A Generic Mapping-based Query Translation from SPARQL to Various Target Database Query Languages

Franck Michel, Catherine Faron-Zucker and Johan Montagnat
I3S, UMR 7271, University Nice Sophia Antipolis, CNRS, Sophia Antipolis, France

Keywords: Linked Data, Query rewriting, SPARQL, RDF, NoSQL, xR2RML.

Abstract: Fostering the development of SPARQL interfaces to heterogeneous databases is a key to efficiently expose legacy data as RDF on the Web. To deal with the variety of modern database formats and query languages, this paper describes a two-step approach to translate a SPARQL query into an equivalent target database query. First, given an xR2RML mapping describing how native database entities can be mapped to RDF, a SPARQL query is translated into a pivot abstract query language independent of the database. In a second step, the pivot query is translated into the target database query language, considering the specific database capabilities. The paper focuses on the first step of the query translation, from SPARQL to a pivot query that takes into account join constraints and SPARQL filters, and embeds conditions entailed by matching SPARQL graph patterns with relevant mappings. It discusses the query optimisations that can be implemented at this level, and briefly describes an application to the case of MongoDB, a NoSQL document store.

1 INTRODUCTION

The exposure of legacy data as RDF is an increasingly hot topic as new data integration challenges emerge. Notably, the Web-scale data integration progressively gives birth to the Web of Data thanks to the open publication, in RDF, of data sets on the Web. Two approaches generally apply: legacy data can all be translated into a materialized RDF graph or data can be accessed on-the-fly as a virtual RDF graph using the SPARQL query language. Although the materialization is of interest in some contexts it is hardly a one-fits-all solution in practice, due to the size of the generated graphs. Dynamic access on the other hand scales better and guarantees data freshness.

In the last decade, translation methods for relational databases (RDB) have matured, spurred by the publication of the R2RML mapping language (Das et al., 2012). Several methods were proposed to achieve SPARQL access to relational data, either in the context of RDB-backed RDF stores (Chebotko et al., 2009; Sequeda and Miranker, 2013; Elliott et al., 2009) or using arbitrary relational schemas (Bizer and Cyganiak, 2006; Unbehauen et al., 2013a; Priyatna et al., 2014; Rodríguez-Muro and Rezk, 2015). The mapping of XML data to RDF has been addressed extensively by works such as (Bischof et al., 2012) and (Bikakis et al., 2015), among which the latter proposes a translation from SPARQL to XQuery. Furthermore, extensions of R2RML were proposed such as RML (Dimou et al., 2014) to map heterogeneous data formats (e.g. CSV/TSV, XML or JSON), and xR2RML (Michel et al., 2015a) to enable the mapping of an extensible scope of databases to RDF.

At the same time, new actors, the NoSQL databases, have gained a remarkable success. Initially confined to serve as the core system of Big Data applications for which they were designed, they are being increasingly adopted as general-purpose databases, fostered by their open source licenses and the lightweight, easy-to-start packaging of some of them. Today, this overwhelming success makes them a natural candidate for RDF-based data integration systems, and in particular to feed the Web of Linked Data.

In this regard, it shall be necessary to develop SPARQL access methods for heterogeneous databases, that shall vary greatly depending on the target database query languages: for instance RDBs support joins, nested queries and string manipulations, but this is hardly the case of some NoSQL document stores like MongoDB or CouchDB. Thus, to avoid defining yet another SPARQL translation method for each and every query language, this paper introduces a two-step approach. First, given a set of mappings

Michel, F., Faron-Zucker, C. and Montagnat, J.
A Generic Mapping-based Query Translation from SPARQL to Various Target Database Query Languages.
In *Proceedings of the 12th International Conference on Web Information Systems and Technologies (WEBIST 2016) - Volume 2*, pages 147-158
ISBN: 978-989-758-186-1

of the target database to RDF, a SPARQL query is translated into an pivot abstract query by matching SPARQL graph patterns with relevant mappings. This step can be made generic if the mapping language used is generic enough to apply to an extensible set of databases. In a second step, the abstract query is translated into the target database query language, taking into account the specific database capabilities.

Our focus, in this paper, is on the first step. Leveraging previous works on R2RML-based SPARQL-to-SQL methods, we define a pivot abstract query language and a method to translate a SPARQL query into an abstract query, utilizing xR2RML to describe the mapping of a target database to RDF. The method determines a reduced set of mappings matching each SPARQL graph pattern, and takes into account join constraints implied by shared variables and cross-references denoted in the mappings, and SPARQL filters. Lastly, common query optimization techniques are applied to the abstract query in order to alleviate the work required in the second step.

This paper is organized as follows. Section 2 first reviews related works, in particular R2RML-based SPARQL-to-SQL methods that we build upon. Then section 3 describes the xR2RML mapping language, and section 4 presents the main contribution of this paper: the translation of a SPARQL query into an abstract query under xR2RML mappings. Section 5 presents an application of our method to the case of the MongoDB database. Finally we underline current limitations and highlight perspectives in section 6.

2 RELATED WORKS

Various methods have been defined to translate SPARQL queries into another query language, that are generally tailored to the expressiveness of the target query language. For instance, SPARQL-to-SQL translation methods harness the ability of SQL to support joins, unions, nested queries and various string manipulation functions. Typically, a conjunction of two basic graph patterns (BGP) results in the inner join of their respective translations; their union results in an SQL UNION ALL clause; the SPARQL OPTIONAL clause between two BGPs results in a left outer join, and a SPARQL FILTER results in an encapsulating SQL SELECT WHERE clause.

Chebotko's algorithm (Chebotko et al., 2009) focuses on the SPARQL-to-SQL query translation in the context of RDB-based triple stores. Priyatna et al. (Priyatna et al., 2014) have extended it to support custom R2RML mappings; they address the problem of eliminating null answers by adding not null conditions for SPARQL variables. Two limitations can be underlined though: (i) R2RML triples maps must have constant predicate maps, i.e. the predicates of the generated RDF triples cannot be built from database values; and (ii) triple patterns are considered and translated independently of each other, even when they share SPARQL variables. Unnecessary complexity is thus entailed in the SQL query with nested queries whose solutions are ruled out only during the final join step.

