Semantic Web 1 (0) 1–4
IOS Press

Empowering the SDM-RDFizer Tool for Scaling Up to Complex Knowledge Graph Creation Pipelines

Enrique Iglesias [a,*], Maria-Esther Vidal [a,c,b], Samaneh Jozashoori [b,c], Diego Collarana [d] and David Chaves-Fraga [e,f,g]

[a] *L3S Research Center Germany, Hannover Germany*
E-mail: iglesias@l3s.de
[b] *TIB Leibniz Information Centre for Science and Technology, Hannover Germany*
E-mails: maria.vidal@tib.eu, samaneh.jozashoori@tib.eu
[c] *Leibniz University, Hannover, Germany*
[d] *University of Bonn, Fraunhofer IAIS, Germany*
E-mail: diego.collarana.vargas@iais.fraunhofer.de
[e] *Ontology Engineering Group, Universidad Politécnica de Madrid, Spain*
E-mail: david.chaves@upm.es
[f] *Declarative Languages and Artificial Intelligence Group, KU Leuven, Belgium*
[g] *Flanders Make, DTAI-FET, Belgium*

Abstract. Data has grown exponentially in the last years, and knowledge graphs have gained momentum as data structures to integrate heterogeneous data and metadata. This explosion of data has created many opportunities to develop innovative technologies. Still, it brings attention to the lack of standardization for making data available, raising questions about interoperability and data quality. Data complexities such as large volume, heterogeneity, and high duplicate rates affect knowledge graph creation. This work addresses these issues to scale up knowledge graph creation guided by the RDF Mapping Language (RML). For that purpose, we present the SDM-RDFizer, a two-fold solution to address these two sources of complexity. First, RML triples maps are reordered in a way that the most selective maps are evaluated first, while non-selective rules are considered at the end, reducing the number of triples that are kept in the main memory. In the second step, an RDF compression strategy and novel operators are implemented to avoid the generation of duplicated RDF triples and the reduction of the number of comparisons during the execution of RML operators between mapping rules. We test our tool on two well-known benchmarks, overcoming state-of-the-art RML engines, and hence, demonstrating the benefits of the proposed techniques.

Keywords: Data Integration Systems , Knowledge Graphs, RDF Mapping Languages

1. Introduction

Knowledge graphs (KGs) have gained momentum in industrial and academic organizations to represent the convergence of data and metadata in a unified way [1]. KGs integrate heterogeneous data sources; they can be formally

*Corresponding author. E-mail: iglesias@l3s.de.

defined as data integration systems [2] composed of a unified schema, data sources, and mapping rules that state correspondences among the data sources and the unified schema. Declarative mapping languages, such as R2RML [3], RML [4], and SPARQL Generate [5], enable the definition of concepts in the unified schema (e.g., classes, properties, and attributes) in terms of data gathered from data sources in various formats (e.g., tabular, CSV, JSON, or RDF). These rules increase the modularity, maintenance, and reusability of KG creation pipelines.

The Semantic Web community has actively contributed to the problem of integrating heterogeneous datasets into KGs and developed various frameworks to execute declarative mapping rules [6–9]. Moreover, many engines are available such as Morph-KGC [10], RMLMapper[1], RocketRML [11], and SDM-RDFizer v3.2 [12]. These engines have implemented techniques to execute RML mapping rules efficiently. However, given the variety of parameters that can affect the performance of the KG creation process [13], existing engines may not scale up to KG creation pipelines defined in terms of complex mapping rules or large data sources. In this paper, we define data management techniques implemented in a new version of SDM-RDFizer to efficiently execute complex KG creation pipelines. Thus, this paper extends the work reported in Iglesias et al.[12] and makes the following novel contributions:

- Data structures for RDF data compression and planning the mapping rule execution.
- Physical operators able to exploit the data structures to efficiently scale up to complex RML mappings composed of multiple joins and large data sources with a high rate of duplicates.
- A new version of the SDM-RDFizer tool (named SDM-RDFizer v4.5.6) that implements these data structures with their associated physical operators and executes complex KG creation pipelines.
- An empirical evaluation of our new version on two state-of-the-art benchmarks, GTFS-Madrid-Bench [14] and SDM-Genomic-Datasets[2]. This study comprises 416 testbeds, and includes Morph-KGC [10], a recently published RML engine, RMLMapper, RocketRML, and the previous version of SDM-RDFizer (v3.2). The results of this study reveal the benefits of data management techniques presented in this paper and implemented in SDM-RDFizer v4.5.6. Specifically, the outcomes of this evaluation put in perspective the role of data structures and physical operators executing complex configurations such as the ones offered by the SDM-Genomic-Datasets.

The rest of the paper is structured as follows: Section 2 describes the main concepts and motivates our work. Section 3 summarizes previous approaches. Section 4 defines the data management methods implemented in SDM-RDFizer v4.5.6. Section 5 reports the results of our experimental studies. Section 6 describes the main characteristics of our tool and, finally, our conclusions and future work are outlined in Section 7.

2. Motivation and Requirements

2.1. Main RML Concepts

The RDF Mapping Language (RML) [15] allows specifying declarative mapping rules to transform into an RDF knowledge graph data collected from heterogeneous formats, e.g., JSON, CSV, and XML. Mapping rules are known as *Triples Map*s in the RML vocabulary[3]; Figure 1 depicts the specification of three triples maps (lines 1, 13, and 24). A *Triples Map* defines a group of mapping rules that define RDF triples with the same subject. For example, the triples map tagged <TriplesMap1> (lines 1 to 12) defines resources in the class `ex:Gene` and the RDF triples for these resources with the predicates, `ex:geneLabel` and `ex:gene_isRelatedTo_sample`. The data to populate the RDF triples are collected from the data sources defined with the term `rml:logicalSource` element (lines 2, 14, and 25). Resources of a class (e.g., ex:Gene, ex:Sample, and ex:Tumor) are specified with the term `rr:subjectMap`; the function `rr:template` allows defining the resources; e.g., respectively, lines 4, 16, and 27 define the URI of the resources of the classes ex:Gene, ex:Sample, and ex:Tumor. The term `rr:predicateObjectMap` defines the predicate and object values for the RDF subject defined by the

[1] https://github.com/RMLio/rmlmapper-java
[2] https://doi.org/10.6084/m9.figshare.17142371.v2
[3] https://rml.io/specs/rml/

```
1  <TriplesMap1>
2    rml:logicalSource [ rml:source "dataSource1" ];
3    rr:subjectMap [
4      rr:template "http://example.org/Gene/{Gene name}";
5      rr:class ex:Gene];
6    rr:predicateObjectMap [
7      rr:predicate ex:geneLabel;
8      rr:objectMap [ rml:reference "Gene name" ] ];
9    rr:predicateObjectMap [
10     rr:predicate ex:gene_isRelatedTo_sample;
11     rr:objectMap [
12       rr:parentTriplesMap <TriplesMap2> ] ].
```

```
13 <TriplesMap2>
14   rml:logicalSource [ rml:source "dataSource1" ];
15   rr:subjectMap [
16     rr:template "http://example.org/Sample/{ID_sample}";
17     rr:class ex:Sample] ;
18   rr:predicateObjectMap [
19     rr:predicate ex:sample_isTakenFrom_tumor;
20     rr:objectMap [
21       rr:parentTriplesMap <TriplesMap3>;
22       rr:joinCondition [ rr:child "ID_sample";
23                          rr:parent "ID_sample" ;];].
24 <TriplesMap3>
25   rml:logicalSource [ rml:source "dataSource2" ];
26   rr:subjectMap [
27     rr:template "http://example.org/Tumor/{ID_tumor}";
28     rr:class ex:Tumor ] .
```

Fig. 1. **Example.** Main concepts of the RDF Mapping Language (RML).

`rr:subjectMap`. For instance, in lines 6 and 8, the terms `rr:predicateObjectMap`, `rr:predicate`, and `rr:objectMap` define the instances of the RDF triples whose subject is defined by the `rr:subjectMap` in lines 3 and 5, the predicate is `ex:geneLabel` and the object value corresponds to the attribute `"Gene name"` from `"dataSource1"`. The object values of an RDF triple can be defined as the subject on a triples map defined over the same logical source. In this case, the term `rr:parentTriplesMap` is utilized to reference the triples map. For example, the values of the predicate `ex:gene_isRelatedTo_sample` correspond to the resources of the class `ex:Sample` specified by `TriplesMap2`. Finally, RML enables the data integration from different sources; the term `rr:joinCondition` specifies the attributes to be joined in the data sources of the triples maps to be merged. The term `rr:predicateObjectMap` in lines 18 and 23 defines the predicate `ex:sample_isTakenFrom_tumor` as the subject of `TriplesMap3` whose values for the attributes named `"ID_sample"` in `"dataSource1"` and `"dataSource2"` have the same values.

