value can be typed, e.g. a string can be typed as a date, or untyped. A string can also have a language tag, which is used to indicate the language of the literal.

RDF is only a framework, and as such does not define any serialization syntax. There are however a few common serialization standards for example RDF/XML, Turtle, N-Triples, and JSON-LD.

2.2.1 Turtle

Turtle, or Terse RDF Triple Language, is a human-readable serialization format for RDF. It is the most used serialization format for RDF, and is used in many tools and specifications. In its simplest form turtle consists of triple statements, sequences of subject-predicate-object separated by spaces and terminated by a dot. An example of this can be seen in listing 2.1. This is very verbose, but Turtle offers many features to make it more concise. Below is a list of some of these features and, if possible, how they can be used to make the example more concise.

- **Prefix notation** allows us to shorten URIs by defining a prefix.
  - Using `@prefix schema: <https://schema.org/>` allows us to shorten `https://schema.org/Person` to `schema:Person`
- **Base prefix** allows us to shorten URIs by defining a base URI.
  - Using `@base <http://example.com/>` allows us to shorten `http://example.com/r0785695` to `r0785695`
- **Predicate lists** allow us to shorten multiple triples with the same subject to a list of predicates.
  - Our example only has two subjects, we can split their predicates with `;` instead of repeating the subject.
- **Object lists** allow us to shorten multiple triples with the same subject and predicate to a list of objects.
  - `r0785695` is both a Person and a Student, so we can split the objects with `,` instead of repeating the subject and predicate.
- **Literals** allow identifying values, e.g. strings, integers, dates, etc. with a datatype or language tag.
- **Blank nodes** allow us to define anonymous resources by using the `_:` prefix.
  - The address of `r0785695` is only relevant to `r0785695`, so we can define it as a blank node instead of using a URI, this shortens `<http://example.com/addrr0785695>` to `_:addrr0785695`. While not exactly the same, functionally it is equivalent as we do not expect addresses to be addressed outside of the context of a person.
- **Unlabeled blank nodes** allow us to define anonymous resources without a unique identifier by using the [] notation instead of `_:name`.
  - As we do not need to refer to `_:addrr0785695` from outside `r0785695`, we can use an unlabeled blank node and include it in `r0785695` instead of a labeled blank node.
- **Collections** allow us to define a list of blank nodes by using the () notation.

The example in listing 2.1 can be rewritten using these features, as shown in listing 2.2.

```
<http://example.com/r0785695> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org/Person> .
<http://example.com/r0785695> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org/Student> .
<http://example.com/r0785695> <http://schema.org/givenName> "Tijs" .
<http://example.com/r0785695> <http://schema.org/familyName> "Van Kampen" .
<http://example.com/r0785695> <http://schema.org/address> <http://example.com/addrr0785695> .
<http://example.com/addrr0785695> <http://schema.org/postalCode> "2800"^^<http://www.w3.org/2001/XMLSchema#integer> .
<http://example.com/addrr0785695> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org/PostalAddress> .
```

```
<http://example.com/addrr0785695> <http://schema.org/streetAddress> "
    Gentsesteenweg XXXX"@nl .
<http://example.com/addrr0785695> <http://schema.org/addressCountry> "
    Belgium" .
```

Listing 2.1: Basic naive turtle document

```
@prefix schema: <https://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@base <http://example.com/> .
<r0785695> a schema:Person, schema:Student ;
    schema:givenName "Tijs" ;
    schema:familyName "Van Kampen" ;
    schema:address [
        a schema:PostalAddress ;
        schema:postalCode "2800"^^xsd:integer ;
        schema:streetAddress "Gentsesteenweg XXXX"@nl ;
        schema:addressCountry "Belgium"
    ] .
```

Listing 2.2: Basic turtle document using turtle features

2.3 SPARQL

SPARQL Protocol And RDF Query Language (SPARQL) is the W3C standard query language for RDF. It is the main way to query RDF data and shows many similarities to SQL. SPARQL queries mostly consist of a pattern of triples, which are matched against the RDF graph, a basic example can be found in listing 2.3. Querying is very feature-rich, with support for aggregation, subqueries, negation, regex, string manipulation, etc. It also supports different return types, federated queries, entailment, etc (Harris and Seaborne, 2013). Aside from the query protocol it also defines the graph store protocol, which can be used to manipulate graph databases directly (Aranda et al., 2013).

```
PREFIX schema: <https://schema.org/>
SELECT ?name ?address
WHERE {
    ?student schema:givenName ?name .
    ?student schema:address ?address .
}
```

Listing 2.3: Example of a basic SPARQL SELECT query

2.4 Mapping languages

Mapping languages are used to define a mapping between a source and a target. The target in the context of linked data is of course RDF, with the source being any structured data source. Some mapping languages exist for a single source, e.g. Relational Database to RDF Mapping Language (R2RML) for relational databases, a query language combining XQuery and SPARQL (XSPARQL) for XML, etc. Others are more general purpose, e.g. RDF Mapping Language (RML) and Data to RDF Mapping Language (D2RML). We will discuss both R2RML and RML in more detail, as one extends the other. RML we will discuss because it is one of the more feature-rich general mapping languages, and it is the mapping language we will use in our implementation. R2RML we discuss because it is the most widely used mapping language, as it supports virtualization *ontop* of databases. Most RML implementations also support R2RML, as RML is a nearly superset of R2RML.

