Mapping-based SPARQL access to a MongoDB database

Franck Michel, Catherine Faron Zucker, Johan Montagnat

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES DE SOPHIA ANTIPOLIS
UMR7271

Mapping-based SPARQL Access to a MongoDB Database

Franck Michel, Catherine Faron-Zucker, Johan Montagnat

SPARKS Team

Rapport de Recherche

Version	Date	Description
V1	Dec. 2015	Initial version
V2	Dec. 2015	Rename operator WHERE into FILTER, minor fixes
V3	Jan. 2016	Project constant term maps in function *genProjection* and add AS operator. Move section 4 on bindings to section 3.3. Add section 3.5 about abstract query optimizations.
V4	Feb. 2016	Merge the *join* condition with INNER JOIN abstract operator. Exemplify abstract query optimizations.
V5	Nov. 2016	Deal with RDF collections and containers. Add operator LIMIT in abstract query language and clarify semantics. Deal with rewriting of projection into MongoDB query. Many rewordings, precisions in query optimization. Discussion on the find vs. aggregate MongoDB queries.

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

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[1] 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 also they are increasingly used as a generic-purpose database in many domains. Today, this overwhelming success makes NoSQL databases a natural candidate for RDF-based data integration systems, and potential significant contributors to feed the Web of Data.

In this regard, it shall be necessary to develop SPARQL access methods for heterogeneous databases with different query languages. These methods shall vary greatly depending on the target database query capabilities: for instance, RDBs support joins, nested queries and string manipulations, but this is hardly the case of some NoSQL document stores like MongoDB or CouchDB. Thus, rather than defining yet another SPARQL translation method for each and every query language, we think it is beneficial to consider a two-step approach. First, given a set of mappings of the target database to RDF, a SPARQL query is translated into a pivot abstract query by matching SPARQL graph patterns with relevant mappings. This step can be made generic if the mapping language used is generic enough to apply to a large and extensible set of databases. In a second step, the abstract query is translated into the target database query language, taking into account the specific database capabilities.

Our goal, in this document, is to address this two-step method. Firstly, leveraging previous works on R2RML-based SPARQL-to-SQL methods, we define a method to translate a SPARQL query into a pivot abstract query, utilizing xR2RML [12] to describe the mapping of a target database to RDF. The method determines the minimal set of mappings matching each SPARQL graph pattern, and takes into account join constraints implied by shared variables, cross-references denoted in the mappings, and SPARQL filters. Common query optimization techniques are applied to the abstract query in order to alleviate the work required in the second step. Secondly, we define a method to translate such an abstract query into a concrete query using MongoDB as a target database. In recent years, MongoDB[2] has become the leader in the NoSQL market, as suggested by several indicators including Google searches[3], job offerings[4] and LinkedIn member profiles mentioning MongoDB skills[5]. Some methods have been proposed to translate MongoDB documents into RDF [12], or to use MongoDB as an RDF triple store [22]. Yet, to the best of our knowledge, no work has been proposed so far to query arbitrary MongoDB documents using SPARQL.

In the rest of this section, we review previous works related to the translation of various data sources into RDF. Section 2 presents the xR2RML mapping language and introduces a running example. In section 3, we first describe a method to rewrite a SPARQL query into a pivot abstract query under xR2RML mappings. This relies on bindings

[1] http://www.w3.org/2013/data/
[2] https://www.mongodb.org/
[3] https://www.google.com/trends/explore#q=mongodb,couchdb,couchbase,membase,hbase
[4] http://www.indeed.com/jobtrends/mongodb,mongo,cassandra,hbase,couchdb,couchbase,membase,redis.html
[5] https://blogs.the451group.com/information_management/tag/nosql/

between a SPARQL triple pattern and xR2RML mappings, detailed in section 3.4. Section 0 focuses more specifically on the translation of an abstract query into MongoDB concrete queries. Section 5 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 6, appendix B (section 8) goes over the running example that is been detailed throughout the previous sections.

Related works.

Much work has been achieved during the last decade to expose legacy 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 [5,18,8] or using arbitrary relational schemas [3,20,15,16]. R2RML [6], the W3C RDB-to-RDF mapping language recommendation is now a well-accepted standard and various SPARQL-to-SQL rewriting approaches rely on it [20,15,16]. Other solutions intend to map XML data to RDF [2,1], and the CSV on the Web W3C working group[6] makes a recommendation for the description of and access to CSV data on the Web. RML [7] 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 [12] 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 [22]. MongoGraph[7] is an extension of AllegroGraph[8] to query MongoDB documents with SPARQL queries. It follows an approach very similar to the Direct Mapping approach defined in the context of RDBs [19]: 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 [12] is designed to map an extensible scope of relational and non-relational databases to RDF. Its flexibly adapts to heterogeneous query languages and data models thereby 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 [13]. 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#>
```

[6] http://www.w3.org/2013/csvw/wiki
[7] http://franz.com/agraph/support/documentation/4.7/mongo-interface.html
[8] http://allegrograph.com/

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 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 join 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:bithdate;
     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 xR2RLM 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 collections and containers. 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 collection (`rdf:List`) or container (`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:nestedTermMap` is a means to create nested collections and containers, and to qualify terms of a collection or container 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 Normalization and Restriction of xR2RML within this Document

A standard principle when rewriting a SPARQL query based on mappings is to identify which mappings are good candidates for each triple pattern in the SPARQL graph pattern. For this matching to be accurate, mappings should be expressed such that they shall produce only one type of triple.

However, xR2RML and R2RML triples map may contain any number of predicate-object maps, and each predicate-object map may contain any number (>1) of predicate maps and object maps. Therefore, a triples map with multiple predicate-object maps (and/or multiple predicate and object maps) may generate varying types of RDF triple, and result in the coarse matching of this triples map with triple patterns.

This issue has been addressed in R2RML-based rewriting approaches. Rodríguez-Muro and Rezk [16] propose an algorithm to *normalize* R2RML mappings, that is, rewrite the mapping graph so that a triples map contain at the most one predicate-object map, each having exactly one predicate map and one object map. Although they do not explicitly mention it, authors of [15] and [20] do the same assumption.

Furthermore, the R2RML `rr:class` property introduces a specific way of producing triples such as `"<A> rdf:type <B>"`. Rodríguez-Muro and Rezk propose to replace any `rr:class` property by an equivalent predicate-object map: `[rr:predicate rdf:type; rr:object <A>.]`. This allows for the definition of a rewriting method consistently dealing with all kinds of triple patterns, may they have the `rdf:type` property or any other property.

Consequently, we comply with both propositions above as they apply to R2RML and xR2RML alike. ***Definition 1*** summarizes this:

> ***Definition 1. Normalization of xR2RML Mappings***
>
> *An xR2RML triples map is said to be normalized when:*
> - *It contains exactly at the most one predicate-object map with exactly one predicate map and one object map;*
> - *Its logical source definition does not use the `rr:class` property, instead a regular predicate-object map is used to generate RDF triples with constant predicate `rdf:type`.*

In the rest of this document, we only consider normalized xR2RML triples map.

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 8 goes through the whole method and provides additional explanations. To keep the document focused on the query translation question and for the sake of clarity, the running example does not use iterators nor mixed syntax paths.

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". Similarly, the query in <#Departments>'s logical source 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({})"; xrr:uniqueRef "$.code" ];
  rr:subjectMap [ rr:template "http://example.org/dept/{$.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"
      ] ] ].
```

Finally, we consider the SPARQL query below, that we shall use throughout this document to illustrate the method. Its semantics is to retrieve senior members of departments managed by "H. Dunbar". The query consists of one basic graph pattern composed of two triple patterns tp_1 and tp_2.

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

3 Rewriting a SPARQL Query into an Abstract Query under Normalized xR2RML Mappings

In this section, we first review several SPARQL translation methods (§3.1), with a focus on R2RML-based SPARQL-to-SQL translation. We then define the abstract query language (§3.2), and we describe the translation of a SPARQL query into this abstract query language, along four steps sketched in Figure 1.

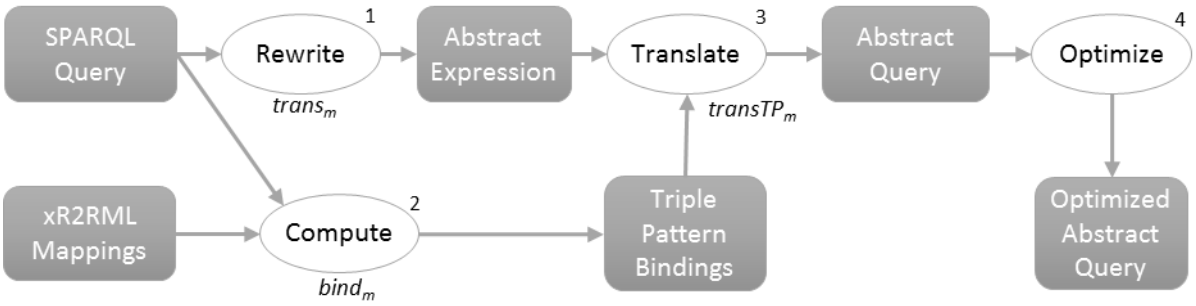


Figure 1: Translation of a SPARQL graph pattern into an optimized abstract query

1. A SPARQL graph pattern is decomposed into an abstract expression exhibiting only operators from the abstract query language (see function $trans_m$, §3.3);
2. Then we identify the xR2RML triples maps likely to generate RDF triples matching each triple pattern (see function $bind_m$, §3.4);
3. Each triple pattern is translated into one or several atomic abstract queries (`<AtomicQuery>`), under the set of xR2RML triples maps identified in step 2. Each atomic query is made as selective as possible by pushing relevant SPARQL filter conditions (function $transTP_m$, §).
4. Finally, the abstract query is optimized by removing e.g. self-joins, self-unions (§3.6).

3.1 R2RML-based SPARQL-to-SQL methods

Various methods have been defined to translate SPARQL queries into another query language, which are generally tailored to the expressiveness of the target query language. For instance, 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 [8,16,18,21]. 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 ALL 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.

Priyatna et al. [15] extend Chebotko's algorithm [5] 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 consider such constraints at the earliest step.

Unbehauen et al. [20] 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 effectively manage variable predicate maps, which clears the first aforementioned 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 aforementioned 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.

Rodríguez-Muro and Rezk [16] 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. NoSQL databases typically make a trade-off between query language expressiveness and scalability. This is particularly the case of MongoDB: joins are not supported, and unions and nested queries are supported under strong restrictions. 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. Thus, an xR2RML-based query processing engine for MongoDB shall evaluate several queries separately (e.g. one per triple pattern), and perform joins and unions afterwards.

Therefore, in order to address a large scope of target databases, we must generalize those approaches to make them independent of any target query language. This will be the object of the subsequent sections.

3.2 Abstract Query Language

Our pivot abstract query language complies with the following grammar, that directly derives from the syntax and semantics of SPARQL [14].

```
<AbstractQuery> ::= <AtomicQuery> | <Query> |
                    <Query> FILTER <SPARQL filter> | <Query> LIMIT <integer>
<Query>         ::= <AbstractQuery> INNER JOIN <AbstractQuery> ON {v1, … vn} |
                    <AtomicQuery> AS child INNER JOIN <AtomicQuery> AS parent
                                  ON child/<Ref> = parent/<Ref> |
                    <AbstractQuery> LEFT OUTER JOIN <AbstractQuery> ON {v1, … vn} |
                    <AbstractQuery> UNION <AbstractQuery>
<AtomicQuery>   ::= {From, Project, Where, Limit}
<Ref>           ::= a valid xR2RML reference
```

The language keeps the names of several SPARQL operators (UNION, LIMIT) and prefers the SQL terms INNER JOIN ON and LEFT OUTER JOIN ON to define join operations more explicitly. Like in the case of SPARQL, an abstract query can be represented as a tree. However, the abstract query language differs from SPARQL in that the tree leaves are *Atomic* A*bstract Queries* (defined later on) whereas they are triple patterns in SPARQL.

Note. As an alternative, we could have used a relational algebra-based notation. However, extending it to account for the semantics of abstract atomic queries (that we define later) would have made the notation cumbersome and error prone. Thus, we felt like a notation based on usual SQL syntax was easier to manipulate, while keeping the required expressive power.

The abstract query language operators are entailed by the dependencies between graph patterns of the SPARQL query. The first INNER JOIN notation is entailed by the join constraints implied by shared variables. The second INNER JOIN notation, including the "AS child", "AS parent" and "ON child/<Ref> = parent/<Ref>" notations, is entailed by the join constraints expressed in xR2RML mappings using referencing object maps. Notation $\{v_1, \ldots v_n\}$, in the join operators, stands for the set of SPARQL variables on which the join is to be performed. "<Ref>" stands for any valid xR2RML data element reference, i.e. this shall be a column name for a tabular data source, an XPath expression for an XML database or a JSONPath expression for a NoSQL document store such as MongoDB and CouchDB.

The computation of these abstract operators shall be delegated to the target database if it supports them (i.e. if the target query language has equivalent operators, this is the case of a relational database), or they may be computed by the query processing engine otherwise (case of MongoDB). Atomic abstract queries (`<AtomicQuery>`) are entailed by translating a triple pattern under a set of xR2RML triples maps.

3.3 Translation of a SPARQL Graph Pattern

Function $trans_m$ (Definition 2) translates a well-designed SPARQL graph pattern [14] into an abstract query that makes no assumption on the target database capabilities. It extends the translation algorithms defined in [5], [20] and [15].

Running Example. Let us give a first simple illustration: 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:

$trans_m(bgp)$
$= trans_m(bgp, true, \infty)$
$= trans_m(tp_1, true, \infty)$ INNER JOIN $trans_m(tp_2, true, \infty)$ ON $var(tp_1) \cap var(tp_2)$
$= transTP_m(tp_1, true, \infty)$ INNER JOIN $transTP_m(tp_2, true, \infty)$ ON {?dept}

Definition 2: 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 well-designed SPARQL graph pattern.
We denote by ***$trans_m(gp)$*** *the translation, under m, of gp into an abstract query.* ***$trans_m$*** *is defined as follows:*

- ***$trans_m(gp) = trans_m(gp, true, \infty)$***
- *if gp consists of a single triple pattern tp,* ***$trans_m(gp, f, l) = transTP_m(tp,$ sparqlCond$(tp, f), l)$***
- if *gp* is (*P* LIMIT *l'*), ***$trans_m(gp, f, l) = trans_m(P, f,$*** min(*l*, *l'*))
- if *gp* is (*P* FILTER *f'*), ***$trans_m(gp, f, l)$*** =
  $trans_m(P, f$ && *f'*, ∞) **FILTER** ***sparqlCond***(*P*, *f* && *f'*) **LIMIT** *l*
- if *gp* is (P_1 AND P_2), ***$trans_m(gp, f, l)$*** =
  $trans_m(P_1, f, \infty)$ **INNER JOIN** ***$trans_m(P_2, f, \infty)$*** **ON** $var(P_1) \cap var(P_2)$ **LIMIT** *l*
- if *gp* is (P_1 OPTIONAL P_2), ***$trans_m(gp, f, l)$*** =
  $trans_m(P_1, f, \infty)$ **LEFT OUTER JOIN** ***$trans_m(P_2,$*** f, ∞) **ON** $var(P_1) \cap var(P_2)$ **LIMIT** *l*
- if *gp* is (P_1 UNION P_2), ***$trans_m$*** (gp, f, l) =
  $trans_m(P_1, f, l)$ **UNION** ***$trans_m(P_2, f, l)$*** **LIMIT** *l*

Simplification: notations "FILTER true" and "LIMIT ∞" may be omitted.