Unbehauen et al. (Unbehauen et al., 2013a) define the concept of compatibility between the RDF terms of a triple pattern and R2RML term maps. This more general approach effectively manages variable predicate maps, which clears the first aforementioned limitation. Furthermore, they reduce the number of candidate triples maps for each triple pattern by pre-checking join constraints implied by shared variables. This clears the second aforementioned limitation. Yet, two limitations can be noticed: (i) R2RML referencing object maps are not considered, therefore joins implied by shared variables are dealt with but joins declared in the mapping graph are ignored. (ii) The rewriting maps each term map to a set of columns, called *column group*, that enables filtering, join and data type compatibility checks. This relies on SQL capabilities (CASE, CAST, string concatenation, etc.), making it hardly applicable out of the scope of SQL-based systems.

Similarly, approaches have been proposed to deal with XML databases. SPARQL2XQuery (Bikakis et al., 2015) relies on the ability of XQuery to support joins, nested queries and complex filtering. For instance a SPARQL FILTER is translated into an encapsulating For-Let-Where XQuery clause.

The rich expressiveness of SQL and XQuery makes it possible to translate a SPARQL query into a single, possibly deeply nested, target query, whose semantics is strictly equivalent to that of the SPARQL query. In the general case however, the target query language may not support joins, unions and/or subqueries. NoSQL databases typically make a trade-off between query language expressiveness and scalability. For instance, MongoDB does not support joins, and only supports nested queries under strong restrictions. To tackle this issue, we propose to translate the SPARQL query into a pivot abstract query independent of the target database, under xR2RML mappings. Our method relies on and extends the aforementioned SPARQL-to-SQL approaches. It supports variable predicate maps, it deals both with implicit joins (shared variable) and explicit joins (mapping-defined). It also considers query execution performance issues by pushing as much as possible of the

```
{ "id": 105632, "firstname": "John",
  "emails": ["john@foo.com","john@example.org"],
  "contacts": ["chris@example.org", "alice@foo.com"] }

{ "id": 327563, "firstname": "Alice",
  "emails": ["alice@foo.com"],
  "contacts": ["john@foo.com"] }
```

Listing 1: MongoDB collection “people” containing two documents.

```
<#Mbox>
  xrr:logicalSource [ xrr:query "db.people.find({'emails':{$ne: null}})" ];
  rr:subjectMap [ rr:template "http://example.org/member/{$.id}" ];
  rr:predicateObjectMap [
    rr:predicate foaf:mbox;
    rr:objectMap [ xrr:reference "$.emails.*" ] ].

<#Knows>
  xrr:logicalSource [ xrr:query "db.people.find({'contacts':{$size: {$gte:1}}})" ];
  rr:subjectMap [ rr:template "http://example.org/member/{$.id}" ];
  rr:predicateObjectMap [
    rr:predicate foaf:knows;
    rr:objectMap [
      rr:parentTriplesMap <#Mbox>;
      rr:joinCondition [ rr:child "$.contacts.*"; rr:parent "$.emails.*" ] ] ].
```

Listing 2: xR2RML example mapping graph.

original query to the native query engines, thus making sub-queries as selective as possible.

3 THE xR2RML MAPPING LANGUAGE

The xR2RML mapping language (Michel et al., 2015a) is designed to map an extensible scope of relational and non-relational databases to RDF. It is independent of any query language or data model. It is backward compatible with R2RML and it relies on RML for the handling of various data formats. It can translate data with mixed embedded formats and generate RDF lists and containers. Below we shortly describe the main xR2RML features and propose a running example.

xR2RML Language Description. An xR2RML mapping defines a logical source (property `xrr:logicalSource`) as the result of executing a query (property `xrr:query`) against an input database. Data from the logical source is mapped to RDF triples using *triples maps*. A triples map consists of several term maps: they are functions that extract values from a query result set, and translate them into RDF terms. A subject map generates the subject of RDF triples, and multiple predicate-object maps produce the predicate and object terms. Optionally, a graph map is used to name a target graph. Listing 2 depicts two triples maps `<#Mbox>` and `<#Knows>`, each consisting of a subject map, a predicate map and an object map.

Term maps extract data from query results by evaluating xR2RML data element references, hereafter named *xR2RML references*. The syntax of xR2RML references depends on the target database *e.g.* a column name in case of a relational database, an XPath expression in case of a native XML database, or a JSONPath[1] expression in case of JSON documents like in MongoDB. xR2RML references are used with properties `xrr:reference` and `rr:template`. The value of a `xrr:reference` property is a single xR2RML reference, whereas the value of a `rr:template` property is a template string possibly involving several references. Properties `xrr:reference` and `rr:template` may also accept *mixed-syntax path* expressions that are useful to deal with content of mixed formats: for instance, a JSON value embedded in the cells of a relational table can be addressed by specifying the column name followed by a JSONPath expression.

Running Example. We define a running example that we shall use throughout this paper. The reader interested in more detailed examples of xR2RML may look at (Michel et al., 2015a; Callou et al., 2015). We consider a MongoDB database with a collection `people` depicted in Listing 1: each JSON document provides the identifier, email addresses and contacts of a person; contacts are given by their email ad-

[1] http://goessner.net/articles/JsonPath/

```
<AbstractQuery> ::= <AtomicQuery> | <Query> | <Query> FILTER <SPARQL filter>
<Query>         ::= <AbstractQuery> INNER JOIN <AbstractQuery> ON {v1,... vn} |
                    <AbstractQuery> AS child INNER JOIN <AbstractQuery> AS parent
                                    ON child/<Ref> = parent/<Ref> |
                    <AbstractQuery> LEFT OUTER JOIN <AbstractQuery> ON {v1,... vn}|
                    <AbstractQuery> UNION <AbstractQuery>
<AtomicQuery>   ::= {From, Project, Where}
```

Listing 3: Grammar of the Abstract Pivot Query Language.

dresses. Listing 2 defines two xR2RML triples maps. The logical source of triples map <#Mbox>, respectively <#Knows>, is a MongoDB query that retrieves documents having a non-null emails field, respectively a contacts array field with at least one element. Both subject maps use a template to build IRI terms by concatenating http://example.org/member/ with the value of JSON field id. Applied to the first document in Listing 1, the triples maps generate three RDF triples:

```
<http://example.org/member/105632>
  foaf:mbox "john@foo.com";
  foaf:mbox "john@example.org";
  foaf:knows
    <http://example.org/member/327563>.
```

When the evaluation of an xR2RML reference produces several RDF terms, the xR2RML processor creates one triple for each term. Alternatively, it can group them in an RDF list (rdf:List) or collection (rdf:Seq, rdf:Bag and rdf:Alt). This is achieved using specific values of the rr:termType property. Besides, property xrr:nestedTermMap is a means to create nested lists and collections, and to qualify terms of a list or collection with a language tag or data type.

