Chapter 4
Creation of Knowledge Graphs

Anastasia Dimou

Department of Electronics and Information Systems
Ghent University, Belgium

Abstract. This chapter introduces how Knowledge Graphs are generated. The goal is to gain an overview of different approaches that were proposed and find out more details about the current prevalent ones. After reading this chapter, the reader should have an understanding of the different solutions available to generate Knowledge Graphs and should be able to choose the mapping language that best suits a certain use case.

1 Introduction

The real power of the Semantic Web will be realized once a significant number of software agents requiring information from different heterogeneous data sources become available. However, human and machine agents still have limited ability to interact with heterogeneous data as most data is not available in the form of knowledge graphs, which are the fundamental cornerstone of the Semantic Web. They have *different structures* (e.g., tabular, hierarchical), appear in *heterogeneous formats* (e.g., CSV, XML, JSON) and are accessed via *heterogeneous interfaces* (e.g., database interfaces or Web APIs).

Therefore, different approaches were proposed to generate knowledge graphs from existing data. In the beginning, custom implementations were proposed [66, 290] and they remain prevalent today [70, 177]; however, more generic approaches emerged as well. Such approaches were originally focused on data with specific formats, namely dedicated approaches for, e.g., relational databases [92], data in Excel (e.g. [272]), or in XML format (e.g. [270]). However, data owners who hold data in different formats need to learn and maintain several tools [110].

To deal with this, different approaches were proposed for integrating heterogeneous data sources while generating knowledge graphs. Those approaches follow different directions, but detaching the rules definition from their execution prevailed, because they render the rules interoperable between implementations, whilst the systems that process those rules are use-case independent. To generate knowledge graphs, on the one hand, *dedicated mapping languages* were proposed, e.g., RML [110], and, on the other hand, existing languages for other tasks were *repurposed as mapping languages*, e.g., SPARQL-Generate [276].

We focus on *dedicated mapping languages*. The most prevalent dedicated mapping languages are extensions of R2RML [96], the W3C recommendation on

knowledge graph generation from relational databases. RML was the first language proposed as an extension of R2RML, but there are more alternative approaches and extensions beyond the originally proposed language. For instance, xR2RML [303], for generating knowledge graphs from heterogeneous databases, and KR2RML [405]), for generating knowledge graphs from heterogeneous data.

In the remainder of this chapter, we introduce the Relational to RDF Mapping Language (R2RML) [96] and the RDF Mapping Language (RML) [110] which was the first mapping language extending R2RML to support other heterogeneous formats. Then we discuss other mapping languages which extended or complemented R2RML and RML, or their combination.

2 R2RML

The Relational to RDF Mapping Language (R2RML) [96] is the W3C recommendation to express customized mapping rules from data in relational databases to generate knowledge graphs represented using the Resource Description Framework (RDF) [93]. R2RML considers any custom target semantic schema which might be a combination of vocabularies. The R2RML vocabulary namespace is `http://www.w3.org/ns/r2rml#` and the preferred prefix is `r2rml`.

In R2RML, RDF triples are generated from the original data in the relational database based on one or more *Triples Maps* (`rr:TriplesMap`, listing 4.1, line 3). Each *Triples Map* refers to a *Logical Table* (`rr:LogicalTable`, line 4), specified by its *table name* (`rr:tableName`). A *Logical Table* (`rr:LogicalTable`) is either a SQL base table or view, or an R2RML view. An R2RML view is a logical table whose contents are the result of executing a SQL query against the input database. The SQL query result is used to generate the RDF triples.

