A Mapping-based Method to Query MongoDB Documents with SPARQL

Franck Michel, Catherine Faron Zucker, Johan Montagnat

To cite this version:

Franck Michel, Catherine Faron Zucker, Johan Montagnat. A Mapping-based Method to Query MongoDB Documents with SPARQL. 27th International Conference on Database and Expert Systems Applications (DEXA 2016), Sep 2016, Porto, Portugal. hal-01330146

HAL Id: hal-01330146
https://hal.archives-ouvertes.fr/hal-01330146

Submitted on 10 Jun 2016

A Mapping-based Method to Query MongoDB Documents with SPARQL

Franck Michel, Catherine Faron-Zucker, and Johan Montagnat

Univ. Nice Sophia Antipolis, CNRS, I3S (UMR 7271), France
{fmichel,faron,johan}@i3s.unice.fr

Abstract. Accessing legacy data as virtual RDF stores is a key issue in the building of the Web of Data. In recent years, the MongoDB database has become a popular actor in the NoSQL market, making it a significant potential contributor to the Web of Linked Data. Therefore, in this paper we address the question of how to access arbitrary MongoDB documents with SPARQL. We propose a two-step method to (i) translate a SPARQL query into a pivot abstract query under MongoDB-to-RDF mappings represented in the xR2RML language, then (ii) translate the pivot query into a concrete MongoDB query. We elaborate on the discrepancy between the expressiveness of SPARQL and the MongoDB query language, and we show that we can always come up with a rewriting that shall produce all correct answers.

Keywords: SPARQL access to legacy data, MongoDB, virtual RDF store, Linked Data, xR2RML

1 Introduction

The Web-scale data integration progressively becomes a reality, giving birth to the Web of Linked Data through the open publication and interlinking of data sets on the Web. It results from the extensive works achieved during the last years, aimed to expose legacy data as RDF and develop SPARQL interfaces to various types of databases.

At the same time, the success of NoSQL databases is no longer questioned today. Initially driven by major Web companies in a pragmatic effort to cope with large distributed data sets, they are now adopted in a variety of domains such as media, finance, transportation, biomedical research and many others[1]. Consequently, harnessing the data available from NoSQL databases to feed the Web of Data, and more generally achieving RDF-based data integration over NoSQL systems, are timely questions. In recent years, MongoDB[2] has become a very popular actor in the NoSQL market[3]. Beyond dealing with large distributed data sets, its popularity suggests that it is also increasingly adopted as a general-purpose database. Arguably, it is likely that many MongoDB instances host

[1] Informally attested by the manifold domains of customers claimed by major NoSQL actors.

[2] https://www.mongodb.org/

[3] http://db-engines.com/en/system/MongoDB

valuable data about all sorts of topics, that could benefit a large community at the condition of being made accessible as Linked Open Data. Hence the research question we address herein: *How to access arbitrary MongoDB documents with SPARQL?*

Exposing legacy data as RDF has been the object of much research during the last years, usually following two approaches: either by materialization, *i.e.* translation of all legacy data into an RDF graph at once, or based on on-the-fly translation of SPARQL queries into the target query language. The materialization is often difficult in practice for big datasets, and costly when data freshness is at stake. Several methods have been proposed to achieve SPARQL access to relational data, either in the context of RDB-backed RDF stores [8,21,11] or using arbitrary relational schemas [4,23,17,18]. R2RML [9], the W3C RDB-to-RDF mapping language recommendation is now a well-accepted standard and several SPARQL-to-SQL rewriting approaches hinge upon it [23,17,19]. Other solutions intend to map XML [3,2] or CSV[4] data to RDF. RML [10] tackles the mapping of heterogeneous data formats such as CSV/TSV, XML and JSON. xR2RML [14] is an extension of R2RML and RML addressing the mapping of an extensible scope of databases to RDF. Regarding MongoDB specifically, Tomaszuk proposed a solution to use MongoDB as an RDF triple store [22]. The translation of SPARQL queries that he proposed is closely tied to the data schema and does not fit with arbitrary documents. MongoGraph[5] is an extension of the AllegroGraph triple store to query arbitrary MongoDB documents with SPARQL. Similarly to the Direct Mapping [1] the approach comes 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. More in line with our work, Botoeva et al. recently proposed a generalization of the OBDA principles to MongoDB [6]. They describe a two-step rewriting process of SPARQL queries into the MongoDB aggregate query language. In the last section we analyse in further details the relationship between their approach and ours.

In this paper we propose a method to query arbitrary MongoDB documents using SPARQL. We rely on xR2RML for the mapping of MongoDB documents to RDF, allowing for the use of classes and predicates from existing (domain) ontologies. In section 2 we shortly describe the xR2RML mapping language. Section 3 defines a database-independent abstract query language, and summarizes a generic method to rewrite SPARQL queries into this language under xR2RML mappings. Then section 4 presents our method to translate abstract queries into MongoDB queries. Finally in section 5 we conclude by emphasizing some technical issues and highlighting perspectives.

