Morph-KGC: Scalable Knowledge Graph Materialization with Mapping Partitions

Julián Arenas-Guerrero a,*, David Chaves-Fraga a, Jhon Toledo a, María S. Pérez a and Oscar Corcho a

*Ontology Engineering Group, Universidad Politécnica de Madrid, Spain
E-mails: julian.arenas.guerrero@upm.es, david.chaves@upm.es, ja.toledo@upm.es, maria.s.perez@upm.es, oscar.corcho@upm.es

Abstract. Knowledge graphs are often constructed from heterogeneous data sources, using declarative rules that map them to a target ontology and materializing them into RDF. When these data sources are large, the materialization of the entire knowledge graph may be computationally expensive and not suitable for those cases where a rapid materialization is required. In this work, we propose an approach to overcome this limitation, based on the novel concept of mapping partitions. Mapping partitions are defined as groups of mapping rules that generate disjoint subsets of the knowledge graph. Each of these groups can be processed separately, reducing the total amount of memory and execution time required by the materialization process. We have included this optimization in our materialization engine Morph-KGC, and we have evaluated it over three established benchmarks. Our experimental results show that, compared with state-of-the-art techniques, Morph-KGC presents the following advantages: i) it decreases significantly the time required for materialization, ii) it reduces the maximum peak of memory used, and iii) it scales to data sizes that other engines are not capable of processing currently.

Keywords: Knowledge Graphs, R2RML, RML, Scalability

1. Introduction

The amount of data that is being published in RDF has been steadily increasing in recent years. The generalized acceptance and use of knowledge graphs (KGs) [1] in a wide range of domains and organizations has contributed to this increase. Given that most of the data available inside organizations are structured in heterogeneous data formats, data integration techniques are often used in the data transformation and homogenization process required for knowledge graph construction (KGC).

KGC engines can be considered as data integration systems DIS = \langle O, S, M \rangle where O is the global schema expressed in terms of an ontology (or network of ontologies), S is a set of input data sources and M are the mapping rules describing the relationships between O and S [2]. Mappings are usually expressed as declarative rules, using standard specifications such as the W3C Recommendation R2RML [3] and its well-known extension for data sources beyond relational databases (RDBs), RML [4]. The construction of KGs can be done using a materialization process or by virtualization [5]. Materialization (also known as semantic extract-transform-load) uses the rules in M to transform all data into RDF. Virtualization uses M to translate SPARQL queries into the native query language of S, i.e., data integration is performed on-the-fly during query processing [6].

There are many techniques and associated implementations that can be used to create knowledge graphs integrating heterogeneous data sources using declarative mapping rules [7–14]. In the specific case of materialization, different optimizations have been proposed to speed up the materialization process in complex data integration scenarios (e.g., high rate of duplicates, large data sources, or transformation functions). Approaches such as SDM-RDFizer [15], RML-Streamer [16] and FunMap [17] propose optimizations to enhance the performance of the materialization process. In our previous work [18], in which we analyzed
several KGC engines, the experimental evaluation suggests that more efficient solutions are still needed, especially when the volume of data is large.

**Problem and objectives:** We address the problem of scalability in knowledge graph construction from heterogeneous data sources using declarative mapping rules. Our main objective is to propose the theoretical background and a set of techniques that can enhance the process of KGC in complex data integration systems, increasing the performance in both time and memory consumption.

**Proposed approach:** We present the novel concept of *mapping partitions*, which can be used to reduce the time required for the materialization of a knowledge graph and the peak amount of memory required in the process. Mapping partitions group rules in the input mapping documents ensuring the generation of disjoint sets of RDF triples by each of them. The experimental evaluation reveals that our proposal outperforms state-of-the-art engines significantly in terms of execution time and memory consumption, as well as our own implementation in the absence of this optimization.

**Contributions:** i) The novel concept of mapping partition, which allows the identification of rules that produce disjoint sets of RDF triples; ii) algorithms to find a partition of a mapping document and to remove redundant self-joins within mapping documents; iii) Morph-KGC, a scalable interpreter of R2RML and RML that implements mapping partition-based construction of the knowledge graph; iv) an empirical evaluation of our approach and a comparison against four well-known KGC engines using three different benchmarks (GTFS-Madrid-Bench [19], SDM-Genomic-Datasets [15], and NPD [20]).

The remainder of the article is structured as follows. Section 2 introduces R2RML and RML, and presents a set of concepts, notations, and conventions that will be used throughout the rest of the paper. In Section 3 we delve into the foundations of mapping partitions. Section 4 presents the experimental evaluation comparing Morph-KGC with other R2RML and RML engines. Finally, Section 5 summarizes the related work and Section 6 wraps up and outlines future work.

2. Preliminaries

In this section, we first provide some background by introducing R2RML and RML, the mapping languages that this work focuses on. Then, we present some concepts, notations, and conventions that will be used in the following sections.

2.1. R2RML and RML

R2RML [3] is the W3C Recommendation declarative mapping language that links relational databases to the RDF data model. RML [4] is a well-known extension of R2RML that supports input data formats beyond RDBs (e.g., CSV, JSON, or XML). As RML is a superset of R2RML, in this section we present the main notions of these mapping languages focusing solely on RML.

An RML mapping is represented as an RDF graph. An *RML mapping document* is an RDF document that encodes an RML mapping, and it consists of one or more *triples maps*. A triples map has one *logical source* and contains the rules to generate the RDF triples. A triples map consists of one *subject map* and zero or more *predicate-object maps*. Each predicate-object map has in turn, one or more *predicate maps* and *object maps*. Subject, predicate, and object maps are *term maps* specifying how to generate the RDF terms in the homonymous positions of the triples.

