

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

Claus Stadler¹, Lorenz Bühmann¹, Lars-Peter Meyer¹ and Michael Martin¹

¹Institute for 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 with respect to the 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, RDF, Knowledge Graph, Big data, Semantic Query Optimization, Apache Spark

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 XML into RDF triples. SPARQL is an expressive query language used to retrieve data from RDF triple stores. Its federated query feature allows developers to write complex queries that can retrieve data from multiple sources. 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. The motivation for this work is to devise a unified approach to the problem of large-scale knowledge graph construction.

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) for processing CSV, JSON and XML. These extensions comprise additional functions and special SERVICE implementations with IRIs in the “norse:” namespace

KGCW'23: 4th International Workshop on Knowledge Graph Construction, May 28, 2023, Crete, GRE


✉ cstadler@informatik.uni-leipzig.de (C. Stadler); buehmann@informatik.uni-leipzig.de (L. Bühmann);

lpmeyer@infai.org (L. Meyer); martin@infai.org (M. Martin)

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



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

<https://w3id.org/aksw/norse#>. The implementations are part of our JenaX resource¹ which is available on Maven Central. It features several unofficial extensions for the Apache Jena framework. (3) We furthermore present an Apache Spark-based SPARQL engine that executes NORSE-enhanced SPARQL by leveraging its massive parallel processing model. We show that performance- and scalability-wise this approach surpasses the state of the art in several scenarios. It is worth noting that achieving these results is not the sole merit of Apache Spark, but also that of the optimizations we used.

The remainder of this paper is structured as follows: We present related work in Section 2. The translation of RML models to extended SPARQL CONSTRUCT queries is described in Section 3. Optimizations of query workloads with respect to the 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 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.

The implementation of our approach is part of our RDF Processing Toolkit (RPT)² which is based on JenaX.

2. Related Work

In this section we provide an overview of contemporary SPARQL and RML based knowledge graph construction approaches as well as a brief summary of Apache Spark. As there exist many mapping languages[1, 2, 3, 4], a discussion of general concepts and translations between them can be found in [5, 6].

2.1. SPARQL-based Mapping Approaches

SPARQL is a W3C standard for processing (loading, retrieving, transforming and updating) RDF data³. SPARQL engines can be leveraged to build advanced features on top. Two prominent representatives of the category of SPARQL-based mapping approaches are SPARQL-Generate[7] and SPARQL Anything[8]. SPARQL Anything extends SERVICE evaluation such that references to remote non-RDF data can be made. The data is converted to an opinionated RDF graph (as per documentation) which then serves as the base for evaluating the remainder of the SERVICE clause. SPARQL-Generate features a SPARQL-based template language which can produce output beyond what is possible with conventional SPARQL.

Our JenaX project compares to these approaches as follows: JenaX provides (among other things) SPARQL plugins for the Apache Jena ecosystem that allow for processing heterogeneous data within the SPARQL syntax already supported by the framework. As part of this effort we contributed a plugin system to facilitate interoperability of custom SERVICE execution implementations⁴. The SPARQL extensions for processing RML sources, as described in this

¹<https://github.com/Scaseco/jenax>

²<https://github.com/SmartDataAnalytics/RdfProcessingToolkit>

³<https://www.w3.org/TR/sparql11-query/>

⁴<https://github.com/apache/jena/pull/1388>

paper, are built on this system. While some of JenaX’s SPARQL extension functions for XML, JSON and CSV processing conceptually overlap with those provided by SPARQL-Generate, there are yet differences in the implementations. For example, JenaX provides dedicated RDF datatype implementations that internally retain JSON and XML data in an object model, whereas SPARQL-Generate (as of version 2.0.12) falls back to string representations. In order to avoid clashes we use IRIs in the *norse* namespace for our implementations.