2 The xR2RML Mapping Language

The xR2RML mapping language [14] is designed to map an extensible scope of relational and non-relational databases to RDF. It is independent of any query

[4] http://www.w3.org/2013/csvw/wiki

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

language or data model. It is backward compatible with R2RML and it relies on RML for the handling of various data formats. It can translate data with mixed embedded formats and generate RDF collections and containers. Below we shortly describe the main xR2RML features and propose a running example.

An xR2RML mapping defines a logical source (`xrr:logicalSource`) as the result of executing a query (`xrr:query`) against an input database. An optional iterator (`rml:iterator`) can be applied to each query result. 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, and multiple predicate and object maps produce the predicate and object terms. An optional graph map is used to name a target graph. Listing 1.2 depicts the `<#TmLeader>` xR2RML mapping.

Term maps extract data from query results by evaluating *xR2RML references*. The syntax of xR2RML references depends on the target database: a column name in case of a relational database, an XPath expression in case of a 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` that contains a single xR2RML reference, and `rr:template` that may contain several references in a template string. In the running example below, the subject map uses a template to build IRI terms by concatenating `http://example.org/project/` with the value of JSON field `"code"`. When the evaluation of an xR2RML reference produces several RDF terms, by default the xR2RML processor creates one triple for each term. Alternatively, it can group them in an RDF collection (`rdf:List`) or container (`rdf:Seq`, `rdf:Bag` and `rdf:Alt`) of terms optionally qualified with a language tag or data type.

Like R2RML, xR2RML allows to model cross-references by means of *referencing object maps*. A referencing object map uses values produced by the subject map of a mapping (the parent) as the objects of triples produced by another mapping (the child). 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. This short example is specifically tailored to address the issues related to the SPARQL-to-MongoDB translation, it does not illustrate advanced xR2RML features, but more detailed use cases are provided in [14,7]. Let us consider a MongoDB database with one collection `"projects"` (Listing 1.1), that lists the projects held in a company. Each project is described by a name, a code and a set of teams. Each team is an array of members given by their name, and we assume that the last member is always the team leader. The xR2RML mapping graph in Listing 1.2 has one mapping: `<#TmLeader>`. The logical source is the MongoDB query `"db.projects.find({})"` that simply retrieves all documents from collection `"projects"`. The mapping associates projects (subject) to team leaders (object) with predicate `ex:teamLeader`. This is done by means of a JSONPath expression that selects the last member of each team using the calculated array index `"[(@.length - 1)]"`.

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

```
{ "project":"Finance & Billing", "code":"fin",
  "teams":[
   [ {"name":"P. Russo"}, {"name":"F. Underwood"}],
   [ {"name":"R. Danton"}, {"name":"E. Meetchum"} ]] },
{ "project":"Customer Relation", "code":"crm",
  "teams":[
   [ {"name":"R. Posner"}, {"name":"H. Dunbar"}]] }
```

Listing 1.1. MongoDB collection "projects" containing two documents

```
<#TmLeader>
   xrr:logicalSource [xrr:query "db.projects.find({})"];
   rr:subjectMap [rr:template
     "http://example.org/project/{$.code}".];
   rr:predicateObjectMap [
      rr:predicate ex:teamLeader;
      rr:objectMap [ xrr:reference
        "$.teams[0,1][(@.length - 1)].name" ] ].
```

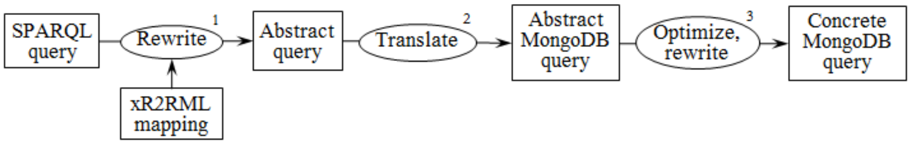
Listing 1.2. xR2RML example mapping graph

3 Translating SPARQL Queries into Abstract Queries under xR2RML Mappings

Various methods have been defined to translate SPARQL queries into another query language, that are generally tailored to the expressiveness of the target query language. Notably, the rich expressiveness of SQL and XQuery makes it possible to define semantics-preserving SPARQL rewriting methods [8,2]. By contrast, NoSQL databases typically trade off expressiveness for scalability and fast retrieval of denormalised data. For instance, many of them hardly support joins. Therefore, to envisage the translation of SPARQL queries in the general case, we propose a two-step method. Firstly, a SPARQL query is rewritten into a pivot abstract query under xR2RML mappings, independently of any target database (illustrated by step 1 in Figure 1). Secondly, the pivot query is translated into concrete database queries based on the specific target database capabilities and constraints. In this paper we focus on the application of the second step to the specific case of MongoDB. The rest of this section summarizes the first step to provide the reader with appropriate background. A complete description is provided in [16].