2.2. *Parameters Affecting the Performance of a Knowledge Graph Creation Pipeline*

According to Chaves-Fraga et al. [13], several parameters affect the performance of the KG creation pipeline. These parameters provide the basis for elucidating the requirements satisfied by the SDM-RDFizer. These parameters are grouped into five dimensions: mapping, data, platform, source, and output. The *data dimension* considers the characteristics of the data stored in a data source; it includes size, frequency distribution, data partitioning, and format. The impact of the hardware resources is represented in the *hardware dimension*, while the *source dimension* includes parameters such as data source transfer time and data initial delays. Finally, the *output dimension* groups parameters related to how the RDF triples are generated. They include duplicate removal, RDF triple generation at once, or in a streaming manner. Figure 2 illustrates the impact of some of these parameters on the performance of three state-of-the-art RML engines, i.e., RMLMapper v6.0[4], Morph-KGC v2.1.1[5], and SDM-RDFizer v3.2[6]. Figure 2a comprises six RML triples maps, five of them referencing to the same triples map, defining, thus, a star triples map. Therefore, this pipeline is composed of five child triples maps of the star (i.e., `TriplesMap2`, `TriplesMap3`, `TriplesMap4`, `TriplesMap5`, and `TriplesMap6`), and one parent triples map of the star (i.e., `TriplesMap1`). These triples maps are executed over the SDM-Genomic-Datasets [12]; they are of size 100k, 1M, and 5M rows; each dataset is presented with a different percentage of the data duplicate rate, i.e., 25% or 75%. As indicated by Chaves-Fraga et al. [13], these characteristics of mapping rules and data of the KG creation pipeline negatively impact the performance of the state-of-the-art engines, and making these engines to time out in five hours. The main limitation of these engines is caused by the lack of data structures and physical operators to efficiently execute complex joins among triples maps. In this paper, we present a new version of the SDM-RDFizer

[4] https://github.com/RMLio/rmlmapper-java/releases/tag/v6.0.0
[5] https://doi.org/10.5281/zenodo.6524684
[6] https://doi.org/10.5281/zenodo.3872104

```
<TriplesMap1>   Parent Triples Map of the Star
  rml:logicalSource [ rml:source "./source1.csv";
                      rml:referenceFormulation ql:CSV];
  rr:subjectMap [
    rr:template "http://project-iasis.eu/Mutation/{Accession Number}";
    rr:class iasis:Mutation;];
    rr:predicateObjectMap [
       rr:predicate rdf:type;
  rr:objectMap [rr:constant "http://project-iasis.eu/vocab/Mutation";];].

<TriplesMap2>
  rml:logicalSource [ rml:source "./source2.csv";
                      rml:referenceFormulation ql:CSV];
  rr:subjectMap [
    rr:template "http://project-iasis.eu/Gene/{Gene name}";
    rr:class iasis:Gene;];
  rr:predicateObjectMap [
    rr:predicate iasis:mutation_isRelatedTo_gene;
    rr:objectMap [
      rr:parentTriplesMap <TriplesMap1>;
      rr:joinCondition [                    Child Triples Map of the Star
        rr:child "HGNC ID";
        rr:parent "HGNC ID";];];].

<TriplesMap3>
  rml:logicalSource [ rml:source "./source3.csv";
                      rml:referenceFormulation ql:CSV];
  rr:subjectMap [
    rr:template "http://project-iasis.eu/Sample/{ID_sample}";
    rr:class iasis:Sample;];
  rr:predicateObjectMap [
    rr:predicate iasis:mutation_isRelatedTo_gene;
    rr:objectMap [
      rr:parentTriplesMap <TriplesMap1>;
      rr:joinCondition [                    Child Triples Map of the Star
        rr:child "HGNC ID";
        rr:parent "HGNC ID";];];].

<TriplesMap4>
  rml:logicalSource [ rml:source "./source4.csv";
                      rml:referenceFormulation ql:CSV];
  rr:subjectMap [
    rr:template "http://project-iasis.eu/GenomePosition/{Gene CDS length}";
    rr:class iasis:GenomePosition;];
  rr:predicateObjectMap [
    rr:predicate iasis:mutation_isObservedIn_GenomePosition;
    rr:objectMap [
      rr:parentTriplesMap <TriplesMap1>;
      rr:joinCondition [                    Child Triples Map of the Star
        rr:child "HGNC ID";
        rr:parent "HGNC ID";];];].

<TriplesMap5>
  rml:logicalSource [ rml:source "./source7.csv";
                      rml:referenceFormulation ql:CSV];
  rr:subjectMap [
    rr:template "http://project-iasis.eu/Tumor/{Primary histology}";
    rr:class iasis:Tumor;];
  rr:predicateObjectMap [
    rr:predicate iasis:mutation_isObservedIn_tumor;
    rr:objectMap [
      rr:parentTriplesMap <TriplesMap1>;
      rr:joinCondition [                    Child Triples Map of the Star
        rr:child "HGNC ID";
        rr:parent "HGNC ID";];];].

<TriplesMap6>
  rml:logicalSource [ rml:source "./source6.csv";
                      rml:referenceFormulation ql:CSV];
  rr:subjectMap [
    rr:template "http://project-iasis.eu/Tumor/{Mutation somatic status}";
    rr:class iasis:SomaticStatus;];
  rr:predicateObjectMap [
    rr:predicate iasis:mutation_isObservedIn_tumor;
    rr:objectMap [
      rr:parentTriplesMap <TriplesMap1>;
      rr:joinCondition [                    Child Triples Map of the Star
        rr:child "HGNC ID";
        rr:parent "HGNC ID";];];].
```

(a) A Star Triple Map

RML Engine	Time (secs)	
	Percentage of Duplicates: 25%	Percentage of Duplicates: 75%
Data Size: 100K		
RMLMapper	11,961.81	12,669.84
Morph-KGC	23.24	23.71
SDM-RDFizer v3.2	79.54	45.3
Data Size: 1M		
RMLMapper	Timeout	Timeout
Morph-KGC	2,411.1	2,013.16
SDM-RDFizer v3.2	1,323.61	756.24
Data Size: 5M		
RMLMapper	Timeout	Timeout
Morph-KGC	Timeout	Timeout
SDM-RDFizer v3.2	8,092.01	7,307.18

(b) Performance of State-of-the-art RML Engines. Timeout of five hours.

Fig. 2. **Motivating Example.** A Star Triples Map and its impact on the performance of a KG creation pipeline.

(a.k.a. v4.5.6[7]) which implements data management techniques to plan the execution of the triples map, and efficiently compressed intermediate results and merged them during the execution of a KG creation pipeline.

2.3. Requirements of a Knowledge Graph Creation Pipeline

Based on the parameters defined by Chaves-Fraga et al., we elucidate the requirements to be satisfied by an RML engine to efficiently execute KG creation pipelines. These requirements (REs) are defined as follows:

- **RE1-Execution of Complex Triples Maps**: An RML engine should be able to produce the RDF KG independently of the type of the triples maps and their complexity.

[7]https://doi.org/10.5281/zenodo.7027549

– **RE2-Scalable Integration of Large Data Sources**: An RML engine should be able to execute KG creation pipelines even in presence of large data sources.
– **RE3-Efficient Management of Duplicates**: An RML engine should detect duplicated data in a KG creation pipeline data sources and avoid the generation of duplicated RDF triples.
– **RE4-Performance Agnostic of the Data Source Format**: The performance of an RML engine should not be affected by the format of the data sources in the KG creation pipeline.

3. Related Work

The creation of KGs is defined as a data integration system [16], where input data sources are intended to be represented in a global view, usually defined as an ontology [17]. The relationship between these input sources and the ontology is generally declared using mapping rules such as the W3C recommendation R2RML [3] (for RDB) or its main extension for heterogeneous data formats (e.g., CSV, XML, and JSON), RML [18], although there are other solutions that adapt other languages (e.g., SPARQL) to perform the integration (e.g., SPARQL-Anything [19], SPARQL-Generate [20]). The integration process can be virtually (without actually transforming the input sources) or materialized (creating an integrated physical RDF dataset).

3.1. Virtual Data Integration in Knowledge Graphs

In a virtual knowledge graph creation process (formerly known as ontology-based data access), mapping rules are used to translate an input query that follows the target model to a semantic equivalent query (or set of queries) that can be executed directly on the input source(s) [21]. Virtual KG engines usually perform the translation of the input SPARQL query together with a set of optimizations over the translated query and, finally, the creation of the answer result-set according to the ontology terms. Most of the proposed solutions focus on translating SPARQL into SQL. The most well-known virtual KG engine is Ontop [7], which supports R2RML mappings and proposes many optimizations during the translation of the SPARQL query [22, 23]. Similarly, [9] presents Morph-RDB, an R2RML-compliant engine that translates and optimizes SPARQL queries into equivalent SQL queries. Morph-CSV [8] propose a system to efficiently apply domain-specific constraints on raw tabular data (i.e., CSV files) for enhancing SPARQL2SQL engines. More recently, virtual RML engines appear to allow the execution of SPARQL queries over heterogeneous data sources. Ontario [24] and Squerall [25] are two engines that translate and distribute SPARQL queries in federation scenarios using RML mappings.