2.4.1 R2RML

Relational Database to RDF Mapping Language (R2RML) is a mapping language for mapping relational databases to RDF. As opposed to Direct Mapping (DM), which results in a direct mapping from the relational database to RDF without any changes to structure or naming, R2RML allows for more flexibility. R2RML mappings consist of zero or more TriplesMaps, which are used to map a table to RDF. A TriplesMap consists of a logical table, a subject map, and one or more predicate object maps (POMs).

The logical table is used to define the table that is being mapped, with each row in the table being mapped to a subject and its corresponding POMs. It is possible to create a view of a table by using a SQL query, and then map this view. This allows for more complex mappings, e.g. mapping a join of two tables or a computed column.

Each of the SubjectMap, PredicateMap, ObjectMap, (and GraphMap) is a subclass of TermMap, which is a function that generates an RDF term. The map type can be constant, template, or column. The resulting term is then used as the subject, predicate, object, or graph of the triple. The termType of the map determines the type of the term, which can be IRI, blank node, or literal. If the termType is literal, optionally the datatype or language can be added. Following the RDF specification, not all combinations of termType and map are possible, this is shown in table 2.1. The object map has an additional subclass, a reference object map, in which we refer to another TriplesMap. Using a reference map we can map a foreign key to the subject of another TriplesMap, with a join condition. (Cyganiak et al., 2012)

TermType	Subject	Predicate	Object	Graph
IRI	✓	✓	✓	✓
Blank node	✓	✗	✓	✓
Literal	✗	✗	✓	✗

Table 2.1 Possible combinations of TermType and Map type

R2RML		RML	
Logical Table (relational database)	rr:logicalTable	Logical Source	rml:logicalSource
Table Name	rr:tablename	URI (pointing to the source)	rml:source
column	rr:column	reference	rml:reference
(SQL)	rr:SQLQuery	Reference Formulation	rml:referenceFormulation
per row iteration		defined iterator	rml:iterator

Table 2.2 Differences between R2RML and RML

A constant value is a fixed value, e.g. a URI or a string. A template is a string with placeholders, which are replaced by values from the logical row. A column is the value of a column in the logical row.

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ex: <http://example.com/ns#>.

<#TriplesMap1>
    rr:logicalTable [ rr:tableName "EMP" ];
    rr:subjectMap [
        rr:template "http://data.example.com/employee/{EMPNO}";
        rr:class ex:Employee;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:id;
        rr:objectMap [ rr:column "EMPNO"; rr:datatype xsd:positiveInteger ].
    ].
```

Listing 2.4: Example of an R2RML mapping

2.4.2 RML

RDF Mapping Language (RML) is a mapping language for mapping any (semi-)structured data source to RDF. It is a generalization of R2RML and as such supports all the features of R2RML. It extends R2RML by extending database-specific features to make them more general. The differences in usage can be seen in table 2.2. (Meester et al., 2022)

RML uses the same structure as R2RML, with TriplesMaps consisting of a logical source, a subject map, and zero or more POMs. The changes it has all relate to the logical source. Whereas in R2RML the source is always a database, from which we select a table or view, in RML the source can be one of many different source types like XML, JSON, CSV, etc. Where in R2RML we simply iterate over the rows of a table, in RML we can have a source without an explicit iteration pattern, and as such we need to define an iterator.

2.5 State of the art

The state of the art in updating or creating the data source from a knowledge graph can be split in two categories, depending on the methodology used. The first methodology applies to virtualization, where the data is exposed as a virtual knowledge graph over the source data. The other methodology is for materialized knowledge graphs, where the knowledge graph is created as a file that can be loaded into a triple store. We will discuss the state of the art in both methodologies, concluding with the relevance of this work.

2.5.1 Virtualization

Virtualization is the process of exposing a virtual knowledge graph over the source data. This virtual knowledge graph can be queried as if it were a real knowledge graph. To achieve this mappings are used to translate queries over the knowledge graph to queries over the source data.

This is most commonly used to expose a database as a virtual knowledge graph. This way an organization can expose a knowledge graph without having to completely transition to a new system. Most implementations of virtualization layers are read-only though, as the translation of SELECT queries is relatively easy but translating INSERT, DELETE or DELETE/INSERT (update) queries is much less straightforward, or even impossible in many cases. Though propagating changes trough the virtualization layer to the source data could be a big part of the linked data lifecycle, related work on this is scarce. In both ”SPARQL Update queries over R2RML mapped data sources” (Unbehauen and Martin, 2017) and ”Practical Update Management in Ontology-Based Data Access” (De Giacomo et al., 2017) the authors propose a similar way of handling updates. Compatible updates are propagated to the source data, while incompatible updates are held in an ’overflow’ triple store. Further changes may make the incompatible updates compatible, at which point they too are propagated to the source data. Larger changes affecting the general structure of the data are not stored, but instead the mapping is updated to reflect the new structure. The handling of updates is shown in figure 2.4.

2.5.2 Materialization

Materialization is the process of creating a knowledge graph as a file that can be loaded into a triple store. The knowledge graph is then loaded into a dedicated triple store for querying. This method benefits from increased performance, at the cost of having the knowledge graph not in sync with the source data.

Materialization allows for a wider range of source formats, as the knowledge graph can be created from any structured data source. This is especially useful when the source data is not easily queryable, e.g. when the source data is a CSV, XML, or JSON file. For knowledge graphs created from structured data sources, use cases exist for propagating changes back to the original data source. This is not done using a direct update (as the source data is not directly connected to the knowledge graph), but by creating a new version of the source data. This process is called

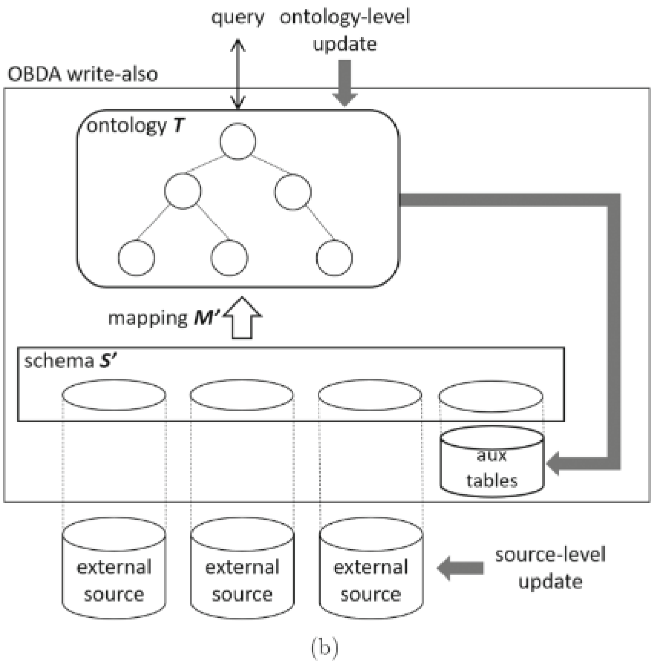

Figure 2.4: Propagating changes in a virtualization layer

lowering. Below we discuss some of the state of the art in lowering.

2.5.2.1 XSPARQL

XSPARQL is a mapping language that allows for the transformation of XML to and back from RDF. It expands XQUERY with SPARQL-like syntax, structure and features. Its predecessors would do lifting by querying the source XML using XQuery to transform it into the XML serialization of RDF, while lowering was done using XSLT to do the inverse. Using its combined vocabulary XSPARQL simplifies lifting and lowering, using a single language for both and having the ability to target the turtle serialization (Polleres et al., 2009). An example lifting and lowering query can be found in listing 2.5 and listing 2.6 respectively.