Like R2RML, xR2RML can model cross-references: a *referencing object maps* uses subject values produced by the subject map of another triples map (the parent) as objects. Properties rr:child and rr:parent specify the join condition between documents of the current triples map (the child), and the parent triples map. This is illustrated in triples map <#Knows>, that joins email addresses from the emails and contacts fields.

4 REWRITING A SPARQL QUERY INTO AN ABSTRACT QUERY

4.1 Abstract Query Language

Our pivot abstract query language complies with the grammar defined in Listing 3. Operators INNER JOIN ON, LEFT OUTER JOIN ON and UNION follow the semantics of SQL operators of the same name, with the difference that the semantics of UNION is that of the SQL UNION ALL, i.e. it keeps duplicate entries. They are entailed by the dependencies between graph patterns of the SPARQL query. The first INNER JOIN notation is entailed by join constraints implied by shared variables. The second INNER JOIN notation, including the AS child, AS parent and ON child/<Ref> = parent/<Ref> notations, is entailed by join constraints expressed in xR2RML mappings using referencing object maps. The computation of these operators shall be delegated to the target database if it supports them (i.e. if the target query language has equivalent operators like in the case of a relational database), or to the query processing engine otherwise (case of MongoDB).

The translation of a SPARQL query into an abstract query consists of three steps:

1. A SPARQL graph pattern is decomposed into an abstract expression exhibiting only operators from the abstract query language and SPARQL triple patterns: see function $trans_m$ in section 4.2;
2. The xR2RML triples maps that are likely to generate RDF triples matching each triple pattern are identified: see function $bind_m$ in section 4.3; and
3. Each triple pattern is translated into one or several atomic abstract queries (<AtomicQuery>), under the set of xR2RML triples maps identified in step 2. Each atomic query is made as selective as possible by pushing relevant SPARQL filter conditions: see function $transTP_m$ in section 4.4.

4.2 Translation of a SPARQL Graph Pattern

Function $trans_m$ (Definition 1) translates a well-designed SPARQL graph pattern (Pérez et al., 2009) into an abstract query, while making no assumption with respect to the target database query capabilities. It relies on function $transTP_m$ (section 4.4) to translate each triple pattern, and extends the translation algorithms defined in (Chebotko et al., 2009), (Unbehauen et al., 2013a) and (Priyatna et al., 2014). In particular we propose a generalized management of SPARQL filters: the goal is to push down SPARQL

Definition 1. ***Translation of a SPARQL query into an abstract query under xR2RML mappings.*** *Let m be an xR2RML mapping graph consisting of a set of xR2RML triples maps. Let gp be a well-designed SPARQL graph pattern. $trans_m(gp)$ is the translation, under m, of gp into an abstract query.* $\boldsymbol{trans}_m$ *is defined as follows:*

- $\boldsymbol{trans}_m(gp) = \boldsymbol{trans}_m(gp, true)$
- *if gp consists of a single triple pattern tp,* $\boldsymbol{trans}_m(gp, f) = \boldsymbol{transTP}_m(tp, sparqlCond(tp, f))$
- *if gp is (P FILTER f'),* $\boldsymbol{trans}_m(gp, f) = \boldsymbol{trans}_m(P, f \,\&\&\, f')$ ***FILTER*** $sparqlCond(P, f \,\&\&\, f')$
- *if gp is (P1 AND P2),* $\boldsymbol{trans}_m(gp, f) = \boldsymbol{trans}_m(P1, f)$ ***INNER JOIN*** $\boldsymbol{trans}_m(P2, f)$ ***ON*** $var(P1) \cap var(P2)$
- *if gp is (P1 OPTIONAL P2),* $\boldsymbol{trans}_m(gp, f) =$
  $\boldsymbol{trans}_m(P1, f)$ ***LEFT OUTER JOIN*** $transF_m(P2, f)$ ***ON*** $var(P1) \cap var(P2)$
- *if gp is (P1 UNION P2),* $\boldsymbol{trans}_m(gp) =$
  $\boldsymbol{trans}_m(P1, f)$ ***LEFT OUTER JOIN*** $\boldsymbol{trans}_m(P2, f)$ ***ON*** $var(P1) \cap var(P2)$
  UNION
  $\boldsymbol{trans}_m(P2, f)$ ***LEFT OUTER JOIN*** $\boldsymbol{trans}_m(P1, f)$ ***ON*** $var(P1) \cap var(P2)$

filters into the translation of each triple pattern, in order to make inner queries more selective and limit the size of intermediate results. A SPARQL filter *f* can be considered as a conjunction of n conditions ($n \geq 1$): C_1 && ... C_n. Function *sparqlCond*, further detailed in (Michel et al., 2015b), discriminates between conditions with regards to two criteria:

(i) A condition wherein all variables show in a single triple pattern *tp* of the SPARQL query is pushed into the translation of *tp* by function $transTP_m$. This ensures that filters are applied at the earliest stage, as opposed to the encapsulating SELECT WHERE strategy in SPARQL-to-SQL translations.

(ii) For a condition wherein at least one variable is shared by several triple patterns, a FILTER operator is created to represent the join criteria.

Note that a condition may match both criteria.

