Query-Specific Pruning of RML Mappings

Sitt Min Oo[1] and Olaf Hartig[2]

[1] Ghent University - imec, Ghent, Belgium
x.sittminoo@ugent.be
[2] Linköping University, Linköping, Sweden
olaf.hartig@liu.se

Abstract. Current approaches for knowledge graph construction with RML focus on full RDF graph materialization without considering user queries, which is inefficient in dynamic query environments where often only a specific subset of the full graph is needed to answer a given query. This paper introduces an approach to prune RML mappings such that the resulting partially-materialized graph is still sufficient to answer a given SPARQL query completely. By evaluating the approach based on a well-know RML materialization benchmark, we show that such pruning significantly reduces both the materialization time and the size of the produced graph, while also noticeably reducing querying time.

1 Introduction

State-of-the-art knowledge graph generation approaches support mapping languages such as R2RML [21], RML [6,10], and SPARQL-Generate [12] to provide access to RDF views of other forms of structured data (e.g., relational databases) as well as semi-structured data (e.g., JSON, XML). These approaches either materialize the RDF view [9,1,7,14,17], making it available for querying or further processing directly as RDF graphs, or translate SPARQL queries over the RDF views into a query language supported by the underlying data sources [5,3,13].

In the context of use cases that require integrated query access over a federation of multiple data sources, including non-RDF ones, we observe that both of these two types of approaches pose practical limitations: Relying on query translation (also called virtualization) would limit a federation engine to types of data sources that support a query language. While materialization approaches, in contrast, can access arbitrary types of data sources, including those without a query language, materialization approaches can easily become inefficient in this setting as they are designed to always produce the full RDF view of the source data, even if the part(s) of the federation query to be answered via this view need only a smaller part of the view to be answered completely.

This work introduces an approach that combines the support for a wider range of data sources from materialization with the query-aware efficiency of virtualization. The approach focuses on mappings described in RML and the core idea is to prune away the parts of an RML mapping that define portions of the resulting RDF view that are irrelevant for answering a given SPARQL

query (which may be a sub-query assigned to a non-RDF data source within a federation query [8]). The pruned mapping can then be used to materialize a smaller RDF graph for which the query returns the same result as for the full RDF view. Section 2 illustrates the approach with an example.

Our main technical contribution is an algorithm that captures the pruning approach formally (Section 6), for which we assume that the mapping to be pruned is given in the algebraic form introduced in our previous work [15] (summarized in Section 3). As a first step, we define a notion of satisfiability of SPARQL graph patterns over such algebraic mapping expressions and show that this notion of satisfiability is undecidable, which constitutes another technical contribution (Section 4). We then introduce a syntactic property to identify a class of cases in which a triple pattern is guaranteed to be not satisfiable over a specific mapping expression, and we show the correctness of this property (Section 5). This formal result provides the foundation of the pruning algorithm.

As our second main contribution, we evaluate the effectiveness of our pruning approach based on the GTFS-Madrid benchmark (Section 7); our evaluation confirms that pruning can reduce the materialization time significantly (down to at most 8% of the full materialization time in 2/3 of the considered cases), while the pruning time is negligible. Moreover, the resulting RDF graphs are much smaller as well, which can also lead to a noticeable reduction of query times.

2 Demonstration of the Approach

We begin by illustrating our pruning approach informally, for which we assume familiarity with the concepts of RDF, SPARQL, and RML. Listing 1 presents a snippet of an example RML document that defines a mapping to generate RDF triples about the transit routes (lines 4–7) and the longitude coordinate (lines 8–11) of airports. Listing 2 provides a SPARQL query with two triple patterns to query the resulting RDF data. We go through each predicate-object map of the RML mapping and decide if it can be pruned by considering each of the triple patterns of the query.

```
...                                                          1
rml:subjectMap [ rml:reference "airport_id" ;                2
                 rml:termType rml:IRI ];                     3
rml:predicateObjectMap [                                     4
 rml:predicate ex:route;                                     5
 rml:objectMap [ rml:template                                6
    "http://example.com/route/{transitRoute}" ] ];           7
rml:predicateObjectMap [                                     8
 rml:predicate gtfs:long;                                    9
 rml:objectMap [ rml:reference "longitude";                  10
                 rml:datatType xsd:double ] ].               11
```

Listing 1: Snippet of an RML mapping to convert airport data into an RDF graph.

```
SELECT * WHERE {
 ?airportId ex:route <http://transit.api/route/43> .
 ?airportId gtfs:long "23.0"^^xsd:double .  }
```

Listing 2: A SPARQL query to retrieve information about airports (prefix declarations omitted).

For the first predicate-object map (lines 4–7), we first consider the first triple pattern. While the subject and the predicate of possible triples produced from the predicate-object map match the triple pattern, the object of the triple pattern cannot be produced by the corresponding object-term

map. To determine this type of unsatisfiability, we transform the `rml:template` value of the object-term map, `"http://example.com/route/{transitRoute}"`, into the regular expression `"http:\/\/example.com\/route\/.+"`. It is evident that the object IRI of the triple pattern, which begins with the substring `"http://transit.api"`, does not match the regular expression. Thus, the first triple pattern is not satisfiable for the first predicate-object map. Moving to the second triple pattern, it is also evident that the first predicate-object map cannot produce triples that match the pattern. The predicate-term is the constant `ex:route`, which is not the same as the predicate IRI `gtfs:long` of the triple pattern. We thus conclude that, since none of the triple patterns is satisfiable with it, the first predicate-object map can be pruned.

For the second predicate-object map (lines 8–11), triples produced from it do not match the first triple pattern because they would have `gtfs:long` as predicate, not `ex:route`. However, the second triple pattern may be satisfiable with that predicate-object map since the predicate IRIs match in this case and, for checking the object terms, we apply the following procedure. We first transform the `rml:reference` value `"long"` of the object-term map into the regular expression `".+"`. The lexical form of the object literal in the second triple pattern matches this regular expression. Moreover, the datatype declared in the object-term map, `xsd:double`, matches the datatype of the object literal in the triple pattern. Thus, at least one triple pattern may be satisfiable with the second predicate-object map, which means this predicate-object map should not be pruned away. Hence, at the end, only the second predicate-object map of the snippet of RML in Listing 1 is kept after pruning.

The *main use case of this approach* is SPARQL-based query federation with non-RDF data sources that cannot be accessed via a query language. Since virtualization is not an option for such sources, their data needs to be converted directly to RDF to be queried via SPARQL. As a typical example of such cases, we refer to recent work by Hartig and Westman [8] which integrates REST/Web APIs within a federation engine that materializes RML-based RDF representations of the API data at query time. Query-time materialization is relevant in this setting because each federation query may require data from different API endpoints; moreover, the API data may change often (e.g., a weather API [8]).

3 Preliminaries

We formalize our pruning approach in terms of the RML-related mapping algebra of our earlier work [15]. To provide the relevant background for this formalization, this section introduces the concepts of that algebra, and of RDF and SPARQL.