3.2. Materialized Knowledge Graph Creation

Lately, the interest in the creation of materialized KGs has grown because they are the backbone of several artificial intelligence techniques [26], such as question answering systems, entity linking, data modeling, etc. The community has actively contributed with data management techniques and ETL tools that integrate large and heterogeneous data sources into KGs using declarative mapping rules. RMLMapper[8] is the reference implementation for an RML processor. It aims to be a complete-feature engine to ensure compliance with the RML specification, but is generally not able to handle complex knowledge graph creation pipelines [27]. The first version of our engine, SDM-RDFizer [12], presents a set of physical data structures and their corresponding operators to efficiently perform duplicate removal and join conditions, but stores these data structures in the main memory, which affects its scalability. Morph-KGC [10] processes R2RML, RML, and RML-star [28] and scales up the creation of KGs implementing the concept of mapping-partitions. This technique address some of the parameters that affect the construction of KGs (e.g., do not generate duplicates, parallelize the execution), but it is only focused on mapping planning, and the authors already report the need of new techniques or data structures for managing complex [R2]RML operators (i.e., join conditions). There are other RML-compliant engines such as CARML[9], Chimera [29] and Rock-

[8] https://github.com/RMLio/rmlmapper-java
[9] https://github.com/carml/carml

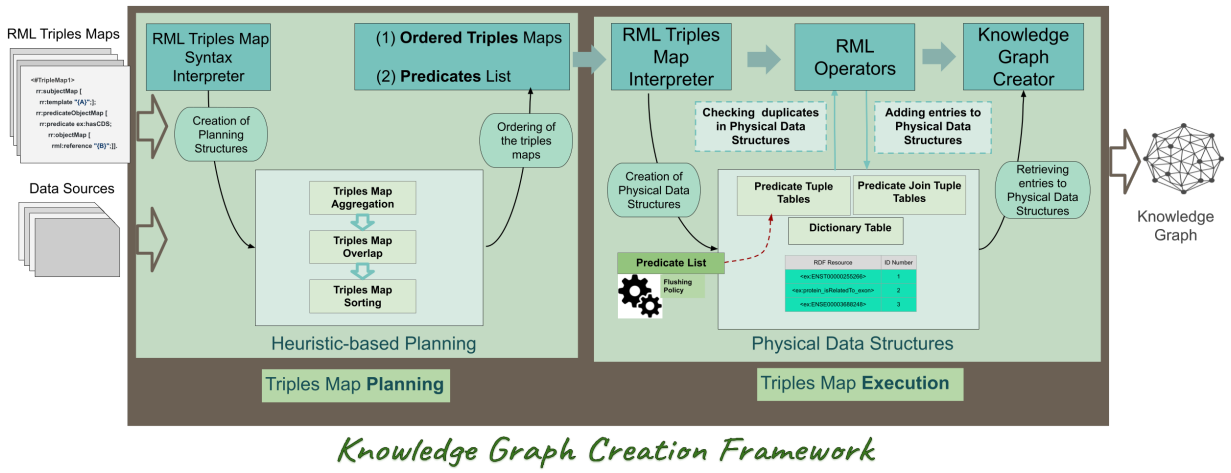


Fig. 3. **The SDM-RDFizer Architecture.** SDM-RDFizer receives as an input RML triples maps with its corresponding data sources. The Triples Map Planning component plans and orders the execution of the triples maps. The Triples Map Execution component resorts to physical operators, data structures, and compression techniques to efficiently execute the triples maps in the order stated during planning.

etRML [30] but to the best of our knowledge, they do not implement any technique to ensure high performance and scalability in the creation of KGs. Despite the enormous effort in the development of these solutions, there is still room to research and implement more scalable approaches that guarantee their adoption in industry and academia.

4. SDM-RDFizer: A tool for the Materialized Creation of Knowledge Graphs

This section describes the SDM-RDFizer tool. We present the SDM-RDFizer architecture and data management techniques that enable this engine to satisfy the requirements described in subsection 2.3.

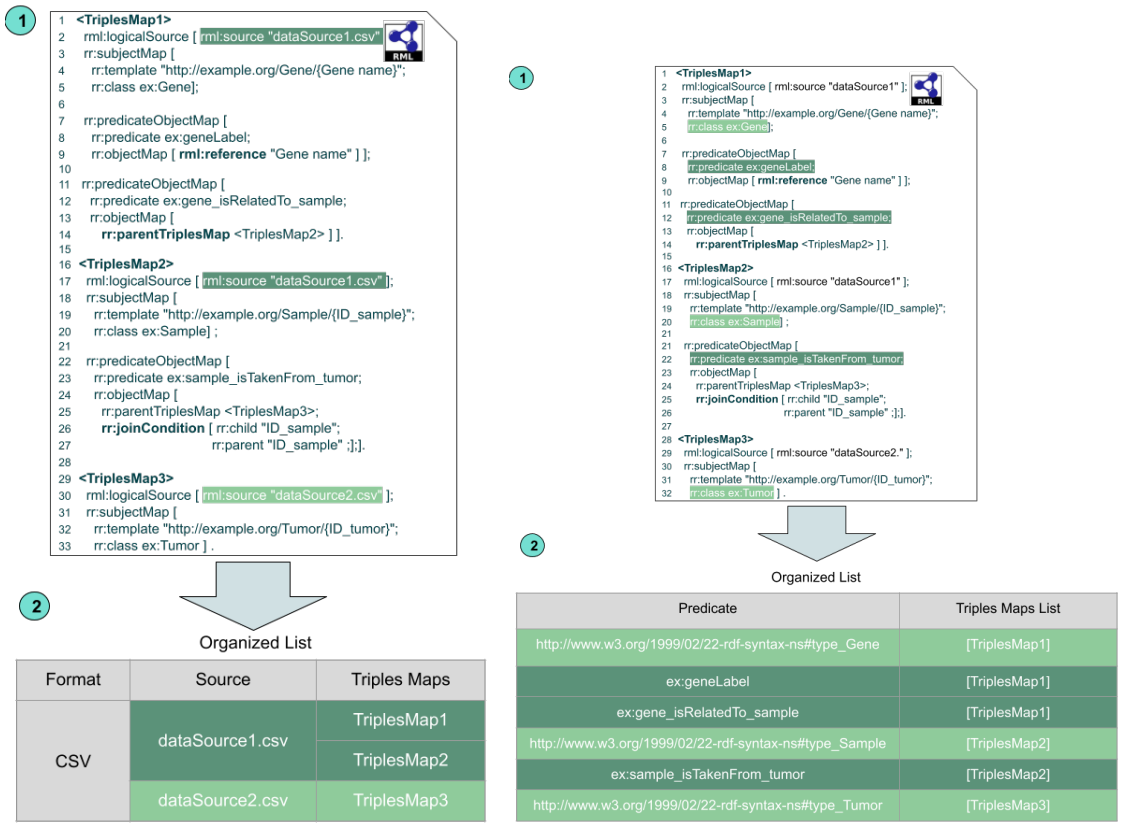
4.1. The SDM-RDFizer Architecture

SDM-RDFizer implements multiple data structures that optimize different aspects of the KG creation process such as duplicate removal, join execution, and data compression, and operators that execute various types of triples maps efficiently. Additionally, SDM-RDFizer is able to plan the execution of the RML triples maps to reduce execution time and secondary memory consumption.

Figure 3 depicts the SDM-RDFizer architecture in terms of its components. The SDM-RDFizer comprises two main modules: **Triples Map Planning** (TMP) and **Triples Map Execution** (TME). TMP determines the order of evaluation of the triples maps so that the amount of memory used is kept at a minimum. Next, TME evaluates the triples maps in the generated order. Each triples map is defined in terms of a physical RML operator (**Simple Object Map** (SOM), **Object Reference Map** (ORM), and **Object Join Map** (OJM)) so that RDF triples can be generated as the execution of these operators. Each generated triple is compared to the corresponding **Predicate Tuple Table** (PTT) to determine if the triple is a duplicate. If the triple is a duplicate, it is discarded. If the triple is not a duplicate, it is added to the corresponding PTT and the KG. The generated RDF resources and literal are represented in the **Dictionary Table** (DT). In case there is a join condition, the **Predicate Join Tuple Table** (PJTT) is used to store the results of the join condition.

4.2. The SDM-RDFizer Planning Phase

The Triples Map Planning (TMP) module reorders RML triples maps so that the most selective ones are evaluated first, while non-selective rules are executed at the end. TMP organizes the triples maps and data sources so that the number of RDF triples kept in the main memory is reduced. As a result, the KG creation process consumes the



(a) Organized Triples Map List. (b) Predicate List.

Fig. 4. **The Data Structures for the Planning Phase.** The Organized Triples Map List groups Triples Maps (TMs) first by data source and then by data source format, thus reducing the need to open any particular data source more than once. Predicate List groups the TMs by predicate. Each time a TM is finished processing, it is removed from the predicate list, and if a predicate becomes empty, the corresponding PTT is flushed. PL also defines the order of execution of the Organized Triples Map List by determining which TMs have overlapping predicates.

minimum amount of memory and execution time. TMP defines two data structures, the **Organized Triples Maps List** (OTML) and the **Predicate List** (PL); they encapsulate the order in which the TMs should be executed. OTML groups the TMs by data source, while PL groups the TMs by predicate and orders them. Triples maps are sorted to determine which of them define the same predicates.