Running Example. We illustrate this process with the running example introduced in section 3 and the SPARQL query Q depicted below, in which tp_1, tp_2 and tp_3 denote the triple patterns and c_1 and c_2 denote the conditions of the SPARQL filter.

```
SELECT ?x WHERE {
   ?x foaf:mbox ?mbox1.  # tp1
   ?y foaf:mbox "john@foo.com".  # tp2
   ?x foaf:knows ?y.  # tp3
   FILTER {
    contains(str(?mbox1), "foo.com") && # c1
    ?x != ?y } # c2
}
```

Let us compute function *sparqlCond* for each triple pattern:

- tp_1 has two variables, ?x and ?mbox1. No condition involves both variables, but c_1 involves ?mbox1 and has no other variable, thereby c_1 matches criteria (i) for tp_1. Condition c_2 involves ?x but it also involves ?y that is not in tp_1. Hence, $sparqlCond(tp_1, c_1 \,\&\&\, c_2) = c_1$.
- tp_2 has one variable, ?y, and no condition involves only ?y. Hence, no condition can be pushed into the translation of tp_2, that we denote by $sparqlCond(tp_2, c_1 \,\&\&\, c_2) = true$.
- tp_3 has two variables ?x and ?y, and only condition c_2 involves them both. Hence, $sparqlCond(tp_3, c_1 \,\&\&\, c_2) = c_2$.
- Lastly, only condition c_2 involves variables shared by several triples patterns: ?x and ?y, this shall entail a FILTER operator.

Finally we come up with the following abstract query:

$trans_m(Q, c_1 \,\&\&\, c_2) = transTP_m(tp_1, c_1)$
INNER JOIN $transTP_m(tp_2, true)$ ON {}
INNER JOIN $transTP_m(tp_3, c_2)$ ON {?x,?y}
FILTER(c_2)

Note that an INNER JOIN on an empty set of variables is equivalent to a Cartesian product (a CROSS JOIN in SQL).

4.3 Binding xR2RML Triples Maps to Triple Patterns

Before we define function $transTP_m$, that translates SPARQL triple patterns into atomic abstract queries, we elaborate on how to figure out which ones of the xR2RML triple maps are likely to generate RDF triples matching the triple pattern.

In the following, we assume that xR2RML triples are *normalized* in the sense defined by (Rodríguez-Muro and Rezk, 2015) for R2RML: a normalized triples map contains exactly one predicate-object map with exactly one predicate map and one object map, and any rr:class property is replaced by an equivalent predicate-object map with a constant predicate rdf:type. Also, we denote by TM.sub, TM.pred and

Definition 2. Binding of xR2RML mappings to SPARQL triple patterns.
Let m be a set of xR2RML triples maps, and gp be a well-designed graph pattern. $\boldsymbol{bind}_m(gp)$ *is the set of triple pattern bindings of gp under m, defined as follows:*

- $\boldsymbol{bind}_m(gp) = \boldsymbol{bind}_m(gp, true)$
- *if gp consists of a single triple pattern tp,* $\boldsymbol{bind}_m(gp, f)$ *is the pair (tp, TMSet) where* $TMSet = \{TM \mid TM \in m \wedge \boldsymbol{compatible}(TM.sub, tp.sub, f) \wedge \boldsymbol{compatible}(TM.pred, tp.pred, f) \wedge \boldsymbol{compatible}(TM.obj, tp.obj, f)\}$
- *if gp is (P1 AND P2),* $\boldsymbol{bind}_m(gp, f) = \boldsymbol{reduce}(\boldsymbol{bind}_m(P1, f), \boldsymbol{bind}_m(P2, f)) \cup \boldsymbol{reduce}(\boldsymbol{bind}_m(P2, f), \boldsymbol{bind}_m(P1, f))$
- *if gp is (P1 OPTIONAL P2),* $\boldsymbol{bind}_m(gp, f) = \boldsymbol{bind}_m(P1, f) \cup \boldsymbol{reduce}(\boldsymbol{bind}_m(P2, f), \boldsymbol{bind}_m(P1, f))$
- *if gp is (P1 UNION P2),* $\boldsymbol{bind}_m(gp, f) = \boldsymbol{bind}_m(P1, f) \cup \boldsymbol{bind}_m(P2, f)$
- *if gp is (P FILTER f'),* $\boldsymbol{bind}_m(gp, f) = \boldsymbol{bind}_m(P, f \,\&\&\, f')$

TM.obj respectively the subject map, the predicate map and the object map of triples map TM. Lastly, we adapt the concept of *triple pattern binding* defined by Unbehauen et al. as follows:

Definition 3. *Let m be an xR2RML mapping graph consisting of a set of xR2RML triples map, and tp be a triple pattern. A triples map TM ∈ m is* ***bound*** *to tp if it is likely to produce triples matching tp. A* ***triple pattern binding*** *is a pair (tp, TMSet) where TMSet is the set of triples maps of m that are bound to tp.*

Function $bind_m$, depicted in Definition 2, determines, for a graph pattern *gp*, the bindings of each triple pattern of *gp*. It takes into account join constraints implied by shared variables, and the SPARQL filter constraints whose unsatisfiability can be verified statically. This is achieved by means of two functions: *compatible* and *reduce*. These functions were introduced by (Unbehauen et al., 2013a), but important details were left untold. Especially, the authors did not formally define what the compatibility between a term map and a triple pattern term means, and they did not investigate the static compatibility between a term map and a SPARQL filter. Below we describe these functions in details and extend them to fit our context of an abstract query language.

Function *compatible* (Definition 4) checks if a term map is compatible with a triple pattern term and a SPARQL filter. A term map is always considered compatible with a variable triple pattern term, unless a SPARQL filter contradicts the term map. These situations are identified in function *compatibleFilter* (Definition 5), they pertain to type constraints expressed using SPARQL operators *isIRI*, *isLiteral* or *isBlank*, as well as language and data type constraints expressed using operators *lang* and *datatype*. For instance, if variable `?var` is matched with an object map that produces literals (`rr:termType rr:Literal`), the SPARQL constraint `isIRI(?var)` is unsatisfiable. When the triple pattern term is not a variable, function *compatible* identifies the similar situations wherein the triple pattern term and the term map cannot match with regards to the type of the triple pattern term (literal, IRI, blank node), its language tag (e.g. `"string"@en`) or its data type (e.g. `10^^xsd:integer`).