3.1 Relevant Concepts of RDF and SPARQL

Let $\mathcal{S}$ be the countably infinite set of all possible strings, and $\mathcal{I}$ be the subset of $\mathcal{S}$ that consists of all IRIs. $\mathcal{L}$ is the countably infinite set of all RDF literals where every such literal is a pair $(lex, dt) \in \mathcal{S} \times \mathcal{I}$ in which lex is the lexical

form and dt is the datatype IRI of the literal. Moreover, $\mathcal{B}$ is the countably infinite set of blank nodes (and is disjoint from $\mathcal{S}$ and $\mathcal{L}$). IRIs, literals, and blank nodes are jointly referred to as *RDF terms*. An *RDF triple* is a tuple $(s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$, and an *RDF graph* is a set of RDF triples.

For our definitions related to SPARQL we adopt the algebraic syntax of Pérez et al. [18]. Queries are formed using *graph patterns*, of which the most basic type is a *triple pattern*, that is, a tuple $(s, p, o) \in (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{L} \cup \mathcal{V})$, where $\mathcal{V}$ is a countably infinite set of variables (disjoint from $\mathcal{S}$, $\mathcal{L}$, and $\mathcal{B}$). Other graph patterns can then be constructed recursively, using operators such as `AND` and `OPT` [18]. The result of evaluating any such graph pattern P over an RDF graph G is a set, denoted by $[\![P]\!]_G$, that consists of so-called *solution mappings*, which are partial functions of the form $\mu : \mathcal{V} \to \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. If P is a triple pattern $tp = (s, p, o)$, then $[\![tp]\!]_G$ consists of every solution mapping μ for which $\mathrm{dom}(\mu) = \{s, p, o\} \cap \mathcal{V}$ and $\mu[tp] \in G$, where $\mu[tp]$ denotes the triple obtained by replacing the variables in tp according to μ. For other forms of graph patterns, we refer to Pérez et al.'s work for the definition of $[\![P]\!]_G$ [18].

3.2 Data Model of the Mapping Algebra

The mapping algebra of our earlier work is defined over so-called *mapping relations* [15] in which the possible values are RDF terms, plus a special value, ϵ, that captures processing errors (and is not an RDF term). For the following formal definition of these relations, let $\mathcal{A}$ be a countably infinite set of *attributes*.

Definition 1.[15] A **mapping tuple** is a partial function $t : \mathcal{A} \to \mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \{\epsilon\}$

Definition 2.[15] A **mapping relation** r is a tuple (A, I), where $A \subset \mathcal{A}$ is a finite, non-empty set of attributes and I is a set of mapping tuples such that, for every such tuple $t \in I$, it holds that $\mathrm{dom}(t) = A$.

While the mapping algebra operates over such mapping relations, the final mapping relation that results from a relevant sequence of such operations is meant to capture an RDF dataset. At this point we diverge slightly from the original formalism [15], which considers the creation of whole RDF datasets (i.e., including named graphs); in this paper we limit ourselves to single RDF graphs. For this purpose, we assume three special attributes, $a_\mathrm{s}, a_\mathrm{p}, a_\mathrm{o} \in \mathcal{A}$, and adapt the definition of an RDF representation of mapping relations as follows.

Definition 3. Let $r = (A, I)$ be a mapping relation with $a_\mathrm{s}, a_\mathrm{p}, a_\mathrm{o} \in A$. The **RDF graph resulting from** r is the RDF graph

$$G = \big\{ \big(t(a_\mathrm{s}), t(a_\mathrm{p}), t(a_\mathrm{o})\big) \,\big|\, t \in I \text{ such that } \big(t(a_\mathrm{s}), t(a_\mathrm{p}), t(a_\mathrm{o})\big) \text{ is an RDF triple} \big\}.$$

For examples of these concepts, refer to our earlier work [15].

3.3 Syntax of RML-Specific Mapping Expressions

This section introduces the syntax of the mapping expressions that we consider in this paper, which is based on our earlier-introduced mapping algebra [15].

While this algebra is of a more general nature, not specific to any concrete mapping language such as RML, we also introduced a translation of RML into the algebra [15, Section 5]. The fragment of the algebra that this translation uses is the focus of our work in this paper. Therefore, instead of re-introducing the complete algebra here, we introduce only the relevant types of expressions that cover any possible output of translating RML into the algebra as per the translation algorithm of our earlier work [15]. To define these types of mapping expressions we need to introduce a number of related concepts first.

We begin with concepts related to the Extract operator[3] of the algebra, which initializes a mapping relation that provides a relational view of data that can be extracted from input data sources. The definition of this operator is based on an abstraction of source data and corresponding query languages: The infinite sets $\mathcal{D}$ and $\mathcal{Q}$ capture all possible data objects and all possible query languages, respectively. Examples of data objects are: the whole content of a particular JSON file, a single JSON object, and a value of a JSON field. Data objects of the same kind (e.g., all possible JSON objects) would be captured as a dedicated subset of $\mathcal{D}$. Each query language $L \in \mathcal{Q}$ is considered as a set, where every element $q \in L$ is one of the queries written in L. Types of data sources, as considered by the Extract operator, are captured as a tuple $type = (\mathcal{D}^{ds}, \mathcal{D}^{c1}, \mathcal{D}^{c2}, L, L', eval, eval', cast)$ where $\mathcal{D}^{ds} \subseteq \mathcal{D}$ specifies the kind of data objects that can be accessed from data sources of this type; $L \in \mathcal{Q}$ is a language to enumerate components of any data object in $\mathcal{D}^{ds}$, where $\mathcal{D}^{c1} \subseteq \mathcal{D}$ is the set of all possible such components; $L' \in \mathcal{Q}$ is a language to select values from the components, with $\mathcal{D}^{c2} \subseteq \mathcal{D}$ being the set of all values possible for the type of data source; *cast* is a function defining how these values map to RDF literals; and *eval* and *eval′* are functions that define the evaluation semantics of L and L', respectively. Further details and examples are in the original paper [15].

RML mappings contain references to the data sources from which the input data is meant to be obtained (e.g., file names). As a corresponding abstraction, we assume a countably infinite set $\mathcal{R}$ of so-called *source references*. As we shall see, every Extract operator is parameterized with such a source reference.

Another relevant operator is Extend, which is parameterized with an attribute a and a so-called *extend expression* [15] that can be evaluated with a mapping tuple as input and that produces an RDF term or the error symbol ϵ as output. For every input tuple, Extend uses this output to extend the tuple with a value for attribute a. The specific type of extend expressions used by the fragment of the mapping algebra considered in this paper is defined as follows.

Definition 4. A **template-or-reference-based extend expression (torb-extend expression)** is an extend expression φ of any of the following forms:

1. φ is an RDF literal (lex, dt) with $dt =$ `xsd:string`.
2. φ is an attribute in $\mathcal{A}$.
3. φ is of the form $\texttt{concat}(\varphi_1, \ldots, \varphi_n)$, where $n \geq 2$ and every φ_i $(1 \leq i \leq n)$ is either a literal (lex, dt) with $dt =$ `xsd:string` or an attribute in $\mathcal{A}$.

[3] In the original work this operator is called Source [15]. We have renamed it for this paper because the name Extract captures more clearly the purpose of this operator.

Hereafter, we write Φ_{torb} to denote the set of all possible torb-extend expressions and, for every such expression $\varphi \in \Phi_{\mathsf{torb}}$, attrs($\varphi$) is the set of all attributes in φ.