The grammar of our pivot query language is depicted in Listing 1.3. Operators INNER JOIN ON, LEFT OUTER JOIN ON and UNION are entailed by the dependencies between graph patterns of the SPARQL query, and SPARQL filters involving variables shared by several triple patterns result in a FILTER operator. The computation of these operators shall be delegated to the target database if it supports them (i.e. if the target query language has equivalent operators like

Fig. 1. Overview of the SPARQL-to-MongoDB Query Translation Process



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

Listing 1.3. Grammar of the Abstract Pivot Query Language

SQL), or to the query processing engine otherwise (e.g. MongoDB cannot process joins). Each SPARQL triple pattern *tp* is translated into a union of atomic abstract queries (<`AtomicQuery`>), under the set of xR2RML mappings likely to generate triples matching *tp*. Components of an atomic abstract query are as follows:

- *From* is the mapping's logical source, i.e. the database query string (`xrr:query`) and its optional iterator (`rml:iterator`).

- *Project* is the set of xR2RML references that must be projected, i.e. returned as part of the query results. In SQL, projecting an xR2RML reference simply means that the column name shall appear in the SELECT clause. As to MongoDB, this amounts to projecting the JSON fields mentioned in the JSONPath reference.

- *Where* is a conjunction of abstract conditions entailed by matching each term of triple pattern *tp* with its corresponding term map in an xR2RML mapping: the subject of *tp* is matched with the subject map of the mapping, the predicate with the predicate map and the object with the object map. Three types of condition may be created:
(i) a SPARQL variable in the triple pattern is turned into a not-null condition on the xR2RML reference corresponding to that variable in the term map, denoted by *isNotNull(<xR2RML reference>)*;
(ii) A constant triple pattern term (IRI or literal) is turned into an equality condition on the xR2RML reference corresponding to that RDF term in the term map, denoted by *equals(<xR2RML reference>, value)*;
(iii) A SPARQL filter condition *f* about a SPARQL variable is turned into a filter condition, denoted by *sparqlFilter(<xR2RML reference>, f)*.

Finally, an abstract query is optimized using classical query optimization techniques such as the self-join elimination, self-union elimination or projection pushing. In [16] we show that, during the optimization phase, a new type of abstract condition may come up, *isNull(<xR2RML reference>)*, in addition to logical operators *Or()* and *And()* to combine conditions.

Running Example. We consider the following SPARQL query that aims to retrieve projects in which "*H. Dunbar*" is a team leader.

```
SELECT ?proj WHERE {?proj ex:teamLeader "H. Dunbar".}
```

The triple pattern, denoted by `tp`, is translated into the atomic abstract query {*From, Project, Where*}. *From* is the query in the logical source of mapping `<#TmLeader>`, i.e. `"db.projects.find({})"`. The detail of calculating *Project* is out of the scope of this paper; let us just note that, since the values of variable `?proj` (the subject of `tp`) shall be retrieved, only the subject map reference is projected, i.e. the JSONPath expression "`$.code`". The *Where* part is calculated as follows:
- `tp`'s subject, variable `?proj`, is matched with `<#TmLeader>`'s subject map; this entails condition C_1: `isNotNull($.code)`.
- `tp`'s object, `"H. Dunbar"`, is matched with `<#TmLeader>`'s object map; this entails condition C_2: `equals($.teams[0,1][(@.length-1)].name, "H. Dunbar")`.
Thus, the SPARQL query is rewritten into the atomic abstract query below:

```
{ From:    {"db.projects.find({})"},
  Project: {$.code AS ?proj},
  Where:   {isNotNull($.code),
    equals($.teams[0,1][(@.length-1)].name, "H. Dunbar") }}
```

The JSON documents needed to answer this abstract query shall verify condition $C_1 \wedge C_2$. In the next section, we elaborate on the method that allows to rewrite such conditions into concrete MongoDB queries.

4 Translating an Abstract Query into MongoDB Queries

In this section we briefly describe the MongoDB query language, then we define rules to transform an atomic abstract query into an abstract representation of a MongoDB query (step 2 in Figure 1). Finally, we define additional rules to optimize and rewrite an abstract representation of a MongoDB query into a union of executable MongoDB queries (step 3 in Figure 1).

4.1 The MongoDB Query Language

MongoDB provides a JSON-based declarative query language consisting of two major mechanisms. The *find* query retrieves documents matching a set of conditions. It takes a query and a projection parameters, and returns a cursor to the matching documents. Optional modifiers amend the query to impose limits and sort orders. Alternatively, the *aggregate* query allows for the definition of processing pipelines: each document of a collection passes through the stages of

```
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
COND(isNull)                  → $eq:null
NOT_EXISTS(<exp>)             → <exp>:{$exists:false}
COMPARE(<exp>, <op>, <v>)     → <exp>:{<op>:<v>}
NOT_SUPPORTED                 → ∅
CONDJS(equals(v))             → == v
CONDJS(isNotNull)             → != null
```

Listing 1.4. Abstract MongoDB query representation and translation to a concrete query string

the pipeline, that allows for richer aggregate computations. As a first approach, this work considers the *find* query method, hereafter called the *MongoDB query language*. As an illustration let us consider the following query:

```
db.projects.find(
  {"teams.0":{$elemMatch:{"age":{$gt:30}}}}, {"code":1})
```

It retrieves documents from collection "projects", whose first team (array `"teams"` at index 0) has at least one member (operator `$elemMatch`) over 30 years old (operator `$gt`). The projection parameter, {`"code":1`}, states that only the `"code"` field of each matching document must be returned.