2.2. RML Processors and Benchmarks

R2RML is a W3C standard and vocabulary for mapping relational data to RDF⁵. On the one hand these mappings can be used in ETL processes to dump databases as RDF. On the other hand, the same mappings can be used in SPARQL-to-SQL rewriting, a.k.a. OBDA (ontology-based data access). While R2RML is considered quite verbose, several alternatives have been developed, such as the Stardog Mapping Syntax (SMS, currently in version 2), the Ontop Mapping Language[9], and the Sparqlification Mapping Language[10].

*RML*⁶ is an extension of R2RML which adds additional vocabulary for mapping non-relational data[11, 12]. 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 are generated. Like R2RML it suffers from verbosity, for which reason simplified models were derived such as YARRRML[13]. A mapping translation between ShExML[14] and RML is presented by [15].

There exist several RML processors [16, 17]⁷ for the well known extension RML of the W3C standard R2RML. In this paper we are comparing benchmarks with the following: *SDM-RDFizer*⁸ is an RML processor implemented in python with optimized data structures and operators. It is developed with scalability and complex data in mind[18]. *CARML*⁹ and *RMLMapper*¹⁰ are Java implementations which operate single threaded at the time of writing. *Morph-KGC*¹¹ is an RML processor implemented in python and supports partitioning RML assertions[19] for parallel execution.

For measuring the performance of the RML processors we use the following benchmarks: The *Madrid GTFS benchmark*¹² was introduced by [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*¹³ was introduced by [18]. This benchmark is motivated from the biomedical domain and based on the Catalogue Of Somatic Mutations In Cancer¹⁴.

⁵<https://www.w3.org/TR/r2rml/>

⁶<https://rml.io/specs/rml/>

⁷<https://github.com/kg-construct/awesome-kgc-tools>

⁸<https://github.com/SDM-TIB/SDM-RDFizer>

⁹<https://github.com/carml/carml>

¹⁰<https://github.com/RMLio/rmlmapper-java>

¹¹<https://github.com/morph-kgc/morph-kgc>

¹²<https://github.com/oeg-upm/gtfs-bench>

¹³<https://figshare.com/articles/dataset/SDM-Genomic-Datasets/14838342/1>

¹⁴<https://cancer.sanger.ac.uk/cosmic>

2.3. Apache Spark And SANSA

Apache Spark¹⁵ is a framework for high parallelisation. It can scale workload execution from a single node to big clusters. Apache Spark advanced 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. Translating 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

A CONSTRUCT query has the form `CONSTRUCT { template } WHERE { pattern }`. Without loss of generality, for this work we assume generalized RDF¹⁶. 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 noted¹⁷ and several engines already support the production of quads as well. The approach presented in the following can be used in either setting, so instead of talking about a triple and quad (pattern) we generally speak of a *tuple* (pattern). A construct query's template is thus made up of a set of tuple patterns. Substituting all variables of these tuple patterns 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 XML¹⁸ and JSON datatypes as well as corresponding functions, namely `norse:json` and `norse:xml` to capture XML and JSON data efficiently, respectively.

¹⁵<https://spark.apache.org/>

¹⁶<https://www.w3.org/TR/rdf11-concepts/#section-generalized-rdf>

¹⁷<https://github.com/w3c/sparql-12/issues/31>

¹⁸Jena's implementation of the `rdf:xmlLiteral` datatype only stores XML as a string which is not suited for efficient XPath evaluation.