Table 1: Results of female pole vault for 2019 world championship

rank	**name**	**nationality**	**mark**	**notes**
1	Anzhelika Sidorova	Authorized Neutral Athlete	4.95	WL,PB
2	Sandi Morris	United States (USA)	4.90	SB
3	Katerina Stefanidi	Greece	4.85	SB
4	Holly Bradshaw	Great Britain	4.80	
5	Alysha Newman	Canada	4.80	
6	Angelica Bengtsson	Sweden	4.80	NR

```
1  @prefix rr: <http://www.w3.org/ns/r2rml#>.
2
3  <#FemalePoleVault>   rr:logicalTable <#PoleVaultersDBtable> .
4  <#PoleVaultersDBtable> rr:tableName "femalePoleVaulters" .
```

Listing 4.1: A Triples Map refers to a Logical Table specified by its name

A *Triples Map* defines how an RDF triple is generated. It consists of three parts: (i) one *Logical Table* (`rr:LogicalTable`, listing 4.1), (ii) one *Subject Map*

(`rr:SubjectMap`, listing 4.2, line 2), and (iii) zero or more *Predicate-Object Maps* (`rr:PredicateObjectMap`, listing 4.2, lines 3 and 4).

```
1  # Triples Map
2  <#FemalePoleVault>  rr:subjectMap        <#Person_SM> ;
3                      rr:predicateObjectMap <#Mark_POM> ;
4                      rr:predicateObjectMap <#Nationality_POM> .
```

Listing 4.2: A Triples Map consists of one Logical Table one Subject Map and zero or more Predicate Object Maps

The *Subject Map* (`rr:SubjectMap`, listing 4.3, line 2) defines how unique identifiers, using IRIs [117] or blank nodes, are generated. The RDF term generated from the *Subject Map* constitutes the subject of all RDF triples generated from the *Triples Map* that the *Subject Map* is related to.

A *Predicate-Object Map* (`rr:PredicateObjectMap`, listing 4.3, lines 5 and 10) consists of (i) one or more *Predicate Maps* (`rr:PredicateMap`, line 5), and (ii) one or more *Object Maps* (`rr:ObjectMap`, line 6) or *Referencing Object Maps* (`rr:ReferencingObjectMap`, line 11).

```
1  # Subject Map
2  <#Person_SM>.     rr:template           "http:://ex.com/person/{name}"
3
4  # Predicate Object Map with Object Map
5  <#Mark_POM>       rr:predicate          ex:score ;
6                    rr:objectMap          [ rr:column "Mark" ;
7                                            rr:language "en" ] .
8
9  # Predicate Object Map with Referencing Object Map
10 <#Nationality_POM>  rr:predicateMap       <#Country_PM> ;
11                     rr:objectMap          <#Country_ROM> ;
```

Listing 4.3: A Predicate Object Map consists of one or more Predicate Maps and one or more Object Maps or Referencing Object Maps

A *Predicate Map* (`rr:PredicateMap`, listing 4.3, lines 5 and 10) is a *Term Map* (`rr:TermMap`) defining how a triple's predicate is generated. An *Object Map* (`rr:ObjectMap`, line 6) or *Referencing Object Map* (`rr:ReferencingObjectMap`, listing 4.4, line 11) defines how a triple's object is generated.

A *Referencing Object Map* defines how the object is generated based on the *Subject Map* of another *Triples Map*. If the *Triples Maps* refer to different *Logical Tables*, a join between the *Logical Tables* is required. The *join condition* (`rr:joinCondition`, listing 4.4, line 3) performs joins as joins are executed in SQL. The join condition consists of a reference to a column name that exists in the *Logical Table* of the *Triples Map* that contains the *Referencing Object Map* (`rr:child`, line 4) and a reference to a column name that exists in the *Logical Table* of the *Referencing Object Map*'s *Parent Triples Map* (`rr:parent`, line 5).

```
1  # Referencing Object Map
2  <#Country_ROM>   rr:parentTriplesMap <#Country_TM> ;
3                   rr:join  [
4                              rr:cild "nationality" ;
5                              rr:parent "country_name"] .
6
7  <#Country_TM>    rr:logicalTable [ rr:tableName "country" ];
```

```
8                rr:subjectMap   rr:template "http://ex.com/country/{country_name}" .
```

Listing 4.4: A Referencing Object Map generates an object based on the Subject Map of another Triples Map

A *Term Map* (`rr:TermMap`) defines how an RDF term (an IRI, a blank node, or a literal) is generated and it can be constant-, column- or template-valued.

