



**HAL**  
open science

## Mapping-based SPARQL access to a MongoDB database

Franck Michel, Catherine Faron Zucker, Johan Montagnat

► **To cite this version:**

Franck Michel, Catherine Faron Zucker, Johan Montagnat. Mapping-based SPARQL access to a MongoDB database. [Research Report] CNRS. 2015. hal-01245883v1

**HAL Id: hal-01245883**

**<https://hal.science/hal-01245883v1>**

Submitted on 17 Dec 2015 (v1), last revised 7 Nov 2016 (v5)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INFORMATIQUE, SIGNAUX ET SYSTÈMES DE SOPHIA ANTIPOLIS  
UMR7271

## Mapping-based SPARQL Access to a MongoDB Database

*Franck Michel, Catherine Faron-Zucker, Johan Montagnat*

Equipe SPARKS

Rapport de Recherche, Dec. 2015

Version	Date	Description
v1	Dec. 2015	Initial version

Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S) - UMR7271 - UNS CNRS  
2000, route des Lucioles - Les Algorithmes - bât. Euclide B 06900 Sophia Antipolis - France <http://www.i3s.unice.fr>

Membre de UNIVERSITÉ CÔTE D'AZUR 

<b>1</b>	<b>INTRODUCTION</b> .....	<b>3</b>
<b>2</b>	<b>THE XR2RML MAPPING LANGUAGE</b> .....	<b>4</b>
2.1	Recalls on R2RML .....	4
2.2	xR2RML language description.....	5
2.3	Restrictions to the use of xR2RML within this document.....	6
2.4	Running Example .....	6
<b>3</b>	<b>REWRITING A SPARQL QUERY INTO AN ABSTRACT QUERY</b> .....	<b>8</b>
3.1	Abstract Query .....	9
3.2	Atomic Abstract Query.....	11
<b>4</b>	<b>BINDING TRIPLES MAPS TO TRIPLE PATTERNS</b> .....	<b>14</b>
<b>5</b>	<b>TRANSLATION OF AN ABSTRACT QUERY INTO A MONGODB QUERY</b> .....	<b>16</b>
5.1	The MongoDB query language .....	17
5.2	The JSONPath language.....	19
5.3	Conventions and formalism.....	21
5.4	Query translation rules .....	22
5.4.1	<i>Rule R0</i> .....	23
5.4.2	<i>Rule R1</i> .....	23
5.4.3	<i>Rule R2</i> .....	24
5.4.4	<i>Rule R3</i> .....	24
5.4.5	<i>Rule R4</i> .....	24
5.4.6	<i>Rule R5</i> .....	24
5.4.7	<i>Rule R6</i> .....	25
5.4.8	<i>Rule R7</i> .....	26
5.4.9	<i>Rule R8</i> .....	26
5.4.10	<i>Rule R9</i> .....	26
5.4.11	<i>Translation of a JavaScript filter to MongoDB</i> .....	26
5.5	Query optimization and translation to a concrete MongoDB query .....	27
5.5.1	<i>Query optimization</i> .....	27
5.5.2	<i>Pull up WHERE clauses</i> .....	29
5.5.3	<i>Function rewrite</i> .....	31
<b>6</b>	<b>OVERALL QUERY TRANSLATION AND EVALUATION PROCESS</b> .....	<b>33</b>
<b>7</b>	<b>CONCLUSION, DISCUSSION AND PERSPECTIVES</b> .....	<b>35</b>
7.1	Conclusion.....	35
7.2	Query optimization.....	36
7.3	Support of the SPARQL query language.....	36
7.4	Dealing with the \$where operator .....	37
<b>8</b>	<b>APPENDIX A</b> .....	<b>38</b>
8.1	Function genProjection.....	38
8.2	Function genCond .....	38
<b>9</b>	<b>APPENDIX B: COMPLETE RUNNING EXAMPLE</b> .....	<b>41</b>
9.1	Translation of tp2 .....	42
9.2	Translation of tp1 .....	43
9.3	Complete trans <sub>m</sub> processing.....	45
<b>10</b>	<b>REFERENCES</b> .....	<b>46</b>

# 1 Introduction

The Web-scale data integration progressively becomes a reality, giving birth to the Web of Data. It is sustained and promoted by the W3C Data Activity<sup>1</sup> working group that aims at overcoming data diversity and support public and private sector organizations in this matter. A key-point to the achievement of the Web of Data is that data be published openly on the Web in a standard, machine-readable format, and linked with other related data sets. In this matter, an extensive work has been achieved during the last years to expose legacy data as RDF.

At the same time, the success of NoSQL database platforms is no longer questioned today. Driven by major Web companies, they have been developed to meet requirements of novel applications, hardly available in relational databases (RDB), such as a flexible schema, high throughput, high availability and horizontal elasticity. Not only NoSQL platforms are at the core of many applications dealing with big data, but they are also increasingly used as a generic-purpose database in many domains. Consequently, NoSQL databases must now be considered as significant contributors to feed the Web of Linked Open Data.

In recent years, MongoDB<sup>2</sup> has become the leader in the NoSQL market, as suggested by several indicators including Google searches<sup>3</sup>, job offerings<sup>4</sup> and LinkedIn member profiles mentioning MongoDB skills<sup>5</sup>. Some methods have been proposed to translate MongoDB documents into RDF [10], or to use MongoDB as an RDF triple store [19]. Yet, to the best of our knowledge, no work has been proposed so far to query arbitrary MongoDB documents using SPARQL. In this paper we propose a method to access MongoDB documents using SPARQL. The translation is based on custom mappings from arbitrary MongoDB documents to RDF described using the xR2RML mapping language [10].

In the rest of this section we review previous works related the translation of various data sources into RDF. Section 2 presents the xR2RML mapping language and its use within this document, and introduces a running example (section 2.4). In section 3 we first describe a method to rewrite a SPARQL query into an abstract query that makes no assumption as to the target database query language. This relies on bindings between a SPARQL triple pattern and xR2RML mappings, detailed in section 4. Section 5 focuses more specifically on the translation of an abstract query into MongoDB concrete queries. Section 6 recaps the whole method through an algorithm that orchestrates the different steps, until the evaluation of MongoDB queries and the generation of the RDF triples matching the SPARQL query. After a discussion and conclusion in section 7, appendix B (section 9) goes over the running example that is been detailed throughout the previous sections.

## Related works.

Much work has been achieved during the last years to expose existing data as RDF, in which two approaches generally apply: either the RDF graph is materialized by translating the data into RDF and loading it in a triple store (in an ETL – Extract, Transform and Load - manner), or the raw data is unchanged and a query language such as SPARQL is used to access the virtual RDF graph through query rewriting techniques. While materializing the RDF graph can be needed in some contexts, it is often impossible in practice due to the size of generated graphs, and not desirable when data freshness is at stake. Several methods have been proposed to achieve SPARQL access to relational data, either in the context of RDF stores backed by RDBs [4,15,7] or using arbitrary relational schemas [3,17,12,13]. R2RML [5], the W3C RDB-to-RDF mapping language recommendation is now a well accepted standard and various SPARQL-to-SQL rewriting approaches rely on it [17,12,13]. Other solutions intend to map XML data to RDF [2,1], and the CSV on the Web W3C working group<sup>6</sup> makes a recommendation for the description of and access

<sup>1</sup> <http://www.w3.org/2013/data/>

<sup>2</sup> <https://www.mongodb.org/>

<sup>3</sup> <https://www.google.com/trends/explore#q=mongodb,couchdb,couchbase,membase,hbase>

<sup>4</sup> <http://www.indeed.com/jobtrends/mongodb,mongo,cassandra,hbase,couchdb,couchbase,membase,redis.html>

<sup>5</sup> [https://blogs.the451group.com/information\\_management/tag/nosql/](https://blogs.the451group.com/information_management/tag/nosql/)

<sup>6</sup> <http://www.w3.org/2013/csvw/wiki>

to CSV data on the Web. RML [6] is an extension of R2RML that tackles the mapping of data sources with heterogeneous data formats such as CSV/TSV, XML or JSON. The xR2RML mapping language [10] is an extension of the R2RML and RML addressing the mapping of a large and extensible scope of non-relational databases to RDF. Some works have been proposed to use MongoDB as an RDF triple store, and in this context they designed a method to translate SPARQL queries into MongoDB queries [19]. MongoGraph<sup>7</sup> is an extension of AllegroGraph<sup>8</sup> to query MongoDB documents with SPARQL queries. It follows an approach very similar to the Direct Mapping approach defined in the context of RDBs [16]: each field of a MongoDB JSON document is translated into an ad-hoc predicate, and a mapping links MongoDB document identifiers with URIs. SPARQL queries use the specific *find* predicate to tell the SPARQL engine to query MongoDB. Despite those approaches, to the best of our knowledge, no work has been proposed yet to translate a SPARQL query into the MongoDB query language and map arbitrary MongoDB documents to RDF.

## 2 The xR2RML mapping language

The xR2RML mapping language [10] is designed to map an extensible scope of relational and non-relational databases to RDF. It flexibly adapts to heterogeneous query languages and data models thus remaining independent from any specific database. It is backward compatible with R2RML and it relies on RML for the handling of various data formats.

Below we shortly describe the main xR2RML features, a complete specification of the language is available in [11]. We assume the following namespace prefix definitions:

```
xrr: <http://www.i3s.unice.fr/ns/xr2rml#>
rr: <http://www.w3.org/ns/r2rml#>
rml: <http://semweb.mmlab.be/ns/rml#>
ex: <http://example.com/ns#>
```

### 2.1 Recalls on R2RML

R2RML is a generic language meant to describe customized mappings that translate data from a relational database into an RDF data set. An R2RML mapping is expressed as an RDF graph that consists of *triples maps*, each one specifying how to map rows of a logical table to RDF triples. A triples map is composed of exactly one *logical table* (property `rr:logicalTable`), one *subject map* (property `rr:subjectMap`) and any number of *predicate-object maps* (property `rr:predicateObjectMap`). A logical table may be a table, an SQL view (property `rr:tableName`), or the result of a valid SQL query (property `rr:sqlQuery`). A predicate-object map consists of *predicate maps* (property `rr:predicateMap`) and *object maps* (property `rr:objectMap`). For each row of the logical table, the subject map generates a subject IRI, while each predicate-object map creates one or more predicate-object pairs. Triples are produced by combining the subject IRI with each predicate-object pair. Additionally, triples are generated either in the default graph or in a named graph specified using *graph maps* (property `rr:graphMap`).

Subject, predicate, object and graph maps are all R2RML *term maps*. A term map is a function that generates RDF terms (either a literal, an IRI or a blank node) from elements of a logical table row. A term map must be exactly one of the following: a *constant-valued term map* (property `rr:constant`) always generates the same value; a *column-valued term map* (property `rr:column`) produces the value of a given column in the current row; a *template-valued term map* (property `rr:template`) builds a value from a template string that references columns of the current row.

When a logical resource is cross-referenced, typically by means of a foreign key relationship, it may be used as the subject of some triples and the object of some others. In such cases, a *referencing object map* uses IRIs produced by

<sup>7</sup> <http://franz.com/agraph/support/documentation/4.7/mongo-interface.html>

<sup>8</sup> <http://allegrograph.com/>

the subject map of a (parent) triples map as the objects of triples produced by another (child) triples map. In case both triples maps do not share the same logical table, a joint query must be performed. A join condition (property `rr:joinCondition`) names the columns from the parent and child triples maps, that must be joined (properties `rr:parent` and `rr:child`).

Below we provide a short illustrative example. Triples map `<#R2RML_Directors>` uses table `DIRECTORS` to create triples linking movie directors (whose IRIs are built from column `NAME`) with their birth date (column `BIRTH_DATE`).

```
<#R2RML_Directors>
  rr:logicalTable [rr:tableName "DIRECTORS" ];
  rr:subjectMap [
    rr:template "http://example.org/dir/{NAME}";
    rr:class ex:Manager ];
  rr:predicateObjectMap [
    rr:predicate ex:birthdate;
    rr:objectMap [
      rr:column "BIRTH_DATE";
      rr:datatype xsd:date ] ].
```

## 2.2 xR2RML language description

An xR2RML mapping defines a logical source (property `xrr:logicalSource`) as the result of executing a query (property `xrr:query`) against an input database. The query is expressed in the query language of the target database. Data from the logical source is mapped to RDF using *triples maps*. Like in R2RML a triples map consists of several *term maps* that extract values from a query result set and translate them into terms of RDF triples. A subject map generates the subject of RDF triples, and multiple predicate-object maps produce the predicate and object terms. Optionally, a graph map is used to name a target graph. Listing 3 depicts two xR2RML triples map `<#Departments>` and `<#Staff>`.

**xR2RML references.** Term maps extract data from query results by evaluating xR2RML data element references, hereafter named *xR2RML references*. The syntax of xR2RML references is called the *reference formulation* (as a reference to the RML property of the same name), it depends on the target database: a column name in case of a relational database, an XPath expression in case of a native XML database, or a JSONPath expression in case of JSON documents like in MongoDB. An xR2RML processor is provided with a connection to the target database and the reference formulation applicable to results of queries run against the connection. xR2RML references are used with properties `xrr:reference` and `rr:template`. The `xrr:reference` property contains a single xR2RML reference, whereas the `rr:template` property may contain several references in a template string.

**Iteration model.** xR2RML implements a document-based iteration model: a document is basically one entry of a result set returned by the target database, e.g. a JSON document retrieved from a NoSQL document store, rows of an SQL result set or an XML document retrieved from an XML native database. In some contexts, this iteration model may not be sufficient to address all needs: it may be needed to iterate on explicitly specified entries of a JSON document or elements of an XML tree. To this end, xR2RML leverages the concept of iterator introduced in RML. An iterator (property `rml:iterator`) specifies the iteration pattern to apply to data read from the input database. Its value is an expression written using the syntax specified in the reference formulation. For instance, in the collection in database Listing 2, if we were interested in team members rather than in departments, we would define an iterator in the logical source of triples map `<#Departments>` to explicitly specify to iterate on elements of the `members` array:

```
<#Departments>
  xrr:logicalSource [ xrr:query "db.departments.find({})"; rml:iterator "$.members.*" ];
```

**Mixed-syntax paths.** xR2RML extends RML's principle of data element references to allow referencing data elements within mixed content. For instance, a JSON value may be embedded the cells of a relational table. In such cases, properties `xrr:reference` and `rr:template` may accept *mixed-syntax path* expressions. An xR2RML *mixed-syntax path* consists of the concatenation of several path expressions, each path being enclosed in a *syntax path constructor* that makes explicit the path syntax. Existing constructors are: `Column()`, `CSV()`, `TSV()`, `JSONPath()` and `XPath()`. For example, in a relational table, a text column `NAME` stores JSON-formatted values containing people's first and last names, e.g.: `{"First":"John", "Last":"Smith"}`. Field `FirstName` can be referenced with the following mixed-syntax path: `Column(NAME)/JSONPath($.First)`.

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

**Cross-references.** 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 another triples map (the parent) as objects. Properties `rr:child` and `rr:parent` specify the join condition between documents of the current triples map (the child), and the parent triples map. In Listing 3 this is exemplified by triples map `<#Staff>` that has a referencing object map whose parent triples map is `<#Departments>`.

The objects produced by a referencing object map can be grouped in an RDF collection or container, instead of being the objects of multiple triples, using specific values of the property `rr:termType`, mentioned above.

Results of the joint query are grouped by child value, i.e.: objects generated by the parent triples map, referring to the same child value, are grouped as members of an RDF collection or container.

## 2.3 Restrictions to the use of xR2RML within this document

To keep the document focused on the query translation question and for the sake of clarity, the running example in section 2.3 does not use iterators nor mixed syntax paths.

In xR2RML, as well as in R2RML, a triples map may contain any number of predicate-object maps, and a predicate-object map may contain any number (>1) of predicate maps and object maps. Although they do not explicitly mention it, authors of [12] and [17] assume that a triples map contains only one predicate-object map, each having exactly one predicate map and one object map. In [13] (appendix A), the authors propose an algorithm to normalize R2RML mappings so as to comply with this assumption. We comply with it as it significantly simplifies the description of the algorithms, while keeping the full expressiveness of R2RML. In the following, we assume that a triples map contains exactly one predicate-object map with exactly one predicate map and one object map.

Furthermore, the R2RML `rr:class` property introduces a specific way of producing triples such as `"<A> rdf:type <B>"`. The mapping normalization in [13] proposes to replace any `rr:class` property by an equivalent predicate-object map: `[rr:predicate rdf:type; rr:object <A>.]`. We also comply with this proposition as it allows for the definition of a general method consistently dealing with all kinds of triple patterns, may they have the `rdf:type` property or any other property.

## 2.4 Running Example

To illustrate the description of our method, we define a running example that we refer to all along this document. Additionally, section 9 goes through the whole method and provides additional explanations.

Let us consider a MongoDB database with two collections “staff” and “departments” given in Listing 1 and Listing 2 respectively. Collection “departments” lists the departments within a company, including a department code and its members. Members are given by their name and age. Collection “staff” lists people by their name (that may be either field “familyname” or “lastname”), and provides a list of departments that they manage, if any, in array field “manages”.

---

**Listing 1: Collection “staff”**


---

```
{ "familyname": "Underwood", "manages": ["Sales"] },
{ "lastname": "Dunbar", "manages": ["R&D", "Human Resources"] },
{ "lastname": "Sharp", "manages": ["Support", "Business Dev"] }
```

---



---

**Listing 2: Collection “departments”**


---

```
{ "dept": "Sales", "code": "sa",
  "members": [{"name": "P. Russo", "age": 28}, {"name": "J. Mendez", "age": 43}] },
{ "dept": "R&D", "code": "rd",
  "members": [{"name": "J. Smith", "age": 32}, {"name": "D. Duke", "age": 23}] },
{ "dept": "Human Resources", "code": "hr",
  "members": [{"name": "R. Posner", "age": 46}, {"name": "D. Stamper", "age": 38}] },
{ "dept": "Business Dev", "code": "bdev",
  "members": [{"name": "R. Danton", "age": 36}, {"name": "E. Meetchum", "age": 34}] }
```

---

Let us consider the xR2RML mapping graph in Listing 3, consisting of two triples maps <#Staff> and <#Departments>. The logical source in triples map <#Staff> provides a MongoDB query `db.staff.find({})` that retrieves all documents in collection “staff”. The parameter “{}” is basically an empty filter. Similarly, the query in <#Departments>’s logical sources retrieves all documents in collection “departments”. Triples map <#Staff> has a referencing object map whose parent triples map is <#Departments>. Triples map <#Departments> generates triples with predicate `ex:hasSeniorMember` for each member of the department who is 40 years old or more. For the sake of simplicity the queries in both triples maps retrieve all documents of the collection with no other query filter.

