RML-view-to-CSV: A Proof-of-Concept Implementation for RML Logical Views

Els de Vleeschauwer[1], Pano Maria[2], Ben De Meester[1] and Pieter Colpaert[1]

[1]IDLab, Dept. Electronics & Information Systems, Ghent University – imec, Belgium
[2]Skemu, Schiedam, The Netherlands

Abstract

Although the W3C Community Group on Knowledge Graph Construction (KGC)'s work on the modular RDF Mapping Language (RML) specification has taken great strides, open issues and respective solution proposals remain. Some of these issues are (i) inability to handle hierarchy in nested data, (ii) limited join functionality, and (iii) inability to handle mixed data formats. To combat these issues, the RML Logical Views module is proposed. However, proper but efficient validation of this module requires an implementation that allows short development cycles. In this workshop paper, we propose a proof-of-concept RML Logical Views implementation, independent of and complementary to existing RML mapping engines. Our proof-of-concept covers three important features of the new RML Logical Views module: (i) flattening of nested data, (ii) extended joining of data sources, and (iii) handling mixed data formats. Our implementation supports one nested source format (JSON) and one tabular source format (CSV), and can be used independently, as preprocessor, by any RML Engine. With this implementation, we successfully executed the available relevant test cases of the RML Logical Views module. Additionally, we measured the knowledge graph construction times on GTFS-Madrid-Bench. To accomplish this we added an option to our implementation that replaces referencing object maps with joins in RML Logical Views. When we included our implementation in the knowledge graph construction pipeline, we noticed considerable execution time reductions. We conclude that the RML Logical Views specification can be implemented, and can solve needs that were not yet solvable by RML. The current implementation can already be realized as a modular part of a knowledge graph construction process. Although boosting performance was not the aim of our work, our implementation reduces the execution time of GTFS-Madrid-Bench scale 100 by 16%, 33%, and 39% when combined respectively with SMD-Rdfizer or RPT/Sansa, Morph-KGC, and Carml. RMLStreamer, when used alone, times out after two hours on this task, but, in conjunction with our implementation, completes it in 236 seconds. We hope this proof-of-concept inspires the developers of existing RML engines to integrate the RML Logical Views module and benefit from its features.

Keywords

RML Logical View, flattening, joining, mixed content, proof-of-concept

KGCW'24: 5th International Workshop on Knowledge Graph Construction, May 27, 2024, Crete, GRE
els.devleeschauwer@ugent.be (E. de Vleeschauwer); pano@skemu.com (P. Maria); ben.demeester@ugent.be (B. De Meester); pieter.colpaert@ugent.be (P. Colpaert)
https://skemu.com (P. Maria); https://ben.de-meester.org/#me (B. De Meester); https://pietercolpaert.be/#me (P. Colpaert)
0000-0002-8630-3947 (E. de Vleeschauwer); 0009-0000-2598-1894 (P. Maria); 0000-0003-0248-0987 (B. De Meester); 0000-0001-6917-2167 (P. Colpaert)

1. Introduction

The W3C Community Group on Knowledge Graph Construction (KGC)[1] works on a declarative approach to construct RDF graph data from existing, heterogeneous data sources. The group recently proposed a new modular specification, ontology, and accompanying SHACL shapes for the RDF Mapping Language (RML)[2], including novel features which increase its expressiveness and empowers practitioners to define mapping rules for constructing RDF graph data that were previously unattainable [1]. Nevertheless, challenges such as handling hierarchy of nested data, more flexible joining (also across data hierarchies), and handling data sources that mix source formats (e.g., a table that contains a column storing data as JSON) remain unsolved.

As of July 2023, a dedicated task force of the W3C Community Group on KGC works on an additional RML module: RML Logical Views[3]. This module aims to resolve the aforementioned challenges by allowing to specify a *logical view*: a flattened, source format-agnostic view over one or more existing data sources.

RML Logical views are new in RML, and still under development. The specification[4] is not finalized and still subject to change. Hence, there are no implementations available to validate the feasibility of the theoretical concepts. Proper but efficient validation of the RML Logical Views module requires an implementation that allows short development cycles.

A proof-of-concept implementation can validate if the proposed RML constructions are implementable, and reveal ambiguities in the specification. This allows for corrective iterations during the development of the specification, and can also support the creation of test cases.