The MongoDB documentation[7] provides a rich description of the query language, that however lacks formal semantics. Recently, attempts were made to clarify this semantics while underlining some limitations and ambiguities: [5] focuses mainly on the *aggregate* query and ignores some of the operators we use in our translation, such as $`where`, $`elemMatch`, $`regex` and $`size`. On the other hand, [13] describes the *find* query, yet some restrictions on the operator $`where` are not formalized. Hence, in [15] we specified the grammar of the subset of the query language that we consider. We also defined an abstract representation of MongoDB queries, that allows for handy manipulation during the query construction and optimization phases. Listing 1.4 details the constructs of this representation and their equivalent concrete query string. In the COMPARE clause definition, `<op>` stands for one of the MongoDB comparison operators: `$eq`, `$ne`, `$lt`, `$lte`, `$gt`, `$gte`, `$size` and `$regex`. The NOT_SUPPORTED clause helps keep track of parts of the abstract query that cannot be translated into an equivalent MongoDB query element; it shall be used when rewriting the abstract query into a concrete query (section 4.3).

[7] https://docs.mongodb.org/manual/tutorial/query-documents/

4.2 Query Translation Rules

Section 3 introduced a method that rewrites a SPARQL query into an abstract query in which operators INNER JOIN, LEFT OUTER JOIN and UNION relate atomic abstract queries of the form {*From, Project, Where*}. The latter are created by matching each triple pattern with candidate xR2RML mappings. The *Where* part consists of *isNotNull, equals* and *sparqlFilter* abstract conditions about xR2RML references (JSONPath expressions in the case of MongoDB).

MongoDB does not support joins, while unions and nested queries are supported under strong restrictions, and comparisons are limited (e.g. a JSON field can be compared to a literal but not to another field of the same document). Consequently, operators INNER JOIN, LEFT OUTER JOIN, and to some extend UNION and FILTER, shall be computed by the query processing engine. Conversely, the abstract conditions of atomic queries can be translated into MongoDB queries[8].

Given the subset of the MongoDB query language considered, the recursive function *trans* in Table 2 translates an abstract condition on a JSONPath expression into a MongoDB *find* query using the formalism defined in Listing 1.4. It consists of a set of rules applicable to a certain pattern. The JSONPath expression in argument is checked against each pattern in the order of the rules (0 to 9) until a match is found. We use the following notations:
- <JP>: denotes a possibly empty JSONPath expression.
- <JP:F>: denotes a non-empty JSONPath sequence of field names and array indexes, e.g. `.p.q.r`, `.p[10]["r"]`.
- <bool_expr>: is a JavaScript expression that evaluates to a boolean.
- <num_expr>: is a JavaScript expression that evaluates to a positive integer.

Rule R0 is the entry point of the translation process (JSONPath expressions start with a '$' character). Rule R1 is the termination point: when the JSONPath expression has been fully parsed, the last created clause is the condition clause COND, producing e.g. `"$eq:value"` for an equality condition, or `"$exists:true, $ne:null"` for a not-null condition. Rules R2 to R8 deal with the different types of JSONPath expressions. In case no rule matches, the translation fails and rule R9 creates the NOT_SUPPORTED clause, that shall be dealt with later on. Rule R4 deals with the translation of JavaScript filters on JSON arrays, where character '@' stands for each array element. It delegates their processing to function *transJS* (described in [15]). For instance, the filter `"[?(@.age>30)]"` is translated into the MongoDB sub-query `"age"`:{`$gt:30`}.

Due to the space constraints, we do not go through the comprehensive justification of each rule in Table 2, however the interested reader is referred to [15].

Running Example. The *Where* part of the abstract query presented in section 3 comprises two conditions:
C_1: `isNotNull($.code)`, and
C_2: `equals($.teams[0,1][(@.length - 1)].name, "H. Dunbar")`.

Here are the rules applied at each step of the translation of C_1 and C_2.

[8] In the current state of this work we do not consider SPARQL filter conditions.

Fig. 2. Translation of a condition on a JSONPath expression into an abstract MongoDB query (function *trans*)

R0 **trans**($, <cond>) → ∅
trans($<JP>, <cond>) → **trans**(<JP>, <cond>)

R1 **trans**(∅, <cond>) → **COND**(<cond>)

R2 *Field alternative (a) or array index alternative (b)*
(a) **trans**(<JP:F>["p","q",...]<JP>, <cond>) →
OR(**trans**(<JP:F>.p<JP>, <cond>), **trans**(<JP:F>.q<JP>, <cond>), ...)
(b) **trans**(<JP:F>[i,j,...]<JP>, <cond>) →
OR(**trans**(<JP:F>.i<JP>, <cond>), **trans**(<JP:F>.j<JP>, <cond>), ...)