---

**Listing 3: xR2RML Example Mapping Graph**


---

```
<#Departments>
  xrr:logicalSource [ xrr:query "db.departments.find({})" ];
  rr:subjectMap [ rr:template "http://example.org/dept/{$.code}" ];
  rr:predicateObjectMap [
    rr:predicate ex:hasSeniorMember;
    rr:objectMap [ xrr:reference "$.members[?(@.age >= 40)].name" ] ].

<#Staff>
  xrr:logicalSource [ xrr:query "db.staff.find({})" ];
  rr:subjectMap [ rr:template "http://example.org/staff/{['lastname','familyname']}" ];
  rr:predicateObjectMap [
    rr:predicate ex:manages;
    rr:objectMap [
      rr:parentTriplesMap <#Departments>;
      rr:joinCondition [
        rr:child "$.manages.*";
        rr:parent "$.dept" ] ] ].
```

---



We wish to query the above MongoDB database with the SPARQL below query to retrieve senior members of departments whose manager is “Dunbar”. The query consists of one basic graph pattern *bgp*, itself consisting of two triple patterns *tp1* and *tp2*:

```
SELECT ?senior WHERE {
  <http://example.org/staff/Dunbar> ex:manages ?dept.      // tp1
  ?dept ex:hasSeniorMember ?senior. }                    // tp2
```

We shall use this query throughout this document to illustrate the method.

### 3 Rewriting a SPARQL query into an abstract query

SPARQL-to-SQL methods harness the ability of SQL to support joins, unions, nested queries and various string manipulation functions, to translate a SPARQL query into a single, possibly deeply nested SQL query. Some of them rely on modern RDBs optimization engines to rewrite the query in a more efficient way, although this is often not sufficient as attested by the focus on the generation of pre-optimized queries e.g. using self-join elimination or by pushing down projections and selections [7,13,15,18]. A conjunction of two basic graph patterns (BGP) generally results in the inner join of their respective translations; their union results in an SQL UNION clause; the SPARQL OPTIONAL keyword between two BGPs results in a left outer join, and a SPARQL FILTER results in an encapsulating SQL SELECT in which the filter is translated into an equivalent SQL WHERE clause. Similarly, the SPARQL-to-XQuery method proposed in [1] relies on the ability of XQuery to support the same features. For instance a SPARQL FILTER is translated into an XPath condition and/or an encapsulating XQuery For-Let-Where clause.

Below we review several works on R2RML-based SPARQL-to-SQL translation. Priyatna et al. [12] extend Chebotko's algorithm [4] that focused on the SPARQL-to-SQL query translation in the context of a RDB-based triple stores. They redefine the original mappings to comply with the context of custom mappings described in R2RML. Their method addresses the problem of eliminating null answers by adding not null conditions for variables of a triple pattern. However it has two limitations:

- (i) R2RML triples maps must have constant predicate maps, i.e. the predicates of the generated RDF triples cannot be built using a value from the database.
- (ii) Triple patterns are considered and translated independently of each other, even when variables are shared by several triple patterns of a basic graph pattern; solutions that do not match a join between two or more triple patterns are ruled out only during the final join step. The risk is to retrieve more data than actually necessary to answer queries. This may be avoided by using query optimization techniques; however it seems more natural and probably more efficient to take such constraints into account at the earliest step.

Unbehauen et al. [17] define the concept of compatibility between the RDF terms of a triple pattern and R2RML term maps (subject, predicate or object map), and subsequently the concept of triple pattern binding. This helps to effectively manage variable predicate maps, which clears the first above mentioned limitation. Furthermore, this method considers the dependencies between triple patterns of a basic graph pattern. This helps reduce the number of candidate triples maps for each triple pattern by pre-checking filters and join constraints implied by the variables shared by several triple patterns. This clears the second above mentioned limitation. This whole mapping selection process is generic and can be reused for xR2RML. Yet, two limitations can be noticed:

- (i) Referencing object maps are not addressed, and therefore only a subpart of R2RML is supported: joins implied by shared variables are dealt with but joins declared in the mapping graph are ignored.
- (ii) The rewriting maps each term map to a set of columns, called column group, that enables filtering, join and data type compatibility checks. This strongly relies on SQL capabilities (CASE, CAST, string concatenation, etc.), making it hardly applicable out of the scope of SQL-based systems.

[12] and [17] differ in the way they figure out which R2RML triples maps should be considered in the translation of a triple pattern, and how to translate a triple pattern into an effective query.

Rodríguez-Muro and Rezk [13] propose a different approach. They extend the *ontop* system that performs Ontology-Based Data Access (OBDA), 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. It must be noticed that, at the time of writing, this is the only state-of-the-art method fully supporting SPARQL 1.1.

The rich expressiveness of SQL and XQuery makes it possible to translate a SPARQL query into a single, possibly deeply nested, target query, whose semantics is strictly equivalent to that of the SPARQL query. In the general case however, i.e. beyond the scope of SQL and XQuery, joins, unions and/or sub-queries may not be supported. Unions, joins and sub-queries may be delegated to the target database when it supports these operations, or processed by the query processing engine otherwise. This is particularly the case of MongoDB: joins are not supported, and unions and nested queries are supported under strong restrictions. An xR2RML-based query processing engine for MongoDB shall evaluate several queries separately (e.g. one per triple pattern of a graph pattern), and perform joins and unions afterwards. To limit the negative impact on performances of running multiple separate queries, each query must be as selective as possible. In the SQL context a SPARQL filter is turned into an encapsulating SELECT-WHERE clause. But again, in the general case, query nesting may not be supported; this is the case of MongoDB. Thus, if filtering cannot be done in a top-level encapsulating query, it must be done at the earliest stage by pushing down filters in all separate queries, in order to reduce the size of intermediate results. To address this issue, we generalize the SPARQL translation algorithms defined in [17] and [4], this is the object of sections 3.1 and 3.2.

### 3.1 Abstract Query

Function  $trans_m$  (Definition 1) translates a SPARQL query into an abstract query that makes no assumption on the target database capabilities. The abstract query complies with the following grammar:

```

<AbstractQuery> ::= <AtomicQuery> | <Query> | <Query> WHERE <SPARQL filter>
<Query>          ::= <AbstractQuery> INNER JOIN <AbstractQuery> ON {v1, ... vn} |
                   <AbstractQuery> LEFT OUTER JOIN <AbstractQuery> ON {v1, ... vn} |
                   <AbstractQuery> UNION <AbstractQuery>
<AtomicQuery>   ::= {FromPart, ProjectPart, WherePart}

```

Operators *INNER JOIN...ON*, *LEFT OUTER JOIN...ON*, *WHERE* and *UNION* use the SQL syntax as an analogy. They are entailed by the dependencies between graph patterns of the SPARQL query, and by SPARQL filters involving variables from several triple patterns. Atomic abstract queries (<AtomicQuery>) are entailed by translating a triple pattern under a set of xR2RML triples maps.

In the next step, the operators of the abstract query may be translated into equivalent operators of the target database if they are supported (case of a relational database), or they may be computed by the query processing engine otherwise (case of MongoDB).

**Running Example.** Our running example does not include any SPARQL filter to keep it easy to follow. The application of the  $trans_m$  function to the basic graph pattern *bgp* is as follows:

```

transm(bgp, true)
= transm(tp1, true) INNER JOIN transm(tp2, true) ON var(tp1) ∩ var(tp2)
= transTPm(tp1, true) INNER JOIN transTPm(tp2, true) ON {?dept}

```

**Definition 1: Function  $trans_m$ , translation of a SPARQL query into an abstract query**

Let  $m$  be an  $xR2RML$  mapping graph consisting of a set of  $xR2RML$  triples maps. Let  $gp$  be a SPARQL graph pattern.  $trans_m(gp)$  is the translation, under  $m$ , of  $gp$  into an abstract query.  $trans_m$  is defined as follows:

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

The main difference between function  $trans_m$  and algorithms in [17] and [4] lies in the management of SPARQL filters. We assume that a SPARQL filter  $f$  is a conjunction of conditions  $c_1 \ \&\& \ \dots \ c_n$ . We distinguish conditions with regards to two criteria: (i) a condition of which all variables show in a single triple pattern, (ii) a condition of which at least one variable shows in (i.e. is shared by) several triple patterns. Note that a condition may match both criteria simultaneously. In case (i), the condition is pushed down into the translation of the triple pattern by  $transTP_m$ . This allows for filtering at the earliest stage, as opposed to the encapsulating SELECT WHERE clause in SPARQL-to-SQL methods. In case (ii), i.e. the condition involves at least one variable that shows in several triple patterns, a WHERE operator is created to ensure the join criteria. This selection of the SPARQL filter conditions is implemented by the  $sparqlCond(gp, f)$  function.

**Definition 2: Function  $sparqlCond$ , splitting SPARQL filter conditions per graph pattern**

Let  $gp$  be a SPARQL graph pattern and  $f$  be the conjunctive SPARQL filter " $C_1 \ \&\& \ \dots \ \&\& \ C_n$ ", where  $C_1$  to  $C_n$  are any SPARQL conditions. Function  $sparqlCond$  is defined as follows:

- if  $gp$  consists of a single triple pattern  $tp$ ,  $sparqlCond(tp, f)$  is the conjunction of conditions  $C_i$  such that all the variables in  $C_i$  appear in  $tp$ .
- if  $gp$  is any other graph pattern,  $sparqlCond(gp, f)$  is the conjunction of conditions  $C_i$  such that at least one variable in  $C_i$  is shared by several triple patterns.

We illustrate this process with a dedicated example (out of the scope of the running example). We apply the  $trans_m$  function to the SPARQL below, in which we denote by  $tp_1$  to  $tp_4$  the triple patterns and  $C_1$  to  $C_4$  the conditions of the SPARQL filter.

```
SELECT ?name1 ?name2 WHERE
{
  ?x foaf:name ?name1.           // tp1
  ?x foaf:mbox ?mbox1.         // tp2
  ?y foaf:name ?name2.         // tp3
  OPTIONAL {?y foaf:mbox ?mbox2.} // tp4
  FILTER { lang(?name1) IN ("EN", "FR") && // C1
           ?y != ?mbox2 && // C2
           contains(str(?mbox2), "astring") && // C3
           (?mbox1 != ?mbox2 || ?name1 != ?name2) // C4
        }
}
```

We denote by  $F$  the whole SPARQL filter, i.e.  $C_1 \ \&\& \ C_2 \ \&\& \ C_3 \ \&\& \ C_4$ .

$tp_1$ : no condition involves both  $?x$  and  $?name1$ , but  $C_1$  involves only  $?name1$ . Condition  $C_4$  involves  $?name1$  but it also involves variables that are not in  $tp_1$ . Thus  $sparqlCond(tp_1, F)$  returns only  $C_1$ .

$tp_2$ : no condition involves both  $?x$  and  $?mbox1$ , nor either  $?x$  or  $?mbox1$ ,  $sparqlCond(tp_2, F)$  returns  $\emptyset$ .

$tp_3$ : no condition involves both  $?y$  and  $?name2$ , nor either  $?y$  or  $?name2$ ,  $sparqlCond(tp_3, F)$  returns  $\emptyset$ .

$tp_4$ : condition  $C_2$  involves both variables  $?y$  and  $?mb\text{box}2$ , and  $C_3$  involves only  $?mb\text{box}2$ . Therefore  $\text{sparqlCond}(tp_4, F)$  returns “ $C_2 \ \&\& \ C_3$ ”.

Lastly, only conditions  $C_2$  and  $C_4$  involve variables from several triples patterns.

Finally we come up with the following abstract query:

```
transTPm(tp1, C1) INNER JOIN transTPm(tp2, ∅) ON {?x}
      INNER JOIN transTPm(tp3, ∅) ON ∅
      LEFT OUTER JOIN transTPm(tp4, C2 && C3) ON {?y}
WHERE C2 && C4
```

## 3.2 Atomic Abstract Query

To continue the description we need to introduce the concept of “triple pattern binding”. In section 4, we propose an algorithm that computes triple pattern bindings.

### Definition 3: Triple pattern binding (extension of Unbehauen et al. [17])

Let  $m$  be an  $xR2RML$  mapping graph consisting of a set of  $xR2RML$  triples map and  $tp$  be a triple pattern.

A triples map  $TM \in m$  is **bound to  $tp$**  if it is **compatible** with  $tp$ , i.e. it is likely to produce triples matching  $tp$ .

A **triple pattern binding** is a pair  $(tp, TMS\text{Set})$  where  $TMS\text{Set}$  is the set of triples maps of  $m$  that are bound to  $tp$ .

The  $\text{trans}_m$  function relies on the  $\text{transTP}_m$  function (Definition 4) to translate a single triple pattern into an abstract query using the set of compatible triples maps (the triples maps bound to the triple pattern).

### Definition 4: Function $\text{transTP}_m$ :

Let  $m$  be an  $xR2RML$  mapping graph consisting of a set of  $xR2RML$  triples maps,  $tp$  be a triple pattern, and  $f$  be a SPARQL filter expression. Let  $\text{getBoundTriplesMaps}_m$  be the function that, given a triple pattern  $tp$ , returns the set of triples maps of  $m$  that are bound to  $tp$ .

$\text{transTP}_m(tp, f)$  is the translation, under  $\text{getBoundTriplesMaps}_m(tp)$ , of  $(tp \wedge f)$  into an abstract query whereof results can be translated into triples matching  $tp \wedge f$ . The resulting abstract query is a union of atomic abstract queries denoted by **{FromPart, ProjectPart, WherePart}**:

- *FromPart* consists of the triples map logical source, and optionally the parent's logical source if the triples map contains a referencing object map;
- *ProjectPart* is the set of  $xR2RML$  projected references, i.e. the references used to generate RDF terms of the result triples;
- *WherePart* is the set of conditions applied to  $xR2RML$  references.

Several triples maps can be bound to one triple pattern  $tp$ , each of them may produce a subset of the triples matching  $tp$ . Therefore, the result query is a union of all per-triple-map queries, as detailed in Algorithm 1.

---

### Algorithm 1 : Translation of a triple pattern into an abstract query (function $\text{transTP}_m$ )

---

**Function**  $\text{transTP}_m(tp, f)$ :

Query  $\leftarrow$  <empty query>

BoundTMs  $\leftarrow$   $\text{getBoundTriplesMaps}_m(tp)$

**for each** TM  $\in$  BoundTMs **do**

**if** TM has a referencing object map **then**

    FromPart  $\leftarrow$  {child: <TM.logicalSource>, parent: <Parent triples map's logical source>}

**else**

    FromPart  $\leftarrow$  {child: <TM's logicalSource>}

**end if**

  ProjectPart  $\leftarrow$   $\text{genProjection}(tp, TM)$

  WherePart  $\leftarrow$   $\text{genCond}(tp, TM, f)$

  Query  $\leftarrow$  Query UNION {FromPart, ProjectPart, WherePart}

**end for**

---

---

**return Query**


---

The algorithms of functions *genProjection* and *genCond* are given in Appendix A. Let us illustrate Algorithm 1 with the triple patterns of the SPARQL query in our running example.

**Running Example.** Function *getBoundTriplesMaps<sub>m</sub>* selects bindings calculated by function *bind<sub>m</sub>* (section 4): *getBoundTriplesMaps<sub>m</sub>*(tp1) returns {<#Staff>}, and *getBoundTriplesMaps<sub>m</sub>*(tp2) returns {<#Departments>}.

```
tp2 = ?dept ex:hasSeniorMember ?senior.
transTPm(tp2, true) =
transTPm(tp2, true) =
  FromPart    ← {child: [xrr:query "db.departments.find({})"]}
  ProjectPart ← genProjection(tp2, <#Departments>)
  WherePart   ← genCond(tp2, <#Departments>, true)

tp1 = <http://example.org/staff/Dunbar> ex:manages ?dept
transTPm(tp1, true) =
transTPm(tp1, true):
  FromPart    ← { child: [xrr:query "db.staff.find({})"],
                  parent: [xrr:query "db.departments.find({})"] }
  ProjectPart ← genProjection(tp1, <#Staff>)
  WherePart   ← genCond(tp1, <#Staff>, true)
```

**FromPart.** The *FromPart* provides the concrete MongoDB queries that the abstract query relies on. It contains at least the logical source of triples map TM that consists of the *xrr:query* property – called the child query - and an optional iterator. In our running example, the *FromPart* of tp2 is simply:

```
{child: [xrr:query "db.departments.find({})"]}
```

In addition the *FromPart* may contain the logical source of the parent triples map if TM has a referencing object map. In this latter, the *xrr:query* property is called the parent query. This is the case of tp1, in turn its *FromPart* is:

```
{ child: [xrr:query "db.staff.find({})"], parent: [xrr:query "db.departments.find({})"] }
```

**ProjectPart.** The *genProjection* function selects the xR2RML references that must be projected, i.e. returned as part of the query result. An xR2RML reference may be e.g. a column name in an RDB, a JSONPath expression for MongoDB or an XPath expression for a native XML database. If a reference corresponds to a variable in the triple pattern then the reference is always projected. In our running example, the subject and object of tp2 are both variables; “?dept”, respectively “?senior”. The references of the subject map (*\$.code*) and object map (*\$.members[?(@.age >= 40)].name*) that they are matched with must be projected. Consequently:

```
genProjection(tp2, <#Departments>) = ($.code, $.members[?(@.age >= 40)].name)
```

Other projected references shall vary depending on the target database capabilities: in an RDB, columns of a join condition do not need to be projected since the database can compute the join operation. Conversely, in MongoDB, since a join shall be processed by the query processing engine, joined references must be projected. This is illustrated by tp1 in our running example. <#Staff> has a referencing object map, thus the child and parent joined references are projected and denoted by: *child/\$.manages.\** and *parent/\$.dept*. Besides, the object of tp1 is the variable “?dept”, matched with the subject map of <#Departments>, whose reference is *\$.code*. Finally we get:

```
genProjection(tp1, <#Staff>) = (child/$.manages.*, parent/$.dept, parent/$.code)
```

In the relational case, projecting an xR2RML reference simply means that a column name appears in the SQL SELECT clause. When dealing with MongoDB, projecting an xR2RML reference means projecting fields mentioned in the JSONPath expression. A MongoDB query projection clause can refer to document fields but it cannot refer to elements of an array. In the example of tp2 above, in reference *\$.members[...].name* we cannot project field “name” of elements of array “members”, we can only project field “members”. Finally, *genProjection*(tp2, <#Departments>) entails the following projection clause in the MongoDB query: {code:1, members:1}

**WherePart.** The *genCond* function computes the *WherePart* by matching each triple pattern term with its corresponding term map in each triples map.