```
declare namespace foaf="http://xmlns.com/foaf/0.1/";
declare namespace rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#";
let $persons := //*[@name or ../knows]
return

for $p in $persons
let $n := if( $p[@name] )
            then $p/@name else $p
let $id := count($p/preceding::*)
            +count($p/ancestor::*)
where
    not(exists($p/following::*[
```

```
        @name=$n or data(.)=$n]))
construct {
    _:b{$id} a foaf:Person;
                foaf:name {data($n)}.
    {
        for $k in $persons
        let $kn := if( $k[@name] )
                    then $k/@name else $k
        let $kid := count($k/preceding::*)
                    +count($k/ancestor::*)
        where
            $kn = data(//*[@name=$n]/knows) and
            not(exists($kn/../following::*[
                @name=$kn or data(.)=$kn]))
        construct {
        _:b{$id} foaf:knows _:b{$kid}.
        _:b{$kid} a foaf:Person.
        }
    }
}
```

Listing 2.5: Example of XSPARQL lifting

```
<relations>{
    for $Person $Name from <relations.rdf>
    where {$Person foaf:name $Name}
    order by $Name
    return
        <person name="{$Name}">{
            for $FName from <relations.rdf>
            where {
                $Person foaf:knows $Friend.
                $Person foaf:name $Name.
                $Friend foaf:name $Fname. }
            return
            <knows>{$FName}</knows>
        }</person>
}</relations>
```

Listing 2.6: Example of XSPARQL lowering

2.5.2.2 POSER: A Semantic Payload Lowering Service (Spieldenner., 2022)

POSER(Payload lOwering SERvice) is a service that lowers RDF to JSON. To achieve this a mapping is created in two parts: first the source patterns are defined from which to extract the data, then the json structure is defined. The mapping is written in turtle, using a custom json ontology describing the json structure. A proof of concept implementation was made, but never got out of the prototype phase. It also only handles direct literal types, more complex composite values that are generated from templates are not supported. An example mapping is shown in listing 2.7.

```
@prefix ctd: <http://connectd.api/> .
@prefix onto: <http://ontodm.com/OntoDT#> .
@prefix iots: <http://iotschema.org/> .
@prefix json: <http://some.json.ontology/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix time: <https://www.w3.org/TR/2020/CR-owl-time-20200326/> .

# Which inputs to expect and to start mapping from
json:InputDataType {
    json:EntryPoint a iots:TimeSeries;
        iots:providesTemperatureData iots:TemperatureData;
        iots:providesTimeData iots:TimeData .

    iots:TemperatureData iots:numberDataType iots:Number .
    iots:TimeData  time:dateTime iots:Number .
}

#Semantic description of the json objects to be found in the expected API

json:ApiDescription {
    ctd:JsonModel json:hasRoot ctd:Node .

    ctd:TemperatureValue a json:Number ;
        json:key "value"^^xsd:string ;
        json:dataType iots:TemperatureData .

    ctd:TimeStamp a json:String ;
        json:key "timestamp"^^xsd:string ;
        json:dataType iots:TimeData .

    ctd:Node a json:Object;
        json:key "node"^^xsd:string ;
        json:value ctd:TimeStamp, ctd:TemperatureValue ;
        json:dataType iots:TimeSeries .