R3 *Heading field alternative (a) or heading array index alternative (b)*
(a) **trans**(["p","q",...]<JP>, <cond>) →
OR(**trans**(.p<JP>, <cond>), **trans**(.q<JP>, <cond>), ...)
(b) **trans**([i,j,...]<JP>, <cond>) →
OR(**trans**(.i<JP>, <cond>), **trans**(.j<JP>, <cond>), ...)

R4 *JavaScript filter on array elements, e.g.,* `$.p[?(@.q)].r`
trans([?(<bool_expr>)]<JP>, <cond>) →
ELEMMATCH(**trans**(<JP>, <cond>), **trans**JS(<bool_expr>))

R5 *Array slice: n last elements (a) or n first elements (b)*
(a) **trans**(<JP:F>[-<start>:]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(<JP:F>, -<start>)
(b) **trans**(<JP:F>[:<end>]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(<JP:F>, <end>)
trans(<JP:F>[0:<end>]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(<JP:F>, <end>)

R6 *Calculated array index, e.g.,* `$.p[(@.length - 1)].q`
(a) **trans**(<JP$_1$>[(<num_expr>)]<JP$_2$>, <cond>) → **NOT_SUPPORTED**
if <JP$_1$> contains a wildcard or a JavaScript filter expression
(b) **trans**(<JP:F>[(<num_expr>)], <cond>) → **AND**(
EXISTS(<JP:F>),
WHERE('this<JP:F>[replaceAt("this<JP:F>", <num_expr>)] **CONDJS**(<cond>')))
(c) **trans**(<JP:F$_1$>[(<num_expr>)]<JP:F$_2$>, <cond>) → **AND**(
EXISTS(<JP:F$_1$>),
WHERE('this<JP:F$_1$>[replaceAt("this<JP:F$_1$>", <num_expr>)]<JP:F$_2$>
CONDJS(<cond>')))

R7 *Heading wildcard*
(a) **trans**(.*<JP>, <cond>) → **ELEMMATCH**(**trans**(<JP>, <cond>))
(b) **trans**([*]<JP>, <cond>) → **ELEMMATCH**(**trans**(<JP>, <cond>))

R8 *Heading field name or array index*
(a) **trans**(.p<JP>, <cond>) → **FIELD**(p) **trans**(<JP>, <cond>)
(b) **trans**(["p"]<JP>, <cond>) → **FIELD**(p) **trans**(<JP>, <cond>)
(c) **trans**([i]<JP>, <cond>) → **FIELD**(i) **trans**(<JP>, <cond>)

R9 *No other rule matched, expression <JP> is not supported*
trans(<JP>, <cond>) → **NOT_SUPPORTED**

$M_1 \leftarrow$ trans(C_1) = trans(`$.code, isNotNull`):
R0: $M_1 \leftarrow$ trans(`.code, isNotNull`)
R8 then R1: $M_1 \leftarrow$ FIELD(`code`) COND(`isNotNull`)

$M_2 \leftarrow$ trans(C_2) =
      trans(`$.teams[0,1][(@.length-1)].name, equals("H. Dunbar")`)
R0: $M_2 \leftarrow$ trans(`.teams[0,1][(@.length-1)].name, equals("H. Dunbar")`)

R2 splits the alternative `"[0,1]"` into two members of an OR clause:

```
M2 ← OR(  trans(teams.0.[(@.length-1)].name, equals("H. Dunbar")),
          trans(teams.1.[(@.length-1)].name, equals("H. Dunbar"))).
```

R6(c) processes the calculated array index `"(@.length-1)"` in each OR member:

```
M2 ← OR( AND(EXISTS(.teams.0),
     WHERE('this.teams[0][this.teams[0].length-1)].name=="H. Dunbar"')),
    AND(EXISTS(.teams.1),
     WHERE('this.teams[1][this.teams[1].length-1)].name=="H. Dunbar"')))
```

4.3 Rewriting of the abstract MongoDB query representation into a concrete MongoDB query

Rules R0 to R9 translate a condition on a JSONPath expression into an abstract MongoDB query. Yet, several potential issues hinder the rewriting into a concrete query: (i) a NOT_SUPPORTED clause may indicate that a part of the JSONPath expression could not be translated into an equivalent MongoDB operator; (ii) a WHERE clause may be nested beneath a sequence of AND and/or OR clauses although the MongoDB `$where` operator is valid only in the top-level query; (iii) unnecessary complexity such as nested ORs, nested ANDs, etc., may hamper performances. Those issues are addressed by two sets of rewriting rules, O1 to O5 and W1 to W6. They require the addition of the UNION clause to those in Listing 1.4. UNION is semantically equivalent to the OR clause but, whereas ORs are processed by the MongoDB database, UNIONs shall be computed by the query processing engine.