The forms of algebra expressions that the RML-to-algebra translation algorithm [15] produces and, thus, that we consider in this paper all share a common form of sub-expressions, which result from translating individual RML triples maps into the algebra. As the last ingredient needed for defining the considered fragment of the algebra, we introduce this form of sub-expressions:

Definition 5. Let u_{base} be an IRI (considered as *base IRI*) and $S2B\colon \mathcal{S} \to \mathcal{B}$ be an injective function that maps every string to a unique blank node. A **Triples-Map-specific expression (TrMap-expression)** with u_{base} and $S2B$ is either

$$\mathsf{Extend}^{a_{\mathrm{o}}}_{\varphi_{\mathrm{o}}}(\quad \mathsf{Extend}^{a_{\mathrm{p}}}_{\varphi_{\mathrm{p}}}(\mathsf{Extend}^{a_{\mathrm{s}}}_{\varphi_{\mathrm{s}}}(\mathsf{Extract}^{(type,q,\mathbb{P})}_{sr}))\ )\quad \text{or}$$

$$\mathsf{Extend}^{a_{\mathrm{o}}}_{\varphi'_{\mathrm{o}}}(\ \mathsf{EqJoin}^{\mathbb{J}}(\mathsf{Extend}^{a_{\mathrm{p}}}_{\varphi_{\mathrm{p}}}(\mathsf{Extend}^{a_{\mathrm{s}}}_{\varphi_{\mathrm{s}}}(\mathsf{Extract}^{(type,q,\mathbb{P})}_{sr})), \mathsf{Extract}^{(type',q',\mathbb{P}')}_{sr'})\ ),$$

where:

1. φ_{o} is an extend expression [15] of any of the following specific forms:
   (a) a literal, (b) an IRI, (c) a blank node,
   (d) $\texttt{toLiteral}(\varphi', dt)$ with $\varphi' \in \Phi_{\mathsf{torb}}$, attrs($\varphi$) $\subseteq$ dom($\mathbb{P}$), and $dt \in \mathcal{I}$,
   (e) $\texttt{toIRI}(\varphi', u_{\mathsf{base}})$ with $\varphi' \in \Phi_{\mathsf{torb}}$ and attrs(φ) $\subseteq$ dom($\mathbb{P}$), or
   (f) $\texttt{toBNode}^{S2B}(\varphi')$ with $\varphi' \in \Phi_{\mathsf{torb}}$ and attrs(φ) $\subseteq$ dom($\mathbb{P}$);
2. φ_{p} is an extend expression that may be only of the form (b) or (e);
3. φ_{s} and φ'_{o} are extend expressions of the form (b), (c), (e), or (f), respectively;
4. sr and sr' are source references (potentially the same);
5. $type = (\mathcal{D}_1^{\mathrm{ds}}, \mathcal{D}_1^{\mathrm{c1}}, \mathcal{D}_1^{\mathrm{c2}}, L_1, L'_1, eval_1, eval'_1, cast_1)$ is a source type, $q \in L_1$, and $\mathbb{P}\colon \mathcal{A} \to L'_1$ is a partial function such that dom($\mathbb{P}$) $\cap \{a_{\mathrm{s}}, a_{\mathrm{p}}, a_{\mathrm{o}}\} = \emptyset$;
6. $type' = (\mathcal{D}_2^{\mathrm{ds}}, \mathcal{D}_2^{\mathrm{c1}}, \mathcal{D}_2^{\mathrm{c2}}, L_2, L'_2, eval_2, eval'_2, cast_2)$ is a source type, $q' \in L_2$, and $\mathbb{P}'\colon \mathcal{A} \to L'_2$ is a partial function such that dom($\mathbb{P}'$) $\cap \{a_{\mathrm{s}}, a_{\mathrm{p}}, a_{\mathrm{o}}\} = \emptyset$;
7. dom($\mathbb{P}$) $\cap$ dom($\mathbb{P}'$) $= \emptyset$;
8. $\mathbb{J} \subseteq$ dom($\mathbb{P}$) $\times$ dom($\mathbb{P}'$).

Example 1. The triples map that consists of the first predicate-object map of the RML mapping in Listing 1 is captured by the following TrMap-expression:

$$\mathsf{Extend}^{a_{\mathrm{o}}}_{\texttt{toIRI}(\texttt{concat}(\ell,a_1),u_{\mathsf{base}})}(\mathsf{Extend}^{a_{\mathrm{p}}}_{u_1}(\mathsf{Extend}^{a_{\mathrm{s}}}_{\texttt{toIRI}(a_2,u_{\mathsf{base}})}(\mathsf{Extract}^{(type,q,\mathbb{P}_1)}_{sr}))),$$

where a_1 and a_2 are attributes different from a_{s}, a_{p}, and a_{o}, respectively, ℓ is the literal (`"http://example.com/route/"`, `xsd:string`), u_1 is the IRI `ex:route`, $\mathbb{P}_1 = \{a_1 \mapsto$ `"transitRoute"`, $a_2 \mapsto$ `"airport_id"`$\}$, and sr is an arbitrary source reference. The other two arguments of the Extract operator, *type* and q, are not specified further in this example as they depend on the `rml:logicalSource` of the triples map, which is not in Listing 1 (and is irrelevant for our pruning approach).

We can now define the notion of an RML-specific mapping expression. Informally, such an expression is built by wrapping a TrMap-expression into a Project operator that keeps only the attributes a_{s}, a_{p}, and a_{o}, and by combining multiple such Project-wrapped TrMap-expressions using Union operators. Formally:

Definition 6. Let u_{base} be an IRI and $S2B : \mathcal{S} \to \mathcal{B}$ be an injective function that maps every string to a unique blank node. An **RML-specific mapping expression** with u_{base} and $S2B$ is defined recursively as follows:

1. For every TrMap-expression M_{TM} with u_{base} and $S2B$, and the (fixed) set $P = \{a_{\mathrm{s}}, a_{\mathrm{p}}, a_{\mathrm{o}}\}$, $\mathsf{Project}^{P}(M_{\mathsf{TM}})$ is an RML-specific mapping expression.
2. For two RML-specific mapping expressions M' and M'' such that M'' is of the form $\mathsf{Project}^{P}(M_{\mathsf{TM}})$, $\mathsf{Union}(M', M'')$ is an RML-specific mapping expression.

For every RML-specific mapping expression M, we write $\mathrm{TrMaps}(M)$ to denote the set of all TrMap-expressions contained in M.

While the notions of an RML-specific mapping expression and of a TrMap-expression are defined with respect to an IRI u_{base} and a function $S2B$, hereafter, we mention u_{base} and $S2B$ only in cases in which they are explicitly relevant.

3.4 Semantics of RML-Specific Mapping Expressions

This section introduces a formal semantics of RML-specific mapping expressions. As a basis for evaluating such an expression, it is necessary to assign concrete data objects to the source references mentioned in the $\mathsf{Extract}$ operators of the expression, for which we introduce the notion of a source assignment.

Definition 7. A **source assignment** is a partial function $\sigma \colon \mathcal{R} \to \mathcal{D}$.

Notice that such a source assignment may not be applicable to a given mapping expression. For instance, it may not cover all of the source references that occur within the expression or it may assign data objects that are not of the expected types. The following definition formalizes the conditions for a source assignment to be applicable, for the types of expressions considered in this paper.

