Bridging the Semantic Web and NoSQL Worlds: Generic SPARQL Query Translation and Application to MongoDB

Franck Michel, Catherine Faron Zucker, Johan Montagnat

Bridging the Semantic Web and NoSQL Worlds: Generic SPARQL Query Translation and Application to MongoDB

Franck Michel[0000−0001−9064−0463], Catherine Faron-Zucker[0000−0001−5959−5561], and Johan Montagnat

Université Côte d'Azur, Inria, CNRS, I3S, France
franck.michel@cnrs.fr, faron@i3s.unice.fr, johan.montagnat@cnrs.fr

Abstract. RDF-based data integration is often hampered by the lack of methods to translate data locked in heterogeneous silos into RDF representations. In this paper, we tackle the challenge of bridging the gap between the Semantic Web and NoSQL worlds, by fostering the development of SPARQL interfaces to heterogeneous databases. To avoid defining yet another SPARQL translation method for each and every database, we propose a two-phase method. Firstly, a SPARQL query is translated into a pivot abstract query. This phase achieves as much of the translation process as possible regardless of the database. We show how optimizations at this abstract level can save subsequent work at the level of a target database query language. Secondly, the abstract query is translated into the query language of a target database, taking into account the specific database capabilities and constraints. We demonstrate the effectiveness of our method with the MongoDB NoSQL document store, such that arbitrary MongoDB documents can be aligned on existing domain ontologies and accessed with SPARQL. Finally, we draw on a real-world use case to report experimental results with respect to the effectiveness and performance of our approach.

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

1 Introduction

The Resource Description Framework (RDF) [11] is increasingly adopted as the pivot format for integrating heterogeneous data sources. It offers a unified data model that allows building upon countless existing vocabularies and domain ontologies, while benefiting from Semantic Web's reasoning capabilities. It also allows leveraging the growing, world-scale knowledge base referred to as the Web of Data. Today, increasing amounts of RDF data are published on the Web, notably following the Linked Data principles [2,19]. These data often originate from heterogeneous silos that are inaccessible to data integration systems and search engines. Hence, a first step to enabling RDF-based data integration consists in translating legacy data from heterogeneous formats into RDF representations.

During the last fifteen years, much work has investigated how to translate common databases and data formats into RDF. Relational databases were primarily targeted [36,34], along with a handful of data formats such as XML [3] and CSV [28]. Meanwhile, the database landscape has significantly diversified with the adoption of various non-relational models. Initially designed as the core system of Big Data Web applications, NoSQL databases have gained momentum and are now increasingly adopted as general-purpose, commonplace databases. Today, companies and institutions store massive amounts of data in NoSQL instances. So far however, these data often remain inaccessible to RDF-based data integration systems, and consequently invisible to the Web of Data. although unleashing their data could potentially spur new integration opportunities and push the Web of Data forward.

The Semantic Web and NoSQL worlds build upon very different paradigms that are challenging to bridge over: whereas the former handles highly connected graphs along with the rich expressiveness of SPARQL, the latter trades off query expressiveness for scalability and fast retrieval of denormalized data[1]. As a result of these discrepancies, bridging the gap between those two worlds is a challenging endeavor.

Two strategies generally apply when it comes to access non-RDF data as RDF. In the *graph materialization* strategy, the transformation is applied exhaustively to the database content, the resulting RDF graph is loaded into a triple store and accessed through a SPARQL query engine [18] or by dereferencing URIs (as Linked Data). On the one hand, this strategy easily supports further processing or analysis, since the graph is made available at once. On the other hand, the materialized RDF graph may rapidly become outdated if the pace of database updates is high. Running the transformation process periodically is a common workaround, but in the context of large data sets, the cost (in time, memory and CPU) of materializing and reloading the graph may become out of reach. To work out this issue, the *query rewriting* strategy aims to access heterogeneous databases as virtual RDF graphs. A query processor rewrites a SPARQL query into the query language of the target database. The target database query is evaluated at run-time such that only relevant data are fetched from the database and translated into RDF triples. This strategy better scales to big data sets and guarantees data freshness, but entails overheads that may penalize performances if complex analysis is needed.

In previous works we defined a generic mapping language, xR2RML [25], that enables the translation of a broad scope of data sources into RDF. The mapping instructs how to translate each data item from its original format into RDF triples, by adapting to the multiplicity of query languages and data models. We applied xR2RML to the MongoDB NoSQL document store[2] and we implemented the *graph materialization* strategy.

[1] We refer to key-value stores, document stores and column family stores but leave out graph stores that generally come with a richer query expressiveness.
[2] https://www.mongodb.org/

To cope with large and frequently updated data sets though, we wish to tackle the question of accessing such databases using the query rewriting strategy. Hence, to avoid defining yet another SPARQL translation method for each and every database, in this paper we investigate a general two-phase method. Firstly, given a set of xR2RML mappings, a SPARQL query is rewritten into a pivot abstract query. This phase achieves as much of the translation process as possible regardless of the database, and enforces early query optimizations. Secondly, the abstract query is translated into the target database query language, taking into account the specific database capabilities and constraints. We demonstrate the effectiveness of our method in the case of MongoDB, accessing arbitrary MongoDB documents with SPARQL. We show that we can always rewrite an abstract query into a union of MongoDB *find* queries that shall return all the documents required to answer the SPARQL query.

The rest of this article is organized as follows. After a review of SPARQL query rewriting approaches in section 2, we quickly remind the principles and main features of the xR2RML mapping language in section 3. Then, in sections 4 and 5 we describe the two-phase method introduced above. In section 6, we describe a real-world use case and we report experimental results with respect to the effectiveness and performance of our approach. Finally, we discuss our solution and envision some perspectives in section 7, and we draw some conclusions in section 8.

2 Related Works

2.1 Rewriting SPARQL to SQL and XQuery

Since the early 2000's, various works have investigated methods to query legacy data sources with SPARQL. Relational databases (RDB) have caught much attention, either in the context of RDB-backed RDF stores [10,35,14] or using arbitrary relational schemas [5,38,29,31,32]. These methods harness the ability of SQL to support joins, unions, nested queries and various string manipulation functions. Typically, a conjunction of two SPARQL basic graph patterns (BGP) results in the inner join of their respective translations; their union results in a SQL UNION ALL clause; the SPARQL OPTIONAL clause results in a left outer join, and a SPARQL FILTER results in an encapsulating SQL SELECT WHERE clause.

Chebotko's algorithm [10] focused on RDB-based triple stores. Priyatna et al. [29] extended it to support custom R2RML mappings (the W3C recommendation of an RDB-to-RDF mapping language [12]) while applying several query optimizations. Two limitations can be emphasized though: (i) R2RML mappings must have constant predicates, *i.e.* the predicate term of the generated RDF triples cannot be built from database values; (ii) Triple patterns are considered and translated independently of each other, even when they share SPARQL variables. The resulting SQL query embeds unnecessary complexity that is taken care of later on, in the SQL query optimization step. Unbehauen et al. [38] clear the first limitation by defining the concept of compatibility between the RDF

terms of a SPARQL triple pattern and R2RML mappings, which enables managing variable predicates. Furthermore, to address the second limitation, they pre-checking join constraints implied by shared variables in order to reduce the number of candidate mappings for each triple pattern. Yet again, two limitations can be noticed: (iii) References between R2RML mappings are not considered, hence joins implied by shared variables are dealt with but joins declared in the R2RML mapping graph are ignored. (iv) The rewriting process associates each part of a mapping to a set of columns, called column group, which enables filter, join and data type compatibility checks. This leverages SQL capabilities (CASE, CAST, string concatenation, etc.), making it hardly applicable out of the scope of SQL-based systems. In the three aforementioned approaches, the optimization is dependent on the target database language, and can hardly be generalized. In our attempt to rewrite SPARQL queries in the general case, such optimization are performed earlier, regardless of the target database capabilities.

In a somewhat different approach, Rodríguez-Muro and Rezk [32] extend the *ontop* Ontology-Based Data Access (OBDA) system to support R2RML mappings. A SPARQL query and an R2RML mapping graph are translated into a Datalog program. This formal representation is used to combine and apply optimization techniques from logic programming and SQL querying. The optimized program is then translated into an executable SQL query.

Other approaches investigated the querying of XML databases in a rather similar philosophy. For instance, SPARQL2XQuery [4] relies on the ability of XQuery to support joins, nested queries and complex filtering. Typically, a SPARQL FILTER is translated into an encapsulating For-Let-Where XQuery clause.

Finally, it occurs that the rich expressiveness of SQL and XQuery makes it possible to translate a SPARQL 1.0 query into a single, possibly deeply nested, target query, whose semantics is provably strictly equivalent to that of the SPARQL query. Commonly, query optimization issues are addressed at the level of the produced target query, or they may even be delegated to the target database optimization engine. Hence, the above reviewed methods are tailored to the expressiveness of the target query language, such that SQL or XQuery specificities are woven into the translation method itself, which undermines the ability to use such methods beyond their initial scope.

2.2 Rewriting SPARQL to NoSQL

To the best of our knowledge, little work has investigated how to perform RDF-based data integration over the NoSQL family of databases. An early work[3] has tackled the translation of CouchDB[4] documents into RDF, but did not addressed SPARQL rewriting. MongoGraph[5] is an extension of the AllegroGraph triple store to query arbitrary MongoDB documents with SPARQL. But very

[3]https://github.com/agrueneberg/Sessel

[4]http://couchdb.apache.org/

[5]http://franz.com/agraph/support/documentation/4.7/mongo-interface.html

much like the Direct Mapping [1] defined in the context of RDBs, both works come up with an ad-hoc ontology (*e.g.* each JSON field name is turned into a predicate) and hardly supports the reuse of existing ontologies. Tomaszuk proposed to use a MongoDB database as an RDF triple store [37]. In this context, the author devised a translation of SPARQL queries into MongoDB queries, that is however closely tied to the specific database schema and thus is unfit for arbitrary documents.

More in line with our work, Botoeva et al. proposed a generalization of the OBDA principles [30] to MongoDB [8]. They describe a two-step rewriting process of SPARQL queries into a MongoDB *aggregate* pipeline. In section 7, we analyze in further details the relationship between their approach and ours. Interestingly, to the best of our knowledge, only one approach tackled the key-value store subset of NoSQL databases. Mugnier et al. [26] define the NO-RL rule language that can express lightweight ontologies to be applied to key-value stores. Leveraging the formal semantics of NO-RL, they propose an algorithm to reformulate a query under a NO-RL ontology, but SPARQL is not considered.

Finally, since NoSQL document stores are based on JSON, let us mention the JSON-LD syntax that is meant for the serialization of Linked Data in the JSON format. When applied to existing JSON documents, a JSON-LD profile can be considered as a lightweight method to interpret JSON data as RDF. Such a profile could be exploited by a SPARQL rewriting engine to enable the querying of document stores with SPARQL. This approach would be limited though, since JSON-LD is not meant to describe rich mappings from JSON to RDF, but simply to interpret JSON as RDF. It lacks the expressiveness and flexibility required to align JSON documents with domain ontologies that may model data in a rather different manner. Besides, we do not want to define a method specifically tailored to MongoDB; our point is to provide a generic rewriting method that can be applied to the concrete case of MongoDB as well as various other databases.

3 The xR2RML Mapping Language

The xR2RML mapping language [25] intends to foster the translation of legacy data sources into RDF. It can describe the mapping of an extensible scope of databases to RDF, independently of any query language or data model. It is backward compatible with R2RML and relies on RML [13] for the handling of various data formats. It can translate data with mixed embedded formats and generate RDF collections and containers.