Term maps can be constant-valued (always generate the same value), reference-valued (the values are obtained directly from the logical source, e.g., a column in a table of a RDB), or template-valued (which generate RDF terms with some parts given by constants and others given by references). A template-valued term map comprises a *string template*, that defines how RDF terms are generated from one or more references (enclosed in curly braces) over the logical source. *Constant shortcut properties* are a compact method of expressing constant-valued term maps. A *referring object map* allows to generate triples in which the object map is given by the subject map of another triples map, known as the *parent triples map*. A *join condition* is used when the logical sources of both triples maps are different. The evaluation of a triples map\(^1\) produces an RDF graph whose triples consist of the generated RDF terms that result from applying its subject, predicate and object maps to the input logical source.

2.2. Assumptions, Notation and Conventions

In our work, we rely on the normalization of mapping rules as defined in [21]. A normalized mapping does not contain shortcuts, it uses predicate-object maps to type resources, and all its triples maps contain a single predicate-object map. This is not restric-

---

\(^1\)https://www.w3.org/TR/r2rml#generated-triples
tive as any R2RML or RML document can be normalized [21].

We target KGC where the resulting RDF graph does not contain duplicated triples, as assumed by most engines [7, 13, 15]. Given that an RDF graph is a set of triples [22], the presence of duplicated triples in the serialization of the RDF graph does not affect the final result. We add this restriction as it has an impact on memory and time consumption, as well as on the size of the resulting files.

We use [R2]RML to refer to R2RML and RML. Our proposal may be easily extended to other R2RML-based mapping languages. We refer to R2RML columns and RML references indistinctly as references. We refer to RDF triples and quads indistinctly. TM, SM, POM and OM denote triples map, subject map, predicate-object map and object map respectively.

Let $T$ be a term map, $\Sigma$ the set of all possible term maps, and $\Pi$ the set of positions that the RDF terms generated by the term map $T$ can occupy in a quad, i.e., $\{subject, predicate, object, graph\}$. We define position as a function mapping $\Sigma$ to $\Pi$. Assume $\Pi$ to be the set of possible positions of term maps, i.e., $\{IRI, Literal, BlankNode\}$, then type is a function mapping $\Sigma$ to $\Pi$. Assume $\pi$ to be the set of possible values that a term map can have, i.e., $\{constant, reference, template\}$, then value is a function that maps $\Sigma$ to $\pi$. Assume $C$ to be the set of possible constant values that a constant-valued term map can take, then we define const as a function mapping $\Sigma$ to $C$ where $\Sigma' = \{T \in \Sigma \mid value(T) = constant\}$.

Let $\|\$ be the set of all possible values of the specified language tags or specified datatypes (as defined in [3]), then we define literaltype as a function mapping $\Sigma'$ to $\|\$ where $\Sigma'' = \{T \in \Sigma' \mid type(T) = Literal\}$.

3. The Morph-KGC Approach

In this section, we introduce the foundations of Morph-KGC, an [R2]RML engine for constructing knowledge graphs at scale. First, we introduce self-join elimination at the mapping level. Next, we formalize the novel concept of mapping partitions. After that, we propose two algorithms to generate mapping partitions of [R2]RML documents and tackle KGC based on them. Finally, we validate the feasibility of this approach over different benchmarks and real use cases.

3.1. Self-Join Elimination in Materialization

Most virtualization engines in the state of the art (e.g., [13, 23]) remove redundant self-joins in the SQL queries that are generated, to make them more efficient. However, most materialization engines do not address self-joins that can occur in a mapping, which in most cases are executed locally by the engine.

**Definition 1 (Redundant self-join in an [R2]RML document).** A redundant self-join in an [R2]RML document appears when a referencing object map joins two triples maps with the same logical source and it has join conditions with the same references.

Redundant self-joins in an [R2]RML document can be removed by replacing the referencing object maps with object maps given by the subject map of the parent triples map, producing the same set of RDF triples.

**Example 1.** Consider the mapping rules with a self-join taken from GTFS-Madrid-Bench [19]:

```xml
<#shapes>
  rr:logicalTable [ rr:tableName "SHAPES" ];
  rr:subjectMap [ rr:template "metro:shape/{shape_id}" ];
  rr:predicateObjectMap [ rr:predicate gtf:s:shapePoint; # referencing object map (self-join) rr:objectMap [ rr:parentTriplesMap <#shapePoints> ; rr:joinCondition [ rr:child "shape_id"; rr:parent "shape_id"; ]; ]; # object map (no join) rr:objectMap [ rr:template "metro:shape_point/ {shape_id}-{shape_pt_sequence}" ] ];
</#shapes>
```

Both triples maps use the same database table, and the referencing object map uses the same column to join both triples maps. This can be transformed into an object map without a join condition (the second object map in Triples Map #shapes).

The impact of self-joins in materialization engines has been previously reported by us in [18]. We propose
Algorithm 1: Canonicalization of an [R2]RML document, M.

Result: Canonical M
1  \( M = \text{normalize}(M) \) // see [21]
2  for TM ∈ M do
3      for OM ∈ TM do
4          if isRefOM(OM) then
5              parentTM = OM.parentTM
6              if TM.source == parentTM.source
7                  removeJoin = True
8                  for joinCond ∈ OM do
9                      if joinCond.child ≠ parentT
10                         removeJoin = false
11                  end
12          end
13      end
14 end

Algorithm 1: Canonicalization of an [R2]RML document, M.