Definition 8. A source assignment σ is a **valid input** for a TrMap-expression M_{TM} if the following conditions hold:

1. If M_{TM} is of the form $\mathsf{Extend}^{a_{\mathrm{o}}}_{\varphi_{\mathrm{o}}}(\mathsf{Extend}^{a_{\mathrm{p}}}_{\varphi_{\mathrm{p}}}(\mathsf{Extend}^{a_{\mathrm{s}}}_{\varphi_{\mathrm{s}}}(\mathsf{Extract}^{(type,q,\mathbb{P})}_{sr})))$ with $type = (\mathcal{D}^{\mathrm{ds}}, \mathcal{D}^{\mathrm{c1}}, \mathcal{D}^{\mathrm{c2}}, L, L', eval, eval', cast)$, then it must hold that $\sigma(sr) \in \mathcal{D}^{\mathrm{ds}}$.
2. If M_{TM} is of the form
$\mathsf{Extend}^{a_{\mathrm{o}}}_{\varphi'_{\mathrm{o}}}(\mathsf{EqJoin}^{\mathbb{J}}(\mathsf{Extend}^{a_{\mathrm{p}}}_{\varphi_{\mathrm{p}}}(\mathsf{Extend}^{a_{\mathrm{s}}}_{\varphi_{\mathrm{s}}}(\mathsf{Extract}^{(type,q,\mathbb{P})}_{sr})), \mathsf{Extract}^{(type',q',\mathbb{P}')}_{sr'}))$
with $type = (\mathcal{D}^{\mathrm{ds}}_1, \mathcal{D}^{\mathrm{c1}}_1, \mathcal{D}^{\mathrm{c2}}_1, L_1, L'_1, eval_1, eval'_1, cast_1)$ and
and $type' = (\mathcal{D}^{\mathrm{ds}}_2, \mathcal{D}^{\mathrm{c1}}_2, \mathcal{D}^{\mathrm{c2}}_2, L_2, L'_2, eval_2, eval'_2, cast_2)$,
then it must hold that $\sigma(sr) \in \mathcal{D}^{\mathrm{ds}}_1$ and $\sigma(sr') \in \mathcal{D}^{\mathrm{ds}}_2$.

A source assignment σ is a **valid input** for an RML-specific mapping expression M if σ is a valid input for every TrMap-expression in $\mathrm{TrMaps}(M)$.

Given the notion of valid inputs for evaluating RML-specific mapping expressions, we can now define the semantics of such an evaluation.

Definition 9. Let M_{rml} be an RML-specific mapping expression and σ be a source assignment that is a valid input for M_{rml}. For every sub-expression M of M_{rml}, including M_{rml} itself, the **evaluation of** M **based on** σ, denoted by $M[\sigma]$, is the mapping relation (A, I) that is defined recursively as follows:

1. If M is $\mathsf{Extract}_{sr}^{(type,q,\mathbb{P})}$ with $type = (\mathcal{D}^{\mathsf{ds}}, \mathcal{D}^{\mathsf{c1}}, \mathcal{D}^{\mathsf{c2}}, L, L', eval, eval', cast)$, then

   $A = \mathrm{dom}(\mathbb{P})$ and

   $I = \big\{\{a_1 \to cast(v_1), \ldots, a_n \to cast(v_n)\} \;\big|\; d \text{ is in } \bar{O} \text{ and } ((a_1, v_1), \ldots, (a_n, v_n)) \in X_d\big\}$,

   where $\bar{O} = eval(D, q)$ with $D = \sigma(sr)$, and

   $$X_d = \bigtimes_{a \in \mathrm{dom}(\mathbb{P})} \big\{(a, v) \;\big|\; v \text{ is in } eval'(D, d, q') \text{ with } q' = \mathbb{P}(a)\big\}.$$

2. If M is of the form $\mathsf{Extend}_{\varphi}^{a}(M')$, and given $M'[\sigma] = (A', I')$, then

   $$A = A' \cup \{a\} \quad \text{and} \quad I = \big\{t \cup \{a \to eval(\varphi, t)\} \mid t \in I'\big\},$$

   where $eval(\varphi, t)$ is the evaluation of φ over t, as per [15, Definition 9].
3. If M is of the form $\mathsf{Project}^{P}(M')$, and given $M'[\sigma] = (A', I')$, then

   $$A = A' \cap P \quad \text{and} \quad I = \{t[A] \mid t \in I'\},$$

   where $t[A]$ is the mapping tuple t' s.t. $\mathrm{dom}(t') = A$, $t(a) = t'(a)$ for all $a \in A$.
4. If M is $\mathsf{EqJoin}^{\mathbb{J}}(M', M'')$, and given $(A', I') = M'[\sigma]$ and $(A'', I'') = M''[\sigma]$,

   $A = A' \cup A''$

   $I = \{t_1 \cup t_2 \mid t_1 \in I' \text{ and } t_2 \in I'' \text{ such that } t_1(a_1) = t_2(a_2) \text{ for all } (a_1, a_2) \in \mathbb{J}\}$.
5. If M is $\mathsf{Union}(M', M'')$, and given $(A', I') = M'[\sigma]$ and $(A'', I'') = M''[\sigma]$,

   $$A = A' \cup A'' \quad \text{and} \quad I = I' \cup I''.$$

4 Satisfiability

Our pruning approach is based on a notion of satisfiability of triple patterns with respect to RDF data obtained via mappings. This section provides the relevant formal results, for which we begin by defining this notion of satisfiability.

Definition 10. Let M be a TrMap-expression. A triple pattern tp is **satisfiable over** M if there exists a source assignment σ that is valid input for M such that $[\![tp]\!]_G \neq \emptyset$ where G is the RDF graph resulting from the mapping relation $M[\sigma]$.

The following result[4] is the first building block of our pruning approach. It shows that, when evaluating a SPARQL graph pattern over an RDF graph created by applying an RML-specific mapping expression, the correct query result may be produced without explicitly considering every TrMap-expression of the given mapping expression. In particular, it is possible to ignore every TrMap-expression over which none of the triple patterns of the graph pattern is satisfiable.

[4] The proofs of all formal results in this paper are provided in an extended version [16].

Proposition 1. Let P be a graph pattern and let M and M' be RML-specific mapping expressions such that $\text{TrMaps}(M') \subseteq \text{TrMaps}(M)$ and, for every TrMap-expression $M_{\mathsf{TM}} \in \big(\text{TrMaps}(M) \setminus \text{TrMaps}(M')\big)$, *none* of the triple patterns in P is satisfiable over M_{TM}. Then, for every source assignment σ that is valid input for M, it holds that $[\![P]\!]_G = [\![P]\!]_{G'}$, where G and G' are the RDF graphs resulting from the mapping relations $M[\sigma]$ and $M'[\sigma]$, respectively.

Based on Proposition 1, we may prune TrMap-expressions from a given RML-specific mapping expression (when used in the context of evaluating graph patterns over the RDF graph produced by applying the mapping expression). To do so, however, we need to be able to determine which triple patterns are satisfiable over which TrMap-expressions, which leads us to the following decision problem.

Problem: SATISFIABILITY(TPOVERTRMAP)
Input: a triple pattern tp and a TrMap-expression M
Question: Is tp satisfiable over M?

The following result shows that we cannot answer this question in general.

Proposition 2. SATISFIABILITY(TPOVERTRMAP) is undecidable.