Note. As we describe in subsequent sections, we deal with the FILTER and LIMIT SPARQL clauses in a way that pushes them down into the translation of each triple pattern, in order to make inner queries as selective as possible. In a simplified approach, we do not consider SPARQL solution modifiers OFFSET, ORDER BY and DISTINCT. However, they could be managed in the very same way: additional parameters *offset*, *order by* and *distinct* of the $trans_m$ and $transTP_m$ functions would allow for early selection and sorting at the level of inner queries, while abstract operators OFFSET, ORDER BY and DISTINCT would perform selection and sorting at the level of outer queries.

3.3.1 Management of SPARQL filters

In the usual bottom-up evaluation of a SPARQL query, filters in the outer query do not contribute to the selectivity of inner-queries. A usual consequence, in SPARQL-to-SQL translations, is to rewrite a SPARQL FILTER into a SELECT-WHERE clause that encapsulate sub-queries. The problem in such a strategy is that sub-queries may return very large intermediate results. Consequently, this step is generally followed by an optimization phase, either by implementing specific SQL query optimizations or by relying on the underlying database engine.
In our generalized context, we do not know anything about the target database. Hence, we cannot assume (i) that the target query that we shall come up with can be optimized, or (ii) that the database query evaluation is capable of such optimization. We must therefore consider SPARQL filters at the earliest stage: we propose a generalized management of SPARQL filters that pushes down SPARQL filters into the translation of each triple pattern, in order to make inner queries as selective as possible, thereby limiting the size of intermediate results. This is achieved in function $trans_m$ by the introduction of a SPARQL filter argument initialized to "true" in the expression $trans_m(gp) = trans_m(gp, true, \infty)$. The filter argument shall be updated if the query graph pattern contains a FILTER clause.

Note: Function $trans_m$ relies on function $transTP_m$ to translate each triple pattern into a sub-query. At this stage though, we do not yet explicit the way function $transTP_m$ shall deal with SPARQL filters. We just take care of the fact that, to filter data as early as possible, $transTP_m$ will need to know about the filters that are relevant for the translation of a given triple pattern. Therefore, we have to devise a method to select appropriate conditions from a SPARQL filter.

A SPARQL filter *f* can be considered as a conjunction of n conditions (n >= 1): `C1 && ... Cn`. Function *sparqlCond* (Definition 3) discriminates between conditions with respect to two criteria:

(i) A condition C_i is pushed into the translation of triple pattern *tp* if all variables of C_i show in a *tp*. More simply, a SPARQL condition involving variables ?x and ?y can be pushed to the translation of a triple pattern *tp* only if *tp* contains (at least) both variables ?x and ?y.

(ii) A condition C_i is part of the abstract FILTER operator if at least one variable of C_i is shared by several triple patterns. This FILTER operator represents the join criteria. Example: if C_i contains variable ?x, and variable ?x shows in triple patterns tp_1 and tp_2, then condition C_i will be in the FITLER operator.

Those two criteria are formalized in Definition 3 that defines how function *sparqlCond* discriminates between the filter conditions. Notice that the two criteria are not exclusive: a condition may match both criteria, and thereby show simultaneously in the FILTER operator and in the translation of a triple pattern.

Definition 3: Function *sparqlCond*, splitting SPARQL filter conditions per graph pattern

Let gp be a well-designed SPARQL graph pattern, and TP_{gp} the set of triple patterns of gp.

Let f be the conjunctive SPARQL filter "C_1 && … && C_n", where C_1 to C_n are SPARQL conditions. Let $V(C_n)$ be the set of SPARQL variables named in condition C_n, and V(tp) be the set of SPARQL variables named in triple pattern tp.

Function **sparqlCond** is defined as follows:

- *if gp consists of a single triple pattern tp,* ***sparqlCond****(tp, f) is the conjunction of "true" and the conditions C_i such that all the variables in C_i appear in tp, denoted $V(Ci) \subset V(tp)$.*
- *if gp is any other graph pattern,* ***sparqlCond****(gp, f) is the conjunction of "true" and the conditions C_i such that at least one variable in Ci is shared by several triple patterns of gp, denoted as $\exists v \in V(C_i)$, $\exists tp_1, tp_2 \in TP_{gp}$, $v \in V(tp_1) \cap V(tp_2)$.*

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 query Q depicted 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. *F* = C_1 && C_2 && C_3 && C_4.

- tp_1 involves variables `?x` and `?name1`: no condition involves both `?x` and `?name1`, but C_1 involves `?name1` and no other variable. C_4 involves `?name1` but it also involves variables that are not in tp_1. Hence,

  `sparqlCond(tp1, F) = true && C1`

- tp_2 involves variables `?x` and `?mbpx1`: no condition involves both of them, and no condition involves either `?x` or `?mbox1`. Hence no condition can be pushed into the translation of tp_2, that we denote by:

  `sparqlCond(tp2, F) = true`

- tp_3 involves variables `?y` and `?name1`: no condition involves both `of them`, and no condition involves either `?y` or `?name2`, again:

  `sparqlCond(tp3, F) = true`

- tp_4 involves variables `?y` or `?mbox2`: condition C_2 involves both variables, and C_3 involves only `?mbox2`. Therefore,

  sparqlCond(tp4, F) = C_2 && C_3

Lastly, only conditions C_2 and C_4 involve variables from several triples patterns. We come up with the following abstract query:

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

3.3.2 Management of the LIMIT clause

The way we deal with the LIMIT clause is motivated by the same concern as in the case of SPARQL filters. In the bottom-up evaluation of a SPARQL query, the LIMIT in the outer query does not contribute to the selectivity of inner-queries. Thus, sub-queries may return unnecessary large intermediate results. This issue is generally taken care of by implementing specific query optimizations or by relying on the underlying database engine to do the optimization. But again, in our generalized context, we do not know whether (i) it will be possible to optimize the target query, or (ii) if the database query evaluation engine is capable of such optimization.

Therefore, we propose a method to push down the LIMIT solution modifier into the translation of each triple pattern, in order to make inner queries as selective as possible. Function $trans_m$ has a limit argument *l*, initialized to "∞" in the expression $trans_m(gp) = trans_m(gp, true, \infty)$. The limit argument shall be modified depending on the type of graph pattern passed to $trans_m$. Below we elaborate on the different situations encountered in Definition 2:

- In the rule:

  $\boldsymbol{trans_m}(P\ LIMIT\ l', f, l) = \boldsymbol{trans_m}(P, f, min(l, l'))$

  the SPARQL LIMIT *l'* is passed to the subsequent graph pattern translation. If there is already a limit (from the outer query), then the smallest limit is considered, hence the parameter *min(l, l')*.In the case of a simple triple pattern, the limit argument is passed to the $transTP_m$ function.

- In a graph pattern *P FILTER f'*, we cannot know in advance how many results will be filtered out by the FILTER clause. Consequently, we have to run the query with no limit and apply the filter. The explains the "∞" parameter in:

  $\boldsymbol{trans_m}(gp, f, l) = \boldsymbol{trans_m}(P, f \&\& f', \infty)$ **FILTER** $\boldsymbol{sparqlCond}(P, f \&\& f')$ **LIMIT** *l*

- Similarly, in the case of an inner or left join, we cannot know in advance how many results will be returned. Consequently, we have to run both queries with no limit, apply the join, and only then limit the number of results. This explains the "∞" parameter in expressions:

  $\boldsymbol{trans_m}(P_1, f, \infty)$ **INNER JOIN** $\boldsymbol{trans_m}(P_2, f, \infty)$ **ON** $var(P_1) \cap var(P_2)$ **LIMIT** *l*

  and

  $\boldsymbol{trans_m}(P_1, f, \infty)$ **LEFT OUTER JOIN** $\boldsymbol{trans_m}(P2, f, \infty)$ **ON** $var(P_1) \cap var(P_2)$ **LIMIT** *l*

- Finally, in the union case, none of the two operands of the UNION should return more than the limit parameter. Hence the translation:

  $\boldsymbol{trans_m}(P_1\ UNION\ P_2, f, l) = \boldsymbol{trans_m}(P_1, f, l)$ **UNION** $\boldsymbol{trans_m}(P_2, f, l)$ **LIMIT** *l*

3.4 Binding xR2RML triples maps to triple patterns

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

In the following, we assume that xR2RML triples are normalized in the sense defined in section 2.3. Also, we denote by *TM.sub*, *TM.pred* and *TM.obj* respectively the subject map, the predicate map and the object map of the normalized triples map *TM*. Furthermore, we adapt the concept of *triple pattern binding* defined by Unbehauen et al. as follows:

Definition 4: Triple pattern binding (adapted from Unbehauen et al. [20])
Let m be an xR2RML mapping graph consisting of a set of xR2RML triples map, and tp be a triple pattern.
A triples map TM ∈ m is ***bound to tp*** *if it is likely to produce triples matching tp.*
A ***triple pattern binding*** *is a pair (tp, TMSet) where TMSet is the set of triples maps of m that are bound to tp.*

Function $bind_{m,}$ determines, for a graph pattern *gp*, the bindings of each triple pattern of *gp*. It takes into account join constraints implied by shared variables, and the SPARQL filter constraints whose unsatisfiability can be verified statically. This is achieved by means of two functions: *compatible* and *reduce.* These functions were introduced by Unbehauen et al [20] in the SPARQL-to-SQL context, but important details were left untold. Especially, the authors did not formally define what the compatibility between a term map and a triple pattern term means, and they did not investigate the static compatibility between a term map and a SPARQL filter. Below, we define these functions in details and extend them to fit in our context of an abstract query language.

Definition 5: function $bind_m$
Let m be a mapping graph consisting of a set of xR2RML triples maps, and gp be a well-designed graph pattern. We denote by $\boldsymbol{bind_m}(gp)$ *the set of triple pattern bindings of gp under m, defined recursively as follows:*

- $\boldsymbol{bind_m}(gp) = \boldsymbol{bind_m}(gp, true)$
- *if gp consists of a single triple pattern tp,*
  - *if tp.pred is one of rdf:first, rdf:rest, rdf:nil, rdf:_1, rdf:_2 etc., or tp.pred is rdf:type and tp.obj is one of rdf:List, rdf:Bag, rdf:Seq, rdf:Alt, then ignore this triple pattern.*
  - *Otherwise,* $\boldsymbol{bind_m}(gp, f)$ *is the pair (tp, TMSet) where TMSet = {TM | TM ∈ m ∧* $\boldsymbol{compatible}(TM.sub, tp.sub, f) \wedge \boldsymbol{compatible}(TM.pred, tp.pred, f) \wedge \boldsymbol{compatible}(TM.obj, tp.obj, f)$*}*
- *if gp is* $(P_1\ AND\ P_2)$, $\boldsymbol{bind_m}(gp, f) = \boldsymbol{reduce}(\boldsymbol{bind_m}(P_1, f), \boldsymbol{bind_m}(P_2, f)) \cup \boldsymbol{reduce}(\boldsymbol{bind_m}(P_2, f), \boldsymbol{bind_m}(P_1, f))$
- *if gp is* $(P_1\ OPTIONAL\ P_2)$, $\boldsymbol{bind_m}(gp, f) = \boldsymbol{bind_m}(P_1, f) \cup \boldsymbol{reduce}(\boldsymbol{bind_m}(P_2, f), \boldsymbol{bind_m}(P1, f))$
- *if gp is* $(P_1\ UNION\ P_2)$, $\boldsymbol{bind_m}(gp, f) = \boldsymbol{bind_m}(P_1, f) \cup \boldsymbol{bind_m}(P_2, f)$
- *if gp is (P FILTER f'),* $\boldsymbol{bind_m}(gp, f) = \boldsymbol{bind_m}(P, f \,\&\&\, f')$

Running Example: Before we get into the details, let us illustrate informally the way function $bind_m$ infers triple pattern bindings. Triple pattern tp_1 is:

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

The subject term, `<http://example.org/staff/Dunbar>`, could be produced by the template string

```
http://example.org/staff/{$['lastname','familyname']}
```

in the subject map of triples map <#Staff>. On the contrary, it could not be produced by the template string in triples map <#Department>:

```
http://example.org/dept/{$.code}
```

Additionally, the predicate part of tp_1 matches the constant predicate map of triples map <#Staff>. Consequently, triples map <#Staff> is bound to tp_1 and <#Department> is not.

The very same reasoning lets us deduce that triple pattern tp_2 could *be pr*oduced by triples map <#Department>, but not by triples map <#Staff>. Finally, we obtain the following bindings:

$bind_m(bgp) = \{ (tp_1, \{<\#Staff>\}), (tp_2, \{<\#Departments>\}) \}$

3.4.1 Case of RDF collections and containers

An xR2RML triples may generate RDF collections and containers using specific term type values `xrr:RdfList`, `xrr:RdfSeq`, etc. Some SPARQL queries may access the members of such collections and containers. In our running example let us assume that triples map `<#Departments>` generates a list of senior members instead of one triple per senior member, with the amended predicate-object map below:

```
rr:predicateObjectMap [
   rr:predicate ex:seniorMembers;
   rr:objectMap [
      xrr:reference "$.members[?(@.age >= 40)].name";
      rr:termType xrr:RdfList. ] ].
```

In this context, a user may issue queries about the list terms, such as:

```
SELECT ?senior WHERE {
    ?dept ex:seniorMembers ?seniors.       // tp1
    ?seniors a rdf:List.                   // tp2
    ?seniors rdf:first ?senior. }          // tp3
```

Trivially, triples map `<#Departments>` can be bound to triple pattern tp_1 since their predicate parts match (`ex:seniorMembers`). Let us now consider tp_2 and tp_3. In our running example, there is no triples map with constant predicate "`rdf:type`" and constant object "`rdf:List`" that could be bound to tp_2. Similarly, there is no triples map with constant predicate "`rdf:first`" that could be bound to tp_3. With no bindings, no RDF triples matching tp_2 nor tp_3 are generated. Consequently, the join (logical AND) between triples patterns tp_1, tp_2 and tp_3 will return an empty result set, whereas there are solutions to that SPARQL query.