Result: Canonical M
1  \( M = \text{normalize}(M) \) // see [21]
2  for TM ∈ M do
3      for OM ∈ TM do
4          if isRefOM(OM) then
5              parentTM = OM.parentTM
6              if TM.source == parentTM.source
7                  removeJoin = True
8                  for joinCond ∈ OM do
9                      if joinCond.child ≠ parentT
10                         removeJoin = false
11                  end
12          end
13      end
14 end

to remove self-joins within the mapping documents to improve the performance of KGC engines without the need to modify their current materialization procedures. In this way, an [R2]RML document without redundant self-joins does not contain referencing object maps involving two triples maps with the same logical source and with the same references in the join conditions. This self-join elimination approach is independent from the underlying data format, as opposed to the previous techniques (e.g., [13, 23] address RDBs only).


An [R2]RML document and its canonicalization are equivalent, this entails that any transformation of a mapping document comprised in Definition 2 (i.e., normalization and redundant self-join elimination) also generates an equivalent mapping document. Algorithm 1 obtains the canonicalization of any mapping document. First, it normalizes the document (see [21]). Next, it discards referencing object maps with different logical sources in the triples map and the parent triples map (lines 4-6). After that, the algorithm checks that the fields in all join conditions match (lines 7-11). When that happens, the self-join can finally be removed, and the object map is replaced by the subject map of the parent triples map (lines 12-13).

3.2. Mapping Partitions

Given an initial set of mapping rules, we aim at identifying those that produce disjoint sets of triples, i.e., the initial mapping rules will be grouped so that those in different groups generate sets of triples that do not overlap. In the following, when we refer to the sets of generated triples, we consider them to be composed of all the triples that a mapping rule, group of mappings rules, or mapping document generate given a data source.

Definition 3 (Mapping Partition of an [R2]RML document). Let M be a canonical [R2]RML document with a set of mapping rules \( m_1, m_2, ..., m_n \) that generates the triple set \( T \). Then, a mapping partition \( P \) of M is a set of subsets of \( M \), designated as mapping groups \( \mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_k \) that generate the triple sets \( t_1, t_2, ..., t_k \), and satisfy the following conditions:

- \( \bigcap_{i=1}^{k} t_i = \emptyset \), i.e., the triple sets generated by each mapping group are disjoint.
- \( \bigcup_{i=1}^{k} t_i = T \), i.e., the union of the triple sets generated by all the mapping groups is equivalent to \( T \).

Multiple partitions can exist for \( M \). The trivial mapping partition is the one with only one mapping group, and we denote it with \( P_0 \). Mind that this definition of mapping partition does not entail that a mapping group can be considered as a new mapping document. A mapping rule in a mapping group can still have a join condition involving a rule from a different group of mappings.

Example 2. Consider the canonical mapping rules (taken from [3]):

```xml
<#TM1>
  rrl:logicalTable [ rr:tableName "DEPT" ];
  rrl:subjectMap [ 
    rr:template "ex:department/{DEPTNO}";
  ];
  rrl:predicateObjectMap [ 
    rr:predicate ex:name;
    rr:objectMap [ rr:column "DNAME" ];
  ].
</#TM1>

<#TM2>
  rrl:logicalTable [ rr:tableName "EMP" ];
  rrl:subjectMap [ 
    rr:template "ex:employee/{EMPNO}";
  ];
  rrl:predicateObjectMap [ 
```
The prefixes are obtained by eliminating the first reference and what follows. The beginning of the latter string template is data source-dependent and therefore its prefix is $\emptyset$.

**Definition 5 (Invariant of an [R2]RML term map).** The invariant $I$, of a term map $T$, is the longest common initial part of all the RDF terms that can be generated by $T$. $I$ is an intrinsic property of $T$ and remains immutable regardless of data coming from the input sources. $I$ depends on $\forall$, and it is obtained as follows:

- $I = \text{const}(T)$, when $\text{value}(T) = \text{constant}$.
- $I = \text{prefix}(T)$, when $\text{value}(T) = \text{template}$.
- $I = \emptyset$, when $\text{value}(T) = \text{reference}$.

**Notation 1 (Invariants).** For simplicity and clarity, we define the following notation for the invariants of term maps:

- $I_0$ denotes the empty invariant.
- $I_1 < I_2$, denotes that the length of $I_1$ is shorter than $I_2$, with the length given by the number of characters of the invariants.
- $I_1 \subset I_2$, denotes that $I_1$ matches the beginning of $I_2$. This entails $I_1 < I_2$.

**Example 4.** Consider the three templates in Example 3 and their invariants $I_1$, $I_2$ and $I_3$ respectively (given by their prefixes). Then, the following applies:

- $I_3 = I_0$.
- $I_3 < I_2 < I_1$.
- $I_3 \subset I_1$, $I_3 \subset I_2$, $I_2 \not\subset I_1$ and $I_1 \not\subset I_2$.