In this workshop paper, we present RML-view-to-CSV: a proof-of-concept implementation made available under MIT license and designed following state-of-the-art best practices. RML-view-to-CSV[5] materializes all RML Logical Views specified in a given set of RML mapping rules as CSV files, and rewrites these RML mapping rules to RML mapping rules without RML Logical Views. Any RML engine that supports CSV files, can then use these resulting RML mapping rules and the generated CSV files to construct RDF graph data using existing RML constructs. The implementation supports two source file formats (CSV and JSON) and supports the following features: flattening of nested data, handling of mixed data formats, and more flexible joining of data sources (also across data hierarchies). We added two optional functionalities: the elimination of referencing object maps by moving the joins to logical views, and the optimization of logical views based on the linked triples maps. The first option allows us to test our implementation with existing RML benchmarks. Combined with the second option, we obtained considerable execution time reductions for the knowledge graph construction pipeline.

After discussing related work (Section 2) and explaining the main principles of logical views and how they are specified using RML Logical Views (Section 3), we describe our approach and the implemented features (Section 4). Finally, we evaluate (Section 5), and conclude (Section 6).

[1] https://www.w3.org/community/kg-construct/
[2] https://w3id.org/rml/portal/
[3] https://github.com/kg-construct/rml-lv
[4] https://kg-construct.github.io/rml-lv/dev.html
[5] https://github.com/RMLio/rml-view-to-csv

2. Related work

In this section, we first list solutions that were formulated in the past and influenced the development of RML Logical Views (Section 2.1). Then, we describe similar modular implementation approaches (Section 2.2).

2.1. Related proposals to extend RML

Partial solutions that also apply the concept of a (logical) view were formulated in the past to address (parts of) the challenges presented in this paper [2, 3]. RML Fields is a proposed solution to deal with nested data and mixed content in RML [2]. The proposal includes a language construct and algorithm, but no implementation. RML Logical Views is a revised extension of this proposal. Another solution is expanding RML's existing SQL views to general tabular sources instead of solely relational databases [3], to better support transformation functions, complex joins, and mixed content in RML. These views are formulated as SQL queries over tabular sources. The proposal is implemented as an extension of Morph-KGC[6] [4]: a state-of-the-art RML mapping engine implemented using the pandas Python package[7] [5].

Other proposed solutions include Facade-X [6], which directly maps the data source into RDF graph data (so not a flattened, but a graph-based source-agnostic view), allowing to, e.g. support joining over hierarchy via an iteration index[8]; and xR2RML, which (among others) supports mixed syntax paths for handling data sources that mix source formats [7].

2.2. RML engine module implementations

State-of-the-art RML engine module implementations typically provide (integrated or separate) preprocessing steps. SDM-RDFizer introduces a preprocessing step for grouping mapping rules, which leads to optimizations due to parallelization [8]. FunMap [9] is an interpreter of RML+FnO (the RML module that allows to describe data transformations in the mapping process), that converts a data integration system defined using RML+FnO into an equivalent one where RML mappings are function-free.

3. RML Logical Views Primer

Within RML (https://w3id.org/rml/portal), RDF triple generation is defined using *triples maps*. Within these triples maps, *expressions* are used to fill in data values in specified places in the triples. On which data these triples are generated, is currently specified using a *logical source*: a construct to describe how to extract *logical iterations* out of heterogeneous data sources, e.g. extracting individual rows from CSV files or extracting specific objects from JSON data from a Web API response. *Reference formulations* allow specifying the manner to create logical iterations and expressions (e.g. using the JSONPath reference formulation to handle JSON data).

[6] https://github.com/morph-kgc/morph-kgc/releases/tag/2.3.0
[7] https://pandas.pydata.org/
[8] https://github.com/kg-construct/mapping-challenges/issues/43

For example, when mapping person data from a local CSV file that contains a person's (national) ID and name, the logical source describes how to extract rows as logical iterations from that CSV file, using a default CSV reference formulation. On these logical iterations, specific `ID` and `name` expressions are evaluated to create triples, using the `ID` expression as part of the subject identifier, and the `name` expression as a literal object.

With RML Logical Views, the W3C Community Group on KGC aims to describe a virtual flattened view on top of a logical source. A *logical view* allows defining *fields* where each field is an expression on its parent in terms of a reference formulation. A field's parent is either the logical iteration of the logical source when a field is defined at root-level of the logical view, or another field. The result of evaluating field's expression is a list of values called an iteration sequence [2]. The reference formulation of the expression of the field is the reference formulation of its parent, unless it is specified on the field. Each field has a *declared name* which is an alphanumeric string, and a *field name* which is the concatenation of the name of the parent field, a dot, and the field's declared name.

The evaluation of the fields of a logical view on a given logical iteration forms a new view iteration, which is the natural, full outer join of the logical iteration of the logical source and all the fields' iteration sequences in order of the defined field hierarchy. Following this, expression result values derived along the same root-to-leaf path in the input data's tree structure, will end up in the same view iteration [2].