- A variable in a triple pattern entails a non-null condition on the corresponding reference in the term map. Let us exemplify this: the subject part of tp2, variable *?dept*, is matched with the subject map of triples map *<#Departments>*, whose template string is "http://example.org/dept/{\$.code}". Without any further knowledge on *?dept*, the match states that the subject map must return a valid value, in other words the reference "\$.code" must not return null. This entails a condition: `isNotNull($.code)`. When applied to the object of tp2, the same method entails a second not-null condition: `isNotNull($.members[?(@.age >= 40)].name)`.
- When a term of the triple pattern is matched with a constant term map, no condition is entailed. E.g.: the predicate part of tp2, *ex:hasSeniorManager*, matches the constant predicate map of triples map *<#Departments>*. There is nothing more we can deduct from this. As a result, the evaluation of function *genCond* on tp2 is as follows:

```
genCond(tp2, <#Departments>, true) = isNotNull($.code),
                                     isNotNull($.members[?(@.age >= 40)].name)
```

- A constant term in the triple pattern (literal or IRI) entails an equality condition. In our running example (section 9.2), the subject part of tp1, *<http://example.org/staff/Dunbar>*, is matched with the subject map of *<#Staff>*, whose template string is "http://example.org/staff/{\$['lastname','familyname']}". This entails the equality condition:

```
equals("Dunbar", child/$['lastname','familyname']),
```

stating that either “lastname” or “familyname” must equal “Dunbar”.

- Besides, when a referencing object map is involved with either a variable or a constant term, a join condition must be added to make the joined references explicit. In our running example the object of tp1, variable *?dept*, is matched with the referencing object map of *<#Staff>*. This entails a new condition in which each joined reference is prepended with “child” or “parent” to denote the query it refers to:

```
join(child/$.manages.*, parent/$.dept)
```

As a result, the evaluation of function *genCond* on tp1 is as follows:

```
genCond(tp1, <#Staff>, true) = equals("Dunbar", child/$['lastname','familyname']),
                               isNotNull(parent/$.code),
                               join(child/$.manages.*, parent/$.dept)
```

Furthermore, if a variable of the triple pattern is mentioned in the SPARQL filter *f* passed as argument of *transTP<sub>m</sub>*, function *genCond* generates a condition *sparqlFilter(xR2RML reference, f)*. Let us mention that if *f* is *bound(?var)*, then the condition simply amounts to a *isNotNull(xR2RML reference)* condition.

**Summary.** Let us sum up the whole process so far. Function *trans<sub>m</sub>* (using function *transTP<sub>m</sub>*) translates a SPARQL query into an abstract query. INNER JOIN, LEFT OUTER JOIN, WHERE and UNION operators are entailed by the dependencies between graph patterns of the SPARQL query. UNION operators may also arise from the rewriting of a triple pattern: function *transTP<sub>m</sub>* translates a triple pattern tp into a union of atomic abstract queries {FromPart, ProjectPart, WherePart}, under a set of triples maps bound to tp. The *WherePart* is calculated by matching triple pattern terms with term maps; this shall generate either **not-null conditions** for SPARQL variables or **equality conditions** for constant terms. **Join conditions** shall be generated to deal with referencing object maps, and optional SPARQL filters are encapsulated in a specific **sparqlFilter condition**. Conditions apply to the xR2RML references mentioned in the mapping.

In sections 5 and 6 we continue the process to the concrete case of MongoDB. In this case, xR2RML references are JSONPath expressions. Joins are not supported by MongoDB and shall be executed by the query processing engine. Therefore we study further-on how to translate not-null and equality conditions on JSONPath expressions into valid MongoDB queries.

## 4 Binding triples maps to triple patterns

The  $trans_m$  function defined in the previous section (Definition 1) uses the  $getBoundTriplesMaps_m$  function that, given a SPARQL graph pattern  $gp$  and a triple pattern  $tp \in gp$ , returns the set of triples maps that are bound to  $tp$  in  $bind_m(gp)$ . Function  $bind_m$ , along with functions  $join$ ,  $reduce$  and  $compatible$  (defined in this section), implements the algorithm that binds a graph pattern to a set of triples map that are candidates to generate triples matching the graph pattern. It takes into account dependencies between triple patterns of the graph pattern, to reduce the number of candidate triples maps for each triple pattern, by pre-checking filters and join constraints implied by shared variables. Functions  $bind_m$ ,  $join$ ,  $reduce$  and  $compatible$  were introduced by Unbehauen et al [17] in the SPARQL-to-SQL context. In this section we define them in much deeper details and we extend them to fit in our context of an abstract query language. In particular, our method includes the static verification of the compatibility between a term map and a SPARQL filter, that [17] does not tackle, e.g. a SPARQL filter `isIRI(?person)` would be incompatible with a term map that generate literals.

We denote by  $TM.sub$ ,  $TM.pred$  and  $TM.obj$  respectively the subject map, the predicate map and the object map of triples map  $TM$ .  $TM.pred = TM.predicateObjectMap.predicateMap$ , and  $TM.obj = TM.predicateObjectMap.objectMap$ .

### Definition 5: function $bind_m$

Let  $m \in M$  be a set of xR2RML triples maps, and  $gp$  be a graph pattern.

$bind_m(gp)$  is the set of triple pattern bindings of  $gp$  under  $m$ , defined recursively as follows:

- $bind_m(gp) = bind_m(gp, true)$
- if  $gp$  consists of a single triple pattern  $tp$ ,  $bind_m(gp, f)$  is the pair  $(tp, TMSet)$  where  $TMSet = \{TM \mid TM \in m \wedge compatible(TM.sub, tp.sub, f) \wedge compatible(TM.pred, tp.pred, f) \wedge compatible(TM.obj, tp.obj, f)\}$
- if  $gp$  is  $(P_1 \text{ AND } P_2)$ ,  $bind_m(gp, f) = reduce(bind_m(P_1, f), bind_m(P_2, f)) \cup reduce(bind_m(P_2, f), bind_m(P_1, f))$
- if  $gp$  is  $(P_1 \text{ OPTIONAL } P_2)$ ,  $bind_m(gp, f) = bind_m(P_1, f) \cup reduce(bind_m(P_2, f), bind_m(P_1, f))$
- if  $gp$  is  $(P_1 \text{ UNION } P_2)$ ,  $bind_m(gp, f) = bind_m(P_1, f) \cup bind_m(P_2, f)$
- if  $gp$  is  $(P \text{ FILTER } f')$ ,  $bind_m(gp, f \wedge f')$

Function  $compatible$  is detailed in Definition 8, function  $reduce$  in Definition 7.

In our running example, function  $bind_m$  infers two triple pattern bindings:

```
bind_m(bgp) = { (tp1, {<#Staff>}) , (tp2, {<#Departments>}) }
```

### Definition 6: function $join$

Let  $m \in M$  be a set of xR2RML triples maps,  $tpb_1=(tp_1, TMSet_1)$  and  $tpb_2=(tp_2, TMSet_2)$  be triple pattern bindings with  $TMSet_1 \subseteq m$  and  $TMSet_2 \subseteq m$ ,  $V$  be the set of variables shared by  $tp_1$  and  $tp_2$ .

Let  $pos_{tp}: V \rightarrow \{sub, pred, obj\}$  be the function that returns the position of a variable  $v \in V$  in triple pattern  $tp$ .

$join(tpb_1, tpb_2)$  is the set of pairs  $(TM_1, TM_2) \in TMSet_1 \times TMSet_2$ , such that for each  $v \in V$ , it holds that  $compatible(TM_1.pos_{tp_1}(v), TM_2.pos_{tp_2}(v))$ .

In other words, function  $join$  returns the pair  $(TM_1, TM_2)$  if, for each variable  $v$  shared by  $tp_1$  and  $tp_2$ , the term maps associated to  $v$  in  $TM_1$  and  $TM_2$  are compatible, i.e. the term map of  $TM_1$  at the position of  $v$  in  $tp_1$  is compatible with the term map of  $TM_2$  at the position of  $v$  in  $tp_2$ .

Example:

```
tp1 = ?x knows ?y,      pos_tp1(?y) = obj,
tp2 = ?y knows <#me>,   pos_tp2(?y) = sub.
tpb1 = (tp1, {TM1}),   tpb2 = (tp2, {TM2})
join(tpb1, tpb2) = {(TM1, TM2)} if the object map of TM1 is compatible with the subject map of TM2.
```

Note that  $join(tpb_1, tpb_2)$  and  $join(tpb_2, tpb_1)$  contain the same pairs with the difference that in each pair the terms are switched.

**Definition 7: function reduce**

Let  $m \in M$  be a set of triples maps,  $tpb_1 = (tp_1, TMSet_1)$  and  $tpb_2 = (tp_2, TMSet_2)$  be triple pattern bindings with  $TMSet_1 \subseteq m$  and  $TMSet_2 \subseteq m$ .

$reduce(tpb_1, tpb_2)$  is the binding of  $tp_1$  to triples maps from the projection of the first component of pairs obtained from  $join(tpb_1, tpb_2)$ .

In other words, if  $tp_1$  and  $tp_2$  have a shared variable  $v$ , function  $reduce(tpb_1, tpb_2)$  returns the triple pattern binding 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 bindings of  $tp_2$ .

Example:

```
join(tpb1, tpb2) = {(TM1, TM2), (TM1, TM3)} => reduce(tpb1, tpb2) = (tp1, {TM1})
join(tpb2, tpb1) = {(TM2, TM1), (TM3, TM1)} => reduce(tpb2, tpb1) = (tp2, {TM2, TM3})
```

Function  $compatible(termMap, tpTerm, f)$  checks if a term map is compatible with a term of a triple pattern ( $tpTerm$ ) and a SPARQL filter  $f$ , i.e. that there is no contradiction between  $tpTerm$  and the term map, and between  $f$  and the term map. Note that [17] simply defines the compatibility of  $termMap$  and  $tpTerm$  by  $tpTerm \in range(termMap)$ , but no description of the  $range$  function is provided. Below we give a complete description of what it means in our context. In Definition 8 and Definition 9 we mention the *term type* of a term map. Recall that the term type may be explicitly stated with the `rr:termType` property, or have a default value as per the xR2RML language specification. For instance a template-valued term map has the `rr:IRI` default term type and a reference-valued term map has the `rr:Literal` default term type.

**Definition 8: compatibility between a term map, a triple pattern term and a SPARQL filter**

Let  $tpTerm$  be a term of a triple pattern,  $termMap$  be a term map of a triples map  $TM$  and  $f$  be a SPARQL filter.

It holds that  $termMap$  is compatible with  $tpTerm$  and  $f$ , denoted as  $compatible(termMap, tpTerm, f)$  if (1)  $tpTerm$  is a variable and  $termMap$  is compatible with filter  $f$  denoted by  $compatibleFilter(termMap, f)$ , or (2) if none of the following assertions is true:

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

Function  $compatibleFilter(termMap, f)$  checks if a term map is compatible with a SPARQL filter  $f$ , i.e. that there is no contradiction between the definition of the term map and the filter.



**Definition 9: compatibility between a term map and a SPARQL filter**

Let  $termMap$  be a term map and  $f$  be a SPARQL filter. It holds that  $termMap$  is compatible with  $f$ , denoted as  $compatibleFilter(termMap, f)$  if  $f = \text{“true”}$  or none of the following assertions is true:

- a necessary condition of  $f$  is  $isIRI(?var)$  and the term type of  $termMap$  is not  $rr:IRI$ ;
- a necessary condition of  $f$  is  $isLiteral(?var)$  and the term type of  $termMap$  is not  $rr:Literal$ ;
- a necessary condition of  $f$  is  $isBlank(?var)$  and the term type of  $termMap$  is not  $rr:BlankNode$ ;
- a necessary condition of  $f$  is  $lang(?var) = \text{“L”}$  or  $langMatches(lang(?var), \text{“L”})$ , and the language of  $termMap$  is either not defined or different from  $L$ ;
- a necessary condition of  $f$  is  $datatype(?var) = \langle T \rangle$  and the datatype of  $termMap$  is either undefined or different from  $\langle T \rangle$ ;

The compatibility between two term maps is defined by [17] as the condition:

$$range(termMap1) \cap range(termMap2) \neq \emptyset$$

Again, no description of the  $range$  function is provided, which leaves much room for interpretation. We give a complete description of what it means in our context.

**Definition 10: compatibility between term maps**

Let  $termMap1$  and  $termMap2$  be two term maps.

It holds that  $termMap1$  and  $termMap2$  are compatible, denoted as  $compatibleTermMaps(termMap1, termMap2)$  if none of the following assertions is true:

- (1)  $termMap1$  and  $termMap2$  have different term types ( $rr:Literal$ ,  $rr:BlankNode$ ,  $rr:IRI$ ,  $xrr:RdfList$ ,  $xrr:RdfSeq$ ,  $xrr:RdfBag$ ,  $xrr:RdfAlt$ ).
- (2)  $termMap1$  and  $termMap2$  have different language tags, or one has a language tag and the other does not.
- (3)  $termMap1$  and  $termMap2$  are both template-valued, and they have incompatible template strings.
- (4)  $termMap1$  (resp.  $termMap2$ ) is a *ReferencingObjectMap* and the subject map of its parent triples maps is not compatible with  $termMap2$  (resp.  $termMap1$ ), i.e.  $\neg compatible(termMap1.parentTriplesMap.subjectMap, termMap2)$ .

The negation of any of the assertions (1) to (4) is a sufficient condition to entail that two term maps are not compatible. Note that we could have considered the additional assertion (5):

*$termMap1$  and  $termMap2$  have different types (constant-valued, reference-valued or template-valued).*

In practice, if assertion (5) is true, then indeed both term maps will often generate different values, thus they are not compatible. However, in some contexts, assertion (5) may be true although term maps are compatible. For instance, a reference-valued term map returning a URL from the database and a template-valued term map building a URL from some other value may return some common values. Therefore, considering assertion (5) in our definition may lead to state that two term maps are not compatible although they are, in turn the evaluation result will lack some matching triples.

## 5 Translation of an abstract query into a MongoDB query

Function  $transTP_m$ , defined in section 3, returns a union of abstract  $\{FromPart, ProjectPart, WherePart\}$  queries. The *FromPart* contains the logical sources of the child and the optional parent triples maps, each logical source consisting of a concrete MongoDB query (property  $xrr:query$ ) and an optional iterator (property  $rml:iteraotr$ ). The *WherePart* is a set of conditions on JSONPath expressions, either  $isNotNull(query/JSONPath)$ ,  $equals(query/JSONPath, value)$ ,  $join(query/JSONPath, query/JSONPath)$  or  $sparqlFilter(query/JSONPath, filter)$ , where “query” stands for the child or parent queries of the *FromPart*.

*In the current status of this work, we do not consider SPARQL filters in the translation into the MongoDB query language.*

The process we define in this section first translates *isNotNull* and *equals* conditions of the *WherePart* into MongoDB queries. Since conditions of the *WherePart* are about JSONPath expression we have to investigate how to rewrite JSONPath expressions into equivalent MongoDB queries. For instance in our running example, the condition:

```
equals("Dunbar", child/[$['lastname','familyname']])
```

shall be translated into a concrete MongoDB query:

```
$or[{"lastname": {$eq: "Dunbar"}}, {"familyname": {$eq: "Dunbar"}}]
```

The generated MongoDB query shall augment the query of the *FromPart*. In this regards our example is trivial since the query in the `<#Staff>` triples map is empty: `"{}"`. The final query is exactly what we generated above.

The query produced by the translation process may contain several shortcomings: (i) the query may contain unnecessary complexity such as nested operators; (ii) it is not always possible to translate any arbitrary JSONPath expression into an equivalent MongoDB query; (iii) the query may contain \$where operators at any depth although this is not valid in the MongoDB query language. Thus, in a second step, the translation process performs various rewritings and optimizations.

## 5.1 The MongoDB query language

The MongoDB database comes with a rich set of APIs to allow applications to query a database in an imperative way. In addition, the MongoDB shell is a JavaScript interface that defines a declarative query language that we hereafter denote by the MongoDB query language<sup>9</sup>. In this work we refer to the language as described in the MongoDB Manual 3.0 (the latest at the time of writing). The `db.collection.find()` method accepts two parameters: a query string and a projection string, and returns a cursor to the matching documents. Optional modifiers amend the query to impose limits, skips, and sort orders. Both the query and projection parameters are JSON documents.

(1) The query parameter describes conditions about the documents to search for in the database. In the query document, specific query operators are marked with a heading '\$' character. We illustrate this with a few examples:

- `{"decade": {$exists: true}}`: matches all documents with a field "decade".
- `{"person.age": {$gte: 18}}`: matches all documents with a field "person" whose value is a document having a field "age" whose value is 18 or more.
- `{"staff.0.role": {$eq: "manager"}}`: matches all documents with an array "staff" whose first element is a document having a field "role" with value "manager".
- `{"staff": {$elemMatch: {"role": "developer"}}`: matches all documents with an array "staff" in which at least one element is a document having a field "role" with value "developer".

(2) The projection parameter specifies the fields from the matching documents to return. In this example request:

```
db.collection.find({"person.age": {$gte: 18}}, {"person.name": true})
```

the first parameter matches all documents about people whose age is at least 18, and the second parameter specifies that only their name must be returned: no other fields, including "age", are returned

The MongoDB documentation provides a rich description of the query language. Nevertheless, it lacks precision as to the formal semantics of some operators. For instance the query `{$or: [{"p.q": 10}, {"p.q": 11}]}` retrieves documents where field "p" is a document having a field "q" whose value is either 10 or 11. We may be tempted to write the same query in another way: `{"p": {$or: [{"q": 10}, {"q": 11}]}}`, however this query is invalid. It is unclear in the documentation why the \$or and \$and operators cannot be used as a condition on a field, but have to be at the top-level of the query document, or nested in an \$elemMatch, an \$and or an \$or operator. To the best of our knowledge, at the time of writing, there is no published work that clarifies the semantics of the language. Therefore, in Definition 11 we describe the subset of the query language that we use in our approach, and we underline some limitations and ambiguities. Operator keywords are bold, square brackets ('[', ']'), curly brackets ('{', '}') and characters

<sup>9</sup> <https://docs.mongodb.org/manual/tutorial/query-documents/>

“.”, “,”, “/” and “.” are part of the language. Parenthesis groups “(...)”, characters “\*”, “+” and “|” are the syntactic notation denoting occurrences and alternatives.