**Definition 6 (Disjoint Term Maps).** Let $T_1$ and $T_2$ be two term maps and $I_1$, $I_2$ their respective invariants. $T_1$ and $T_2$ are disjoint iff the sets of RDF terms that they can generate are in turn disjoint, regardless of the input data. The disjoint property for $T_1$ and $T_2$ applies iff at least one of the following conditions hold:

1. $\text{type}(T_1) \neq \text{type}(T_2)$.
2. $I_1 \neq I_2$, $I_1 \not\subset I_2$ and $I_2 \not\subset I_1$.
3. $I_1 < I_2$, $\text{type}(T_1) = \text{constant}$, or vice versa.
4. $\text{type}(T_1) = \text{type}(T_2) = \text{Literal}$, and $\text{literaltype}(T_1) \neq \text{literaltype}(T_2)$.

Disjointness of term maps depends on: $T_i$ invariants, and $T$ Term maps with different $T$ enforces the generation of distinct (disjoint) RDF terms (first condition). We focus now on term maps with similar $T$. For term maps with distinct invariants, the generated triple sets are disjoint if none of the invariants matches the beginning of the other (second condition). The latter is necessary to not make any assumptions on data coming from the sources. Note that in this case $I_0$ prevents term map disjointness to apply. When the value of the term map with the shortest invariant is constant, the second condition can be relaxed, and it is only neces-
disjoint iff they do not generate common triples, i.e., \( t_1 \cap t_2 = \emptyset \), regardless of the input data sources. Disjointness of \( m_1 \) and \( m_2 \) can be determined as follows:

\[
\exists T_1 \in m_1, \exists T_2 \in m_2 \mid T_1 \cap T_2 = \emptyset, \quad \text{position}(T_1) = \text{position}(T_2)
\]

For two mapping rules to be disjoint, it is required that at least two position-wise term maps are disjoint. This is true because once two triples have a different subject, predicate, object or graph then the triples are immediately distinct. Abusing of notation, given two mapping rules \( m_1 \) and \( m_2 \), we use \( m_1 \cap m_2 = \emptyset \) to denote that they are disjoint.

**Definition 8 (Disjoint Mapping Groups of an [R2]RML document).** Two mapping groups \( \mathcal{G}_1 \) and \( \mathcal{G}_2 \) of \( \mathcal{M} \) are disjoint iff they generate disjoint sets of triples. This property holds when all the mapping rules in \( \mathcal{G}_1 \) are disjoint of all the mapping rules in \( \mathcal{G}_2 \). As a consequence, a mapping rule cannot belong simultaneously to disjoint mapping groups. Formally:

\[
\forall m_1 \in \mathcal{G}_1, \forall m_2 \in \mathcal{G}_2 \mid m_1 \cap m_2 = \emptyset
\]

**Definition 9 (Maximal Mapping Partition of an [R2]RML document).** The maximal mapping partition of \( \mathcal{M} \) (denoted with \( \mathcal{P}_{\text{max}} \)) is the one with the largest number of mapping groups.

Given a mapping document, its maximal mapping partition is not necessarily unique, i.e., there may be several maximal mapping partitions for the original mapping document. When all the mapping rules in a mapping document are disjoint, each mapping group in \( \mathcal{P}_{\text{max}} \) is a singleton set.

### 3.3. Mapping Partition-Based Knowledge Graph Construction

Knowledge graph construction can leverage mapping partitions to reduce execution time and memory consumption. Before this, a partition of the mapping needs to be performed. We propose Algorithm 2, which generates partial mapping partitions by \( \mathcal{P} \), and aggregates them for further partitioning. The purpose of this algorithm is to find a good partition (i.e., with a high number of mapping groups) while keeping it simple and with a low computational cost.

The outer for-loop (line 3) of Algorithm 2 iterates four times to retrieve partial mapping partitions by subject, predicate, object and graph. The canonical map-

Result: $\mathcal{P}$ of $\mathcal{M}$
1. $\mathcal{M} = \text{canonicalize}(\mathcal{M})$
2. // Repeat for subj., pred., obj., and graph
3. for $p \in \mathbb{P}$ do
4. // Lexicographic sort
5. $\mathcal{M} = \text{sortByTermTypeAndLitTypeAndInv}(\mathcal{M}, p)$
6. $\text{curLitType} = \emptyset$
7. $\text{curTermType} = \emptyset$
8. $\text{curGroup} = 0$
9. // Iterate over TMs in $\mathcal{M}$ with a certain $p$
10. for $T \in \mathcal{M}[p]$ do
11. $I = \text{invariant}(T)$
12. if type($T$) $\neq$ curTermType then
13. $\text{curTermType} = \text{type}(T)$
14. $\text{curInv} = \emptyset$
15. $\text{curGroup} = +$
16. if type($T$) = Literal then
17. if literaltype($T$) $\neq$ curLitType then
18. $\text{curLitType} = \text{literaltype}(T)$
19. $\text{curGroup} = +$
20. else if
21. allTMsConstants($\mathcal{M}, p$) $\land$ (curInv < $I$) then
22. $\text{curGroup} = +$
23. else if curInv $\not\subset I$ then
24. $\text{curGroup} = +$
25. $\text{curInv} = I$
26. $T \text{- group} = \text{curGroup}$
27. end
28. $\mathcal{P} = \text{aggregatePartialPartitions}(\mathcal{M})$


Result: $\mathcal{P}_{\text{max}}$ of $\mathcal{M}$
1. $\mathcal{M} = \text{canonicalize}(\mathcal{M})$
2. $\mathcal{P}_{\text{max}} = \emptyset$
3. for order $\in \text{permutations}(\mathbb{P})$ do
4. for $p \in \text{order}$ do
5. $\mathcal{P} = \text{aggregatePartialPartitions}(\mathcal{M})$
6. for $G \in \mathcal{P}$ do
7. $\mathcal{P} = \text{aggregatePartialPartitions}(\mathcal{M})$
8. $\mathcal{P}_{\text{max}} = \mathcal{P}$
9. $\mathcal{M} = \text{resetPartition}(\mathcal{M})$