A *constant-valued Term Map* (`rr:constant`, listing 4.3, line 5) always generates the same RDF term which is by default an IRI.

A *column-valued term map* (`rr:column`, listing 4.3, line 6) generates a literal by default that is a column in a given *Logical Table*'s row. The language (`rr:language`, line 7) and datatype (`rr:datatype`) may be optionally defined.

A *template-valued* Term Map (`rr:template`, listing 4.3, line 8) is a valid string template containing referenced columns and generates an IRI by default. If the default termtype is desired to be changed, the term type (`rr:termType`) needs to be defined explicitly (`rr:IRI`, `rr:Literal`, `rr:BlankNode`).

3 RML

The RDF Mapping Language (RML) [109, 110] expresses customized mapping rules from heterogeneous data structures, formats and serializations to RDF. RML is a superset of R2RML, aiming to extend its applicability and broaden its scope, adding support for heterogeneous data. RML keeps the mapping rules as in R2RML but excludes its database-specific references from the core model. This way, the input data that is limited to a certain database in R2RML (because each R2RML processor may be associated to only one database), becomes a broad set of one or more input data sources in RML.

RML provides a generic way of defining mapping rules referring to different data structures, combined with case-specific extensions, but remains backwards compatible with R2RML, as relational databases form such a specific case. RML enables mapping rules defining how a knowledge graph is generated from a set of sources that altogether describe a certain domain, can be defined in a combined and uniform way. The mapping rules may be re-used across different sources describing the same domain to incrementally form well-integrated datasets.

The RML vocabulary namespace is `http://semweb.mmlab.be/ns/rml#` and the preferred prefix is `rml`.

In the remainder of this subsection, we will talk in more details about data retrieval and transformations in RML, as well as other representations of RML.

3.1 Data Retrieval

Data can originally (i) reside on **diverse locations**, e.g., files or databases on the local network, or published on the Web; (ii) be accessed using **different interfaces**, e.g., raw files, database connectivity for databases, or different interfaces from the Web such as Web APIs; and (iii) have **heterogeneous structures and formats**, e.g., tabular, such as databases or CSV files, hierarchical, such as XML or JSON format, or semi-structured, such as HTML.

In this section, we explain how RML performs the retrieval and extraction steps required to obtain the data whose semantic representation is desired.

Logical Source RML's *Logical Source* (`rml:LogicalSource`, listing 4.5) extends R2RML's *Logical Table* and determines the data source with the data to generate the knowledge graph. The R2RML *Logical Table* definition determines a database table, using the *Table Name* (`rr:tableName`). In the case of RML, a broader reference to any input source is required. Thus, the *Logical Source* (`rml:source`) is introduced to specify the source with the original data.

For instance, if the data about countries were in an XML file, instead of a *Logical Table*, we would have a *Logical Source* `<#PoleVaultersXML>` (listing 4.5, line 3):

```
1  @prefix rml: <http://semweb.mmlab.be/ns/rml#>.
2
3  <#Countries> rml:logicalSource <#CountriesXML> ;
4  <#CountriesXML> rml:source <http://rml.io/data/lambda/countries.xml> .
```

Listing 4.5: A Triples Map refers to a Logical Source whose data is in XML format

The countries data can then be in XML format as below:

```
1  <countries>
2    <country continent="Europe">
3      <country_abb>GR</country_abb>
4      <country_name country_language="en">Greece</country_name>
5      <country_name country_language="nl">Griekenland</country_name>
6    </country>
7    <country continent="Europe">
8      <country_abb>UK</country_abb>
9      <country_name country_language="en">United Kingdom</country_name>
10     <country_name country_language="nl">Verenigd Koninkrijk</country_name>
11   </country>
12   <country continent="America">
13     <country_abb>CA</country_abb>
14     <country_name country_language="en">Canada</country_name>
15     <country_name country_language="nl">Canada</country_name>
16   </country>
17 ...
18 </countries>
```

