Scaling RML and SPARQL-based Knowledge Graph Construction with Apache Spark

Claus Stadler[1], Lorenz Bühmann[1], Lars-Peter Meyer[1] and Michael Martin[1]

[1]*Institute of Applied Informatics (InfAI), Leipzig*

Abstract

Approaches for the construction of knowledge graphs from heterogeneous data sources range from ad-hoc scripts to dedicated mapping languages. Two common foundations are thereby RML and SPARQL. So far, both approaches are treated as different: On the one hand there are tools specifically for processing RML whereas on the other hand there are tools that extend SPARQL in order to incorporate additional data sources. In this work, we first show how this gap can be bridged by translating RML to a sequence of SPARQL CONSTRUCT queries and introduce the necessary SPARQL extensions. In a subsequent step, we employ techniques to optimize SPARQL query workloads as well as individual query execution times in order to obtain an optimized sequence of queries w.r.t. order and uniqueness of the generated triples. Finally, we present a corresponding SPARQL query execution engine based on the Apache Spark Big Data framework. In our evaluation on benchmarks we show that our approach is capable of achieving RML mapping execution performance that surpasses the current state of the art.

Keywords

RML, SPARQL, Semantic Query Optimization, Big data, Knowledge Graph, RDF

1. Introduction

RML (RDF Mapping Language) and SPARQL (SPARQL Protocol and RDF Query Language) are two important tools in the construction of knowledge graphs and the Semantic Web. RML is an RDF-based language used for mapping data from different sources to RDF. It allows developers to transform and integrate data from various formats, such as CSV, JSON, and/or XML, into RDF triples. On the other hand, SPARQL is a powerful query language used to retrieve data from RDF triple stores. It allows developers to write complex queries that can retrieve data from multiple sources and can be used to construct complex queries across multiple datasets. Together, RML and SPARQL play a crucial role in the construction of knowledge graphs, which are used to represent and interlink data from multiple sources, creating a more comprehensive view of a particular domain. As the amount of data on the web continues to grow, the use of RML and SPARQL is becoming increasingly important for enabling the creation of powerful applications. So far, both approaches are treated as different: On the one hand there are tools specifically for processing RML whereas on the other hand there are tools that extend SPARQL in order to incorporate additional data sources. One of our visions is to enable research into the

Fourth International Workshop on Knowledge Graph Construction, May 28/29, 2023, Hersonissos

cstadler@informatik.uni-leipzig.de (C. Stadler); buehmann@informatik.uni-leipzig.de (L. Bühmann); lpmeyer@infai.org (L. Meyer); martin@infai.org (M. Martin)

0000-0001-9948-6458 (C. Stadler); 0000-0001-5260-5181 (L. Meyer); 0000-0003-0762-8688 (M. Martin)

extent to which findings from SPARQL (query) execution optimizations can be leveraged for RML and vice versa.

The contributions of this work are: (1) We present an approach to translate RML to SPARQL with corresponding optimizations. (2) We introduce the *Not Only RDF SPARQL Extensions* (NORSE) SPARQL extensions for processing CSV, JSON and XML. These extensions are available as a resource that can be plugged in into Apache Jena's ARQ query engine. These extensions comprise additional functions and special SERVICES which are referenced using special IRIs in the `norse:` namespace. The IRI is `https://w3id.org/aksw/norse#`. (3) We furthermore present an Apache Spark-based SPARQL engine that executes NORSE-enhanced SPARQL by leveraging its massive parallel processing model and show that performance- and scalability-wise this approach surpasses the state of the art in several scenarios.

The remainder of this paper is structured as follows: We present related work about mapping languages and tools in the knowledge graph domain in Section 2. The translation of RML models to extended SPARQL CONSTRUCT queries is described in Section 3. Optimizations of query workloads w.r.t. uniqueness and ordering of the produced RDF triples and/or quads are shown in Section 4. In Section 5 we present our implementations for (1) converting RML to SPARQL (2) the NORSE SPARQL extensions for Apache Jena's ARQ query engine and (3) the implementation of a SPARQL engine on Apache Spark using the SANSA Big Data RDF framework. Subsequently, in Section 6 we present an evaluation of our approach based on the GTFS Madrid Bench and one dataset of the SDM Genomic Datasets. We conclude our paper in Section 7 and point out future work.

2. Related Work and Preliminaries

In this section we give an short overview on contemporary mapping languages, RML processors, and the parallelization framework Apache Spark.

2.1. Mapping Languages

Mapping of Data to RDF graphs is usually either done with custom implementations, direct mappings, or using dedicated mapping languages[1].

R2RML is a W3C standard and vocabulary that was originally developed for declaratively mapping relational data to RDF[1]. On the one hand these mappings could be used in ETL processes to dump databases as RDF. On the other hand, the same mappings could be used in SPARQL-to-SQL rewriting, a.k.a. OBDA (ontology-based data access). While R2RML is considered quite verbose, several simplifications have been developed, such as the Starddog Mapping Syntax (SMS, currently in version 2), the Ontop Mapping Language[2], and the Sparqlification Mapping Language[3].

RML[2] is an extension of R2RML which adds additional vocubulary for mapping non-relational data[4, 1]. In essence these additional declarations allow for expressing a mapping of non-relational data (such as XML and JSON) into a relational model where from each row RDF tuples

[1]https://www.w3.org/TR/r2rml/
[2]https://rml.io/specs/rml/

are generated. Like R2RML it suffers from verbosity, for which reason simplified models were derived like YARRRML[5].

SPARQL is a W3C standard for processing (loading, retrieving, transforming and updating) RDF data[3]. Consequently, *SPARQL engines* are systems that can evaluate *SPARQL statements*. Many SPARQL engines feature extension points. Two prominent representatives of the category of SPARQL-based mapping approaches are SPARQL Generate[6] and SPARQL Anything[7].

As there exist many mapping languages[8, 9, 10, 11], the general concept of translations between mapping languages is discussed already by Corcho et al. [12] and Iglesias-Molina et al. [13]. A mapping translation between ShExML[14] and RML is presented by García-González and Dimou [15].