Query Optimization. Rules O1 to O5 in Table 3 perform several query optimizations. Rules O1 to O4 address issue (iii) by flattening nested OR, AND and UNION clauses, and merging sibling WHEREs. Rule O5 addresses issue (i) by removing the clauses of type NOT_SUPPORTED while still making sure that the query returns all the correct answers:

- O5(a): If a NOT_SUPPORTED clause occurs in an AND clause, it is simply removed. Let $C_1, ...C_n$ be any clauses and N be a NOT_SUPPORTED clause. Since $C_1 \wedge ... \wedge C_n \supseteq C_1 \wedge ... \wedge C_n \wedge N$, the rewriting widens the condition. Hence, all matching documents are returned. However, non-matching documents may be returned too, that shall be ruled out later on.
- O5(b): A logical AND implicitly applies to members of an ELEMMATCH clause. Therefore, removing the NOT_SUPPORTED has the same effect as in O5(a).
- O5(c) and (d): A NOT_SUPPORTED is managed differently in an OR or UNION clause. Since $C_1 \vee ... \vee C_n \subseteq C_1 \vee ... \vee C_n \vee N$, removing N would return a subset of the matching documents. Instead, we replace the whole OR or UNION clause

Fig. 3. Optimization of an abstract MongoDB query

O1	*Flatten nested AND, OR and UNION clauses:*
	AND(C_1,... C_n, AND(D_1,... D_m,)) → AND(C_1,... C_n, D_1,... D_m)
	OR(C_1,... C_n, OR(D_1,... D_m,)) → OR(C_1,... C_n, D_1,... D_m)
	UNION(C_1,... C_n, UNION(D_1,... D_m,)) → UNION(C_1,... C_n, D_1,... D_m)

O2	*Merge ELEMMATCH with nested AND clauses:* ELEMMATCH(C_1,... C_n, AND(D_1,... D_m,)) → ELEMMATCH(C_1,... C_n, D_1,... D_m)

O3	*Group sibling WHERE clauses:*
	OR(..., WHERE("W_1"), WHERE("W_2")) → OR(..., WHERE("(W_1) || (W_2)"))
	AND (..., WHERE("W_1"), WHERE("W_2")) → AND(..., WHERE("(W_1) && (W_2)"))
	UNION(..., WHERE("W_1"), WHERE("W_2")) → UNION(..., WHERE("(W_1) || (W_2)"))

O4	*Replace AND, OR or UNION clauses of one term with the term itself.*
O5	*Remove NOT_SUPPORTED clauses:*
(a)	AND(C_1,... C_n, NOT_SUPPORTED) → AND(C_1,... C_n)
(b)	ELEMMATCH(C_1,... C_n, NOT_SUPPORTED) → ELEMMATCH(C_1,... C_n)
(c)	OR(C_1,... C_n, NOT_SUPPORTED) → NOT_SUPPORTED
(d)	UNION(C_1,... C_n, NOT_SUPPORTED) → NOT_SUPPORTED
(e)	FIELD(...)... FIELD(...) NOT_SUPPORTED → NOT_SUPPORTED

Fig. 4. Pulling up WHERE clauses to the top-level query

W1	OR(C_1,...C_n, W) → UNION(OR(C_1,...C_n), W)
W2	OR(C_1,...C_n, AND(D_1,...D_m, W)) → UNION(OR(C_1,...C_n), AND(D_1,...D_m, W))
W3	AND(C_1,...C_n, W) → (C_1,...C_n, W) if the AND clause is a top-level query object or under a UNION clause.
W4	AND(C_1,...C_n, OR(D_1,...D_m, W)) → UNION(AND(C_1,...C_n, OR(D_1,...D_m)), AND(C_1,...C_n, W))
W5	AND(C_1,...C_n, UNION(D_1,...D_m)) → UNION(AND(C_1,...C_n, D_1),... AND(C_1,...C_n, D_m))
W6	OR(C_1,...C_n, UNION(D_1,...D_m)) → UNION(OR(C_1,...C_n), D_1, ...D_m))

with a NOT_SUPPORTED clause. This way, the NOT_SUPPORTED issue is raised up to the parent clause and shall be managed at the next iteration. Iteratively, the NOT_SUPPORTED clause is raised up until it is eventually removed (cases AND and ELEMMATCH above), or it ends up in the top-level query. The latter is the worst case in which the query shall retrieve all documents.

- O5(e): Similarly to O5(c), a sequence of fields followed by a NOT_SUPPORTED clause must be replaced with a NOT_SUPPORTED clause to raise up the issue to the parent clause.

Pulling up WHERE Clauses. By construction, rule R6 ensures that WHERE clauses cannot be nested in an ELEMMATCH, but they may show in AND and OR clauses. Besides, rules O1 to O4 flatten nested OR and AND clauses, and merge sibling WHERE clauses. Therefore, a WHERE clause may be either in the top-level query (in this case the query is executable) or it may show in one of the following patterns (where `W` stands for a WHERE clause):

`OR(...,W,...)`, `AND(...,W,...)`, `OR(..., AND(...,W,...), ...)`, `AND(..., OR(...,W,...), ...)`.
In such patterns, rules W1 to W6 (Table 4) address issue (ii) by "pulling up" WHERE clauses into the top-level query. Here is an insight into the approach:

- Since `OR(C, W)` is not a valid MongoDB query, it is replaced with query `UNION(C, W)` which has the same semantics: `C` and `W` are evaluated separately against the database, and the UNION is computed later on by the query processing engine.
- `AND(C,OR(D,W))` is rewritten into `OR(AND(C,D), AND(C,W))` and the OR is replaced with a UNION: `UNION(AND(C,D), AND(C,W))`. Since an logical AND implicitly applies to the top-level terms, we can finally rewrite the query into `UNION((C,D), (C,W))` which is valid since `W` now shows in a top-level query.