Listing 4.6: Country data in XML format

Reference Formulation RML deals with different data serialisations which use different ways to refer to data fractions. Thus, a dedicated way of referring to the data's fractions is considered, while the mapping definitions that define how the RDF terms and triples are generated remain generic. RML considers that any reference to the *Logical Source* should be defined in a form relevant to the input data, e.g. XPath for XML data or JSONPath for JSON data. To this end, the *Reference Formulation* (rml:referenceFormulation) declaration is introduced (listing 4.7, line 4), indicating the formulation (for instance, a standard, query language or grammar) used to refer to its data.

```
1  @prefix rml: <http://semweb.mmlab.be/ns/rml#>.
2
```

```
3  <#Countries>      rml:logicalSource <#CountriesXML> .
4  <#CountriesXML>   rml:referenceFormulation ql:XPath .
5  <#CountriesXML>   rml:iterator "/countries/country" .
```

Listing 4.7: A Logical Source specifies its Reference Formulation and iterator

Iterator While in R2RML it is already known that a per-row iteration occurs, as RML remains generic, the iteration pattern, if any, cannot always be implicitly assumed, but it needs to be determined. Thereafter, the *iterator* (`rml:iterator`) is introduced (listing 4.7, line 5). The *iterator* determines the iteration pattern over the data source and specifies the extract of the data during each iteration. The *iterator* is not required to be explicitly mentioned in the case of tabular data sources, as the default per-row iteration is implied.

Source Data can originally reside on diverse, distributed locations and be accessed using different access interfaces [111]. Data can reside locally, e.g., in files or in a database at the local network, or can be published on the Web. Data can be accessed using diverse interfaces. For instance, metadata may describe how to access the data, such as dataset's metadata descriptions in the case of data catalogues, or dedicated access interfaces might be needed to retrieve data from a repository, such as database connectivity for databases, or different Web interfaces, such as Web APIs.

RML considers an original data source, but the way this input is retrieved remains out of scope, in the same way it remains out of scope for R2RML how the SQL connection is established. Corresponding vocabularies can describe how to access the data, for instance the dataset's metadata (listing 4.8), hypermedia-driven Web APIs or services, SPARQL services, and database connectivity frameworks (listing 4.9) [111].

```
1  <#FemalePoleVault>   rr:logicalTable   <#PoleVaultersCSVtable> ;
2  <#PoleVaultersCSVtable> rml:source <#CSVW_source> .
3
4  <#CSVW_source> a csvw:Table;
5                 csvw:url "femalePoleVaulters.csv" ;
6                 csvw:dialect [ a csvw:Dialect; csvw:delimiter ";" ] .
```

Listing 4.8: A CSV file on the Web as RML Data Source

```
1  <#FemalePoleVault>   rr:logicalTable   <#PoleVaultersDBtable> ;
2  <#PoleVaultersDBtable> rml:source <#DB_source>;
3                         rr:sqlVersion rr:SQL2008;
4                         rr:tableName "femalePoleVaulters" .
5
6  <#DB_source> a d2rq:Database;
7               d2rq:jdbcDSN "CONNECTIONDSN";
8               d2rq:jdbcDriver "com.mysql.cj.jdbc.Driver";
9               d2rq:username "root";
10              d2rq:password "" .
```

Listing 4.9: A table as RML Data Source

Logical Reference According to R2RML, a *column-valued* or *template-valued term map* is defined as referring to a column name. In the case of RML, a more generic notion is introduced, the logical reference. Its value must be a valid reference to the data of the input dataset according to the specified reference formulation. Thus, the reference's value should be a valid expression according to the *Reference Formulation* defined at the *Logical Source*.

```
1  # Predicate Object Map with Object Map
2  <#CountryName_POM>   rr:predicate ex:name ;
3                       rr:objectMap [
4                          rml:reference "country_name" ;
5                          rml:languageMap [ rml:reference "@country_language"] ] .
```

Listing 4.10: An Object Map in RML with a reference to data according to the Reference Formulation and a language Map to define the language.