Organized Triples Maps List (OTML) groups TMs by their data source; see Figure 4a. OTML is only used with TMs with file data sources (e.g., CSV, JSON, and XML), i.e., it is not utilized for relational databases. This is because, when processing relational databases, the RML engine should collect the data indicated in the TM using SQL queries.

During the Triples Map Planning phase, TMs are classified based on the logical data source format (i.e., CSV, JSON, and XML). Afterward, they are grouped by their data source; thus, a data source is opened once to execute all the TMs. Implementing this data structure into the SDM-RDFizer causes the processor to adopt a hybrid approach. A data-driven approach is used for TMs with file data sources, while a mapping-driven approach is used for TMs with relational databases. Figure 4a depicts the OTML for the TMs in Figure 1. Since all these TMs are over CSV files, only one group is created.

Predicate List (PL) groups TMs by their predicates by creating a list of TMs associated with a particular predicate; see Figure 4b. PL has two purposes, the first is to determine when a PTT associated with a certain predicate is ready to be flushed from main memory, and the second is to organize OTML. Each time a triples map is processed, it is removed from the list of the corresponding predicates. If the list becomes empty, no TMs require the correspond-

ing predicate, and the associated PTT is safe to be flushed. We add the range of the predicate to the predicate in the case of generic predicates, such as `rdf:type`. For example, as it can be seen in Figure 4b in line 20, there is the predicate `rdf:type` and its range is `ex:Sample`. The corresponding entry in the PL is `rdf:type_Sample`. PL is also utilized to organize OTML, determined by which TMs have overlapping predicates. By default, the order of execution considers the TMs with the least overlap first and the one with the most overlap last. Following each TM is the TM that overlaps with it so that when a TM is finished executing, the most memory is released, and if any PTT remains, they will be flushed when the following TM culminates.

4.3. The SDM-RDFizer Triples Map Execution Phase

4.3.1. The SDM-RDFizer Data Structures

SDM-RDFizer implements three data structures to efficiently manage and store the intermediate RDF triples generated during the execution of RML triples maps. These data structures avoid the generation of duplicated triples. Intermediate results are stored in these data structures independently of the format of the data sources. Thus, SDM-RDFizer exhibits a performance agnostic of the format of the data sources (requirement **RE4**).

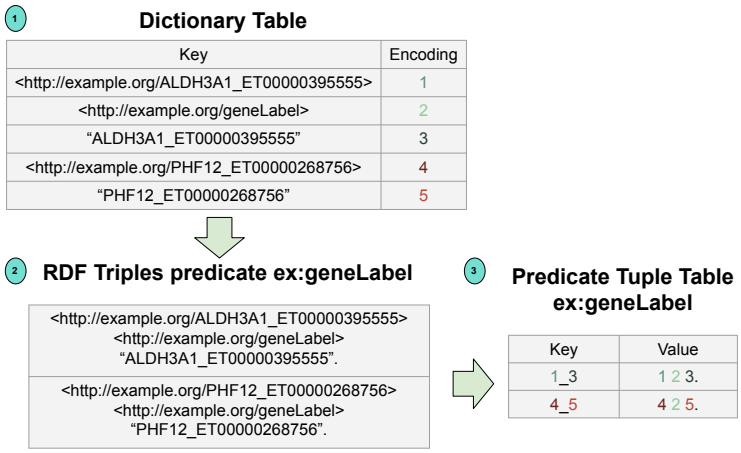


Fig. 5. **Dictionary Table and Predicate Triple Table.** Each generated RDF resource or literal is encoded in terms of an identification number in base 36. The Dictionary Table stores this encoding. RDF triples generated for a predicate, e.g., `ex:geneLabel`, are stored into its corresponding Predicate Tuple Table using the identification number of each RDF resource and literal.

The Dictionary Table (DT) encodes each RDF resource generated during the execution of a KG creation pipeline, with an identification number. It is implemented as a hash table where the key is the RDF resource and the value is the identification number in base 36. An encoding function, *encode(.)*, is implemented to transform each RDF resource into its corresponding identification number. Figure 5 illustrates an example of the DT that includes five entries with the identification numbers 1-5.

Predicate Tuple Table (PTT) is a table that stores all the RDF triples generated so far for a given predicate p. Figure 5 presents a PTT for the predicate `ex:geneLabel`. PTTs correspond to hash tables where the hash key of an entry corresponds to the encoding of the subject and object of a generated RDF triple, and the value of the entry is the encoding of the RDF triple. As shown in Figure 5, the identification number stored in the Dictionary Table for the corresponding RDF resources and literals, are used to encode the entries of a PTT. A PPT avoids the duplicate generation of an RDF triple, i.e., whenever an RDF triple is part of a PTT, this triple has been previously created, and the new triple should not be considered and added to the KG. However, when an RDF triple is not within PTT, then it is unique and must be added to PTT and the RDF knowledge graph. Whenever an RDF triple is generated during the execution of a TM, the corresponding PTT is checked. PTTs bring significant savings not only in sources with high-duplicated rates, but also when data sources create RDF triples of p also overlap. Thus, the Dictionary Table and Predicate Triple Tables empower SDM-RDFizer to satisfy the requirements **RE2** and **RE3**.

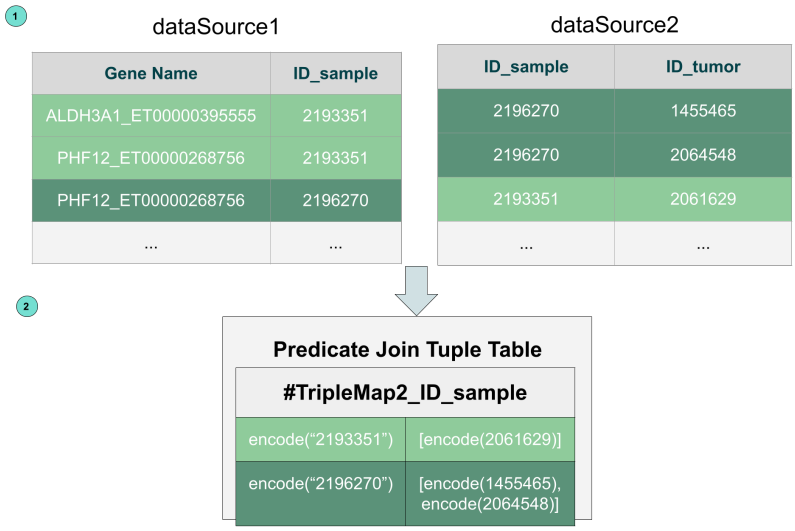


Fig. 6. **Predicate Join Triple Table.** The PJTT for the join between triples maps `TriplesMap2` and `TriplesMap3` in Figure 1 (lines 21-23). The encoding stored in the Dictionary Table is used to minimize the space to store intermediate results.

Predicate Join Triple Table (PJTT) is a table that stores the subjects of the triples generated by a join. It is implemented as an index hash table to the data source of the parent triples map in a join condition. A PJTT key corresponds to the encoding of each value(s) of the attributes in the join condition. The value of a key in a PJTT corresponds to the encoding of the subject values in the data source of the parent triples maps, which are associated with encoding the values of the attributes in the hash key. The Dictionary Table is used to retrieve the encodings of the RDF resources or literals. Thus, a PJTT minimizes the space to store intermediate results. Figure 6 illustrates the PJTT for the join condition between triples maps `TriplesMap2` and `TriplesMap3` in Figure 1 (lines 21-23). This PJTT corresponds to an index hash from the encoding values of the attribute `ID_sample` to the values of the attribute `ID_tumor` in `dataSource2`. The set of encoded values enables to directly access all the values of `ID_tumor` that join with a value of `ID_sample`. Thus, a PJTT enables direct access to the subjects associated with a join condition, and implements an index join which corresponds to the most efficient implementation of a join [31]. As a result, the Predicate Join Triple Tables provide the basis for the implementation of the physical operators which allow SDM-RDFizer to satisfy the requirements **RE1** and **RE2**.

4.3.2. The SDM-RDFizer Physical Operators

SDM-RDFizer resorts to three physical operators to efficiently execute triples maps in a KG creation pipeline.

Simple Object Map (SOM) generates an RDF triple by performing a simple predicate object map statement; Algorithm 1 sketches the SOM implementation. Given a TM and its respective data source, SOM generates the RDF triples. The encoding of each generated RDF triple is checked against the corresponding predicate tuple table (**PTT**). If the RDF triple already exists in PTT, it is discarded. If not, it is added both to PTT and to the KG.

Object Reference Map (ORM) implements the object reference between two triples maps defined over the same data source. It extends SOM by using the subject of the parent triples map as the object of another TM. Thus, the same process as in SOM is applied to the generated RDF triples, i.e., the encoding of the triples are checked against PTT to determine if they will be added to the KG. Algorithm 2 illustrates the ORM implementation.

Object Join Map (OJM) implements an index join in executing a join condition between two TMs defined over two different data sources. OJM resorts to the corresponding PJTT to access the encoded values in the child map associated with the encoded values of the data source of the parent triples map. Thus, if the encoded value $encode(e)$ of a value in the data source of the child triples map exists in PJTT, then the set of values for $encode(e)$ is used to generate the resulting RDF triples. Finally, similar to the last two operations, the generated RDF triples are checked against the corresponding PTT to avoid duplicate generation. Algorithm 3 sketches the OJM implementation for a join between triples maps $tm1$ and $tm2$.