    ctd:Edges a json:Array      ;
        json:key "edges"^^xsd:string ;
        json:value ctd:Node .
}
```

Listing 2.7: Example of a POSER mapping

2.5.3 Relevance

As shown in this section the state of the art in updating the source in virtualization is pretty mature, only limited by fundamental limitations. It is however limited to databases. To work with other (semi-)structured data sources materialization is needed. For materialization the state of the art is much more limited. Though methods exist to lower RDF to other formats, each method is intimately linked to a source type. The mappings are also unidirectional, even XSPARQL which offers both lifting and lowering requires a separate mapping for each.

Our work expands RML, which can map from any structured data source to RDF, with the ability to lower the RDF back to the original source. We use a single mapping to do both lifting and lowering, as information on where to find the data during lifting can also be used to find where to put the data during lowering. This makes our work unique in the field of lowering RDF to other formats.

Chapter 3

Implementation

In this chapter we present the design, limitation and implementation of the proposed system. The design and any of its components are not bound to any specific technology, as such they can be implemented using any programming language, framework or mapping language. Using a mapping outside its intended purpose has some limitations, we present these fundamental issues in their own section. We go into more detail about other issues where relevant later. For the implementation we first present the general setup of the environment. We then present the implementation of the data retrieval. Lastly we present some implementations of template creation and filling.

3.1 System Design

The system we propose is a pipeline that takes as input a knowledge graph and set of mapping rules and outputs the source files from which it would have been constructed. It has the advantage of being very modular, consisting of just data retrieval and template creation/filling. Depending on the mapping rules and config files different modules will be used. The graph source will decide the data retrieval strategy while the output file will determine the templating engine to be used. An overview of the design can be seen in figure 3.1.

3.2 Limitations

We observe three types of limitation inherent to the system. As they stem from the very basis of the system, some are impossible to overcome. The first two are caused by irreversible processing, transforming the data in such a way that it is impossible to reconstruct the original data. The third is caused by the design of nested data structures, where the method of data access can limit the ability to reconstruct the source.

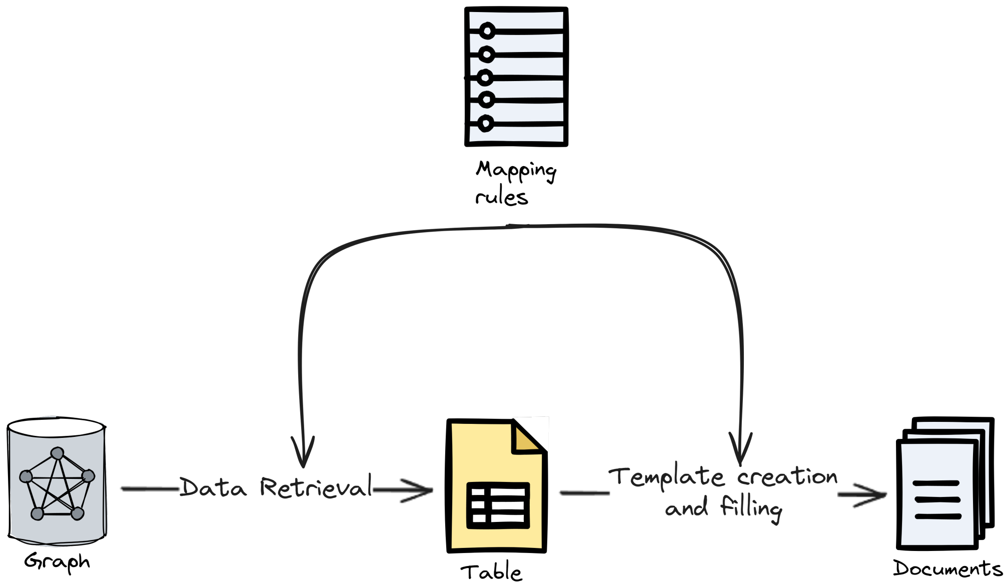

Figure 3.1: Design of the proposed system

3.2.1 Irreversible source transformation

The first limitation is caused by the transformation of the source data before mapping. The most common irreversible transformation of a source in RML is a view over a database. While some views could be reversible, like a simple projection, many are not. To be able fully recreate the source it must be passed on in full to the mapper, without any aggregation, filtering, or other irreversible transformations. The mapping must also use all the data in de table. An example irreversible source mapping on a database can be found in listing 3.1. This limitation is not limited to database sources. Query languages working on data formats like JSON or XML also have the capability to filter or select data, making full reconstruction impossible. Some examples of this can be found in table 3.1.

XPath	JSONPath	Description
//book[last()]	$..book[(@.length-1)] $..book[-1:]	the last book in order.
//book[position()¡3]	$..book[0,1] $..book[:2]	the first two books
//book[isbn]	$..book[?(@.isbn)]	filter all books with isbn number
//book[price<10]	$..book[?(@.price<10)]	filter all books cheapier than 10

Table 3.1 Examples of irreversible source access in XPath and JSONPath