RDF Term Maps are instantiated with data fractions referred to using a reference formulation relevant to the corresponding data format. Those fractions are derived from data extracted at a certain iteration from a *Logical Source*. Such a *Logical Source* is formed by data retrieved from a repository accessed as defined by the corresponding dataset or service description vocabulary.

Language Map RML introduces a new *Term Map* for defining the language, the *Language Map* (`rml:LanguageMap`, listing 4.10, line 5), which extends R2RML's language tag (`rr:language`). The *Language Map* allows not only constant values for language but also references derived from the input data. `rr:language` is considered then an abbreviation for the `rml:languageMap`, as `rr:predicate` is for the `rr:predicateMap`.

3.2 Data Transformations: FnO

Mapping rules involve (re-)modeling the original data, describing how objects are related by specifying correspondences between data in different schemas [126], and deciding which vocabularies and ontologies to use. *Data transformations*, as opposed to *schema transformations* that the mapping rules represent, are needed to support any changes in the structure, representation or content of data [365], for instance, performing string transformations or computations.

The *Function Ontology* (FnO) [101, 103] describes functions uniformly, unambiguously, and independently of the technology that implements them. As RML extends R2RML with respect to *schema transformations*, the combination of RML with FnO extends R2RML with respect to *data transformations*.

A *function* (`fno:Function`) is an activity which has input parameters, output, and implements certain algorithm(s) (listing 4.11, line 1). A *parameter* (`fno:Parameter`) is a function's input value (listing 4.11, line 4). An *output* (`fno:Output`) is the function's output value (listing 4.11, line 5). An *execution* (`fno:Execution`) assigns values to the parameters of a function for a certain execution. An *implementation* (`fno:Implementation`) defines the internal workings of one or more functions.

```
grel:string_split a fno:Function;
    fno:name "split";
    dcterms:description "split";
    fno:expects (grel:string_s grel:string_sep);
    fno:returns (grel:output_array).
```

Listing 4.11: A function described in FnO that splits a string

The *Function Map* (`fnml:FunctionMap`) is another *Term Map*, introduced as an extension of RML, to facilitate the alignment of the two, RML and FnO. A *Function Map* is generated by executing a function instead of using a constant or a reference to the raw data values. Once the function is executed, its output value is the term generated by this *Function Map*. The `fnml:functionValue` property indicates which instance of a function needs to be executed to generate an output and considering which values.

```
<#FemalePoleVault> rr:predicateObjectMap [
    rr:predicate ex:record;
    rr:objectMap [
      fnml:functionValue [
        rr:predicateObjectMap [
                rr:predicate fno:executes ;
                rr:objectMap [ rr:constant grel:split ] ] ;
        rr:predicateObjectMap [
                rr:predicate grel:string_s ;
                rr:objectMap [ rml:reference "notes" ] ] ;
        rr:predicateObjectMap [
                rr:predicate grel:string_sep ;
                rr:objectMap [ rr:constant "," ] ] ] ].
```

Listing 4.12: A Function Map aligns FnO with RML

3.3 Other Representations: YARRRML

YARRRML [195, 102] is a human readable text-based representation for mapping rules. It is expressed in YAML [45], a widely used human-friendly data serialization language. YARRRML can be used with both R2RML and RML.