Algorithm 1 Simple Object Map (SOM)

Input: Triples Map $tm1$ defines predicate p on logical source S and $tm1$ `subjectMap` is $f1(att1)$ and $tm1$ `objectMap` for p is $f2(att2)$
Output: RDF Triples for p generated from $tm1$
for each: row $\in S$
1: Create RDF triple $t = (f1(row.att1), p, f2(row.att1))$
2: Add $f1(row.att1)$ and $f2(row.att1))$ to the Dictionary Table
3: **if** $encode(f1(row.att1), f2(row.att1))$ does not belong to the PPT for p **then**
4: Add $encode(f1(row.att1), f2(row.att1))$ to PPT for p
5: Add $(f1(row.att1), p, f2(row.att1))$ to the KG
6: **end if**
7: **return** KG

Algorithm 2 Object Reference Map (ORM)

Input: Triples Map $tm1$ refers to Triples Map $tm2$ to define predicate p on logical source S and $tm1$ `subjectMap` is $f1(att1)$ and $tm2$ `subjectMap` is $f2(att2)$
Output: RDF Triples for p generated from $tm1$
for each: row $\in S$
1: Create RDF triple $t = (f1(row.att1), p, f2(row.att1))$
2: Add $f1(row.att1)$ and $f2(row.att1))$ to the Dictionary Table
3: **if** $encode(f1(row.att1), f2(row.att1))$ does not belong to the PPT for p **then**
4: Add $encode(f1(row.att1), f2(row.att1))$ to PPT for p
5: Add $(f1(row.att1), p, f2(row.att1))$ to the KG
6: **end if**
7: **return** KG

Algorithm 3 Object Join Map (OJM)

Input: Triples Map $tm1$ refers to Triples Map $tm2$ to define predicate p on logical sources $S1$ and $S2$ and join condition B on attributes $S1Att$ and $S2Att$, respectively, and $tm1$ `subjectMap` is $f1(att1)$ and $tm2$ `subjectMap` is $f2(att2)$
Output: RDF Triples for p generated from $tm1$
1: **for** row1 $\in S1$ **do**
2: **if** $encode(row1.S1Att)$ belongs to the PJTT for $tm2$ **then**
3: **for** v $\in$ valueSet($encode(row1.S1Att)$) in PJTT for $tm2$ **do**
4: Create $t = (f1(row1.att1), p, f2(decode(v)))$
5: Add $f1(row1.att1)$ and $f2(decode(v))$ to the Dictionary Table
6: **if** $encode(f1(row1.att1), f2(decode(v)))$ does not belong to the PPT for p **then**
7: Add $encode(f1(row1.att1), f2(decode(v))$ to PPT for p
8: Add $(f1(row1.att1), p, f2(decode(v)))$ to the KG
9: **end if**
10: **end for**
11: **end if**
12: **end for**
13: **return** KG

5. Empirical Evaluation

We evaluate SDM-RDFizer with the goal of answering the following research questions: **RQ1)** *What is the impact of the data duplicate rates in the execution time of a knowledge graph creation approach?* **RQ2)** *What is the impact of the input data size in the execution time of a knowledge graph creation approach?* **RQ3)** *How the types of a triples maps affect the existing engines?* The experimental settings are as follows:

5.1. Experimental Settings

Benchmarks

Experiments are performed over datasets from **GTFS-Madrid-Bench** and **SDM-Genomic-Datasets**. Therefore, our experiments cover a large range of parameters that affect the KG creation task, i.e., dataset size, TM type and complexity, selectivity of the results, and types of join between the TMs. In total, we consider three different mapping types: Simple Object Map (SOM), Object Reference Map (ORM), and Object Join Map (OJM). All resources and experimental settings used in this evaluation are publicly available[10] and Table 1 summarizes the used configurations.

GTFS-Madrid-Bench [14]: This benchmark enables the generation of different configurations that affect the characteristics of creating a KG. We generate four logical sources with the scaling factor 1-csv, 5-csv, 10-csv, and 50-csv. The scale value influences the size of the resulting KG. For example, the KG generated from 5-csv is five times bigger than the KG generated from 1-csv. We consider mapping rules comprised of 13 TMs with 73 SOMs, and 12 OJMs involving ten data sources.

SDM-Genomic-Datasets [12]: This benchmark is created by randomly sampling data records from somatic mutation data collected in COSMIC[11]. SDM-Genomic-Datasets include eight different logical sources of various sizes, including 10k, 100k, 1M, and 5M rows. For every pair of sources of the same size, they differ in the percentage of the data duplicate rate, which can be either 25% or 75%, where each duplicate value is repeated 20 times. For example, a 10k logical source with 75% data duplicate rate has 25% duplicate-free records (i.e., 2500 rows) and the rest of the 75% records (i.e., 7500 rows) correspond to 375 different records which are duplicated 20 times; in total there are 2,875 unique values. The SDM-Genomic-Datasets offers ten different configurations. **Conf1:** A TM containing one SOM. **Conf2:** A TM containing four SOMs. **Conf3:** Set of two TMs with one ORM. **Conf4:** Set of five TMs with four ORMs. **Conf5:** Set of two TMs with one OJM. **Conf6:** Set of five TMs with four OJMs. We group the aforementioned TM configuration into a set named **AllTogether**. Furthermore, the benchmark includes three additional configurations to evaluate the impact of two other influential parameters on the performance of KG creation frameworks[12]. **Conf7** aims at evaluating the impact of defining the same predicates using different TMs. It has a set of four TMs with two OJMs. For each pair of TMs, there is an OJM. The data sources of one pair of the TMs are a subset of the other pair. Both pairs of TMs share the same predicate. **Conf8** provides a TM that is connected to five other TMs with different logical sources through join, i.e., this TM is connected via a five-star join with the other TMs. It comprises a set of six TMs with five OJMs; five child TMs refer to the same parent TM. **Conf9** combines the first two configurations in one testbed; it is composed of a set of ten TMs with seven OJMs.

State-of-the-art RML Engines and Metrics.

The following RML tools are included in the empirical study. RMLMapper v6.0[13], RocketRML v1.11.3 [30][14], Morph-KGC v2.1.1 [10][15], and SDM-RDFizer v4.5.6[16]; it implements all the techniques described in this paper, i.e., planning techniques, physical operators, and data compression techniques to reduce the size of the main memory structures required to store intermediate results that were generated. The SDM-RDFizer v3.2[17] is also used to determine if there is an increase in performance between the older and newer version.

The performance of the RML engines is evaluated in terms of the following metrics. *Execution time:* Elapsed time spent to create a KG. The execution time is measured using the Python library `time`. The experiments are executed five times and the average is reported. The timeout is five hours. *Maximum memory usage:* The most memory used to generate a KG. The memory usage is measured using the Python library `malloc`. The `malloc` library measures the memory usage in Kilobytes; we convert the result into Megabytes. The experiments are executed in an

[10] https://github.com/SDM-TIB/SDM-RDFizer-Experiments/tree/master/swj2022
[11] https://cancer.sanger.ac.uk/cosmicGRCh37,version90,releasedAugust2019
[12] https://doi.org/10.6084/m9.figshare.17142371
[13] https://github.com/RMLio/rmlmapper-java
[14] https://github.com/semantifyit/RocketRML/
[15] https://github.com/oeg-upm/Morph-KGC
[16] https://github.com/SDM-TIB/SDM-RDFizer
[17] https://doi.org/10.5281/zenodo.3872104

Parameter: Dataset Size		
Benchmark	Size	Description
GTFS-Madrid-Bench	1-CSV	Ten different data sources are 4.8 Mb in total.
	5-CSV	Ten different data sources are 10 Mb in total. The generated KG is five times bigger than the KG generated from 1-CSV.
	10-CSV	Ten different data sources are 21 Mb in total. The generated KG is ten times bigger than the KG generated from 1-CSV.
	50-CSV	Ten different data sources are 102 Mb in total. The generated KG is fifty times bigger than the KG generated from 1-CSV.
SDM-Genomic-Datasets	10k	Each data source has 10,000 rows.
	100k	Each data source has 100,000 rows.
	1M	Each data source has 1,000,000 rows.
	5M	Each data source has 5,000,000 rows.
Parameters: Mapping Assertion (MA) Type and Complexity, Selectivity of the Results, and Type of Joins		
Benchmark	Mapping Configuration	Description
GTFS-Madrid-Bench	Standard Config	13 TMs with 73 SOMs, and 12 OJMs.
SDM-Genomic-Datasets	Conf1	A TM containing one SOM.
	Conf2	A TM containing four SOMs.
	Conf3	Set of two TMs, with one ORM.
	Conf4	Set of five TMs with four ORMs.
	Conf5	Set of two TMs, with one OJM.
	Conf6	Set of five TMs, with four OJMs.
	AllTogether	Combines Conf1- Conf6.
	Conf7	Set of four TMs with two OJMs. Evaluates the impact of defining the same predicates using different TMs.
	Conf8	Set of six TMs with five OJMs. Recreates a five-star join where five MAs refer to the same parent MA.
	Conf9	Set of ten TMs with seven OJMs. Combines Conf7+Conf8

Table 1

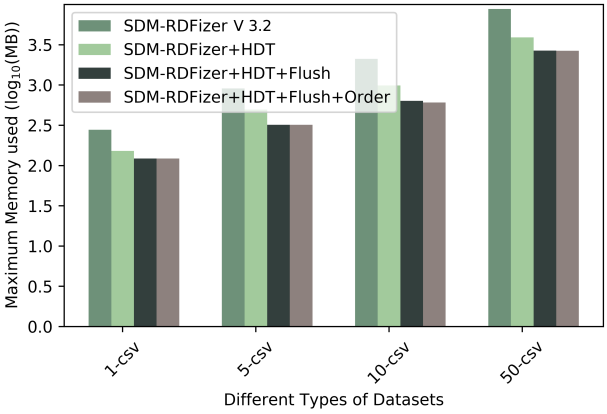
Datasets and Configurations of Triples Maps. The table describes each data source and configuration of TMs used in the experiments and their corresponding benchmarks. Configuration of TMs in bold are considered complex cases. They include several types of TMs of various complexity and complex joins (e.g., five-start joins).