To reference a field value while generating triples in a triples map, its field name can be used. This approach gives a mapping author the control to map a hierarchical source in a variety of ways whilst still being able to respect the context of the source hierarchy.

A logical view can be further extended with fields from one or more other logical views by defining *joins* with other logical views. Since the logical iterations of a logical view can be represented as relation in a relational algebra, we can also define joins in terms of relational algebra (left join, inner join). These joins will define which field values from a parent logical view will be incorporated in a logical iteration of the child logical view.

In this section, we give an example-based explanation of the main RML Logical Views functionalities. The examples[9] (Figures 1 to 3) include an RML mapping illustrating the declarative description of a logical view, the source data for the logical source used as source for the logical view, and an intermediate representation of the logical view. The headers of the columns of this intermediate representation can be used as reference expression in expression maps.

3.1. Flattening of nested data structures

Problem: References to nested data structures, like JSON or XML, may return multiple values. These values can be composite: they may again contain multiple values. RML defines mapping constructs that produce results by combining the results of other mapping constructs in a specific order. For example, a triples map combines the results of a subject map and a predicate-object map in that order. Another example is a template expression, which combines character strings and zero or more reference expressions in declared order. When mapping constructs produce multiple results, the combining mapping constructs will apply an n-ary Cartesian product[10]

[9]Prefixes are omitted but can be found on https://prefix.cc

[10]https://w3id.org/rml/core/spec#dfn-n-ary-cartesian-product

```
:jsonSource a rml:LogicalSource ;
  rml:source "json_data.json" ;
  rml:referenceFormulation rml:JSONPath ;
  rml:iterator "$.people[*]" .
:jsonView a rml:LogicalView ;
  rml:onLogicalSource :jsonSource ;
  rml:field [
    rml:fieldName "name" ;
    rml:reference "$.name" ; ] ;
  rml:field [
    rml:fieldName "item" ;
    rml:reference "$.items[*]" ;
    rml:field [
      rml:fieldName "type" ;
      rml:reference "$.type" ; ] ;
    rml:field [
      rml:fieldName "weight" ;
      rml:reference "$.weight" ; ] ; ] .
```

(a) mapping_json_view.ttl (part 1)

```
{ "people": [
    { "name": "alice",
      "items": [
        { "type": "sword",
          "weight": 1500 },
        { "type": "shield",
          "weight": 2500 }
        ] },
    { "name": "bob",
      "items": [
        { "type": "flower",
          "weight": 15 }
        ] }
    ] }
```

(b) json_data.json

name	item	item.type	item.weight
alice	{”type”: ”sword”, ”weight”: 1500}	sword	1500
alice	{”type”: ”shield”, ”weight”: 2500}	shield	2500
bob	{”type”: ”flower”, ”weight”: 15}	flower	15

(c) Intermediate representation of :jsonView

Figure 1: Example of a logical view on a logical source containing nested JSON data

over the sets of results, maintaining the order of the mapping constructs. In the case of nested data structures, this may cause the generation of results that do not match the source hierarchy, i.e. do not follow the root-to-leaf paths in the source data, since values are combined irrespective of it.

Furthermore, there is varying expressiveness in data source expression and query languages, and many languages have limited support for hierarchy traversal. For example, JSONPath has no operator to refer to an ancestor in the document hierarchy.

This limits the ability of RML to map nested data.

Solution: Fields can be defined in a hierarchy following the source document to produce iterations from which source values along the same root-to-leaf path can be referenced. The view iteration can be seen as flattening the hierarchy of the source document (Figure 1).

3.2. Handling of mixed data formats

Problem: Data in one format can contain multiple or composite values stored in another format[11] [2], e.g. a CSV dataset could contain columns containing JSON values.

[11] https://webusers.i3s.unice.fr/~fmichel/xr2rml_specification_v5.1.html#_Toc466307454

```
:mixedSource a rml:LogicalSource ;
    rml:source "./mixed_data.csv";
    rml:referenceFormulation rml:CSV .
:mixedView a rml:LogicalView ;
  rml:onLogicalSource :mixedSource;
  rml:field [
    rml:fieldName "name" ;
    rml:reference "name" ; ] ;
  rml:field [
    rml:fieldName "item" ;
    rml:reference "item" ;
    rml:referenceFormulation rml:JSONPath ;
    rml:field [
      rml:fieldName "type" ;
      rml:reference "$.type" ; ] ;
    rml:field [
      rml:fieldName "weight";
      rml:reference "$.weight" ;] ; ] .
```

(a) mapping_mixed_view.ttl

```
name, item
alice,"{""type"":""sword"",""weight"":2500}"
alice,"{""type"":""shield"",""weight"":1500}"
bob,"{""type"":""flower"",""weight"": 15 }"
```

(b) mixed_data.json

name	item	item.type	item.weight
alice	{”type”: ”sword”, ”weight”: 1500}	sword	1500
alice	{”type”: ”shield”, ”weight”: 2500}	shield	2500
bob	{”type”: ”flower”, ”weight”: 15}	flower	15

(c) Intermediate representation of :mixedView

Figure 2: Example of a logical view on a logical source containing nested JSON data

To define the expected form of references to input data RML employs the notion of a reference formulation that is a property of every logical source. However, currently a logical source is limited to having a single reference formulation, meaning mixed format data can only be referenced using a query language that supports just one of the formats.