An xR2RML mapping defines a logical source (property `xrr:logicalSource`) as the result of executing a query against an input database (`xrr:query` and `rr:tableName`). An optional iterator (value of property `rml:iterator`) can be applied to each query result, and a `xrr:uniqueRef` property can identify unique fields. Data from the logical source is mapped to RDF terms (literal, IRI, blank node) by term maps. There exists four types of term maps: a subject map generates the subject of RDF triples, predicate and object maps produce the predicate

and object terms, and an optional graph map is used to name a target graph. Listing 1.1 depicts two mappings `<#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 references* whose syntax depends on the target database and is an implementation choice: typically, this may be a column name in case of a relational database, an XPath expression in case of an XML database, or a JSONPath[6] expression in case of NoSQL document stores like MongoDB or CouchDB. xR2RML references are used with property `xrr:reference` whose value is a single xR2RML reference, and property `rr:template` whose value is a template string which may contain several references. In Listing 1.1, both subject maps use a template to build IRI terms by concatenating `http://example.org/member/` with the value of the `"id"` JSON field.

```
<#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 [ rr:template "mailto:{$.emails.*}"; rr:termType rr:IRI ]
  ].
<#Knows>
  xrr:logicalSource [
    xrr:query "db.people.find({'contacts':{$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 1.1. xR2RML example mapping graph

When the evaluation of an xR2RML reference produces several RDF terms, the xR2RML processor creates one triple for each term. Alternatively, the `rr:termType` property of a term map can be used to group the terms in an RDF collection while specifying a language tag or data type. Besides, the default iteration model can be modified using *nested term maps*, notably useful to parse nested collections of values and generate appropriate triples.

xR2RML allows to model cross-references by means of *referencing object maps* that use values produced by the subject map of a parent mapping as the objects of triples produced by a child mapping. Properties `rr:child` and `rr:parent` specify the join condition between documents of both mappings.

Running Example. To illustrate the description of our method, we define a running example that we shall use throughout this paper. Let us consider a MongoDB database with a collection `people` depicted in Listing 1.2: each JSON document provides the identifier, email addresses and contacts of a person; contacts are identified by their email addresses.

Let us now consider the xR2RML mapping graph in Listing 1.1, consisting of two mappings `<#Mbox>` and `<#Knows>`. The logical source of mappings `<#Mbox>`,

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

```
{ "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 1.2. MongoDB collection "people" containing two documents

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 documents in Listing 1.2, the xR2RML mapping graph generates the following RDF triples:

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

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

4 From SPARQL to Abstract Queries

Section 2 emphasized that SPARQL rewriting methods for SQL or XQuery rely on prior knowledge about the target query language expressiveness. This makes possible the semantics-preserving translation of a SPARQL query into a single equivalent target query. In the general case however (beyond SQL and XQuery), the target query language may not support joins, unions, sub-queries and/or filtering. To tackle this challenge, our method first enacts the database-independent steps of the rewriting process. To generate the abstract query, we rely on and extend the R2RML-based SPARQL rewriting approaches reviewed in section 2, while taking care of avoiding the limitations highlighted. More specifically, we focus on rewriting a SPARQL 1.0 graph pattern, whatever the query form (SELECT, ASK, DESCRIBE, etc.). The translation of a SPARQL graph pattern into an abstract query consists of four steps, sketched in Fig. 1 and described in the next sub-sections. §4.1: A SPARQL 1.0 graph pattern is rewritten into an abstract expression exhibiting operators of the abstract query language. §4.2: We identify candidate xR2RML mappings likely to generate RDF triples that match each triple pattern. §4.3: Each triple pattern is translated into a sub-query according to the set of xR2RML mappings identified. A sub-query consists of operators of the abstract query language and atomic abstract queries. §4.4: We enforce several optimizations on the resulting abstract query, *e.g.* self-joins or self-unions elimination.

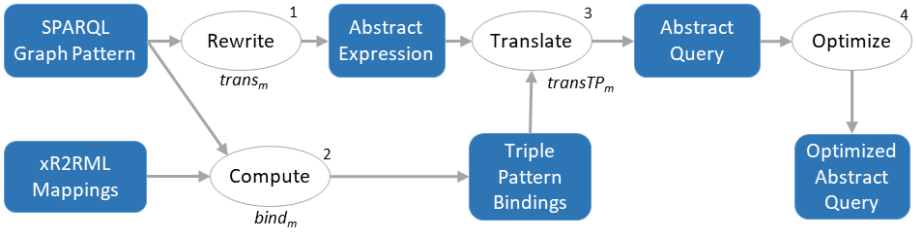

Fig. 1. Translation of a SPARQL 1.0 graph pattern into an optimized abstract query

4.1 Translation of a SPARQL Graph Pattern

Our pivot abstract query language complies with the grammar depicted in Def. 1. It derives from the syntax and semantics of SPARQL [27]: the language keeps the names of several SPARQL operators (UNION, LIMIT, FILTER) and prefers the SQL terms INNER JOIN ON and LEFT OUTER JOIN ON to refer to join operations more explicitly. A notable difference with SPARQL is that, in the tree representation of a query, the leaves of a SPARQL query are triple patterns. Conversely, the leaves of an abstract query are *Atomic Abstract Queries* (§4.3).

The INNER JOIN and LEFT OUTER JOIN operators stem from the join constraints implied by shared variables. Somehow, the second INNER JOIN in Def. 1, including the "*AS child*" and "*AS parent*" notations, is entailed by the join constraints expressed in xR2RML mappings using referencing object maps and properties `rr:child` and `rr:parent`. Notation v_1,... v_n, in the join operators, stands for the set of SPARQL variables on which the join is to be performed. Notation <*Ref*> stands for any valid xR2RML data element reference, *i.e.* a column name for a tabular data source, an XPath expression for an XML database, a JSONPath expression for a NoSQL document store such as MongoDB and CouchDB, etc.

Definition 1. ***Grammar of the Abstract Pivot Query Language***

```
<AbstractQuery> ::= <AtomicQuery> | <Query> |
                    <Query> FILTER <SPARQL filter> | <Query> LIMIT <integer>
<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, Limit}
```

The first query transformation step is implemented by function $trans_m$ depicted in Def. 2. It rewrites a well-designed SPARQL graph pattern [27] into an abstract query while making no assumption with respect to the target database query capabilities. It extends the algorithms proposed in [10], [38] and [29].

Definition 2. ***Translation of a SPARQL query into an abstract query under xR2RML mappings (function $trans_m$).***
Let m be an xR2RML mapping graph consisting of a set of xR2RML mappings. Let gp be a well-designed SPARQL graph pattern, f be a SPARQL filter and l an integer limit value representing the maximum number of results.
We denote by $\boldsymbol{trans}_m(gp, f, l)$ *the translation, under m, of "gp FILTER f" into an abstract query that shall not return more than l results. We denote by* $\boldsymbol{trans}_m(gp)$ *the result of* $\boldsymbol{trans}_m(gp, true, \infty)$. *Function* $\boldsymbol{trans}_m$ *is defined recursively as follows:*

- *if gp consists of a single triple pattern tp,* $\boldsymbol{trans}_m(gp, f, l) = \boldsymbol{transTP}_m(tp, \boldsymbol{sparqlCond}(tp, f), l)$
  where $\boldsymbol{transTP}_m$ *translates a single triple pattern into an abstract query (§4.3) and* ***sparqlCond*** *discriminates SPARQL filter conditions (§4.1).*
- *if gp is (P LIMIT l'),* $\boldsymbol{trans}_m(gp, f, l) = \boldsymbol{trans}_m(gp, f, min(l, l'))$
- *if gp is (P FILTER f'),* $\boldsymbol{trans}_m(gp, f, l) = \boldsymbol{trans}_m(P, f \wedge f', \infty)$ ***FILTER sparqlCond***$(P, f \wedge f')$ ***LIMIT*** l
- *if gp is* $(P_1$ *AND* $P_2)$, $\boldsymbol{trans}_m(gp, f, l) = \boldsymbol{trans}_m(P_1, f, \infty)$ ***INNER JOIN*** $\boldsymbol{trans}_m(P_2, f, \infty)$ ***ON*** $var(P_1) \cap var(P_2)$ ***LIMIT*** l
- *if gp is* $(P_1$ *OPTIONAL* $P_2)$, $\boldsymbol{trans}_m(gp, f, l) =$ $\boldsymbol{trans}_m(P_1, f, \infty)$ ***LEFT OUTER JOIN*** $trans_m(P_2, f, \infty)$ ***ON*** $var(P_1) \cap var(P_2)$ ***LIMIT*** l
- *if gp is* $(P_1$ *UNION* $P_2)$, $\boldsymbol{trans}_m(gp, f, l) = \boldsymbol{trans}_m(P_1, f, l)$ ***UNION*** $\boldsymbol{trans}_m(P_2, f, l)$ ***LIMIT*** l

As a simplification, notations "FILTER true" and "LIMIT ∞" may be omitted.

Example. Let us give a first simple illustration. SPARQL query Q_1 contains a graph pattern gp_1 that consists of two triple patterns, tp_1 and tp_2:

```
Q1: SELECT ?x WHERE {
      ?x foaf:mbox ?mbox. # tp1
      ?x foaf:knows ?y. } # tp2
```

The application of function $trans_m$ to the graph pattern gp_1 is as follows:

```
transm(gp1)
 = transm(gp1, true, ∞)
 = transTPm(tp1, true, ∞) INNER JOIN
   transTPm(tp2, true, ∞) ON {var(tp1) ∩ var(tp2)}
   LIMIT ∞
 = transTPm(tp1) INNER JOIN transTPm(tp2) ON {?x}
```

Dealing with SPARQL filters. SPARQL rewriting methods reviewed in section 2 generally adopt a bottom-up approach where, typically, a SPARQL FILTER translates into an encapsulating query (*e.g.* a SELECT-WHERE clause in the case of SQL). Thus, filters in the outer query do not contribute to the selectivity of inner-queries that may return large intermediate results. This flaw is commonly worked out in a subsequent SQL query optimization step, or by assuming that the underlying database engine can take care of this optimization.

In our context though, we cannot assume that the target query can be optimized nor that the database query engine is capable of doing it. We therefore

consider SPARQL filters at the earliest stage: function $trans_m$ pushes SPARQL filters down into the translation of each inner query in order to return only necessary intermediate results.

Let us consider a SPARQL filter f as a conjunction of n conditions (n $\geq$ 1): $C_1 \wedge ... \ C_n$. Function *sparqlCond*, formally defined in [22], discriminates between these conditions with regards to two criteria:
(i) A condition C_i is pushed into the translation of triple pattern tp if all variables of C_i show up in tp, *e.g.* a condition involving variables $?x$ and $?y$ is pushed into the translation of tp only if tp involves at least $?x$ and $?y$.
(ii) A condition C_i is part of the abstract FILTER operator if at least one variable of C_i is shared by several triple patterns, *e.g.* if C_i contains variable $?x$, and variable $?x$ also shows up in two different triple patterns, then C_i is in the condition of the abstract FILTER operator.
Note that both criteria are not exclusive: a condition may simultaneously show up in the translation of a triple pattern and in the FILTER abstract operator.

Example. SPARQL query Q_2, depicted in Listing 1.3, contains the graph pattern gp_2 that consists of three triple patterns tp_1, tp_2 and tp_3, and a filter consisting of the conjunction of two conditions c_1 and c_2:

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

Listing 1.3. SPARQL query Q_2

Let us compute function *sparqlCond* for each triple pattern:
- tp_1 has two variables, $?x$ and *?mbox*. No condition involves both variables, but c_1 involves *?mbox* and has no other variable, thereby c_1 matches criterion (i) for tp_1. Condition c_2 involves *?x* but it also involves $?y$ that is not in tp_1. Hence, c_2 does not match criterion (i) for tp_1, and `sparqlCond(tp1, c1 ∧ c2) = c1`.
- tp_2 has one variable, $?y$, and no condition involves only $?y$. Hence, no condition can be pushed into the translation of tp_2, denoted `sparqlCond(tp2, c1 ∧ c2) = true`.
- tp_3 has two variables $?x$ and $?y$, and only condition c_2 involves them both. Hence, only c_2 matches criterion (i) for tp_3 and `sparqlCond(tp3, c1 ∧ c2) = c2`.
- Lastly, only condition c_2 involves variables shared by several triples patterns: $?x$ and $?y$. Thus, only c_2 matches criterion (ii), which entails the generation of the abstract filter `FILTER(c2)`.

As a result, gp_2 is rewritten into the following abstract query:

```
trans_m(gp2, c1 ∧ c2) = transTP_m(tp1, c1)
  INNER JOIN transTP_m(tp2, true) ON {}
  INNER JOIN transTP_m(tp3, c2) ON {?x,?y}
  FILTER(c2)
```

Dealing with the LIMIT solution modifier. Similar to the case of SPARQL filters, the common bottom-up approach of SQL rewriting methods consists in

rewriting a LIMIT into an encapsulating query. Thus, again, sub-queries may return unnecessary large intermediate results. Therefore, function $trans_m$ pushes the LIMIT value down into the translation of each triple pattern using the limit argument l, initialized to ∞. During the parsing of the graph pattern by function $trans_m$, the limit argument is updated according to the graph pattern encountered. Below, we elaborate on some of the situations tackled in Def. 2:

- In a graph pattern *P LIMIT l'*, the smallest limit is kept, hence the *min(l, l')* in ***trans***$_m$*(gp, f, min(l, l'))*.
- In a graph pattern *P FILTER f'*, we cannot know in advance how many results will be filtered out by the FILTER clause. Consequently, we have to run the query with no limit and apply the filter afterward. Hence the ∞ argument in ***trans***$_m$*(P, f* $\wedge$ *f', ∞)* ***FILTER sparqlCond****(...)* ***LIMIT*** *l*.
- Similarly, in the case of an inner or left join, we cannot know in advance how many results will be returned. Consequently, the left and right queries alike are run with no limit first, the join is computed, and only then can we limit the number of results. Hence the ∞ argument in the expressions:
trans$_m$*(*P_1*,f,∞) ...* ***INNER JOIN trans***$_m$*(*P_2*,f,∞) ...* ***LIMIT*** *l*.

Dealing with other solution modifiers. For the sake of simplicity, we do not describe in further details the management of SPARQL solution modifiers OFFSET, ORDER BY and DISTINCT. Let us simply mention that they are managed in the very same way as the SPARQL FILTER clause and LIMIT solution modifier, *i.e.* as additional parameters of the $trans_m$ and $transTP_m$ functions, and additional operators of the abstract query language.