A *mapping* (listing 4.13, line 1) contains all definitions that state how subjects, predicates, and objects are generated. Each mapping definition is a key-value pair. The key *sources* (line 3) defines the set of data sources that are used to generate the entities. Each source is added to this collection via a key-value pair. The value is a collection with three keys: (i) the key *access* defines the local or remote location of the data source; (ii) the key *reference formulation* defines the reference formulation used to access the data source; and (iii) the key *iterator* (conditionally required) defines the path to the different records over which to iterate. The key *subjects* (line 5) defines how the subjects are generated. The key *predicateobjects* (line 6) defines how combinations of predicates and objects are generated. Below the countries example (listing 4.6) is shown in YARRRML:

```
mappings:
  country:
    sources:
      - ['countries.xml~xpath', '/countries/country']
    s: http://ex.com/$(country_abb)
    po:
      - [ex:name, $(country_name)]
```

```
8       - [ex:abbreviation, $(country_abb)]
```

Listing 4.13: A YARRRML set of mapping rules

4 [R2]RML extensions and alternatives

Other languages were proposed based on differentiation on (i) data retrieval and (ii) data transformations. The table below (table 2) summarizes the mapping languages extensions, their prefixes and URIs. xR2RML [304] and KR2RML [405] are the two most prominent solutions that showcase extensions and alternatives respectively for data retrieval. On the one hand, xR2RML extends R2RML following the RML paradigm to support heterogeneous data from non-relational databases. On the other hand, KR2RML extends R2RML relying on the Nested Relational Model (NRM) [452] as an intermediate form to represent data originally stored in relational databases. KR2RML also provided an alternative for data transformations, but FunUL is the most prominent alternative to FnO.

Table 2: [R2]RML extensions, their URIs and prefixes

language	**prefix**	**URI**
R2RML	`rr`	`http://www.w3.org/ns/r2rml#`
RML	`rml`	`http://semweb.mmlab.be/ns/rml#`
xR2RML	`xrr`	`http://www.i3s.unice.fr/ns/xr2rml#`
FnO+RML	`fnml`	`http://semweb.mmlab.be/ns/fnml#`
FnO	`fno`	`https://w3id.org/function/ontology#`

4.1 xR2RML

xR2RML [304] was proposed in 2014 in the intersection of R2RML and RML. xR2RML extends R2RML beyond relational databases and RML to include non-relational databases. xR2RML extends R2RML following the RML paradigm but is specialized for non-relational databases and, in particular, NoSQL and XML databases. NoSQL systems have heterogeneous data models (e.g., key-value, document, extensible column, or graph store), as opposed to relational databases. xR2RML assumes, as R2RML does, that a processor executing the rules is connected to a certain database. How the connection or authentication is established against the database is out of the language's scope, as in R2RML.

The xR2RML vocabulary preferred prefix is `xrr` and the namespace is the following: `http://www.i3s.unice.fr/ns/xr2rml#`.

Data Source Similarly to RML, an xR2RML Triples Map refers to a Logical Source (`xrr:logicalSource`, listing 4.14, line 3), but similarly to R2RML, this Logical Source can be either an xR2RML base table (`xrr:sourceName`, for

databases where tables exist) or an xR2RML view representing the results of executing a query against the input database (`xrr:query`, line 4).

```
1  @prefix xrr: <http://www.i3s.unice.fr/ns/xr2rml#> .
2
3  <#CountriesXML> xrr:logicalSource [
4      xrr:query """for $i in ///countries/country return $i; """;
5      rml:iterator "//countries/country";];
6   <#CountryName_POM>   rr:predicate ex:name ;
7                        rr:objectMap [ xrr:reference "country_name"] .
```

Listing 4.14: xR2RML logical source over an XML database supporting XQuery

Iterator xR2RML originally introduced the `xrr:iterator`, according to the `rml:iterator`, to iterate over the results. In a later version, xR2RML converged using the `rml:iterator` (listing 4.14, line 5).