2.2. RML processors and benchmarks

There exist several RML processors [16, 17][4] for the well known extension RML of the W3C standard R2RML. In this paper we are comparing benchmarks with the following: *SDM-RDFizer*[5] is an RML processor implemented in python with optimized data structures and operators. It is developed with scalability and complex data in mind[18]. *Morph-KGC*[6] is an RML processor implemented in python and tries to parallelize the knowledge graph generation by partitioning RML assertions[19]. *CARML*[7], RMLMapper[8].

For measuring the performance of the RML processors we use the following benchmarks: The *Madrid GTFS benchmark*[9] was introduced by Chaves-Fraga et al. [20]. It is based on data from subway network of Madrid and the benchmark data can be scaled up. A survey on RML tools[16] conducted in 2021 evaluated 3 virtualizers and 6 materializers on the GTFS Madrid Benchmark.The *SDM-Genomic-Datasets benchmark*[10] was introduced by Iglesias et al. [18]. This benchmark is motivated from the biomedical domain and based on the Catalogue Of Somatic Mutations In Cancer [11].

2.3. Apache Spark And SANSA

Apache Spark[12] is a framework for high parallelisation. It can scale workload execution from a single node to big clusters. Apache Spark adavanced Hadoop's Map-Reduce paradigm with an abstraction called "resilient distributed datasets (RDDs)" The SANSA framework[21] is an effort to enable various forms of RDF processing on Apache Spark.

[3]https://www.w3.org/TR/sparql11-query/
[4]https://github.com/kg-construct/awesome-kgc-tools
[5]https://github.com/SDM-TIB/SDM-RDFizer
[6]https://github.com/morph-kgc/morph-kgc
[7]https://github.com/carml/carml
[8]https://github.com/RMLio/rmlmapper-java
[9]https://github.com/oeg-upm/gtfs-bench
[10]https://figshare.com/articles/dataset/SDM-Genomic-Datasets/14838342/1
[11]https://cancer.sanger.ac.uk/cosmic
[12]https://spark.apache.org/

3. RML to SPARQL

In this section we describe our approach to translate RML to SPARQL. For this purpose we first briefly summarize the notion of a SPARQL CONSTRUCT query.

3.1. CONSTRUCT Queries

The general form of a CONSTRUCT query is `CONSTRUCT { template } WHERE { pattern }` . Without loss of generality, for this work we assume generalized RDF[13]. Let there be the pairwise disjoint sets of IRIs I, blank nodes B and literals L. The set of *RDF terms* is defined as $T := I \cup B \cup L$. Furthermore, let there be another set of SPARQL variables V. We define the set of *SPARQL terms* $S := T \cup V$. A concrete triple is an element of $T \times T \times T$ whereas a triple pattern is an element of $S \times S \times S$. Likewise a concrete quad is an element of $T \times T \times T \times T$ whereas a quad pattern is an element of $S \times S \times S \times S$. The current SPARQL standard only allows for a CONSTRUCT template to specify the creation of triples using triple patterns. However the importance of this issue has been noted[14]. and several engines already provide support the production of quads as well. The approach presented in the following can be used in either setting, so instead of talking about triple and quad (patterns) we generally speak of *tuple* (patterns). A construct query's template is thus made up of a set up tuple patterns. Substituting all variables of these tuples with RDF terms thus produces a set of concrete tuples.