4.2 Binding xR2RML Mappings to Triple Patterns

An important step in the rewriting process consists in figuring out which of the mappings are good candidates to answer the SPARQL query. More precisely, for each triple pattern *tp* of the SPARQL graph pattern, we must figure out which mappings can possibly generate triples that match *tp*. We call this the *triple pattern binding*[7], defined in Def. 3:

Definition 3. ***Triple Pattern Binding.***
Let m be an xR2RML mapping graph consisting of a set of xR2RML mappings, and tp be a triple pattern. A mapping M $\in$ *m is* ***bound*** *to tp if it is likely to produce triples that match tp. A* ***triple pattern binding*** *is a pair (tp, MSet) where MSet is the set of mappings of m that are bound to tp.*

Function $bind_m$ (Def. 4) determines, for a graph pattern *gp*, the bindings of each triple pattern of *gp*. It takes into account join constraints implied by shared

[7] We adapt the *triple pattern binding* proposed by Unbehauen et al. in [38], and we assume that xR2RML mappings are *normalized* in the sense defined by [32], i.e. they contain 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`

variables and by cross-references defined in the mapping (xR2RML referencing-object map), 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 in [38] 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 compatibility between a term map and a SPARQL filter. In this section we give a detailed insight into these functions. A formal definition is provided in [22].

Definition 4. ***Binding xR2RML mappings to triple patterns (bind$_m$).*** *Let m be a set of xR2RML mappings, gp be a well-designed graph pattern, and f be a SPARQL filter. Let M.sub, M.pred and M.obj respectively denote the subject map, the predicate map and the object map of an xR2RML mapping M.*
We denote by $\textbf{bind}_m(gp, f)$ *the set of triple pattern bindings of "gp FILTER f" under m, and we denote by* $\textbf{bind}_m(gp)$ *the result of* $\textbf{bind}_m(gp, true)$.
Function $\textbf{bind}_m(gp, f)$ *is defined recursively as follows:*

- *if gp consists of a single triple pattern tp,* $\textbf{bind}_m(gp, f)$ *is the pair (tp, MSet) where* $MSet = \{M \mid M \in m \wedge \textbf{compatible}(M.sub, tp.sub, f) \wedge \textbf{compatible}(M.pred, tp.pred, f) \wedge \textbf{compatible}(M.obj, tp.obj, f)\}$
  where ***compatible*** *verifies the compatibility between a term map, a triple pattern term and a SPARQL filter*
- *if gp is (P1 AND P2),* $\textbf{bind}_m(gp, f) = \textbf{reduce}(\textbf{bind}_m(P1, f), \textbf{bind}_m(P2, f)) \cup \textbf{reduce}(\textbf{bind}_m(P2, f), \textbf{bind}_m(P1, f))$
  where ***reduce*** *utilizes dependencies between graph patterns to reduce their bindings*
- *if gp is (P1 OPTIONAL P2),* $\textbf{bind}_m(gp, f) = \textbf{bind}_m(P1, f) \cup \textbf{reduce}(\textbf{bind}_m(P2, f), \textbf{bind}_m(P1, f))$
- *if gp is (P1 UNION P2),* $\textbf{bind}_m(gp, f) = \textbf{bind}_m(P1, f) \cup \textbf{bind}_m(P2, f)$
- *if gp is (P FILTER f'),* $\textbf{bind}_m(gp, f) = \textbf{bind}_m(P, f \wedge f')$

Function *compatible* checks whether a term map is compatible with (i) a term of a triple pattern and (ii) a SPARQL filter, so as to rule out incompatible associations. When the triple pattern term is constant (literal, IRI or blank node), incompatibilities may occur when its type does not mach the term map type (*e.g.* when the triple pattern term is a literal whereas the term map produces IRIs). Incompatibilities may also occur for literals when language tags or data types do not match. When the triple pattern term is a variable, incompatibilities may arise from unsatisfiable SPARQL filters. These situations pertain to type constraints expressed using SPARQL functions *isIRI*, *isLiteral* or *isBlank*, as well as language and data type constraints expressed using functions *lang*, *langMatches* and *datatype*. For instance, if variable $?v$ is associated with a term map that produces literals, the SPARQL filter *isIRI(?v)* can never be satisfied, which ensures that the association is invalid. We provided a formal definition of function *compatible* in [23].

Function *reduce* uses the variables shared by two triple patterns to detect unsatisfiable join constraints, and accordingly to reduce the set of mappings bound to each triple pattern. For instance, let us consider two triple patterns tp_1 and tp_2 that have a shared variable *?v*. Mapping M_1 is bound to tp_1 and mapping M_2 is bound to tp_2. If the term map associated to *?v* in M_1 generates literals whereas the term map associated to *?v* in M_2 generates IRIs, we say that the term maps are incompatible. Consequently, function *reduce* rules out M_1 from the bindings of tp_1 and M_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.

Running Example. Let us consider query Q_2 depicted in Listing 1.3. 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 mapping `<#Mbox>`. The subject and object of tp_1 are both variables, and the constant object of tp_2 (<mailto:john@foo.com>) is compatible with the object map of `<#Mbox>`. Hence, `<#Mbox>` is bound to both triple patterns:

$bind_m$(tp$_1$, c$_1$ ∧ c$_2$) = (tp$_1$, {<#Mbox>})

$bind_m$(tp$_2$, c$_1$ ∧ c$_2$) = (tp$_2$, {<#Mbox>})

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

$bind_m$(tp$_3$, c$_1$ ∧ c$_2$) = (tp$_3$, {<#Knows>}).

Let us consider the join constraint implied by variable ?y:

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

?y is the subject in tp_2 that is bound to `<#Mbox>`, ?y is thereby associated to `<#Mbox>`'s subject map. ?y is also the object in tp_3 that is bound to `<#Knows>`, ?y is thereby associated to `<#Knows>`'s object map. Therefore, the expression

reduce($bind_m$(tp$_2$, c$_1$ ∧ c$_2$), $bind_m$(tp$_3$, c$_1$ ∧ c$_2$))

checks whether the subject map of `<#Mbox>` is compatible with the object map of `<#Knows>`. But since the object map of `<#Knows>` is a referencing object map whose parent is `<#Mbox>`, this amounts to check whether the subject map of `<#Mbox>` is compatible with itself, which is obvious. Consequently, the join constraint implied by variable ?y does not rule out any binding.

Similarly, 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, the set of triple pattern bindings for the graph pattern of query Q_2 is as follows:

$bind_m$(tp$_1$ AND tp$_2$ AND tp$_3$, c$_1$ ∧ c$_2$) =
(tp$_1$,{<#Mbox>}), (tp$_2$,{<#Mbox>}), (tp$_3$,{<#Knows>})

4.3 Translation of a SPARQL Triple Pattern

The last step of the rewriting towards the abstract query language consists in the translation of each triple pattern into an abstract query, under the set of xR2RML mappings bound to that triple pattern by function $bind_m$. This is

achieved by function $transTP_m$ defined in Def. 5, that may have to deal with various situations.

Definition 5. ***Translation of a SPARQL Triple Pattern into Atomic Abstract Queries (function $transTP_m$).***
Let m be an xR2RML mapping graph consisting of a set of xR2RML mappings, gp be a well-designed graph pattern, and tp a triple pattern of gp. Let l be the maximum number of query results, and f be a SPARQL filter expression. Let $getBoundM_m(gp, tp, f)$ be the function that, given gp, tp and f, returns the set of mappings of m that are bound to tp in $bind_m(gp, f)$.
We denote by $\boldsymbol{transTP}_m(tp, f, l)$ the translation, under $getBoundM_m(gp, tp, f)$, of tp into an abstract query whose results can be translated into at most l RDF triples matching "tp FILTER f". The resulting abstract query, denoted <ResultQuery> in the grammar below, is a union of per-mapping subqueries, where a subquery is either an Atomic Abstract Query or the inner join of two Atomic Abstract Queries.
As a simplification, arguments f and l may be omitted when their values are "true" and ∞ respectively.

```
<ResultQuery> ::= <SubQuery> (UNION <SubQuery>)*
<SubQuery>    ::= <AtomicQuery> |
                  <AtomicQuery> AS child INNER JOIN <AtomicQuery> AS parent
                  ON child/<Ref>=parent/<Ref>
```

Let us now give an insight into how $transTP_m$ deals with these situations.
(1) The most simple situation is encountered when a simple triple pattern tp is bound with a single xR2RML mapping M. If M has a regular object map (not a referencing object map denoting a cross-reference), then tp translates into an *atomic abstract query*. We will define the concept of atomic abstract query further on in this section. At this point, let us just notice that it is an abstract query obtained by matching the terms of a triple pattern with their respective term maps in a mapping.
(2) If the mapping M denotes a cross-reference by means of a referencing object map, *.i.e.* it refers to another mapping for the generation of object terms, then the result of $transTP_m$ is the INNER JOIN of two atomic abstract queries, denoted:

```
<AtomicQuery1> AS child INNER JOIN
<AtomicQuery2> AS parent ON
child/childRef=parent/parentRef
```

where `childRef` and `parentRef` denote the values of properties `rr:child` and `rr:parent` respectively.
(3) We have seen, in the definition of $bind_m$, that several mappings may be bound to a single triple pattern tp, each one may produce a subset of the RDF triples that match tp. In such a situation, $transTP_m$ translates tp into a UNION of per-mapping atomic abstract queries.

Interestingly enough, we notice that INNER JOINs may be implied either by shared SPARQL variables (Def. 2) or cross-references denoted in the mappings (situation (2) described above). Similarly, UNIONs may arise either from the

SPARQL UNION operator (Def. 2) or the binding of several mappings to the same triple pattern (situation (3) described above).

Due to size constraints, we do not go through the full algorithm of $transTP_m$ in this paper, however the interested reader is referred to [22] for a comprehensive description.

Atomic Abstract Query. An atomic abstract query consists of four parts, denoted by {*From, Project, Where, Limit*}. We now describe these components and the way they are computed by function $transTP_m$.

- ***From***. The *From* part provides the concrete query that the abstract query relies on. It contains the logical source of an xR2RML mapping, that consists of the `xrr:query` or `rr:tableName` properties, an optional iterator (property `rml:iterator`) and the optional `xrr:uniqueRef` property. With the example of query Q_2 (Listing 1.3), the *From* part for tp_1 simply consists of the logical source of `<#Mbox>: db.people.find({'emails':{$ne: null}})`.

- ***Project***. Traditionally, the projection part of a database query restricts the set of attributes that must be returned in the query response. In relational algebra, this is denoted by the projection operator π: $\pi_{a_1,...a_n}(R)$ denotes the tuple obtained when the attributes of tuple R are restricted to the set $\{a_1, ...a_n\}$. Similarly, the *Project* part of an atomic abstract query is a set of xR2RML references. For each variable in the triple pattern, the xR2RML references in the term map matched with that variable are projected. In our running example, the subject and object of tp_1 are *?x* and *?mbox1*. They are matched with the subject and object maps of mapping `<#Mbox>`. Thus, the corresponding xR2RML references within these subject map and object map 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 accommodate databases that do not support joins. In the relational database case, these projections would be useless since the database can compute the join internally. But the abstract query must accommodate any target database, hence the systematic projection of joined references.

- ***Where***. The *Where* part is a set of conditions about xR2RML references. They are produced by matching each term of a triple pattern *tp* with its corresponding term map in mapping *M*: the subject of *tp* is matched with *M*'s subject map, the predicate with *M*'s predicate map and the object with *M*'s object map. Additional conditions are entailed from the SPARQL filter *f*. In [22], 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 term in the triple pattern (IRI or literal) is turned into an equality condition on the xR2RML reference corresponding to that 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)*.

```
trans_m(gp_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(?x != ?y)

transTP_m(tp_1, c_1) =
   { 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(tp_2, true) =
   { From:    {"db.people.find({'emails': {$ne: null}})"},
     Project: {$.id AS ?y},
     Where:   {isNotNull($.id), equals($.emails.*,"john@foo.com")}}

transTP_m(tp_3, c_2) =
   { From:    {"db.people.find({'contacts':{$size: {$gte:1}}})"},
     Project: {$.id AS ?x, $.contacts.*},
     Where:   {isNotNull($.id), isNotNull($.contacts.*),
               sparqlFilter(?x != ?y)} AS child
   INNER JOIN
   { From:    {"db.people.find({'emails':{$ne: null}})" },
     Project: {$.emails.*, $.id AS ?y},
     Where:   {isNotNull($.emails.*), isNotNull($.id),
               sparqlFilter(?x != ?y)} AS parent
   ON child/$.contacts.* = parent/$.emails.*
```

Listing 1.4. Rewriting of the graph pattern gp_2 of query Q_2 (Listing 1.3) into an abstract query

Running Example. In the case of query Q_2 (Listing 1.3), triple pattern tp_2 is matched with mapping `<#Mbox>`. It has the variable $?y$ in the subject position, which entails an *isNotNull* condition. It also has a constant term in the object position, which entails an *equals* condition. Finally, 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 graph pattern gp_2 of SPARQL query Q_2 into the abstract query depicted in Listing 1.4.