Partial mappings are sorted lexicographically by the $T$, I, and invariants (line 5) in the position being processed. Term maps for each $P$ are then iterated (line 10). The first condition in Definition 6 is fulfilled with lines 12-15, that create a new $G$ when a $T$ with a different $T$ is reached. When processing literal term maps (lines 16-19), if I coincides with that of the previous $T$, then they are assigned the same $G$, otherwise, a new $G$ is generated according to the fourth condition in Definition 6. If $T$ is an IRI or a blank node, we proceed to partition by invariant. When all the term maps in a specific $P$ have constant values (line 20), the third condition in Definition 6 can be applied. Otherwise, it is checked if condition 2, which is more restrictive, is fulfilled (line 22). The final $P$ is the result of aggregating the partial mapping partitions by $P$ (line 28), e.g., a mapping rule with partial mapping partitions subject(4), predicate(23), object(11) and graph(1) would be assigned the final partition 4-23-11-1.

The time complexity of Algorithm 2 is $O(n \log n)$, where $n$ is the number of mapping rules. This is because the outer for-loop can be removed by repeating its inner code four times (once per each $P$). The complexity is then determined by the lexicographic sorting of the mapping rules.

We also propose Algorithm 3 which generates the maximal mapping partition of an [R2]RML document, i.e., it solves the problem of finding $\mathcal{P}_{\text{max}}$ of a mapping document. To do so, it considers all orderings of $\mathbb{P}$ (line 3) and iterates over them (line 4). Partitioning is done independently by $G$ (lines 6-7), thus the aggregation of the partial partitions is done before (line 5) to generate these groups. Once the full partition has been created for an order of $\mathbb{P}$ (line 10), it is checked whether it has more groups than any other previously created. Finally, the partial mapping partitions are reset (line 13) to prepare them for the next order processing.

The time complexity of Algorithm 3 is upper bounded by $O(n \log n)$, where $n$ is the number of mapping rules. The worst case for lines 6-7 happens when the mapping partition has only one mapping group and they all need to be sorted (line 5 of Algorithm 2). Note that lines 3-4 do not affect time complexity since they
are fixed by $P$. Although the time complexity of Algorithms 2 and 3 are similar, the latter needs to perform more operations, since it considers every permutation of $P$.

The construction of a knowledge graph based on a mapping partition can be done in two different ways. The first one (Figure 2a) processes each mapping group sequentially. Hence, only the triples of a single mapping group are kept in memory simultaneously to remove duplicated triples. Memory usage is bounded by the largest group of mappings (in terms of the number of triples that it generates). The second one (Figure 2b) processes each group of mappings in parallel. As a consequence, the execution time is reduced at the cost of increasing the maximum memory required, as multiple triple sets of different groups of mappings are maintained in memory at the same time.

3.4. Significance of Mapping Partitions

The performance of partition-based KGC strongly depends on the ability to partition mapping documents. If the conditions required to generate mapping partitions are not generally met, then mapping partitioning would not be feasible in practice (for instance, $Z_0$ prevents partitioning in the general case). In addition, the ability to generate a high number of mapping groups affects the improvement in the performance. In general, a higher number of mapping groups entails a higher parallelization capacity (bounded by the number of CPU cores), and a lower number of mapping rules in each of the groups, and therefore less memory consumption in the case of sequential processing.

We have compiled information on mapping partitioning for several well-known benchmarks (namely, NPD [20], BSBM [24], GTFS-Madrid-Bench [19] and LSLOD [25]), the DevOps ICT knowledge graph [26], and other real uses cases from the KGC W3C Community Group\(^2\) in Table 1. We have included whether the third condition in Definition 6 applies. We select predicate maps for this purpose because in real settings constant-valued term maps usually appear in this position (and in graph maps, but they are not used in the selected cases). The number of mapping groups and the maximum number of mapping rules in a group have been obtained using Algorithms 2 and 3. In all cases it has been possible to obtain a mapping partition beyond...

\(^2\)https://github.com/kg-construct/use-cases
In this section, we experimentally evaluate our proposal. The research questions that we aim to answer are: **RQ1**: What is the impact of mapping partitions in the execution time and the memory consumption during the materialization of KGs?; **RQ2**: How does the number of groups in a mapping partition affect the materialization process?; **RQ3**: What are the benefits of this approach for constructing KGs at scale w.r.t. state-of-the-art techniques?. In the following, we describe the setup of the evaluation.

**Benchmarks.** We evaluate our proposal on three different testbeds. First, we use GTFS-Madrid-Bench [19], a benchmark in the transport domain, for testing the performance and scalability of our proposal over different tabular data formats and sizes. After that, we use the SDM-Genomic-Datasets [15] from the biomedical domain to evaluate our proposal over different mapping configurations. Finally, we use the Norwegian Petroleum Directorate (NPD) benchmark [20], from the energy domain, to compare different configurations of Morph-KGC. We have used MySQL 8.0 as DBMS for RDB. The NPD benchmark provides the mappings in R2RML, the SDM-Genomic-Datasets in RML, and GTFS-Madrid-Bench provides the mappings in both languages.