Solution: For every field, the reference formulation can be adapted (Figure 2). The default reference formulation for a field is the reference formulation of its parent, or of the source of the logical view for the fields at the root of the iteration. Thus, every iteration level can iterate over data in a different format.

3.3. Extended joining of data sources

Problem: RML restricts join operations to referencing object maps. Since a referencing object map can only generate an object that is an IRI or blank node subject as specified by a parent triples map, it is not possible to combine data from two sources in one term, use data from a join on another position than the object, or generate a literal using data from a join [3]. Moreover, RML cannot join correctly across hierarchies [2].

Solution: A logical view can be extended with fields from one or more other logical views as a result of a join operation (Figure 3). The logical iteration will be adapted according to the type

```
:csvSource a rml:LogicalSource ;
    rml:source "./csv_data.csv";
    rml:referenceFormulation rml:CSV .
:joinedView a rml:LogicalView ;
  rml:onLogicalSource :csvSource;
  rml:field [
    rml:fieldName "name" ;
    rml:reference "name" ;
  ] ;
  rml:field [
    rml:fieldName "birthyear" ;
    rml:reference "birthyear" ;
  ] ;
  rml:leftJoin [
    rml:parentLogicalView :jsonView ;
    rml:joinCondition [
      rml:parent "name" ;
      rml:child "name" ;
    ] ;
    rml:field [
      rml:fieldName "item_type" ;
      rml:reference "item.type" ;
    ] ; ] .
```

(a) mapping.ttl (part 3)

```
name,birthyear
alice,1995
bob,1999
tobias,2005
```

(b) csv_data.csv

name	birthyear	item_type
alice	1995	sword
alice	1995	shield
bob	1999	flower
tobias	2005	

(c) Intermediate representation of :joinedView

Figure 3: Example of a logical view on a logical source containing CSV data and extended with data from the logical view from Figure 1

of join specified (*left join* or *inner join*). Any needed flattening of hierarchical data is done in the logical view before applying the join operation. Data that originally comes from a different data source is thus treated equally in a joined logical view, allowing more flexibility as to where to apply that joined data.

4. Approach and Implementation

We have designed our proof-of-concept implementation as a standalone application, independent of and complementary to existing RML mapping engines, that can be used as a preprocessing step in a knowledge graph construction pipeline (Figure 4). As such, we designed our proof-of-concept following the state-of-the-art best practices of, amongst others, FunMap [9], i.e. we rely on a set of lossless rewriting rules to push down and materialize the execution of RML Logical Views in the initial step of knowledge graph construction process.

Our implementation, named RML-view-to-CSV, is available online[12] under the permissive MIT license. Our code is built on top of pandas [5], a Python library with data frames for manipulation of structured data sets.

At the moment of writing, our implementation supports one nested source format (JSON),

[12] https://github.com/RMLio/rml-view-to-csv/ (v0.0.0), https://doi.org/10.5281/zenodo.11045497

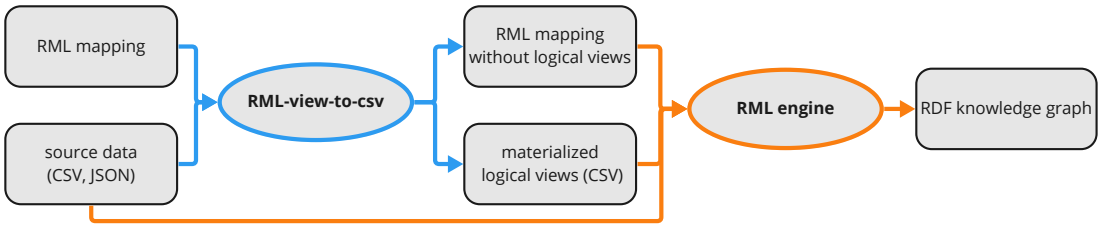

Figure 4: RML-view-to-CSV materializes all logical views specified in a given set of RML mapping rules as CSV files, and rewrites these RML mapping rules to RML mapping rules without logical views. The original source data, the materialized logical views and the new RML mapping without logical views are input for an RML engine to generate RDF graph data.

```
:jsonSource a rml:LogicalSource; rml:source "json_data.json";
    rml:referenceFormulation rml:JSONPath; rml:iterator "$.people[*]".

:jsonView a rml:LogicalSource; rml:source "./view0.csv";
    rml:referenceFormulation rml:CSV.
```