4.4 Abstract Query Optimization

At this point, the method we have exposed translates a SPARQL graph pattern into an effective abstract query, *i.e.* that preserves the semantics of the SPARQL query. Yet, shortcomings such as unnecessary complexity or redundancy may lead to the generation of inefficient queries, and consequently yield poor performances. Although we may postpone the query optimization to the translation into a concrete query language, it is beneficial to figure out which optimizations can be done at the abstract query level first, and leave only database-specific optimizations to the subsequent stage.

SPARQL-to-SQL methods proposed various SQL query optimizations such as [39,32,14]. In this section, we review some of these techniques, referring to the

```
trans_m(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($.contacts.*),
                sparqlFilter(?x != ?y)} AS child
    INNER JOIN
    { From:    {"db.people.find({'emails':{$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.* )
    ON {?x,?y}
    FILTER(?x != ?y)
```

Listing 1.5. Optimization of $\text{trans}_m(\text{gp}_2)$ (Listing 1.4) by self-join elimination

terminology defined in [39]. We show how these optimizations can be adapted to fit in the context of our abstract query language. In particular, we show that our translation method implements some of these optimizations by construction. In addition, we propose a new optimization, the *Filter Propagation*, that, to our knowledge, was not proposed in any SPARQL-to-SQL rewriting method.

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. Typically, when the translation of an R2RML template relies on the SQL string concatenation, a SPARQL query can ben rewritten into something like this:

```
SELECT... FROM... WHERE
  ('http://domain/' || TABLE.ID) = 'http://domain/1'
```

Such a query returns the expected results but is likely to perform very poorly: due to the concatenation, the query evaluation engine cannot take advantage of existing database indexes. Conversely, a much more efficient query would be:

```
SELECT ('http://domain/' || TABLE.ID)... FROM...
  WHERE TABLE.ID = 1
```

In our approach, equality conditions apply to xR2RML references rather than on the template-generated values, 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 lowers 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 enforced by construction by the *sparqlCond* function: relevant SPARQL conditions are pushed down, as much as possible, in the translation of individual triple patterns.

Self-Join Elimination. A self-join may occur when several mappings share the same logical source. This can lead to several triple patterns being translated

into atomic abstract queries with the same *From* part. The *Self-Join Elimination* consists in merging the criteria of several atomic queries into a single equivalent query. In Listing 1.4, the atomic query in $\texttt{transTP}_m\texttt{(tp}_2\texttt{, true)}$ and the second atomic query in $\texttt{transTP}_m\texttt{(tp}_3\texttt{, c}_2\texttt{)}$ have the same *From* part and project the same JSONPath expression as variable `?y`. Using joins commutativity, those two queries can be merged into a single one depicted in the third atomic abstract query in Listing 1.5[8].

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 mappings are bound (in function $transTP_m$). Analogously to the *Self-Join Elimination*, a union of several atomic abstract queries sharing the same logical source can be merged into a single query when they have the same *From* part.

Constant Projection. The *Constant Projection* optimization detects cases where the only projected variables in the SPARQL query are matched with constant values in the bound mappings. In the relational database context, it has been referred to as the *Projection Pushing* optimization [39]. Let us consider the example query below:

```
SELECT DISTINCT ?p WHERE {?s ?p ?o}.
```

In a naive approach, all mappings are bound to the triple pattern $?s\ ?p\ ?o$. Hence, the resulting abstract query is a union of the atomic queries derived from all the possible mappings. In other words, this query will materialize the whole database before it can provide an answer. Very frequently, xR2RML predicate maps are constant-valued: the predicate is not computed from a database value, on the contrary it is defined statically in the mapping. This is typically the case in our running example that has only constant predicate maps (values of property `rr:predicate`: `foaf:knows` and `foaf:mbox` (Listing 1.1). In such cases, given that the SPARQL query retrieves only DISTINCT values of the predicate variable $?p$, no query needs to be run against the database at all: it is sufficient to collect the distinct constant values that variable $?p$ can be matched with. More generally, this optimization checks if the variables projected in the SPARQL query are matched with constant term maps. If this is verified, the SPARQL query is rewritten such that the values of the projected variables be provided as an inline solution sequence using the SPARQL 1.1 VALUES clause. Using the mapping graph of our running example, we would rewrite the query in this way:

```
SELECT DISTINCT ?p WHERE
  { VALUES ?p ( foaf:mbox foaf:knows )}
```

Filter Propagation. We identified another type of optimization that was not implemented in the SPARQL-to-SQL context. This optimization applies to the inner join or left outer join of two atomic queries, and seeks to narrow down one of the joined queries by propagating filter conditions from the other query. In an inner join, if the two queries have shared variables, then *equals* and *isNotNull*

[8]Note that for a self-join elimination to be safe, additional conditions must be met, that we do not detail here.

```
AND(<exp1>, <exp2>, ...)      → $and:[<exp1>,<exp2>,...]
OR(<exp1>, <exp2>, ...)       → $or:[<exp1>,<exp2>,...]
WHERE(<JavaScript exp>)       → $where:'<JavaScript exp>'
ELEMMATCH(<exp1>,<exp2>...)   → $elemMatch:{<exp1>,<exp2>...}
FIELD(p1) ... FIELD(pn)       → "p1. ... .pn":
SLICE(<exp>, <number>)        → <exp>:{$slice:<number>}
COND(equals(v))               → $eq:v
COND(isNotNull)               → $exists:true, $ne:null
EXISTS(<exp>)                 → <exp>:{$exists:true}
NOT_EXISTS(<exp>)             → <exp>:{$exists:false}
COMPARE(<exp>, <op>, <v>)     → <exp>:{<op>:<v>}
NOT_SUPPORTED                 → ∅
UNION(<query1>, <query2>...) Same semantics as OR, although OR is processed
                              by the NoSQL engine whereas UNION is processed
                              by the query processing engine
```

Listing 1.6. Abstract representation of a MongoDB query and translation to a concrete query string. `<op>` stands for one of the MongoDB comparison operators: $\$eq$, $\$ne$, $\$lt$, $\$lte$, $\$gt$, $\$gte$, $\$size$ and $\$regex$.

conditions of one query on those shared variables can be propagated to the other query. In a left join, propagation can happen only from right to left query since null values must still be allowed in the right query.

5 Application to the MongoDB NoSQL Database

In the previous section, we have exhibited an abstract query model and a method to translate a SPARQL graph pattern into an optimized abstract query, relying on the xR2RML mapping of a target database to RDF. We now want to illustrate the effort it takes to translate from the abstract query language towards a concrete query language with a somewhat different expressiveness.

To this end, we consider the MongoDB NoSQL database. Its JSON-based data model and its query language differ greatly from SQL-based systems for which many rewriting works have been proposed. Hence, we believe that it should provide an interesting illustration of our method. Besides, MongoDB has become a popular NoSQL actor in recent years. It is provided as a service by major cloud service providers and tends to become common within the scientific community, suggesting that it is increasingly adopted as a commonplace database.

In this section, we first glance at the MongoDB query language, and we describe an abstract representation of MongoDB queries (section 5.1). Then, we show that the translation from the abstract query language towards MongoDB is made challenging by the expressiveness discrepancy between the two languages (section 5.2) and we describe a complete method to achieve this. Finally, we summarize the whole SPARQL-to-MongoDB process orchestration, from the SPARQL graph pattern translation until the generation of the RDF triples that match this graph pattern (section 5.3).

5.1 The MongoDB Query Language

MongoDB comes with a rich set of APIs to allow applications to query a database in an imperative way. In addition, the MongoDB interactive interface defines a JSON-based declarative query language consisting of two query methods. The *find* method retrieves documents matching a set of conditions and returns a cursor to the matching documents. Optional modifiers amend the query to impose limits and sort orders. Alternatively, the *aggregate* method allows for the definition of processing pipelines: each document of a collection passes through each stage of a pipeline thereby creating a new collection. This allows for a richer expressiveness but comes with a higher resource consumption that entails less predictable performances. Thus, as a first approach, this work considers the *find* query method, hereafter called the *MongoDB query language*.

The MongoDB *find* query method takes two arguments formatted as JSON documents. The first argument describes conditions about the documents to search for. Query operators are denoted by a heading '$' character. The optional projection argument specifies the fields of the matching documents to return. For instance, the query below matches all documents with a field "emails" and returns only the "id" field of each matching document.

```
db.people.find({"emails": {$exists: true}}, {"id": true})
```

The MongoDB documentation provides a rich description of the *find* query that however lacks precision as to the formal semantics of some operators. Attempts were made to clarify this semantics while underlining some limitations and ambiguities: Botoeva et al. [7] mainly focus on the *aggregate* query and ignore some of the operators we use in our translation, such as $\$where$, $\$elemMatch$, $\$regex$ and $\$size$. On the other hand, Husson [20] describes the *find* query, yet some restrictions on the operator $\$where$ are not formalized.

Hence, in [22] we specified the grammar of the subset of the query language that we consider. We also defined an abstract representation of MongoDB queries, that allows for handy manipulation during the query construction and optimization phases. Listing 1.6 details the constructs of this representation and their equivalent concrete query string, when relevant. The NOT_SUPPORTED clause helps keep track of any location, within the query, where a condition cannot translate into an equivalent MongoDB query element. It shall be used in the last rewriting and optimization phase.

Let us consider the following abstract representation of a MongoDB query (or "abstract MongoDB query" for short):

```
AND( COMPARE(FIELD(p) FIELD(0), $eq, 10),
     FIELD(q) ELEMMATCH(COND(equals("val")) )
```

It matches all documents where "p" is an array field whose first element (at index 0) is 10, and "q" is an array field in which at least one element has value "val". Its concrete representation is:

```
$and: [ {"p.0": {$eq:10}},
        {"q": {$elemMatch: {$eq:"val"}}} ]
```

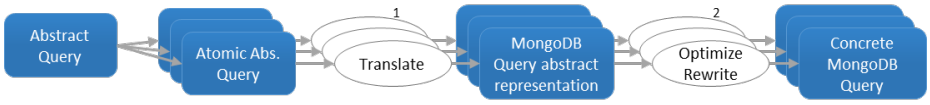

Fig. 2. Translation of atomic abstract queries into concrete MongoDB queries

5.2 Translation of an Abstract Query into MongoDB queries

Section 4 elaborated on how a SPARQL graph pattern translates into an abstract query based on xR2RML mappings. Abstract operators INNER JOIN, LEFT OUTER JOIN and UNION relate sub-queries. The lowest level of sub-queries consists of atomic abstract queries of the form {*From, Project, Where, Limit*}, that stem from the translation of individual triple patterns. The *From* part contains the logical source of a mapping bound to the triple pattern to translate. The *Project* part lists the xR2RML data element references that are projected, *i.e.* that are part of the query result. In the context of MongoDB, these xR2RML data element references are JSONPath expressions. The *Where* part is calculated by matching triple pattern terms with relevant xR2RML term maps. This generates conditions on JSONPath expressions (*isNotNull* conditions for SPARQL variables or *equals* conditions for constant triple pattern terms) and *sparqlFilter* conditions that encapsulate SPARQL filters. Finally, the *Limit* part denotes an optional maximum number of results.

To achieve a translation from the abstract query language towards the MongoDB query language, we must figure out which components of an abstract query have an equivalent MongoDB rewriting, and, conversely, which components shall be computed by the query-processing engine. Below, we analyze the possible situations.

- **Inner and left outer joins**. MongoDB *find* queries do not support joins. Consequently, there does not exist any MongoDB query that would be equivalent to the INNER JOIN and LEFT OUTER JOIN operators. These operators need to be processed by the query-processing engine by joining the RDF triples generated for both sub-queries.

- **UNION**. The rewriting of the UNION operator depends on the graph patterns to which it applies. Let us consider the following SPARQL graph pattern, where tp_n is any triple pattern: { tp_1. tp_2. } UNION { tp_3. tp_4. } Each member of the union translates into an INNER JOIN. Since joins cannot be processed within MongoDB, the outer UNION operator cannot be processed within MongoDB either. The issue occurs likewise as soon as one of the members is either an INNER JOIN or LEFT OUTER JOIN. Under some circumstances, a UNION operator may be translated into the MongoDB $*or* operator. Yet, the MongoDB language definition imposes specific restrictions as to how operators can be nested. Consequently, in a first approach, we always shift the processing of the UNION abstract operator to the query-processing engine. Further works could attempt to characterize more specifically the situations where a UNION can be processed within MongoDB.

- **FILTER and LIMIT**. In section 4, we showed that the FILTER and LIMIT SPARQL solution modifiers are pushed down into relevant atomic abstract queries (as *sparqlFilter* conditions of the *Where* part or as the *Limit* part of an atomic query, respectively). When FILTER and LIMIT SPARQL clauses cannot be pushed down in atomic queries, they end up as abstract operators with the same names, FILTER and LIMIT. The latter apply to abstract sub-queries made of UNION, INNER JOIN and/or LEFT OUTER JOIN operators. Hence, given that UNION and INNER/LEFT OUTER JOIN operators are not processed within MongoDB, the FILTER and LIMIT operators cannot be processed within MongoDB either.

