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

©Copyright KU Leuven

This master's thesis is an examination document that has not been corrected for any errors.

Reproduction, copying, use or realisation of this publication or parts thereof is prohibited without prior written consent of both the supervisor(s) and the author(s). For requests concerning the copying and/or use and/or realisation of parts of this publication, please contact KU Leuven De Nayer Campus, Jan De Nayerlaan 5, B-2860 Sint-Katelijne-Waver, +32 15 31 69 44 or via e-mail iiw.denayer@kuleuven.be.

Prior written consent of the supervisor(s) is also required for the use of the (original) methods, products, circuits and programmes described in this Master's thesis for industrial or commercial purposes and for the submission of this publication for participation in scientific prizes or competitions.

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(SPARQL 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 map[/lift] and query[/lower], but the syntax for mapping and querying differs, so it could be argued that XSPARQL is two languages.

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 sim-

ulating 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. This final chapter currently contains a roadmap for future work to be done in the second semester.

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.

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.

The subject must always be an entity, which can either be represented by an URI or be a blank

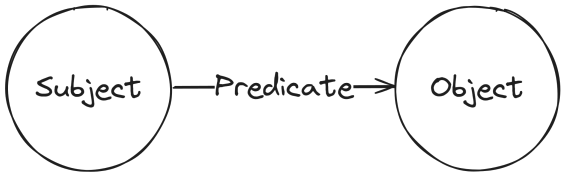

Figure 2.1: An RDF triple

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

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.

Chapter 3

Implementation

This chapter is split into three sections. In the first section the structure of the algorithm is outlined. In the second and third sections, we go into detail about the two sub-research questions: how to construct the schema and how to populate the schema.

For the implementation, we assume that the mapping rules never map to a superset of the knowledge graph.

The implementation is written in Python. As we seek to extend Morph-KGC we make use of its internal functions and the libraries to work with those, like pandas. Querying the knowledge graph is done with the SPARQLWrapper library.

3.1 General structure

The algorithm consists of 4 main parts: setting up, creating the schema, retrieving the data, and applying the data to the schema. Creating the schema and retrieving the data are the two main parts of the implementation, and are covered in their sections. Setting up consists of setting up the SPARQL endpoint if necessary and processing the mapping files. Applying the data to the schema consists of applying the data of each iteration to the schema, and creating the output file. A graphical representation of the algorithm can be found in figure 3.1. A more formal representation can be found in algorithm 1. The functions used in the algorithm are described in the relevant sections.

3.1.1 Loading the mapping rules

We start by loading the mapping rules from the mapping files. We use Morph-KGC's internal `retrieve_mappings` function for this. This function takes a loaded config file (.ini format) as input, which specifies the location of the mapping files and various other settings for which we have little use as they are only used for the materialization process. When relevant later on we could add our settings to this config file. The mappings are returned as a pandas DataFrame. As the pandas library has the functionality to group by columns, we can easily group the mapping rules by source