To avoid coming up with empty bindings, the $bind_m$ function simply ignores triple patterns that pertain to RDF collections and containers (triple patterns whose predicate is one of `rdf:first`, `rdf:rest`, `rdf:nil`, `rdf:_1`, `rdf:_2` etc., or the predicate is `rdf:type` and the object is one of `rdf:List`, `rdf:Bag`, `rdf:Seq`, `rdf:Alt`). This does not prevent from generating a query, executing it and obtaining relevant results; what we do is simplify the SPARQL query by ignoring those triple patterns that we cannot deal with at this stage, thus making the SPARQL query less specific. As a result, the generated target database query may return more results than actually expected; finally, we may generate RDF triples that do not match the SPARQL query. To work out this issue, we propose to perform a *late SPARQL query evaluation* that shall rule out all unneeded triples. This step is described further on in section 5.

3.4.2 Reduction of bindings, function reduce

At this point, we have come up with bindings of xR2RML triples maps to each triple pattern of the SPARQL query. This is performed for each triple pattern, regardless of other triple patterns. Yet, triple patterns are not independent of each other: shared variables induce join constraints that can help us find out inconsistent bindings. For instance, consider two triple patterns tp_1 and tp_2 that have a shared variable *v*, triples map TM_1 is bound to tp_1 and triples map TM_2 is bound to tp_2. If the term map associated to *v* in TM_1 generates literals, whereas the term map associated to *v* in TM_2 generates IRIs, these term maps are incompatible. Consequently, we can rules out TM_1 from the bindings of tp_1 and TM_2 from the bindings of tp_2. This is what we call "reduction of bindings".

To that end, function *join* examines the variables shared by two triple patterns to detect unsatisfiable join constraints:

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.
*We denote by **join**(tpb_1, tpb_2) the set of pairs $(TM_1, TM_2) \in TMSet_1 \times TMSet_2$, such that, for each $v \in V$, it holds that **compatibleTermMaps**$(TM_1.pos_{tp1}(v), TM_2.pos_{tp2}(v))$.*

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.
Example. Let us consider two triple patterns tp_1 *and* tp_2 with a shared variable ?y:
tp_1 = `?x knows ?y` ⇨ variable ?y is in object position of tp_1: pos_{tp1}(?y) = obj.
tp_2 = `?y knows <#me>` ⇨ variable ?y is in subject position of tp_2: pos_{tp2}(?y) = sub.
We assume the following bindings: $tpb_1 = (tp_1, \{TM_{1a}, TM_{1b}\})$, $tpb_2 = (tp_2, \{TM_{2a}, TM_{2b}\})$.
Finally, we assume the following compatibility matrix between the term maps concerned by variable ?y in the triples maps bound of tp_1 and tp_2:

	TM_{1a}.obj	TM_{1b}.obj
TM_{2a}.sub	✓	✓
TM_{2b}.sub	×	×

In this context, we obtain:

$join(tpb_1, tpb_2) = \{ (TM_{1a}, TM_{2a}), (TM_{1b}, TM_{2a}) \}$
$join(tpb_2, tpb_1) = \{ (TM_{2a}, TM_{1a}), (TM_{2a}, TM_{1b}) \}$

We can now define function *reduce*, that computes the minimal set of triple maps bound to each triple pattern (minimal with respect to the join constraints considered).

Definition 7. Reduction of bindings: 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 $TMSet_2 \subseteq m$.

*We denote by **reduce**(tpb_1, tpb_2) the binding to tp_1 of triples maps that appear as the left component of pairs obtained from **join**(tpb_1, tpb_2). Formally:*

***reduce**$(tpb_1, tpb_2) = (tp_1, \{ TM_1 \in TMSet_1, \exists (TM_1, TM_2) \in join(tpb_1, tpb_2) \})$*

Following up on the example above, *reduce* makes a simple projection of the left term of the pairs computed by *join*:

$join(tpb_1, tpb_2) = \{ (\mathbf{TM_{1a}}, TM_{2a}), (\mathbf{TM_{1b}}, TM_{2a}) \}$
$reduce(tpb_1, tpb_2) = (tp_1, \{\mathbf{TM_{1a}}, \mathbf{TM_{1b}}\})$

$join(tpb_2, tpb_1) = \{ (\mathbf{TM_{2a}}, TM_{1a}), (\mathbf{TM_{2a}}, TM_{1b}) \}$
$reduce(tpb_2, tpb_1) = (tp_2, \{\mathbf{TM_{2a}}\})$

3.4.3 Compatibility of term maps, triple pattern terms and SPARQL filters

To decide whether an xR2RML mapping (a triples map) can be bound to a triple pattern, we must verify whether the triples map can potentially generate RDF triples matching the triple pattern. More precisely, we look for incompatibilities between each term map of the triples map, and the corresponding term in the triple pattern (a triple pattern term may be a literal, an IRI, a blank node or a variable). In function $bind_m$, this is denoted by the expression:

compatible(TM.sub, tp.sub, f) ∧ compatible(TM.pred, tp.pred, f) ∧ compatible(TM.obj, tp.obj, f)}

Function *compatible* (Definition 8) checks if a term map (*termMap*) is compatible with a term of a triple pattern (*tpTerm*) and a SPARQL filter *f*, i.e. it verifies whether there is any contradiction between *termMap* and *tpTerm*, or between *termMap* and *f*. Unbehauen et al defined the compatibility of *termMap* and *tpTerm* as: *tpTerm ∈ range(termMap)*, but no description of the *range* function was provided. Therefore, we precise this definition.

A term map is always considered compatible with a variable, unless a SPARQL filter contradicts the term map. The later situation is identified in function *compatibleFilter* (Definition 9). It pertains to type constraints expressed using SPARQL operators *isIRI*, *isLiteral* or *isBlank*, as well as language and data type constraints expressed using operators *lang, langMatches* or *datatype*. For instance, if variable `?var` is matched with an object map that produces literals (`rr:termType rr:Literal`), and the SPARQL contains the necessary condition `isIRI(?var)`, then this condition is unsatisfiable.

When the triple pattern term is not a variable, function *compatible* identifies the similar situations wherein the triple pattern term and the term map cannot match with regards to the type of the triple pattern term[9] (literal, IRI, blank node), its language tag (e.g. `"string"@en`) or its data type (e.g. `10^^xsd:integer`).

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 an xR2RML triples map TM and f be a SPARQL filter.

It holds that termMap is compatible with tpTerm and f, denoted by ***compatible(termMap, tpTerm, f)****, if termMap is compatible with filter f, denoted by* ***compatibleFilter****(termMap, f), and either (i) tpTerm is a variable or (ii) none of the following assertions holds:*

- *tpTerm is a literal and the term type of termMap is not rr:Literal;*
- *tpTerm is an IRI and the term type of termMap is not rr:IRI;*
- *tpTerm is a blank node and the term type of termMap is not one of {rr:BlankNode, xrr:RdfList, xrr:RdfBag, xrr:RdfSeq, xrr:RdfAlt};*
- *tpTerm is a literal with a language tag L, and the language of termMap is 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.* ***¬compatible*** *(termMap.parentTriplesMap.subjectMap, tpTerm, f).*

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

Let termMap be an xR2RML term map and f be a SPARQL filter. It holds that termMap is compatible with f, denoted as ***compatibleFilter****(termMap, f), if f ="true" or none of the following assertions holds:*

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

[9] Recall that the term type may be explicitly stated with the `rr:termType` property, or have a default value. 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.

The compatibility between two term maps is defined by [20] as the condition:

$range(tm_1) \cap range(tm_2) \neq \emptyset$

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 tm_1 and tm_2 be two xR2RML term maps. It holds that tm_1 and tm_2 are compatible, denoted by **compatibleTermMaps**(tm_1, tm_2), if none of the following assertions holds:*

(1) tm_1 and tm_2 have different term types (rr:Literal, rr:BlankNode, rr:IRI, xrr:RdfList, xrr:RdfSeq, xrr:RdfBag, xrr:RdfAlt).

(2) tm_1 and tm_2 have different language tags, or one has a language tag and the other does not.

(3) tm_1 and tm_2 have different data types, or one has a data type and the other does not.

(4) tm_1 and tm_2 are both template-valued, and they have incompatible template strings.

*tm_1 (resp. tm_2) is a ReferencingObjectMap and the subject map of its parent triples maps is not compatible with tm_2 (resp. tm_1), i.e. **¬compatibleTermMaps**(tm_1.parentTriplesMap.subjectMap, tm_2), (resp. **¬compatibleTermMaps**(tm_1, tm_2.parentTriplesMap.subjectMap)).*

Any of the assertions (1) to (5) is a sufficient condition to entail that two term maps are incompatible. We could be tempted to consider the additional assertion (6):

tm_1 and tm_2 have different types (constant-valued, reference-valued or template-valued).

This would be wrong however. For instance, a reference-valued term map returning a URL from the database with a term type rr:IRI could be compatible with a template-valued term map building a URL from some other value of the database. Therefore, considering assertion (6) in our definition may lead to state that two term maps are incompatible although they are.

3.5 Translation of a SPARQL Triple Pattern: Atomic Abstract Query

The $trans_m$ function relies on the $transTP_m$ function (Definition 11) to translate to translate a single triple pattern *tp* into an abstract query under the set of compatible xR2RML triples maps (the triples maps of *m* bound to *tp* by function $bind_m$). Below, we first define function $transTP_m$ as well as the concept of concept of Atomic Abstract Query. In subsequent sections, we go through the algorithm of $transTP_m$ and the details of how atomic abstract queries are $transTP_m$ computed.

Definition 11: Function $transTP_m$:

Let m be an xR2RML mapping graph consisting of a set of xR2RML triples maps, gp be a well-designed graph pattern, and tp a triple pattern of gp. Let l be the maximum number of query results, and f be a SPARQL filter expression. Let $getBoundTMs_m$(gp, tp, f) be the function that, given gp, tp and f, returns the set of triples maps of m that are bound to tp in $bind_m$(gp, f).

*We denote by **$transTP_m$**(tp, f, l) the translation, under $getBoundTMs_m$(gp, tp, f), of tp into an abstract query whereof results can be translated into at most l RDF triples matching "tp FILTER f". The resulting abstract query is a union of per-triples-map subqueries, where a subquery is either an Atomic Abstract Query or the join of two Atomic Abstract Queries.*

Definition 12: ***Atomic Abstract Query***:

An Atomic Abstract Query is an abstract query obtained by matching a SPARQL triple pattern tp with an xR2RML triples map bound to tp. It is denoted by ***{From, Project, Where}****, where:*

- *"**From**" consists of the triples map's logical source;*
- *"**Project**" is the set of xR2RML data element references that are projected, i.e. returned as part of the query results. There are three types of projection:*
  - *<xR2RML reference>*
  - *<xR2RML reference>* ***AS*** *<SPARQL variable>*
  - *<Constant value>* ***AS*** *<SPARQL variable>*
- *"**Where**" is a set of conditions on xR2RML data element references, entailed by matching terms of tp with (i) their corresponding term map in the triples map, or (ii) with a SPARQL filter. Three types of condition exist:*
  - ***isNotNull****(<xR2RML reference>)*
  - ***equals****(value, <xR2RML reference>)*
  - ***sparqlFilter****(<SPARQL filter>)*
- ***"Limit"*** *is the maximum number of results that must be returned by the atomic query.*

{From, Project, Where} *may be used as a simplified notation of* ***{From, Project, Where, ∞}****.*

3.5.1 Algorithm of function transTP$_m$

Function *transTP$_m$* is described in further details in Algorithm 1. It consists on a loop on all triples maps bound to *tp* (line 4 to 23). The result query is a UNION of all per-triples-map subqueries (line 22). For each triples map *TM*, the algorithm constructs the *From*, *Project* and *Where* parts of an atomic abstract query (lines 5-7). Then, two cases are distinguished:

- If the object maps if a regular object map (no cross-reference), then a single atomic abstract query is created: {*From, Project, Where*}.
- When the object map is a referencing object map, e.g. child triples map *TM$_1$* produces the subject and predicate terms while parent triples map *TM$_2$* produces object terms, a second atomic abstract query is constructed (lines 12-14) to account for *TM$_2$*'s information: *PFrom* is *TM$_2$*'s logical source, *PProject* projects the data element references from *TM$_2$*'s subject map, and *PWhere* embeds conditions on the data element references from *TM$_2$*'s subject map. Then, the abstract query corresponding to the couple (*tp*, *TM$_1$*) is the INNER JOIN of the two atomic abstract queries (lines 15-18):

```
{From, Project, Where} AS child
INNER JOIN
{PFrom, PProject, PWhere} AS parent
ON child/childRef = parent/parentRef
```

  where `childRef` and `parentRef` denote the values of properties `rr:child` and `rr:parent` respectively.

Note: Interestingly, INNER JOIN operators of the abstract query language may be implied by shared SPARQL variables (in function *trans$_m$*), as well as cross-references denoted in the mappings as explained above. Similarly, UNION operators of the abstract query language may arise from the SPARQL UNION (in function *trans$_m$*), or the binding of several triples maps to the sale triple pattern.

Algorithm 1: Translation of a triple pattern into an abstract query (function transTP$_m$).
f is a SPARQL filter, l is the maximum number of results.