Rewriting rules W1 to W6 are a generalization of these examples. They ensure that a query containing a nested WHERE can always be rewritten into a union of queries wherein the WHERE shows only in a top-level query. Hence we formulate Theorem 1, for which a proof is provided in [15].

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 $Q^{'} =$* UNION*$(Q^{'}_1, .., Q^{'}_m)$ 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: *$Q^{'}$ 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.*

Running Example. For the sake of readability, below we denote the JavaScript conditions in M_1 and M_2 as follows: *$JScond_0$* stands for
`this.teams[0][this.teams[0].length-1)].name=="H. Dunbar"`, and *$JScond_1$* for
`this.teams[1][this.teams[1].length-1)].name=="H. Dunbar"`.

In section 4.2 we have translated conditions C_1 and C_2 into abstract MongoDB queries M_1 and M_2. The MongoDB documents needed to answer the SPARQL query shall be retrieved by the query AND(M_1, M_2) =

AND(FIELD(code) COND(isNotNull), OR(
AND(EXISTS(.teams.0), WHERE('JScond$_0$'))
AND(EXISTS(.teams.1), WHERE('JScond$_1$'))))

Applying subsequently rules W2 and O4 replaces the inner OR with a UNION:
AND(FIELD(code) COND(isNotNull), UNION(
AND(EXISTS(.teams.0), WHERE('JScond$_0$'))
AND(EXISTS(.teams.1), WHERE('JScond$_1$'))))

Rule W5 pulls up the UNION clause:
UNION(
AND(FIELD(code) COND(isNotNull), AND(EXISTS(.teams.0), WHERE('JScond$_0$'))),
AND(FIELD(code) COND(isNotNull), AND(EXISTS(.teams.1), WHERE('JScond$_1$'))))

Finally, O1 merges the nested ANDs and W3 removes the resulting top-level AND:

UNION(
    (FIELD(code) COND(isNotNull), EXISTS(.teams.0), WHERE('JScond$_0$')),
    (FIELD(code) COND(isNotNull), EXISTS(.teams.1), WHERE('JScond$_1$')))

The abstract query can now be rewritten into a union of two valid queries:
{"code":{$exists:true, $ne:null}, "teams.0":{$exists:true},
    $where:'this.teams[0][this.teams[0].length-1)].name == "H. Dunbar"'}

{"code":{$exists:true, $ne:null}, "teams.1":{$exists:true},
    $where:'this.teams[1][this.teams[1].length-1)].name == "H. Dunbar"'}

The first query retrieves the document below, whereas the second query returns no document.

```
{ "project":"Customer Relation", "code":"crm",
  "teams":[ [ {"name":"R. Posner"}, {"name":"H. Dunbar"}]]}
```

Finally, the application of triples map `<#TmLeader>` to the query result produces one RDF triple that matches the triple pattern `tp`:
`<http://example.org/project/crm> ex:teamLeader "H. Dunbar".`

5 Discussion, Conclusion and Perspectives

In this document we proposed a method to access arbitrary MongoDB documents with SPARQL. This relies on custom mappings described in the xR2RML mapping language which allows for the reuse of existing domain ontologies. First, we introduced a method to rewrite a SPARQL query into a pivot abstract query independent of any target database, under xR2RML mappings. Then, we devised a set of rules to translate this pivot query into an abstract representation of a MongoDB query, and we showed that the latter can always be rewritten into a union of concrete MongoDB queries that shall return all the documents required to answer the SPARQL query.

Due to the limited expressiveness of the MongoDB *find* queries, some JSONPath expressions cannot be translated into equivalent MongoDB queries. Consequently, the query translation method cannot guarantee that query semantics be preserved. Yet, we ensure that rewritten queries retrieve all matching documents, possibly with additional non-matching ones. The RDF triples thus extracted are subsequently filtered by evaluating the original SPARQL query. This preserves semantics at the cost of an extra SPARQL query evaluation.

In a recent work, Botoeva et al. proposed a generalization of the OBDA principles to support MongoDB [6]. Both approaches have similarities and discrepancies that we outline below. Botoeva et al. derive 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 a 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 the relational algebra) under xR2RML mappings. The mappings are quite similar in both approaches although xR2RML is slightly more flexible: class names (in triples `?x rdf:type A`) and predicates can be built from database values whereas they are fixed in [6], and xR2RML allows to turn an array field into an RDF collection or container. To deal with the tree form of JSON documents we use JSONPath expressions. This avoids the definition of a relational view over the database, but this also comes with additional complexity in the translation process. Finally, [6] produces MongoDB *aggregate* queries, with the advantage that a SPARQL 1.0 query may be translated into a single target query, thus delegating all the processing to MongoDB. Yet, in practice, some aggregate queries may be very inefficient, hence the need to decompose RA queries into sub-queries, as underlined by the authors. Our approach produces *find* queries that are less expressive but whose performance is easier to anticipate, thus putting a higher burden on the query processing engine (joins, some unions and filtering). In the future, it would be interesting to characterise mappings with respect to the type of query that shall perform best (single vs. multiple separate queries, find vs. aggregate). A lead may be to involve query plan optimization logics such as the bind join [12] and the join reordering methods applied in the context of distributed SPARQL query engines [20].