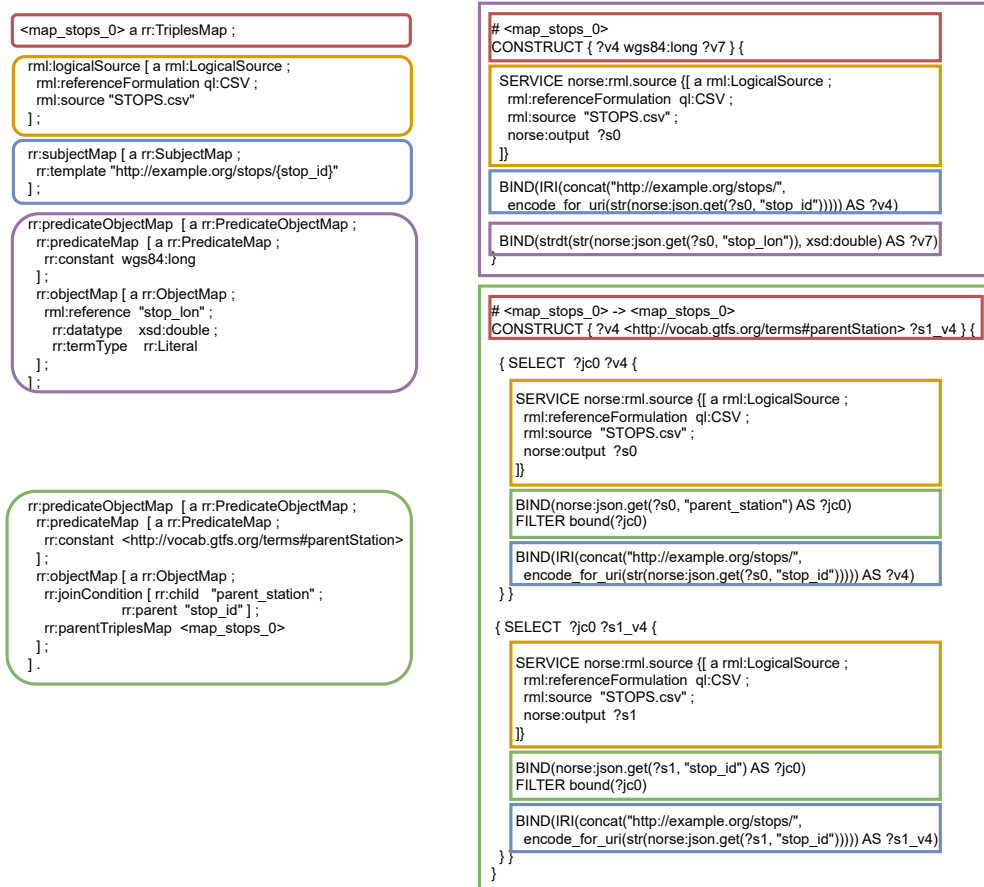


Figure 1: Juxtaposition of an RML document and its representation as SPARQL queries. The RML join condition is transformed into a natural join of SPARQL graph patterns where the same variable (?jc0) is bound on both sides.

3.3. Translating RML TermMaps

RML TermMaps 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, STRLANG and BNODE¹⁹ 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. 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.

¹⁹Unfortunately the standard BNODE function is not deterministic so SPARQL-based knowledge graph construction tools typically either alter the semantics or provide an alternative function.

- [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"), "en") AS ?v0)

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

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 treated as "unbound". For JSON arrays and objects an RDF term of type `norse:json` is returned.
- CSV: In our approach we represent CSV rows as JSON documents and thus access could be performed using the aforementioned `norse:json.path` function. If headers are absent then every row is represented as a JSON array, otherwise every row is turned into a JSON object whose keys are the CSV headers. However, in order to avoid the overhead of JSON path evaluation we also 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 an XPath evaluation is generally another XML node, such as `<lon>42.5</lon>`. The function `norse:xml.text` extracts an XML element's content as text, in this example `42.5`.

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 as the ones used in the GTFS-Madrid-Benchmark, it is known that a form of self-join elimination must be performed[18, 19]. An RML join condition can be generally omitted if the following conditions are met:

- The same logical source is used for the child and the parent 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 subject maps only mentions a subset of the references used in the join.

In such a case a referencing object map can be replaced with a simple object map based on the referenced TriplesMap's subject map. The underlying principle is sketched as follows. Let R be the TriplesMap's logical relation. Let C and P be the set of attributes referenced by the child and parent subject maps, respectively. Let J be the set of joining attributes. Then the following transformation can be applied if the condition $P \subseteq J$ or $C \subseteq J$ is met (c and p are SQL aliases):

SELECT DISTINCT $C \cup P$ FROM R_c JOIN R_p USING (J) \rightarrow SELECT DISTINCT $C \cup P$ FROM R

If the condition is met but DISTINCT is omitted then the JOIN can only introduce additional duplicates. Applying the transformation then reduces the duplicates to only those present in R .