```
<TriplesMap1>
a rr:TriplesMap;

 rr:logicalTable [ rr:sqlQuery """
     SELECT "Name", COUNT("Sport")
     FROM "Student"
     """ ];

 rr:subjectMap [ rr:template "http://example.com/resource/student_{\"Name\"}"; ];

 rr:predicateObjectMap
 [
     rr:predicate    foaf:name ;
     rr:objectMap    [ rr:column "\"Name\""; ];
 ];

 rr:predicateObjectMap
 [
               rr:predicate    ex:numSport ;
               rr:objectMap    [ rr:column "SPORTCOUNT"; ];
 ];
 .
```

Listing 3.1: Irreversible source mapping from R2RML Test cases

3.2.2 Irreversible data transformation

The second limitation is caused by the transformation of data during mapping. RML allows for the transformation of data using functions. These functions offer a lot of functionality, from string functions (concat, replace, etc.) to math to custom (self-defined) functions. The result of these functions can either be stored in the graph as a value or used to join data. This transformation is often irreversible, with information from the original data lost. Take for example `grel:toUpperCase`: this function transforms the string to an all uppercase string, here we lose information about the original case.

3.2.3 Unreconstructable data structures

The third limitation is caused by the design of nested data structures. To access the data multiple paths are valid to retrieve the data. The examples in table 3.1 also apply here, as the examples use 'recursive descent' to access the data. In the example any element with the name 'book' is taken from the JSON, no matter the depth. Alternatively a direct path could be written out like `$.store.book.title`. Another valid option for this particular dataset would be to use wildcard operators like `$.*.*.title`. Only a direct path gives us enough information to reconstruct the source file using the mapping rules. A recursive descent gives no indications as to the depth of the results, and using wildcards gives no information about the naming of the elements. While a 'suggested' source file could still be reconstructed, and it would still work with the mapping rules, it would almost certainly not share the same structure as the original source.

3.3 Setup

We choose to make our implementation in Python as it both has an extensive set of libraries and is well suited for rapid development. We use Morph-KGC to process the mapping rules for its ease of use, being well made and its use of the pandas library to represent the mapping rules. As the graph source we choose a standalone SPARQL endpoint on a triple store. For working with graph(RDF) files, we first upload them to a triple store.

First the endpoint is set up, if necessary. If the source is a file, it is uploaded to the endpoint using the SPARQL 1.1 Graph Store HTTP Protocol. In our setup this is a locally running the free version of GraphDB. The url of the repository is then wrapped using the SPARQLWrapper library and some settings are set.

Next we load the mapping rules from the mapping(RML) files using Morph-KGC's internal `retrieve_mappings` function. This function takes a config file (.ini format) as input, which specifies the location of the mapping files and various other settings which can be used to configure the behavior of the mapping. The mappings are returned as a pandas DataFrame which we enrich with various helper columns. An example of a single mapping rule (row in the DataFrame) can be found in listing 3.2. Marked in bold are the helper columns we add.

```
source_name: DataSource1
triples_map_id: #TM0
triples_map_type: http://w3id.org/rml/TriplesMap
logical_source_type: http://w3id.org/rml/source
logical_source_value: student.csv
iterator: nan
subject_map_type: http://w3id.org/rml/template
subject_map_value: http://example.com/{Name}
subject_references_template: http://example.com/([^\/]*)$
subject_references: ['Name']
subject_reference_count: 1
subject_termtype: http://w3id.org/rml/IRI
```

```
predicate_map_type: http://w3id.org/rml/constant
predicate_map_value: http://xmlns.com/foaf/0.1/name
predicate_references_template: None
predicate_references: []
predicate_reference_count: 0
object_map_type: http://w3id.org/rml/reference
object_map_value: Name
object_references_template: None
object_references: ['Name']
object_reference_count: 1
object_termtype: http://w3id.org/rml/Literal
object_datatype: nan
object_language: nan
graph_map_type: http://w3id.org/rml/constant
graph_map_value: http://w3id.org/rml/defaultGraph
subject_join_conditions: nan
object_join_conditions: nan
source_type: CSV
mapping_partition: 1-1-1-1
```

Listing 3.2: Example of a mapping rule in Morph-KGC

3.4 Retrieving the data

Retrieving the data is done by querying the SPARQL endpoint using an automatically generated query. All the required data from the source is retrieved at once. This does result in some degree of duplication for nested structures, but doing all processing and joining server-side is preferable. This sadly has a disadvantage of disjointed mappings currently resulting in a Cartesian product of the fragments. This is further discussed in subsection 3.4.2. The data is returned as a CSV-table which is then loaded into a pandas DataFrame for further processing. The only post processing done is the decoding of url-encoded strings where necessary.

3.4.1 Generating the queries

To generate the queries we first select the triple maps we want to use to generate the query. While constant values are always included, we can lighten the load on the server by reducing the amount and complexity of mapping rules we use. We design three operating modes:

- **Full**: All triple maps are used to generate the query. This is guaranteed to give predicatable results as it checks each instance of a mapped value.
- **Reduced**: All triple maps with a object map type of reference are used to generate the query. Only where necessary to retrieve all data template map types are used.
- **Minimal**: Only a single datapoint is used for each reference, with a preference for reference object map types.

We then generate the queries by translating the mapping rules into patterns. Each of the three map types, constant, reference, and template, generates a different pattern. For the constant map type, we know that it will always be present with a constant value, so we can simply add it as a triple pattern like `?s $predicate $object_value`. Both reference and template map types will not generate a triple during materialization if any of the references are not present during the generation. We take this into account by allowing for blank fields using the `optional` keyword. This does however bring some problems with it, as we do want to check if multiple references to the same value contain the same value. The optional keyword would just fail silently if we try to assign to the same output variable, regardless of the value. As such we add some more logic using a combination of `BIND` and `FILTER` to check if the values of all occurrences of a reference are the same. Reference maps are also easy to work with as they directly translate back to the source. An example of a basic query using only constant and reference maps can be found in listing 3.3.