(a) MappingWithoutViews.ttl

```
name,item,item.type,item.weight
alice,"{""type"": ""sword"", ""weight"": 1500}",sword,1500
alice,"{""type"": ""shield"", ""weight"": 2500}",shield,2500
bob,"{""type"": ""flower"", ""weight"": 15}",flower,15
```

(b) view0.csv

Figure 5: Output of RML-view-to-CSV when using Figure 1a and Figure 1b as input

one tabular source format (CSV), and the three important features of the new RML Logical Views module: flattening of nested data, more flexible joining of data sources (also across data hierarchies), and handling of mixed data formats. Its main functionality is the materialization of logical views (Section 4.1). We added two optional functionalities: the elimination of referencing object maps (Section 4.2) and the optimization of logical views based on the linked triples maps (Section 4.3).

4.1. Materialization of logical views

RML-view-to-CSV takes as input a given set of RML mapping rules and the source data used in these mapping rules. It produces CSV files with the intermediate representation of every logical view specified in the RML mapping rules, and a new set of RML mapping rules in which all RML Logical Views are replaced by logical sources (Figure 5).

4.2. Elimination of referencing object maps

With the introduction of logical views, joins of data sources can be expressed within the

```
:map_services1_0 a rml:TriplesMap; rml:logicalSource :source_5.
    rml:subjectMap [rr:template "http://gtfs/services/{service_id}"].

:map_trips_0 a rml:TriplesMap; rml:logicalSource :source_1;
    rml:subjectMap [rml:template "http://gtfs/trips/{trip_id}"];
    rml:predicateObjectMap [
        rml:predicateMap [rml:constant gtfs:service];
        rml:objectMap [
            rml:parentTriplesMap :map_services1_0];
            rml:joinCondition [rml:child "service_id"; rml:parent "service_id"]].
```

(a) MappingWithRefObjectMap.ttl

```
:new_map_0 a rml:TriplesMap; rml:logicalSource :new_child_view_0;
    rml:subjectMap [rml:template "http://gtfs/trips/{trip_id}"];
    rml:predicateObjectMap [
        rml:predicateMap [rml:constant gtfs:service];
        rml:objectMap [rml:template "http://gtfs/services/{service_id_from_parent}"]] .

:new_child_view_0 a rml:LogicalView; rml:onLogicalSource :source_1;
    rml:field
        [rml:fieldName "trip_id"; rml:reference "trip_id"],
        [rml:fieldName "service_id"; rml:reference "service_id"];
    rml:leftJoin [a rml:ViewJoin; rml:parentLogicalView :new_parent_view_0;
        rml:joinCondition [rml:child "service_id"; rml:parent "service_id"];}
        rml:field [rml:fieldName "service_id_from_parent"; rml:reference "service_id"]].

:new_parent_view_0 a rml:LogicalView; rml:onLogicalSource :source_5;
    rml:field [rml:fieldName "service_id"; rml:reference "service_id"].
```

(b) MappingWithoutRefObjectMap.ttl

Figure 6: The referencing object map from MappingWithRefObjectMap.ttl is replaced by an equivalent combination of two new logical views and one new triples map in MappingWithoutRefObjectMap.ttl.

triples map (via referencing object maps) or within the logical view. We added an option to RML-view-to-CSV to delegate the execution of joins expressed in triples maps to logical views. With this option selected, RML-view-to-CSV rewrites the RML mapping rules before materializing the logical views.

It is known that self-join elimination must be performed for time-efficient execution of RML mappings, e.g. the mappings used in the GTFS-Madrid-Benchmark [10, 4, 8]. Therefore, RML-view-to-CSV first eliminates unnecessary self-joins, i.e. when the same logical source is used for the child and the parent triples map, and all involved join conditions use the same references for both parent and child, and either of subject map of the parent triples map or the subject map, predicate map and graph map of the child triples map only mention a subset of the references used in the join conditions, the referencing object map is replaced with a simple object map based on the subject map of the parent triples maps.

All remaining referencing object maps are rewritten as an equivalent combination of two new logical views and a new triples map (Figure 6).

test case number	test case description
RMLLVTC0001	logical view over JSON source
RMLLVTC0002	logical view over JSON source with flattening of nested data
RMLLVTC0003	logical view over CSV source, extended with data from a logical view over JSON source using a left join
RMLLVTC0004	logical view over CSV source, extended with data from a logical view over JSON source using an inner join
RMLLVTC0005	logical view over CSV source, extended with data from a logical view over JSON source using an inner join, and use of references to field indexes

Table 1
Description of the test cases included in the RML Logical View module at the moment of writing.