5 Incompatibility

While Proposition 2 shows that there is no general way to determine, for any given triple pattern tp and any given TrMap-expression M, whether tp is satisfiable over M or not, there is a class of cases for which we can at least determine whether a triple pattern is guaranteed to be *not* satisfiable over a TrMap-expression (which is indeed what we need to know to use Proposition 1 for pruning).

In this section, we introduce a syntactic property to identify such cases and prove its correctness. We call this property *incompatibility*; more specifically, we say that a triple pattern tp is *incompatible with* a TrMap-expression M if the pair of tp and M has the property that we aim to introduce. To define the property formally, we need two auxiliary concepts. The first of them are regular expressions that we construct from torb-extend expressions as follows.

Definition 11. Let φ be a torb-extend expression (as per Definition 4). The **matching pattern of** φ, denoted by $\text{regex}(\varphi)$, is a string to be used as a regular expression[5] and is defined recursively as follows:

1. If φ is an RDF literal (lex, dt), then the string $\text{regex}(\varphi)$ is a version of lex in which every character that has a special meaning in regular expressions is escaped with a backslash.
2. If φ is an attribute, then $\text{regex}(\varphi)$ is the string ".+".
3. If φ is of the form $\texttt{concat}(\varphi_1, \ldots, \varphi_n)$, then $\text{regex}(\varphi)$ is the string obtained by concatenating the strings $\text{regex}(\varphi_1)$, ..., and $\text{regex}(\varphi_n)$, in this order.

[5] We assume the extended regular expression notation of the POSIX standard: `https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/V1_chap09.html`

The second auxiliary concept that we need is a notion of incompatibility between IRIs and extend expressions, which is defined as follows.

Definition 12. An IRI u is **incompatible with** an extend expression φ [15] if *any* of the following three properties holds:

1. φ is any RDF term but not the IRI u.
2. φ is either of the form $\texttt{toLiteral}(\varphi', dt)$ or of the form $\texttt{toBNode}^{S2B}(\varphi')$.
3. φ is of the form $\texttt{toIRI}(\varphi', u_{\mathsf{base}})$ with φ' being a torb-extend expression and u_{base} an IRI such that i) u does not match the regular expression $\text{regex}(\varphi')$ and ii) u does not match the regular expression formed by prefixing $\text{regex}(\varphi')$ with a version of u_{base} in which every character that has a special meaning in regular expressions is escaped with a backslash.

Example 2. Let φ_{ex} be the extend expression $\texttt{toIRI}(\texttt{concat}(\ell, a_1), u_{\mathsf{base}})$ as used by the outermost Extend operator of the TrMap-expression in Example 1. Since the matching pattern of its torb-extend expression, $\texttt{concat}(\ell, a_1)$, is the string `"http:\/\/example.com\/route\/.+"`, the IRI `http://transit.api/route/43` is incompatible with φ_{ex}, whereas `http://example.com/route/43` is not incompatible.

Now we are ready to define our main incompatibility property.

Definition 13. A triple pattern (s, p, o) is **incompatible with** a TrMap-expression M (with some u_{base} and $S2B$) if either of the following conditions holds.

1. M is of the first of the two forms in Definition 5, i.e., M is of the form:

$$\mathsf{Extend}^{a_{\mathrm{o}}}_{\varphi_{\mathrm{o}}}(\mathsf{Extend}^{a_{\mathrm{p}}}_{\varphi_{\mathrm{p}}}(\mathsf{Extend}^{a_{\mathrm{s}}}_{\varphi_{\mathrm{s}}}(\mathsf{Extract}^{(type,q,\mathbb{P})}_{sr})))$$

   and *any* of the following conditions holds:
   (a) s is an IRI that is incompatible with φ_{s}.
   (b) p is an IRI that is incompatible with φ_{p}.
   (c) o is an IRI that is incompatible with φ_{o}.
   (d) o is a literal and φ_{o} is of the form $\texttt{toIRI}(\varphi', u_{\mathsf{base}})$.
   (e) o is a literal and φ_{o} is of the form $\texttt{toBNode}^{S2B}(\varphi')$.
   (f) o is a literal (lex, dt) and φ_{o} is of the form $\texttt{toLiteral}(\varphi', dt')$ such that lex does not match the regular expression $\text{regex}(\varphi')$ or $dt \neq dt'$.
2. M is of the second of the two forms in Definition 5, i.e., M is of the form:

$$\mathsf{Extend}^{a_{\mathrm{o}}}_{\varphi'_{\mathrm{o}}}(\mathsf{EqJoin}^{\mathbb{J}}(\mathsf{Extend}^{a_{\mathrm{p}}}_{\varphi_{\mathrm{p}}}(\mathsf{Extend}^{a_{\mathrm{s}}}_{\varphi_{\mathrm{s}}}(\mathsf{Extract}^{(type,q,\mathbb{P})}_{sr})), \mathsf{Extract}^{(type',q',\mathbb{P}')}_{sr'}))$$

   and *any* of the following conditions holds:
   (a) s is an IRI that is incompatible with φ_{s}.
   (b) p is an IRI that is incompatible with φ_{p}.
   (c) o is an IRI that is incompatible with φ'_{o}.
   (d) o is a literal.

Example 3. Both triple patterns of the SPARQL query in Listing 2 are incompatible with the TrMap-expression in Example 1. For the first triple pattern, the incompatibility is due to condition 1(c) of Definition 13 (see also Example 2), and for the second triple pattern, it is due to conditions 1(b) and 1(d).

The following result shows that incompatibility implies unsatisfiability, which makes it another main building block of our pruning approach because it guarantees that we can rely on incompatibility checks to make correct pruning decisions.

Proposition 3. Let M be a TrMap-expression. For every triple pattern tp that is incompatible with M, it holds that tp is *not* satisfiable over M.

Based on Proposition 3 we can replace the satisfiability-based condition in Proposition 1 by an incompatibility-based condition. That is, combining both propositions gives us the following result, which shows the correctness of the pruning algorithm that we shall develop from it in the next section.

Corollary 1. Let P be a graph pattern and let M and M' be RML-specific mapping expressions such that $\mathrm{TrMaps}(M') \subseteq \mathrm{TrMaps}(M)$ and, for every TrMap-expression $M_{\mathsf{TM}} \in \big(\mathrm{TrMaps}(M) \setminus \mathrm{TrMaps}(M')\big)$, it holds that every triple pattern in P is incompatible with M_{TM}. Then, for every source assignment σ that is valid input for M, it holds that $[\![P]\!]_G = [\![P]\!]_{G'}$, where G and G' are the RDF graphs resulting from the mapping relations $M[\sigma]$ and $M'[\sigma]$, respectively.

6 Pruning Algorithm

To capture our pruning approach in an algorithmic form, we convert Corollary 1 into Algorithm 1. The input to this algorithm is a SPARQL graph pattern P and an RML-specific mapping expression M, and the output is an RML-specific mapping expression that is constructed to become the M' in Corollary 1.

The algorithm first collects the TrMap-expressions of M that cannot be pruned (lines 1–6). To this end, the algorithm iterates over all TrMap-expressions of M. For each such TrMap-expression M_{TM}, the algorithm checks whether there is a triple pattern in P that is *not* incompatible with M_{TM} (lines 3–4). As soon as the first such triple pattern is found, M_{TM} is collected (line 5) and the algorithm

Algorithm 1: Prunes an RML mapping for a given SPARQL graph pattern.