Function *reduce* uses the variables shared by two triple patterns to detect unsatisfiable join constraints, and thus reduces the set of triple maps bound to each triple pattern. For instance, let us consider two triple patterns tp_1 and tp_2 that have a shared variable `v`, triples map TM_1 is bound to tp_1 and triples map TM_2 is bound to tp_2. If the term map associated to `v` in TM_1 generates literals whereas the term map associated to `v` in TM_2 generates IRIs, we say that the term maps are incompatible. Consequently, function *reduce* rules out TM_1 from the bindings of tp_1 and TM_2 from the bindings of tp_2. In other words, $reduce(bind_m(tp_1), bind_m(tp_2))$ returns the reduced bindings of tp_1 such that the term maps associated to `v` in the bindings of tp_1 are compatible with the term maps associated to `v` in the bindings of tp_2. A formal definition of function *reduce* is given in (Michel et al., 2015b), and the concept of compatibility between term maps is shown in Definition 6.

Running Example. Let us consider the SPARQL query Q proposed in section 4.2. We first compute the triple pattern bindings for tp_1, tp_2 and tp_3 independently. The constant predicate of tp_1 and tp_2 matches the constant predicate map of triples map `<#Mbox>`. The subject and object of tp_1 are variables and the constant object of tp_2 ("john@foo.com") is compatible with the object map of `<#Mbox>`. Consequently `<#Mbox>` is bound to both:

$bind_m(tp_1, c_1 \,\&\&\, c_2) = (tp_1, \{<\#Mbox>\})$
$bind_m(tp_2, c_1 \,\&\&\, c_2) = (tp_2, \{<\#Mbox>\})$

Similarly we can show that `<#Knows>` is bound to tp_3:

$bind_m(tp_3, c_1 \,\&\&\, c_2) = (tp_3, \{<\#Knows>\})$.

Now let us consider the join constraint implied by the shared variable `?y`:

```
?y foaf:mbox "john@foo.com". # tp2
?x foaf:knows ?y. # tp3
```

`?y` is the subject in tp_2 that is bound to `<#Mbox>`, thus `?y` is associated to `<#Mbox>`'s subject map. `?y` is also

Definition 4. Compatibility between a term map, a triple pattern term and a SPARQL filter.
Let tpTerm be a triple pattern term, termMap be a term map of an xR2RML triples map TM and f be a SPARQL filter. It holds that termMap is compatible with tpTerm and f, denoted by ***compatible****(termMap, tpTerm, f), if termMap is compatible with filter f denoted by* ***compatibleFilter****(termMap, f), and either (i) tpTerm is a variable or (ii) none of the following assertions holds:*

- *tpTerm is a literal and the term type of termMap is not rr:Literal;*
- *tpTerm is an IRI and the term type of termMap is not rr:IRI;*
- *tpTerm is a blank node and the term type of termMap is not one of rr:BlankNode, xrr:RdfList, xrr:RdfBag, xrr:RdfSeq, xrr:RdfAlt;*
- *tpTerm is a literal with a language tag L, and the language of termMap is undefined or different from L;*
- *tpTerm is a literal with a datatype T, and the datatype of termMap is either undefined or different from T;*
- *termMap is constant-valued with value V, and tpTerm is different from V;*
- *termMap is template-valued with template string T, and tpTerm cannot match T;*
- *termMap is a ReferencingObjectMap and the subject map of the parent triples map is not compatible with tpTerm, i.e.* ¬***compatibleTermMaps****(termMap.parentTriplesMap.subjectMap, tpTerm).*

Definition 5. Compatibility between a term map and a SPARQL filter.
Let termMap be an xR2RML term map and f be a SPARQL filter. termMap is compatible with f, denoted as ***compatibleFilter****(termMap, f) if f =“true” or none of the following assertions holds:*

- *a necessary condition of f is isIRI(?var) and the term type of termMap is not rr:IRI;*
- *a necessary condition of f is isLiteral(?var) and the term type of termMap is not rr:Literal;*
- *a necessary condition of f is isBlank(?var) and the term type of termMap is not rr:BlankNode;*
- *a necessary condition of f is lang(?var)=“L” or langMatches(lang(?var),“L”), and the language of termMap is either not defined or different from L;*
- *a necessary condition of f is datatype(?var)=<T> and the datatype of termMap is either undefined or different from <T>.*

Definition 6. Compatibility between two term maps.
Let $termMap_1$ and $termMap_2$ be two xR2RML term maps. It holds that $termMap_1$ and $termMap_2$ are compatible, denoted by ***compatibleTermMaps****($termMap_1$, $termMap_2$) if none of the following assertions holds:*

- *$termMap_1$ and $termMap_2$ have different term types (property rr:termType).*
- *$termMap_1$ and $termMap_2$ have different language tags, or one has a language tag and the other does not.*
- *$termMap_1$ and $termMap_2$ are both template-valued, and they have incompatible template strings.*
- *$termMap_1$ (resp. $termMap_2$) is a referencing object map and the subject map of its parent triples maps is not compatible with termMap2 (resp. termMap1), i.e.* ¬***compatibleTermMaps****($termMap_1$.parentTriplesMap.subjectMap, $termMap_2$), (resp.* ¬***compatibleTermMaps****($termMap_1$, $termMap_2$.parentTriplesMap.subjectMap))*

the object in tp_3 that is bound to <#Knows>, thus ?y is associated to <#Knows>'s object map. The latter is a referencing object map whose parent is <#Mbox>. Therefore, reduce($bind_m(tp_2, c_1\&\&c_2)$, $bind_m(tp_3, c_1\&\&c_2)$) checks if the subject map of <#Mbox> is compatible with the object map of <#Knows>, that amounts to check if the subject map of <#Mbox> is compatible with itself, which is obvious. We can then show that
reduce($bind_m(tp_2, c_1\&\&c_2)$,$bind_m(tp_3, c_1\&\&c_2)$)=
(tp_2, {<#Mbox>}), and
reduce($bind_m(tp_3, c_1\&\&c_2)$,$bind_m(tp_2, c_1\&\&c_2)$)=
(tp_3, {<#Knows>})

In a similar manner, we can show that the join constraint implied by variable ?x, shared by tp_1 and tp_3, does not rule out any binding. Lastly, we obtain:
$bind_m(tp_1$ AND tp_2 AND $tp_3, c_1\&\&c_2)$ =
{(tp_1, {<#Mbox>}),
(tp_2, {<#Mbox>}),
(tp_3, {<#Knows>})}

4.4 Translation of a SPARQL Triple Pattern

Below we define the $transTP_m$ function and we elaborate on its main concepts. The interested reader is referred to (Michel et al., 2015b) for the comprehensive algorithm $transTP_m$.