This option allows us to test our proof-of-concept implementation with existing RML benchmarks, although these existing benchmarks do not include RML Logical Views yet, as logical views are new in RML.

4.3. Optimization of logical views

By default, RML-view-to-CSV does not take the content of triples maps into account. The materialized logical views represent all fields and logical iterations of the declared RML Logical Views. Thanks to this behaviour, we can verify if the processing of the logical views remains aligned with the specification.

However, logical views can contain fields and logical iterations that are not used by any triples map. As the size of the source data impacts the knowledge graph construction process, we added an option in RML-view-to-CSV to eliminate unnecessary fields and logical iterations. With this option selected, RML-view-to-CSV first removes fields that are not used in any triples map linked to the logical view. Then, all duplicate logical iterations are removed, except when any linked triples map produces blank nodes that are not based on a field from the logical view (as this latter case results in always generate a new unique identifier, hence non-duplicate logical iterations).

5. Evaluation

We tested our proof-of-concept implementation against the test cases defined in the RML Logical Views modules, and added an evaluation on the GTFS-Madrid-Bench [11].

5.1. Test cases in the RML Logical Views module

At the moment of writing the RML Logical Views module includes 5 test cases[13] (Table 1). With RML-view-to-CSV as preprocessor to RMLMapper, we generated the expected output for the relevant test cases (i.e. test cases RMLLV0001 to RMLLV0004). We excluded test case RMLLV0005 as it makes use of field indexes, which we have not yet included in our implementation due to

[13]https://github.com/kg-construct/rml-lv/tree/main/test-cases

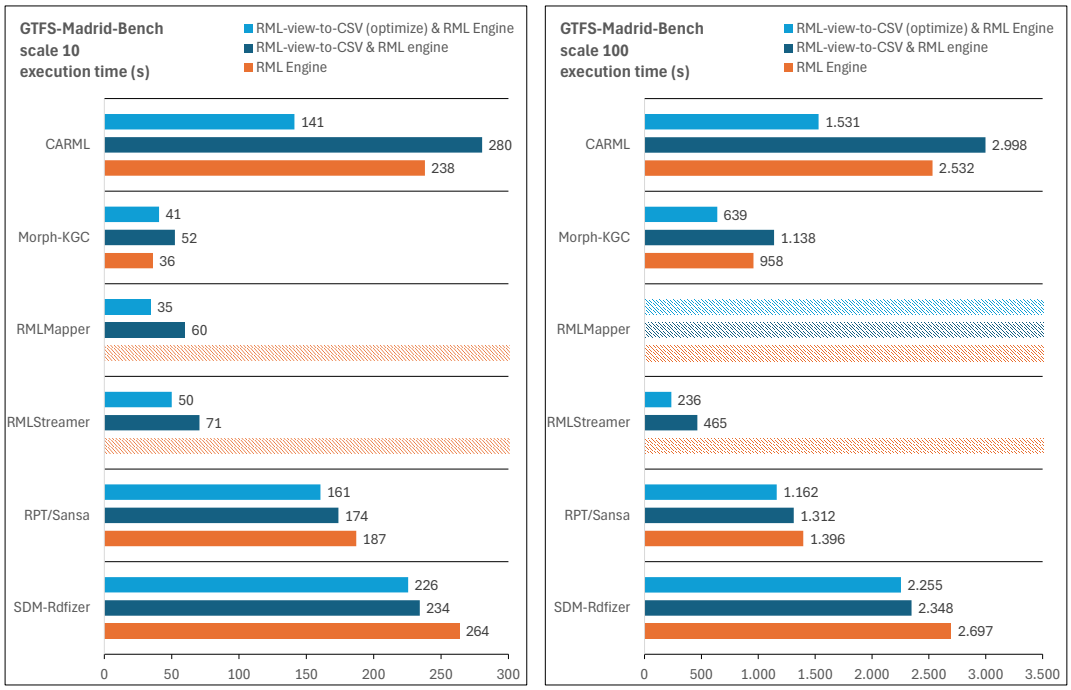

Figure 7: GTFS-Madrid-Bench scale 10 and 100 executed with three pipelines (only RML engine, RML-view-to-csv and RML engine, and RML-view-to-CSV with optimization and RML engine) and six RML engines (Carml, Morph-KGC, RMLMapper, RMLStreamer, RPT/Sansa, and SDM-Rdfizer), average of five runs, time-out after one hour. The pipelines with RML-view-to-CSV with optimization as preprocessor have the lowest execution time.

an unclarity about the expected behaviour[14]. During the evaluation of the test cases, we noticed and corrected human mistakes in the mappings and expected output[15]. This confirms the need and benefit of a proof-of-concept implementation during the development phase of a new RML module: a proof-of-concept implementation helps to spot ambiguities in the specification, ontology and shapes as well as errors in the test cases in an early development stage.