```
SELECT DISTINCT ?Name ?ID
WHERE {
    ?s a foaf:Person .

    optional{
        ?s foaf:name ?Name .
    }
    optional{
        ?s ex:id ?ID .
    }
}
```

Listing 3.3: Simple query example

Template maps are more complex to work with as, though build in, they are effectively a (possibly irreversible) data transformation. The simplest operation, which will work regardless of the template is in fact irreversible, is to confirm the mapped value fits the template using regex like: `FILTER(regex(str(?s), "http://example.com/([^\\*)/([^\\/]*)$"))`. Taking this further we can use string manipulation to split the variable into the different references. For templates that use multiple values the characters splitting the two values should consist of "reserved characters" defined by rfc3986 (Berners-Lee et al., 2005), these are as follows: `:/?#[]@!$&'()*+,;=`. An example query using a template can be found in listing 3.4.

```
SELECT DISTINCT ?Name ?ID
WHERE {
    ?s a foaf:Person .
    FILTER(regex(str(?s), "http://example\\.com/([^\\/]*)/([^\\/]*)$")) .
    BIND(STRAFTER(str(?s), "http://example.com/") as ?temp) .
    BIND(STRBEFORE(str(?temp), "/") as ?ID)
    BIND(STRAFTER(?temp, "/") as ?Name)
```

```
    optional{
        ?s foaf:name ?Name .
    }
    optional{
        ?s ex:id ?ID .
    }
}
```

Listing 3.4: Template query example

The algorithm for generating the queries can be found in algorithm 1. While abstracting some details, this follows the flow of the implementation.

Algorithm 1 Generating the queries

```
Require: mapping_rules is a set of mapping rules
query_lines ← []
for all rule ∈ mapping_rules do
    object_encoded = rule.is_encoded()
    if rule.is_constant() then
        query_lines.append(rule.to_triple())
    else if rule.is_reference() then
        query_lines.append(rule.to_optional_triple(encode = object_encoded))
    else if rule.is_template() then
        query_lines.append(test_object_regex(rule))
        remainder ← rule['object_template']
        for all reference ∈ rule['object_references'] do
            query_lines.append(bind_next_partial(rule, reference))
            remainder ← next_segment(remainder)
            if remainder = "" then
                query_lines.append(rule.bind_last_slice())
            else
                query_lines.append(rule.bind_next_slice())
            end if
        end for
    end if
end for
query ← wrap_query_lines(query_lines)
```

3.4.2 Disjointed mappings

We generate a query to retrieve the whole source at once, this can lead to issues when subjects in the mapping has no path/link between them. The effect this creates is not unlike joining two tables in SQL without join conditions. For example, using the mapping listed in listing 3.6 we get the query in listing 3.6. When applied to the knowledge graph in listing 3.7 we get the result in listing 3.8 instead of the original source in listing 3.9. When converting the badly generated source back to the knowledge graph, we do get the same knowledge graph as the original as the duplicate data is ignored. The amount of duplicate data increases exponentially with the number of subjects.

This is solvable by analysing the mapping rules to look for disjointed mappings and split them into separate queries. At the templating stage we can then merge the results back together. This would solve the issue, but the reconstructed source is highly unlikely to be the same as the original source. An example for how the result would look can be found in listing 3.10.

```
<TriplesMap1> a rr:TriplesMap;
rml:logicalSource [
    rml:source "student_sport.csv";
    rml:referenceFormulation ql:CSV
];
rr:subjectMap [
    rr:template "http://example.com/{Student}";
    rr:class ex:Student
];
rr:predicateObjectMap [
    rr:predicate foaf:name ;
    rr:objectMap [
        rml:reference "Student"
    ]
].
```

```
<TriplesMap2> a rr:TriplesMap;
rml:logicalSource [
    rml:source "student_sport.csv";
    rml:referenceFormulation ql:CSV
];
rr:subjectMap [
    rr:template "http://example.com/{Sport}";
    rr:class ex:Sport
];
rr:predicateObjectMap [
    rr:predicate foaf:name ;
    rr:objectMap [
        rml:reference "Sport"
    ]
].
```

Listing 3.6: Bad join mapping

```
SELECT DISTINCT ?Student_name ?Sport
WHERE {
    ?s1 a ex:Student .
    optional{
        ?s1 foaf:name ?Student_name .
    }
    ?s2 a ex:Sport .
    optional{
        ?s2 foaf:name ?Sport .
    }
```

```
}
```

Listing 3.6: Bad join query (trimmed)

```
@prefix ex: <http://example.com/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:Venus a ex:Student ;
    foaf:name "Venus" .
ex:Tom a ex:Student ;
    foaf:name "Tom" .
ex:Tennis a ex:Sport ;
    foaf:name "Tennis" .
ex:Football a ex:Sport ;
    foaf:name "Football" .
```

Listing 3.7: Bad join knowledge graph

```
Student,Sport
Venus,Tennis
Venus,Football
Tom,Tennis
Tom,Football
```

Listing 3.8: Bad join result

```
Student,Sport
Venus,Tennis
Tom,Football
```

Listing 3.9: Bad join original source

```
Student,Sport
Venus,
Tom,
,Tennis
,Football
```

Listing 3.10: Better bad join result

3.5 Contructing the schema

Constructing the schema is done by reversing the mapping rules' source. We do this using the iterator and the mapping rule's references. RML supports many different types of sources and referenceFormulations. We will implement the CSV, xPath, and JSONPath referenceFormulations. Not every source is a file, so for query-based sources, we will generate the query output. In a later stage, we could look into taking it a step further, generating the actual source behind those intermediate results.

As each referenceFormulation has its reference syntax, we will have to tailor the implementation to each referenceFormulation. For the PoC, we only implement the CSV referenceFormulation.

3.5.1 CSV

The CSV referenceFormulation is the simplest of the three as it describes a simple two-dimensional table, with columns having the names of the references and rows being the iterated values. The example TriplesMap in listing 3.11 results in the CSV template in listing 3.12. Unlike the other referenceFormulations, CSV has no uncertainty in terms of structure.