**Engines.** We use Morph-KGC v1.1.0\(^3\) and consider five configurations of it: i) Morph-KGC as the baseline (without mapping partitioning); ii) Morph-KGC\(^p\), which uses partial-aggregations for mapping partitioning and sequential processing; iii) Morph-KGC\(^m\), which uses partial-aggregations for mapping partitioning and parallel processing; iv) Morph-KGC\(^m\) which uses maximal partitioning and sequential processing; and v) Morph-KGC\(^m+p\) which uses maximal partitioning and parallel processing. We also compare our proposal against state-of-the-art KGC engines. Based on the results reported in [18], we select two R2RML engines, Onтоп v4.1.0 and R2RML-F v1.2.3, and two others that parse RML, SDM-RDFizer v4.1.1 and Chimera v2.1. In our evaluation, we consider relational databases and CSV files, SDM-RDFizer and R2RML-F can process both of them, Onтоп only processes the former and Chimera the latter. It is important to men-

---

### Table 1

Mapping partitioning of benchmarks and real use cases.

<table>
<thead>
<tr>
<th>Benchmark or Real Use Case</th>
<th>all pred. constants</th>
<th># mapping rules</th>
<th>partial-aggregations (Alg. 2)</th>
<th>maximal (Alg. 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td># (G)</td>
<td>max # rules in (G)</td>
</tr>
<tr>
<td>GTFS-Madrid-Bench</td>
<td>yes</td>
<td>86</td>
<td>83</td>
<td>2</td>
</tr>
<tr>
<td>LSLOD - Bio2RDF</td>
<td>yes</td>
<td>182</td>
<td>117</td>
<td>37</td>
</tr>
<tr>
<td>LSLOD - Linkedct</td>
<td>yes</td>
<td>143</td>
<td>126</td>
<td>14</td>
</tr>
<tr>
<td>LSLOD - TCGA</td>
<td>yes</td>
<td>2450</td>
<td>388</td>
<td>380</td>
</tr>
<tr>
<td>LSLOD - Dailymed</td>
<td>yes</td>
<td>261</td>
<td>212</td>
<td>18</td>
</tr>
<tr>
<td>NPD</td>
<td>yes</td>
<td>1177</td>
<td>477</td>
<td>116</td>
</tr>
<tr>
<td>BSBM</td>
<td>yes</td>
<td>75</td>
<td>57</td>
<td>4</td>
</tr>
<tr>
<td>Open Cities - UPM</td>
<td>yes</td>
<td>122</td>
<td>99</td>
<td>6</td>
</tr>
<tr>
<td>Btw Our Worlds - IDLab</td>
<td>yes</td>
<td>62</td>
<td>47</td>
<td>4</td>
</tr>
<tr>
<td>SDM-Genomics - TIB</td>
<td>yes</td>
<td>169</td>
<td>105</td>
<td>8</td>
</tr>
<tr>
<td>Drugs4Covid - UPM</td>
<td>yes</td>
<td>75</td>
<td>27</td>
<td>29</td>
</tr>
<tr>
<td>Data Hub - Ontopic</td>
<td>yes</td>
<td>270</td>
<td>69</td>
<td>83</td>
</tr>
<tr>
<td>DevOps ICT KG</td>
<td>yes</td>
<td>326</td>
<td>299</td>
<td>3</td>
</tr>
</tbody>
</table>

---

\(^3\)https://github.com/oeg-upm/morph-kgc
Fig. 3. Morph-KGC over GTFS-Madrid-Bench. Memory over time in the materialization of $GTFS^{rdb}$ for three different configurations of the Morph-KGC engine: without mapping partitions (Morph-KGC), with mapping partitions and sequential processing (Morph-KGC$_p$), and with mapping partitions and parallel processing (Morph-KGC$_{p+}$).

4.1. GTFS-Madrid-Bench

We consider two distributions of the GTFS-Madrid-Bench based on the data format: $GTFS^{csv}$ and $GTFS^{rdb}$. We have also generated different data sizes of these distributions considering the scaling factors: 1, 10, 100, and 1000. As reported in Table 1, the partial-aggregations and maximal partitioning algorithms return very similar mapping partitions (differing only in one mapping group). Thus, in this experiment, we only take into account partial-aggregations for mapping partitioning, avoiding the extra computational cost of maximal partitioning. While the performance of Morph-KGC and Ontop are not impacted by self-joins because they remove them, the rest of the considered engines are extraordinarily affected by them. For this reason, we have manually removed self-joins from the original mappings. Mind that Ontop is only able to process $GTFS^{rdb}$ and Chimera $GTFS^{csv}$.

The impact of mapping partitions on the materialization of large input data sources can be observed in Figure 3. Regarding memory consumption, we can observe that the baseline, Morph-KGC, follows a growing trend over time. The reason is that it keeps the entire KG in memory to avoid the generation of duplicate triples. Indeed, Figure 3b shows that this approach produces an out-of-memory issue due to the size of the final KG. In the case of Morph-KGC$_p$, it is observed
how the memory is freed every time a group of mappings is materialized. In this configuration, the maximum peak of memory is given by the largest group of mapping rules (in terms of the total number of triples generated), and it is significantly lower than the other two configurations. However, this comes at the cost of a small overhead in the execution time w.r.t. the baseline. Morph-KGC\textsuperscript{p} demonstrates a great improvement w.r.t. the baseline regarding execution time, although the maximum peak of memory used is similar due to Morph-KGC\textsuperscript{p} maintaining multiple groups of mappings in memory at the same time, as they are being processed concurrently. Note that mapping partition-based KGC is bounded by the parallelization capacity of the processor and by the mapping partition itself (e.g., the number of mapping groups or the differences in size among them).