5.2. GTFS-Madrid-Bench

We tested our proof-of-concept implementation on the GTFS-Madrid-Bench, comparing the execution of joins by our implementation versus the execution joins by existing RML engines, i.e. Carml, Morph-KGC, RMLMapper, RMLStreamer, RPT/Sansa, and SDM-Rdfizer[16]. Our test setup included three pipelines: RML engine only, RML-view-to-CSV and RML engine, and RML-view-to-CSV with optimization (Section 4.3) and RML engine. In the latter two pipelines, RML-view-to-CSV first eliminates all referencing object maps in the GTFS-Madrid-Bench mapping,

[14] https://github.com/kg-construct/rml-lv/issues/20
[15] https://github.com/kg-construct/rml-lv/pull/22
[16] https://github.com/carml/carml-jar (V1.3.0), https://github.com/morph-kgc/morph-kgc (v2.6.4), https://github.com/RMLio/rmlmapper-java (v6.3.0), https://github.com/RMLio/RMLStreamer (v2.5.0), https://github.com/SDM-TIB/SDM-RDFizer (v4.7.3.5), and https://github.com/SmartDataAnalytics/RdfProcessingToolkit/ (v.1.9.5) respectively.

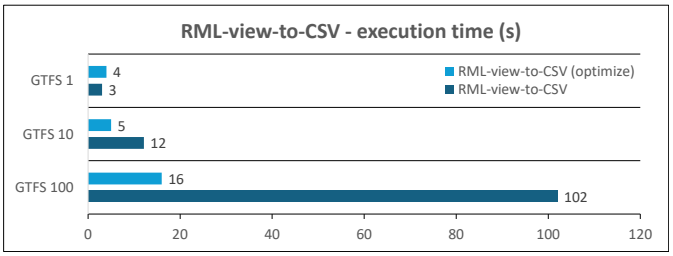

Figure 8: Execution time of RML-view-to-CSV (first step of the pipelines in Figure 7). With the optimization option, RML-view-to-CSV eliminates unnecessary logical view fields and duplicate logical iterations. This reduces its executions time with a factor of six on the largest scale measured.

rewriting them as logical views whenever applicable (Section 4.2). We measured the knowledge graph construction time per pipeline (i.e. including the execution time of RML-view-to-CSV where applicable) and per RML engine for scales 1, 10 and 100 of the GTFS-Madrid-Bench, with CSV as source format, using a device with following specifications: 2 x Hexacore Intel E5645 (2.4GHz) CPU, 24GB RAM, 1x 250GB harddisk. All experiments were performed 5 times and the average of the measurements is reported (Figure 7). The test scripts are available online[17].

RMLMapper and RMLStreamer cannot generate any output for the GTFS-Madrid-Bench within one hour when we count on these RML engines to execute the joins. However, when delegating the joins to RML-view-to-CSV, these mapping engines were able to generate correct output, with timings similar or better to using a state-of-the-art RML engine like Morph-KGC. We note that RMLMapper still cannot handle GTFS scale 100: the RMLMapper loads all data in memory during mapping, and the testing device ran out of memory during GTFS scale 100. We also note that the elimination of unnecessary fields and duplicate logical iterations (RML-view-to CSV with optimization) reduces the execution time of RML-view-to-CSV by a factor of six for scale 100 (Figure 8), and leads to the fastest pipelines in combination with all tested engines.

The combination of RML-view-to-CSV with optimization and RMLStreamer emerges as the most efficient approach[18]. This showcases the potential of modular mapping engines, delegating each task to the most suitable framework, i.e. the dataframes from pandas (used in RML-view-to-CSV) are optimized for data transformations and joins, while streaming and parallelization of Flink enables RMLStreamer to create RDF graph data with a linear scaling of execution time and CPU usage, proportional to the size of the input data, while maintaining a constant memory usage [12].

6. Conclusion

In this paper, we show how the RML Logical Views specification can be implemented and can solve needs that were not solvable yet by RML. The implementation can be realized as a modular

[17] 10.5281/zenodo.10987733

[18] The current set-up of using a preprocessing materialization step prevents the RMLStreamer of currently using this optimization for streaming data.

part of a knowledge graph construction process.