```
<TriplesMap1> a rr:TriplesMap;

rml:logicalSource [
    rml:source "student.csv";
    rml:referenceFormulation ql:CSV
];

rr:subjectMap [
    rr:template "http://example.com/Student/{ID}/{Name}";
    rr:graph ex:PersonGraph ;
    rr:class foaf:Person
];

rr:predicateObjectMap [
    rr:predicate ex:id ;
    rr:objectMap [ rml:reference "ID" ]
];

rr:predicateObjectMap [
    rr:predicate foaf:name ;
    rr:objectMap [ rml:reference "Name" ]
].
```

Listing 3.11: Example mapping for a CSV file

```
ID,Name
<ID>,<Name>
```

Listing 3.12: Example CSV template

3.6 Applying the data to the schema

Generating the final output is done by iterating over the rows of the data and applying them to the schema. Each column of the data corresponds to a reference in the schema. A short version of the algorithm can be found in algorithm 2. For the PoC this approach is sufficient, even too complex as we can simply dump the query-result DataFrame to a CSV file. For more complex, possibly nested, sources we will have to adapt this algorithm.

Algorithm 2 Applying the data to a simple (non-nested) schema

Require: $schema$ is a schema
Require: $data$ is a DataFrame

```
output ← new_file
for all row ∈ data do
    output ← schema
    for all column ∈ row do
        schema.replace(column.name, column.value)
    end for
    output.write(schema)
end for
```

Chapter 4

Evaluation

In this chapter we will evaluate our implementation. We will do this by testing it again various datasets, comparing the expected results with the actual results. For each dataset we will go over our testing methodology and its results. For the PoC implementation we only test the RML test cases. The implementation is under constant development, as such we hope to be able to show better results in the presentation.

4.1 RML test cases

The RML test cases (Heyvaert et al., 2019) are a set of test cases to evaluate the conformity of an RML processor. Though these test cases are not a perfect match, they offer expected outputs for certain inputs and mapping rules, making them a good starting point for testing our implementation. The test cases are designed with edge cases in mind, making them a good set of test cases to test a mapper. Using them to test inversion, however, is stretching their purpose a bit. As such we have to filter out test cases that are expected to fail, as they do not produce a result. As the reading of the mappings is handled by Morph-KGC, some test cases that test things like alternative syntax look the same to us, providing little value.

In table 4.1 the results of the RML test cases can be found. The failed tests are divided into categories with colors according to the reason they failed. The red failures are caused by solvable issues in the implementation. Tests marked orange are flawed but solvable in some cases, `0002b-CSV` has disconnected mappings as discussed in section 3.4, it only passes the test because only a single row of data is mapped. In the other orange test, `0004a-CSV` the data would only be retrievable from a blank node identifier. The light-red failures are because of the limitations of the Morph-KGC library as it does not support reference-type subjects. Finally, the blue failures are impossible due to incompatibility with inversion. Many blue tests contain duplicated data, which gets removed when materializing.

Test	Status	Test	Status
0000-CSV	✓	0008a-CSV	✗
0001a-CSV	✓	0008b-CSV	✗
0001b-CSV	✓	0008c-CSV	✗
0002a-CSV	✓	0009a-CSV	✗
0002b-CSV	✗	0009b-CSV	✗
0003c-CSV	✓	0010a-CSV	✗
0004a-CSV	✓	0010b-CSV	✗
0005a-CSV	✗	0010c-CSV	✗
0006a-CSV	✗	0011b-CSV	✗
0007a-CSV	✗	0012a-CSV	✗
0007b-CSV	✗	0012b-CSV	✗
0007c-CSV	✗	0015a-CSV	✗
0007d-CSV	✗	0019a-CSV	✗
0007e-CSV	✓	0019b-CSV	✗
0007f-CSV	✗	0020a-CSV	✓
0007g-CSV	✗	0020b-CSV	✗

Table 4.1 Results of the RML test cases

Chapter 5

Roadmap

This thesis is far from done, as such no conclusion can be made for now. Here we discuss what will be done in the second semester.

5.1 Implementation

We will improve the implementation in various ways:

- Support for more referenceFormulations:
  - JSON-path
  - XPath
- Joins
- More robustness
- More in-depth hybrid approach between SPARQL and local processing
- Inverting query-based sources further by inverting the query.

5.2 Evaluation

The number of benchmarks used will be expanded to include: LUBM4OBDA, GTFS-Madrid-Bench, SDM-Genomic-dataset, and any other benchmark that has RML (or R2RML) mappings available. We will also look into creating a version of the RML test cases for inversion, as the ones designed to test the conformance of RML processors are not ideal for testing the inversion algorithm.

Bibliography

Aranda, C. B., Corby, O., and Das, S. (2013). SPARQL 1.1 overview. W3C recommendation, W3C. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/.

Arenas-Guerrero, J., Chaves-Fraga, D., Toledo, J., Pérez, M. S., and Corcho, O. (2022). Morph-KGC: Scalable knowledge graph materialization with mapping partitions. *Semantic Web*.