Ultimately, it occurs that only the atomic abstract queries can be processed within MongoDB, while other abstract operators shall be taken care of by the query-processing engine. More generally, the translation from the abstract query language towards MongoDB consists of two steps depicted in Fig. 2. In step 1 (detailed in section 5.2), the translation of each atomic abstract query towards MongoDB amounts to translate projections of JSONPath expressions (*Project* part) into MongoDB projection arguments, and conditions on JSONPath expressions (*Where* part) into equivalent abstract MongoDB queries. Several shortcomings may appear at this stage, such as unnecessary complexity or untranslatable conditions. Thus, in step 2 (detailed in section 5.2) each abstract MongoDB query is optimized and rewritten into valid, concrete MongoDB queries.

In the current status of this work, we do not consider the translation of SPARQL filters (conditions *sparqlFilter*) for the sake of simplicity. SPARQL 1.0 filters come with a broad set of conditional expressions including logical comparisons, literal manipulation expressions (string, numerical, boolean), XPath constructor functions, casting functions for additional data types of the RDF data model, and SPARQL built-in functions (*lang, langmatches, datatype, bound, sameTerm, isIRI, isURI, isBlank, isLiteral, regex*). Handling these expressions within the translation towards MongoDB would yield a significant additional complexity without changing the translation principles though. Yet, an implementation should handle them for the sake of performance and completeness.

Translation of Projections and Conditions Two functions, named *proj* and *trans*, handle the translation of the *Project* and *Where* parts of an atomic abstract query respectively. Below, we illustrate their principles on an example. The interested reader shall find their formal definition in [22].

In Listing 1.5, the third atomic abstract query is as follows (the *sparqlFilter* condition has been omitted):

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

Function *proj* converts the JSONPath expressions of the *From* part into a list of paths to be projected. In the example, expressions `$.emails.*` and `$.id` translate into their MongoDB projection counterparts: `"emails":true` and `"id":true`.

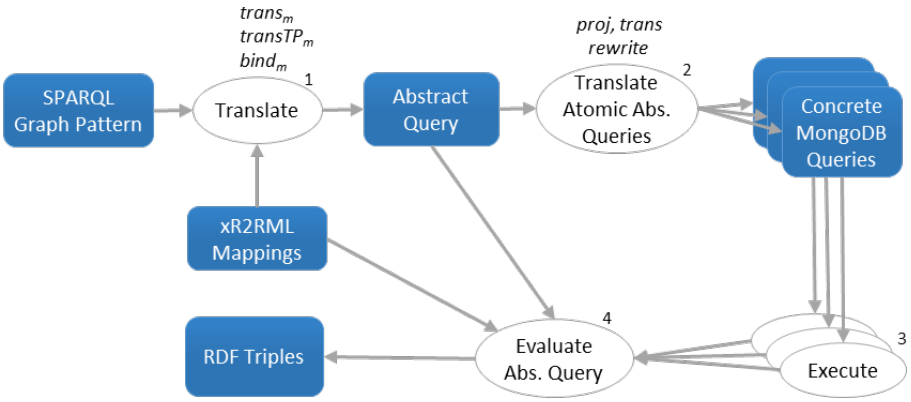

Fig. 3. Complete SPARQL-to-MongoDB Query Translation and Evaluation

Function *trans* translates a condition of the *Where* part into a MongoDB query element expressed using the abstract representation in Listing 1.6. In the example, condition `isNotNull($.emails.*)` is translated into the following abstract representation:

`FIELD(emails) ELEMMATCH(COND(isNotNull)).`

Later on, this abstract representation will be translated into an equivalent concrete query:

`"emails": {$elemMatch: {$exists:true, $ne:null}}.`

Similarly, condition `isNotNull($.id)` will be translated into: `"id": {$exists:true, $ne:null}`, and condition `equals($.emails.*,"john@foo.com")` will be translated into: `"emails": {$elemMatch: {$eq:'john@foo.com'}}`.

These conditions are used to augment the query of the *From* part, initially provided by the mapping's logical source. When we put all the pieces together, the atomic abstract query is translated into the concrete MongoDB query below, where all conditions are operands of an $*and* operator:

```
db.people.find(
 # Query argument
 { $and: [
   {"emails": {$ne:null}}, # from the From part
   {"emails": {$elemMatch: {$exists:true,$ne:null}}},
   {"id":     {$exists:true,$ne:null}},
   {"emails": {$elemMatch: {$eq:'john@foo.com'}}} ]
 },
 # Projection argument
 { "emails": true, "id": true }
)
```

Optimization and Rewriting into Concrete MongoDB Queries In the previous section, function *trans* produces abstract MongoDB queries that can be rewritten into concrete queries straightaway. Yet, this rewriting may be hindered by three potential issues:

(i) During the translation process, nested OR or AND clauses may be produced, as well as sibling WHERE clauses. Such unnecessary complexity may yield an underperforming query.
(ii) It may not be possible to translate some JSONPath expressions into equivalent MongoDB operators. This occurs with specific JSONPath array slice notations, or in JSONPath expressions assuming that the root document is an array field and not a document field (which is forbidden in MongoDB). In such cases, a NOT_SUPPORTED clause tracks the location of this failed translation.
(iii) The MongoDB $*where* operator passes a JavaScript expression or function to the query system. It provides greater flexibility than other operators, however it is valid only in the top-level query document: it cannot be used inside a nested query such as the $*elemMatch* operator. During the translation process though, function *trans* may nest a WHERE clause beneath other clauses, yielding an invalid query.

To take care of these issues, in [24] we described a post-translation function *rewrite*, depicted by step 2 in Fig. 2. First, a set of rewriting rules address issue (i) by flattening nested OR, nested AND and nested UNION clauses, and merging sibling WHERE clauses.

To address issue (ii), these rules remove NOT_SUPPORTED clauses while ensuring that the resulting query returns a superset of the valid answers: all the correct answers are returned, along with possibly incorrect answers. In turn, the transformation of this superset into RDF triples shall produce all the triples that match the SPARQL query, in addition to triples that may not match the query. The latter are ruled out during the query evaluation process by running a late SPARQL query evaluation.

A second set of rewriting rules address issue (iii) by "pulling up" WHERE clauses at the top-level query. This is notably achieved by replacing OR clauses with UNION clauses that have the same semantics but are processed differently. An OR clause represents the $*or* operator and is processed by MongoDB. Conversely, the UNION clause has no equivalent MongoDB operator: it is processed outside of MongoDB by the query processing engine. As a consequence, an abstract MongoDB query may be rewritten into a union of valid, concrete MongoDB queries.

Finally, Theorem 1 captures two key properties of the rewriting process. It has been proved in [22].

Theorem 1. *Let C be an equality or not-null condition on a JSONPath expression. Let $Q = (Q_1 \ldots Q_n)$ be the abstract MongoDB query produced by trans(C).* ***Rewritability:*** *It is always possible to rewrite Q into a query $Q' = \text{UNION}(Q'_1, \ldots, Q'_m)$ such that $\forall i \in [1, m]$ Q'_i is a valid MongoDB query, i.e. Q'_i does not contain any* NOT_SUPPORTED *clause, and a* WHERE *clause only shows at the top-level of Q'_i.*
Completeness: *Executing Q' against the database retrieves all the documents matching condition C. If Q contains at least one* NOT_SUPPORTED *clause, then Q' may retrieve additional documents that do not match condition C.*

A corollary of Theorem 1 is that, using the xR2RML mapping of a MongoDB database to RDF, we can rewrite any SPARQL 1.0 graph pattern into an abstract query whose atomic abstract queries are valid MongoDB queries or unions of valid MongoDB queries.

5.3 Complete SPARQL-to-MongoDB Query Translation and Evaluation

Fig. 3 summarizes the whole SPARQL-to-MongoDB process orchestration, from the graph pattern translation to the subsequent MongoDB queries evaluation and the production of RDF triples.

In step 1, function $trans_m$ (section 4.1) translates a SPARQL graph pattern into an abstract query under a set of xR2RML mappings denoted by m. It leverages function $transTP_m$ (section 4.3) to translate a triple pattern tp into an abstract query under the set of mappings bound to tp by function $bind_m$ (section 4.2). The resulting abstract query contains atomic abstract queries of the form {*From, Project, Where, Limit*}, combined with abstract operators INNER JOIN, LEFT OUTER JOIN, UNION, FILTER, LIMIT. The *Project* part of an atomic abstract query is a set of xR2RML references (*i.e.* JSONPath expressions for MongoDB) that must be projected. The *Where* part consists of *isNotNull*, *equals* and *sparqlFilter* conditions on JSONPath expressions. In step 2, function *proj* translates each projected JSONPath expression into a MongoDB projection argument, function *trans* translates each *isNotNull* and *equals* condition into an abstract representation of a MongoDB query (section (section 5.2), and function *rewrite* (section 5.2) optimizes and rewrites this abstract representation into a concrete MongoDB query or a union of concrete MongoDB queries.

Two steps remain, that we have not described yet. In step 3, the concrete queries are executed against the database. In step 4, the result JSON documents are translated into RDF triples according to the xR2RML mappings, then the query processing engine evaluates the abstract query by computing the INNER/LEFT OUTER JOIN, UNION, FILTER and LIMIT operators. Finally, in case one atomic abstract query contained a NOT_SUPPORTED clause, a late SPARQL evaluation is performed to rule out the RDF triples that do not match the query (as explained in section 5.2).

6 Experimentation and Evaluation

To date, to our knowledge, the method proposed in this paper and the MongoDB-enabled *ontop* software [8] are the only approaches meant to query arbitrary MongoDB documents with SPARQL. So far though, this *ontop* version is not available for test, which hinders possible performance comparison. Additionally, no benchmark similar to the Berlin SPARQL Benchmark for relational databases [6] exists so far for querying NoSQL databases with SPARQL.

Therefore, in this section, we describe a real-world use case that we used to build a test database, and we report experimental results with respect to the effectiveness and performance of our approach.

6.1 Prototype Implementation

Morph-xR2RML is the prototype implementation we developped to evaluate the effectiveness of the xR2RML mapping language and the SPARQL-to-MongoDB method proposed in this paper. It comes with connectors for the MySQL and Postgres relational databases, and for the MongoDB document store. It can process an xR2RML mapping graph in either the data materialization or the query rewriting modes.

Morph-xR2RML is available on GitHub[9] under the Apache 2.0 license, it is written in the Scala programming language. It is based on and extends the Morph-RDB[29] R2RML implementation. We performed a substantial code refactoring in order to isolate any RDB-related code into a dedicated software module. As a result, our prototype is extensible by design: supporting a new type of database amounts to create a new software module that implements a given set of interfaces, thereby encapsulating and isolating any database-specific concerns from the rest of the project code. Following this approach, we developed a connector for the MongoDB document store, to translate MongoDB JSON documents into RDF and rewrite SPARQL queries into MongoDB queries.

Morph-xR2RML relies on several open-source Java APIs, the most salient ones are listed here. Jena[10] is a well known Java framework consisting of several APIs meant to build Semantic Web Data applications. We use the Jena RDF API that helps handle RDF triples and graphs. MongoDB comes with a native Java API[11] that allows for imperative style querying only. The Jongo API[12] builds on top of it to translate a declarative MongoDB query (a *find* query in our case) into imperative code. Lastly, Jayway JsonPath[13] is a Java implementation of the JSONPath language.

The query rewriting experimentation we report in this section was conducted on a server equipped with a 3.0 GHz CPU with two physical cores, and 8 GB RAM. The MongoDB engine and the Morph-xR2RML Java virtual machine alike were running on the same server. The Java virtual machine was allowed a maximum of 4 GB memory.

6.2 Experimentation Database

TAXREF[15] is the French national taxonomic register for fauna, flora and fungus, maintained and distributed by the French National Museum of Natural History (MNHN). It is a manually curated register of all the species inventoried in metropolitan France and overseas territories, organized as a hierarchy of over 485.000 scientific names (in version 9) that mark a national and international consensus. As an example, the listing below shows a JSON excerpt from TAXREF's Web service[14], describing the common dolphin species (*Delphinus*

[9] https://github.com/frmichel/morph-xr2rml/
[10] http://jena.apache.org/
[11] https://mongodb.github.io/mongo-java-driver/
[12] http://jongo.org/
[13] https://github.com/json-path/JsonPath
[14] https://taxref.mnhn.fr/taxref-web/api/doc

delphis). Annotation `"habitat":1` states that it lives in a marine habitat, annotation `"rang":"ES"` states that the taxon belongs to the "species" taxonomical rank. Annotation `"fr":"P"` characterizes one of its biogeographical statuses: it states that *Delphinus delphis* is present in mainland France.

```
{
  "codeTaxon":"60878",
  "codeReference":"60878", "codeParent":"191591",
  "rang":"ES",
  "libelleNom":"Delphinus delphis",
  "libelleAuteur":"Linnaeus, 1758",
  "nomVernaculaire":"Dauphin commun",
  "nomVernaculaireAnglais":"Common Dolphin",
  "url":"http://inpn.mnhn.fr/espece/cd_nom/60878",
  "habitat":"1",
  "fr":"P",
  (...)
}
```

We are involved in an on-going collaboration with TAXREF experts from MNHN, aimed to publish TAXREF on the Web of Data as a SKOS thesaurus [9]. In this context, we imported into a MongoDB database the JSON representation of TAXREF v9.0, wherein each of the 485.189 JSON documents accounts for one scientific name, may it be a taxon reference of synonymous name. Listing 1.7 exemplifies the SKOS modeling with taxon *Delphinus delphis* and its synonym *Delphinus vulgaris*. The taxon is represented as a SKOS concept (line 10). The `skos:broader` property models the relationships towards the parent taxon in the classification (line 13), *i.e.* genus *Delphinus* in this example. The taxon reference and synonymous names are represented as SKOS-XL labels (lines 23-33), referred to with properties `skosxl:prefLabel` and `skosxl:altLabel` respectively (lines 14-15). The taxonomical rank, habitat and bio-geographical status are properties of the SKOS concept (lines 16-21), while the authorities and vernacular names are properties of SKOS labels (lines 25-27 and 31-33).