Our proof-of-concept, RML-views-to-CSV, as a preprocessor to any RML engine (that supports CSV input) did not only help to validate and improve the RML Logical Views module, but benchmarks also show performance gains for handling joins between CSV sources. The modular approach showcases the potential of modular mapping engines, allowing to use specialized data structures and delegating each task to the most suitable framework, offering best-of-breed performance enhancements.

The RML Logical Views module is still under development. Its finalization, including formal definitions and more features (e.g. indexes per field, data transformation functions, groups and aggregations), is future work. We intend to gradually integrate the additional features in RML-view-to-CSV, as they are discussed in the W3C Community Group on KGC and described in the RML Logical Views specification. Furthermore, we will investigate whether the detected performance improvements hold as well for sources other than CSV.

We will continue to share our code as inspiration for the developers who want to implement RML Logical Views directly in RML Engines once this new RML module has been finalized.

Acknowledgments

The described research activities were supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project VV023/10), and the European Union's Horizon Europe research and innovation program under grant agreement no. 101058682 (Onto-DESIDE). The authors want to thank David Chaves-Fraga for the discussions that were a source of inspiration for this proof-of-concept implementation.

References

[1] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, B. De Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga, A. Dimou, The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF, in: Proceedings of the International Semantic Web Conference (ISWC), Lecture Notes in Computer Science, Springer, Cham, 2023. doi:10.1007/978-3-031-47243-5_9.

[2] T. Delva, D. Van Assche, P. Heyvaert, B. De Meester, A. Dimou, Integrating nested data into knowledge graphs with RML fields, in: D. Chaves-Fraga, A. Dimou, P. Heyvaert, F. Priyatna, J. Sequeda (Eds.), Proceedings of the 2nd International Workshop on Knowledge Graph Construction co-located with 18th Extended Semantic Web Conference (ESWC 2021), volume 2873, CEUR, 2021. URL: http://ceur-ws.org/Vol-2873/paper9.pdf.

[3] J. Arenas-Guerrero, A. Alobaid, M. Navas-Loro, M. S. Pérez, O. Corcho, Boosting knowledge graph generation from tabular data with RML views, in: The Semantic Web, Springer Nature Switzerland, 2023, pp. 484–501. doi:10.1007/978-3-031-33455-9_29.

[4] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M. S. Pérez, O. Corcho, Morph-KGC: Scalable knowledge graph materialization with mapping partitions, Semantic Web (2022) 1–20. doi:10.3233/sw-223135.

[5] W. McKinney, pandas: a foundational python library for dataanalysis and statistics (2011).

[6] E. Daga, L. Asprino, P. Mulholland, A. Gangemi, Facade-X: An Opinionated Approach to SPARQL Anything, in: Further with Knowledge Graphs – Proceedings of the 17th International Conference on Semantic Systems, 6–9 September 2021, Amsterdam, The Netherlands, volume 53 of *Studies on the Semantic Web*, IOS Press, 2021, pp. 58–73. doi:10.3233/SSW210035.

[7] F. Michel, L. Djimenou, C. Faron-Zucker, J. Montagnat, Translation of Heterogeneous Databases into RDF, and Application to the Construction of a SKOS Taxonomical Reference, in: International Conference on Web Information Systems and Technologies, Springer, 2015, pp. 275–296. doi:10.1007/978-3-319-30996-5_14.

[8] E. Iglesias, S. Jozashoori, M.-E. Vidal, Scaling up knowledge graph creation to large and heterogeneous data sources, Journal of Web Semantics 75 (2023). URL: http://arxiv.org/abs/2201.09694. doi:10.1016/j.websem.2022.100755.

[9] S. Jozashoori, D. Chaves-Fraga, E. Iglesias, M.-E. Vidal, O. Corcho, Funmap: Efficient execution of functional mappings for knowledge graph creation, in: International Semantic Web Conference, Springer, 2020, pp. 276–293. doi:10.1007/978-3-030-62419-4_16.

[10] C. Stadler, L. Bühmann, L.-P. Meyer, M. Martin, Scaling RML and SPARQL-based Knowledge Graph Construction with Apache Spark, in: Knowledge Graph Construction Workshop, co-located with ESWC, 2023.

[11] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus, O. Corcho, Gtfs-madrid-bench: A benchmark for virtual knowledge graph access in the transport domain, Journal of Web Semantics 65 (2020) 100596. doi:10.1016/j.websem.2020.100596.

[12] E. de Vleeschauwer, G. Haesendonck, , D. Van Assche, B. De Meester, RMLStreamer with Reference Conditions in the KGCW Challenge 2023, in: Knowledge Graph Construction Workshop, co-located with ESWC, 2023.