Intel(R) Xeon(R) equipped with a CPU E5-2603 v3 @ 1.60GHz 20 cores, 64GB memory and with the O.S. Ubuntu 16.04LTS.

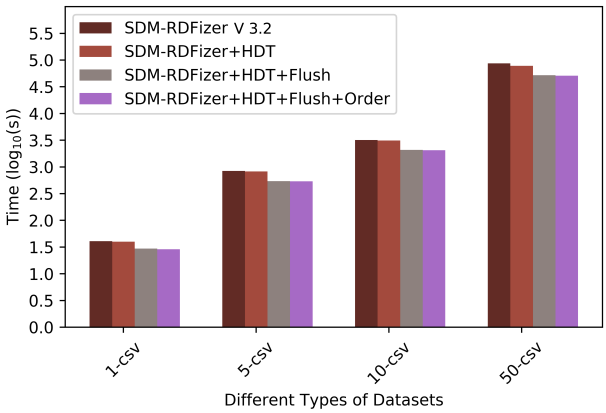
5.2. *Performance of the RML Engines in the GTFS-Madrid-Bench*

This experiment aims to illustrate the performance increase regarding execution time and memory consumption that the proposed data structures and operators will bring when implementing them into a KG creation engine. We evaluate the performance of different versions of the SDM-RDFizer. The previous version of the SDM-RDFizer (i.e., SDM-RDFizer v3.2) only contains the operators for TM transformation and the data structures for duplicate removal and join execution. In contrast, the much more complete version of the SDM-RDFizer (i.e., SDM-RDFizer v4.5.6, a.k.a. SDM-RDFizer+HDT+Flush+Order) contains all the proposed data structures and operators. We also include other combination of the proposed techniques: one only applies data compression (i.e., SDM-RDFizer+HDT), and the other has data compression and main memory flushing but no ordering (i.e., SDM-RDFizer+HDT+Flush).

It can be seen in Figure 7b that SDM-RDFizer v3.2 and SDM-RDFizer+HDT do not have much difference in execution time; this is because compressing data requires time to be executed; thus, any savings that result from this step can only be appreciated in terms of memory consumption (Figure 7a). On the other hand, SDM-



(a) Maximum Memory used.



(b) Execution time.

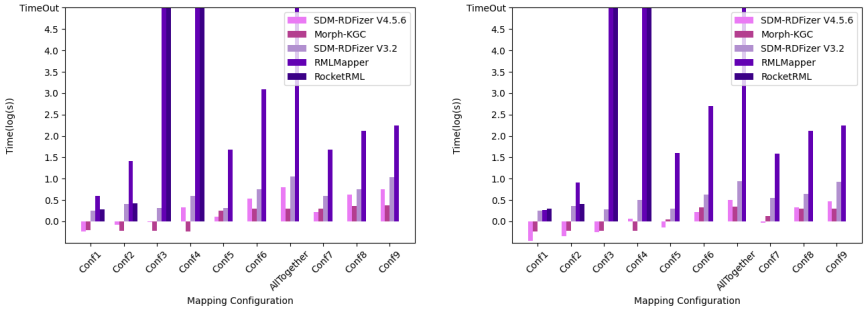
Fig. 7. **Results of the execution of the GTFS-Madrid-Bench benchmark.** Execution time and memory consumption of various version of the SDM-RDFizer when transforming the GTFS-Madrid-Bench benchmark.

RDFizer+HDT+Flush and SDM-RDFizer+HDT+Flush+Order reduce execution time compared to the two previous versions of the SDM-RDFizer. This reduction in execution time is thanks to the fact that unneeded data is flushed from main memory, thus making the duplicate removal process faster. Unfortunately, there are few savings between these last two configurations of the SDM-RDFizer. This mapping rules contain 13 TMs, thus making the organization process take longer and negatively impacting the execution time.

The Figure 7a illustrates the maximum memory consumption of the different versions of the SDM-RDFizer used. It can be seen in Figure 7a that the SDM-RDFizer v3.2 is the one that consumes the most memory. By applying data compression, there is a significant reduction in memory consumption thanks to the fact that the data stored in PTT for duplicate removal is much smaller than the data stored in the SDM-RDFizer v3.2. The flushing of unneeded data reduces the maximum memory used even further, but not as much as with data compression. Finally, SDM-RDFizer+HDT+Flush and SDM-RDFizer+HDT+Flush+Order have the same maximum memory consumption, since the only difference between them is the order in which the TMs are executed. The benefit of executing the TMs in a predetermined order is that the maximum amount of data is flushed after finishing the execution of a TM, thus minimizing the amount of memory being used.

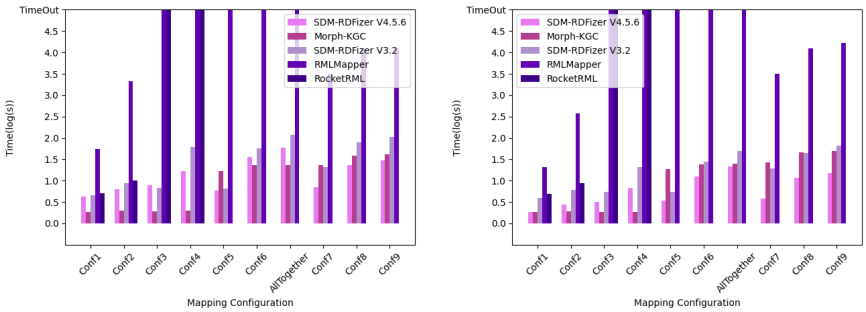
5.3. Performance of the RML Engines in the SDM-Genomic-Datasets

This experiment seeks to prove the impact of using real-world data for KG creation. Even though the TMs defined for SDM-Genomic-Datasets are simpler than those defined for GTFS-Madrid-Bench, they cover all the TM types defined in Figure 1. We evaluate the performance of each engine, i.e., SDM-RDFizer v3.2 and v4.5.6 (i.e., SDM-RDFizer+HDT+Flush+Order), Morph-KGC, RMLMapper, and RocketRML, by measuring the overall execution time it took the engine to complete the KG creation process. As it can be seen in Figure 8, in case of having ORMs (i.e., **Conf3**, **Conf4**, and **AllTogether**), the engines RMLMapper and RocketRML are not capable of completing these configurations before timing out (i.e., five hours). In addition, RocketRML is not capable of executing N-M joins, thus the corresponding test cases were not executed (i.e., **Conf5**, **Conf6**, **AllTogether**, **Conf7**, **Conf8**, and **Conf9**). Regarding the simpler cases (i.e., **Conf1** and **Conf2**), Morph-KGC presents the best performance, since they partition each data source into chunks and then apply the TM to multiple rows simultaneously. In the cases with ORMs (i.e., **Conf3**, **Conf4**, and **AllTogether**), Morph-KGC also presents the best performance; this is because they apply a transformation that turns all ORMs into equivalent SOMs. For all the engines, there is a reduction in execution time when transforming cases with high duplicates compared to the execution time of cases with lower duplicate rates. Mapping configurations transformed with larger data sources have longer execution times regardless of the engine. We compared the performance of the SDM-RDFizer v3.2 and SDM-RDFizer v4.5.6, and Figure 8 illustrates that SDM-RDFizer v4.5.6 has a reduction in execution time, especially in the configurations with OJMs.



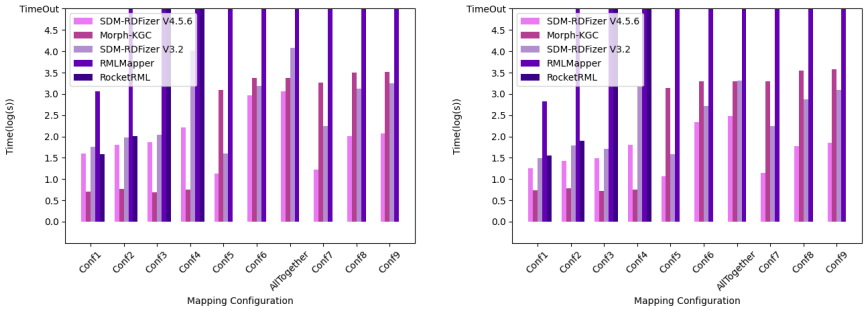
(a) 10k records with 25% duplicate rate.