More generally, the NoSQL trend pragmatically gave up on properties such as consistency and rich query features, as a trade-off to high throughput, high availability and horizontal elasticity. Therefore, it is likely that the hurdles we encountered with MongoDB shall occur with other NoSQL databases.

Implementation and evaluation. To validate our approach we have developed a prototype implementation[9] available under the Apache 2 open source licence. Further developments on query optimization are on-going, and in the short-term we intend to run performance evaluations. Besides, we are working on two real-life use cases. Firstly, in the context of the Zoomathia research project[10], we proposed to represent a taxonomic reference, designed to support studies in Conservation Biology, as a SKOS thesaurus [7]. It is stored in a MongoDB database, and we are in the process of testing the SPARQL access to that thesaurus using our prototype. Secondly, we are having discussions with researchers in the fields of ecology and agronomy. They intend to explore the added value of Semantic Web technologies using a large MongoDB database of phenotype information. This context would be a significant and realistic use case of our method and prototype.

References

1. M. Arenas, A. Bertails, E. Prud'hommeaux, and J. Sequeda. A Direct Mapping of Relational Data to RDF, 2012.
2. N. Bikakis, C. Tsinaraki, I. Stavrakantonakis, N. Gioldasis, and S. Christodoulakis. The SPARQL2XQuery interoperability framework. *WWW*, 18(2):403–490, 2015.

[9] https://github.com/frmichel/morph-xr2rml

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

3. S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres. Mapping between RDF and XML with XSPARQL. *J. Data Semantics*, 1(3):147–185, 2012.
4. C. Bizer and R. Cyganiak. D2R server - Publishing Relational Databases on the Semantic Web. In *ISWC*, 2006.
5. E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk, and G. Xiao. A formal presentation of MongoDB (Extended version). Technical report, 2016.
6. E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk, and G. Xiao. OBDA beyond relational DBs: A study for MongoDB. In *Int. Ws. DL 2016*, volume 1577, 2016.
7. C. Callou, F. Michel, C. Faron-Zucker, C. Martin, and J. Montagnat. Towards a Shared Reference Thesaurus for Studies on History of Zoology, Archaeozoology and Conservation Biology. In *SW For Scientific Heritage, Ws. of ESWC*, 2015.
8. A. Chebotko, S. Lu, and F. Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering*, 68(10):973–1000, 2009.
9. S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF mapping language, 2012.
10. A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle. RML: A generic language for integrated RDF mappings of heterogeneous data. In *LDOW*, 2014.
11. B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z. M. Ozsoyoglu. A complete translation from SPARQL into efficient SQL. In *IDEAS'09*, pages 31–42. ACM, 2009.
12. L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries across Diverse Data Sources. In *VLDB*, pages 276–285, 1997.
13. A. Husson. Une sémantique statique pour MongoDB. In *25th Journées Francophones des Langages Applicatifs (JFLA)*, pages 77–92, 2014.
14. F. Michel, L. Djimenou, C. Faron-Zucker, and J. Montagnat. Translation of Relational and Non-Relational Databases into RDF with xR2RML. In *WebIST*, pages 443–454, 2015.
15. F. Michel, C. Faron-Zucker, and J. Montagnat. Mapping-based SPARQL access to a MongoDB database. Technical report, CNRS, 2015. https://hal.archives-ouvertes.fr/hal-01245883.
16. F. Michel, C. Faron-Zucker, and J. Montagnat. A Generic Mapping-Based Query Translation from SPARQL to Various Target Database Query Languages. In *WebIST*, 2016.
17. F. Priyatna, O. Corcho, and J. Sequeda. Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph. In *WWW*, 2014.
18. M. Rodríguez-Muro, R. Kontchakov, and M. Zakharyaschev. Ontology-based data access: Ontop of databases. In *The Semantic Web-ISWC 2013*. Springer, 2013.
19. M. Rodríguez-Muro and M. Rezk. Efficient SPARQL-to-SQL with R2RML mappings. *J. Web Semantics*, 33:141–169, 2015.
20. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on Linked Data. In *ISWC*. 2011.
21. J. F. Sequeda and D. P. Miranker. Ultrawrap: SPARQL execution on relational data. *J. Web Semantics*, 22:19–39, 2013.
22. D. Tomaszuk. Document-oriented triplestore based on RDF/JSON. In *Logic, philosophy and computer science*, pages 125–140. University of Bialystok, 2010.
23. J. Unbehauen, C. Stadler, and S. Auer. Accessing relational data on the web with sparqlmap. In *Semantic Technology*, pages 65–80. Springer, 2013.