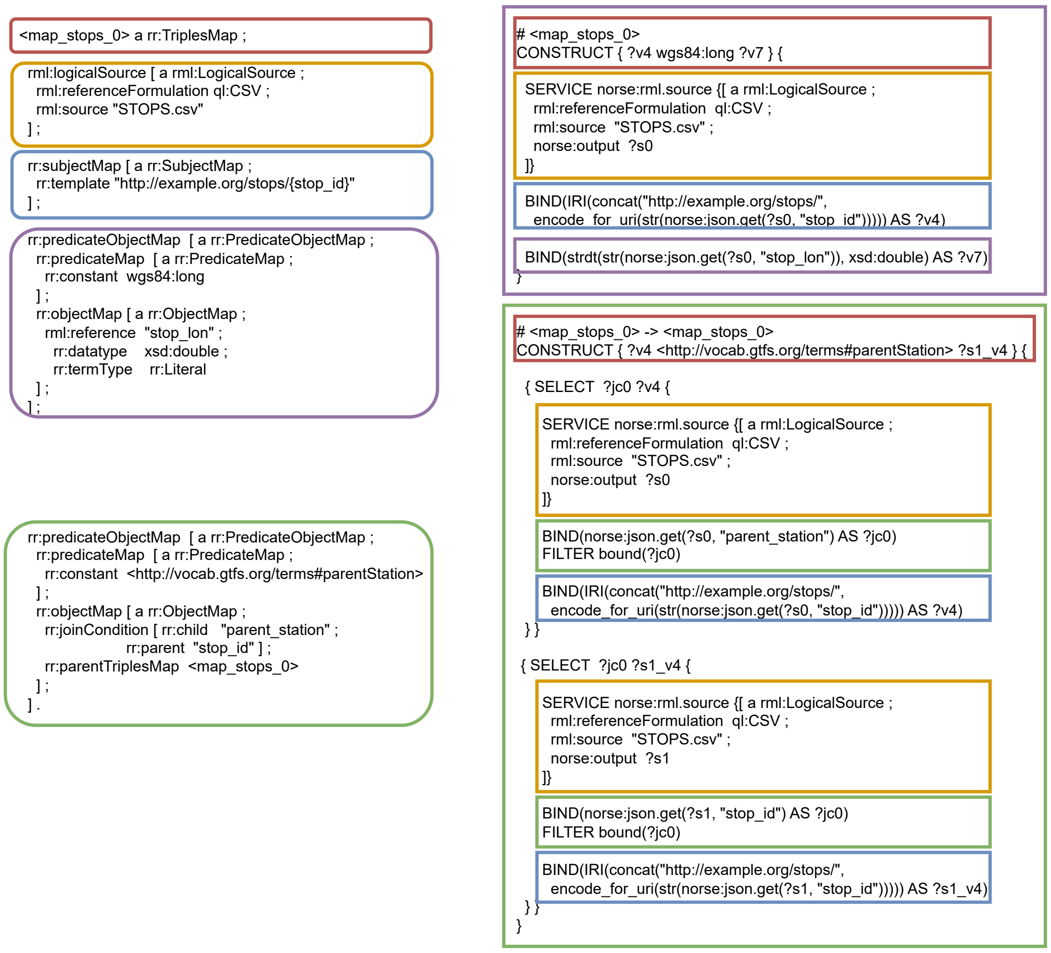
3.2. Translating RML Logical Sources

The two main issues that need to be solved are how to translate (1) RML sources and (2) RML references to SPARQL elements. RML sources conceptually emit a set of records whose attribute access is specified via `rml:references`. On the SPARQL side, the `SERVICE` clause can be used to generate a set of bindings based on its contained pattern. We can thus introduce a special SERVICE IRI `norse:rml.source` which contains a graph pattern that represents an RML source. In addition, we add an additional triple pattern with the special predicate `norse:output` in order to bind the source records as RDF terms to a SPARQL variable. Therefore, we introduce custom XML[15] and JSON datatypes as well as corresponding functions, namely `norse:json` and `norse:xml` to capture XML and JSON data efficiently, respectively.

3.3. Translating RML TermMaps

RML TermMaps – as the name suggests – specify how to map the referenced data to RDF terms. SPARQL operates at the level of bindings where variables are bound to RDF terms. Hence, we can represent RML TermMaps in SPARQL by using `BIND` to define variables as expressions over a source's data. SPARQL provides the functions `IRI`, `STRDT` and `STRLANG` for the construction of RDF terms. Consequently, every TriplesMap's term map can be represented using a freshly allocated variable that is bound to a corresponding definition using a SPARQL `BIND` statement.

[13]https://www.w3.org/TR/rdf11-concepts/#section-generalized-rdf
[14]https://github.com/w3c/sparql-12/issues/31
[15]Jena's implementation of the `rdf:xmlLiteral` datatype only stores XML as a string which is not suited for efficient XPath evaluation.



```
<map_stops_0> a rr:TriplesMap ;

  rml:logicalSource [ a rml:LogicalSource ;
    rml:referenceFormulation ql:CSV ;
    rml:source "STOPS.csv"
  ] ;

  rr:subjectMap [ a rr:SubjectMap ;
    rr:template "http://example.org/stops/{stop_id}"
  ] ;

  rr:predicateObjectMap  [ a rr:PredicateObjectMap ;
    rr:predicateMap  [ a rr:PredicateMap ;
      rr:constant  wgs84:long
    ] ;
    rr:objectMap [ a rr:ObjectMap ;
      rml:reference  "stop_lon" ;
        rr:datatype    xsd:double ;
        rr:termType    rr:Literal
    ] ;
  ] ;

  rr:predicateObjectMap  [ a rr:PredicateObjectMap ;
    rr:predicateMap  [ a rr:PredicateMap ;
      rr:constant  <http://vocab.gtfs.org/terms#parentStation>
    ] ;
    rr:objectMap [ a rr:ObjectMap ;
      rr:joinCondition [ rr:child   "parent_station" ;
                         rr:parent  "stop_id" ] ;
      rr:parentTriplesMap  <map_stops_0>
    ] ;
  ] .
```

```
# <map_stops_0>
CONSTRUCT { ?v4 wgs84:long ?v7 } {
  SERVICE norse:rml.source {[ a rml:LogicalSource ;
    rml:referenceFormulation  ql:CSV ;
    rml:source  "STOPS.csv" ;
    norse:output  ?s0
  ]}
  BIND(IRI(concat("http://example.org/stops/",
    encode_for_uri(str(norse:json.get(?s0, "stop_id"))))) AS ?v4)
  BIND(strdt(str(norse:json.get(?s0, "stop_lon")), xsd:double) AS ?v7)
}
```

```
# <map_stops_0> -> <map_stops_0>
CONSTRUCT { ?v4 <http://vocab.gtfs.org/terms#parentStation> ?s1_v4 } {
  { SELECT  ?jc0 ?v4 {
    SERVICE norse:rml.source {[ a rml:LogicalSource ;
      rml:referenceFormulation  ql:CSV ;
      rml:source  "STOPS.csv" ;
      norse:output  ?s0
    ]}
    BIND(norse:json.get(?s0, "parent_station") AS ?jc0)
    FILTER bound(?jc0)
    BIND(IRI(concat("http://example.org/stops/",
      encode_for_uri(str(norse:json.get(?s0, "stop_id"))))) AS ?v4)
  } }
  { SELECT  ?jc0 ?s1_v4 {
    SERVICE norse:rml.source {[ a rml:LogicalSource ;
      rml:referenceFormulation  ql:CSV ;
      rml:source  "STOPS.csv" ;
      norse:output  ?s1
    ]}
    BIND(norse:json.get(?s1, "stop_id") AS ?jc0)
    FILTER bound(?jc0)
    BIND(IRI(concat("http://example.org/stops/",
      encode_for_uri(str(norse:json.get(?s1, "stop_id"))))) AS ?s1_v4)
  } }
}
```

Figure 1: Juxtaposition of an RML document and its representation as SPARQL queries. The RML join condition is transformed into a natural join of SPARQL graph patterns where the same variable (`?jc0`) is bound on both sides.

A summary for mapping RML term maps to SPARQL is shown in Figure 2. The function `access` is thereby a placeholder that needs to be replaced with a concrete variant based on the type of the logical source (e.g. XML, JSON, CSV) as explained in Section 3.4.

3.4. Translating RML References

The concrete expression of the `access` function depends on the logical sources' format. Because the format is specified, we can rewrite `access` with the following concrete functions, where `REF` is substituted with reference expression string.

- JSON: `norse:json.path(?x, "$['REF']")` If the result of the JSON path evaluation is a primitive JSON object then it is converted to an RDF term. JSON null is effectively treated as "unbound". Otherwise, for JSON arrays and objects an RDF term of type `norse:json` is returned.