1 ***Function*** transTP$_m$(tp, f, l):

```
2   Query ← <empty query>
3   BoundTMs ← getBoundTMs_m(gp, tp, f)
4   for each TM ∈ BoundTMs do
5       From ← <TM's logicalSource>
6       Project ← genProjection(tp, TM)
7       Where ← genCond(tp, TM, f)
8       OM ← TM.predicateObjectMap.objectMap
9       if OM is a referencing object map then
10          childRef ← OM.joinCondition.child
11          parentRef ← OM.joinCondition.parent
12          PFrom ← <OM.parentTriplesMap's logical source>
13          PProject ← genProjectionParent(tp, TM)
14          PWhere ← genCondParent(tp, TM, f)
15          Q ←  {From, Project, Where, ∞} AS child
16                  INNER JOIN
17                  {PFrom, PProject, PWhere, ∞} AS parent
18                  ON child/childRef = parent/parentRef
19                  LIMIT l
20      else
21          Q ← {From, Project, Where, l}
22      end if
23      Query ← Query UNION Q LIMIT l
24  end for
25  return Query
```

Running Example. We have already show that:

$bind_m(bgp) = \{ (tp_1, \{<\#Staff>\}), (tp_2, \{<\#Departments>\}) \}$

Hence,

$getBoundTMs_m(gp, tp_1, true) = \{<\#Staff>\}$

$getBoundTMs_m(gp, tp_2, true) = \{<\#Departments>\}$

Now let us run function *transTP*$_m$ for tp_2:

```
tp2 = ?dept ex:hasSeniorMember ?senior.
transTPm(tp2, true, ∞) =
    { From    ← {[xrr:query "db.departments.find({})"]}
      Project ← genProjection(tp2, <#Departments>)
      Where   ← genCond(tp2, <#Departments>, true)
      Limit   ← ∞ }
```

In the case of tp_1, the bound triples map, <#Staff>, contains a referencing object map. Consequently, the translation entails an INNER JOIN operator on the xR2RML references mentioned in the *joinCondition* property of the referencing object map:

```
tp1 = <http://example.org/staff/Dunbar> ex:manages ?dept
transTPm(tp1, true, ∞) =
```

```
{ From    ← {[xrr:query "db.staff.find({})"]}
  Project ← genProjection(tp1, <#Staff>)
  Where   ← genCond(tp1, <#Staff>, true)
} AS child
INNER JOIN
{ From    ← {[xrr:query "db.departments.find({})"]}
  Project ← genProjectionParent(tp1, <#Staff>)
  Where   ← genCondParent(tp1, <#Staff>, true)
} AS parent
ON child/$.manages.* = parent/$.dept
LIMIT ∞
```

Note: From now on, we shall omit the *Limit* part in an atomic query, and the *LIMIT* abstract query operator, when the limit value is ∞.

3.5.2 Computing Atomic Abstract Queries

We now go through further details about how the *From, Project* and *Where* parts of an abstract atomic query are computed. The detailed algorithms of functions *genProjection, genProjectionParent, genCond* and *genCondParent* are given in section 7.

From. The *From* part provides the concrete query that the abstract query relies on. It contains the logical source of triples map TM that consists of `the xrr:query` property and an optional iterator (property `rml:iterator`). In our running example, the *From* part of tp_2 is simply:

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

In the case of tp_1, two atomic abstract queries are created, each referring to the logical source of one triples map.

Project. The projection part of a database query restricts the set of attributes that shall be returned in the query response. In relational algebra, this is denoted by the π operator: $\pi_{a1, \ldots an}(R)$ is the set obtained when the components of the tuple *R* are restricted to the set $\{a_1, \ldots, a_n\}$. In the context of a relational database, the attributes are columns, whereas in the context of a JSON document store, attributes are fields of JSON documents.

The *genProjection* and *genProjectionParent* functions select the xR2RML data element references that must be projected. When an xR2RML reference is matched with a SPARQL variable in the triple pattern, it is projected with notation "AS <*variable name*>". In the running example, the subject and object of tp_2 are variables "?dept" and "?senior", respectively matched with subject map's reference "`$.code`" and object map's reference "`$.members[?(@.age >= 40)].name`". Consequently:

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

How the JSON fields named in the JSONPath expression (members, age, name) are actually projected is not relevant at this point: the atomic abstract query simply names xR2RML references to be projected, it does not tell how they will be projected in a concrete target database query.

Additionally, when a referencing object map is involved (cross-reference), functions *genProjection* and *genProjectionParent* project the joined references mentioned in an xR2RML *rr:joinCondition*. This is illustrated with tp_1 in the running example. Triples map <#Staff> has a referencing object map whose child and parent references are projected: function *genProjection* projects the child reference "`$.manages.*`", while function *genProjectionParent* projects the parent reference "`$.dept`":

genProjection(tp_1, <#Staff>) = {**$.manages.***}
genProjectionParent(tp_1, <#Staff>) = {**$.dept**, $.code AS ?dept}

Note that, since the joined references are not matched with a variable of the SPARQL query, they are projected without the AS operator.

A last case regards constant term maps: when a SPARQL variable is matched with a constant term map, that constant value is projected as the variable. For instance, let us consider a new triple pattern tp_3:

<http://example.org/staff/Dunbar> ?predicate ?dept

Variable ?predicate is matched with the constant predicate map in triples map <#Staff>. To account for this constant projection, we would write:

genProjection(tp_3, <#Staff>) = {**ex:manages AS ?predicate**}

Where. The *genCond* function computes the *Where* part by matching each triple pattern term with its corresponding term map. In relational algebra, this would be denoted by the selection operator σ: $\sigma_{\varphi}(R)$ selects all tuples in R for which the proposition φ holds. In our context, R is the triples map logical source, and φ is a conjunction of conditions of three types: not-null, equality or SPARQL filter. Below, we determine the type of condition entailed according to the type of triple pattern term and the type of term map that are matched with each other.

(a) A SPARQL variable in the triple pattern entails a non-null condition on the corresponding xR2RML reference(s). Let us exemplify this: the subject part of tp_2 is variable ?dept; it is matched with the subject map of triples map <#Departments>, whose template string is "http://example.org/dept/{$.code}". Without any further knowledge on ?dept, the match simply states that the subject map must return a valid value, in other words the reference "$.code" must not return null. This entails a condition: isNotNull($.code). When applied to the object of tp_2, the same method entails a second not-null condition: isNotNull($.members[?(@.age >= 40)].name).

As a result, we can already deduce the evaluation of function *genCond* on tp_2:

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

(b) A constant term in the triple pattern (literal or IRI) entails an equality condition. In our running example, the subject part of tp_1, <http://example.org/staff/Dunbar>, is matched with the template string "http://example.org/staff/{$['lastname','familyname']}" of <#Staff>'s subject map. This entails the equality condition:

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

stating that either "lastname" or "familyname" must equal "Dunbar".

(c) When a constant term map is matched with a triple pattern term,

- If the triple pattern term is also constant (literal or IRI), then no condition is entailed, *e.g.*: the predicate part of tp_2, ex:hasSeniorManager, matches the constant predicate map of triples map <#Departments>. There is nothing more we can deduct from this.
- If the triple pattern term is a variable, then the variable is bound to the constant value of the term map. This case is already taken care of in the projection part, that we illustrated above with triple pattern tp_3:

genProjection(tp_3, <#Staff>) = {ex:manages AS ?predicate}

(d) When a referencing object map is matched with a triple pattern term, a not-null condition must be added for each of the joined references to ensure that only valid values are inner joined. This is achieved by function *genCond* for the child triples map and function *genCondParent* for the parent triples map. In the running example, the object of tp_1, variable ?dept, is matched with the referencing object map of <#Staff>, whose join condition is:

```
rr:joinCondition [ rr:child "$.manages.*"; rr:parent "$.dept"]
```

This entails a not-null condition on the child reference in the first atomic query: `isNotNull($.manages.*)`, and a not-null condition on the parent reference in the second atomic query: `isNotNull($.dept)`. Finally, we get the following conditions for tp_1:

```
genCond(tp1, <#Staff>, true) = {
                equals("Dunbar", $['lastname','familyname']),
                isNotNull($.manages.*) }                          // join condition
genCondParent(tp1, <#Staff>, true) = {
                isNotNull($.dept),                                // join condition
                isNotNull($.code) }                               // variable ?dept
```

(e) SPARQL filter. In (a) to (d), not-null or equality conditions are entailed. SPARQL filters, on the other hand, have a much richer variety of functions and operators, including a subset of XQuery 1.0 and XPath 2.0. By construction, the SPARQL filter *f* passed as argument of $transTP_m$ mentions only variable of the triple pattern. The atomic abstract query keeps track of the filter using notation `sparqlFilter(f)`. If variables mentioned in the filter are matched with an xR2RML reference (a reference-valued term map or a template-valued term map), the xR2RML reference is provided in the *Project* part of the atomic query. Let us consider the example below:

```
{ From    ← { … }
  Project ← { $.arrayField.* AS ?x }
  Where   ← { sparqlFilter(?x >= 5 && ?x < 10) }
}
```

The SPARQL filter "`?x >= 5 && ?x < 10`", and the *Project* part states that the values of ?x are generated by the xR2RML reference "$.arrayField.*". At the level of the abstract query language, the SPARQL filter and the projection are kept as is. The filter shall be translated into a target query in the subsequent translation step, *i.e.* when translating from the abstract query language to the target database query language.

Limit. The Limit part is a positive integer value representing the maximum number of results that the atomic query should return. It is provided in the atomic query with the incentive of limiting the size of intermediate results from inner queries.

Running Example. We now sum up the way function transTPm computes abstract queries.

The SPARQL basic graph pattern *bgp* consists of two triple patterns:

```
tp1 = <http://example.org/staff/Dunbar> ex:manages ?dept
tp2 = ?dept ex:hasSeniorMember ?senior.
```

Given the bindings for each triple pattern:

$bind_m(bgp) = \{ (tp_1, \{<\#Staff>\}), (tp_2, \{<\#Departments>\}) \}$

We can write each triple pattern into an abstract query:

```
transTPm(tp2, true, ∞) =
   {  From    ← { [xrr:query "db.departments.find({})"] }
      Project ← { $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior }
      Where   ← { isNotNull($.code), isNotNull($.members[?(@.age >= 40)].name) }
    }
```

and

```
transTPm(tp1, true, ∞) =
    { From    ← { [xrr:query "db.staff.find({})"] }
      Project ← { $.manages.* }
```

```
    Where   ← { equals("Dunbar", $['lastname','familyname']),
                isNotNull($.manages.*) }
  } AS child
  INNER JOIN
  { PFrom    ← {[xrr:query "db.departments.find({})"]}
    PProject ← { $.dept, $.code AS ?dept }
    PWhere   ← { isNotNull($.dept), isNotNull($.code) }
  } AS parent
ON child/$.manages.* = parent/$.dept
```

3.6 Abstract query optimization

At this point, the method we have exposed translates SPARQL graph patterns into effective abstract queries, *i.e.* they preserve the semantics of SPARQL queries. Yet, shortcomings such as unnecessary complexity or redundancy may lead to the generation of inefficient queries, and in turn entail poor performances. Although we may postpone the query optimization to the translation into a concrete query language, it is interesting to figure out what optimizations can be done on the abstract representation first, and leave only database-specific optimizations to the subsequent stage.

SPARQL-to-SQL methods proposed various query optimizations [21,16,8,18]. In this section, we review some of these techniques, referring to the terminology defined in [21]. We show how these optimizations can be relevantly adapted to fit in the context of our abstract query language. In particular, we show that our translation method implements some of these optimizations by construction. In addition, we propose a new optimization, the *Filter Propagation*. To our knowledge, it was not proposed in SPARQL-to-SQL rewriting methods.

Examples of this section are provided relying on the MongoDB example. Yet again, recall that these optimizations apply at the abstract query level and, consequently, they are generic and may apply with any other target database.

3.6.1.1 Filter Optimization

In a naive approach, the management of template strings can lead to inefficient target queries. Typically, when the translation of an R2RML template relies on the SQL string concatenation, a SPARQL ca ben rewritten into something like this:

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

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

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

In our approach, equality conditions generated by the *genCond* and *genCondParent* functions apply to xR2RML references rather than on template-generated values, hence the *Filter Optimization* is enforced by construction.

3.6.1.2 Filter pushing

As we have mentioned in section 3.3, the translation of a SPARQL filter into an encapsulating SELECT-WHERE clause makes inner queries very little selective, and the query evaluation process may have to deal with unnecessarily large intermediate results. In our approach, *Filter pushing* is enforced by construction by the *sparqlCond* function: relevant SPARQL filters are pushed down, as much as possible, in the translation of individual triple patterns.

3.6.1.3 Self-Join Elimination

The self-join issue has been investigated for R2RML-based SPARQL-to-SQL translation [8,21]: it occurs when a relational table is joined with itself. We generalize this in our xR2RML-based translation: a self-join may occur when two triples maps, bound to two joined triple patterns, share the same logical source (`xrr:query` and `rml:iterator`). The atomic abstract queries Q_1 and Q_2 representing the two triple patterns are in a self-join situation when the following conditions are met:

(a) Both queries have the same *From* part, *i.e.* they refer to the same logical source, or one logical source is a subset of the other.

(b) They have at least one shared variable on which the join is to be performed.

(c) Both queries project the same xR2RML data element reference(s) as the same shared variable(s), *e.g.* if the xR2RML reference `"$.x"` is projected as variable `?x` is the left query, then the same projection must exist in the right query for the join to be a self-join. On the contrary, if projections are different: `"$.x1 AS ?x"` in Q_1 and `"$.x2 AS ?x"` is Q_2, then this is a regular join.

(d) Each reference projected as a shared variable must uniquely identify a document within query results: if Q_1 and Q_2 both have projection `"$.x AS ?x"`, then the xR2RML reference "$.x" must be declared as unique in at least one of the logical sources:

```
xrr:logicalSource [ xrr:query "…"; xrr:uniqueRef "$.x" ]
```

The *Self-Join Elimination* consists in merging both atomic queries into a single one, wherein the *Project* part merges the *Project* parts of both queries, and the *From* part is the most specific of the two *From* parts.

Condition (d) is illustrated in Figure 2 : on both sides, Q1 and Q2 depict the result of the atomic queries. Since they have the same *From* part, Q1 and Q2 actually contain the same results. On the left, two documents have a field x with value 1. Yet, x is not unique: several documents may have field x=1. Therefore, the self-join must not be eliminated as this is a regular join: documents with id 4 and 5 can be joined together. On the right, the logical source of one of the atomic queries declares the xR2RML reference "$.x" as unique with property `xrr:uniqueRef`. In that case, both documents with x=1 are necessarily the exact same document. In turn, this self-join can be eliminated.

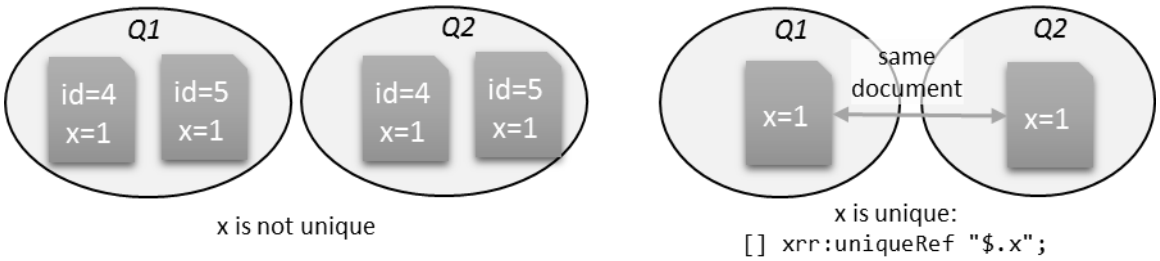


Figure 2: Self-join elimination on a unique xR2RML reference

Note: the `xrr:uniqueRef` property has been added to the xR2RML mapping language to enable the self-join elimination at the abstract query level. In conventional SPARQL-to-SQL approaches, it is actually not necessary: the query rewriting application can inspect the database schema, looking primary key or unicity constraints. On the contrary, with schema-less databases like MongoDB, unicity constraints must be stated declaratively.