Figure 4a shows the total execution time of Morph-KGC compared to the rest of the selected engines. Morph-KGC\textsuperscript{p} clearly outperforms the rest of the engines for all data formats and data scaling factors. The optimizations implemented by SDM-RDFizer make them the only competitor able to scale to GTFS\textsubscript{1000}. Figure 4b depicts the maximum peak of memory used by each engine. It is observed that the operators compressing data in the data structures of SDM-RDFizer achieve the lowest memory usage for GTFS\textsubscript{1} and GTFS\textsubscript{10}. In the case of larger distributions of GTFS, Morph-KGC\textsuperscript{p} outperforms SDM-RDFizer given that it reduces the maximum peak of memory used to that of the largest mapping group.

4.2. SDM-Genomic-Datasets

The SDM-Genomic-Datasets\textsuperscript{4} provide a set of configurations taking into account different parameters that are relevant for constructing knowledge graphs [27] such as the type of mappings, the number of duplicates, and data size. More in detail, regarding the latter, four different datasets are provided with different number of rows: 10K, 100K, 1M, and 10M. Although simpler configurations are also provided in terms of the number of duplicates, for this experimental evaluation we select the most complex one, i.e., 75% of duplicates with each duplicated value repeated 20 times.

\textsuperscript{4}https://figshare.com/articles/dataset/SDM-Genomic-Datasets/14838342/1
In addition, three mapping files with different types of predicate-object maps are considered: simple object map (POM), referencing object map with self-reference (REF), and referencing object map (JOIN). A number is used together with the name of each mapping type to specify the number of rules (e.g., 4-POM indicates 4 object maps). The testbeds provide the mappings in RML and data is in the form of CSV files. For this reason, we do not consider the R2RML processors in this experiment. In the same manner as for GTFS-Madrid-Bench, we only report the time and memory consumption for the case of partial-aggregations partitioning as the maximal partitioning algorithm generates the same mapping partition.

The total execution times for SDM-Genomic-Datasets are reported in Figure 5a. As Morph-KGC\textsuperscript{px} and Morph-KGC perform a self-join elimination over the REF mappings, they obtain similar results as in the POM ones, which is not the case of SDM-RDFizer and Chimera. While Morph-KGC\textsuperscript{px} obtains the best results for the former configurations, SDM-RDFizer clearly outperforms it for JOIN mappings. The main reason is that SDM-RDFizer implements the Predicate Join Tuple Table as a specific physical data structure for improving the join conditions during the construction of the KG, and Morph-KGC does not implement any join optimization beyond redundant self-join elimination.

Figure 5b shows that Morph-KGC\textsuperscript{px} outperforms its baseline and state-of-the-art engines regarding memory consumption for POM and REF mappings. In the case of JOIN mappings, SDM-RDFizer obtains the best results given the data compression techniques of its structures.

4.3. Norwegian Petroleum Directorate Benchmark

The NPD benchmark [20] presents a comprehensive evaluation system for virtual KGC engines. It provides a set of SPARQL queries, a scalable instance of an RDB from the energy domain, and the corresponding mapping rules in R2RML. Although it has not been previously used for testing the performance of materialization engines, we notice that it is the only benchmark among those considered in Table 1 in which the difference in the number of mapping groups obtained by the partial-aggregations and the maximal partitioning algorithms is significant. This will allow us to compare the impact of maximal partitioning when it achieves a mapping partition with more
groups. To further evaluate mapping partitions with different number of groups, we also include the configurations Morph-KGC$^m$-s and Morph-KGC$^m$-p which means that only subject and predicate, respectively, are taken into account to perform mapping partitioning (instead of every P). We use the data generator of the benchmark [28] to obtain three different distributions with data scaling factors: 1, 10 and 100. Apart from our proposal, the only engine able to parse the R2RML mappings and generate the correct KG is R2RML-F.

The results obtained are shown in Figure 6. We observe that the best performance regarding execution time is obtained by Morph-KGC$^p$. Surprisingly, the partition generated by Morph-KGC$^p$ reports better results than the Morph-KGC$^m$ one, showing that a higher number of mapping groups does not always entail a better execution time in the construction of the KG. A possible reason for this could be that the parallel processing is bounded by the number of cores of the machine (20 in our case), and increasing the number of mapping groups (477 for Morph-KGC$^p$ while there are 745 for Morph-KGC$^m$) does not result in a higher parallelization rate. Moreover, a higher number of mapping groups introduces an overhead as we saw previously in Figure 3, and maximal partitioning is computationally more expensive. However, we observe that the materialization time of Morph-KGC$^m$ is very close to Morph-KGC$^p$, and that these two perform significantly better than Morph-KGC$^m$-s and Morph-KGC$^m$-p, with a lower number of mapping groups (17 and 327 respectively). This indicates that in general, it is desirable to have a high number of mapping groups to increase the parallelization rate. Regarding memory consumption, we see that Morph-KGC$^p$ and Morph-KGC$^m$ obtain similar results. Note that in sequential processing, the peak in the amount of memory used is determined by the largest mapping group. If maximal partitioning is not able to further partition that specific mapping group, then a reduction in the peak of memory consumption is not expected.