- `[ rr:reference "ref" ]` → `BIND(access(?source, "ref") AS ?v0)`
- `[ rr:reference "ref" ; rr:termType rr:IRI ]` →
  `BIND(IRI(access(?source, "ref")) AS ?v0)`
- `[ rr:reference "ref" ; rr:termType rr:BlankNode ]` →
  `BIND(BNODE(access(?source, "ref")) AS ?v0)`
- `[ rr:reference "ref" ; rr:datatype xsd:float ]` →
  `BIND(STRDT(access(?source, "ref"), xsd:float) AS ?v0)`
- `[ rr:reference "ref" ; rr:language "en" ]` →
  `BIND(STRLANG(access(?source, "ref"), access(?source, "lang")) AS ?v0)`

Figure 2: Translating RML term maps to SPARQL BIND expressions.

- CSV: In our approach we represent CSV rows as JSON documents and thus access could be performed using the aforementioned `norse:json.path` function. However, in order to avoid to overhead of JSON path evaluation we instead introduce the function `norse:json.get(?obj, "REF")` for accessing a JSON object's immediate keys directly.
- XML: `norse:xml.text(norse:xml.path(?xmlNode, "//:REF"))` The result of the xpath expression is generally another XML node, such as `<lon>42.5</lon>`. The `norse:xml.text` function ensures that the (string) representation is extracted.

3.5. Translating RefObjectMaps (Joins)

Joins in RML are declared using `rr:RefObjectMap`. The outcome of the translation of an RML join is a CONSTRUCT query which involves a natural join based on the references to the sources that act as child and parent as shown in Section 3.2. Every `rr:RefObjectMap` results in an independent CONSTRUCT query with only one tuple pattern in its template.

3.5.1. Duplicate-Reducing Self Join Elimination

For time-efficient execution of RML mappings, such ones used in the GTFS-Madrid-Benchmark, it is known that a form of self-join elimination must be performed[18][19]. Given an arbitrary relation, such as a CSV file, it is not generally possible to assert the uniqueness of columns because usually there is no metadata available.

As a consequence, schematic self-join elimination based on uniqueness constraints is typically not possible without prior computation of metadata. However, an RML join condition can be generally omitted if the following conditions hold:

- The parent TriplesMap's logical source as the same as that of the child TriplesMap
- All involved join conditions use the same reference expression for both parent and child, such as `rr:parent = "ref" ; rr:child "ref"`.

- Either of the involved subject maps only mentions a subset of of the references used in the join.

In such a case a referencing object map can be replaced with a simple object map of the referenced TriplesMap's subject map. Note, that this is not an equivalence transformation as it may reduce the cardinalities of bindings in the result set.

4. Optimizing SPARQL CONSTRUCT Query Workloads

By transforming RML mappings into a set of SPARQL the problem of efficient RML mapping execution becomes one of optimizing a workload of SPARQL CONSTRUCT queries w.r.t. uniqueness and ordering of the produced tuples.

SPARQL itself does directly provide DISTINCT and ORDER BY operators for CONSTRUCT queries. However, recent advancements towards the next version of SPARQL make it possible to convert CONSTRUCT queries into equivalent SELECT ones that project three or for variables for triples or quads, respectively.

4.1. Merging CONSTRUCT Queries using LATERAL

There are two main issues with SPARQL 1.1 CONSTRUCT queries for the purpose of producing sorted and unique knowledge graph output:

- Although an ORDER BY and/or DISTINCT can be specified, these solution modifiers only apply to the underlying bindings and not the produced tuples. This is particularly an issue when a CONSTRUCT query's template mentions multiple RDF tuple patterns. In other words, it is not generally possible to "push" an implied DISTINCT operation of a CONSTRUCT query down into an explicit one over its graph pattern.

- While multiple SELECT queries can be combined with UNION, no such operator exists for CONSTRUCT queries.

These two issues make it difficult to devise a general procedure to efficiently combine tuples generated by a set of CONSTRUCT queries. A recent effort towards the next version of the SPARQL specification is the introduction of the `LATERAL` keyword which is already supported by a few SPARQL engines[16]. The keyword's corresponding operation first evaluates the left-hand-side. Each obtained binding is then used to substitute all (in-scope) variables on the right-hand-side before the substituted right-hand-side is evaluated:

$$[[\texttt{Lateral}(\texttt{left}, \texttt{right})]] := \{\mu_l \cup \mu_2 | \mu_1 \in [[\texttt{left}]] \texttt{ and } \mu_2 \in [[\texttt{subst}(\texttt{right}, \mu_1)]]\}$$

With this keyword it is now possible to "normalize" *any* CONSTRUCT query into an equivalent one with a canonical template of the form `GRAPH ?g { ?s ?p ?o }` for quad-based approaches or `?s ?p ?o` for triple-based ones. Without loss of generality, any clashes in variable naming can be resolved with appropriate renaming. This way, a set of normalized CONSTRUCT queries

[16]https://github.com/w3c/sparql-12/issues/100

can be UNION'd simply by creating a UNION of their graph patterns and adding the uniform template. The operations ORDER BY and DISTINCT can be applied likewise. The general CONSTRUCT-to-LATERAL rewrite is described in Figure 3. Note, that `DEFAULT` is thereby an implementation dependent constant for the default graph[17]. Given a set of CONSTRUCT queries, a generic *merge* can be accomplished based on their lateral form as shown in Figure 4.

```
CONSTRUCT {
  s1 p1 o1
  ...
  GRAPH gn { sn pn on }
} WHERE
  PATTERN
}
```

```
CONSTRUCT { GRAPH ?g { ?s ?p ?o } }
WHERE {
  SELECT DISTINCT ?g ?s ?p ?o {
    PATTERN
    LATERAL {
        { BIND(DEFAULT AS ?g)
          BIND(s1 AS ?s) BIND(p1 AS ?p) BIND(o1 AS ?o) }
      UNION
        ...
      UNION
        { BIND(gn AS ?g)
          BIND(sn AS ?s) BIND(pn AS ?p) BIND(on AS ?o) }
    }
  } ORDER BY ?s ?p ?o ?g
}
```

Figure 3: Rewrite of a CONSTRUCT query to its LATERAL form. The identifiers s_i, p_i, o_i and g_i used in the snippet on the left are placeholders for any SPARQL term. The use of DISTINCT and ORDER BY is exemplary to demonstrate the production of truly unique and ordered "intra-query" tuples which is hard to achieve by conventional means if at all. The identifier `DEFAULT` is meant as a placeholder for to the default graph.