4. Optimizing SPARQL CONSTRUCT Query Workloads

By transforming RML mappings into a set of SPARQL queries, the problem of efficient RML mapping execution becomes one of workload optimization of a set of SPARQL CONSTRUCT queries. The essential optimization goals are to efficiently produce tuples that are unique and/or ordered: For RDF data it is desirable to avoid duplicates as they needlessly increase size and processing time. Sorted RDF data eases inspection of the available information and assessment of fitness for use. It also enables efficient lookups using e.g. binary search. In the remainder of this section we detail our employed optimization procedure.

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 ORDER BY and/or DISTINCT can be used with CONSTRUCT queries, these solution modifiers only affect the underlying bindings and not the produced tuples. This is especially an issue when a CONSTRUCT query's template mentions multiple RDF tuple patterns. With SPARQL 1.1 there is no generic procedure to compute unique tuples in an efficient way that only has to evaluate the query pattern once.
- 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 engines²⁰. 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:

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

²⁰<https://github.com/w3c/sparql-12/issues/100>

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 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 graph²¹. Given a set of CONSTRUCT queries, a generic *merge* can be accomplished based on their lateral form as shown in Figure 5.

```

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.

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 introduce a model where the set(s) of possible RDF terms (produced by a tuple’s component) are represented as

²¹See discussion <https://github.com/w3c/sparql-12/issues/43>

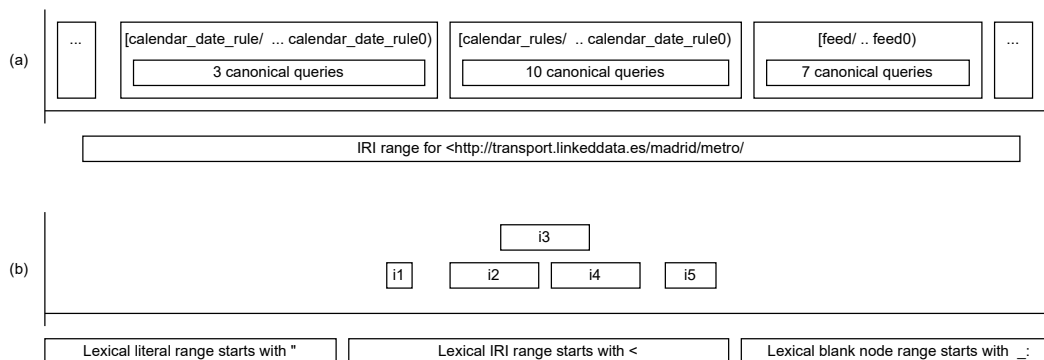


Figure 4: (a) An excerpt of the concrete range partitioning of canonical queries obtained from the GTFS-Madrid-Bench mapping based on their produced subjects. (b) An abstract model for RDF Term serialization in N-Quads with example intervals of which some (i3, i4, i5) overlap.