(b) 10k records with 75% duplicate rate.



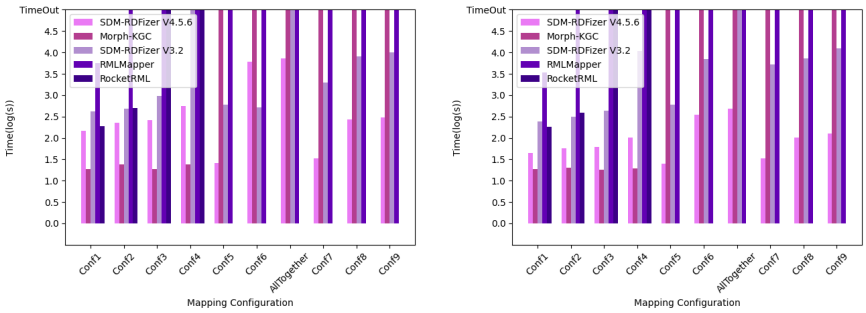
(c) 100k records with 25% duplicate rate.

(d) 100k records with 75% duplicate rate.



(e) 1M records with 25% duplicate rate.

(f) 1M records with 75% duplicate rate.



(g) 5M records with 25% duplicate rate.

(h) 5M records with 75% duplicate rate.

Fig. 8. **Results of the execution of the SDM-Genomic-Datasets benchmark.** Execution time of Conf1, Conf2, Conf3, Conf4, Conf5, Conf6, AllTogether, Conf7, Conf8, and Conf9 for SDM-RDFizer, Morph-KGC, RMLMapper, and RocketRML.

We can attribute this reduction in execution time to the new data structures (i.e., DT, OTML, and PL) introduced in this work, which help to reduce memory consumption, and, by extension, execution time. Additionally, Figure 8 shows that SDM-RDFizer v4.5.6 is the only engine capable of executing all the test cases. For most of the cases containing OJMs except the cases with data sources with 10k rows (i.e., Figure 8a and Figure 8b), the SDM-RDFizer v4.5.6 presents the best performance among the tested engines. Note that SDM-RDFizer v3.2 can also execute all the TMs; however, SDM-RDFizer v4.5.6 is more efficient. Thanks to the PJTT data structure, the TMs are executed as efficiently as possible. PJTT performs the join between the parent and child data source and stores the results in main memory, thus avoiding uploading the parent data source multiple times. The SDM-RDFizer v3.2 is also capable of executing these TMs without timing out but in a much less efficient manner; the reason for this is that the amount of memory consumed is much greater because it lacks techniques that the newer version has. In the case of Morph-KGC, this engine depends on the Python library `Pandas` for the execution of joins, which has a decent execution time for small data sources (i.e., Figure 8a, Figure 8b, Figure 8c, and Figure 8d). Still, when dealing with larger data sources (i.e., Figure 8e, Figure 8f, Figure 8g, and Figure 8h), there is an increase in the execution time. Finally, RMLMapper lacks operators capable of executing OJMs efficiently; thus, it executes a Cartesian product between the two data sources.

6. The SDM-RDFizer Characteristics

The SDM-RDFizer engine presents a distinctive set of characteristics that make it a valuable contribution for users and practitioners who create KGs for their own projects and use cases.

Novelty: SDM-RDFizer is the first RML engine that implements a hybrid approach to ensure high performance and scalability in complex data integration scenarios. On the one hand, the heuristic-based mapping planning based on the input sources and the list of predicates ensures efficient use of the main memory (requirements **RE2** and **RE4**). On the other hand, the encoding approach and the physical data structures with their corresponding operators guarantee the fulfillment of the requirements **RE1** and **RE3**. To the best of our knowledge, this is the first KG creation engine based on RML that implements mapping planning and physical data structures, demonstrating its efficiency over several testbeds on well-known benchmarks.

Availability: SDM-RDFizer is available for (re)use in multiple ways. The source code is accessible through the GitHub repository[18] under an Apache 2.0 license, so it can be extended and modified by any developer. The GitHub repository is also linked to the Zenodo platform, which provides a Digital Object Identifier (DOI) for the general repository[19] and also a DOI for each specific software release[20]. The engine is also available on the Python Package Index (PyPI), so it can be easily installed and integrated in other developments[21]. Finally, we also provide a docker image that deploys the SDM-RDFizer as a web service.

Utility: As we demonstrated in our experimental evaluation, the SDM-RDFizer is one of the best RML engines in terms of performance and scalability. Thanks to the implementation of heuristic-based planning and physical data structures, SDM-RDFizer scales up the construction of KGs, overcoming other state-of-the-art solutions. Thanks to the different configurations and optimizations implemented in our engine, it can be applied to different use cases, efficiently handling the parameters that affect the creation of KGs and providing a useful tool for different and heterogeneous use cases. In addition, SDM-RDFizer passed all proposed RML test-cases [32][22], which means that our engine is fully compliant with the RML specification.

Impact: Since its first release, the SDM-RDFizer has caught the attention of practitioners and knowledge engineers due to its good results w.r.t. other RML engines. The amount of commits in the GitHub repository[23] and the number of releases reflect the continuous improvements and support we give to our tool. At the time of writing, more than

[18] https://github.com/SDM-TIB/SDM-RDFizer
[19] The DOI for the SDM-RDFizer repository is: https://doi.org/10.5281/zenodo.3872103
[20] For example, the DOI for the v4.5.6 used in the experiments of this paper is: https://doi.org/10.5281/zenodo.7027549
[21] https://pypi.org/project/rdfizer/
[22] https://rml.io/implementation-report
[23] https://github.com/SDM-TIB/SDM-RDFizer

20 users have forked the source code to reuse or extend it with additional features, and the repository has almost 70 stars. With the implementation of the novel techniques presented in this paper, we expect that the SDM-RDFizer will become a reference implementation for RML and also convince industry partners to adopt declarative data integration solutions that ensure the maintenance of their KG creation pipelines.

Adoption: The SDM-RDFizer is used in multiple industrial and research projects to create KGs from the integration of heterogeneous data sources. These projects are summarized as follows: **a)** iASiS [24], EU H2020 funded project to exploit patient data insights towards precision medicine. The iASiS RDF knowledge graph comprises more than 1.2B RDF triples collected from more than 40 heterogeneous sources using over 1,300 RML TMs [33]. **b)** Lung Cancer Pilot of BigMedilytics [25], where the KG is defined in terms of 800 RML TMs from around 25 data sources; it comprises around 500M RDF triples. **c)** In CLARIFY [26], the KG integrates data from lung and breast cancer patients. It comprises 76M RDF triples and 16M RDF resources. 626 RML TMs define the CLARIFY KG. **d)** P4-LUCAT [27] has 676 RML TMs that define the P4-LUCAT KG in terms of a unified schema of 318 attributes and 177 classes; it comprises 178M of RDF triples. **e)** The ImProVIT KG [28] integrates immune system data into a unified schema of 176 classes, 66 predicates, and 151 attributes. **f)** PLATOON[29] project creates the KG for a pilot [34] that is defined in terms of 2,093 RML mapping rules; it comprises 220M RDF triples and 80M of RDF resources. **g)** The Knowledge4COVID-19 KG [35] comprises 80M RDF triples integrating COVID-19 scientific publications and COVID-19 related concepts (e.g., drugs, drug-drug interactions, and molecular dysfunctions). It is defined in terms of 57 RML triples maps. **h)** H2020 - SPRINT [30] studies performance and scalability of different semantic architecture for the Interoperability Framework on Transport across Europe. **i)** EIT-SNAP [31] is an innovation project on the application of semantic technologies for transport national access points. **j)** Open Cities [32] is a spanish national project on creating common and shared vocabularies for Spanish Cities. **k)** Virtual Platform for the H2020 European Joint Programme on Rare Disease [33]. **l)** CoyPU[34] is a German-funded project, where the tool is used to generate KGs for various types of events collected from economic value networks in the industrial environment and social context.

7. Conclusions and Future Work

This paper proposes data management techniques that resort to novel data structures and physical operators to execute RML triples maps efficiently. These techniques have been implemented in SDM-RDFizer v4.5.6, and the efficiency of these techniques are empirically evaluated in 416 testbeds that include the state-of-the-art RML engines and benchmarks. The outcomes put the computational power of well-designed data structures and algorithm operators in perspective, particularly in complex scenarios with star joins between several triples maps. We expect that the results reported in this paper and the available new version of SDM-RDFizer will encourage the community to declaratively define KG creation pipelines using RML and define data management techniques that can enhance the performance of their engines. For future work, we seek to define a flushing policy for the PJTT so that when the values of a particular join are no longer needed, they can be flushed, thus reducing memory consumption even further. Additionally, we want to optimize the SOM and ORM operators so that they can apply their respective transformations more efficiently. Finally, we wish to make the SDM-RDFizer capable of evaluating observational data, e.g., data from sensors.