Running Example. The translation of tp_1 and tp_2 entails the following abstract query:

$trans_m(bgp, true, \infty)$ =

```
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← {$.dept, $.code AS ?dept}
      Where   ← {isNotNull($.code), isNotNull($.dept)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
    INNER JOIN
    { From    ← [xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]
      Project ← {$.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← {isNotNull($.code), isNotNull($.members[?(@.age>=40)].name)}
    ON {?dept}
```

First, we change the natural left-to-right joins processing order: we embed the 2nd and 3rd atomic queries in curly brackets.

```
trans_m(bgp, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    {
        { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
          Project ← {$.dept, $.code AS ?dept}
          Where   ← {isNotNull($.code), isNotNull($.dept)} }
        INNER JOIN
        { From    ← [xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]
          Project ← {$.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
          Where   ← { isNotNull($.code), isNotNull($.members[?(@.age>=40)].name) }
        ON {?dept}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

The 2nd and 3rd atomic queries have the same *From* part, they are joined on variable `?dept`, variable `?dept` has the same projection in both queries: "`$.code AS ?dept`", and finally, that xR2RML reference "`$.code`" is declared as unique in both logical sources. Hence, this self-join can be eliminated.
We perform the self-join elimination by merging the two queries together: we merge the *Project* parts on the one hand, and the *Where* parts on the other hand. We obtain the following optimized abstract query:

```
trans_m(bgp, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← {$.dept, $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← {isNotNull($.code), isNotNull($.dept),
```

```
                isNotNull($.members[?(@.age>=40)].name)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

3.6.1.4 Self-Union Elimination

The UNION operator of the abstract query language can either be created when translating the SPARQL UNION operator, or during the translation of a triple pattern to which several triples maps are bound (in function $transTP_m$). Similar to the *Self-Join Elimination*, a union of several atomic abstract queries can be merged into a single one at the condition that they have the same *From* part, i.e. they share the same logical source.

The resulting atomic abstract query Q merges atomic abstract queries Q_1 and Q_2 this way:

- The *Project* part of Q is the union of the *Project* parts of Q_1 and Q_2.
- The *Where* part of Q must be a condition that allows either the conditions of Q_1 or the conditions of Q_2, or both. Toward that end, we introduce the new condition operator *OR*. The *Where* part of Q is defined as: $OR(Q_1.Where, Q_2.Where)$.

3.6.1.5 Constant Projection

The *Constant Projection* optimization detects cases where the only projected variables in the SPARQL query are matched with constant values in the bound triples maps. In the relational database context, it has been referred to as the *Projection Pushing* optimization [21]. Nevertheless, we find this term somehow unintuitive, and we prefer the term *Constant Projection*.

Let us consider the example query below:

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

In a first approach, all triples maps of the mapping graph are bound to the triple pattern "`?s ?p ?o`". Hence, the resulting abstract query is a union of the atomic queries derived from all the triples maps in the mapping graph. In other words, this query will materialize the whole database before it can provide an answer. Besides, this type of query is critical since it is typical of schema exploration queries.

Very frequently, the xR2RML predicate maps are constant-valued: the predicate is not computed from a database value, on the contrary it is defined statically in the mapping. This is typically the case in our running example that has only constant predicate maps defined by: "`rr:predicate ex:hasSeniorMember`" and "`rr:predicate ex:manages`". In such cases, given that the SPARQL query retrieves only DISTINCT values of the predicate variable ?p, no query needs to be run against the database at all: it is sufficient to collect the distinct constant values that variable ?p can be matched with.

More generally, this optimization checks if all variables projected in the SPARQL query are matched with constant term maps. If this is verified, the SPARQL query is rewritten such that the values of all projected variables are provided as an inline solution sequence using the SPARQL 1.1 VALUES clause. Following up on the example above, we would rewrite the query in this way:

```
SELECT DISTINCT ?p WHERE { VALUES ?p ( _:prop1 _:propr2 … )}.
```

The latter query can be evaluated straightaway, without requiring any query to the target database.

3.6.1.6 Filter Propagation

We identified another type of optimization that was not implemented in the SPARQL-to-SQL context. This optimization applies in the inner join or left outer join of two atomic queries, and seeks to narrow down one of the joined queries by propagating filter conditions from the other query.

In an inner join, if the two queries have shared variables, then *Equality* and *IsNotNull* conditions of one query on those shared variables can be propagated to the other query. In a left join, propagation can happen only from right to left query since null values must still be allowed in the right query.

Example. Assume that abstract query Q is defined as the inner join of atomic queries Q_1 and Q_2:

```
{ From    ← { … }
  Project ← {$.field1 AS ?x}
  Where   ← {equals("value", $.field1)}
}
INNER JOIN
{ From    ← { … }
  Project ← {$.field2 AS ?x}
  Where   ← {}
} ON { ?x }
```

The equals condition in Q_1's *Where* part applies to an xR2RML reference that happens to be the reference projected as ?x. In other words, the join will only select documents from Q_1 and Q_2 where variable ?x equals "value".

In the right query, Q_2, another xR2RML reference is projected as ?x: "`$.field2`". Thus, the join will only match documents of Q_2 where the reference "`$.field2`" also returns "value". Consequently, we can add (propagate) a new condition to the *Where* conditions of Q_2: `equals($.field2, "value")`. We end up with the optimized query:

```
{ From    ← { … }
  Project ← {$.field1 AS ?x}
  Where   ← {equals("value", $.field1)}
}
INNER JOIN
{ From    ← { … }
  Project ← {$.field2 AS ?x}
  Where   ← {equals("value", $.field2)}
} ON { ?x }
```

In turn, query Q_2 is more selective and the join can be computed faster.

3.7 Conclusion and Perspectives

In this chapter, we have proposed a method aimed at fostering the development of SPARQL interfaces to heterogeneous databases, as we believe this is a key to the advent of the Web of Data.

Leveraging R2RML-based SPARQL-to-SQL works, our method translates a SPARQL query into a pivot abstract query, utilizing xR2RML to describe the mapping of a target database to RDF. The method determines a set of relevant mappings for each SPARQL triple pattern; this set is reduced with respect to the join constraints and SPARQL filters.

Lastly, query optimization techniques are enforced in order to produce an efficient abstract query and facilitate the subsequent translation into the target query language. In the next chapter, we shall demonstrate the effectiveness of the method, taking the MongoDB document store as an example target database.

In the next chapter, we demonstrate the effectiveness of the method exposed here, taking the MongoDB document store as a target database. Before that, below we highlighted several limitations of our method and discuss possible future works.

SPARQL support. At this point, SPARQL named graphs are not considered in the translation. However, it would be relatively easy to extend the method that computes triple pattern bindings in order to match named graphs (FROM, FROM NAMED) with xR2RML graph maps.

Abstract Query Optimization. The management of SPARQL filters is delegated to the translation into the target query language, using the *sparqlFilter* condition. Yet, some types of filter may be dealt with at the abstract query level, in order to alleviate the work required in the translation towards the target query language. For instance, operator BIND may be turned into equivalent *equals* conditions, and *bound* into *isNotNull* conditions.

Beyond the optimizations we have implemented, further query optimization challenges shall arise in order to develop an efficient query-processing engine. For instance, what is the most efficient order to compute INNER JOINs? In this regard, the query processing engine may need to embark query plan optimization logics such as the bind join [Haas et al., 1997] to inject intermediate results into a subsequent query, and the join re-ordering based on the number of results that queries shall retrieve, very similarly to the methods applied in distributed SPARQL query engines [Schwarte et al., 2011; Görlitz & Staab, 2011; Macina et al., 2016].

Support of xR2RML mixed-syntax paths. Although mixed-syntax paths are useful to materialize RDF terms from database values with embedded formats, they can lead to undecidable situations in the SPARQL rewriting context. Let us take an example from a real life example: we translate a MongoDB database wherein JSON documents have a field `nomVernaculaire` whose value is a comma-separated string. The predicate-object map below builds object terms by selecting the first value (at index 0: `CSV(0)`) of the coma-separated string:

```
[] rr:predicateObjectMap [
     rr:predicate txrp:vernacularName;
     rr:objectMap [ xrr:reference "JSONPath($.nomVernaculaire)/CSV(0)" ] ]
```

Let us consider a SPARQL query containing the following triple pattern:

```
?vern txrp:vernacularName "Delphinus delphis".
```

The rewriting process will create an atomic abstract query wherein a condition that matches value "`Delphinus delphis`" with the mixed-syntax path expression:

```
equals(JSONPath($.nomVernaculaire)/CSV(0), "Delphinus delphis")
```

Unfortunately, there is an infinite number of CSV strings where the first value is "Delphinus delphis". Hence, in this specific context, we could rewrite the condition into something like:

```
startsWith($.nomVernaculaire, "Delphinus delphis")
```

But then, if the field content is not a CSV value but an XML snippet, we may end up with very complex expressions, for instance:

```
equals(JSONPath($.field)/XPath(//root/element[@type="some type"]), "Value")
```

More generally, it occurs that this problems amounts to deal with an arbitrary combination of JSONPath, XPath and CSV expressions. Although we may find solutions to this issue in specific situations, it seems illusory to seek a generic solution. Consequently, our SPARQL rewriting method does not deal with mixed-syntax paths. Nevertheless, in the context of custom functions written using CSVW and R2RML-F, it would be possible to define a transformation

function along with an inverse transformation function, similar to the R2RML `rr:inverseExpression` property. Thus, the inverse transformation would be delegated to a function that embeds domain knowledge.

4 Translation of an abstract query into a MongoDB query

In the previous chapter, we have exhibited a method to translate a SPARQL query into an optimized abstract query, relying on xR2RML to describe the mapping of a target database to RDF. Operators INNER JOIN, LEFT OUTER JOIN and UNION are entailed by the dependencies between graph patterns of the SPARQL query. UNION and INNER JOIN operators may also arise from the rewriting of a triple pattern: a UNION when a triple pattern tp is bound to more than one triples map, and an INNER JOIN when a triples map contains a referencing object map denoting a join query. The abstract operators relate atomic abstract queries of the form *{From, Project, Where, Limit}*. The *From* part contains the triples maps logical source. The *Where* part is calculated by matching triple pattern terms with term maps; this shall generate either *isNotNull* conditions for SPARQL variables, *equals* conditions for constant terms or *sparqlFilter* conditions that encapsulate SPARQL filters. Finally, the *Limit* part denotes an optional maximum number of results.

Translation of an Abstract Query into MongoDB queries. In this chapter, we keep on the process with the second step of our method: the translation of an abstract query into a target query, using the concrete case of MongoDB. In the context of MongoDB, xR2RML data element references are JSONPath expressions. Hence, the translation of an atomic abstract query towards MongoDB amounts to translate (i) projections of JSONPath expressions into MongoDB projection arguments, and (ii) conditions on JSONPath expressions into equivalent MongoDB query operators. Below, we illustrate the expected result using the running example.

Running Example. Previously, we showed that the translation of tp_1 entails the atomic abstract queries below:

```
{ From    ← { [xrr:query "db.staff.find({})"] }
  Project ← { $.manages.* }
  Where   ← { equals("Dunbar", $['lastname','familyname']),
              isNotNull($.manages.*) }
}
```

The projection of the "`$.manages.*`" xR2RML reference shall be turned into the MongoDB projection "`"manages": true`".

The condition `equals("Dunbar", $['lastname','familyname'])` shall be translated into a concrete MongoDB query as follows:

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

Similarly, the condition `isNotNull($.manages.*)` shall be translated into:

```
"manages": {$exists:true, $ne:null]
```

Those conditions shall augment the query of the *From* part, provided by the `xxr:query` and `rml:iterator` properties. In this regards, our example is trivial since the query in triples map <#Staff> is empty ("{}") and there is no iterator.

Finally, the atomic abstract query shall be translated in the MongoDB query:

```
db.staff.find(
       { $or: [{"lastname": {$eq: "Dunbar"}}, {"familyname": {$eq: "Dunbar"}}],
         "manages": {$exists:true, $ne:null] },
       { "manages": true }
```

)

More generally, the translation towards the MongoDB query language consists of two steps depicted in Figure 3. First, we translate abstract each query condition into the abstract representation of a MongoDB query. Several shortcomings may appear at this stage, such as untranslatable JSONPath expressions or unnecessary complexity. Thus, step 2 rewrites and optimizes this abstract representation into a union of valid, executable MongoDB queries.

Figure 3: Translation for the Abstract Query Language into the MongoDB Query Language

In this chapter, we describe the MongoDB and the abstract representation of a MongoDB query (section 4.1), and the JSONPath language (section 4.2). Then, we define two sets of the rules addressing (i) the translation of a projection from an atomic abstract query into an abstract MongoDB projection argument (section 4.4), (and (ii) the translation of a condition from an atomic abstract query into an abstract representation of a MongoDB query (section 4.4). Finally, helped by a second set of rules, we rewrite and optimize an abstract representation of a MongoDB query into a union of executable MongoDB queries (section 4.5). We also demonstrate that a condition on a JSONPath expression can always be rewritten into a union of valid MongoDB queries (rewritability property), and that the result shall return all matching documents (completeness property)

***Limitations**. In the current status of this work, we consider the translation of non-null and equality conditions into MongoDB, however we do not consider the translation of SPARQL filters.*

4.1 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 interactive interface defines a JSON-based declarative query language consisting of two mechanisms. The *find* method retrieves documents matching a set of conditions and returns a cursor to the matching documents. Optional modifiers amend the query to impose limits and sort orders. Alternatively, the *aggregate* method allows for the definition of processing pipelines: each document of a collection passes through the stages of a pipeline that creates a new collection and so on. That allows for richer aggregate computations but comes with a much higher resource consumption that entails more unpredictable performance issues. Thus, as a first approach, this work considers the *find* query method, hereafter called the *MongoDB query language*. We describe it here below, referring to the MongoDB Manual[10] 3.0 (the latest at the time of writing).

4.1.1 MongoDB *Find* Query Method

The MongoDB find query method takes two arguments:

(1) The query parameter is a JSON document that describes conditions about the documents to search for in the database. Specific query operators are denoted by a heading '$' character. Here are a few examples:

- `{"decade":{$exists:true}}`: matches all documents with a field "decade".

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

- `{"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 (at index 0) has 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 optional projection parameter specifies the fields from the matching documents to return. In the example below, collection “collection” is searched for documents about people whose age is at least 18; only the “person.name” field of each matching document is returned.