Leveraging this existing database, we set up an experimentation of the SPARQL-to-MongoDB query rewriting. In the next section, we shortly describe the xR2RML mappings designed for the experimentation.

```
1  @prefix txrp:   <http://inpn.mnhn.fr/taxref/properties/> .
2  @prefix txrbgs: <http://inpn.mnhn.fr/taxref/bioGeoStatus#> .
3  @prefix nt:     <http://purl.obolibrary.org/obo/ncbitaxon#> .
4  @prefix dwc:    <http://rs.tdwg.org/dwc/terms/> .
5  @prefix txn:    <http://lod.taxonconcept.org/ontology/txn.owl#> .
6  @prefix dct:    <http://purl.org/dc/elements/1.1/> .
7  @prefix skos:   <http://www.w3.org/2004/02/skos/core#> .
8  @prefix skosxl: <http://www.w3.org/2008/05/skos-xl#> .
9
10 <http://inpn.mnhn.fr/taxref/9.0/taxon/60878> a skos:Concept ;
11   skos:inScheme        <http://inpn.mnhn.fr/taxref/9.0/Taxref> ;
12   skos:note            "Delphinus delphis" ;
13   skos:broader         <http://inpn.mnhn.fr/taxref/9.0/taxon/191591> ;
14   skosxl:prefLabel     <http://inpn.mnhn.fr/taxref/label/60878> ;
15   skosxl:altLabel      <http://inpn.mnhn.fr/taxref/label/577834> ;
16   txrp:habitat         <http://inpn.mnhn.fr/taxref/habitat#Marine> ;
17   nt:has_rank          <http://inpn.mnhn.fr/taxref/taxrank#Species> ;
18   txrp:bioGeoStatusIn  [
19     rdfs:label            "Metropolitan France" ;
20     dct:spatial           <http://sws.geonames.org/3017382/> ;
21     dwc:locationId        "TDWG:FRA; WOEID:23424819" ;
22     dwc:occurrenceStatus  txrbgs:P ] .
```

```
23
24 <http://inpn.mnhn.fr/taxref/label/60878> a skosxl:Label ;
25   txrp:isPrefLabelOf    <http://inpn.mnhn.fr/taxref/9.0/taxon/60878> ;
26   txn:authority         "Linnaeus, 1758" ;
27   txrp:vernacularName   "Common Dolphin"@en, "Dauphin commun"@fr ;
28   skosxl:literalForm    "Delphinus delphis" .
29
30 <http://inpn.mnhn.fr/taxref/label/577834> a skosxl:Label ;
31   txrp:isAltLabelOf     <http://inpn.mnhn.fr/taxref/9.0/taxon/60878> ;
32   txn:authority         "Lacepede, 1804" ;
33   txrp:vernacularName   "Common Dolphin"@en, "Dauphin commun"@fr ;
34   skosxl:literalForm    "Delphinus vulgaris" .
```

Listing 1.7. SKOS representation of the *Delpinus delphis* taxon

6.3 Experimentation xR2RML Mapping Graph

The xR2RML mapping graph designed to generate the TAXREF-based SKOS thesaurus is provided in the xR2RML GitHub repository[15]. It consists of 90 mappings, a somewhat high number that spawns from the distance between the internal structure of TAXREF JSON documents and the targeted SKOS modeling. We illustrate this distance with an example.

Habitats are coded in TAXREF with integer values, *e.g.* value '1' represents the marine habitat, '2' represents fresh water, etc. Translating the marine habitat into URI `http://inpn.mnhn.fr/taxref/habitat#1` would be straightforward using a template that would append the value read from the database to `http://inpn.mnhn.fr/taxref/habitat#`. A single mapping would be sufficient to generate the triples related to all types of habitat. However, our modeling targets the generation of more meaningful URIs that cannot be generated by a template, e.g. `http://inpn.mnhn.fr/taxref/habitat#Marine`; instead, we must write a mapping whose query filters only taxa with habitat '1':

```
<#TM_Habitat_Marine>
 xrr:logicalSource [ xrr:query """db.taxrefv9.find(
    {$where: 'this.codeTaxon==this.codeReference',
     'habitat':'1'} )""" ];
 rr:subjectMap <#SM_Taxon>;
 rr:predicateObjectMap [
  rr:predicate txrfp:habitat;
  rr:objectMap [
   rr:constant
    <http://inpn.mnhn.fr/taxref/habitat#Marine>;
   rr:termType rr:IRI ]].
```

Such a mapping must be written for each of the 8 habitat values. A similar situation is observed for the 48 taxonomical ranks and 30 bio-geographical statuses, that all comme with dedicated mappings.

6.4 Experimentation Results

In section 5, we showed that atomic abstract queries can be translated into equivalent MongoDB queries, but other operators of the abstract query language (INNER JOIN, LEFT OUTER JOIN, UNION) must be computed by the

[15] xR2RML mapping graph for TAXREF v9: https://github.com/frmichel/morph-xr2rml/blob/master/morph-xr2rml-dist/example_taxref/xr2rml_taxref_v9.ttl

Table 1. Execution time of SPARQL queries with one triple pattern

Q. Id	Query semantics and SPARQL triple pattern	No. results	Exec. time ± std dev. (ms)	Exec. time per result (ms)
Q0	*Find the reference name for taxon 60587* ?t skosxl:prefLabel <http://inpn.mnhn.fr/taxref/label/60587>	1	451 ± 36	451.00
Q1	*Get synonyms of taxon 95372* <http://inpn.mnhn.fr/taxref/9.0/taxon/95372> skosxl:altLabel ?a	164	522 ± 14	3.18
Q2	*Get all bio-geographical statuses in St Pierre et Miquelon* ?bgs dct:spatial <http://sws.geonames.org/3424932/>	4835	4.056 ± 65	0.84
Q3	*Get all bio-geographical statuses in Guadeloupe* ?bgs dct:spatial <http://sws.geonames.org/3579143/>	17956	9665 ± 45	0.54
Q4	*Get all bio-geographical statuses in New Caledonia* ?bgs dct:spatial <http://sws.geonames.org/2139685/>	35703	17289 ± 78	0.48
Q5	*Get bio-geographical statuses in mainland France* ?bgs dct:spatial <http://sws.geonames.org/3017382/>	128018	61645 ± 671	0.48
Q6	*Get all taxa (that are SKOS concepts)* ?c a skos:Concept	227224	108508 ± 459	0.48

query-processing engine, *i.e.* Morph-xR2RML. Therefore, a first series of tests aimed to assess the performance of Morph-xR2RML with a SPARQL query consisting of a single triple pattern, bound to exactly one mapping and producing a single MongoDB query (section 6.4). In a second series of tests, we measured the completion time of SPARQL queries involving joins and/or unions, and we compared them to the time needed for a single triple pattern. Furthermore, we measured the gain obtained by performing optimizations at the level of the abstract query (section 6.4).

Processing a Single Triple Pattern To measure the performance of Morph-xR2RML in the case of a single triple pattern translated into a single MongoDB

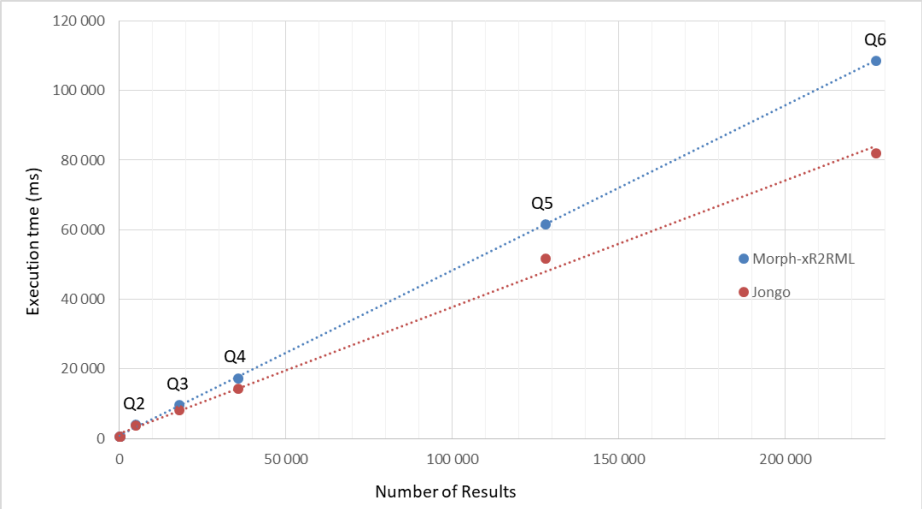

Fig. 4. Average query processing time as a function of the number of results. Dotted lines represent the linear regression lines of both series.

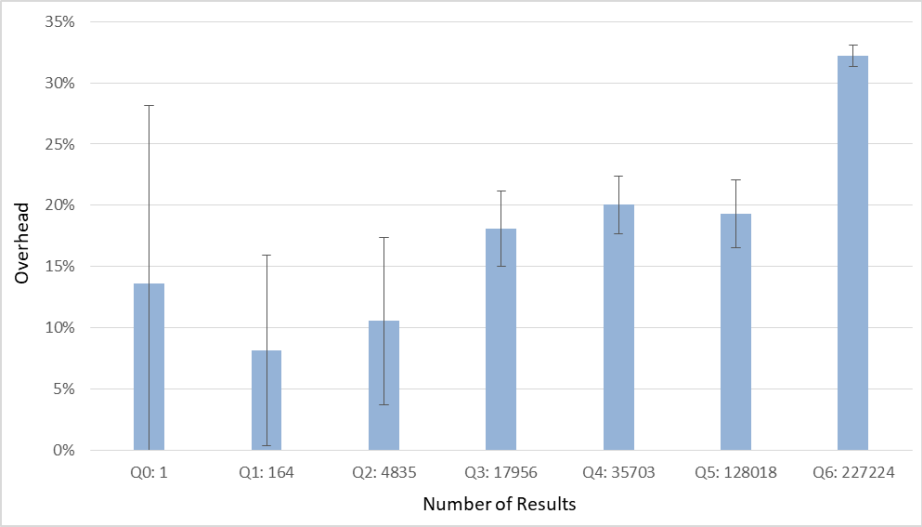

Fig. 5. Processing time overhead imposed by Morph-xR2RML, compared to a direct database query. The overhead comprises rewriting the SPARQL query and translating the MongoDB results into RDF triples

query, we selected seven SPARQL SELECT queries (Q0 to Q6) tailored to produce an increasing number of results: from 1 result in Q0 to 227,224 results in Q6. In each case, one JSON document yields one RDF triple. Table 1 lists each

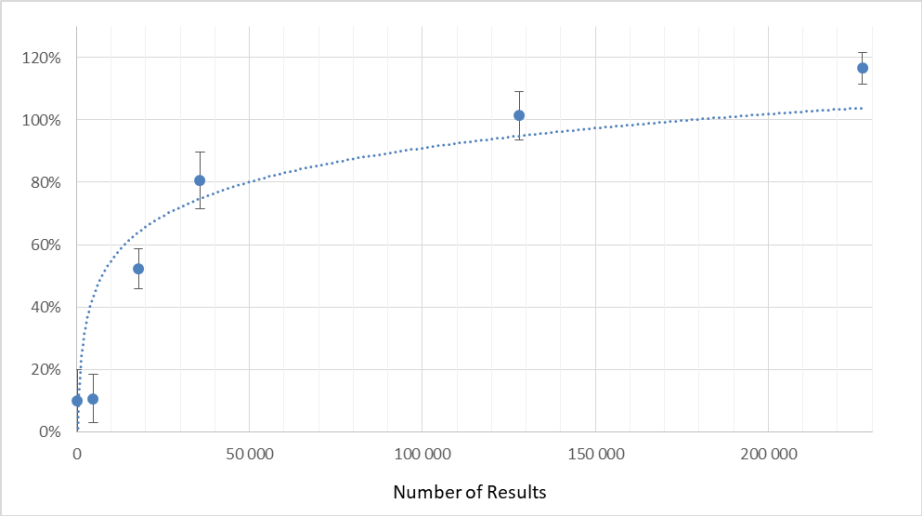

Fig. 6. Overhead of querying MongoDB through the Jongo API compared to a direct query through MongoDB's Java API

query along with the corresponding triple pattern and semantics, the number of results it retrieves from the database, and the average time it took to process the query (the query processing spans the SPARQL query rewriting, the query evaluation against MongoDB and the RDF triples generation). For each query, 10 measures were performed: we report the average value and standard deviation. The last column gives the average processing time per query result, that converges towards 0.48ms.