[24] http://project-iasis.eu/
[25] https://www.bigmedilytics.eu/
[26] https://www.clarify2020.eu
[27] https://p4-lucat.eu/
[28] https://www.tib.eu/en/research-development/project-overview/project-summary/improvit
[29] https://platoon-project.eu/
[30] http://sprint-transport.eu/
[31] https://www.snap-project.eu/
[32] https://ciudades-abiertas.es/
[33] https://www.ejprarediseases.org
[34] https://coypu.org/

References

[1] C. Gutiérrez and J.F. Sequeda, Knowledge graphs, *Commun. ACM* **64**(3) (2021), 96–104.

[2] M. Lenzerini, Data Integration: A Theoretical Perspective, in: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, L. Popa, S. Abiteboul and P.G. Kolaitis, eds, ACM, 2002, pp. 233–246.

[3] S. Das, S. Sundara and R. Cyganiak, R2RML: RDB to RDF Mapping Language, W3C Recommendation 27 September 2012, *W3C* (2012).

[4] A. Dimou, T.D. Nies, R. Verborgh, E. Mannens and R.V. de Walle, Automated Metadata Generation for Linked Data Generation and Publishing Workflows, in: *Proceedings of the Workshop on Linked Data on the Web, LDOW 2016, co-located with 25th International World Wide Web Conference (WWW 2016)*, S. Auer, T. Berners-Lee, C. Bizer and T. Heath, eds, CEUR Workshop Proceedings, Vol. 1593, CEUR-WS.org, 2016.

[5] M. Lefrançois, A. Zimmermann and N. Bakerally, A SPARQL Extension for Generating RDF from Heterogeneous Formats, in: *ESWC*, 2017.

[6] A. Chebotko, S. Lu and F. Fotouhi, Semantics preserving SPARQL-to-SQL translation, *Data & Knowledge Engineering* **68**(10) (2009).

[7] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro and G. Xiao, Ontop: Answering SPARQL queries over relational databases, *Semantic Web* **8**(3) (2017), 471–487.

[8] D. Chaves-Fraga, E. Ruckhaus, F. Priyatna, M.-E. Vidal and O. Corcho, Enhancing virtual ontology based access over tabular data with Morph-CSV, *Semantic Web* **12**(6) (2021), 869–902.

[9] F. Priyatna, Ó. Corcho and J.F. Sequeda, Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph, in: *World Wide Web Conference, WWW*, C. Chung, A.Z. Broder, K. Shim and T. Suel, eds, 2014.

[10] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M.S. Pérez and O. Corcho, Morph-KGC: Scalable knowledge graph materialization with mapping partitions, *Semantic Web* (2022).

[11] U. Şimşek, E. Kärle and D. Fensel, RocketRML-A NodeJS implementation of a use-case specific RML mapper, in: *Proceeding of the First International Workshop on Knowledge Graph Building*, 2019.

[12] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, D. Collarana and M.-E. Vidal, SDM-RDFizer: An RML interpreter for the efficient creation of RDF knowledge graphs, in: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 3039–3046.

[13] D. Chaves-Fraga, K.M. Endris, E. Iglesias, Ó. Corcho and M. Vidal, What Are the Parameters that Affect the Construction of a Knowledge Graph?, in: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, Springer, 2019, pp. 695–713.

[14] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus and O. Corcho, GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain, *Journal of Web Semantics* **65** (2020), 100596.

[15] A. Dimou, M.V. Sande, P. Colpaert, R. Verborgh, E. Mannens and 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 co-located with WWW*, 2014.

[16] M. Lenzerini, Data integration: A theoretical perspective, in: *ACM Symposium on Principles of Database Systems*, 2002.

[17] A. Poggi, D. Lembo, D. Calvanese, G.D. Giacomo, M. Lenzerini and R. Rosati, Linking data to ontologies, in: *Journal on data semantics X*, Springer, 2008, pp. 133–173.

[18] A. Dimou, T. De Nies, R. Verborgh, E. Mannens and R. Van de Walle, Automated Metadata Generation for Linked Data Generation and Publishing Workflows, in: *Proceedings of the 9th Workshop on Linked Data on the Web*, S. Auer, T. Berners-Lee, C. Bizer and T. Heath, eds, CEUR Workshop Proceedings, Vol. 1593, 2016. ISSN 1613-0073.

[19] L. Asprino, E. Daga, A. Gangemi and P. Mulholland, Knowledge Graph Construction with a Façade: a Unified Method to Access Heterogeneous Data Sources on the Web, *Transactions on Internet Technology* (2022), accepted for publication.

[20] M. Lefrançois, A. Zimmermann and N. Bakerally, A SPARQL extension for generating RDF from heterogeneous formats, in: *European Semantic Web Conference*, Springer, 2017, pp. 35–50.

[21] G. Xiao, L. Ding, B. Cogrel and D. Calvanese, Virtual knowledge graphs: An overview of systems and use cases, *Data Intelligence* **1**(3) (2019), 201–223.

[22] G. Xiao, R. Kontchakov, B. Cogrel, D. Calvanese and E. Botoeva, Efficient handling of SPARQL optional for OBDA, in: *International Semantic Web Conference*, Springer, 2018, pp. 354–373.

[23] G. Xiao, D. Hovland, D. Bilidas, M. Rezk, M. Giese and D. Calvanese, Efficient ontology-based data integration with canonical IRIs, in: *European Semantic Web Conference*, Springer, 2018, pp. 697–713.

[24] K.M. Endris, P.D. Rohde, M.-E. Vidal and S. Auer, Ontario: Federated query processing against a semantic data lake, in: *International Conference on Database and Expert Systems Applications*, Springer, 2019, pp. 379–395.

[25] M.N. Mami, D. Graux, S. Scerri, H. Jabeen, S. Auer and J. Lehmann, Squerall: Virtual ontology-based access to heterogeneous and large data sources, in: *International Semantic Web Conference*, Springer, 2019, pp. 229–245.

[26] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G.d. Melo, C. Gutierrez, S. Kirrane, J.E.L. Gayo, R. Navigli, S. Neumaier et al., Knowledge graphs, *Synthesis Lectures on Data, Semantics, and Knowledge* **12**(2) (2021), 1–257.

[27] J. Arenas-Guerrero, M. Scrocca, A. Iglesias-Molina, J. Toledo, L. Pozo-Gilo, D. Doña, O. Corcho and D. Chaves-Fraga, Knowledge Graph Construction with R2RML and RML: An ETL System-based Overview, in: *Proceedings of the 2nd International Workshop on Knowledge Graph Construction*, CEUR Workshop Proceedings, Vol. 2873, 2021. http://ceur-ws.org/Vol-2873/paper11.pdf.

[28] T. Delva, J. Arenas-Guerrero, A. Iglesias-Molina, O. Corcho, D. Chaves-Fraga and A. Dimou, RML-star: A declarative mapping language for RDF-star generation, in: *ISWC2021, the International Semantic Web Conference*, 2021, pp. 1–5.

[29] M. Scrocca, M. Comerio, A. Carenini and I. Celino, Turning transport data to comply with EU standards while enabling a multimodal transport knowledge graph, in: *International Semantic Web Conference*, Springer, 2020, pp. 411–429.
[30] U. Şimşek, E. Kärle and D. Fensel, RocketRML - A NodeJS implementation of a use-case specific RML mapper, in: *Proceeding of the First International Workshop on Knowledge Graph Building*, 2019.
[31] H. Lei and K.A. Ross, Faster Joins, Self-Joins and Multi-Way Joins Using Join Indices, *Data Knowl. Eng.* **28**(3) (1998), 277–298.
[32] P. Heyvaert, D. Chaves-Fraga, F. Priyatna, O. Corcho, E. Mannens, R. Verborgh and A. Dimou, Conformance test cases for the RDF mapping language (RML), in: *Iberoamerican Knowledge Graphs and Semantic Web Conference*, Springer, 2019, pp. 162–173.
[33] M. Vidal, K.M. Endris, S. Jazashoori, A. Sakor and A. Rivas, Transforming Heterogeneous Data into Knowledge for Personalized Treatments - A Use Case, *Datenbank-Spektrum* **19**(2) (2019), 95–106.
[34] V. Janev, M.-E. Vidal, D. Pujić, D. Popadić, E. Iglesias, A. Sakor and A. Čampa, Responsible Knowledge Management in Energy Data Ecosystems, *Energies* **15**(11) (2022). doi:10.3390/en15113973.
[35] A. Sakor, S. Jozashoori, E. Niazmand, A. Rivas, K. Bougiatiotis, F. Aisopos, E. Iglesias, P.D. Rohde, T. Padiya, A. Krithara, G. Paliouras and M. Vidal, Knowledge4COVID-19: A Semantic-based Approach for Constructing a COVID-19 related Knowledge Graph from Various Sources and Analysing Treatments' Toxicities, *CoRR* **abs/2206.07375** (2022). doi:10.48550/arXiv.2206.07375.