Definition 7. Function transTP$_m$.
Let m be an xR2RML mapping graph consisting of a set of xR2RML triples maps, gp be a well-designed graph pattern, tp be a triple pattern of gp, and f be a SPARQL filter expression. Let getBoundTMs$_m$ be the function that, given gp, tp and f, returns the set of triples maps of m that are bound to tp in bind$_m$(gp,f). We denote by ***transTP***$_m$*(tp,f) the translation, under getBoundTMs$_m$(gp,tp,f), of "tp* FILTER *f" into an abstract query whereof results can be translated into RDF triples matching "tp* FILTER *f".*

The abstract query generated by function $transTP_m$ consists of operators from the abstract query language and *atomic abstract queries*. An atomic abstract query is obtained by matching a triple pattern with a triples map and denoted by {*From, Project, Where*} that we describe below. We have seen in the definition of function $bind_m$ that several triples maps may be bound to a single triple pattern *tp*, each one may produce a subset of the RDF triples matching *tp*. In such a case, $transTP_m$ translates *tp* into a UNION of per-triples-map atomics abstract queries. Additionally, we have seen that an xR2RML triples map may denote a cross-reference by means of a referencing object map, e.g. child triples map TM_1 produces the subject and predicate terms while parent triples map TM_2 produces object terms. This construct is translated by $transTP_m$ into the INNER JOIN of two atomic abstract queries:

```
{From1, Project1, Where1} AS child
INNER JOIN
{From2, Project2, Where2} AS parent
ON child/childRef = parent/parentRef
```

where `childRef` and `parentRef` denote the values of properties `rr:child` and `rr:parent` respectively. This is illustrated by the translation of tp_3 in Listing 4. Interestingly, we notice that INNER JOINs may be implied by shared SPARQL variables as well as cross-references denoted in the mappings. Similarly, UNIONs may arise from the SPARQL UNION operator or the binding of several triples maps to a triple pattern.

We now describe the three components of an atomic abstract query.

- The *From* part provides the concrete query that the abstract query relies on. It contains the logical source of a triples map i.e. the `xrr:query` property and an optional iterator (property `rml:iterator`). In our running example, triples map `<#Mbox>` is bound to tp_1, the *From* part consists of the query in the logical source of `<#Mbox>`: `db.people.find({'emails':{$ne:null}})`

- *Project* is the set of xR2RML references that must be projected, i.e. returned as part of the query results. An xR2RML reference may be e.g. a column name in a relational database, a JSONPath expression for MongoDB database or an XPath expression for a native XML database. If an xR2RML reference corresponds to a variable of the triple pattern, it is always projected. In our running example, the subject and object of tp_1 are the `?x` and `?mbox1` variables. The references in the subject map and object map of triples map `<#Mbox>` must be projected, hence the *Project* part for tp_1: {`$.id AS ?x, $.emails.* AS ?mbox1`}. Furthermore, the child and parent joined references of a referencing object map must be projected in order to fit databases that do not support joins. In the relational database case, those projected references (columns) are useless since the database can compute the join operation. Conversely, in MongoDB for instance, the join shall be processed by the query processing engine, therefore joined references are necessary.

- *Where* is a set of conditions about xR2RML references. They are entailed by matching each term of triple pattern *tp* with its corresponding term map in triples map *TM*: the subject of *tp* is matched with *TM*'s subject map, the predicate with *TM*'s predicate map and the object with *TM*'s object map. Additional conditions are entailed from the SPARQL filter *f*. In (Michel et al., 2015b) we show that three types of condition may be created:
(i) a SPARQL variable in the triple pattern is turned into a not-null condition on the xR2RML reference corresponding to that variable in the term map, denoted by *isNotNull(<xR2RML reference>)*;
(ii) A constant triple pattern term (IRI or literal) is turned into an equality condition on the xR2RML reference corresponding to that RDF term in the term map, denoted by *equals(<xR2RML reference>, value)*;
(iii) A SPARQL filter condition about a SPARQL variable is turned into a filter condition, denoted by *sparqlFilter(<xR2RML reference>, f)*.

Running Example. Triple pattern tp_2 is matched with `<#Mbox>`. It has the variable `?y` in the subject position and a constant term in the object position. Consequently the *Where* part for tp_2 contains two conditions: `isNotNull($.id)` and `equals($.emails.*, "john@foo.com")`. When we put all the pieces together, we can rewrite the

```
trans_m(tp1 AND tp2 AND tp3, c1 && c2) = transTP_m(tp1,c1)
    INNER JOIN transTP_m(tp2,true) ON {}
    INNER JOIN transTP_m(tp3,c2) ON {?x,?y}
FILTER(?x != ?y)

transTP_m(tp1,c1) =
   { From:    {"db.people.find({'emails': {$ne: null}})"},
     Project: {$.id AS ?x, $.emails.* AS ?mbox1},
     Where:   {isNotNull($.id), isNotNull($.emails.*),
               sparqlFilter(contains(str(?mbox1),"foo.com"))}}

transTP_m(tp2,true) =
   { From:    {"db.people.find({'emails': {$ne: null}})"},
     Project: {$.id AS ?y},
     Where:   {isNotNull($.id), equals($.emails.*,"john@foo.com")}}

transTP_m(tp3,c2) =
   { From:    {"db.people.find({'contacts':{$size: {$gte:1}}})"},
     Project: {$.id AS ?x, $.contacts.*},
     Where:   {isNotNull($.id), isNotNull($.contacts.*),
               sparqlFilter(?x != ?y)} AS chlid
   INNER JOIN
   { From:    {"db.people.find({'emails':{$ne: null}})" },
     Project: {$.emails.*, $.id AS ?y},
     Where:   {isNotNull($.emails.*), isNotNull($.id),
               sparqlFilter(?x != ?y)} AS parent
   ON child/$.contacts.* = parent/$.emails.*
```

Listing 4: Rewriting of SPARQL query Q into an abstract query.