Input: P - a SPARQL graph pattern, M - an RML-specific mapping expression
Output: an RML-specific mapping expression

1 $R \leftarrow \emptyset$ // will be used to collect the TrMap-expressions that cannot be pruned
2 **foreach** TrMap-expression $M_{\mathsf{TM}} \in \mathrm{TrMaps}(M)$ **do**
3 **foreach** triple pattern tp in P **do**
4 **if** tp is *not* incompatible with M_{TM} **then**
5 $R \leftarrow R \cup \{M_{\mathsf{TM}}\}$
6 **continue** // break out of loop at line 3
7 $M' \leftarrow \mathsf{Project}^{\{a_s,a_p,a_o\}}(M_{\mathsf{TM}})$, where M_{TM} is an arbitrary TrMap-expression of R
8 **foreach** TrMap-expression $M'_{\mathsf{TM}} \in R \setminus \{M_{\mathsf{TM}}\}$ **do**
9 $M' \leftarrow \mathsf{Union}\big(M', \mathsf{Project}^{\{a_s,a_p,a_o\}}(M'_{\mathsf{TM}})\big)$
10 **return** M'

moves on to the next TrMap-expression of M. Hence, every TrMap-expression that has not been collected into R at the end of the for-loop in lines 2–6 is one that every triple pattern of P is incompatible with (and, thus, can be ignored as per Corollary 1). After collecting the TrMap-expressions to be kept, the algorithm reconstructs an RML-specific mapping expression with them (lines 7–9).

7 Evaluation

We now evaluate how pruning of RML mappings affects the execution time of each step of the evaluation pipeline shown in Figure 1. Specifically, we conduct a self-evaluation where, given a SPARQL query and an RML mapping, we first prune the RML mapping for the query and then run the pruned RML mapping on a chosen mapping engine, instead of comparing the performance across multiple mapping engines. Self-evaluation enables us to isolate the performance impact of the pruning. Moreover, by using an existing mapping engine, we show that such engines can benefit from the performance improvements of our approach.

We chose not to compare against virtualization systems for the following reason: In addition to selecting relevant mapping rules, such systems rewrite the given SPARQL query to the query language of the underlying source database system, which in turn uses rewritten and optimized query plans to retrieve relevant data. Such a cascading application of optimizations makes it difficult to isolate the performance impact of the pruning strategy used by these systems.

In the following subsections, we first introduce the chosen benchmark and our rationale for choosing it (Section 7.1). Thereafter, we describe our evaluation setup (Section 7.2), and present and analyze the results (Section 7.3).

7.1 Benchmark

For our evaluation, we use the GTFS-Madrid benchmark [4]. It can generate synthetic source datasets based on real-world data about the Madrid subway

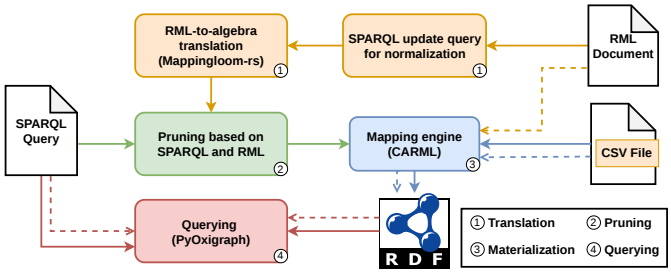

Fig. 1: The evaluation pipeline consists of four steps: i) translation, ii) pruning, iii) materialization, and iv) querying. The solid-lined arrows represent the execution flow that applies our query-specific pruning approach, whereas dashed arrows capture the baseline execution flow without pruning.

network, scaled by a factor, and comes with 18 SPARQL queries based on actual user queries. The benchmark task is to answer these queries over the RDF representation of the generated dataset, referred to as the *full RDF graph* in the rest of this section. The benchmark includes an RML mapping document containing 86 TrMap-expressions to generate the full RDF graph. The provided SPARQL queries are diverse, using SPARQL features such as `FILTER`, `OPTIONAL`, and `GROUP BY`, and contain 3–15 triple patterns each [4]. While the benchmark is mainly used to evaluate virtualization engines [4], it has also been adapted for evaluating materialization engines [1], making it relevant for our approach. For each of the benchmark queries, we prune the benchmark RML mapping to generate an RDF graph that is a subset of the full RDF graph. We configure the GTFS-Madrid benchmark to generate the synthetic dataset at scale factor 10. This small scale factor fits our aim, which is to evaluate the performance impact of our pruning technique rather than the scalability of an end-to-end data-mapping pipeline. Furthermore, to limit the impact of a potential implementation error in the underlying mapping engine when parsing complex input data, we configure the benchmark to generate data in CSV format.

7.2 Setup

Figure 1 illustrates our evaluation pipeline. For each step of the pipeline, we measure the time needed to complete the step. To ensure that these time measurements can be clearly isolated from one another, we have implemented each step of the pipeline via a separate component. First, the translation step normalizes the RML document of the benchmark using SPARQL update queries and, then, translates the normalized RML mappings into the mapping algebra, both as defined in our earlier work [15] (where the normalization step is an adaptation of normalizations by Kontchakov et al. [11] and by Rodríguez-Muro and Rezk [20]). The time required for this step is measured as the *translation time*. The resulting (RML-specific) mapping expression is then pruned by using our approach of Section 6, separately for each of the 18 queries of the benchmark. The time required for this process per query is measured as the *pruning time*. For every query, we then create an RML mapping, containing each of the remaining TrMap-expressions as a separate RML triples map, and serialize it into a file as the input for the next step. In that step, the pruned RML mappings are executed using the CARML[6] mapping engine, and the resulting partial RDF graphs are serialized into files, one per query. The time taken to materialize each such RDF graph is measured as the *materialization time* (per query). Next, for every query, the file with the corresponding RDF graph from the materialization step is loaded into an Oxigraph[7] triple store. Finally, we conduct the actual *querying time* evaluation by running the SPARQL query under consideration on the loaded RDF graph, with a Python wrapper for Oxigraph.[8]

[6] `https://github.com/carml/carml-jar/releases/tag/v1.4.0`
[7] `https://github.com/oxigraph/oxigraph`
[8] `https://pypi.org/project/pyoxigraph/`

Table 1: For each query, the execution time for different steps of the evaluation pipeline is averaged over four runs. The last column reports the querying time when querying the full RDF graph. `TO` means the execution timed out (3600s).