---

**Definition 11: Grammar of a subset of the MongoDB query language**


---

```

TOP_LEVEL_QUERY = { } |
                  { QUERY(, QUERY)* (, WHERE_QUERY)* } |
                  { WHERE_QUERY(, WHERE_QUERY)* }
QUERY            = FIELD_QUERY | OR_QUERY | AND_QUERY
FIELD_QUERY     = PATH: {OP: LITERAL} |
                  PATH: { $elemMatch: { QUERY(, QUERY)* } } |
                  PATH: { $regex: /REGEX/ }
OP              = $eq | $ne | $lt | $lte | $gt | $gte | $size
OR_QUERY        = $or: [ { QUERY } (, { QUERY } ) + ]
AND_QUERY       = $and: [ { QUERY } (, { QUERY } ) + ]
PATH            = "(FIELD_NAME|ARRAY_INDEX).(FIELD_NAME|ARRAY_INDEX)*"
WHERE_QUERY     = $where: JS_BOOL_EXP
LITERAL         = literal value possibly in double quotes,
                  including specific values null, true, false
FIELD_NAME      = valid JSON field name
ARRAY_INDEX     = positive integer value
JS_BOOL_EXP     = valid JavaScript boolean expression
REGEX           = Perl compatible regular expression

ARRAY_SLICE     = { PATH: { $slice: <nb_of_elts> } } | { PATH: { $slice: [ <skip>, <limit> ] } }

```

---

A sequence of comma-separated QUERY elements (in the top-level query and in the \$elemMatch operator) is implicitly interpreted as a logical AND between the elements. Additionally, the \$and operator performs a logical AND operation on an array of QUERY expressions and selects the documents that satisfy all the expressions in the array. The \$and operator is necessary when the same field or operator has to be specified in multiple expressions (as queries are valid JSON documents, thus they cannot have twice the same field name).

The \$elemMatch operator matches documents with an array field in which at least one element matches all the specified QUERY criteria.

The \$where operator passes a JavaScript expression or function to the query system. It provides greater flexibility than other operators. However, the JavaScript evaluation cannot take advantage of existing indexes and requires that the database processes the JavaScript expression for each document. This issue can seriously hinder performances, and MongoDB strongly recommends to use \$where only when the query cannot be expressed using another operator.

The \$where operator is valid only in the top-level query document: it cannot be used inside a nested query such as an \$elemMatch. This restriction makes a strong difference with SQL, and has a major impact on the rewriting process.

The ARRAY\_SLICE definition is separated from the above ones as an array slice does not apply in the query part but in the projection part of a MongoDB request (second parameter of the find method). For instance, query

```
db.collection.find({comments:{$size: 100}}, {comments:{$slice: 5}})
```

selects documents that have an array “comments” with 100 elements, and projects only the first five elements.

### **Ambiguous semantics of field names:**

The MongoDB query language allows ambiguous short-cut expressions to name paths in the JSON documents. For instance, query {“p”:{ \$eq: 3 }} matches documents where p is a field with value 3, such as {p:3}. Surprisingly it also

matches documents where  $p$  is an array wherein at least one element has value 3, e.g.  $\{p: [3, 4]\}$ , that would equally be matched by query  $\{ "p": \{ \$elemMatch: \{ \$eq: 3 \} \} \}$ . This gets even worse with a sequence of field names, as each field name may be considered for what it is, exactly one field, or for a short-cut for the elements of an array field. With this logic, query  $\{ "p.q": \{ \$eq: 3 \} \}$  matches several types of documents depending on how we interpret  $p$  and  $q$ , such as  $\{p: \{q: 3, r: 4\}\}$ ,  $\{p: [\{q: 3, r: 4\}, \{q: 5\}]\}$  and  $\{p: [\{q: [3, 4], r: 5\}, \{q: [6, 7]\}]\}$ .

These simple examples entail an important conclusion: given the ambiguous notation of the MongoDB query language, it is hardly possible to write a MongoDB query whose semantics would be strictly equivalent to a SPARQL query. Consequently, whatever the rewriting we can come up with, we shall always have to run the initial SPARQL query against the generated triples to make sure that we rule out triples generated because of this ambiguity, but that do not match the SPARQL query.

## 5.2 The JSONPath language

JSONPath<sup>10</sup> is a domain specific language designed to read, parse and extract data from JSON documents. It was defined in 2007 by Stefan Goessner as an analogy to the XPath<sup>11</sup> standard for XML documents. As of today JSONPath is not a standard, however its definition remains stable and a large community provides and maintains implementations for various programming languages. Definition 12: describes the grammar of JSONPath. Bold characters are part of the language. In particular note that characters “(“ and “)” are part of the language in the FILTER and CALC\_INDEX expressions, whereas in other expressions they simple denote groups. Similarly, the “\*” character is part of the language in expression WILDCARD, but denotes 0 to any occurrences in other expressions.

---

### Definition 12: JSONPath grammar

---

JSONPATH	= $\$(WILDCARD \mid FIELD\_NAME \mid ARRAY\_INDEX \mid DESCENDANT \mid FIELD\_ALT \mid INDEX\_ALT \mid ARRAY\_SLICE \mid FILTER \mid CALC\_INDEX)^*$
WILDCARD	= $.^* \mid [^*]$
FIELD_NAME	= $FIELD\_NAME\_DOT \mid FIELD\_NAME\_BRKT$
FIELD_NAME_DOT	= $.\langle name \rangle$
FIELD_NAME_BRKT	= $[\langle name \rangle]$
ARRAY_INDEX	= $[\langle int \rangle]$
DESCENDANT	= $..$
FIELD_ALT	= $[\langle name \rangle (, \langle name \rangle)^+]$
INDEX_ALT	= $[\langle int \rangle (, \langle int \rangle)^+]$
ARRAY_SLICE	= $[\langle start \rangle : \langle end \rangle : \langle step \rangle] \mid [\langle start \rangle : \langle end \rangle] \mid [\langle start \rangle : ]$
FILTER	= $[?(\langle script \ expression \rangle)]$
CALC_INDEX	= $[(\langle script \ expression \rangle)]$

---

Let us give a few illustrating examples:

- $\$.names.^*$ : selects all elements of array “names” like in:  $\{names: [ "mark", "john" ]\}$ , or all fields of document “names” like in  $\{names: \{firstname: "mark", lastname: "john"\}\}$ .
- $\$.books[1, 3]$ : selects the second (index 1) and fourth (index 3) elements of array “books”.
- $\$.books[1: 3]$ : selects all books from index 1 (inclusive) until index 3 (exclusive), that is at indexes 1 and 2.
- $\$.books[( @.length - 1 )]$  or  $\$.books[-1: ]$ : select the last element of array “books”. In the “[()]” notation, “@” refers to the parent element “members”.
- $\$.team[?(@.members <= 10)].name$ : select the name of teams that have 10 members or less, i.e. “team” is an array, among its elements we select those that have a field “members” whose value is 10 or less, and finally we select the field “name” of those elements. Unlike above, in the “[?()]” notation “@” refers to elements of the array.

<sup>10</sup> <http://goessner.net/articles/JsonPath/>

<sup>11</sup> <http://www.w3.org/TR/1999/REC-xpath-19991116/>

- `$.author`: selects all “author” fields anywhere in the document.

In an array slice, if the <start> is omitted it defaults to 0, e.g. `$.books[:2]` selects the first two books. If <end> is omitted it defaults to the index of the last element of the array. <end> can be positive (the index is counted from the start of the array), or negative (the index is counted from the end of the array), e.g. `$.books[-2:]` selects the last two books.

## **Restrictions on the usage of JSONPath expressions**

### **Script expressions:**

The FILTER expression filters elements of an array based on <script expression> that must evaluate to a boolean. CALC\_INDEX selects the element of an array at index <script expression> that evaluates to a positive integer. In both cases, the language definition says <script expression> is written in “the syntax of the underlying script engine”. This design choice has a strong shortcoming: it binds the language definition to its implementations, since the underlying script engine depends on the implementation, and in the worst case there may even not be any underlying script engine at all. That made sense in the initial JavaScript implementation of Goessner, but this is subject to various interpretations in other implementations. For instance in the Java port<sup>12</sup> of Goessner's implementation, developers have chosen to implement a very limited subset of JavaScript.

In our rewriting approach, we stick to the idea that those expressions are JavaScript, keeping in mind that its support may vary depending on the JSONPath implementation that is being used.

### **Wildcard semantics:**

In JSONPath, the wildcard, '\*' is equally applicable to arrays and documents. In an array it stands for any element of the array, while in a document it stands for any field of the document. In MongoDB conversely, documents and arrays are not treated equally: the \$elemMatch operator applies specifically to arrays, and it is not possible to match any field in a document (there is no equivalent of the “\*” for a document). Therefore, to be able to translate JSONPath expressions into MongoDB, we restrict the use of the wildcard to arrays only, which is its most common usage.

### **Filters:**

In the JSONPath reference, it is unclear whether the filter notation [?(<script expression>)] applies to arrays, or to arrays and documents. Some implementations apply both with somehow confusing semantics, e.g. in the expression `$.p[?(@.q)]`:

- if “p” is an array then “@” refers to each of its elements, meaning that only elements with a field “q” are matched. The drawback is that it is not possible to write a condition about an element given by its index, e.g. to match arrays in which the 11<sup>th</sup> element is 0, we would like to write `$.p[?(@[10] == 0)]`, which is invalid because in that case “@” should refer to the array p but not to its elements.
- Conversely if “p” is a document, “@” refers to “p” itself, meaning that “p” matches only if it is a document with a field “q”.

Besides some tests show that different implementations have made different interpretations in this matter. To get rid of any confusion, in this work we restrict the usage of filters to arrays only. Therefore expressions like `$.p[?(...)]` shall be understood a “p” being an array field, the “@” character refers to its elements.

### **Root element of JSON documents:**

In MongoDB the root element of a document cannot be an array, e.g. `["mark","john"]` is not a valid MongoDB document, but `{"people":["mark","john"]}` is valid. Consequently, the JSONPath expressions we consider must not start with array-specific elements. For instance, expressions `“$[0]”` and `“$[1,3,5]”` are invalid in our context. Additionally, given the above restriction on the wildcard, expressions starting like `“$.*”` or `“$[*]”` are not supported in our context.

<sup>12</sup> <https://github.com/jayway/JsonPath>

**Descendent operator:**

Unlike JSONPath, MongoDB does not provide a descendent operator that would look for a pattern at any depth of the documents. Consequently, our rewriting method does not support JSONPath expressions using the “.” operator.

**5.3 Conventions and formalism**

We define an abstract hierarchical representation of a MongoDB query. This representation allows for handy manipulation during the query construction and optimization phases. Definition 13 lists the clauses of this representation as well as their translation into a concrete query string when relevant.

**Definition 13: Abstract MongoDB query**


---

AND(<expr1>, <expr2>, ...)	→ <b>\$and</b> : [<expr1>, <expr2>, ...]
OR(<expr1>, <expr2>, ...)	→ <b>\$or</b> : [<expr1>, <expr2>, ...]
WHERE(<JavaScript expr>)	→ <b>\$where</b> : '<JavaScript expr>'
ELEMMATCH(<exp1>, <exp2>, ...)	→ <b>\$elemMatch</b> : {<exp1>, <exp2>, ...}
FIELD(p1) FIELD(p2)... FIELD(pn)	→ "p1.p2...pn":
SLICE(<expr>, <number>)	→ <expr>: { <b>\$slice</b> : <number>}
COND(equals(v))	→ <b>\$eq</b> : v
COND(isNotNull)	→ <b>\$exists</b> : true, <b>\$ne</b> : null
EXISTS(<expr>)	→ <expr>: { <b>\$exists</b> : true}
NOT_EXISTS(<expr>)	→ <expr>: { <b>\$exists</b> : false}
COMPARE(<expr>, <op>, <v>)	→ <expr>: {<op>: <v>}
NOT_SUPPORTED	→ ∅
CONDJS(equals(v))	→ == v
CONDJS(equals("v"))	→ == "v"
CONDJS(isNotNull)	→ != null
UNION(<query1>, <query2>, ...)	<i>Same semantics as OR, but processed by the query processing engine.</i>

---

In the COMPARE clause definition, <op> stands for one of the MongoDB query compare operators: \$eq, \$ne, \$lte, \$lt, \$gte, \$gt, \$size and \$regex. The example abstract query below matches all documents where “p” is an array field whose first element is 10, and “q” is an array field in which at least one element has value “val”.

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

Its concrete representation is:

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

The NOT\_SUPPORTED clause helps keep track of any location, within the abstract query, where the condition cannot be translated into an equivalent MongoDB query element. It shall be used in the optimization phase.

The UNION clause represents a logical OR that shall be computed by the query processing engine based on the result of queries <query1>, <query2>, etc. It can be produced by the abstract MongoDB query optimization (Algorithm 4). Note that this UNION clause applies to set of JSON documents retrieved from the database, whereas the UNION operator generated by function  $\text{trans}_m$  applies to triples.

In the definition of the translation rules we use the following notations:

- **<cond>**: is a condition to translate into MongoDB: either *isNotNull* or *equals(value)*.
- **<JP>**: denotes a possibly empty JSONPath expression.
- **<JP:F>**: denotes a non-empty JSONPath sequence of field names and array indexes (F), e.g. .p.q.r, .p[10][ "r"].
- **<bool expr>**: denotes a JavaScript expression that evaluates to a boolean.

- **<num expr>**: denotes a JavaScript expression that evaluates to a positive integer.

Finally, we define the function **replaceAt**(<rep>, <path>), that replaces any occurrence of the '@' character with <rep> in string <path>. E.g. `replaceAt("this.people", "@ < 10")` returns "this.people < 10".

## 5.4 Query translation rules

Given the subset of the MongoDB query language that we consider in section 5.1, the JSONPath language and the restrictions mentioned in section 5.2, and the formalism defined in section 5.3, in this section we define the recursive function *trans*(JSONPath expression, <cond>) that translates a condition <cond> applied to a JSONPath expression into an abstract MongoDB query. <cond> stands for either *isNotNull* or *equals*(value). Function *trans* consists of a set of rules detailed in Algorithm 2, that apply if the JSONPath expression matches a certain pattern. The JSONPath expression is checked against the patterns in the order of the rules (0 to 9). When a match is found the rule is applied and the search stops.

Before getting into the details, let us illustrate the approach using the running example. As already seen, the translation of triple pattern tp1 entails three conditions:

```
isNotNull(parent/$.code),
equals("Dunbar", child/$['lastname','familyname']),
join(child/$.manages.*, parent/$.dept)
```

The *join* condition cannot be processed by MongoDB thus it shall be computed by the query processing engine afterwards. The *isNotNull* and *equals* conditions however can be translated. Let us consider condition `isNotNull(parent/$.code)`. In Algorithm 2, it amounts to evaluating `trans($.code, isNotNull)` that goes through the following steps:

- Rule R0 first matches, returning `trans(.code, isNotNull)`.
- Then, rule R8 matches, it returns `FIELD(code) trans(∅, isNotNull)`.
- Lastly, rule R1 returns the abstract MongoDB query `FIELD(code) COND(isNotNull)`.

Applying Definition 13 to the abstract MongoDB entails the final concrete query:

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

Following the same algorithm, the second condition, `equals("Dunbar", child/$['lastname','familyname'])`, will be translated into the abstract query:

```
OR(FIELD(lastname) COND(equals("Dunbar")), FIELD(familyname) COND(equals("Dunbar")))
```

that is in turn translated into the concrete query:

```
$or: [{"lastname": {$eq: "Dunbar"}}, {"familyname": {$eq: "Dunbar"}}]
```

---

### Algorithm 2: Translation of a condition on a JSONPath expression into an abstract MongoDB query (function *trans*(JSONPath expression, <cond>))

---

- 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 *Heading JavaScript filter on array elements, e.g. \$.p[?(@.q)].r*  
**trans**([?(<bool\_expr>)]<JP>, <cond>) → **ELEMMATCH**(**trans**(<JP>, <cond>), **transJS**(<bool\_expr>))
- R5 *Array slice*  
 (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>)
- R6 *Calculated array index, e.g. \$.p[@.length - 1].q*  
 (a) **trans**(<JP1>[(<num\_expr>)]<JP2>, <cond>) → **NOT\_SUPPORTED**  
     *if <JP1> contains a wildcard or a 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**(<JP1:F>[(<num\_expr>)]<JP2:F>, <cond>) →  
     **AND**(**EXISTS**(<JP1:F> ),  
     **WHERE**('this<JP1:F>[**replaceAt**("this<JP1:F>", <num\_expr>)]<JP2:F> **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, the current expression is not supported*  
**trans**(<JP>, <cond>) → **NOT\_SUPPORTED**
- 

#### 5.4.1 Rule R0

Rule R0 is the entry point of the translation process since a valid JSONPath expression starts with a “\$” character. At this stage, invalid or unsupported JSONPath expressions (see restrictions in section 5.2) shall be taken care of.

#### 5.4.2 Rule R1

Conversely, rule R1 is the termination point: when the JSONPath expression has been fully parsed, the last element that is created is the condition in MongoDB, like “\$eq: value” for an equality condition, or “\$exists:true, \$ne:null” for a not-null condition.



### 5.4.3 Rule R2

A field alternative or array index alternative is translated into an \$or operator. As underlined in section 5.1, the \$or operator cannot be used as a condition on a field, but has to be either at the top-level query or nested in an \$elemMatch, \$and or \$or operator. For this reason, a sequence of field names and array indexes (<JP:F>) must precede the alternative pattern ([`"p","q",...`] or [`[i,j,...`])). In the rewriting, the <JP:F> sequence is prepended to each of the \$or members. In the example below the “p” stands for the <JP:F> term:

Condition

```
equals($.p.["q", "r"], 10)
```

is translated into:

```
$or: [{"p.q": {$eq: 10}}, {"p.r": {$eq: 10}}]
```

Note that no assumption is made as to what may come after the alternative pattern, this is denoted in the rule by JSONPath <JP> following the alternative pattern.