SPARQL query Q into the abstract query depicted in Listing 4.

4.5 Abstract Query Optimization

At this point, our method produces abstract queries that are effective, i.e. that preserve the semantics of SPARQL queries. Yet, their structure may show unnecessary complexity, and entail inefficient queries when translated into a target query language. Although query optimizations may be postponed to the final translation step, it is interesting to figure out which ones can be achieved on the abstract representation first, and leave only database-specific optimizations to the latter stage. SPARQL-to-SQL methods proposed various SQL query optimizations (Unbehauen et al., 2013b; Rodríguez-Muro and Rezk, 2015; Elliott et al., 2009), that are often independent of SQL. Below we review some of these techniques referring to the terminology defined in (Unbehauen et al., 2013b). We show that some of them are implemented in our method by construction, and others apply in the context of our abstract query language.

Filter Optimization. In a naive approach, strings generated by R2RML templates are dealt with using an SQL comparison of the resulting strings rather than the database values used in the template. This is notably the case of IRIs that are generally built according to a string template. As a consequence, the query evaluation cannot take advantage of existing indexes and performs poorly. In our approach, equality conditions apply to xR2RML references rather than on the generated IRIs, hence the *Filter Optimization* is enforced by construction.

Filter Pushing. As mentioned earlier, the translation of a SPARQL filter into an encapsulating SELECT WHERE clause tends to lower the selectivity of inner queries, and the query evaluation process may have to deal with unnecessarily large intermediate results. In our approach, *Filter pushing* is achieved by construction in function $trans_m$ by pushing down SPARQL filters, as much as possible, in the translation of each triple pattern.

Self-join Elimination. A self-join may occur when several triples maps share the same logical source. This can result in several triple patterns being translated into atomic abstract queries with the same *From* part. The *Self-Join Elimination* consists in merging the criteria of both atomic queries into a single equivalent query. In our running example (Listing 4), the atomic query in `transTP_m(tp2, true)` and the second atomic query in `transTP_m(tp3, c2)` have the same *From* part and project the same variable `?y`. Using joins commutativity, those two queries can be merged into a single one depicted in the third atomic abstract query in Listing 5.

```
transm(tp1 AND tp2 AND tp3, c1 && c2) =
    { From:    {"db.people.find({'emails':{$ne:null}})"},
      Project: {$.id AS ?x, $.emails.* AS ?mbox1},
      Where:   {isNotNull($.id), isNotNull($.emails.*),
                sparqlFilter(contains(str(?mbox1),"foo.com"))}}
    INNER JOIN
    { From:    {"db.people.find({'contacts':{$size: {$gte:1}}})"},
      Project: {$.id AS ?x, $.contacts.*},
      Where:   {isNotNull($.id), isNotNull($.contacts.*),
                sparqlFilter(?x != ?y)}} AS chlid
    ON {?x,?y}
    INNER JOIN
    { From:    {"db.people.find({'emails':{$ne: null}})" },
      Project: {$.emails.*, $.id AS ?y},
      Where:   {isNotNull($.emails.*), isNotNull($.id),
                equals($.emails.*,"john@foo.com"),
                sparqlFilter(?x != ?y)}} AS parent
    ON child/$.contacts.* = parent/$.emails.* )
    FILTER(?x != ?y)
```

Listing 5: Optimization of query Q by self-join elimination.

Optional-self-join Elimination. The self-join issue can equally occur in the case of an OPTIONAL triple pattern that is translated into a LEFT OUTER JOIN. Similarly to the *Self-Join Elimination*, we can merge abstract atomic queries with the difference that null values must be allowed for terms that only show in the right operand of the left join. As a result, *isNotNull* conditions of the right operand are removed, and *equals* conditions of the form `equals(expr, value)` are replaced with a new type of condition including an *isNull* condition and an *OR* operator: `isNull(expr) OR equals(expr, value)`.

Self-union Elimination. A UNION operator can be created either due to the SPARQL UNION operator or during the translation of a triple pattern to which several triples maps are bound (in function $transTP_m$). Similarly to the *Self-Join Elimination*, a union of several atomic abstract queries sharing the same logical source can be merged into a single query.

Projection Pushing. In future works, we intend to study the relevance and applicability of the *Projection Pushing* (Elliott et al., 2009) that helps to efficiently deal with queries on distinct values of variables bound with constant term maps, such as:
`SELECT DISTINCT ?p WHERE {?s ?p ?o}`.

5 APPLICATION TO MongoDB

MongoDB is a NoSQL database that stores data as JSON documents (more precisely Binary-JSON). Its JavaScript interface defines a declarative query language exemplified in the logical sources of Listing 2. In recent years, MongoDB has become a leader in the NoSQL market, making it an interesting candidate for RDF-based data integration systems, and a potential contributor to the Web of Data.

In our running example, we have shown how to translate a SPARQL query into an abstract query, under xR2RML mappings of arbitrary MongoDB documents to RDF. Unlike SQL or XQuery whose expressiveness is similar to that of SPARQL, the expressiveness of the MongoDB query language is far more limited: joins are not supported and filters are supported with strong restrictions (e.g. no comparison between fields of a document). Consequently, in the abstract query of Listing 5, the INNER JOIN and FILTER operators on the one hand, and the *sparqlFilter* conditions on the other hand, cannot be translated into equivalent MongoDB queries. Hence, they shall be computed by the query processing engine. In (Michel et al., 2015b), we show that it is possible to rewrite an atomic abstract query with *isNotNull* and *equals* conditions into a union of MongoDB queries that shall retrieve at least all matching documents.