Query	Pruning	# of TrMaps after pruning	# of Triples generated	Materialization	Pruning + Materialization as % of Baseline Materialization	Querying	Querying without pruning
Q1	3.39 ms	7	2,952,240	81,474.83 ms	99.46 %	0.09 ms	0.11 ms
Q2	3.40 ms	9	1,235,330	15,349.36 ms	18.74 %	9.62 ms	11.71 ms
Q3	3.55 ms	10	1,247,950	15,092.22 ms	18.42 %	0.13 ms	0.12 ms
Q4	3.28 ms	14	39,730	2,822.38 ms	3.45 %	4.15 ms	5.64 ms
Q5	4.01 ms	7	3,600	2,044.86 ms	2.50 %	0.08 ms	0.15 ms
Q6	2.88 ms	2	260	1,717.52 ms	2.10 %	0.21 ms	0.35 ms
Q7	4.63 ms	20	1,321,430	17,362.18 ms	21.20 %	TO	TO
Q8	4.29 ms	18	112,970	6,150.15 ms	7.51 %	TO	TO
Q9	4.12 ms	11	1,776,620	68,753.60 ms	83.93 %	0.41 ms	0.62 ms
Q10	3.28 ms	5	80,770	3,889.52 ms	4.75 %	43.59 ms	78.56 ms
Q11	4.48 ms	14	6,370	2,175.37 ms	2.66 %	0.12 ms	0.18 ms
Q12	4.32 ms	13	121,500	4,606.08 ms	5.62 %	72.26 ms	125.79 ms
Q13	3.24 ms	5	45,820	3,007.90 ms	3.67 %	8.37 ms	22.68 ms
Q14	3.43 ms	10	129,660	4,824.48 ms	5.89 %	99.14 ms	165.38 ms
Q15	4.46 ms	86	4,546,610	102,905.99 ms	125.63 %	0.38 ms	0.65 ms
Q16	4.60 ms	11	8,800	2,231.85 ms	2.72 %	0.09 ms	0.17 ms
Q17	3.69 ms	11	62,130	4,390.00 ms	5.36 %	0.12 ms	0.14 ms
Q18	4.46 ms	12	7,060	2,120.87 ms	2.59 %	0.11 ms	0.13 ms

Measurement of translation time, which is query independent (Step 1 in Fig. 1)

Translation
218.22 ms

Baseline measurements for the number of TrMap-expressions in the original RML document, the number of triples generated and the materialization time without pruning nor translation

# of TrMaps before pruning	# of Triples generated	Baseline Materialization w/o pruning
86	4,546,610	81,915.27 ms

To establish a baseline for our evaluation, we measure the time taken with the original RML mapping file of the benchmark to i) execute it to generate the full RDF graph without pruning (*baseline materialization time*), and ii) query the full RDF graph with each of the 18 SPARQL queries (*baseline querying times*). For these measurements, the execution follows the dashed path in Figure 1.

The evaluation is conducted on a machine with an Intel i7 CPU (4.80 GHz) and 16 GB of RAM, running Ubuntu 24.04.3. The whole pipeline is executed five times for each of the 18 SPARQL queries, with the individual steps executed sequentially as shown in Figure 1. The first run is used to warm up the pipeline and its measurements are discarded; the measurements of the remaining four runs are averaged per step per query. Table 1 presents these measurements.

7.3 Results

Pruning Time. We observe that the time used for pruning is negligible: between 3 and 5 ms, which is orders of magnitude shorter than even the shortest materialization time with one of the pruned mappings (1,717.52 ms, for Q6).

Materialization Time. For 12 of the 18 queries, our approach reduces the materialization time after pruning to at most 8% (around 7 secs) of the baseline materialization time (82 secs)! Such significant improvement is due to the small number of remaining TrMap-expressions after pruning. Specifically, the original 86 TrMap-expressions are reduced to at most 20, which significantly lowers the computation effort of the materialization step (ignoring Q15 for the moment).

We also observe that, for Q1 and Q9, even though the pruning step retains fewer TrMap-expressions than for Q7, their materialization times are higher than for Q7, which we explain as follows. These two queries contain the triple pattern `"?shape gtfs:shapePoint ?shapePoint."` Triples that match this triple pattern are produced by a TrMap-expression that is of the second form in Definition 5 (i.e., using a join) and its two source references are the same. Hence, evaluating this expression results in a self-join operation, which is known to cause higher materialization time [22]. To make matters worse, the source reference refers to the largest file of the input dataset, `SHAPES.csv`, which contains about 585K records at the considered scale factor. Despite such limitations, our approach still achieves lower materialization times for Q1 and Q9 than the baseline.

Materializing the RDF graph for Q15 took longer than the baseline materialization time (about 126%). Q15 contains the triple pattern `?stop ?p ?str`, which means that no TrMap-expressions are pruned. Due to the normalization step, each RML triples map that contains multiple predicate-object maps is normalized into several separate RML triples maps, each with one predicate and one object map [15]. As a result, the re-generated RML mapping, using the retained TrMap-expressions, contains more RML triples map definitions than the original RML mapping. However, they are semantically the same (i.e., evaluating them independently produces the same RDF graph). A mapping engine that is not aware of such semantic equivalence of two differently-written RML mappings, may take longer for one than for the other. We can infer from our measurements that CARML is not aware of such semantic equivalence.

Querying Time. Our measurements show that querying the RDF graphs generated via the pruned RML mappings is often faster than querying the full RDF graph. The speed up is especially noticeable for Q10, Q12, and Q14, where the querying time is smaller by around 44%, 42%, and 40%, respectively.

For Q7 and Q8, with and without pruning, the query execution timed out at 3600 seconds. Q7 and Q8 contain the most triple patterns, 15 and 14 respectively, while also using the `OPTIONAL` feature of SPARQL and, for Q7, even `DISTINCT`. Although Q10 also uses `DISTINCT`, its querying time is substantially smaller because the corresponding generated RDF graph is much smaller (around 80K triples, compared 1.3M triples for Q7). For Q8, the timeout occurred due to the combination of having the second most triple patterns (14) and the largest chained star-shaped group, both of which increases querying complexity.

8 Related Work

As mentioned in the introduction, there are two types of approaches to provide access to RDF views of non-RDF data: *materialization* and *virtualization*.

Materialization approaches rely on a mapping language (e.g., RML [6] or R2RML [21]), define a direct mapping using a meta-model [2], or use manually written scripts to transform non-RDF data to generate the RDF graph. Thus, materialization systems [1,2,9,22] can process diverse types of data sources and

formats into RDF graph, provided that the types of data sources and formats are supported by the underlying mapping language, meta-model or implementation. However, none of these systems considers a user's query while generating an RDF graph, which can be inefficient in dynamic environments where a query can be answered by a small subset of the generated RDF graph.

Virtualization approaches provide SPARQL query functionality over a virtual RDF view of the underlying non-RDF data. To this end, these approaches rewrite any given SPARQL query into an equivalent query supported by the underlying data source. The rewritten query is then executed against the data source, and the fetched results are transformed back into SPARQL query results. Such query rewriting and result transformation is usually guided by the usage of a mapping language such as R2RML [21]. Thus, virtualization systems [3,19,5] are capable of answering SPARQL queries over non-RDF data through virtual RDF views. However, query rewriting also limits such systems to using data sources that support a query language into which a SPARQL query can be translated.

While virtualization is query-aware, it is less flexible than materialization in supported data sources, whereas materialization can handle diverse sources but generates unnecessary triples. Our approach bridges this gap by preemptively pruning mappings, to retain mappings relevant to answering the user's query.

9 Concluding Remarks and Future Work

The approach presented in this work opens up new possibilities for research on using non-RDF data sources when evaluating SPARQL queries over a federation of data sources. More concretely, there are two scenarios in which our pruning approach is beneficial. First, if a non-RDF data source is wrapped as a SPARQL endpoint to provide access to an RDF view of the underlying data, generating an up-to-date (and pruned) version of this view at query time is particularly relevant if the source data changes frequently. Second, a query federation engine that can query non-RDF data sources through materialization at runtime [8] will benefit from the pruning of irrelevant mappings. As an example of the latter case, consider a JSON-based REST API as such a data source: if a part of the SPARQL query over the whole federation is meant to be matched in the RDF views of the data from requests to that API, the pruning approach will speed up the process of both materializing the data retrieved from the API and evaluating the relevant sub-pattern of the SPARQL query over this materialized RDF view.