### 5.4.4 Rule R3

Rule R3 matches an expression with a heading field alternative or array index alternative. Contrary to rule R2, the alternative pattern is not preceded by a <JP:F> sequence. This case occurs when the alternative is either the first pattern in the JSONPath expression, or when it comes after a term such as a JavaScript filter (R4), an array slice (R5) or a wildcard (R7). Example:

Condition

```
equals($.p.*["q", "r"], 10)
```

is translated into:

```
"p": {$elemMatch: {$or: [{"q": {$eq: 10}}, {"r": {$eq: 10}}]}}
```

### 5.4.5 Rule R4

A JavaScript (JS) filter is a boolean condition evaluated against elements of an array, where the “@” character stands for each array element, e.g. “\$.people[?(@.role)]” matches all elements of array “people” that are documents having a field “role”. Since a JS filter specifies a condition on all array elements, it is translated into a MongoDB query embedded in an \$elemMatch operator. Function *transJS* (see section 5.4.11) parses the JS expression and translates it. Example:

Condition

```
equals($.p[?(@.q)].r.*, "value")
```

is translated into:

```
"p": {$elemMatch: {
  "r": {$elemMatch: {$eq:"value"}},
  "q": {$exists:true}}}
```

R4 produces the first \$elemMatch as well as the condition "q":{\$exists:true}. The second \$elemMatch is produced by rule R7 when processing the wildcard.

### 5.4.6 Rule R5

JSONPath and MongoDB query language have two different ways of denoting array slices. JSONPath uses notation [`<start>:<end>:<step>`], where any of the three terms are optional, and <start> and <end> may be negative. MongoDB uses notation `{$slice: <count>}` or [`<start>`, `<count>`], <count> may be negative in the first notation only, <start> may be negative. In JSONPath and MongoDB a negative value means “starting from the end of the array”. Due to these discrepancies, the rewriting of JSONPath slices into MongoDB projections has limitations explicated in the table below:

Semantics	JSONPath	MongoDB query language
From index 0 to index n (first n elements)	<code>array[:n]</code>	<code>"array" : {\$slice: n}</code>
Last n elements	<code>array[-n:]</code>	<code>"array" : {\$slice: -n}</code>

From index m until the last element	<code>array[m:]</code>	n/a
From index m to index n	<code>array[m:n]</code>	n/a
From index m to index n by step s	<code>[m:n:s]</code>	n/a

Consequently rules R5 (a) and (b) only cover the first two lines of the table. Other forms of JSONPath slice shall be treated in the default rule R9.

The JSONPath array slice notation is rewritten into the `$slice` operator that, unlike in other rules, is used as a projection parameter of the MongoDB `find()` method. Rule R5 must translate the JSONPath expression that comes before the array slice (`<JP:F>`) as well as the subsequent JSONPath expressions (`<JP>`) to generate the query parameter of the `find()` method. It does so by replacing the array slice by a wildcard `".*"`: `trans(<JP:F>.*<JP>, <cond>)`. Hence, the query part applies to the whole array, while the projection part shall select only the expected elements.

### 5.4.7 Rule R6

A JSONPath calculated array index selects an element from an array using a JavaScript expression that evaluates to a positive integer. The script expression uses the `"@"` character instead of `"this"` to refer to the array element.

Let us consider this example query: `equals($.staff[@.length - 1].name, "John")`, that matches all documents in which the last element of array `"staff"` has a field `"name"` with value `"John"`. In MongoDB, there is no way to retrieve the size of an array nor to calculate such an index (the `$size` operator is not relevant here as it specifies a condition on the size of an array). The only way to specify a condition on an element whose index is calculated is to use the `$where` operator. For instance,

Condition

```
equals($.staff[@.length - 1].name, "John"),
```

shall translated by rule R6(c) into:

```
$and: [{"staff":{$exists: true}}, {$where:"this.staff[this.staff.length - 1].name == 'John'"}]
```

Here we notice that rule R6 (b and c) produces a `$where` operator nested in an `$and` operator. As already underlined, the `$where` operator is valid only in the top-level query. We show in section 5.5 that we can rewrite a query containing a `$where` nested in a combination of `$and` and `$or` operators into a union of MongoDB queries in which a `$where` shows only in the top-level query. If a rule produces a `$where` inside an `$elemMatch` operator, there is no way we can rewrite this query into multiple valid queries. The `$elemMatch` operator is used to translate either a JS filter (R4) or wildcard (R7). Consequently, rule R6(a) makes those cases impossible by returning `NOT_SUPPORTED` in case a calculated array index is preceded by a wildcard or a filter.

If the calculated array index is followed by a JSONPath expression, that subsequent expression has to be part of the JavaScript expression in the `$where` operator. This is exemplified by the `"name"` field in the example above. More generally, anything that follows the calculated array index should be rewritten in JavaScript. This is not always possible however, as illustrated by the two examples below:

(1) Condition `equals($.p[@.length - 1].*, "val")`, could be rewritten in:

`$where:{"this.p[this.p.length-1].* == 'val'"}`. This query is invalid since there is no equivalent to the wildcard in JavaScript.

(2) Similarly, condition `equals($.p[@.length - 1].r[?(@.q)].s, "val")` could be rewritten in:

`$and: [{p:$exists}, {$where: "this.p[this.p.length - 1].r[?(@.q)].s == 'val'"}]`. But again this query is invalid since there is no JavaScript equivalent to the JSONPath notation `?(@.q)`.

Therefore, although JavaScript functions could be written to address this kind of issue, we choose not to go through this solution at this stage and further discuss this choice in section 7.4. Therefore, in rule R6(c) we restrict terms that follow a calculated array index to a sequence of field names or array indexes, denoted `<JP2:F>`.

### 5.4.8 Rule R7

As mentioned in section 5.2, the use of the wildcard is restricted to the context of arrays. Hence, rule R7 simply translates a heading wildcard into an \$elemMatch operator.

### 5.4.9 Rule R8

Other field names and array indexes are translated into their equivalent dot-separated MongoDB path. Example: condition `isNotNull($p[5]["s"])` is translated into `"p.5.s": {$exists: true}`.

### 5.4.10 Rule R9

Rule R9 is the default rule. In case no other rule matched, the translation of the JSONPath expression to MongoDB query language is not supported. This applies in the following cases:

- A calculated array index is preceded or followed by a wildcard, an alternative or a JavaScript filter, as explained in rule R6.
- Unsupported array slice notation such as `[m:n]`.
- JSONPath expressions entailing that the root document is an array and not a document, such as `$.*`, `$(1,2,...)`, `$(? (...))` and `$( (...))`.

### 5.4.11 Translation of a JavaScript filter to MongoDB

Recursive function *transJS* translates a JavaScript filter into a MongoDB query. It consists of a set of rules, explicated in Algorithm 3, that apply if the JavaScript expression matches a certain pattern. The JavaScript expression is checked against the patterns in the order of the rules. When a match is found the corresponding rule is applied and the search stops.

In the rules definitions we use the following notations:

- **<JSpath>**: denotes a non-empty JavaScript sequence of field names and array indexes, e.g. `'p.q.r'`, `'p[10]'`.
- The **dotNotation(<JS\_expr>)** function converts a JavaScript path to a MongoDB query path consisting of field names and array indexes in dot notation. It removes the optional heading dot. e.g. `dotNotation(p[5]r)` returns `"p.5.r"`.
- The **transJsOp(op)** functions converts a JavaScript comparison operator to its MongoDB equivalent: `===`  $\rightarrow$  `$eq`, `==`  $\rightarrow$  `$eq`, `!=`  $\rightarrow$  `$ne`, `<=`  $\rightarrow$  `$lte`, `>=`  $\rightarrow$  `$gte`, `<`  $\rightarrow$  `$lt`, `>`  $\rightarrow$  `$gt`, `~=`  $\rightarrow$  `$regex`.

The expressiveness of the MongoDB query language in terms of comparison is quite limited compared to JavaScript boolean conditions. As a result, when a JavaScript comparison cannot be turned in an equivalent MongoDB query, the rule returns the NOT\_SUPPORTED clause that shall be used later on during the final translation phase.

---

#### Algorithm 3: Translation of a JavaScript filter into a MongoDB query (function transJS)

---

J0	<b>transJS(&lt;JS_expr1&gt; &amp;&amp; &lt;JS_expr2&gt;)</b> $\rightarrow$ <b>AND(transJS(&lt;JS_expr1&gt;), transJS(&lt;JS_expr2&gt;))</b>
J1	<b>transJS(&lt;JS_expr1&gt;    &lt;JS_expr2&gt;)</b> $\rightarrow$ <b>OR(transJS(&lt;JS_expr1&gt;), transJS(&lt;JS_expr2&gt;))</b>
J2	<b>transJS(@&lt;JS_expr1&gt; &lt;op&gt; @&lt;JS_expr2&gt;)</b> $\rightarrow$ <b>NOT_SUPPORTED</b> <i>where &lt;op&gt; stands for one of {==, ===, !=, !==, &lt;=, &lt;, &gt;=, &gt;, %}</i>
J3	<b>transJS(@&lt;JSpath&gt;)</b> $\rightarrow$ <b>EXISTS(dotNotation(&lt;JSpath&gt;))</b>
J4	<b>transJS(!&lt;JSpath&gt;)</b> $\rightarrow$ <b>NOT_EXISTS(dotNotation(&lt;JSpath&gt;))</b>
J5	(a) <b>transJS(@&lt;JSpath&gt;.length == &lt;i&gt;)</b> $\rightarrow$ <b>COMPARE(dotNotation(&lt;JSpath&gt;), \$size, &lt;i&gt;)</b> (b) <b>transJS(@&lt;JSpath&gt;.length &lt;op&gt; &lt;i&gt;)</b> $\rightarrow$ <b>NOT_SUPPORTED</b> <i>where &lt;op&gt; stands for one of {!=, &lt;=, &lt;, &gt;=, &gt;, %}</i>
J6	<b>transJS(@&lt;JSpath&gt; &lt;op&gt; &lt;v&gt;)</b> $\rightarrow$ <b>COMPARE(dotNotation(&lt;JSpath&gt;), transJsOp(&lt;op&gt;), &lt;v&gt;)</b>
J7	<b>transJS(&lt;JS_expr&gt;)</b> $\rightarrow$ <b>NOT_SUPPORTED</b>

---

Rules J0 and J1 deal with the logical AND and OR JavaScript operators.

Rule J2 addresses the comparison of two document fields or two array fields such as “@.name != @.login”. This is not permitted in MongoDB query language, yet it is possible to translate this condition using the \$where operator. Typically rule J2 could return:

```
AND(EXISTS(<JS_expr1>), EXISTS(<JS_expr2>), WHERE("this<JS_expr1> <op> this<JS_expr2>"))
```

However the *transJS* function is used only in the context of an \$elemMatch, and the \$where operator is valid only in the top-level query. Therefore, rule J2 returns NOT\_SUPPORTED.

Rules J3 and J4 deal with existential comparisons.

Rule J5 addresses tests on the length of an array field. The MongoDB \$size operator allows for an equality test on the length of an array, but other types of comparison are not allowed. Similarly to the discussion above regarding rule J2, a \$where operator could be used in J5(b) to return:

```
WHERE(this<JSpath>.length <op> <i>)
```

But again, the \$where operator is valid only in the top-level query, consequently rule J 5(b) returns NOT\_SUPPORTED.

Rule J6 addresses all other types of supported comparison between a field and a literal value <v>.

Finally, rule J7 applies when no other rule matched. It is used as the default for all non-supported types of JavaScript expression.

## 5.5 Query optimization and translation to a concrete MongoDB query

Functions *trans()* and *transJS()*, defined in section 5.4, translate a condition on a JSONPath expression into an abstract MongoDB query. Before rewriting the abstract query into a concrete query, several potential issues must be addressed:

- (i) An abstract query may contain unnecessary complexity, such as nested ORs, nested ANDs, sibling WHEREs, etc., that can hamper performances.
- (ii) An abstract query may contain operators NOT\_SUPPORTED, indicating that a part of the JSONPath expression could not be translated into an equivalent MongoDB operator. Depending on the position of such an operator in the query, we rewrite the query into a concrete query that shall return all matching documents (the certain answers), as well as possibly non-matching documents that shall be ruled out afterwards.
- (iii) The WHERE operator may be nested beneath a sequence of ANDs and/or ORs, which is not valid in the MongoDB query language.

Those issues are addressed by means of two sets of rewriting rules, O1 to O5 and W1 to W6, defined in sections 5.5.1 and 5.5.2 respectively. Lastly, function *rewrite* (section 5.5.3) iteratively uses those rules to perform all possible rewritings and ultimately generate either one concrete MongoDB query or a union of concrete MongoDB queries.

### 5.5.1 Query optimization

Issues (i) and (ii) are addressed by a set of rewriting rules defined in Algorithm 4. A rule applies to a query Q when Q matches the pattern in the head of the rule.

**Algorithm 4: Optimization of an abstract MongoDB query**

The “ $\rightarrow$ ” arrow means “is rewritten as”.

- 
- 01 Flatten nested AND, OR and UNION clauses:  
 $\mathbf{AND}(C_1, \dots C_n, \mathbf{AND}(D_1, \dots D_m)) \rightarrow \mathbf{AND}(C_1, \dots C_n, D_1, \dots D_m)$   
 $\mathbf{OR}(C_1, \dots C_n, \mathbf{OR}(D_1, \dots D_m)) \rightarrow \mathbf{OR}(C_1, \dots C_n, D_1, \dots D_m)$   
 $\mathbf{UNION}(C_1, \dots C_n, \mathbf{UNION}(D_1, \dots D_m)) \rightarrow \mathbf{UNION}(C_1, \dots C_n, D_1, \dots D_m)$
- 02 Merge ELEMATCH with nested AND clauses:  
 $\mathbf{ELEMATCH}(C_1, \dots C_n, \mathbf{AND}(D_1, \dots D_m)) \rightarrow \mathbf{ELEMATCH}(C_1, \dots C_n, D_1, \dots D_m)$ .
- 03 Group WHERE clauses:  
 $\mathbf{OR}(\dots, \mathbf{WHERE}("W1"), \mathbf{WHERE}("W2")) \rightarrow \mathbf{OR}(\dots, \mathbf{WHERE}("(W1) || (W2)"))$ .  
 $\mathbf{AND}(\dots, \mathbf{WHERE}("W1"), \mathbf{WHERE}("W2")) \rightarrow \mathbf{AND}(\dots, \mathbf{WHERE}("(W1) \&\& (W2)"))$ .  
 $\mathbf{UNION}(\dots, \mathbf{WHERE}("W1"), \mathbf{WHERE}("W2")) \rightarrow \mathbf{UNION}(\dots, \mathbf{WHERE}("(W1) || (W2)"))$ .
- 04 Replace AND, OR or UNION clauses of one term with the term itself.  
 This may occur as a consequence of the flattening of nested clauses or the grouping of WHERE clauses.
- 05 Remove NOT\_SUPPORTED clauses:  
 -  $\mathbf{AND}(C_1, \dots C_n, \mathbf{NOT\_SUPPORTED}) \rightarrow \mathbf{AND}(C_1, \dots C_n)$ : since  $C_1 \wedge \dots \wedge C_n \supseteq C_1 \wedge \dots \wedge C_n \wedge N$ , this rewriting widens the condition. Hence, all matching documents (the certain answers) are returned, in addition to possibly non-matching documents.  
 -  $\mathbf{ELEMATCH}(C_1, \dots C_n, \mathbf{NOT\_SUPPORTED}) \rightarrow \mathbf{ELEMATCH}(C_1, \dots C_n)$ : same reason as above given that an AND implicitly applies to members of an ELEMATCH.  
 -  $\mathbf{OR}(C_1, \dots C_n, \mathbf{NOT\_SUPPORTED}) \rightarrow \mathbf{NOT\_SUPPORTED}$ . Contrary to the AND and ELEMATCH cases, we cannot simply remove the NOT\_SUPPORTED. The query would only return a subset of the matching documents since  $C_1 \vee \dots \vee C_n \subseteq C_1 \vee \dots \vee C_n \vee N$ . Instead, we replace the whole OR clause with a NOT\_SUPPORTED clause. This way, the NOT\_SUPPORTED issue is raised up to the parent clause, and it shall be managed at the next execution of the function. Iteratively, we raise up a NOT\_SUPPORTED clause until it is eventually removed (cases AND and ELEMATCH above), or it ends up in the top-level query. The latter is the worst case in which the query shall retrieve all documents.  
 -  $\mathbf{UNION}(C_1, \dots C_n, \mathbf{NOT\_SUPPORTED}) \rightarrow \mathbf{NOT\_SUPPORTED}$ : same reason as above.  
 -  $\mathbf{FIELD}(\dots) \dots \mathbf{FIELD}(\dots) \mathbf{NOT\_SUPPORTED} \rightarrow \mathbf{NOT\_SUPPORTED}$
- 

We illustrate Algorithm 4 in a dedicated example. Assume we wish to translate the condition below into a concrete MongoDB query:

```
equals($.teams.0[?(@.level=="beginner" && @.score>=3 && @.isPlayer<>@.isGoal)].name, "john")
```

The *trans* function translates this condition into an abstract MongoDB query. Below we detail the translation and mention the rules applied at each step:

```
trans($.teams.0[?(@.level=="beginner" &&
  @.score>=3 && @.isPlayer<>@.isGoal)].name, equals("john")) =
R0,R8 FIELD(teams.0) trans([?(@.level=="beginner" &&
  @.score>=3 && @.isPlayer<>@.isGoal)].name, equals("john")) =
R4 FIELD(teams.0) ELEMATCH( trans(.name, equals("john")),
  transJS([?(@.level=="beginner" && @.score>=3 && @.isPlayer<>@.isGoal)])) =
R8,R1 FIELD(teams.0) ELEMATCH( FIELD(name) COND(equals, "john"),
  transJS([?(@.level=="beginner" && @.score>=3 && @.isPlayer<>@.isGoal)])) =
J0,J6 FIELD(teams.0) ELEMATCH(FIELD(name) COND(equals, "john"),
  AND(COMPARE(level, ==, "beginner"), AND(COMPARE(@.score, >=, 3), NOT_SUPPORTED)))
```

Notice that J6 translates condition  $@.isPlayer<>@.isGoal$  into a `NOT_SUPPORTED` clause since MongoDB cannot compare fields of a JSON document. From this stage, rule O1 flattens nested ANDs, and rule O2 removes the unnecessary AND clause beneath the `ELEMMATCH`:

```
O1  FIELD(teams.0) ELEMMATCH(FIELD(name) COND(equals, "john"),
    AND(COMPARE(level, ==, "beginner"), COMPARE(score, >=, 3), NOT_SUPPORTED)) =
O2  FIELD(teams.0) ELEMMATCH(FIELD(name) COND(equals, "john"),
    COMPARE(level, ==, "beginner"), COMPARE(score, >=, 3), NOT_SUPPORTED) =
```

Lastly, rule O5 takes care of removing the `NOT_SUPPORTED` clause:

```
O5  FIELD(teams.0) ELEMMATCH(FIELD(name) COND(equals, "john"),
    COMPARE(level, ==, "beginner"),
    COMPARE(score, >=, 3))
```

This abstract MongoDB query can now be rewritten into the following concrete query:

```
"teams.0": {$elemMatch: {"name":{$eq:"john"}, "level":{$eq:"beginner"}, "score":{$gte:3}}}
```

### 5.5.2 Pull up WHERE clauses

By construction, a WHERE clause cannot be nested in an `ELEMMATCH` clause (rule R6). In addition, Algorithm 4 flattens nested OR and nested AND clauses, and merges sibling WHERE clauses. Consequently, a WHERE clause may be either in the top-level query (the query is thus executable) or it may appear in one of the following patterns:  $OR(..., W, ...)$ ,  $AND(..., W, ...)$ ,  $OR(..., AND(..., W, ...), ...)$ ,  $AND(..., OR(..., W, ...), ...)$ , where “W” stands for a WHERE clause. In the case of those patterns, we have to “pull up” WHERE clauses to the top-level query, in order to address issue (iii).