4.2. Partitioning Mappings

In Section 3 we showed how to translate RML TriplesMaps into a set of SPARQL CONSTRUCT queries. Furthermore, we described how a set of CONSTRUCT queries can be combined into a single one using the novel `LATERAL` keyword. This tooling is already sufficient to produce a single CONSTRUCT query from any RML document where DISTINCT and ORDER BY is applied at the top of its SPARQL algebra expression. However, if it is known that two queries produce disjoint sets of RDF tuples then DISTINCT (and possibly ORDER BY) can be applied independently and their results can be UNION'd. As this leads to operations on fewer data it can significantly improve performance.

In order to achieve this goal it is necessary to obtain a description of the possible set of RDF tuples that can be created from a CONSTRUCT query. For this purpose we present an interval-based model where the set(s) of possible RDF terms (produced by a tuple's component) are represented as intervals on a generalized number line. For brevity, we only focus on sorting RDF terms based on the lexical space of their N-Quads serialization.

[17] See discussion https://github.com/w3c/sparql-12/issues/43

```
# Query $a$
CONSTRUCT {
  $s_{a_1}$ $p_{a_1}$ $o_{a_1}$
  ...
  $s_{a_n}$ $p_{a_n}$ $o_{a_n}$
} WHERE { $\mathrm{PATTERN_a}$ }

# Query $b$
CONSTRUCT {
  $s_{b_1}$ $p_{b_1}$ $o_{b_1}$
  ...
  $s_{b_m}$ $p_{b_m}$ $o_{b_m}$
} WHERE { $\mathrm{PATTERN_b}$ }
```

```
CONSTRUCT { ?s ?p ?o }
WHERE {
  SELECT DISTINCT ?s ?p ?o {
      { $\mathrm{PATTERN}_a$
        LATERAL { { BIND($s_{a_i}, p_{a_i}, o_{a_i}$ AS
          ?s, ?p, ?o) } UNION ... } } # for i in 1..n
    UNION
      { $\mathrm{PATTERN}_b$
        LATERAL { { BIND($s_{b_j}, p_{b_j}, o_{b_j}$ AS
          ?s, ?p, ?o) } UNION ... } } # for j in 1..m
  } ORDER BY ?s ?p ?o
}
```

Figure 4: Generic merge of two (triple-based) CONSTRUCT queries into a single one based on their LATERAL form. The use of DISTINCT and ORDER BY is exemplary to demonstrate the production of truly unique and ordered "inter-query" tuples.

For example, from an expression such as `BIND(IRI(CONCAT("gtfsbench/", ?id)) AS ?x)` we can derive that `?x` may be any of (1) `unbound`[18] or (2) an IRI with a string value in the interval $[\text{"gtfsbench/" .. "gtfsbench0"})$ (under lexicographic order), where $[$ denotes a closed boundary and $)$ an open one, and "0" is the successor character of "/" in (the ASCII-subset of) UTF-8.

Given a tuple of a construct template, we can thus determine a set of possible values for each of its components. If the construct template has multiple quads then we can take the component-wise union of the intervals in order to obtain a single description of its producible quads. If a variable's set of values is unknown we can gracefully represent it as an interval covering the complete range such as $(-\infty\ ..\ +\infty)$. This way, we can "project" every CONSTRUCT query to an interval. Figure 5 (a) shows a concrete projection based on a subset of the mappings of the GTFS-Madrid-Bench. Each interval corresponds to one or more CONSTRUCT queries. Figure 5 (b) shows an abstract example where intervals overlap. A set of queries with overlapping ranges forms a *partition* and can be merged as shown in Figure 4 for the sake of applying DISTINCT and ORDER BY. Extending this approach to SPARQL is possible, but requires segmentation of the "number line" into sub-intervals for each RDF term type and RDF literal datatype.

4.3. Optimizing DISTINCT by Pulling Up BINDs

A short-coming of the generated queries is that the DISTINCT operation runs over variables that may be are assigned to constants. By "pulling" such definitions up in the algebra DISTINCT can operate on significantly fewer data, which in general increases performance by means of lowering the computational overhead. Figure 6 shows an example of rewrite rules we use for optimization. Note, that `EXTEND` is the algebraic correspondence to the `BIND` syntax[19]. Note, that

[18]if `?id` is not a string because SPARQL requires arguments of CONCAT to be strings and
[19]https://www.w3.org/TR/sparql11-query/#sparqlAlgebra

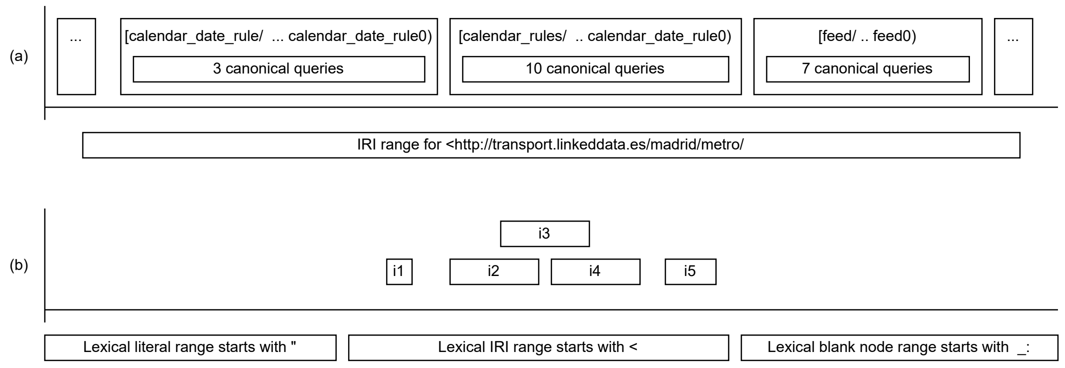

(a)
...
[calendar_date_rule/ ... calendar_date_rule0)
3 canonical queries
[calendar_rules/ .. calendar_date_rule0)
10 canonical queries
[feed/ .. feed0)
7 canonical queries
...
IRI range for <http://transport.linkeddata.es/madrid/metro/
(b)
i3
i1
i2
i4
i5
Lexical literal range starts with "
Lexical IRI range starts with <
Lexical blank node range starts with _:

Figure 5: (a) An excerpt of the concrete range partitioning of canonical queries obtained from the GTFS-Madrid-Bench mapping based on their produced subjects. (b) An abstract model for RDF Term serialization in N-Quads with example intervals of which some (i3, i4, i5) overlap.

more sophisticated rules can be devised to split expressions such as `CONCAT(const, ...)` into a constant and variable part where the constant part can be pulled up.

- `DISTINCT(EXTEND(var, constant, subOp))` → `EXTEND(var, constant, DISTINCT(subOp))`
- `UNION(EXTEND(var, constant, left), EXTEND(var, constant, right))` →
  `EXTEND(var, constant, UNION(left, right))`
- `EXTEND(var, non-constant-expr, EXTEND(var, constant, subOp))` →
  `EXTEND(var, constant, EXTEND(var, non-constant-expr, subOp))`

Figure 6: A brief excerpt of algebra rewrite rules used to pull EXTEND up.

5. Implementation

In this section we provide a brief overview of our related implementations: The NORSE Sparql Extensions, the implementation of the SANSA binding engine (SaBiNe) for evaluating SPARQL on Apache Spark, and finally the RDF Processing Toolking *RPT*[20] which bundles all components together – including the RML to SPARQL tooling – into a single command line toolkit.

NORSE SPARQL Extensions and RPT JenaX[21] is our project of unofficial extensions for the Apache Jena project. Among its features are the *NORSE* SPARQL extensions. Adding the plugin module as a Maven dependency enhances a Jena-based project with the datatypes and functions for processing CSV, XML and JSON[22].

[20]https://github.com/SmartDataAnalytics/RdfProcessingToolkit
[21]https://github.com/Scaseco/jenax
[22]https://mvnrepository.com/artifact/org.aksw.jenax/jenax-arq-plugins-bundle

- [[`SERVICE norse:rml.source {[ ... norse:output ?s ]}`]] `:= Create a RDD<Binding>`
  `where ?s is bound to records of the specified RML source.`

- [[`FILTER(subOp, expr)`]] `:=` [[`subOp`]]`.filter(`$\mu \to \mathrm{exprEval}(\mathrm{expr}, \mu) == true$`)`

- [[`JOIN(left, right)`]] `:=` [[`left`]]`.mapToPair(`$\mu_1 \to \langle \Pi_J(\mu_1), \mu_1 \rangle$`).join(`[[`right`]]`.`
  `mapToPair(`$\mu_2 \to \langle \Pi_J(\mu_2), \mu_2 \rangle$`).map(`$\langle \mathrm{key}, \langle \mu_1, \mu_2 \rangle \rangle \to \mu_1 \cup \mu_2$`)`

  where J is the set of join variables $\mathrm{vars(left)} \cap \mathrm{vars(right)}$ and $\Pi_J(\mu)$ is the projection of a binding to these variables.

- [[`PROJECT(subOp, vars)`]] `:=` [[`subOp`]]`.map(`$\mu \to \Pi_{vars}(\mu)$`)`

- [[`DISTINCT(subOp)`]] `:=` [[`subOp`]]`.distinct()`

- [[`LATERAL(left, right)`]] `:= if right is BGP-free then` [[`left`]]`.mapPartitions(`
  $\mu \to \{\mu \cup \nu | \forall \nu \in \mathrm{convEval}(\mathrm{subst}(\mathrm{right}, \mu)))\}$`)`

  where $\mathrm{convEval}$ is conventional SPARQL evaluation into a (Java) collection of bindings rather than a Spark RDD.

- [[`EXTEND(var, expr, subOp)`]] `:=` [[`subOp`]]`.map(`$\mu \to \mu \cup \{\mathrm{var} \to \mathrm{exprEval}(expr, \mu)\}$`)`

Figure 7: Evaluation of selected SPARQL operations with Apache Spark

Evaluating SPARQL with SANSA and Apache Spark Our approach to evaluating SPARQL in Spark is a direct one: A SPARQL result set is represented as an `RDD<Binding>`. On this basis we present a translation function [[.]] that recursively translates SPARQL algebra operations to operations on (Java) RDDs. The SANSA Framework thereby provides several features that enable use of functionality from Apache Jena with Apache Spark, such as serializers for SPARQL bindings and algebra expressions. Figure 7 shows an excerpt for the evaluation of the SPARQL operations most relevant to RML execution on Apache Spark .

RDF Processing Toolkit RPT is the integration project that provides a powerful frontend for both Jena's ARQ and SANSA's SPARQL engines. Both engines support the NORSE and the RML extensions, however only the latter supports parallelization. Example usage of the tooling is shown in Listing 1.

Listing 1: Example for using RPT to translate and execute RML

```
rpt rmltk rml to sparql mapping.rml.ttl > raw.rq
rpt rmltk optimize workload raw.rq --no-order > mapping.rq
JAVA_OPTS="-Xmx16g" rpt integrate mapping.rq --out-file rpt-arq.nt
JAVA_OPTS="-Xmx16g" rpt sansa query mapping.rq --out-file rpt-sansa.nt
```

6. Evaluation

We evaluate our approach on the GTFS-Madrid-Bench and one of the largest datasets of SDM-Genomics-Datasets[23]. For this purpose we converted the benchmark's RML files to extended SPARQL and ran them using Jena's ARQ and SANSA's SPARQL engine as shown in Listing 1. In a first step we evaluated several RML tools on a server with 128GB RAM, AMD Ryzen 9 5950X 16-Core CPU and SSD storage running Ubuntu 20.04. In order to establish comparability, we used all tools' native unique output feature[24]. The results for the scale factors 1, 10, 100, 300 are shown in Figure 8. We also attempted to evaluate RocketRML, however we ran into memory issues with it[25]. As for *RMLStreamer*[22], on the one hand it requires a Flink setup and on the other hand the initially obtained execution suggested that it lacks the self-join elimination - similar to RMLMapper.