Format or Reference Formulation In contrast to RML that considers a formulation (`rml:referenceFormulation`) to refer to its input data, xR2RML originally specified explicitly the format of data retrieved from the database using the property `xrr:format` (listing 4.15, line 2). For instance, RML considers XPath or XQuery or any other formulation to refer to data in XML format, xR2RML would refer to the format, e.g. `xrr:XML`. While RML allows for other kinds of query languages to be introduced, xR2RML decides exactly which query language to use. In an effort to converge with RML, xR2RML considers optionally a reference formulation.

```
1  <#FemalePoleVault>   xrr:logicalSource   <#PoleVaultersCSVtable> ;
2  <#PoleVaultersCSVtable> xrr:format xrr:Row .
```

Listing 4.15: A CSV file on the Web as xR2RML Logical Source

Reference Similar to RML, xR2RML uses a reference (`xrr:reference`) to refer to the data elements (listing 4.14, line 7). xR2RML extends RML's reference to refer to data elements in data with mixed formats. xR2RML considers cases where different formats are nested; for instance, a JSON extract is embedded in a cell of a tabular structure. A path with mixed syntax consists of the concatenation of several path expressions separated by the slash '/' character.

Collections and Containers Several RDF terms can be generated by a *Term Map* during an iteration if multiple values are returned. This can normally generate several triples, but it can also generate hierarchical values in the form of RDF collections or containers. To achieve the latter, xR2RML extends R2RML by introducing corresponding datatypes to support the generation of containers. xR2RML introduces new *term types* (`rr:termType`): `xrr:RdfList` for an `rdf:List`, `xrr:RdfBag` for `rdf:Bag`, `xrr:RdfSeq` for `rdf:Seq` and `xrr:RdfAlt` for `rdf:Alt`. All *RDF terms* produced by the *Object Map* during one triples map iteration step are then grouped as members of one term. To achieve this, two more constructs are introduced: *Nested Term Maps* and *Push Downs*.

```
1  <#Countries> rr:predicateObjectMap [
2      rr:predicate ex:name;
3      rr:objectMap [
4        xrr:reference "country_name";
5        rr:termType xrr:RdfList;
6        xrr:pushDown [ xrr:reference "@continent"; xrr:as "continent" ];
7        xrr:nestedTermMap [
8          rr:template "{continent}: {country_name}" ;
9          rr:termType rr:Literal ;
10         rr:dataType xsd:string ] ].
```

Listing 4.16: An xrr:RdfList in xR2RML

Nested Term Map A *Nested Term Map* (`xrr:NestedTermMap`, listing 4.16, line 7) accepts the same properties as a *Term Map* and can be used to specify a term type, a language tag or a data type for the members of the generated RDF collection or container.

Push Down Within an iteration, it may be needed to access data elements higher in the hierarchical documents in the context of hierarchical data formats, such as XML or JSON. To deal with this, xR2RML introduces the `xrr:pushDown` property (listing 4.16, line 6).

4.2 KR2RML

KR2RML [405] extends R2RML in a different way than xR2RML. KR2RML relies on the Nested Relational Model (NRM) as an intermediate form to represent data. The data is mapped into tables by translating it into tables and rows where a column in a table can be either a scalar value or a nested table. Besides the data retrieval part, KR2RML extends R2RML with data transformations using User Defined Functions (UDFs) written in Python.

Data Source Mapping *tabular data* (e.g., CSV) into the Nested Relational Model is straightforward. The model has a one-to-one mapping of tables, rows, and columns, unless a transformation like splitting on a column occurs, which will create a new column that contains a nested table.

Mapping *hierarchical data* (e.g., JSON, XML) into the Nested Relational Model requires a translation algorithm for each data format next to the mapping language. Such an algorithm is considered for data in XML and JSON format. If the data is in JSON, an object maps to a single row table in NRM with a column for each field. Each column is populated with the value of the appropriate field. Fields with scalar values do not need translation, but fields with array values are translated to their own nested tables: if the array contains scalar or object values, each array element becomes a row in the nested table. If the elements are scalar values like strings as in the tags field, a default column name “values” is provided. If a JSON document contains a JSON array at the top level, each element is treated like a row in a database table. If the data is in XML format, its elements are treated like JSON objects, and its attributes and repeated child elements as single-row nested table where each attribute is a column.