Rewritings make use of a new clause, `UNION`, that we describe here: its semantics is equivalent to that of the OR clause, although the OR is processed by the MongoDB query (as an `$or` operator), while the `UNION` is computed outside of the database, by the query processing engine: the result of evaluating `UNION(<query1>, <query2>)` is the union of the results produced by evaluating `<query1>` and `<query2>` separately against the MongoDB database.

Recall that an AND clause in the top-level query can be replaced with its members, since the implicit semantics of the top-level query is to apply a logical AND between its members. Therefore, it is sufficient to come up with query rewritings that bring all WHERE clauses to the top-level or in an AND of the top-level query. To give an intuition of the method, the example below shows the rewriting of simple queries. “W” stands for a WHERE clause, “C” and “D” for any sub-query, and “ $\rightarrow$ ” stands for “is rewritten to”.

- $OR(C, W) \rightarrow UNION(C, W)$ : OR substituted with UNION, W is pulled up in the top-level query.
- $AND(C, W) \rightarrow (C, W)$ : top-level AND replaced with its members, W is pulled up in the top-level query.
- $OR(C, AND(D, W)) \rightarrow UNION(C, AND(D, W))$ : OR substituted with UNION, W is pulled up in a top-level AND clause, that can be removed and replaced by its members.
- $AND(C, OR(D, W)) \rightarrow UNION(AND(C, D), AND(C, W))$ : this is a straightforward application of the theorem:  $C \wedge (D \vee W) \Leftrightarrow (C \wedge D) \vee (C \wedge W)$ . W is pulled up in a top-level AND clause, that can be removed and replaced by its members.

Rewriting rules W1 to W6 defined in Algorithm 5 generalize these examples. Rules W1 to W4 reflect exactly the example above. Since they may create UNION clauses nested beneath AND or OR clauses, additional rules W5 and W6 rewrite such queries to pull up UNION clauses in the top-level query. They can be illustrated by those two additional examples:

- $AND(C, UNION(D, W)) \rightarrow UNION(AND(C, D), AND(C, W))$ .
- $OR(C, UNION(D, W)) \rightarrow UNION(C, D, W)$ .

Note that the case of nested UNION clauses is dealt with by rule O1 in Algorithm 4.

**Algorithm 5: Pull-up of WHERE clauses to the top-level query**

The “ $\rightarrow$ ” arrow means “is rewritten as”.

**W1**  $\text{OR}(C_1, \dots, C_n, W) \rightarrow \text{UNION}(\text{OR}(C_1, \dots, C_n), W)$

**W2**  $\text{OR}(C_1, \dots, C_n, \text{AND}(D_1, \dots, D_m, W)) \rightarrow \text{UNION}(\text{OR}(C_1, \dots, C_n), \text{AND}(D_1, \dots, D_m, W))$

*Proof:*  $C_1 \vee \dots \vee C_n \vee (D_1 \wedge \dots \wedge D_m \wedge W) \Leftrightarrow (C_1 \vee \dots \vee C_n) \vee (D_1 \wedge \dots \wedge D_m \wedge W)$

Therefore,  $\text{eval}(C_1 \vee \dots \vee C_n \vee (D_1 \wedge \dots \wedge D_m \wedge W)) = \text{eval}(C_1 \vee \dots \vee C_n) \cup \text{eval}(D_1 \wedge \dots \wedge D_m \wedge W)$ .

**W3**  $\text{AND}(C_1, \dots, C_n, W) \rightarrow (C_1, \dots, C_n, W)$ , *iff the AND clause is a top-level query object or under a UNION clause.*

**W4**  $\text{AND}(C_1, \dots, C_n, \text{OR}(D_1, \dots, D_m, W)) \rightarrow \text{UNION}(\text{AND}(C_1, \dots, C_n, \text{OR}(D_1, \dots, D_m)), \text{AND}(C_1, \dots, C_n, W))$

*Proof:*  $C_1 \wedge \dots \wedge C_n \wedge (D_1 \vee \dots \vee D_m \vee W) \Leftrightarrow (C_1 \wedge \dots \wedge C_n) \wedge ((D_1 \vee \dots \vee D_m) \vee W)$

$\Leftrightarrow ((C_1 \wedge \dots \wedge C_n) \wedge (D_1 \vee \dots \vee D_m)) \vee ((C_1 \wedge \dots \wedge C_n) \wedge W)$

Therefore,  $\text{eval}(C_1 \wedge \dots \wedge C_n \wedge (D_1 \vee \dots \vee D_m \vee W)) = \text{eval}(C_1 \wedge \dots \wedge C_n \wedge (D_1 \vee \dots \vee D_m)) \cup \text{eval}(C_1 \wedge \dots \wedge C_n \wedge W)$

**W5**  $\text{AND}(C_1, \dots, C_n, \text{UNION}(D_1, \dots, D_m)) \rightarrow \text{UNION}(\text{AND}(C_1, \dots, C_n, D_1), \dots, \text{AND}(C_1, \dots, C_n, D_m))$

*Proof:*  $C_1 \wedge \dots \wedge C_n \wedge (D_1 \vee \dots \vee D_m) \Leftrightarrow (C_1 \wedge \dots \wedge C_n) \wedge (D_1 \vee \dots \vee D_m)$

$\Leftrightarrow (C_1 \wedge \dots \wedge C_n \wedge D_1) \vee \dots \vee (C_1 \wedge \dots \wedge C_n \wedge D_m)$

Therefore,  $\text{eval}(C_1 \wedge \dots \wedge C_n \wedge (D_1 \vee \dots \vee D_m)) = \text{eval}(C_1 \wedge \dots \wedge C_n \wedge D_1) \cup \dots \cup \text{eval}(C_1 \wedge \dots \wedge C_n \wedge D_m)$

**W6**  $\text{OR}(C_1, \dots, C_n, \text{UNION}(D_1, \dots, D_m)) \rightarrow \text{UNION}(\text{OR}(C_1, \dots, C_n), D_1, \dots, D_m)$

We illustrate rules W1 to W6 in a second dedicated example. We wish to translate the condition below, stating that the last member of either team “dev” or “test” has the name “john”:

```
trans ($.teams["dev", "test"][(@.length - 1)].name, equals("john"))
```

Function *trans* translates this condition into this abstract MongoDB query:

```
OR( AND(EXISTS(.teams.dev),
      WHERE('this.teams.dev[this.teams.dev.length - 1].name CONDJS(equals("john"))')),
    AND(EXISTS(.teams.test),
      WHERE('this.teams.test[this.teams.test.length - 1].name CONDJS(equals("john"))')))
```

Then we iteratively apply rules O1 to O6 and W1 to W6 as described in function *rewrite* (next section). First, rule W2 replaces the top-level OR with a UNION clause:

```
W2 UNION(
  OR(AND(EXISTS(.teams.dev),
        WHERE('this.teams.dev[this.teams.dev.length - 1].name CONDJS(equals("john"))')),
    AND(EXISTS(.teams.test),
        WHERE('this.teams.test[this.teams.test.length - 1].name CONDJS(equals("john"))')))
```

Then rule O4 replaces the OR of one term with the term itself:

```
O4 UNION(
  AND(EXISTS(.teams.dev),
    WHERE('this.teams.dev[this.teams.dev.length - 1].name CONDJS(equals("john"))')),
  AND(EXISTS(.teams.test),
    WHERE('this.teams.test[this.teams.test.length - 1].name CONDJS(equals("john"))')))
```

Rules W2 and O4 basically replaced the top-level OR with a UNION. Now the abstract query is a union of two top-level AND operators that can simply be removed by rule W3:

```
W3 UNION(
  (EXISTS(.teams.dev),
    WHERE('this.teams.dev[this.teams.dev.length - 1].name CONDJS(equals("john"))')),
  (EXISTS(.teams.test),
    WHERE('this.teams.test[this.teams.test.length - 1].name CONDJS(equals("john"))')))
```

Both queries can now be rewritten into executable concrete queries:

```
UNION( ( "teams.dev": {$exists: true},
        $where: 'this.teams.dev[this.teams.dev.length - 1].name CONDJS(equals("john"))'),
      ( "teams.test": {$exists: true},
        $where: 'this.teams.test[this.teams.test.length - 1].name CONDJS(equals("john"))')
    )
```

### 5.5.3 Function rewrite

Finally we define in Algorithm 6 the complete optimization and translation algorithm that iteratively uses rules O1 to O6 and W1 to W6 to perform all possible rewritings, and ultimately generate either one concrete MongoDB query or a union of concrete MongoDB queries.

---

#### Algorithm 6: Abstract MongoDB query optimization and translation into concrete MongoDB queries

---

**Function** rewrite(Q):

**do**

**do**

Q ← apply rules O1 to O5 that match any sub-query of Q

**until** no more rewriting can be performed

**do**

Q ← apply rules W1 to W6 that match any sub-query of Q

**until** no more rewriting can be performed

**until** no more rewriting can be performed by either rules O1 to O5 or W1 to W6

Q' ← translate Q as defined in Definition 13.

**return** Q'

---

A consequence of function rewrite is that we can always rewrite an abstract MongoDB query into a union of queries in which there is no more NOT\_SUPPORTED clause and any WHERE clause only appears as a top-level object or in a top-level AND clause. This is summarized in the Theorem 1:

**Theorem 1.** Let C be an equality or not-null condition on a JSONPath expression. Let  $Q = (Q_1, \dots, Q_n)$  be the abstract MongoDB query produced by  $trans(C)$ .

**Rewritability:** It is always possible to rewrite Q into a query  $Q' = \text{UNION}(Q'_1, \dots, Q'_m)$  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 certain answers, i.e. 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.

#### Proof of Theorem 1:

**Completeness.** The result on the completeness of results has been proven in the description of rule O5 when dealing with NOT\_SUPPORTED clauses.

**Rewritability, NOT\_SUPPORTED clauses.** By construction, function  $trans$  may generate a NOT\_SUPPORTED clause in the top-level query or in the following patterns: AND(...,N,...), ELEMATCH(...,N,...), OR(...,N,...), UNION(...,N,...), FIELD(...)...FIELD(...) N, where “N” stands for a NOT\_SUPPORTED clause. If it is in the top-level query, then Definition 13 rewrites it into the empty query that shall retrieve all documents of the collection. In the case of other patterns, when applying rewriting rule O5 we obtain:

```
AND(...,N,...) → AND(...)
ELEMATCH(...,N,...) → ELEMATCH(...)
OR(...,N,...) → N
UNION(...,N,...) → N
FIELD(...)...FIELD(...) N → N
```

The first two rewritings remove the NOT\_SUPPORTED clause, coming up with a valid query. The next three



rewritings raise the NOT\_SUPPORTED up to the parent clause. Since nested AND/OR/UNION clauses are merged by rule O1, this may lead to one of the patterns below; we precise the way they are rewritten:

$$\begin{aligned} \text{AND}(\dots, \text{OR}(\dots, N, \dots), \dots) &\rightarrow \text{AND}(\dots, N, \dots) \rightarrow \text{AND}(\dots) \\ \text{AND}(\dots, \text{UNION}(\dots, N, \dots), \dots) &\rightarrow \text{AND}(\dots, N, \dots) \rightarrow \text{AND}(\dots) \\ \text{AND}(\dots, \text{FIELD}(\dots) \dots \text{FIELD}(\dots) N, \dots) &\rightarrow \text{AND}(\dots, N, \dots) \rightarrow \text{AND}(\dots) \\ \text{ELEMATCH}(\dots, \text{OR}(\dots, N, \dots), \dots) &\rightarrow \text{ELEMATCH}(\dots, N, \dots) \rightarrow \text{ELEMATCH}(\dots) \\ \text{ELEMATCH}(\dots, \text{UNION}(\dots, N, \dots), \dots) &\rightarrow \text{ELEMATCH}(\dots, N, \dots) \rightarrow \text{ELEMATCH}(\dots) \\ \text{ELEMATCH}(\dots, \text{FIELD}(\dots) \dots \text{FIELD}(\dots) N, \dots) &\rightarrow \text{ELEMATCH}(\dots, N, \dots) \rightarrow \text{ELEMATCH}(\dots) \end{aligned}$$

The rewritings above show that, wherever the NOT\_SUPPORTED clause shows, it is iteratively removed by the rewritings using rules O1 to O5 and W1 to W6.

Hence the first part of the **rewritability** property: it is always possible to come up with a rewriting that does not contain any NOT\_SUPPORTED clause.

**Rewritability, WHERE clauses.** By construction, function *trans* may generate a WHERE clause in the top-level query or nested in AND or OR clauses, but a WHERE clause cannot be nested in an ELEMATCH clause. Furthermore, rules W1 to W6 may create UNION clauses, and Algorithm 4 flattens nested OR/AND/UNION clauses and merges sibling WHERE clauses. Consequently, a WHERE clause may be either in the top-level query (the query is thus executable) or in the following nine patterns:

$$\begin{aligned} &\text{OR}(\dots, W, \dots) \\ &\text{OR}(\dots, \text{AND}(\dots, W, \dots), \dots) \\ &\text{OR}(\dots, \text{UNION}(\dots, W, \dots), \dots) \\ &\text{AND}(\dots, W, \dots) \\ &\text{AND}(\dots, \text{OR}(\dots, W, \dots), \dots) \\ &\text{AND}(\dots, \text{UNION}(\dots, W, \dots), \dots) \\ &\text{UNION}(\dots, W, \dots) \\ &\text{UNION}(\dots, \text{AND}(\dots, W, \dots), \dots) \\ &\text{UNION}(\dots, \text{OR}(\dots, W, \dots), \dots) \end{aligned}$$

where “W” stands for a WHERE clause.

To prove Theorem 1, we need a measure of the depth of a WHERE clause within a query. We first define the *depth* function as follows:

$$\begin{aligned} \text{depth}(\text{UNION}) &= 0 \\ \text{depth}(\text{AND}) &= 1 \\ \text{depth}(\text{OR}) &= 1 \\ \text{depth}(C_1/\dots/C_n) &= \text{depth}(C_1) + \dots + \text{depth}(C_n) \end{aligned}$$

Intuitively, function *depth* measures the depth of a MongoDB query made of nested clauses AND, OR or UNION, and possibly containing WHERE clauses. AND and OR count for 1, but UNION counts for 0: indeed UNION is not a MongoDB operator, instead it is meant to be processed outside of the database. Notation " $C_1/\dots/C_n$ " represents a nested query in which clause  $C_1$  is parent of clause  $C_2$  which is parent of clause  $C_3$  etc. until clause  $C_n$ .

We define function  $\text{depth}_w(Q)$  as the depth of a clause WHERE within a query Q:

$$\begin{aligned} \text{depth}_w(C_1, \dots, C_n, W) &= 0 \quad (\text{case of a top-level query}) \\ \text{depth}_w(C_1(\dots, C_2(\dots, C_n(\dots, W)))) &= \text{depth}(C_1/C_2/\dots/C_n) \end{aligned}$$

Below we explore how rules W1 to W6 rewrite the nine patterns we listed above. For each one, we give the depth of the WHERE clause in the pattern and in the rewritten query.