```

# Query a
CONSTRUCT {
  sa1 pa1 oa1
  ...
  san pan oan
} WHERE { PATTERNa }

# Query b
CONSTRUCT {
  sb1 pb1 ob1
  ...
  sbm pbm obm
} WHERE { PATTERNb }

CONSTRUCT { ?s ?p ?o }
WHERE {
  SELECT DISTINCT ?s ?p ?o {
    { PATTERNa
      LATERAL { { BIND(sai, pai, oai AS
        ?s, ?p, ?o) } UNION ... } } # for i in 1..n
    UNION
    { PATTERNb
      LATERAL { { BIND(sbj, pbj, obj AS
        ?s, ?p, ?o) } UNION ... } } # for j in 1..m
  } ORDER BY ?s ?p ?o
}

```

Figure 5: 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.

intervals. For brevity, we only focus on sorting RDF terms based on the lexical space of their N-Quads serialization.

For example, from an expression such as `BIND(IRI(CONCAT("gtfsbench/", ?id)) AS ?x)` we can derive that `?x` may be any of (1) unbound²² or (2) an IRI with a string value in the interval `["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 `(-∞ .. +∞)`. This way, we can "project" every CONSTRUCT query

²²This is the case if `?id` is not a string because `CONCAT` only allows for string arguments.

to an interval. Figure 4 (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 4 (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 5 for the sake of applying DISTINCT and ORDER BY. Extending this approach to SPARQL is possible but more complex because then the definition of intervals need to consider of the RDF term types and RDF literal datatypes.

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 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 syntax²³. Note, that 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.

- $\text{DISTINCT}(\text{EXTEND}(\text{var}, \text{constant}, \text{subOp})) \rightarrow \text{EXTEND}(\text{var}, \text{constant}, \text{DISTINCT}(\text{subOp}))$
- $\text{UNION}(\text{EXTEND}(\text{var}, \text{constant}, \text{left}), \text{EXTEND}(\text{var}, \text{constant}, \text{right})) \rightarrow$
 $\text{EXTEND}(\text{var}, \text{constant}, \text{UNION}(\text{left}, \text{right}))$
- $\text{EXTEND}(\text{var}, \text{non-constant-expr}, \text{EXTEND}(\text{var}, \text{constant}, \text{subOp})) \rightarrow$
 $\text{EXTEND}(\text{var}, \text{constant}, \text{EXTEND}(\text{var}, \text{non-constant-expr}, \text{subOp}))$

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* which bundles all components together – including the RML to SPARQL tooling – into a single command line toolkit.

NORSE SPARQL Extensions and RPT JenaX 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 JSON²⁴.

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 $[[\cdot]]$ that recursively translates SPARQL algebra operations

²³<https://www.w3.org/TR/sparql11-query/#sparqlAlgebra>

²⁴<https://mvnrepository.com/artifact/org.aksw.jenax/jenax-arq-plugins-bundle>

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 .

- $[[\text{SERVICE } \text{norse:rml.source } \{[\dots \text{norse:output } ?s]\}]] := \text{Create a RDD}\langle \text{Binding} \rangle$
where $?s$ is bound to records of the specified RML source.
- $[[\text{FILTER}(\text{subOp}, \text{expr})]] := [[\text{subOp}]].\text{filter}(\mu \rightarrow \text{exprEval}(\text{expr}, \mu) == \text{true})$
- $[[\text{JOIN}(\text{left}, \text{right})]] := [[\text{left}]].\text{mapToPair}(\mu_1 \rightarrow \langle \Pi_J(\mu_1), \mu_1 \rangle).\text{join}([[\text{right}]]).$
 $\text{mapToPair}(\mu_2 \rightarrow \langle \Pi_J(\mu_2), \mu_2 \rangle).\text{map}(\langle \text{key}, \langle \mu_1, \mu_2 \rangle \rangle \rightarrow \mu_1 \cup \mu_2)$
where J is the set of join variables $\text{vars}(\text{left}) \cap \text{vars}(\text{right})$ and $\Pi_J(\mu)$ is the projection of a binding to these variables.
- $[[\text{PROJECT}(\text{subOp}, \text{vars})]] := [[\text{subOp}]].\text{map}(\mu \rightarrow \Pi_{\text{vars}}(\mu))$
- $[[\text{DISTINCT}(\text{subOp})]] := [[\text{subOp}]].\text{distinct}()$
- $[[\text{LATERAL}(\text{left}, \text{right})]] := \text{if right has no basic graph patterns then}$
 $[[\text{left}]].\text{mapPartitions}(\mu \rightarrow \{ \mu \cup \nu \mid \forall \nu \in \text{convEval}(\text{subst}(\text{right}, \mu)) \})$
where convEval is conventional SPARQL evaluation into a (Java) collection of bindings rather than a Spark RDD.