Bergman, M. K. (2019). A common sense view of knowledge graphs. *AI3:::Adaptive Information*, 1(1):1–1.

Berners-Lee, T., Fielding, R. T., and Masinter, L. M. (2005). Uniform Resource Identifier (URI): Generic Syntax. RFC 3986.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5).

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.

Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., and Xiao, G. (2017). Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8:471–487. 3.

Chortaras, A. and Stamou, G. (2018). D2rml: Integrating heterogeneous data and web services into custom rdf graphs. In *LDOW@WWW*.

Cyganiak, R., Sundara, S., and Das, S. (2012). R2RML: RDB to RDF mapping language. W3C recommendation, W3C. https://www.w3.org/TR/2012/REC-r2rml-20120927/.

Das, S., Cyganiak, R., and Sundara, S. (2012). R2RML: RDB to RDF mapping language. W3C recommendation, W3C. https://www.w3.org/TR/2012/REC-r2rml-20120927/.

De Giacomo, G., Lembo, D., Oriol, X., Savo, D. F., and Teniente, E. (2017). Practical update management in ontology-based data access. In d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C., and Heflin, J., editors, *The Semantic Web – ISWC 2017*, pages 225–242, Cham. Springer International Publishing.

Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., and Van de Walle, R. (2014). RML: a generic language for integrated RDF mappings of heterogeneous data. In Bizer, C., Heath, T., Auer, S., and Berners-Lee, T., editors, *Proceedings of the 7th Workshop on Linked Data on the Web*, volume 1184 of *CEUR Workshop Proceedings*.

Harris, S. and Seaborne, A. (2013). SPARQL 1.1 query language. W3C recommendation, W3C. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

Heyvaert, P., De Meester, B., Dimou, A., and Verborgh, R. (2018a). Declarative rules for linked data generation at your fingertips! In Gangemi, A., Gentile, A. L., Nuzzolese, A. G., Rudolph, S., Maleshkova, M., Paulheim, H., Pan, J. Z., and Alam, M., editors, *The Semantic Web: ESWC 2018 Satellite Events*, pages 213–217, Cham. Springer International Publishing.

Heyvaert, P., Dimou, A., De Meester, B., Seymoens, T., Herregodts, A.-L., Verborgh, R., Schuurman, D., and Mannens, E. (2018b). Specification and implementation of mapping rule visualization and editing: MapVOWL and the RMLEditor. *Journal of Web Semantics*, 49:31–50.

Heyvaert, P., Dimou, A., and Meester, B. D. (2019). Rml test cases. `https://github.com/kg-construct/rml-test-cases`. Accessed: 2023-12-21.

Ji, S., Pan, S., Cambria, E., Marttinen, P., and Yu, P. S. (2022). A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 33(2):494–514.

Manola, F. and Miller, E. (2004). RDF primer. W3C recommendation, W3C. https://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

Marchi, E. and Miguel, O. (1974). On the structure of the teaching-learning interactive process. *International Journal of Game Theory*, 3(2):83–99.

Meester, B. D., Heyvaert, P., and Delva, T. (2022). RML: RDF mapping language. Unofficial draft, RML. https://rml.io/specs/rml/.

Michel, F., Djimenou, L., Faron Zucker, C., and Montagnat, J. (2015). Translation of relational and non-relational databases into rdf with xr2rml. In *WEBIST (Selected Papers)*, pages 443–454.

Polleres, A., Lopes, N., Decker, S., Krennwallner, T., and Kopecky, J. (2009). Xsparql language specification. Technical report, W3C. https://www.w3.org/submissions/xsparql-language-specification/.

Richens, R. H. (1956). Preprogramming for mechanical translation. *Mech. Transl. Comput. Linguistics*, 3:20–25.

Seaborne, A. and Prud'hommeaux, E. (2008). SPARQL query language for RDF. W3C recommendation, W3C. https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

Singhal, A. (2012). Introducing the knowledge graph: things, not strings. 2020-11-13.

Spieldenner., D. (2022). Poser: A semantic payload lowering service. In *Proceedings of the 18th International Conference on Web Information Systems and Technologies - WEBIST*, pages 249–256. INSTICC, SciTePress.

Unbehauen, J. and Martin, M. (2017). Sparql Update queries over R2RML mapped data sources. In Eibl, M. and Gaedke, M., editors, *INFORMATIK 2017, Lecture Notes in Informatics (LNI)*, Lecture Notes in Informatics (LNI), pages 1891–1901, Chemnitz, Germany. Gesellschaft für Informatik.

Van Assche, D., Delva, T., Haesendonck, G., Heyvaert, P., De Meester, B., and Dimou, A. (2023). Declarative rdf graph generation from heterogeneous (semi-)structured data: A systematic literature review. *Journal of Web Semantics*, 75:100753.

Vu, B., Pujara, J., and Knoblock, C. A. (2019). D-repr: A language for describing and mapping diversely-structured data sources to rdf. In *Proceedings of the 10th International Conference on Knowledge Capture*, K-CAP '19, pages 189–196, New York, NY, USA. Association for Computing Machinery.