<b>OR</b> (...,W,...)	<i>Rule W1:</i> Q: <b>OR</b> ( $C_1, \dots, C_n, W$ ) $\rightarrow$ Q': <b>UNION</b> ( <b>OR</b> ( $C_1, \dots, C_n$ ), W) depth <sub>w</sub> (Q) = 1 depth <sub>w</sub> (Q') = 0
<b>OR</b> (..., <b>AND</b> (...,W,...),...)	<i>Rule W2:</i> Q: <b>OR</b> ( $C_1, \dots, C_n, \text{AND}(D_1, \dots, D_m, W)$ ) $\rightarrow$ Q': <b>UNION</b> ( <b>OR</b> ( $C_1, \dots, C_n$ ), <b>AND</b> ( $D_1, \dots, D_m, W$ )) depth <sub>w</sub> (Q) = 2 depth <sub>w</sub> (Q') = 1
<b>AND</b> (...,W,...)	<i>Rule W3:</i> Q: <b>AND</b> ( $C_1, \dots, C_n, W$ ) $\rightarrow$ Q': ( $C_1, \dots, C_n, W$ ) (W3 applies iff the AND clause is a top-level query object or under a UNION clause)

	$\text{depth}_w(Q) = 1$ $\text{depth}_w(Q') = 0$
<b>AND</b> (..., <b>OR</b> (...,W,...),...)	<i>Rule W4:</i> $Q: \text{AND}(C_1, \dots, C_n, \text{OR}(D_1, \dots, D_m, W)) \rightarrow$ $Q': \text{UNION}(\text{AND}(C_1, \dots, C_n, \text{OR}(c)), \text{AND}(C_1, \dots, C_n, W))$ $\text{depth}_w(Q) = 2$ $\text{depth}_w(Q') = 1$
<b>AND</b> (..., <b>UNION</b> (...,W,...),...)	<i>We first apply rule W5, then rule W3:</i> $Q: \text{AND}(C_1, \dots, C_n, \text{UNION}(D_1, \dots, D_m, W)) \rightarrow$ $\text{UNION}(\text{AND}(C_1, \dots, C_n, D_1), \dots, \text{AND}(C_1, \dots, C_n, D_m), \text{AND}(C_1, \dots, C_n, W)) \rightarrow$ $Q': \text{UNION}((C_1, \dots, C_n, D_1), \dots, (C_1, \dots, C_n, D_m), (C_1, \dots, C_n, W))$ $\text{depth}_w(Q) = 1$ $\text{depth}_w(Q') = 0$
<b>OR</b> (..., <b>UNION</b> (...,W,...),...)	<i>Rule W6:</i> $Q: \text{OR}(C_1, \dots, C_n, \text{UNION}(D_1, \dots, D_m, W)) \rightarrow Q': \text{UNION}(\text{OR}(C_1, \dots, C_n), D_1, \dots, D_m, W)$ $\text{depth}_w(Q) = 1$ $\text{depth}_w(Q') = 0$
<b>UNION</b> (...,W,...)	The WHERE clause is a top-level query, the query is valid as is and no rewriting is needed.
<b>UNION</b> (..., <b>AND</b> (...,W,...),...)	<i>Rule W3:</i> $Q: \text{UNION}(C_1, \dots, C_n, \text{AND}(D_1, \dots, D_m, W)) \rightarrow Q': \text{UNION}(C_1, \dots, C_n, (D_1, \dots, D_m, W))$ <i>(W3 applies iff the AND clause is a top-level query object or under a UNION clause)</i> $\text{depth}_w(Q) = 1$ $\text{depth}_w(Q') = 0$
<b>UNION</b> (..., <b>OR</b> (...,W,...),...)	<i>We first apply rule W1 then rule O1 to merge nested UNIONS:</i> $Q: \text{UNION}(C_1, \dots, C_n, \text{OR}(D_1, \dots, D_m, W)) \rightarrow$ $\text{UNION}(C_1, \dots, C_n, \text{UNION}(\text{OR}(D_1, \dots, D_m), W)) \rightarrow$ $Q': \text{UNION}(C_1, \dots, C_n, \text{OR}(D_1, \dots, D_m), W)$ $\text{depth}_w(Q) = 1$ $\text{depth}_w(Q') = 0$

In all patterns listed above, we have shown that the depth of the WHERE is always decreased by one using rules W1 to W6 and optionally rule O1. By applying this process iteratively it is easy to see that we ultimately come up with a rewriting that contains WHERE clauses only in the top-level query.

Hence the second part of the **rewritability** property.

## 6 Overall query translation and evaluation process

Let us sum up the translation process. Function  $\text{trans}_m$  (section 3) translates a SPARQL query into an abstract query, helped by function  $\text{transTP}_m$  that translates a triple pattern tp into a union of per-triples-map abstract queries {FromPart, ProjectPart, WherePart}, under a set of triples maps bound to tp. The *WherePart* consists of *isNotNull*, *equals* or *join* conditions. Join conditions are not supported by MongoDB, thus they shall be executed by the query processing engine. Functions  $\text{trans}$  (section 5.4) and  $\text{rewrite}$  (section 5.5) translate each *isNotNull* and *equals* condition on a JSONPath expression into a concrete MongoDB query or a union of concrete MongoDB queries. In the latter, the UNION clause applies to sets of JSON documents retrieved from the database, whereas other operators mentioned above apply to triples generated from the results of queries.

The rewritten concrete queries have several limitations though:

- (i) The ambiguous semantics of the MongoDB query language (underlined in section 5.1) entails that a MongoDB query cannot be guaranteed to have the same semantics as the triple pattern it stands for. Consequently, all documents matching the SPARQL query are returned (the certain answers), but in addition, non-matching documents may be returned.

- (ii) Some JSONPath elements are not supported in the rewriting process as they have no equivalent in MongoDB (restrictions listed in section 5.2). Nevertheless, the rewriting process ensures that all matching documents are returned, but again, non-matching documents may be returned too (Algorithm 5).
- (iii) In a MongoDB query, a projection clause can concern document fields but it cannot concern elements of an array. Therefore, it cannot be guaranteed that only needed fields be projected.
- (iv) Lastly, at this stage, our method does not tackle SPARQL filters.

To work around those issues, the overall query processing works in several steps detailed in Algorithm 7:

- For each {FromPart, ProjectPart, WherePart} query, the query translation engine creates concrete MongoDB queries (lines 3-10). It executes the concrete queries against the database, and from the result JSON documents it computes the UNION clauses that may be produced by the *rewrite* function (lines 11-15). It computes the *join* conditions from the *WhereParts*, if any (lines 16-18).
- It produces RDF triples by applying the triples map to the result JSON documents (line 21). This entails the evaluation of the JSONPath expressions (the term maps references) against the documents. This clears hurdles (ii) and (iii). The result triples are materialized and stored in a primary result graph (one graph per {FromPart, ProjectPart, WherePart} query).
- When all {FromPart, ProjectPart, WherePart} queries are materialized in primary graphs, the query processing engine computes the INNER JOIN, LEFT OUTER JOIN and UNION operators (line 23). The result is stored in a secondary result graph.
- Finally, the SPARQL query is evaluated against the secondary result graph (line 25). This rules out all non-matching triples that were generated due to issues (i) and (iv).

In Algorithm 7, the distinction between the child and parent queries is not made explicit for the sake of clarity, but the running example in section 9 goes through it.

**Algorithm 7: Overall SPARQL query processing**


---

```

1  Function process(sparqlQuery):
2  abstractQuery ← transm(sparqlQuery)
3  for each {FromPart, ProjectPart, WherePart} ∈ abstractQuery do
4    Q ← true
5    for each cond ∈ WherePart | cond is a isNotNull or equals condition do
6      <JSONPath>, <condition> ← cond
7      Q ← AND(Q, trans(<JSONPath>, <condition>))
8    end if
9  end for
10 Q' ← rewrite(Q) // Q' is either a concrete query or a union of concrete queries
11 if Q' is a valid MongoDB query
12   R ← execute(Q')
13 else // Q' is UNION(q1, ..., qn)
14   R ← execute(q1) ∪ ... ∪ execute(qn)
15 end if
16 for each cond ∈ WherePart | cond is a join condition do
17   R ← compute the join condition cond on JSON documents of R
18 end for
19 // Generate the triples corresponding to documents of R
20 primaryGraph ← Apply the triples map corresponding to {FromPart, ProjectPart, WherePart}
21 end for
22 // Compute UNION, INNER JOIN and LEFT OUTER JOIN operations of abstractQuery on primary graphs:
23 secondaryGraph ← evaluate abstractQuery on all primaryGraph's
24 // Late SPARQL query evaluation iff Q' contains at least one NOT_SUPPORTED clause
25 resultGraph ← evaluate sparqlQuery on secondaryGraph
26 return resultGraph

```

---

## 7 Conclusion, Discussion and perspectives

### 7.1 Conclusion

In this document we proposed a method to access arbitrary MongoDB JSON documents with SPARQL using custom mappings described in the xR2RML mapping language. We first defined a method that rewrites a SPARQL query into an abstract query independent of the target database, relying on bindings between a SPARQL triple pattern and xR2RML mappings. A set of rules translate the abstract query into an abstract representation of a MongoDB query, and we showed that the latter can always be rewritten into a union of valid concrete MongoDB queries that shall return all the matching documents. Finally we defined an algorithm that orchestrates the different steps until the evaluation of MongoDB queries and the generation of the RDF triples matching the SPARQL query.

The abstract query generated from the SPARQL query suffers from some flaws that most other rewriting methods: it is likely to produce unneeded complexity such as self-joins and self-unions. Therefore, techniques designed in the SPARQL-to-SQL context should be applied when relevant such as the elimination of self-joins and self unions, and the projection pushing [7,13,18]. Notice however that filter pushing is not necessary in our case since, by construction, filters are pushed down in the queries as much as possible.

Unlike SQL or XQuery whose expressiveness is similar to that of SPARQL, the expressiveness of the MongoDB query language is far more limited: joins are not supported and filters are supported with strong restrictions (e.g. no comparison between fields of a document, \$where operator restricted to the top-level query). This discrepancy entails that a SPARQL query shall be translated into possibly several independent queries, thus delegating several steps to the query processing engine. In turn, the query processing engine may need to embark query plan optimization logics in order to perform efficient query evaluation.

Despite a comprehensive documentation, there is no formal description of the semantics of the MongoDB query language, and more importantly, ambiguities are voluntarily part of the language. Let us add that the JSONPath language used in the mappings to extract data from JSON documents is unclear and subject to divergent interpretations. Lastly, some JSONPath expressions cannot be translated into equivalent MongoDB queries. Consequently, the query translation method cannot ensure that query semantics is preserved. Nevertheless, we proved that rewritten queries retrieve all matching documents, in addition to possibly non matching ones. We overcome this issue by evaluating the SPARQL query against the triples generated from the database results. This guarantees semantics preservation, at the cost of an additional SPARQL evaluation.

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 have encountered with MongoDB shall occur with other NoSQL databases.

## 7.2 Query optimization

Function  $trans_m$  translates a SPARQL query into an abstract query containing INNER JOIN, LEFT OUTER JOIN and UNION operators. If the target database is an RDB, then the abstract query can be translated into a single SQL query, as shown in various approaches [4,15,7,17,13,12]. Conversely, such operators are not supported by the MongoDB database, leaving several steps to the query processing engine. This is illustrated in Algorithm 7: lines 13-15 process unions of JSON documents; lines 16-18 process the join conditions declared in triples maps (join between JSON documents); line 23 processes other INNER JOIN, LEFT OUTER JOIN and UNION operators between graphs.

The fact that concrete queries must be evaluated independently of each other can be the cause of performance issues. In our running example, the translation of tp1 (section 9.2) generates two concrete queries; among them, Q2' is very little selective and actually returns all documents in the *departments* collection, although this collection is already queried with more selective criteria in the processing of tp2. Overall, the problem of efficiently evaluating the abstract query generated by function  $trans_m$  amounts to a classical query plan optimization problem. Future works shall include the study of methods such as the bind join [9] to inject intermediary results into a subsequent query. The join re-ordering based on the number of results that queries shall retrieve could also be used, very similarly to the methods applied in distributed SPARQL query engines [14,8].

## 7.3 Support of the SPARQL query language

As we have mentioned in section 3, our method does not tackle SPARQL filters. We plan to manage them by considering additional types of condition. Nevertheless, this support is limited by the capabilities of the underlying database. For instance, SQL supports most of the SPARQL operators such as logics, comparison, arithmetic and unary operators. This is far from being the case in MongoDB. As illustrated in section 7.4, JavaScript functions can help in this matter, although we have to consider this option with reluctance due to the performance issues it entails.

The issue is even more striking if we consider the SPARQL 1.1<sup>13</sup> capabilities such as property paths, aggregates (having, group by), assignments (value, bind), negation (not exists, minus) and functions on strings. Such features

<sup>13</sup> <http://www.w3.org/TR/sparql11-query/>

shall (probably) not be translated into MongoDB queries, and we shall not escape the late evaluation of the SPARQL query against the triples generated at an earlier step, as we propose in Algorithm 7.

## 7.4 Dealing with the \$where operator

In the MongoDB query language, the \$where operator is valid only in the top-level query document. Using rules W1 to W6 we show that we can pull up a \$where operator nested beneath AND or OR operators, but we cannot deal with a \$where operator nested beneath an \$elemMatch. By construction, rules in function *trans* (Algorithm 2) exclude the latter case by generating a NOT\_SUPPORTED operator. In other words, *trans* drops the \$where and postpones the evaluation of the condition to a later step: the effect is to widen the query that shall retrieve more documents than those matching the initial SPARQL query. Then, Algorithm 7 runs a late evaluation of the SPARQL query against the set of generated triples to make sure we produce only the expected triples.

An alternative is to push whatever needs to be in the \$where operator by means of a JavaScript function. Let us consider the following example: a MongoDB instance stores JSON documents about bank account details, such as:

```
{accounts: [
  {current: { credits: 100, debits: 50}},
  {savings: { credits: 80, debits: 80}}
]}
```

We want to retrieve documents where credits equal debits in at least one account. The MongoDB \$eq operator does not allow to specify the equality between two fields, therefore we must use the \$where operator. We cannot write the following query: {"accounts": {\$elemMatch: {\$where: {"credits == debits"}}}} since the \$where operator must be in the top-level query document. But we can write a JavaScript function that browses the "accounts" array to check if the condition is true for at least one element in the array:

```
$where: {function() { \
  result = false; \
  for (i = 0; i < this.accounts.length; i++) \
    result = result || ( this.accounts[i].credits == this.accounts[i].debits); \
  return result }}
```

This option has the advantage of returning only the matching documents, but it has two shortcomings. (i) It may cause a serious performance penalty in the database: as we already mentioned, MongoDB cannot take advantage of indexes when executing JavaScript code, thus it shall retrieve all documents matching all conditions except the \$where, then apply the JavaScript function to all of them. (ii) It can lead to the generation of complex JavaScript functions when it comes to translate rich JSONPath expressions. Conversely, in the method we have chosen, the database query shall be faster but the price is a larger amount of data retrieved and an additional SPARQL query evaluation to rule out non-matching triples. It is unclear, at this stage, whether one solution should be preferred to the other. But most likely, we can assume that the choice shall depend on the context.

## 8 Appendix A

In this appendix we provide the detailed algorithm of functions used in the  $transTP_m$  function, defined in section 3.

### 8.1 Function genProjection

We first describe function *getReferences*, a utility function used in functions *genProjection* and *genCond*.

---

**Algorithm 8: Function getReferences returns the references associated with an xR2RML term map**

---

```

Function getReferences(termMap):
  case type(termMap)
    template-valued : termVal ← getTemplateReferences(termMap.template)
    reference-valued : termVal ← termMap.reference
    constant-valued : termVal ← "
  end case
  return termVal

```

---



---

**Algorithm 9: Generates the list of xR2RML references that must be projected in the MongoDB query**

---

**Input:** tp is a triple pattern, TM is an xR2RML triples map bound to tp.

```

Function genProjection(tp, TM):
  refList ← <empty list>
  if type(tp.sub) is VARIABLE then
    refList ← refList | child/getReferences(TM.subjectMap)
  end if
  if type(tp.pred) is VARIABLE then
    refList ← refList | child/getReferences(TM.predicateObjectMap.predicateMap)
  end if
  OM ← TM.predicateObjectMap.objectMap
  if OM is a ReferencingObjectMap then
    // Since MongoDB cannot compute joins, the xR2RML processor has to do it, thus
    // the joined fields must always be projected, whether tp.obj is an IRI or a variable:
    refList ← refList | child/getReferences(OM.join.child) | parent/getReferences(OM.join.parent)
    // In addition, if tp.obj is a variable, the subject of the parent TM must be projected too
    if type(tp.obj) is VARIABLE then
      refList ← refList | parent/getReferences(OM.parentTriplesMap.subjectMap)
    end if
  else if type(tp.obj) is VARIABLE then
    refList ← refList | getReferences(OM)
  end if
  return refList

```

---

### 8.2 Function genCond

We first describe function *getValue* that is used in function *genCond*.

**Algorithm 10: Function `getValue` returns the value of the RDF term depending on the xR2RML term map where it is applied.**

This is simply a utility function that applies the inverse expression in case of a template-valued term map, and returns the RDF term as is otherwise.

---

```

Function getValue(rdfTerm, termMap):
  case type(termMap)
    template-valued : termVal ← inverseExpression(rdfTerm, termMap.inverseExpression)
    reference-valued : termVal ← rdfTerm
    constant-valued : termVal ← rdfTerm
  end case
  return termVal

```

---

**Algorithm 11: Generate the set of conditions that match a triple pattern**

**Input:** tp is a triple pattern, TM is an xR2RML triples map bound to tp, f is a SPARQL filter.

```

Function genCond(tp, TM, f):
  cond ← <empty list>
  // Subject part
  if type(TM.subject) is reference-valued or template-valued then
    case type(tp.sub)
      IRI: cond ← cond | equals(getValue(tp.sub, TM.subjectMap), getReferences(TM.subjectMap))
      VARIABLE:
        if f contains a condition mentioning tp.sub then
          cond ← cond | sparqlFilter(getReferences(TM.subjectMap), f)
        else
          cond ← cond | isNotNull(getReferences(TM.subjectMap))
        end if
    end case
  end if

  // Predicate part
  PM ← TM.predicateObjectMap.predicateMap
  if type(PM) is reference-valued or template-valued then
    case type(tp.pred)
      IRI : cond ← cond | equals(getValue(tp.pred, PM), getReferences(PM))
      VARIABLE :
        if f contains a condition mentioning tp.pred then
          cond ← cond | sparqlFilter(getReferences(PM), f)
        else
          cond ← cond | isNotNull(getReferences(PM))
        end if
    end case
  end if

  // Object part
  OM ← TM.predicateObjectMap.objectMap
  case type(tp.obj)
    LITERAL:
      if type(OM) is reference-valued or template-valued then
        cond ← cond | equals(getValue(tp.obj, OM), getReferences(OM))
      end if
    IRI:
      if OM is a ReferencingObjectMap then
        // tp.obj is a constant IRI to be matched with the subject of the parent TM:
        // add an equality condition for each reference in the subject map of the parent TM

```

---



---

```

if type(OM.parentTriplesMap.subjectMap) is reference-valued or template-valued then
  obj_value ← getValue(tp.obj, OM.parentTriplesMap.subjectMap)
  cond ← cond | equals(obj_value, getReferences(OM.parentTriplesMap.subjectMap))
end if

  // Add the join conditions
  cond ← cond | join(child/OM.joinCondition.child, parent/OM.joinCondition.parent)
else
  // tp.obj is a constant IRI and there is no parent TM
  if type(OM) is reference-valued or template-valued then
    cond ← cond | equals(getValue(tp.obj, OM), getReferences(OM))
  end if
end if

VARIABLE:
if OM is a ReferencingObjectMap then
  // tp.obj is a SPARQL variable to be matched with the subject of the parent TM
  if type(OM.parentTriplesMap.subjectMap) is reference-valued or template-valued then
    if f contains a condition mentioning tp.obj then
      cond ← cond | sparqlFilter(getReferences(OM.parentTriplesMap.subjectMap), f)
    else
      cond ← cond | isNotNull(getReferences(OM.parentTriplesMap.subjectMap))
    end if
  end if

  // Add the join conditions
  cond ← cond | join(child/OM.join.child, parent/OM.join.parent)
else
  // There is no parent TM
  if type(OM) is reference-valued or template-valued then
    if f contains a condition mentioning tp.obj then
      cond ← cond | sparqlFilter(getReferences(OM), f)
    else
      cond ← cond | isNotNull(getReferences(OM))
    end if
  end if
end if
end case

```

---

## 9 Appendix B: Complete Running Example

In this example we assume we have set up a MongoDB database with two collections “staff” and “departments” given in Listing 1 and Listing 2 respectively. Collection “departments” lists the departments within a company, including a department code and its members. Members are given by their name and age. Collection “staff” lists people by their name (that may be either field “familyname” or “lastname”), and provides a list of departments that they manage, if any, in array field “manages”.