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

The MongoDB documentation provides a rich description of the query language that however 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 `$or` operator. Recently, attempts were made to clarify this semantics while underlining some limitations and ambiguities: Botoeva et al. [4] focus mainly on the *aggregate* query and ignores some of the operators we use in our translation, such as `$where`, `$elemMatch`, `$regex` and `$size`. On the other hand, Husson [11] describes the *find* query, yet some restrictions on the operator `$where` are not formalized.

Therefore, in Definition 13 we specify the subset of the query language that we consider in our approach, and we underline some limitations and ambiguities. Operator keywords are bold, square brackets ('[', ']'), curly brackets ('{', '}') and characters “:”, “,”, “/” and “.” are part of the language. Parenthesis groups "(...)", characters “*”, “+” and “|” are the conventional syntactic notation denoting occurrences and alternatives.

A comma-separated sequence of QUERY elements (in the top-level query and in the `$elemMatch` operator) implicitly denotes a logical AND on the QUERY elements. Similarly, the `$and` operator denotes a logical `AND` on an array of QUERY elements. The `$and` operator is necessary when the same field name or operator has to be specified more than once. For instance, to select documents with a field “age” between 18 and 30, the query “`{"age":{$gte:18}, "age":{$lt:30}}`“ is invalid since a valid JSON document cannot have two fields with the same name. This cases requires using the $and operator: “`$and: [{"age":{$gte:18}}, {"age":{$lt:30}}]`“.

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

The `$where` (WHERE_QUERY) operator passes a JavaScript expression or function to the query system. It provides greater flexibility than other operators. However, the JavaScript evaluation cannot take advantage of existing indexes and requires the database to process the JavaScript expression for each document. This issue can seriously hinder performances, and MongoDB strongly recommends to use `$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 *find* query. 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.

4.1.2 Semantics Ambiguities

The MongoDB query language allows ambiguous short-cut expressions to name paths in the JSON documents. For instance, query `{"p":{$eq:3}}` matches documents where p is a field with value 3, such as `{"p":3}`. Surprisingly, it also matches documents where p is an array wherein at least one element has value 3, *e.g.* `{"p":[3,4]}`. But the latter document would equally be matched by query `{"p":{$elemMatch:{$eq:3}}}`. This gets even worse with a sequence of field names, as each field name may be considered for what it is, *i.e.* exactly one field, or as a short-cut for the elements of an array field. With this logic, query `{"p.q":{$eq:3}}` matches several types of documents depending on how we interpret p and q, such as `{"p":{"q":3}}`, `{"p":[{"q":3}]}` and `{"p":[{"q":[3]}]}`.

Consequently, given the ambiguous notation of the MongoDB query language, it is hardly possible to write a MongoDB query whose semantics would be provably equivalent to a SPARQL query. Let us further elaborate on the possible impact of such ambiguities on the query translation process. The xR2RML mapping of a MongoDB database is written with a certain schema in mind: although the database is schemaless, the mapping designer expects documents to follow a certain schema; this schema is denoted in the JSONPath expressions that he/she embeds in the mapping. In our translation method, we do not use shortcut notations, instead we only use the most specific notation, *e.g.* an array element is always queried using the `$elemMatch` operator. Thus, it is likely that the rewriting we come up with will indeed retrieve only the expected documents. Yet, we cannot ignore the possibility that the database contains other documents, with a somehow different schema, that shall also be retrieved by the query due to the ambiguous notation (even though this was not in the mapping designer's intention).

Example. We consider a MongoDB collection where documents are shaped like this one:

```
{"p": { "q":3, "r":4 } }
```

An xR2RML mapping designer may use JSONPath expression `$.p.q` to get the value of q. When matching the xR2RML mapping with a SPARQL triple pattern, we may come up with an atomic abstract query where a condition is:

```
equals($.p.q, 3)
```

This condition shall be translated into the MongoDB query below:

```
db.collection.find({"p.q":{$eq: 3}})
```

This query indeed matches the document shown above, but `it may also match somehow different documents, although this was not the intention of the mapping designer, e.g.`:

```
{ "p": [
         {"q":3}, …
       ]
}
```

Such additional, non-expected documents may lead to the generation of unpredictable and possibly erroneous RDF triples.

Unfortunately, there does not seems to be a method ensuring that such flaw cannot occur. Yet, a good practice to limit the likelihood thereof is to filter irrelevant documents at the earliest stage. In the example above, we know that field “p” may be used as both a document field and an array field. Therefore, the query in the logical source should try to rule out documents where “p” is an array:

```
[] xrr:logicalSource [
  xrr:query "db.collection.find({'p': {$not:  {$type: 'array'}}})"
];
```

Definition 13: 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>]}}
```

4.1.3 Abstract Representation of a MongoDB Query

We define an abstract hierarchical representation of a MongoDB query. This representation allows for handy manipulation during the query construction and optimization phases. Definition 14 lists the clauses of this representation as well as their translation into a concrete query string, when relevant.

In the COMPARE clause definition, `<op>` stands for one of the MongoDB query compare operators: `$eq`, `$ne`, `$lte`, `$lt`, `$gte`, `$gt`, `$size` and `$regex`. Let us consider the following example abstract query:

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

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

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

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 3).

Note that this UNION clause applies to set of JSON documents retrieved from the database, whereas the UNION operator generated by function $trans_m$ applies to triples.

Definition 14: Abstract MongoDB query	
AND(<expr$_1$>, <expr$_2$>, …)	→ **$and**:[<expr$_1$>,<expr$_2$>, …]
OR(<expr$_1$>, <expr$_2$>, …)	→ **$or**:[<expr$_1$>,<expr$_2$>, …]
WHERE(<JavaScript expr>)	→ **$where**:'<JavaScript expr>'
ELEMMATCH(<exp$_1$>,<exp$_2$>, …)	→ **$elemMatch**:{<exp$_1$>,<exp$_2$>, …}
FIELD(p_1) FIELD(p_2)... FIELD(p_n)	→ "$p_1.p_2….p_n$":
SLICE(<expr>, <number>)	→ <expr>:{**$slice**: <number>}
SLICE(<expr>, <number>, <count>)	→ <expr>:{**$slice**: [<number>,<count>]}
COND(equals(v))	→ **$eq**:v
COND(isNotNull)	→ **$exists:true, $ne:null**
EXISTS(<expr>)	→ <expr>:{**$exists**: **true**}
NOT_EXISTS(<expr>)	→ <expr>:{**$exists**: **false**}
COMPARE(<expr>, <op>, <v>)	→ <expr>:{<op>: <v>}
NOT_SUPPORTED	→ ∅
CONDJS(equals(v))	→ == v
CONDJS(equals("v"))	→ == "v"
CONDJS(isNotNull)	→ != null
UNION(<query$_1$>, <query$_2$>, …)	*Same semantics as OR, but processed by the query processing engine*

4.2 The JSONPath language

JSONPath[11] 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[12] 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 15 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 FIELD_ALT and INDEX_ALT 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.

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 `$.books[-1:]`: select the last element of array "books". In the "[()]" notation, "@" refers to the parent element "books".
- `$.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

[11] http://goessner.net/articles/JsonPath/
[12] http://www.w3.org/TR/1999/REC-xpath-19991116/

select the field “name” of those elements. Unlike above, in the “[?()]” notation “@” refers to elements of the array.

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

Definition 15: JSONPath grammar

```
JSONPATH        = $(WILDCARD | FIELD_NAME | ARRAY_INDEX | DESCENDANT | FIELD_ALT |
                  INDEX_ALT | ARRAY_SLICE | FILTER | CALC_INDEX)*
WILDCARD        = .*|[*]
FIELD_NAME      = FIELD_NAME_DOT | FIELD_NAME_BRKT
FIELD_NAME_DOT  = .<name>
FIELD_NAME_BRKT = ["<name>"]
ARRAY_INDEX     = [<int>]
DESCENDANT      = ..
FIELD_ALT       = ["<name>"(,"<name>")+]
INDEX_ALT       = [<int>(,<int>)+]
ARRAY_SLICE     = [<start>:<end>:<step>] | [<start>:<end>] | [<start>:]
FILTER          = [?(<script expression>)]
CALC_INDEX      = [(<script expression>)]
```

Note: JSON field names with special characters such as ‘#’, ‘&’ or ‘/’ etc. are supported by MongoDB, but in JSONPath they require the bracket notation, e.g. [“field/#1”].

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

Restrictions on the usage of JSONPath expressions

JSONPath is an expressive language that was designed for pragmatic purposes as an analogy of the XPath language. Unfortunately, it lacks the formalization level that can be expected from standards. As a result, its definition leaves room for interpretation; this is attested by the discrepancies between implementations. For the sake of clarity, below we specify some restrictions on the possible interpretations, and we restrict the use of some operators.

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[13] 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.

[13] https://github.com/jayway/JsonPath

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 somewhat 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 11th 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".

Furthermore, 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 as "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.

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 no support JSONPath expressions using the ".." operator.

4.3 Translation of Projections

The *From* part of an atomic abstract query lists JSONPath expressions whom JSON fields should be part of the query results. Given the restrictions on JSONPath expressions defined section 4.2, this section defines the recursive function *proj* that translates a JSONPath expression into the projection argument of MongoDB *find* query.
More precisely, function *proj* builds a list of paths that shall be projected. For instance, the JSONPath expression "`$.p.q`" shall be translated into the abstract representation "`FIELD(p) FIELD(q)`" that, in turn, shall be translated into the concrete projection argument "`"p.q":true`".

Function *proj* implements a set of rules, listed in Algorithm 1, that apply when the JSONPath expression matches a certain pattern. The JSONPath expression is checked against the patterns in the order of the rules. When a match is found, the rule is applied and the search for a match stops.

Algorithm 1: Translation of a JSONPath expression into a MongoDB projection argument (function proj(JSONPath expression))

P0 **proj**($<JP>) → **proj**(<JP>)

P1 **proj**(∅) → ∅

P2 *JavaScript filter and calculated array index*
(a) **proj**(<JP:F>[?(<bool_expr>)]<JP>) → **proj**(<JP :F>) X **projJS**(<bool_expr>, **proj**(<JP>))
(b) **proj**(<JP:F>[<num_expr>)]<JP>) → **proj**(<JP :F>) X **projJS**(<num_expr>, **proj**(<JP>))

P3 *Array expressions: wildcard (a, b), array index alternative (c), array slice (d)*
(a) **proj**(.*<JP>) → **proj**(<JP>)
(b) **proj**([*]<JP>,) → **proj**(<JP>)
(c) **proj**([i, j ,...]<JP>) → **proj**(<JP>)
(d) **proj**([<slice expression>]<JP>) → **proj**(<JP>)

P4 *Field name (a, b) or field alternative (c)*
(c) **proj**(.p<JP>) → **FIELD**(p) **proj**(<JP>)
(d) **proj**(["p"]<JP>) → **FIELD**(p) **proj**(<JP>)
(e) **proj**(["p","q",...]<JP>) → **FIELD**(p) **proj**(<JP>), **FIELD**(q) **proj**(<JP>), ...

The projection argument of a MongoDB find query is fairly simple: it consists of a path followed by "true" to project the path, or "false" to not project the path. Notations "true" and "false" cannot be mixed: either named paths are explicitly projected with "true", or they are restricted, which implicitly projects all unnamed paths. In our case, we use the "true" notation to explicitly list projected paths.

A path may be a single field name *e.g.* "p", or a sequence of field names. In the latter, two semantics can apply to the path "p.q":

- If the value of "p" is an embedded document, then "p" is projected along with only the field "q" of embedded documents;
- If the value of "p" is an array, then "p" is projected along with all its elements, but only the field "q" of the elements is projected.

We now describe the way a JSONPath expression is treated by the rules in Algorithm 1: . Rule P2 is described last as it requires the understanding of other rules.

4.3.1 Rules P0 and P1

Rule P0 is the entry point of the translation process (a valid JSONPath expression starts with a "$" character): the heading "$" character is simply removed.
Conversely, rule P1 is the termination point, that is, when the JSONPath expression has been fully parsed. The rule simply returns ∅ to stop the process. Implicitly, all FIELD clauses generated beforehand are to be translated into a projection argument.

4.3.2 Rule P3: Array Notations

Rule P3 deals with different types of array notations: wildcard, array index alternative and array slice. The MongoDB projection argument cannot select elements from an array: they are all projected, but we can specify which field of

the elements are projected. Thus, whether all elements are selected with the wildcard, or only a subset with the alternative or slice, the translation simply goes on by parsing anything that comes after the array notation. Examples:

- JSONPath expression "$.p.*.q" is translated into the argument "`"p.q":true`" that projects the field "q" of all elements of "p".
  Similarly, in JSONPath expressions "`$.p[1,3,5].q`" and "`$.p[2:4].q`", the alternative and slice notations are ignored. Both are translated into the same projection argument "`"p.q":true`".

4.3.3 Rule P4: Field Names

Rule P4 deals with regular field names, and simply adds a FIELD clause each time a new field name is parsed.
Example: JSONPath expression "`$.p.q.r`" shall be translated into the abstract representation "`FIELD(p) FIELD(q) FIELD(r)`", which shall in be rewritten into the argument "`"p.q.r": true`".

4.3.4 Rule P2: JavaScript Filter and Calculated Array Index

JavaScript expressions, whether in a Boolean filter or a calculated index, are managed in a somewhat specific way. Let us take an example: JSONPath expression "`$.p[?(@.q && @.r)].s`" denotes documents with an array field "p", whose elements have a field "q" and a field "r", and we select the value of field "r", *e.g.* `{"p": ["q":1, "r":2, "s":3]}`.

To project all appropriate fields, we must produce the projection argument:

"`"p.q":true, "p.r":true, "p.s":true`"".

In other words, we have to make a product of (i) the field named before the JavaScript filter, (ii) the fields named in the JavaScript filter, and (iii) the fields named after the JavaScript filter.

To that end, we first define function *projJS*(<JS expr>, <FIELD clause>) that returns a list of FIELD clauses, one for each field named in the JavaScript expression, and the additional FIELD clause provided as a second argument. Then, we denoted by the "X" the product between the FIELD clauses produced by either the *proj* or *projJS* functions:

proj(<JP:F>[?(<bool_expr>)]<JP>) → **proj**(<JP :F>) X **projJS**(<bool_expr>, **proj**(<JP>))

Applied to the example above, we get:

proj(`$.p[[?(@.q && @.r)].s`)

→ **proj**(p) X **projJS**(`@.q && @.r`, **proj**(s))

→ FIELD(p) X (FIELD(q), FIELD(r), FIELD(s))

→ FIELD(p) FIELD(`q`), FIELD(p) FIELD(`r`), FIELD(p) FIELD(`s`)

This indeed generates the three expected paths: p.q, p.r and p.s.
The management of JavaScript calculated array indexes follows the same procedure.

4.4 Query translation rules

Given the subset of the MongoDB query language that we consider in section 4.1, the JSONPath language and the restrictions mentioned in section 4.2, 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.

Running Example. Before getting into the details, let us illustrate the approach using the running example. As already seen, the translation of triple pattern tp_1 entails two atomic abstract queries (see section 3.5), among which the child query contains two conditions:

```
isNotNull($.manages.*),
equals("Dunbar", $['lastname','familyname'])
```

Let us consider condition `isNotNull($.manages.*)`. It amounts to evaluating `trans($.manages.*, isNotNull)` that goes through the following steps:

- Rule R0 first matches, returning `trans(.manages.*, isNotNull)`.
- Then, rule R8 matches, it returns `FIELD(manages) trans(.*, isNotNull)`.
- Lastly rules R7 and R1 translate `trans(.*, isNotNull)` into `ELEMMATCH(COND(isNotNull))`.

This comes up with the abstract MongoDB query:

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

Applying Definition 14 to the abstract MongoDB query entails the final concrete query:

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

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

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

that is translated into the concrete query:

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

Conventions. 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, , without the heading '$' character, *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.

We also 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"`.

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

R0 **trans**($, <cond>) → **NOT_SUPPORTED**
trans($<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: last n elements (a), first n elements (b),), from index m to n-1 or from mth to last to nth to last (c)*
(a) **trans**(<JP:F>[-<n>:]<JP>, <cond>) → **trans**(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), **-**<n>)
(b) **trans**(<JP:F>[:<n>]<JP>, <cond>) → **trans**(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), <n>)
trans(<JP:F>[0:<n>]<JP>, <cond>) → **trans**(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), <n>)
(c) **trans**(<JP:F>[<m>:<n>]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), <m>, (<n>-<m>))
trans(<JP:F>[-<m>:-<n>]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), -<m>, (<m>-<n>))

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 JavaScript filter expression
(b) **trans**(<JP:F >[(<num_expr>)], <cond>) →
AND(**EXISTS**(<JP:F >),
WHERE('this<JP:F>[**replaceAt**("this<JP:F>", <num_expr>)] **CONDJS**(<cond>')))
(c) **trans**(<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*
(e) **trans**(.*<JP>, <cond>) → **ELEMMATCH**(**trans**(<JP>, <cond>))
(f) **trans**([*]<JP>, <cond>) → **ELEMMATCH**(**trans**(<JP>, <cond>))

R8 *Heading field name or array index*
(f) **trans**(.p<JP>, <cond>) → **FIELD**(p) **trans**(<JP>, <cond>)
(g) **trans**(["p"]<JP>, <cond>) → **FIELD**(p) **trans**(<JP>, <cond>)
(h) **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**

4.4.1 Rule R0

Rule R0 is the entry point of the translation process since a valid JSONPath expression starts with a "$" character.

4.4.2 Rue 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. If value is of a string datatype, it is surrounded with quotes.

4.4.3 Rule R2

A field alternative or array index alternative is translated into an OR clause, corresponding to the MongoDB `$or` operator. As underlined in section 4.1, the `$or` operator cannot be used as a condition on a field; for instance, the following query is invalid: `"p.q": {$or: [{$eq: 10}, {$eq: 10}]}`.
Instead, it has to be either at the top-level query or nested in an `$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(FIELD(p) FIELD(p) COND(equals, 10), FIELD(p) FIELD(r) COND(equals, 10))
```

that, in turn, shall be 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.

4.4.4 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}}]}}
```

4.4.5 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 4.4.11) 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.

4.4.6 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 `{$slice: <count>}` or `{$slice: [<start>, <count>]}`, `<count>` may be negative in the first notation only, and in the second notation `<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:

JSONPath	**Semantics**	**MongoDB query language**
array[0:n], array[:n]	First *n* elements: from index *0* to index *n-1*	"array" : {$slice: n}
array[-m:]	Last *m* elements	"array" : {$slice: -m}
array[m:]	From index *m* until the last element	n/a
array[m:n]	From index *m* to index n-1	"array" : {$slice: [m, (n-m)]}
array[-m:-n]	From index m^{th} to last to $n\text{-}1^{th}$ to last	"array" : {$slice: [-m, (m-n)]}

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.

4.4.7 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.

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'"}]
```

However, we have mentioned earlier that the `$where` operator can be used only as a member of the top-level query. Several situations may occur:

- If a calculated array index is preceded by either a wildcard or a filter, its translation shall be embedded into an `$elemMatch` created by rules R7 or R4 respectively. Rule R6(a) makes this case impossible by returning NOT_SUPPORTED.
- Yet, rule R6 (b and c) produces a `$where` operator nested in an `$and` operator. We show in section 4.5 that we can rewrite a query containing a `$where` nested in a combination of `$and` and `$or` operators into a union of valid MongoDB queries in which `$where` operators show only in the top-level query.

If the calculated array index is followed by a non-empty 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)`.

Yet, in both situations, we could figure out a way to achieve the translation through the definition of a JavaScript function that would parse the array. At this stage however, we choose not to go down that road, and we further discuss this choice in section 6.4. Therefore, in rule R6(c) we restrict terms that follow a calculated array index to a sequence of field names or array indexes, denoted by <JP2:F>.

4.4.8 Rule R7

As mentioned in section 4.2, the use of the wildcard within JSONPath expressions is restricted to the context of array fields, but it cannot apply to document fields. Hence, rule R7 simply translates a heading wildcard into an `$elemMatch` operator.

4.4.9 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}`.

4.4.10 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:s].

- JSONPath expressions entailing that the root document is an array and not a document, such as `$.*`, `$[1,2,…]` , `$[?(…)]` and `$[(…)]`.

4.4.11 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 2, 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: `===` → `$eq`, `==` → `$eq`, `!=` → `$ne`, `<=` → `$lte`, `>=` → `$gte`, `<` → `$lt`, `>` → `$gt`, `=~` → `$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 2: Translation of a JavaScript filter into a MongoDB query (function transJS)

J0 **transJS**(<JS_expr1> && <JS_expr2>) → **AND**(**transJS**(<JS_expr1>), **transJS**(<JS_expr2>))

J1 **transJS**(<JS_expr1> || <JS_expr2>) → **OR**(**transJS**(<JS_expr1>), **transJS**(<JS_expr2>))

J2 **transJS**(@<JS_expr1> <op> @<JS_expr2>) → **NOT_SUPPORTED**
where <op> stands for one of {==, ===, !=, !==, <=, <, >=, >, %}

J3 **transJS**(@<JSpath>) → **EXISTS**(**dotNotation**(<JSpath>))

J4 **transJS**(!@<JSpath>) → **NOT_EXISTS**(**dotNotation**(<JSpath>))

J5 (a) **transJS**(@<JSpath>.length == <i>) → **COMPARE**(**dotNotation**(<JSpath>), $size, <i>)
(b) **transJS**(@<JSpath>.length <op> <i>) → **NOT_SUPPORTED**
where <op> stands for one of {!=, <=, <, >=, >, %}

J6 **transJS**(@<JSpath> <op> <v>) → **COMPARE**(**dotNotation**(<JSpath>), **transJsOp**(<op>), <v>)

J7 **transJS**(<JS_expr>) → **NOT_SUPPORTED**

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.

4.5 Query optimization and translation to a concrete MongoDB query

Functions *trans()* and *transJS()*, defined in section 4.4, 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 4.5.1 and 4.5.2 respectively. Lastly, function *rewrite* (section 4.5.3) iteratively uses those rules to perform all possible rewritings and ultimately generate either one concrete MongoDB query or a union of concrete MongoDB queries.

4.5.1 Query optimization

Rules O1 to O5, in Algorithm 3, cope with issues (i) and (iii). Each rule applies to a query Q when Q matches the pattern in the head of the rule.

Algorithm 3: Optimization of an abstract MongoDB query
The "→" arrow means "is rewritten as".

O1 Flatten nested AND, OR and UNION clauses:

$\mathbf{AND}(C_1,\dots C_n, \mathbf{AND}(D_1,\dots D_m,)) \rightarrow \mathbf{AND}(C_1,\dots C_n, D_1,\dots D_m)$

$\mathbf{OR}(C_1,\dots C_n, \mathbf{OR}(D_1,\dots D_m,)) \rightarrow \mathbf{OR}(C_1,\dots C_n, D_1,\dots D_m)$

$\mathbf{UNION}(C_1,\dots C_n, \mathbf{UNION}(D_1,\dots D_m,)) \rightarrow \mathbf{UNION}(C_1,\dots C_n, D_1,\dots D_m)$

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

O3 Group WHERE clauses:
OR(..., **WHERE**("W1"), **WHERE**("W2")) → **OR**(..., **WHERE**("(W1) || (W2)")).
AND (..., **WHERE**("W1"), **WHERE**("W2")) → **AND**(..., **WHERE**("(W1) && (W2)")).
UNION(..., **WHERE**("W1"), **WHERE**("W2")) → **UNION**(..., **WHERE**("(W1) || (W2)")).

O4 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.

O5 Remove NOT_SUPPORTED clauses:
(a) **AND**(C_1,… C_n, **NOT_SUPPORTED**) → **AND**(C_1,… C_n)
(b) **ELEMMATCH**(C_1,… C_n, **NOT_SUPPORTED**) → **ELEMMATCH**(C_1,… C_n)
(c) **OR**(C_1,… C_n, **NOT_SUPPORTED**) → **NOT_SUPPORTED**
(d) **UNION(**C_1,… C_n, **NOT_SUPPORTED**) → **NOT_SUPPORTED**
(e) **FIELD**(…)… **FIELD**(…) **NOT_SUPPORTED** → **NOT_SUPPORTED**

Rules O1 to O4 address issue (iii) by flattening nested OR, AND and UNION clauses, and merging sibling WHERE clauses. Rule O5 addresses issue (i) by removing NOT_SUPPORTED clauses while still making sure that the query returns all the correct answers. Let C_1,… C_n be any clauses and N be a NOT_SUPPORTED clause. We denote by $C_1 \wedge \ldots \wedge C_n$ the set of documents matching all conditions C_1 to C_n.

- O5(a): If a NOT_SUPPORTED clause occurs in an AND clause, it is simply removed. Since "$C_1 \wedge \ldots \wedge C_n \wedge N$" is more specific than "$C_1 \wedge \ldots \wedge C_n$" ($C_1 \wedge \ldots \wedge C_n \subseteq C_1 \wedge \ldots \wedge C_n \wedge N$), by simply removing the N component we widen the whole condition. Consequently, all matching documents shall be returned, but non-matching documents may be returned too, that shall be ruled out later on.
- O5(b): A logical AND implicitly applies to members of an ELEMMATCH clause. Therefore, removing the NOT_SUPPORTED has the same effect as in O5(a).
- O5(c) and O5(d): Contrary to the AND and ELEMATCH cases, we cannot simply remove the NOT_SUPPORTED clause from an OR or UNION clause as the query would only return a subset of the matching documents ($C_1 \vee \ldots \vee C_n \subseteq C_1 \vee \ldots \vee C_n \vee N$). Instead, the OR or UNION clause is replaced with a NOT_SUPPORTED clause. Consequently, the issue is raised up to the parent clause, and it shall be managed at the next rewriting iteration. Iteratively, we raise up the NOT_SUPPORTED clause until it is eventually removed (cases AND or ELEMMATCH above), or it ends up in the top-level query. The latter is the worst case in which the query is no longer selective at and shall retrieve all the documents.
- O5(e): Similarly to O5(c) and O5(d), a sequence of fields followed by a NOT_SUPPORTED clause is replaced with a NOT_SUPPORTED clause to raise up the issue to the parent clause.

Example. We illustrate Algorithm 3 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) ELEMMATCH( trans(.name, equals("john")),
          transJS([?(@.level=="beginner" && @.score>=3 && @.isPlayer<>@.isGoal)])) =

R8,R1 FIELD(teams.0) ELEMMATCH( FIELD(name) COND(equals, "john"),
         transJS([?(@.level=="beginner" && @.score>=3 && @.isPlayer<>@.isGoal)])) =

J0,J6 FIELD(teams.0) ELEMMATCH(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}}}
```

4.5.2 Pull up WHERE clauses

By construction, a WHERE clause cannot be nested in an ELEMMATCH clause (rule R6). In addition, Algorithm 3 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 thereby 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, if 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 "→" stands for "is rewritten to".

- OR(C, W) → UNION(C, W): OR substituted with UNION, W is pulled up in the top-level query.
- AND(C, W) → (C,W): top-level AND replaced with its members, W is pulled up in the top-level query.
- OR(C, AND(D, W)) → 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)) → UNION(AND(C, D), AND(C, W)): this is a straightforward application of the theorem: C ∧ (D ∨ W) ⇔ (C ∧ D) ∨ (C ∧ 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 3 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)) → UNION(AND(C, D), AND(C, W)).
- OR(C, UNION(D, W)) → UNION(C, D, W).

Note that the case of nested UNION clauses is dealt with by rule O1 in Algorithm 3.

Algorithm 3: Pull-up of WHERE clauses to the top-level query

The "→" arrow means "is rewritten as".

W1 **OR**(C_1,...C_n, W) → **UNION**(**OR**(C_1,...C_n), W)

W2 **OR**(C_1,...C_n, **AND**(D_1,...D_m, W)) → **UNION**(**OR**(C_1,...C_n), **AND**(D_1,...D_m, W))
Proof: $C_1 \vee ... \vee C_n \vee (D_1 \wedge ... \wedge D_m \wedge W) \Leftrightarrow (C_1 \vee ... \vee C_n)$ □ $(D_1 \wedge ... \wedge D_m \wedge W)$
Therefore, $eval(C_1 \vee ... \vee C_n \vee (D_1 \wedge ... \wedge D_m \wedge W)) = eval(C_1 \vee ... \vee C_n) \cup eval(D_1 \wedge ... \wedge D_m \wedge W)$.

W3 **AND**(C_1,...C_n, W) → (C_1,...C_n, W), *iif the AND clause is a top-level query object or under a UNION clause.*

W4 **AND**(C_1,...C_n, **OR**(D_1,...D_m, W)) → **UNION**(**AND**(C_1,...C_n, **OR**(D_1,...D_m)), **AND**(C_1,...C_n, W))
Proof: $C_1 \wedge ... \wedge C_n \wedge (D_1 \vee ... \vee D_m \vee W) \Leftrightarrow (C_1 \wedge ... \wedge C_n) \wedge ((D_1 \vee ... \vee D_m) \vee W)$
$\Leftrightarrow ((C_1 \wedge ... \wedge C_n) \wedge (D_1 \vee ... \vee D_m))$ □ $((C_1 \wedge ... \wedge C_n) \wedge W)$
Therefore, $eval(C_1 \wedge ... \wedge C_n \wedge (D_1 \vee ... \vee D_m \vee W)) = eval(C_1 \wedge ... \wedge C_n \wedge (D_1 \vee ... \vee D_m)) \cup eval(C_1 \wedge ... \wedge C_n \wedge W)$

W5 **AND**(C_1,...C_n, **UNION**(D_1,...D_m)) → **UNION**(**AND**(C_1,...C_n, D_1),... **AND**(C_1,...C_n, D_m))
Proof: $C_1 \wedge ... \wedge C_n \wedge (D_1 \vee ... \vee D_m) \Leftrightarrow (C_1 \wedge ... \wedge C_n) \wedge (D_1 \vee ... \vee D_m)$
$\Leftrightarrow (C_1 \wedge ... \wedge C_n \wedge D_1) \vee ... \vee (C_1 \wedge ... \wedge C_n \wedge D_m)$
Therefore, $eval(C_1 \wedge ... \wedge C_n \wedge (D_1 \vee ... \vee D_m)) = eval(C_1 \wedge ... \wedge C_n \wedge D_1) \cup ... \cup eval(C_1 \wedge ... \wedge C_n \wedge D_m)$

W6 **OR(**C_1,...C_n, **UNION**(D_1,...D_m)) → **UNION**(**OR**(C_1,...C_n), D_1, ...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"))')
     )
```

4.5.3 Rewritability and Completeness Properties

Finally, we define in Algorithm 4 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 4: 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 14.
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 that we demonstrate hereafter:

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

Rewritability: It is always possible to rewrite Q into a query Q' = UNION(Q'_1,… Q'_m) such that ∀i ∈ [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 completeness property is a result of how rule O5 deals with NOT_SUPPORTED clauses. For the sake of readability, we copy the demonstration already provided when describing O5.