Figure 4 depicts the average query processing time (fourth column of Table 1) as a function of the number of results (blue line). Since Morph-xR2RML relies on the Jongo API to process a MongoDB query, we also measured the time needed by Jongo to parse the query, pass it on to MongoDB and retrieve the results from MongoDB. Red dots represent the measures when simply querying MongoDB with Jongo, while blue dots represent the measures of the whole process executed by Morph-xR2RML.

The distance between the two lines gives an estimation of the overhead imposed by Morph-xR2RML to rewrite the query and generate the triples. Figure 5 depicts this overhead. The confidence for Q0 and Q1, and to some extend for Q2, is very low as attested by the large error bars. Indeed, materializing a few triples is barely measurable (<1ms for Q0, and in the order of 30ms for Q1), such that the measure is very sensitive to environment variations. Conversely, the confidence for Q3 to Q6 is quite high. Q3, Q4 and Q5 show a similar overhead of approximately 19%. Although we could expect the overhead percentage to be constant with higher numbers of results, it reaches 32% for Q6. A detailed analysis shows that the difference lies in the time needed to generate the RDF triples. Compared to Q5, the number of results in Q6 increases by 77% while

the materialization time increases by 120%. The variable term in Q3, Q4 and Q5 is a blank node whereas it is a URI in Q6. A tentative explanation is that Morph-xR2RML may be faster when producing blank nodes than when producing URIs, unless this difference lies in the Jena API on which Morph-xR2RML relies to handle RDF triples. Further works should consider using more substantial databases to assess this difference with more precision. In any case, the processing performed by Morph-xR2RML adds no more than a 30% overhead to the time needed to query the database and retrieve the results.

Yet, waiting 10 seconds to get 18000 results (query Q3) can be considered surprisingly long compared to native RDF triple stores. To investigate this question, we compared the time it takes to run a query (i) through the Jongo API (the case of Morph-xR2RML) and (ii) directly through MongoDB's own Java API. The results are presented in Figure 6. Surprisingly, they attest that, while Jongo is efficient for few results (in the order of 100), it entails a significant overhead for larger results: 116% overhead for query Q6 (*i.e.* using Jongo more than doubles the query time). Jongo's authors argue that the library is almost as fast as querying MongoDB directly, under the assumption that the marshalling/unmarshalling of JSON documents is left to Jongo. Morph-xR2RML retrieves JSON documents from Jongo as Java strings in order to evaluate them with JSONPath expressions. It is likely that converting documents to strings and evaluating them with a third-party JSONPath library significantly impairs performances. Further investigation should be conducted to figure this out more precisely, keeping in mind that solving this issue could approximately save a factor 2 during the processing of large result sets.

Impact of Query Optimizations In this section, we measure the completion time of two example SPARQL queries involving joins. Notably, we measure the gain obtained by performing optimizations at the level of the abstract query, namely the self-join elimination and the filter propagation. Additional example queries are reported in [22] along with measures of the impact of the self-union elimination and the constant projection optimizations.

Join Query, Self-Join Elimination. SPARQL query Q_7, depicted below, looks for taxa (variable *?t*) that are present in the overseas collectivity of Saint-Pierre-et-Miquelon (http://sws.geonames.org/3424932/). The graph pattern matches 12,708 triples that yield a SPARQL result set of 4,236 solutions.

```
SELECT * WHERE {
  ?t taxrefprop:bioGeoStatusIn ?bgs .      # tp1
  ?bgs dct:spatial
    <http://sws.geonames.org/3424932/> .   # tp2
  ?bgs dwc:occurrenceStatus taxrefbgs:P .  # tp3
}
```

Executed separately, the first triple pattern would be bound to 15 mappings (one for each geographical location) and would yield 311,489 RDF triples; the second one would be bound to one mapping and would yield 4,835 triples, and the third one would be bound to 15 mapping and would yield 260,631 documents. Executed as such, query Q_7 completes in almost 10 minutes (600s).

```
[ { Binding(tp1: ?t taxrefprop:bioGeoStatusIn ?bgs -> TM_SBG_SPM)
    From    : db.taxrefv9.find({$where:'this.codeTaxon==this.codeReference',
                                'spm':{$ne:''},'spm':{$ne:null}})
    Project: $.codeTaxon AS ?t, $.codeTaxon AS ?bgs
    Where   : isNotNull($.codeTaxon) }
] INNER JOIN [
   [ { Binding(tp2: ?bgs dct:spatial http://sws.geonames.org/3424932/
                    -> TM_SBG_SPM_BN2)
       From    : db.taxrefv9.find({$where:'this.codeTaxon==this.codeReference',
                                   'spm':{$ne:''}, 'spm':{$ne:null}})
       Project: $.codeTaxon AS ?bgs
       Where   : isNotNull($.codeTaxon) }
   ] INNER JOIN [
     { Binding(tp3: ?bgs dwc:occurrenceStatus taxrefbgs:P ->TM_SBG_SPM_BN1)
       From    : db.taxrefv9.find({$where:'this.codeTaxon==this.codeReference',
                                   'spm':{$ne:''}, 'spm':{$ne:null}})
       Project: $.codeTaxon AS ?bgs)
       Where   : isNotNull($.codeTaxon), equals($.spm, P) }
   ] ON ?bgs
] ON ?bgs


{ Binding(tp1: ?t   taxrefprop:bioGeoStatusIn ?bgs -> TM_SBG_SPM),
  Binding(tp2: ?bgs dct:spatial http://sws.geonames.org/3424932/
                -> TM_SBG_SPM_BN2),
  Binding(tp3: ?bgs dwc:occurrenceStatus taxrefbgs:P -> TM_SBG_SPM_BN1)
  From    : db.taxrefv9.find({$where:'this.codeTaxon == this.codeReference',
                              'spm':{$ne:''}, 'spm':{$ne:null}})
  Project: $.codeTaxon AS ?t, $.codeTaxon AS ?bgs
  Where   : isNotNull($.codeTaxon), equals($.spm, P)
}
```

Listing 1.8. Top: rewriting of the graph pattern of query Q_7 after bindings reduction. Bottom: the same query after self-join elimination.
Compared to the notation used in previous sections, each atomic abstract query contains heading lines providing the binding(s) of the triple pattern(s) that this atomic query accounts for, denoted by `Binding(triple pattern -> mapping name)`.

The binding reduction step (section 4.2) removes all but one mapping bound to the first and third triple patterns. The query now amounts to the join of three atomic abstract queries depicted in Listing 1.8 (top). The first and second atomic queries yield 4,835 RDF triples while the third query yields 4,236 triples. Under such reduced bindings, query Q_7 completes in 8.53s in average, the querying to MongoDB accounts for 47% of this total time, the generation of the RDF triples accounts for 11% and the processing of joins for 39%.

A closer look to the abstract query shows that it contains two self-joins that can be eliminated for the following reasons: (i) all three queries share the same *From* part (the logical source), (ii) they are joined on the *?bgs* variable that is always projected from the same reference `$.codeTaxon`, and (iii) `$.codeTaxon` is declared as a unique identifier in at least one mapping bound to the three triple patterns (with property `xrr:uniqueRef`). This self-join elimination yields an optimized query that now consists of a single atomic query depicted in Listing 1.9 (bottom). Note that the *Project* and *Where* parts have been merged, and the three bindings now apply to this atomic query: the same MongoDB query is

```
[{ Binding(?t skosxl:prefLabel http://inpn.mnhn.fr/taxref/label/60585
          -> TM_Taxon_PrefLabel)
   From    : db.taxrefv9.find( {
               $where: 'this.codeTaxon==this.codeReference' } )
   Project: $.codeTaxon AS ?t
   Where   : isNotNull($.codeTaxon), equals($.codeTaxon, 60585) }
] INNER JOIN [
    [{ Binding(?t skosxl:altLabel ?a -> TM_Taxon_AltLabel)
       From    : db.taxrefv9.find( {
                   $where: 'this.codeTaxon!=this.codeReference' } )
       Project: $.codeReference AS ?t, $.codeTaxon AS ?a
       Where   : isNotNull($.codeReference), isNotNull($.codeTaxon) }
    ] INNER JOIN [
     { Binding(?t skosxl:altLabel ?b -> TM_Taxon_AltLabel)
       From    : db.taxrefv9.find( {
                   $where: 'this.codeTaxon!=this.codeReference' } )
       Project: $.codeReference AS ?t, $.codeTaxon AS ?b
       Where   : isNotNull($.codeReference), isNotNull($.codeTaxon) }
    ] ON ?t
] ON ?t
```

Listing 1.9. Rewriting of the graph pattern of query Q_8.

used to generate RDF triples matching the three triple patterns. This optimized query completes in 2,966ms in average, *i.e.* a 65% gain compared to the query with reduced bindings.

Filter Propagation. SPARQL query Q_8, pictured herebelow, retrieves the taxon (variable *?t*) whose preferred label has a certain URI, alongside two of its alternate labels (variables *?a* and *?b*).

```
SELECT * WHERE {
  ?t skosxl:prefLabel
    <http://inpn.mnhn.fr/taxref/label/60585> .
  ?t skosxl:altLabel ?a .
  ?t skosxl:altLabel ?b .
  FILTER (?a != ?b)
}
```

In a first step, Q_8 translates into the inner join of three atomic abstract queries, portrayed in Listing 1.9. The first atomic query retrieves 1 document from the database, while the second and third queries retrieve 257,965 documents each. Executed naively, the inner-most join computes the join of 257,965 triples with another 257,965 triples generated from the same database documents. With a smarter join ordering, the triple produced by the first atomic query is joined with the 257,965 triples of the second one to produce two triples (taxon 60585 has two synonyms), that, in turn, are joined with the 257,965 triples of the third query. Yet, two joins of 257,965 triples with one then two triples have to be performed. Some tests show that the time needed to complete this query is in the order of 4 minutes.

The Filter Propagation optimization leverages some situations where, within the join of two sub-queries, a condition on a variable shared by both sub-queries can be propagated from one sub-query to the other. In the example,

```
[{ Binding(?t skosxl:prefLabel http://inpn.mnhn.fr/taxref/label/60585
          -> TM_Taxon_PrefLabel)
   From    : db.taxrefv9.find( {
                $where: 'this.codeTaxon==this.codeReference' } )
   Project: $.codeTaxon AS ?t
   Where   : isNotNull($.codeTaxon), equals($.codeTaxon, 60585) }
] INNER JOIN [
    [{ Binding(?t skosxl:altLabel ?a -> TM_Taxon_AltLabel)
       From    : db.taxrefv9.find( {
                    $where: 'this.codeTaxon!=this.codeReference' } )
       Project: $.codeReference AS ?t, $.codeTaxon AS ?a
       Where   : isNotNull($.codeTaxon), equals($.codeReference, 60585) }
    ] INNER JOIN [
     { Binding(?t skosxl:altLabel ?b -> TM_Taxon_AltLabel)
       From    : db.taxrefv9.find( {
                    $where: 'this.codeTaxon!=this.codeReference' } )
       Project: $.codeReference AS ?t, $.codeTaxon AS ?b
       Where   : isNotNull($.codeTaxon), equals($.codeReference, 60585) }
    ] ON ?t
] ON ?t
```

Listing 1.10. Rewriting of the graph pattern of query Q_8 after enforcing the filter propagation optimization.

the two joins are performed on variable *?t*. The first atomic query projects *?t* as expression `$.codeTaxon` and has condition `equals($.codeTaxon, 60585)`. In the second and third queries, variable *?t* is projected as `$.codeReference`. Therefore, the join condition can only be satisfied if expression `$.codeReference` returns the value 60585. In other words, we can propagate the condition on `$.codeTaxon`, `equals($.codeTaxon, 60585)` to the second and third queries as a condition on `$.codeReference`: `equals($.codeReference, 60585)`. The optimized abstract query is pictured in Listing 1.10. The second and third queries now only yield two RDF triples. Finally, the execution of this query lasts 565ms in average, that is a gain factor in the order of 400.

7 Discussion and Perspectives

In the case of MongoDB, the processing of joins is shifted to the query processing engine, and can ensue poor performances when joined sub-queries are not selective enough. Furthermore, real-world SPARQL queries often contain substantial graph patterns with multiple joined triple patterns. It is therefore critical to be able to process joins efficiently. Thus, beyond the optimizations that we implemented at the abstract query level, query-plan optimization techniques shall be investigated to help answer the following questions:

- Can we rewrite a SPARQL graph pattern in a way that facilitates the production of an efficient abstract query?
- How to inject intermediate results into a subsequent query, as performed in the bind join optimization [17]?
- How to reorder joins considering the number of results of sub-queries, in a way similar to methods proposed by distributed query engines? [33,16,21]

– Can we perform lazy evaluation of joins by progressively materializing triples on each side of the join until the expected number of results is reached? This would typically resemble the method employed in the non-blocking evaluation of queries in the context of Triple Pattern Fragments [40].

Additionally, several leads could be investigated to overcome the limitations of the translation from the abstract query language to MongoDB.