---

### Listing 1: Collection “staff”

---

```
{ "familyname": "Underwood", "manages": ["Sales"] },
{ "lastname": "Dunbar", "manages": ["R&D", "Human Resources"] },
{ "lastname": "Sharp", "manages": ["Support", "Business Dev"] }
```

---



---

### Listing 2: Collection “departments”

---

```
{ "dept": "Sales", "code": "sa",
  "members": [{"name": "P. Russo", "age": 28}, {"name": "J. Mendez", "age": 43}]
},
{ "dept": "R&D", "code": "rd",
  "members": [{"name": "J. Smith", "age": 32}, {"name": "D. Duke", "age": 23}]
},
{ "dept": "Human Resources", "code": "hr",
  "members": [{"name": "R. Posner", "age": 46}, {"name": "D. Stamper", "age": 38}]
},
{ "dept": "Business Dev", "code": "bdev",
  "members": [{"name": "R. Danton", "age": 36}, {"name": "E. Meetchum", "age": 34}]
}
```

---

The xR2RML mapping graph in Listing 3 consists of two triples maps <#Staff> and <#Departments>. Triples map <#Staff> has a referencing object map whose parent triples map is <#Departments>. Triples map <#Departments> generates triples with predicate `ex:hasSeniorMember` for each member of the department who is 40 years old or more. For the sake of simplicity the queries in both triples maps retrieve all documents of the collection with no other query filter.

We wish to translate the SPARQL query below, that aims at retrieving senior members of departments whose manager is “Dunbar”. The query consists of one basic graph pattern *bgp*, itself consisting of two triple patterns *tp1* and *tp2*:

```
SELECT ?senior WHERE {
  <http://example.org/staff/Dunbar> ex:manages ?dept. // tp1
  ?dept ex:hasSeniorMember ?senior. // tp2
}
```

We execute the SPARQL query processing function (Algorithm 7). First, the  $trans_m$  function translates the SPARQL query into an abstract query (Algorithm 7, line 2). The execution of the  $trans_m$  function (Definition 1) follows:

```
transm(bgp, true)
= transm(tp1, true) INNER JOIN transm(tp2, true) ON var(tp1) ∩ var(tp2)
= transTPm(tp1, true) INNER JOIN transTPm(tp2, true) ON {?dept}
```

Function  $bind_m$  (Definition 5) infers two triple pattern bindings:

```
bindm(bgp) = { (t1, {<#Staff>}) , (tp2, {<#Departments>}) }
```

In the subsequent sections we describe the execution of the *transTP<sub>m</sub>* function for each triple pattern, starting with *tp<sub>2</sub>*; then we describe the final computation of the INNER JOIN operator.

---

### Listing 3: xR2RML Example Mapping Graph

---

```
<#Departments>
  xrr:logicalSource [ xrr:query "db.departments.find({})" ];
  rr:subjectMap [ rr:template "http://example.org/dept/{$.code}" ];
  rr:predicateObjectMap [
    rr:predicate ex:hasSeniorMember;
    rr:objectMap [ xrr:reference "$.members[?(@.age >= 40)].name"; ]
  ].

<#Staff>
  xrr:logicalSource [ xrr:query "db.staff.find({})"; ];
  rr:subjectMap [ rr:template "http://example.org/staff/{['lastname','familyname']}" ];
  rr:predicateObjectMap [
    rr:predicate ex:manages;
    rr:objectMap [
      rr:parentTriplesMap <#Departments>;
      rr:joinCondition [
        rr:child "$.manages.*";
        rr:parent "$.dept";
      ];
    ];
  ].
```

---

## 9.1 Translation of *tp<sub>2</sub>*

Triple pattern *tp<sub>2</sub>*: ?dept ex:hasSeniorMember ?senior.

`getBoundTriplesMapsm(tp2)` returns triples map <#Departments>.

```
transTPm(tp2, true) =
  FromPart ← {child: [xrr:query "db.departments.find({})"]}
  ProjectPart ← genProjection(tp2, <#Departments>)
  WherePart ← genCond(tp2, <#Departments>, true)
```

The *FromPart* contains only one query, the child query, thus below we do no longer mention it. Let us detail the calculation of *ProjectPart* (Algorithm 9) and *WherePart* (Algorithm 11):

#### ProjectPart:

```
genProjection(tp2, <#Departments>) = ($.code, $.members[?(@.age >= 40)].name)
```

In a MongoDB query, a projection clause can concern document fields but it cannot concern elements of an array. Thus, we cannot project field “name” of elements of array “members”. Consequently, when translated to the MongoDB query language, the *ProjectPart* only projects fields “code” and “members”: {"code": 1, "members": 1}.

```
WherePart ← genCond(tp2, <#Departments>, true):
```

- The subject of *tp<sub>2</sub>* is a variable, this entails a non-null condition on the references of the subject map of <#Departments>:

```
isNotNull(getReferences(<#Departments>.subjectMap))
```

that we can rewrite:

```
isNotNull($.code)
```

- The predicate of *tp2* is constant, hence no condition is entailed.
- The object of *tp2* is again a variable, this entails a second non-null condition:

```
isNotNull($.members[?(@.age >= 40)].name))
```

Finally,  $\text{transTP}_m(\text{tp2}, \text{true}) =$

```
FromPart ← {child: [xrr:query "db.departments.find({})"]}
ProjectPart ← {code:1, members:1}
```

```
WherePart ← isNotNull($.code), isNotNull($.members[?(@.age >= 40)].name)
```

Each condition of the *WherePart* is translated into an abstract MongoDB query (Algorithm 7, lines 6-10). Below we detail the execution of the *trans* function (section 5.4) by indicating the rules matched at each step:

```
Q1 ← trans($.code, isNotNull) =
R0   trans(.code, isNotNull) =
R8,R1 FIELD(code) COND(isNotNull)
```

```
Q2 ← trans($.members[?(@.age >= 40)].name, isNotNull) =
R0   trans(.members[?(@.age >= 40)].name, isNotNull) =
R8   FIELD(members) trans([?(@.age >= 40)].name, isNotNull) =
R4   FIELD(members) ELEMATCH(trans(.name, isNotNull), transJS(?(@.age >= 40))) =
R8,R1 FIELD(members) ELEMATCH(FIELD(name) COND(isNotNull), transJS(@.age >= 40)) =
J6   FIELD(members) ELEMATCH(FIELD(name) COND(isNotNull), COMPARE(age, $gte, 40))
```

*Q1* and *Q2* are translated into either a concrete query or a union of concrete queries (Algorithm 7, line 11):

```
Q' ← rewrite(AND(Q1,Q2)) =
( "code": {$exists:true, $ne:null},
  "members": {$elemMatch: {"name": {$exists:true, $ne:null}, "age": {$gte:40}}} )
```

*Q'* is inserted in the MongoDB find request along with the *ProjectPart*:

```
db.departments.find(
  {"code": {$exists:true, $ne:null},
   "members": {$elemMatch: {"name": {$exists:true, $ne:null}, "age": {$gte:40}}},
  {"code":1, "members":1} ) // project part
```

The request is executed and returns 3 documents (Algorithm 7, lines 12-16):

```
R ← {"code":"sa", "members":[{"name":"P. Russo", "age":28}, {"name":"J. Mendez", "age":43}],
     {"code":"hr", "members": [{"name":"R. Posner", "age":46}, {"name":"D. Stamper", "age":38}]}
```

The triples map is applied to the documents and generated the following triples (Algorithm 7, line 21):

```
<http://example.org/dept/sa> ex:hasSeniorMember "J. Mendez".
<http://example.org/dept/hr> ex:hasSeniorMember "R. Posner".
```

## 9.2 Translation of *tp1*

Triple pattern *tp1*: `<http://example.org/staff/Dunbar> ex:manages ?dept.`

$\text{getBoundTriplesMaps}_m(\text{tp1})$  returns triples map `<#Staff>`.

$\text{transTP}_m(\text{tp1}, \text{true}):$

```
FromPart ← { child: [xrr:query "db.staff.find({})"],
```

```

    parent: [xrr:query "db.departments.find({})" ] }
ProjectPart ← genProjection(tp1, <#Staff>)
WherePart   ← genCond(tp1, <#Staff>, true)

```

**ProjectPart** (Algorithm 9):

As the subject of *tp1* is a constant, the reference in the subject map of triples map <#Staff> is not projected. Conversely the references in the join condition must be projected. The object of *tp1* is a variable, thus the reference of the corresponding term map must be projected too: this is the subject map of triples map <#Departments>:

```
genProjection(tp1, <#Staff>) = (child/$.manages.*, parent/$.dept, parent/$.code)
```

When translated to the MongoDB query language, the *ProjectPart* consists of:

```

Child query: {"manages": 1}
Parent query: {"dept": 1, "code": 1}

```

**WherePart** (Algorithm 11):

```
WherePart ← genCond(tp1, <#Staff>, true):
```

- The subject of *tp1* is an IRI, this entails an equality condition on the references of the subject map:
 

```
equals(getValue(tp1.sub, <#Staff>.subjectMap), getReferences(<#Staff>.subjectMap))
```

 that we can rewrite:
 

```
equals("Dunbar", child/$['lastname','familyname'])
```
- The predicate of *tp1* is constant, hence no condition is entailed.
- The object of *tp1* is a variable, this entails a non-null condition on the corresponding term map, that is, the subject map of triples map <#Departments>:
 

```
isNotNull(parent/$.code)
```

 In addition we must add a condition for the join clause:
 

```
join(child/$.manages.*, parent/$.dept)
```

Finally,  $\text{transTP}_m(\text{tp1}, \{\text{<#Staff>\}, \text{true}) =$

```

FromPart   ← { child: [xrr:query "db.staff.find({})"],
                parent: [xrr:query "db.departments.find({})"] }
ProjectPart ← Child query: {manages:1}
              Parent query: {dept:1, code:1}
WherePart   ← equals("Dunbar", child/$['lastname','familyname']),
              isNotNull(parent/$.code),
              join(child/$.manages.*, parent/$.dept)

```

Each condition of the *WherePart* is translated into an abstract MongoDB query (Algorithm 7, lines 6-10). Below we detail the execution of the *trans* function (section 5.4) by indicating the rules matched at each step:

```

Q1 ← trans($['lastname','familyname'], equals("Dunbar")) =
R0   trans(['lastname','familyname'], equals("Dunbar")) =
R3   OR(trans(.lastname, equals("Dunbar")), trans(.familyname, equals("Dunbar"))) =
R8,R1 OR(FIELD(lastname) COND(equals("Dunbar")), FIELD(familyname) COND(equals("Dunbar")))

Q2 ← trans($.code, isNotNull) =
R0   trans(.code, isNotNull) =
R8,R1 FIELD(code) COND(isNotNull)

```

*Q1* applies to the child query, *Q2* to the parent query. They are translated into concrete queries (Algorithm 7, line 11):

```

Q1' ← rewrite(Q1) =
     $or: [{"lastname": {$eq: "Dunbar"}}, {"familyname": {$eq: "Dunbar"}}]
Q2' ← rewrite(Q2) =
     "code": {$exists:true, $ne:null}

```

Q1' is inserted in the MongoDB find request along with the ProjectPart, for the child query:

```
db.departments.find(
  { $or: [{"lastname": { $eq: "Dunbar"}}, {"familyname": { $eq: "Dunbar"}}]},
  {"manages": 1} )
```

that returns one document (Algorithm 7, lines 12-16):

```
Rchild ← {"manages":["R&D", "Human Resources"]}
```

Q2' is inserted in the MongoDB find request along with the ProjectPart, for the parent query:

```
db.departments.find(
  {"code": { $exists:true, $ne:null}},
  {"code": 1, "dept": 1} )
```

that returns all documents of the collection:

```
Rparent ← {"dept":"Sales", "code":"sa"},
          {"dept":"R&D", "code":"rd"},
          {"dept":"Human Resources", "code":"hr"},
          {"dept":"Business Dev", "code":"bdev"}
```

Then the query processing engine computes the join condition (Algorithm 7, lines 17-20):

```
join(child/$.manages.*, parent/$.dept)
Rchild ← {"manages":["R&D", "Human Resources"]}
Rparent ← {"dept":"R&D", "code":"rd"}, {"dept":"Human Resources", "code":"hr"}
```

The triples maps are applied to the documents of  $R_{child}$  and  $R_{parent}$  (Algorithm 7, line 21), with the subject map of  $\langle \#Staff \rangle$  being constrained by subject part of the  $tp1$ . This generates the following triples:

```
<http://example.org/staff/Dunbar> ex:manages <http://example.org/dept/hr>.
<http://example.org/staff/Dunbar> ex:manages <http://example.org/dept/rd>.
```

### 9.3 Complete $trans_m$ processing

Recall the abstract query created by the  $trans_m$  function:

```
transTPm(tp1, true) INNER JOIN transTP(tp2, true) ON ?dept
```

Its evaluation, performed by the query processing engine (Algorithm 7, line 24), amounts to joins two graphs on the department IRIs. Informally we can write:

```
{ <http://example.org/staff/Dunbar> ex:manages <http://example.org/dept/hr>.
  <http://example.org/staff/Dunbar> ex:manages <http://example.org/dept/rd>. }
INNER JOIN
{ <http://example.org/dept/sa> ex:hasSeniorMember "J. Mendez".
  <http://example.org/dept/hr> ex:hasSeniorMember "R. Posner". }
ON ?dept
```

This entails the triples that match the graph pattern in the SPARQL query:

```
<http://example.org/staff/Dunbar> ex:manages <http://example.org/dept/hr>.
<http://example.org/dept/hr> ex:hasSeniorMember "R. Posner".
```

In this simple example, it is easy to notice that the final evaluation of the SPARQL query (Algorithm 7, line 26) will not rule out any result. The answer to the SELECT clause is the "R. Posner".

## 10 References

- [1] N. Bikakis, C. Tsinarakis, I. Stavrakantonakis, N. Gioldasis, S. Christodoulakis, The SPARQL2XQuery interoperability framework: Utilizing Schema Mapping, Schema Transformation and Query Translation to Integrate XML and the Semantic Web, *World Wide Web*. 18 (2015) 403–490.
- [2] S. Bischof, S. Decker, T. Krennwallner, N. Lopes, A. Polleres, Mapping between RDF and XML with XSPARQL, *J. Data Semant.* 1 (2012) 147–185.
- [3] C. Bizer, R. Cyganiak, D2R server - Publishing Relational Databases on the Semantic Web, in: *Proceeding 5th Int. Semantic Web Conf. ISWC 2006*, 2006.
- [4] A. Chebotko, S. Lu, F. Fotouhi, Semantics preserving SPARQL-to-SQL translation, *Data Knowl. Eng.* 68 (2009) 973–1000.
- [5] S. Das, S. Sundara, R. Cyganiak, R2RML: RDB to RDF Mapping Language, (2012).
- [6] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, R. Van de Walle, RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, in: *Proc. 7th Workshop Linked Data Web LDOW2014*, Seoul, Korea, 2014.
- [7] B. Elliott, E. Cheng, C. Thomas-Ogbuji, Z.M. Ozsoyoglu, A complete translation from SPARQL into efficient SQL, in: *Proc. Int. Database Eng. Appl. Symp. 2009*, ACM, 2009: pp. 31–42.
- [8] O. Görlitz, S. Staab, SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions, in: *Proc. 2nd Int. Workshop Consum. Linked Data*, Bonn, Germany, 2011.
- [9] L. Haas, D. Kossmann, E. Wimmers, J. Yang, Optimizing Queries across Diverse Data Sources, in: *23rd Int. Conf. Very Large Data Bases VLDB 1997*, San Francisco, CA, 1997: pp. 276–285.
- [10] F. Michel, L. Djimenou, C. Faron-Zucker, J. Montagnat, Translation of Relational and Non-Relational Databases into RDF with xR2RML, in: *Proceeding WebIST2015 Conf.*, Lisbon, Portugal, 2015: pp. 443 – 454.
- [11] F. Michel, L. Djimenou, C. Faron-Zucker, J. Montagnat, xR2RML: Non-Relational Databases to RDF Mapping Language, 2014.
- [12] F. Priyatna, O. Corcho, J. Sequeda, Formalisation and Experiences of R2RML-based SPARQL to SQL query translation using Morph, in: *Proceeding World Wide Web Conf. 2014*, Seoul, Korea, 2014.
- [13] M. Rodríguez-Muro, M. Rezk, Efficient SPARQL-to-SQL with R2RML mappings, *Web Semant. Sci. Serv. Agents World Wide Web*. 33 (2015) 141–169.
- [14] A. Schwarte, P. Haase, K. Hose, R. Schenkel, M. Schmidt, Fedx: Optimization techniques for federated query processing on linked data, in: *10th Int. Conf. Semantic Web ISWC11*, Springer, 2011: pp. 601–616.
- [15] J.F. Sequeda, D.P. Miranker, Ultrawrap: SPARQL execution on relational data, *Web Semant. Sci. Serv. Agents World Wide Web*. 22 (2013) 19–39.
- [16] J. Sequeda, S.H. Tirmizi, Ó. Corcho, D.P. Miranker, Survey of directly mapping SQL databases to the Semantic Web, *Knowl. Eng. Rev.* 26 (2011) 445–486.
- [17] J. Unbehauen, C. Stadler, S. Auer, Accessing relational data on the web with SparqlMap, in: *Semantic Technol.*, Springer, 2013: pp. 65–80.
- [18] J. Unbehauen, C. Stadler, S. Auer, Optimizing SPARQL-to-SQL Rewriting, in: *Proc. IIWAS 13*, ACM, 2013: p. 324.
- [19] D. Tomaszuk, *Polskie Towarzystwo Logiki i Filozofii Nauki*, eds., Document-oriented triplestore based on RDF/JSON, in: *Log. Philos. Comput. Sci.*, University of Białystok, 2010: pp. 125–140.