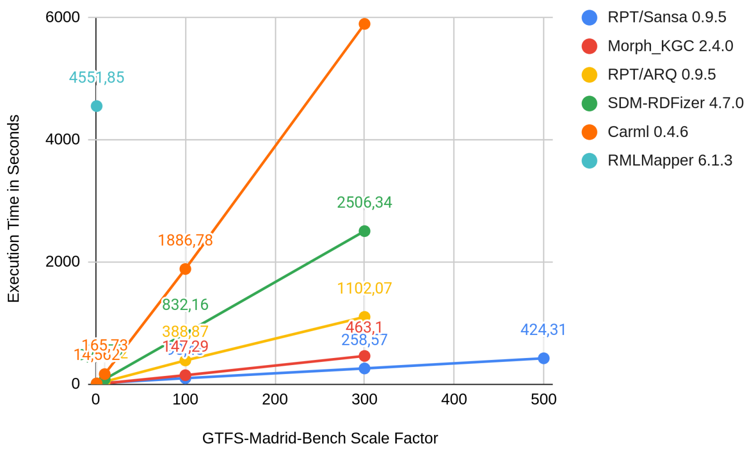
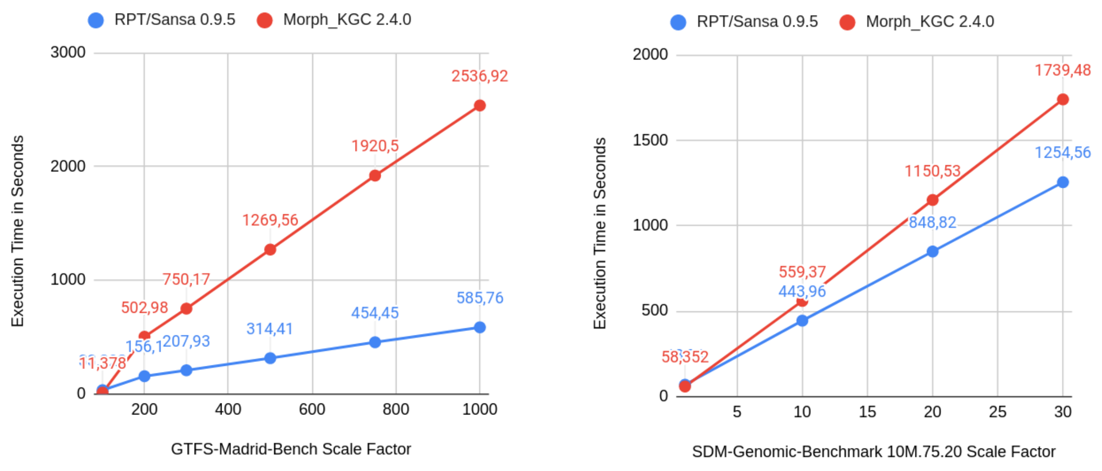


Figure 8: Comparison of RML mapping tool performance with 128G RAM. On scale factor 500, RPT/ARQ and Morph_KGC ran out of memory. Carml and SDM-RDFizer were already a magnitude slower on scale factor 300 and were not further evaluated. RMLmapper already exhibited a very high execution time on scale factor 1.

In a subsequent step, we evaluated the fasted approaches which are the ones rely on parallel processing, namely *Morph_KGC* and *RPT/SANSA*. For this evaluation we needed a machine with more RAM and its specs were: Ubuntu 22.04, 2x Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz (totalling to 64 threads) and 512GB DDR4 RAM at 2133 MHz. In order to avoid I/O bounds in parallel processing, we performed the experiments for both tools with the benchmark datasets served from the default RAM drive `/dev/shm`. With this machine it was possible to scale up to factor 1000. In addition, we evaluated the tools on the SDM-Genomic-Dataset as this is includes a workload that does not involve joins but many duplicates. As can be seen from Figure 9 the execution times for both tools on both workloads converge to scaling linearly.

[23]The used files are 75percent_of_records_with_duplicate_and_each_duplicate_being_repeated_20times.csv and 4POM_Normal.ttl
[24]The only exception was Carml for which we appended a `sort -u` step
[25]https://github.com/semantifyit/RocketRML/issues/44



(a) (b)

Figure 9: Execution time of RPT/SANSA and Morph_KGC with 512G RAM and *all* data (including temporary) in a RAM disk.

On smaller sizes Morph_KGC outperforms RPT/SANSA however with increasing data scale the Apache Spark-based approach gains an advantage. However, on the workload that is mainly about duplicate removal the benefit is quite small considering CPU usage: Morph's average CPU usage in both scenarios is roughly around 400% whereas RPT/SANSA's is around 4000%. There are many aspects that can cause this significant difference: As a primary source we suspect Apache Spark's processing model for DISTINCT which relies on hash partitioning and shuffling of data which involves (de-)serialization overhead. This introduces a significant overhead when compared to e.g. simply keeping records in an in-memory hash set. Furthermore, building on an existing SPARQL engine can be both a blessing and a curse: In constrast to Morph_KGC, Jena validates all created literal and warns about invalid ones. As a further example, one issue which we reported to Jena was about intermediate JSON objects that were needlessly materialized to strings during SPARQL evaluation[26]. Overall, further investigations are necessary to substantiate the explanations of the differences.