– Our method generates the RDF triples resulting from each atomic queries and subsequently performs joins (INNER JOIN, LEFT OUTER JOIN). In some cases though, joins may rule out many of the triples that were just materialized. Hence, it should be studied when joins can be evaluated on the database documents. This would typically rule out unnecessary documents earlier in the process, thus saving the useless generation of RDF triples.
– Our implementation of xR2RML for MongoDB relies on JSONPath to extract data elements from MongoDB results. In turn, the SPARQL rewriting process must handle conditions on JSONPath expressions. Consequently, we have to cope with the expressiveness discrepancy between SPARQL and MongoDB, and between JSONPath and MongoDB alike. While we must cope with the earlier (our goal is specifically to access heterogeneous databases with SPARQL), the latter is somewhat more an implementation choice. Hence, an investigation should figure out whether considering a restricted subset of JSONPath may produce a simpler solution while still enabling to address most mapping situations.
– Beyond this, another promising lead is to determine what type of MongoDB query should be used preferably: *find* or *aggregate* queries. We address this question in section 7, as part of a broader discussion about the similarities and discrepancies between our approach and that of *ontop*'s authors.

Comparison with the MongoDB-enabled *ontop*. To the best of our knowledge, the only other approach meant to access arbitrary MongoDB documents with SPARQL has been proposed by the authors of *ontop*, Botoeva et al. [8]. This approach starts with deriving a set of type constraints (literal, object, array) from the mapping assertions, called the MongoDB database schema. Then, a relational view over the database is defined with respect to that schema, notably by flattening array fields. A SPARQL query is rewritten into relational algebra (RA) query, and RA expressions over the relational view are translated into MongoDB *aggregate* queries. Similarly, we translate a SPARQL query into an abstract representation (that is not relational algebra) under xR2RML mappings. To deal with the tree structure of JSON documents we use JSONPath expressions. On the one hand, this avoids the definition of a relational view over the database, but this comes with additional complexity in the translation process, as translating conditions on JSONPath expressions is not straightforward. On the other hand, the advantage of our method is that the query evaluation relies on existing database indexes, whereas in the case of Botoeva et al., the flattening step prevents from exploiting these indexes.

The mappings are quite similar in both approaches although xR2RML is more flexible: (i) class names (in triples `?x rdf:type A`) and predicates can be built from database values whereas they are constant in the approach proposed by Botoeva et al., and (ii) xR2RML allows to turn an array field into an RDF collection or container, while the latter approach only supports the multiple-triples strategy.

Finally, the main differences pertain to the type of target query. Botoeva et al. produce MongoDB *aggregate* queries, with the major advantage of ensuring a semantics-preserving SPARQL-to-MongoDB query translation, thus delegating the whole processing to MongoDB and making the query translation simpler. In practice however, *aggregate* pipelines may perform poorly. To optimize them, an option suggested by the authors is to decompose the pipeline into smaller queries and have the query-processing engine perform the remaining steps. Our approach works the other way around: it produces less-expressive MongoDB *find* queries, leaving much more work to the query-processing engine. Nevertheless, having the job done outside of the database engine allows to leverage extensive works about smart query optimizations[17,33,16,21], whereas this is not possible when the database performs an *aggregate* query in a black-box manner.

Typically though, in situations involving large joins, *aggregate* queries perform faster than *find* queries as they can leverage database indexes. In the future, it would be interesting to assess whether we could characterize mappings with respect to the type of query that shall perform best: single vs. multiple separate queries, *find* vs. *aggregate*, and figure out a balance between the two approaches.

Furthermore, unlike *ontop*, xR2RML allows for rich JSONPath expressions to evaluate a JSON document and generate RDF terms. In this matter, further studies should figure out how to translate such expressions into *aggregate* queries.

8 Conclusion

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 push the Web of Data forward. In particular, we think that this should help to bridge the gap between the Semantic Web and the NoSQL family of databases.

To achieve this goal without defining yet another SPARQL translation method for each and every database, we proposed a two-phase approach. First, we defined an abstract query language deriving from the syntax and semantics of SPARQL. Utilizing the xR2RML mapping language and leveraging R2RML-based SPARQL-to-SQL works, we introduced a generic method to translate a SPARQL 1.0 graph pattern into an abstract query. We showed how optimizations can be beneficially enforced at this abstract level, saving subsequent work at the level of a target database language. In a second phase, the abstract query is translated into the query language of a target database. To demonstrate the effectiveness of our approach, we applied it to the MongoDB NoSQL document store. We devised a method to translate an abstract query into MongoDB *find*

queries, and we showed that this translation is challenged by the expressiveness discrepancy between SPARQL and the MongoDB query language.

Finally, we conducted an experimentation based on the real-world use case of a taxonomical reference stored in a MongoDB database. Utilizing a mapping of this database to a SKOS thesaurus, we first measured performances in the case of single SPARQL triple patterns that translate into single MongoDB queries. Then, we measured the performances of richer SPARQL queries and we demonstrated the effectiveness of some of the optimizations performed at the level of the abstract query language. We underlined some limitations of the translation from the abstract query language to MongoDB, that can impair performances. In section 7 we discuss several improvement leads that could be investigated.

From a broader perspective, we have shown that translating a SPARQL query into efficient concrete queries can be challenging when it comes to address data sources such as NoSQL databases. These systems are generally optimized for fast storage and retrieval of vast collections of documents. They favor scalability, high throughput and availability over consistency and query language expressiveness. As a consequence, they often come with denormalized data models where redundancy is common, and barely support joins. This is the case of other document stores such as CouchDB that are designed in a way very similar to MongoDB. Column family stores usually allow for a richer data model and provide a more expressive query language. But although their columnar data model makes them easily compared with relational systems, they often suffer the same limitations as document stores with respect to the limited support of joins. Key-value stores are designed for fast retrieval of data *e.g.* accessed by key. They are typically used to implement cache systems, for which a very simple query language (consisting essentially of *put* and *retrieve* by key operations) covers most use cases.

Consequently, it is likely that the hurdles we encountered with MongoDB will be encountered with other NoSQL databases alike. The situation may not be so much different for the last category of NoSQL databases, namely graph stores. By nature, their data models are closer to RDF. Still, whereas RDF predicates can be used with literal values as well as resources, graph databases such as Neo4J[16] manage literals (called node attributes) and other graph nodes in a very different way. As a result, querying a graph database with SPARQL may be more challenging that it seems, and we believe that our two-phase approach may be relevant in this context too.

References

1. Arenas, M., Bertails, A., Prud'hommeaux, E., Sequeda, J.: A Direct Mapping of Relational Data to RDF (2012)
2. Berners-Lee, T.: Linked Data, in Design Issues of the WWW (2006), http://www.w3.org/DesignIssues/LinkedData.html
3. Bikakis, N., Tsinaraki, C., Gioldasis, N., Stavrakantonakis, I., Christodoulakis, S.: The XML and Semantic Web Worlds: Technologies, Interoperability and Integra-

[16]Neo4J: https://neo4j.com/

tion: a Survey of the State of the Art. In: Semantic Hyper/Multimedia Adaptation, pp. 319–360. Springer (2013)
4. Bikakis, N., Tsinaraki, C., Stavrakantonakis, I., Gioldasis, N., Christodoulakis, S.: The SPARQL2XQuery interoperability framework. World Wide Web **18**(2), 403–490 (Mar 2015)
5. Bizer, C., Cyganiak, R.: D2R server - Publishing Relational Databases on the Semantic Web. In: Proceeding of the 5th International Semantic Web Conference (ISWC) (2006)
6. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. International Journal on Semantic Web and Information Systems **5**(2), 1–24 (2009)
7. Botoeva, E., Calvanese, D., Cogrel, B., Rezk, M., Xiao, G.: A formal presentation of MongoDB (extended version) (2016), https://arxiv.org/abs/1603.09291v1
8. Botoeva, E., Calvanese, D., Cogrel, B., Rezk, M., Xiao, G.: OBDA beyond relational DBs: A study for MongoDB. In: Proceedings of the 29th Int. Workshop on Description Logics (2016)
9. Callou, C., Michel, F., Faron-Zucker, C., Martin, C., Montagnat, J.: Towards a Shared Reference Thesaurus for Studies on History of Zoology, Archaeozoology and Conservation Biology. In: Semantic Web For Scientific Heritage (SW4SH), ESWC workshops (2015)
10. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. Data & Knowledge Engineering **68**(10), 973–1000 (2009)
11. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (2014)
12. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language. W3C Recommendation (2012)
13. 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 (2014)
14. Elliott, B., Cheng, E., Thomas-Ogbuji, C., Ozsoyoglu, Z.M.: A Complete Translation from SPARQL into Efficient SQL. In: Proceedings of the International Database Engineering & Applications Symposium. pp. 31–42. ACM (2009)
15. Gargominy, P., Tercerie, S., Régnier, C., Ramage, T., Dupont, P., Vandel, E., Daszkiewicz, P., Poncet, L., Schoelink, C.: TAXREF v9. 0, référentiel taxonomique pour la France: Méthodologie, mise en oeuvre et diffusion
16. Görlitz, O., Staab, S.: SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In: Intl. Ws. COLD (2011)
17. Haas, L., Kossmann, D., Wimmers, E., Yang, J.: Optimizing Queries across Diverse Data Sources. In: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB 1997). pp. 276–285 (1997)
18. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation (2013)
19. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool, 1st edn. (2011)
20. Husson, A.: Une sémantique statique pour MongoDB. In: 25th Journées Francophones des Langages Applicatifs. pp. 77–92 (2014)
21. Macina, A., Montagnat, J., Corby, O.: Optimising SPARQL query processing in distributed knowledge graphs. In: Actes de la Conférence Gestion de Données - Principes, Technologies et Applications (BDA). Poitiers, France (2016)
22. Michel, F.: Integrating Heterogeneous Data Sources in the Web of Data. Ph.d. thesis, Université Côte d'Azur (Mar 2017)

23. Michel, F., Faron-Zucker, C., Montagnat, J.: A Generic Mapping-Based Query Translation from SPARQL to Various Target Database Query Languages. In: Proceeding of the 12th International Conference on Web Information Systems and Technologies (WebIST). vol. 2, pp. 147–158 (2016)
24. Michel, F., Faron-Zucker, C., Montagnat, J.: A Mapping-based Method to Query MongoDB Documents with SPARQL. In: Proceedings of the 27th DEXA International Conference. LNCS, vol. 9828, pp. 52–67. Springer (2016)
25. Michel, F., Faron-Zucker, C., Montagnat, J.: Translation of Heterogeneous Databases into RDF, and Application to the Construction of a SKOS Taxonomical Reference. In: Revised Selected Papers of the 11th WebIST international conference, pp. 275–296. No. 246 in LNBIP, Springer (2016)
26. Mugnier, M.L., Rousset, M.C., Ulliana, F.: Ontology-Mediated Queries for NOSQL Databases. In: Proceedings of the 30th Conference on Artificial Intelligence. Phoenix, Arizona, USA (2016)
27. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems **34**(3), 1–45 (Aug 2009)
28. Pollock, R., Tennison, J., Kellogg, G., Herman, I.: Metadata Vocabulary for Tabular Data. W3C Recommendation (2015)
29. Priyatna, F., Corcho, O., Sequeda, J.: Formalisation and Experiences of R2RML-based SPARQL to SQL query translation using Morph. In: Proceeding of the World Wide Web Conference (WWW) (2014)
30. Rodrıguez-Muro, M., Calvanese, D.: High Performance Query Answering over DL-Lite Ontologies. In: Proceedings of the 13th Int. Conference on Principles of Knowledge Representation and Reasoning (KR 2012) (2012)
31. Rodríguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-Based Data Access: Ontop of Databases. In: The Semantic Web - ISWC 2013, pp. 558–573. Springer (2013)
32. Rodríguez-Muro, M., Rezk, M.: Efficient SPARQL-to-SQL with R2RML mappings. Web Semantics **33**, 141–169 (2015)
33. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: Optimization techniques for federated query processing on Linked Data. In: Proceedings of the 10th International Conference on Semantic Web (ISWC'11), pp. 601–616. Springer (2011)
34. Sequeda, J., Tirmizi, S.H., Corcho, s., Miranker, D.P.: Survey of directly mapping SQL databases to the Semantic Web. Knowledge Eng. Review **26**(4), 445–486 (2011)
35. Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL execution on relational data. Web Semantics **22** (2013)
36. Spanos, D.E., Stavrou, P., Mitrou, N.: Bringing Relational Databases into the Semantic Web: A survey. Semantic Web Journal **3**(2), 169–209 (2012)
37. Tomaszuk, D.: Document-oriented triplestore based on RDF/JSON. In: Logic, philosophy and computer science, pp. 125–140. University of Bialystok (2010)
38. Unbehauen, J., Stadler, C., Auer, S.: Accessing Relational Data on the Web with SparqlMap. In: Semantic Technology, pp. 65–80. Springer (2013)
39. Unbehauen, J., Stadler, C., Auer, S.: Optimizing SPARQL-to-SQL Rewriting. In: Proceedings of Information Integration and Web-based Applications & Services (iiWAS'13). p. 324. ACM (2013)
40. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: a low-cost knowledge graph interface for the web. Web Semantics **37–38**, 184–206 (2016)