- $[[\text{EXTEND}(\text{var}, \text{expr}, \text{subOp})]] := [[\text{subOp}]].\text{map}(\mu \rightarrow \mu \cup \{ \text{var} \rightarrow \text{exprEval}(\text{expr}, \mu) \})$

Figure 7: Evaluation of selected SPARQL operations with 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-Datasets²⁵. 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

²⁵The used files are 75percent_of_records_with_duplicate_and_each_duplicate_being_repeated_20times.csv and 4POM_Normal.ttl

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 feature²⁶. 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 it²⁷. 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.

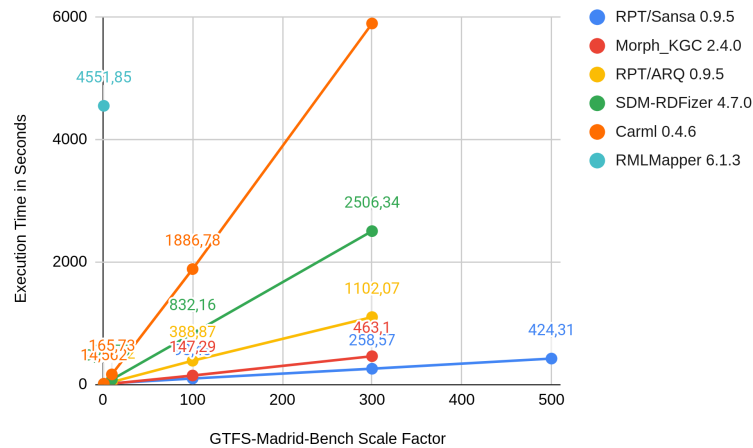


Figure 8: Comparison of RML mapping tool performance with 128G RAM. On scale factor 500, RPT/ARQ and Morph_KGC ran out of memory. Carml and SDM-RDFizer were already a magnitude slower on scale factor 300 and were not further evaluated. RMLmapper already exhibited a very high execution time on scale factor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

In a subsequent step, we evaluated the fastest approaches which are the ones that 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 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. On smaller sizes Morph_KGC outperforms RPT/SANSA. With increasing data scale the Apache Spark-based approach gains an advantage. However, on the workload that is mainly about duplicate removal the performance 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%. This means that the latter requires almost 10 times the CPU resources of the former in order to accomplish the same task. There are many aspects that can cause this significant

²⁶The only exception was Carml for which we appended a `sort -u` step

²⁷<https://github.com/semantifyit/RocketRML/issues/44>



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

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. This introduces a significant overhead when compared to e.g. simply keeping records in an in-memory hash set. Also, Jena introduces additional overhead by having to parse all RDF literals for expression evaluation. The benefit is, that invalid literals are reported whenever they are produced during knowledge graph construction. A performance issue which we detected and fixed during profiling was that Jena would needlessly materialize literals during SPARQL evaluation²⁸. Overall, further investigations are necessary to assess the impact of all relevant aspects on the performance in detail.

7. Conclusions and Future Work

In this work we showed that with the conversion of RML to SPARQL 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 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 system as well as minor

²⁸<https://github.com/apache/jena/pull/1802>

performance improvements to Apache Jena. One direction of future work is 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 the standardization of SPARQL for heterogeneous data as this would 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] 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>.
- [2] D. Chaves-Fraga, Knowledge Graph Construction from Heterogeneous Data Sources exploiting Declarative Mapping Rules, phdthesis, Universidad Politécnic de Madrid, 2021. doi:10.20868/UPM.thesis.67890.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] 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>.
- [7] 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, Springer, 2017, pp. 35–50. doi:10.1007/978-3-319-58068-5_3.
- [8] 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.
- [9] 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.

- [10] 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>.
- [11] 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.
- [12] A. Dimou, R2RML and RML comparison for RDF generation, their rules validation and inconsistency resolution, ArXiv (2020). doi:10.48550/ARXIV.2005.06293.
- [13] 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.
- [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] J. Lehmann, et al., 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. doi:10.1007/978-3-319-68204-4_15.
- [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.