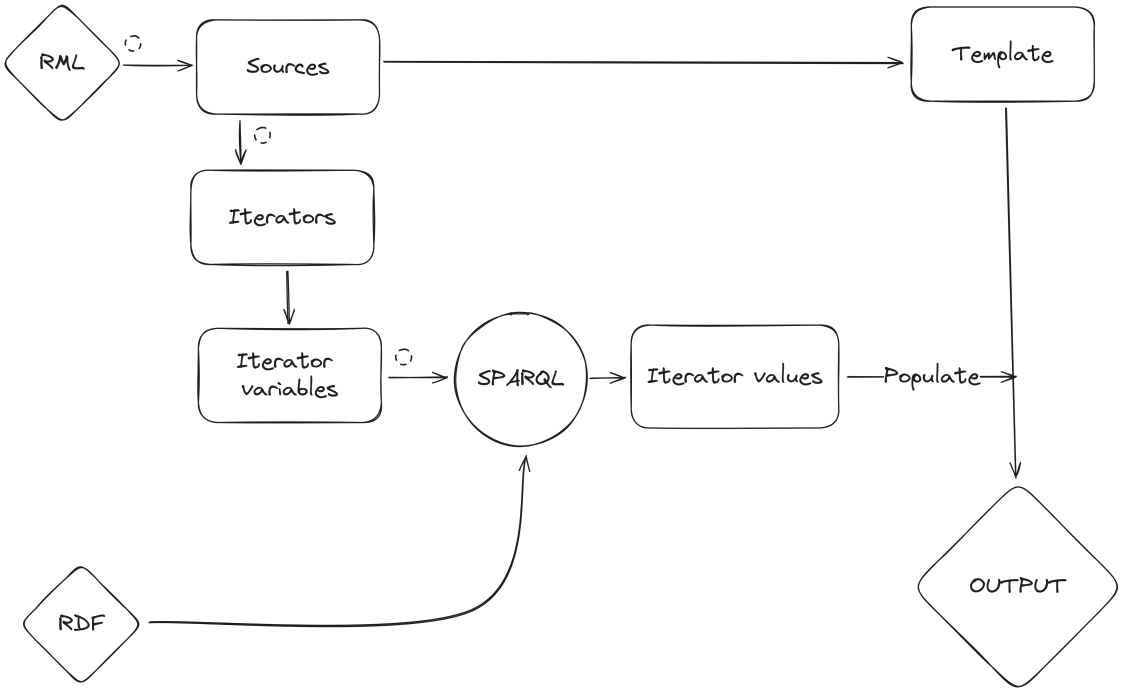

Figure 3.1: Simplified overview of the algorithm

and iterator later. An example of a single mapping rule (row in the DataFrame) can be found in listing 3.1. 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
```

Algorithm 1 Inversion algorithm

Require: $morph\_config$ is a morph-kgc config file describing the mapping files and output location

$config \leftarrow load_config(morph\_config)$
$mapping\_rules \leftarrow retrieve\_mappings(config)$
$mapping\_rules \leftarrow add\_helper\_columns(mapping\_rules)$
$graph\_location \leftarrow config.get\_output\_file()$
$graph\_endpoint \leftarrow load\_graph(graph\_location)$
for all $source, source\_rules \in mapping\_rules.groupby('source')$ **do**
    for all $iterator, iterator\_rules \in source\_rules.groupby('iterator')$ **do**
        $query \leftarrow generate\_query(iterator\_rules)$
        $values[iterator] \leftarrow graph.query(query)$
    end for
    $templates \leftarrow generate\_templates(source\_rules)$
    $source\_output \leftarrow apply\_templates(templates, values)$
end for

```
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 3.1: Example of a mapping rule in Morph-KGC

3.2 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.

3.2.1 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 3.2 results in the CSV template in listing 3.3. 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 3.2: Example mapping for a CSV file

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

Listing 3.3: Example CSV template

3.3 Retrieving the data

Retrieving the data is done by querying the knowledge graph. We do this by generating SPARQL queries matching the mapping rules. We then execute these queries and store the results. Mappings sharing the same iterator are executed together, as they have been generated in the same iteration.

3.3.1 Generating the queries

We generate the queries by translating the mapping rules into triple patterns. For each of the three map types, constant, reference, and template, we adapt our approach. 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 a foaf:Person .`. Both reference and template map types will not generate a triple during materialization if any of the references are not present during the iteration. As such, they are wrapped in an optional block. Reference maps are the easiest to work with as they directly translate back to the source. There is an exception to this, as a reference map can be used for a subject map. In this case, the value is either an URI or appended to the base URI. Sadly Morph-KGC does not support the latter, so we can not take this into account. An example of a basic query using constant and reference maps can be found in listing 3.4.

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

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

Listing 3.4: Simple query example

Template maps are the most complex to work with as their structure can wildly vary. The simplest step we can take is to confirm the mapped value matches with the map template like `FILTER(regex(str(?s), "http://example.com/([^\\/]*)/([^\\/]*)$"))`. Taking this further we can use string manipulation to split the variable into the different references. An example of this can be found in listing 3.5.

```
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 3.5: Template query example

The algorithm for generating the queries can be found in algorithm 2. This is still an early version of the algorithm, which needs to be improved to handle more complex mappings.

Algorithm 2 Generating the queries

Require: $iterator$ is the iterator to generate the query for
Require: $mapping\_rules$ is a set of mapping rules for said iterator

$query\_lines \leftarrow []$
for all $rule \in mapping\_rules$ **do**
  if $rule.is\_constant()$ **then**
    $query\_lines.append(rule.to\_triple())$
  else if $rule.is\_reference()$ **then**
    $query\_lines.append(rule.to\_optional\_triple())$
  else if $rule.is\_template()$ **then**
    $query\_lines.append(test\_object\_regex(rule))$
    $remainder \leftarrow rule['object\_map\_value']$
    for all $reference \in rule['object\_references']$ **do**
      $query\_lines.append(bind_reference_part(rule, reference))$
    end for
  end if
end for
$query \leftarrow wrap\_query\_lines(query\_lines)$

3.3.2 Disjointed mappings

We generate a single query for each iterator. This query can contain multiple subjects. This can, however, lead to issues when the subjects share no references. The effect we get 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 3.7. When applied to the knowledge graph in listing 3.8 we get the result in listing 3.9 instead of the original source in listing 3.10. 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 so even though ignoring it would be a valid solution, it is not viable with larger datasets. The only solution to this problem is updating the mapping rules to either split the source or add shared references. The user is ultimately responsible for this, but we could generate a warning to notify the user.

```
<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 3.7: 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 3.8: Bad join knowledge graph

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

Listing 3.9: Bad join result

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

Listing 3.10: Bad join original source

3.4 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 3. 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 3 Applying the data to a simple (non-nested) schema

Require: $schema$ is a schema
Require: $data$ is a DataFrame
1: $output \leftarrow new\_file$
2: **for all** $row \in data$ **do**
3: $output \leftarrow schema$
4: **for all** $column \in row$ **do**
5: $schema.replace(column.name, column.value)$
6: **end for**
7: $output.write(schema)$
8: **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 3.3, 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., 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/.

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.

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.

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.