4.4. Discussion

The experiments with SDM-Genomic-Datasets show that self-join elimination is an effective technique and that it reduces the execution time of materialization. The same behaviour was also reported by us in [18] for GTFS-Madrid-Bench. Since our technique applies
5. Related Work

Ontology-based data integration [5, 6] systems differentiate between the data layer, composed of the data sources, and the conceptual layer, in which an ontology or network of ontologies are used to abstract the heterogeneity of the former. Mappings are used to specify how to populate entities and relationships in the ontology with data from the underlying data sources, i.e., mappings are the link mechanism between both layers.

Several languages have been proposed to define mappings [29–31], but the one that stands out most notably is R2RML [3], the W3C Recommendation to map relational databases to RDF. However, R2RML does not generalize the underlying data model and cannot deal with heterogeneous data formats. RML [4] is a well-known superset of R2RML that removes specific references to the relational data model and enables data formats beyond RDBs. In addition, other R2RML-related proposals have addressed transformation functions [7, 32], mixed content and RDF collections [11], usability [33] or scalability [34].

There are two approaches to process the mappings: virtualization (or query translation) and materialization (or data translation). Virtualization uses mappings to translate SPARQL queries into the native query languages of the underlying data sources. Research around this technique has focused primarily on relational databases and on efficiently processing the generated SQL queries. We refer the interested reader on virtualization to [8, 12, 13, 23, 35]. Materialization uses the mappings to transform all data in the underlying data sources to the corresponding RDF.

There are several solutions targeting the materialization of knowledge graphs [18]. For the specific case of RDBs, Ontop [13, 36] leverages the fact that predicate maps are generally constant-valued. It generates one SPARQL query for each predicate with unbounded subject and object. These queries are then translated to SQL and optimized by applying a set of structural optimizations (e.g., subqueries elimination) and semantic query optimizations (e.g., redundant self-joins removal). It also avoids to retrieve large query result sets at once by doing it in chunks.

The work presented in [37] exploits knowledge encoded in the mapping documents to project the attributes appearing in a triples map, reducing the size of the data sources that need to be processed. Similarly, to diminish the impact of duplicates in the evaluation of join conditions, it also pushes down projections into joins. SDM-RDFizer [15] proposes physical
data structures to store the knowledge graph in memory in a way that allows to efficiently remove duplicates and avoid unnecessary operations. In particular, it uses one hash table for each predicate (called Predicate Tuple Table, where the hash key combines the subject and the object of the triple, and the value is the triple itself. It also proposes to speed up joins by creating the Predicate Join Tuple Table, a hash table using the values matching the join condition as the hash key, and being the values of the hash the set of the generated values by the parent triples map. These hash tables are checked every time a new triple is to be generated, in the case that the triple already exists, it is discarded, otherwise it is added to the knowledge graph, and the corresponding hash table is updated. Moreover, SDM-RDFizer considers data compression techniques that reduce the memory usage of the data structures that store intermediate results during materialization.

Recently, triples map planning has been proposed in [38]. Here a bushy tree plan is created specifying an optimized execution order for executing mapping assertions. These bushy tree plans are obtained heuristically by relying on a greedy algorithm. Operating system commands are obtained from these execution plans that allow to efficiently execute different knowledge graph materialization engines.

Karma [14] tackles the scalability of knowledge graph construction from large data sources using batch processing. Instead of loading all data into memory, when operating in batch mode, the engine continuously loads fractions of the data, transforming them to the nested relational model and then materializing the corresponding triples. In this manner, memory usage is reduced, as it is not required to maintain the entire knowledge graph in memory.

Parallelization has also been proposed to speed up the materialization process. The work presented in [16] divides this process in three tasks following the producer-consumer paradigm: ingestion of data from the sources, mapping to RDF, and combination of the RDF. Parallelization is done up to the data record level. Nevertheless, the proposed approach does not tackle duplicate elimination. [15, 39] also parallelize at the triples map level.

Regarding the efficient processing of transformation functions, FunMap [17] addresses the materialization of these functions specified using FnO [40]. FunMap proposes a set of lossless rewriting rules so that the functions are executed in the initial steps of the KG materialization process. This approach generates a set of mapping rules that are function-free and that can be executed by any RML engine.

Mapping partitioning can be used together with existing techniques to further optimize the process of knowledge graph materialization. For instance, the Predicate Join Tuple Table could be implemented along with mapping partitioning to speed up joins in parallel and sequential processing. Moreover, redundant self-join elimination could be implemented in a preprocessing step (that can include other techniques such as the rewriting of rules in FunMap), which generates mappings without self-joins that can be executed by any engine.

6. Conclusions and Future Work

We address the problem of scalability in the materialization of knowledge graphs from heterogeneous data sources using declarative mapping rules. We present the novel concept of mapping partitions, which consists in grouping mapping rules that generate disjoint sets of RDF triples. Mapping partitions can be used to reduce memory consumption in KG materialization by processing each mapping group in a partition sequentially, or to decrease the execution time by processing multiple mapping groups in parallel. We implement this novel approach in an [R2]RML engine, Morph-KGC, and we empirically demonstrate that, in the general case, it outperforms state-of-the-art proposals in terms of the total execution time and the amount of memory required in the materialization process.

Our future lines of work include the extension of Morph-KGC and mapping partitions to RML-star [41], which poses the challenge of recursive and more complex mapping rules. We also plan to address the limitation of the current approach that prevents minimizing materialization time and memory consumption at the same time, by using standards such as MPI [42].

References