Implementation. To validate our method, we are developing an open source prototype implementation available on Github[2]. The prototype currently implements the rewriting of an atomic abstract query into concrete MongoDB queries. The translation of a SPARQL query into an abstract query and its evaluation by the query processing engine is an on-going development work at the time of writing.

[2]https://github.com/frmichel/morph-xr2rml/tree/query_rewrite

6 CONCLUSION AND PERSPECTIVES

The method proposed in this paper aims at fostering the development of SPARQL interfaces to heterogeneous databases, as we believe this is a key to the advent of the Web of Data.

Leveraging R2RML-based SPARQL-to-SQL works, we have defined a method to translate a SPARQL query into a pivot abstract query, utilizing xR2RML to describe the mapping of a target database to RDF. The method determines a reduced set of mappings matching each SPARQL triple pattern, and takes into account join constraints and SPARQL filters. Lastly, several query optimizations are applied to the abstract query in order to facilitate the subsequent translation into the target query language.

At this stage, our method has some limitations that we may address in the future. Firstly, SPARQL named graphs and solution modifiers (DISTINCT, OFFSET, LIMIT, ORDER BY, HAVING) are not considered. Besides, like in most SPARQL rewriting approaches so far, SPARQL 1.1 is not fully supported, in particular with respect to property paths and negation operators (NOT EXISTS, MINUS). In the translation of a triple pattern, the management of SPARQL filters is postponed to the translation into a target query using the *sparqlFilter* condition. Yet, further works shall try to raise filters at the abstract query level. For instance SPARQL operators BIND or VALUES may be turned into equivalent *equals* conditions, and BOUND into *isNotNull* conditions. Secondly, thanks to the example of MongoDB, we have shown that bridging the gap between the expressiveness of SPARQL and that of the target query language may entail the generation of multiple independent target database queries, delegating several steps to the query processing engine, e.g. joins or complex filtering. Therefore, some classical query optimization questions shall arise during the development of the query processing engine, such as what is the most efficient order to compute INNER JOINs of intermediate queries. In this regard, the query processing engine may need to embark query plan optimization logics such as the bind join (Haas et al., 1997) to inject intermediary results into a subsequent query, and the join re-ordering based on the number of results that queries shall retrieve, very similarly to the methods applied in distributed SPARQL query engines (Schwarte et al., 2011; Görlitz and Staab, 2011).

We are currently developing a prototype implementation of our method. In the short-term we intend to run performance evaluations. Beyond this, we envisage two real-life use cases. Firstly, in the context of the Zoomathia research project[3], a taxonomic reference designed to support studies in Conservation Biology was translated into a SKOS[4] thesaurus (Callou et al., 2015). The taxonomic reference is stored in a MongoDB database, and the RDF graph is materialized at once. Our perspective is to provide a dynamic access to the SKOS thesaurus using our SPARQL-to-MongoDB prototype. Secondly, we are having discussions with researchers who intend to explore the added value of Semantic Web technologies to support ecology and agronomic studies. They maintain a large MongoDB database of phenotype information about thousands of plants, that they wish to access using SPARQL. This context would be a significant and realistic use case of our method.

REFERENCES

Bikakis, N., Tsinaraki, C., Stavrakantonakis, I., Gioldasis, N., and Christodoulakis, S. (2015). The SPARQL2XQuery interoperability framework. *World Wide Web*, 18(2):403–490.

Bischof, S., Decker, S., Krennwallner, T., Lopes, N., and Polleres, A. (2012). Mapping between RDF and XML with XSPARQL. *J. Data Semantics*, 1(3):147–185.

Bizer, C. and Cyganiak, R. (2006). D2R server - Publishing Relational Databases on the Semantic Web. In *ISWC*.

Callou, C., Michel, F., Faron-Zucker, C., Martin, C., and Montagnat, J. (2015). Towards a Shared Reference Thesaurus for Studies on History of Zoology, Archaeozoology and Conservation Biology. In *SW For Scientific Heritage, ESWC*.

Chebotko, A., Lu, S., and Fotouhi, F. (2009). Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering*, 68(10):973–1000.

Das, S., Sundara, S., and Cyganiak, R. (2012). R2RML: RDB to RDF mapping language.

Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., and Van de Walle, R. (2014). RML: A generic language for integrated RDF mappings of heterogeneous data. In *LDOW*.

Elliott, B., Cheng, E., Thomas-Ogbuji, C., and Ozsoyoglu, Z. M. (2009). A complete translation from SPARQL into efficient SQL. In *IDEAS'09*, pages 31–42. ACM.

Görlitz, O. and Staab, S. (2011). SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Intl. Ws. COLD*.

Haas, L., Kossmann, D., Wimmers, E., and Yang, J. (1997). Optimizing Queries across Diverse Data Sources. In *VLDB*, pages 276–285.

Michel, F., Djimenou, L., Faron-Zucker, C., and Montagnat, J. (2015a). Translation of Relational and Non-Relational Databases into RDF with xR2RML. In *WebIST*, pages 443–454.

[3]http://www.cepam.cnrs.fr/zoomathia

[4]http://www.w3.org/2009/08/skos-reference/skos.html

Michel, F., Faron-Zucker, C., and Montagnat, J. (2015b). Mapping-based SPARQL access to a MongoDB database. Technical report, CNRS. https://hal.archives-ouvertes.fr/hal-01245883v4.

Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45.

Priyatna, F., Corcho, O., and Sequeda, J. (2014). Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph. In *WWW*.

Rodríguez-Muro, M. and Rezk, M. (2015). Efficient SPARQL-to-SQL with R2RML mappings. *J. Web Semantics*, 33:141–169.

Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: Optimization techniques for federated query processing on Linked Data. In *ISWC*, pages 601–616. Springer.

Sequeda, J. F. and Miranker, D. P. (2013). Ultrawrap: SPARQL execution on relational data. *J. Web Semantics*, 22:19–39.

Unbehauen, J., Stadler, C., and Auer, S. (2013a). Accessing relational data on the web with sparqlmap. In *Semantic Technology*, pages 65–80. Springer.

Unbehauen, J., Stadler, C., and Auer, S. (2013b). Optimizing SPARQL-to-SQL Rewriting. In *Proceedings of IIWAS '13*, page 324. ACM.