While our pruning approach works at the triple pattern level, a natural next step is to extend it to whole basic graph patterns: Performing satisfiability checks that consider joins between multiple triple patterns may result in pruning even more TrMap-expressions and, thus, reduce the materialization time even further.

Acknowledgments. This work was supported by the Knut and Alice Wallenberg Foundation (KAW 2023.0111), by the Swedish Research Council (project reg. no. 2025-06246), and by the imec.icon project PACSOI (HBC.2023.0752), which was co-financed by imec and VLAIO and brings together the following partners: FAQIR Foundation, FAQIR Institute, MoveUP, Byteflies, AContrario, and Ghent University – IDLab.

Disclosure of Interests. The authors have no competing interests to declare.

Supplemental Material Statement: The source code and artifacts for the evaluation, including an implementation of the pruning algorithm, are provided in a GitHub repository.[9] Full proofs of Propositions 1–3 are in [16].

Declaration of use of Generative AI: The authors have not employed any Generative AI tools for the presented work.

References

1. Arenas-Guerrero, J., Chaves-Fraga, D., Toledo, J., Pérez, M.S., Corcho, O.: Morph-KGC: Scalable Knowledge Graph Materialization with Mapping Partitions. Semantic Web **15**(1), 1–20 (2022). https://doi.org/10.3233/sw-223135
2. Asprino, L., Daga, E., Gangemi, A., Mulholland, P.: Knowledge graph construction with a façade: a unified method to access heterogeneous data sources on the web. ACM Transactions on Internet Technology **23**(1), 1–31 (2023)
3. Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering SPARQL Queries over Relational Databases. Semantic Web Journal **8**(3), 471–487 (2017). https://doi.org/10.3233/SW-160217
4. Chaves-Fraga, D., Priyatna, F., Cimmino, A., Toledo, J., Ruckhaus, E., Corcho, O.: GTFS-Madrid-Bench: A Benchmark for Virtual Knowledge Graph Access in the Transport Domain. Journal of Web Semantics **65** (2020). https://doi.org/10.1016/j.websem.2020.100596
5. Chaves-Fraga, D., Ruckhaus, E., Priyatna, F., Vidal, M.E., Corcho, O.: Enhancing Virtual Ontology Based Access over Tabular Data with Morph-CSV. Semantic Web **12**(6), 869–902 (2021). https://doi.org/10.3233/sw-210432
6. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In: Proceedings of the 7th Workshop on Linked Data on the Web (LDOW). CEUR Workshop Proceedings, vol. 1184 (2014), http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf
7. Freund, M., Schmid, S., Dorsch, R., Harth, A.: FlexRML: A Flexible and Memory Efficient Knowledge Graph Materializer. In: Proceedings of the 21st Extended Semantic Web Conference (ESWC). pp. 40–56. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-60635-9_3
8. Hartig, O., Westman, J.: Querying Federations of SPARQL Endpoints and JSON-based Web APIs with HeFQUIN. In: Proceedings of Satellite Events of the 23rd European Semantic Web Conference (ESWC). Lecture Notes in Computer Science, Springer (2026)
9. Iglesias, E., Jozashoori, S., Vidal, M.E.: Scaling Up Knowledge Graph Creation to Large and Heterogeneous Data Sources. Journal of Web Semantics **75** (2023). https://doi.org/10.1016/j.websem.2022.100755
10. Iglesias-Molina, A., Assche, D.V., Arenas-Guerrero, J., Meester, B.D., Debruyne, C., Jozashoori, S., Maria, P., Michel, F., Chaves-Fraga, D., Dimou, A.: The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF. In: Proceedings of the 22nd International Semantic Web Conference (ISWC). Lecture Notes in Computer Science, vol. 14266, pp. 152–175. Springer (2023). https://doi.org/10.1007/978-3-031-47243-5_9

[9] https://github.com/s-minoo/satisfiability-experiment

11. Kontchakov, R., Rezk, M., Rodríguez-Muro, M., Xiao, G., Zakharyaschev, M.: Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime. In: The Semantic Web – ISWC 2014. pp. 552–567. Springer International Publishing (2014)
12. Lefrançois, M., Zimmermann, A., Bakerally, N.: A SPARQL Extension for Generating RDF from Heterogeneous Formats. In: Proceedings of the 14th Extended Semantic Web Conference (ESWC). pp. 35–50. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-58068-5_3
13. Lopes, N., Bischof, S., Decker, S., Polleres, A.: On the Semantics of Heterogeneous Querying of Relational, XML and RDF Data with XSPARQL. In: Proceedings of the 15th Portuguese Conference on Artificial Intelligence (EPIA). pp. 10–13 (2011)
14. Min Oo, S., Haesendonck, G., De Meester, B., Dimou, A.: RMLStreamer-SISO: An RDF Stream Generator from Streaming Heterogeneous Data. In: Proceedings of the 21st International Semantic Web Conference (ISWC). pp. 697–713. Springer (2022). https://doi.org/10.1007/978-3-031-19433-7_40
15. Min Oo, S., Hartig, O.: An Algebraic Foundation for Knowledge Graph Construction. In: The Semantic Web - 22nd European Semantic Web Conference, ESWC 2025, Portoroz, Slovenia, June 1-5, 2025, Proceedings, Part I. pp. 3–22. Lecture Notes in Computer Science, Springer (2025). https://doi.org/10.1007/978-3-031-94575-5_1
16. Min Oo, S., Hartig, O.: Query-Specific Pruning of RML Mappings (Extended Version). CoRR **arXiv/2603.26269** (2026). https://doi.org/10.48550/ARXIV.2603.26269
17. Min Oo, S., Verbeken, T., De Meester, B.: RMLWeaver-JS : an algebraic mapping engine in the KGCW challenge 2024. In: Proceedings of the 5th International Workshop on Knowledge Graph (KGCW) (2024)
18. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems (TODS) **34**(3), 1–45 (2009)
19. Priyatna, F., Corcho, O., Sequeda, J.: Formalisation and Experiences of R2RML-based SPARQL to SQL Query Translation using Morph. In: Proceedings of the 23rd International Conference on Worldwide Web (WWW). pp. 479–490. Association for Computing Machinery (2014). https://doi.org/10.1145/2566486.2567981
20. Rodríguez-Muro, M., Rezk, M.: Efficient SPARQL-to-SQL with R2RML mappings. Journal of Web Semantics **33**, 141–169 (2015). https://doi.org/10.1016/j.websem.2015.03.001
21. Sundara, S., Das, S., Cyganiak, R.: R2RML: RDB to RDF Mapping Language. W3C Recommendation (Sep 2012), https://www.w3.org/TR/r2rml/
22. de Vleeschauwer, E., Maria, P., De Meester, B., Colpaert, P.: RML-view-to-CSV: A Proof-of-Concept Implementation for RML Logical Views. In: Proceedings of the 5^{th} International Workshop on Knowledge Graph Construction (KGCW) (2024)