7. Conclusions and Future Work

In this work we showed that with the conversion of RML to SPARQL construct queries we can leverage suitably enhanced SPARQL engines for the task of knowledge graph construction. We further showed that by transforming CONSTRUCT queries to their "lateral" form it is now finally possible to "merge" CONSTRUCT queries and remove duplicates which has direct applications in knowledge graph construction. Using query workload analysis we can push down DISTINCT operations such that this expensive operation can be computed on smaller

[26]https://github.com/apache/jena/pull/1802

RDF graphs. We showed that the same query workload can be executed on different engines yielding the same result sets however with significantly different performance characteristics. By leveraging a Big Data framework this approach can outperform state of the art approaches. We emphasize that as part of this work we contributed the SERVICE extension plugin as well as the initial LATERAL implementation to Apache Jena. As one direction of future work we plan to optimize the generated SPARQL algebra further as to minimize the amount of data that has to be processed in DISTINCT and ORDER BY operations. Also, as shown in the evaluation, the improved overall performance comes at the cost of significant higher resource usage for which we plan in-depth investigation of the reasons and possible mitigation approaches such as using custom Spark operator implementations. Furthermore, we identify the need for standardization of SPARQL for heterogeneous data as this would make not only make it possible to transform RML to SPARQL in a truly interoperable way, but also provide a common ground for query and query workload optimization.

Acknowledgments

The authors acknowledge the financial support by the German Federal Ministry for Economic Affairs and Energy in the project Coypu (project number 01MK21007A and by the German Federal Ministry of Education and Research in the project StahlDigital (project number 13XP5116B).

References

[1] A. Dimou, R2RML and RML comparison for RDF generation, their rules validation and inconsistency resolution, ArXiv (2020). doi:10.48550/ARXIV.2005.06293.

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

[3] C. Stadler, J. Unbehauen, P. Westphal, M. A. Sherif, J. Lehmann, Simplified RDB2RDF mapping., LDOW@ WWW 1409 (2015). URL: https://ceur-ws.org/Vol-1409/paper-09.pdf.

[4] A. Dimou, M. V. Sande, P. Colpaert, R. Verborgh, E. Mannens, R. V. de Walle, RML: A generic language for integrated RDF mappings of heterogeneous data, in: Proceedings of the Workshop on Linked Data on the Web (LDOW) 2014, co-located with the 23rd International WWW Conference, 2014. URL: https://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[5] P. Heyvaert, B. De Meester, A. Dimou, R. Verborgh, Declarative rules for linked data generation at your fingertips!, in: The Semantic Web: ESWC 2018 Satellite Events: ESWC 2018 Satellite Events, Heraklion, Crete, Greece, June 3-7, 2018, Revised Selected Papers 15, Springer, 2018, pp. 213–217.

[6] M. Lefrançois, A. Zimmermann, N. Bakerally, A sparql extension for generating RDF from heterogeneous formats, in: The Semantic Web: 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28–June 1, 2017, Proceedings, Part I 14, Springer, 2017, pp. 35–50.

[7] L. Asprino, E. Daga, A. Gangemi, P. Mulholland, Knowledge graph construction with a façade: A unified method to access heterogeneous data sources on the web, ACM Trans. Internet Technol. (2022). doi:10.1145/3555312.

[8] B. D. Meester, P. Heyvaert, R. Verborgh, A. Dimou, Mapping languages: Analysis of

comparative characteristics, in: KGB@ESWC, 2019. URL: http://ceur-ws.org/Vol-2489/paper4.pdf.

[9] D. Chaves-Fraga, Knowledge Graph Construction from Heterogeneous Data Sources exploiting Declarative Mapping Rules, phdthesis, Universidad Politécnica de Madrid, 2021. doi:10.20868/UPM.thesis.67890.

[10] A. Iglesias-Molina, A. Cimmino, E. Ruckhaus, D. Chaves-Fraga, R. García-Castro, O. Corcho, An ontological approach for representing declarative mapping languages, Semantic Web (2022) 1–31. doi:10.3233/sw-223224.

[11] D. V. Assche, T. Delva, G. Haesendonck, P. Heyvaert, B. D. Meester, A. Dimou, Declarative RDF graph generation from heterogeneous (semi-)structured data: A systematic literature review, Journal of Web Semantics 75 (2023) 100753. doi:10.1016/j.websem.2022.100753.

[12] O. Corcho, F. Priyatna, D. Chaves-Fraga, Towards a new generation of ontology based data access, Semantic Web 11 (2020) 153–160. doi:10.3233/sw-190384.

[13] A. Iglesias-Molina, A. Cimmino, Ó. Corcho, Devising mapping interoperability with mapping translation, in: KGCW@ESWC, 2022. URL: https://ceur-ws.org/Vol-3141/paper6.pdf.

[14] H. García-González, A ShExML perspective on mapping challenges: Already solved ones, language modifications and future required actions (2021). URL: http://ceur-ws.org/Vol-2873/paper2.pdf.

[15] H. García-González, A. Dimou, Why to tie to a single data mapping language? enabling a transformation from shexml to rml, in: International Conference on Semantic Systems, 2022. URL: https://ceur-ws.org/Vol-3235/paper11.pdf.

[16] J. Arenas-Guerrero, A. Iglesias-Molina, J. Toledo, L. Pozo-Gilo, D. Donà, Ó. Corcho, D. Chaves-Fraga, Knowledge graph construction with r2rml and rml: An etl system-based overview, in: KGCW@ESWC, 2021. URL: http://ceur-ws.org/Vol-2873/paper11.pdf.

[17] E. Iglesias, S. Jozashoori, M.-E. Vidal, Scaling up knowledge graph creation to large and heterogeneous data sources, Journal of Web Semantics 75 (2023) 100755. doi:10.1016/j.websem.2022.100755.

[18] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, D. Collarana, M.-E. Vidal, SDM-RDFizer: An RML interpreter for the efficient creation of rdf knowledgegraphs, in: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, ACM, 2020. doi:10.1145/3340531.3412881.

[19] 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.

[20] 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.

[21] e. a. Lehmann Jens, Distributed semantic analytics using the sansa stack, in: The Semantic Web–ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II 16, Springer, 2017, pp. 147–155.

[22] G. Haesendonck, W. Maroy, P. Heyvaert, R. Verborgh, A. Dimou, Parallel RDF generation from heterogeneous big data, in: Proceedings of the International Workshop on Semantic Big Data, ACM, 2019. doi:10.1145/3323878.3325802.