References The *column-valued term map* is not limited to SQL identifiers as it occurs in R2RML to support mapping nested columns in the NRM. A JSON array is used to capture the column names that make up the path to a nested column from the document root. The *template-valued term map* is also extended to include columns that do not exist in the original input but are the result of the transformations applied by the processor.

Joins Joins are not supported because they are considered to be impractical and require extensive planning and external support.

Execution Planning A tag (`km-dev:hasWorksheetHistory`) is introduced to capture the cleaning, transformation and modeling steps.

Data Transformations The Nested Transformation Model can also be used to embed transformation functions. A transformation function can create a new set of nested tables instead of transforming the data values.

4.3 FunUL

FunUL [231] is an alternative to FnO for data transformations. FunUL allows the definition of functions as part of the mapping language. In FunUL, functions have a name and a body. The name needs to be unique. The body defines the function using a standardized programming language. It has a return statement and a call refers to a function with an optional set of parameters.

The FunUL vocabulary namespace is `http://kdeg.scss.tcd.ie/ns/rrf#` and the preferred prefix is `rrf`.

The class `rrf:Function` defines a function (listing 4.17, line 3). A function definition has two properties defining the name (`rrf:functionName`, line 4), and the function body (`rrf:functionBody`, line 5).

A function can be called using the property `rrf:functionCall` (listing 4.17, line 13). This property refers to a `rrf:Function` with the property `rr:function` (line 14). Parameters are defined using `rrf:parameterBindings` (line 15).

```
1  @prefix rrf: <http://kdeg.scss.tcd.ie/ns/rrf#> .
2
3  <#SplitTransformation> a rrf:Function ;
4      rrf:functionName "splitTransformation" ;
5      rrf:functionBody
6          """function split(value, separator) {
7              str = value.split(separator).trim();
8              return str; "}; } """ ; .
9
10 <#FemalePoleVault> rr:predicateObjectMap [
11     rr:predicate ex:record;
12     rr:objectMap [
13       rrf:functionCall [
14         rrf:function <#SplitTransformation> ;
15         rrf:parameterBindings (
16             [ rml:reference "notes" ]
17             [ rml:reference "," ] ); ];
```

Listing 4.17: A Function Map aligns FnO with RML

5 Conclusions

A lack of in-depth understanding of the complexity of generating knowledge graphs and the many degrees of freedom in modeling and representing knowledge prevents human and software agents from profiting of the Semantic Web potential. This chapter identified the different approaches that were proposed in recent years for generating knowledge graphs from heterogeneous data sources. Then, the chapter focused on approaches that distinguish mapping rules definition from their execution. Two types of mapping languages prevailed, *dedicated mapping languages* and *repurposed mapping languages*. The chapter further focused on *dedicated mapping languages* because they follow the W3C-recommended R2RML.

This chapter presents the author's view on knowledge graph generation. It serves as an introductory chapter to knowledge graphs, which are considered in greater detail in the following chapters. The next two chapters will explain how to perform federated querying and reasoning over knowledge graphs.

Table 3: Mapping Languages comparison with respect to data retrieval

	R2RML	**RML**	**xR2RML**	**KR2RML**
extends	–	R2RML	R2RML & RML	R2RML
data source	rr:LogicalTable	rml:LogicalSource	xrr:LogicalSource	rr:LogicalTable
data references	–	reference formulation	xrr:format	–
reference	rr:column rr:template	rml:reference rr:template	xrr:reference rr:template	rr:column rr:template
reference formulation	SQL	SQL/XPath/ JSONPath acc. reference formulation	SQL/XPath/ JSONPath acc. xrr:format	SQL/XPath/ JSONPath
join	rr:join	rr:join (extended)	rr:join (extended)	not supported
declarative iterator	no	yes	yes	no
iterator	–	rml:iterator	xrr:iterator	–
query	rr:sqlQuery	rml:query	xrr:query	rr:sqlQuery
lists	–	–	xrr:RdfList	–