Let C_1,… C_n be any clauses and N be a NOT_SUPPORTED clause. We denote by $C_1 \wedge \ldots \wedge C_n$ the set of documents matching all conditions C_1 to C_n.

- O5(a): If a NOT_SUPPORTED clause occurs in an AND clause, it is simply removed. Since "$C_1 \wedge \ldots \wedge C_n \wedge N$" is more specific than "$C_1 \wedge \ldots \wedge C_n$" ($C_1 \wedge \ldots \wedge C_n \subseteq C_1 \wedge \ldots \wedge C_n \wedge N$), by simply removing the N component we widen the whole condition. Consequently, all matching documents shall be returned, but non-matching documents may be returned too, that shall be ruled out later on.
- O5(b): A logical AND implicitly applies to members of an ELEMMATCH clause. Therefore, removing the NOT_SUPPORTED has the same effect as in O5(a).
- O5(c) and O5(d): Contrary to the AND and ELEMATCH cases, we cannot simply remove the NOT_SUPPORTED clause from an OR or UNION clause as the query would only return a subset of the matching documents ($C_1 \vee \ldots \vee C_n \subseteq C_1 \vee \ldots \vee C_n \vee N$). Instead, the OR or UNION clause is replaced with a NOT_SUPPORTED clause. Consequently, the issue is raised up to the parent clause, and it shall be managed at the next iteration of function *rewrite*. Iteratively, we raise up the NOT_SUPPORTED clause until it is eventually removed (cases AND or ELEMMATCH above), or it ends up in the top-level query. The latter is the worst case in which the query is no longer selective at and shall retrieve all the documents.
- O5(e): Similarly to O5(c) and O5(d), a sequence of fields followed by a NOT_SUPPORTED clause is replaced with a NOT_SUPPORTED clause to raise up the issue to the parent clause.

Rewritability, NOT_SUPPORTED clauses. By construction, function *trans* may generate a NOT_SUPPORTED clause in the top-level query or in the following patterns: $\mathrm{AND}(\ldots,N,\ldots)$, $\mathrm{ELEMMATCH}(\ldots,N,\ldots)$, $\mathrm{OR}(\ldots,N,\ldots)$, $\mathrm{UNION}(\ldots,N,\ldots)$, $\mathrm{FIELD}(\ldots)\ldots\mathrm{FIELD}(\ldots)\ N$, where "N" stands for a NOT_SUPPORTED clause. If it is in the top-level query, then Definition 14 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:

$\mathrm{AND}(\ldots,N,\ldots) \rightarrow \mathrm{AND}(\ldots)$
$\mathrm{ELEMMATCH}(\ldots,N,\ldots) \rightarrow \mathrm{ELEMMATCH}(\ldots)$
$\mathrm{OR}(\ldots,N,\ldots) \rightarrow N$
$\mathrm{UNION}(\ldots,N,\ldots) \rightarrow N$
$\mathrm{FIELD}(\ldots)\ldots\mathrm{FIELD}(\ldots)\ N \rightarrow N$

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:

$\mathrm{AND}(\ldots,\mathrm{OR}(\ldots,N,\ldots),\ldots) \rightarrow \mathrm{AND}(\ldots,N,\ldots) \rightarrow \mathrm{AND}(\ldots)$
$\mathrm{AND}(\ldots,\mathrm{UNION}(\ldots,N,\ldots),\ldots) \rightarrow \mathrm{AND}(\ldots,N,\ldots) \rightarrow \mathrm{AND}(\ldots)$

AND(…,FIELD(…)…FIELD(…) N,…) → AND(…,N,…) → AND(…)
ELEMMATCH(…,OR(…,N,…),…) → ELEMMATCH(…,N,…) → ELEMMATCH(…)
ELEMMATCH(…,UNION(…,N,…),…) → ELEMMATCH(…,N,…) → ELEMMATCH(…)
ELEMMATCH(…,FIELD(…)…FIELD(…) N,…) → ELEMMATCH(…,N,…) → ELEMMATCH(…)

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 ELEMMATCH clause. Furthermore, rules W1 to W6 may create UNION clauses, and Algorithm 3 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:

OR(…,W,…)
OR(…,AND(…,W,…),…)
OR(…,UNION(…,W,…),…)
AND(…,W,…)
AND(…,OR(…,W,…),…)
AND(…,UNION(…,W,…),…)
UNION(…,W,…)
UNION(…,AND(…,W,…),…)
UNION(…,OR(…,W,…),…)

where “W” stands for a WHERE clause.

To prove Theorem 1, we must show that Algorithm 4 can rewrite a query so that the depth of a WHERE clause be 0. Toward that end, we define function *depth* that measures the depth of a MongoDB query consisting of AND, OR, UNION and WHERE clauses. First, we postulate:

depth(C_1/…/C_n) = depth(C_1) + … + depth(C_n)
where C_1/…/C_n are clauses nested within one another
depth(UNION) = 0
depth(AND) = 1
depth(OR) = 1

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/…/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 *$depth_w(Q)$* as the depth of a clause WHERE within a query Q:

$depth_w(C_1, … C_n, W) = 0$ *(case of a top-level query)*
$depth_w(C_1(… C_2(… C_n(… W)))) = depth(C_1/C_2/…/C_n)$

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.

OR(…,W,…)	*Rule W1:* Q: **OR**(C_1,…C_n, W) → Q’: **UNION**(OR(C_1,…C_n), W)

	$depth_w(Q) = 1$ $depth_w(Q') = 0$
OR(…,**AND**(…,W,…),…)	*Rule W2:* Q: **OR**(C_1,…C_n, **AND**(D_1,…D_m, W)) → Q': **UNION**(OR(C_1,…C_n), **AND**(D_1,…D_m, W)) $depth_w(Q) = 2$ $depth_w(Q') = 1$
AND(…,W,…)	*Rule W3 (iif the AND clause is a top-level query object or under a UNION clause):* Q: **AND**(C_1,…C_n, W) → Q': (C_1,…C_n, W) $depth_w(Q) = 1$ $depth_w(Q') = 0$
AND(…,**OR**(…,W,…),…)	*Rule W4:* Q: **AND**(C_1,…C_n, **OR**(D_1,…D_m, W)) → Q': **UNION**(**AND**(C_1,…C_n, **OR**(D_1,…D_m)), **AND**(C_1,…C_n, W)) $depth_w(Q) = 2$ $depth_w(Q') = 1$
AND(…,**UNION**(…,W,…),…)	*Rule W5:* Q: **AND**(C_1,…C_n, **UNION**(D_1,…D_m, W)) → Q': **UNION**((C_1,…C_n, D1),… (C_1,…C_n, D_m), (C_1,…C_n, W)) $depth_w(Q) = 1$ $depth_w(Q') = 1$
OR(…,**UNION**(…,W,…),…)	*Rule W6:* Q: **OR**(C_1,…C_n, **UNION**(D_1,…D_m, W) → Q': **UNION**(**OR**(C_1,…C_n), D_1, …D_m, W)) $depth_w(Q) = 1$ $depth_w(Q') = 0$
UNION(…,W,…)	The WHERE clause is a top-level query, the query is valid as is and no rewriting is needed.
UNION(…,**AND**(…,W,…),…)	*Rule W3 (iif the AND clause is a top-level query object or under a UNION clause):* Q: **UNION**(C_1,…C_n, AND(D_1,…D_m, W)) → Q': **UNION**(C_1,…C_n, (D_1,…D_m, W)) $depth_w(Q) = 1$ $depth_w(Q') = 0$
UNION(…,**OR**(…,W,…),…)	*We first apply rule W1 then rule O1 to merge nested UNIONs:* Q: **UNION**(C_1,…C_n**, OR**(D_1,…D_m, W)) → **UNION**(C_1,…C_n, **UNION**(OR(D_1,…D_m), W)) → Q': **UNION**(C_1,…C_n, OR(D_1,…D_m), W) $depth_w(Q) = 1$ $depth_w(Q') = 0$

In all the patterns above, the depth of the WHERE is always decreased by one using rules W1 to W6 and optionally rule O1, except for one where the depth is constant: in pattern **AND**(…,**UNION**(…,W,…),…), the resulting Q' query is:

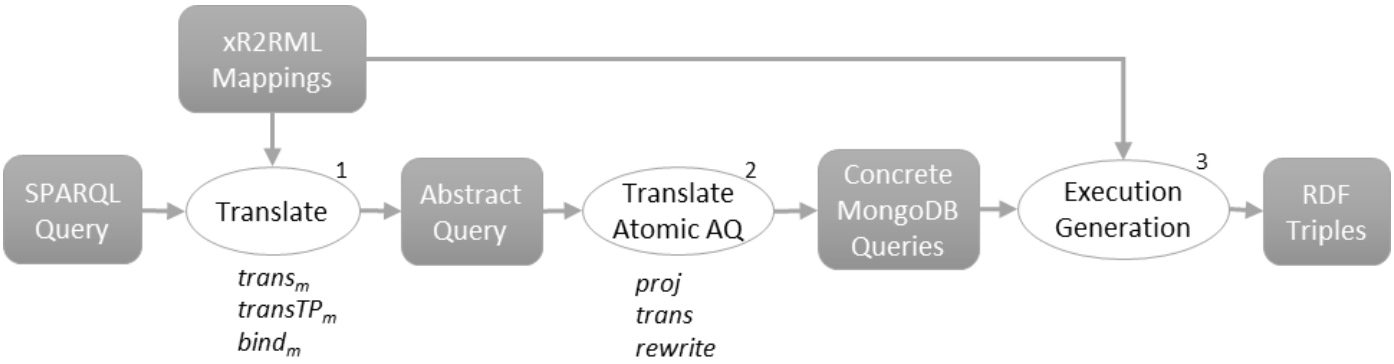
UNION(**AND**(C_1,...C_n, D_1),... **AND**(C_1,...C_n, D_m), **AND**(C_1,...C_n, W))

Thus, $depth_w(Q) = 1$ and $depth_w(Q') = 1$. Nevertheless, a UNION is either a top-level query, and in that case the inner ANDs are replaced by their members, or the UNION is nested within some other query and it will eventually be raised up to the top-level query by rulesW5 or W6. Hence, in all cases, we shall be able to come up with a query Q'' where $depth_w(Q'') = 0$.

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.

5 Overall query translation and evaluation process

Let us sum up the translation process, depicted by the picture below. In step 1, function $trans_m$ (section 3) translates a SPARQL graph pattern into an abstract query under a set of xR2RML mappings denoted by *m.* It leverages function $transTP_m$ that translates a triple pattern tp into an abstract query under the set of mappings bound to *tp* by function $bind_m$. The resulting abstract query contains atomic abstract queries of the form *{From, Project, Where, Limit}*. The *Where* part consists of *isNotNull*, *equals* and *sparqlFilter* conditions. In step 2, function *proj* (section 4.4) translates each projected JSONPath expression into a MongoDB projection argument, function *trans* (section 4.4) translates each *isNotNull* and *equals* condition on a JSONPath expression into an abstract representation of a MongoDB query, and function *rewrite* (section 4.5) optimizes and rewrites this abstract representation into a union of concrete MongoDB queries.



The last step that we have not mentioned yet is the execution of MongoDB queries and translation of results into actual RDF triples, depicted by step 3. To propose a complete approach, Algorithm 3 describes how the different steps are orchestrated, from the SPARQL query rewriting until the final generation of the RDF triples that match the query graph pattern.

- Lines 2 and 3 regard the translation of the SPARQL graph pattern into an optimized abstract query (step 1).
- The for-loop from line 5 to line 25 deals with the individual atomic abstract queries. For each atomic query, the JSONPath expressions of the *Project* part are translated into an projection argument (lines 7-10), and the conditions of the *Where* part are translated into an abstract MongoDB query (lines 12-17) which is in turn optimized and rewritten into a union of concrete MongoDB queries (line 18). Each concrete MongoDB query is then executed against the database and its results stored in set R_i (lines 20-24).
- On each R_i, the triples maps bound to Q_i are applied (line 27): this generates the RDF terms that the query processing engine can now use to compute the abstract query operators (line 28). This produces a primary result RDF graph.

Algorithm 4: Overall SPARQL-to-MongoDB query processing

```
1  Function process(sparqlGraphPattern):
2  abstractQuery ← trans_m(sparqlGraphPattern)
3  abstractQuery ← optimize abstractQuery // filter optimization and pushing,
4                  self-join and self-union elimination, constant projection, filter propagation
5  for each atomic abstract query Q_i = {From, Project, Where, Limit} ∈ abstractQuery do
6     // Translate projections of the Project part into a MongoDB projection argument
7     P_i ← ∅
8     for each projection ∈ Project do
9        P_i ← P_i, proj(projection)
10    end for
11    // Translate conditions of the Where part into an abstract MongoDB query
12    Q ← true
13    for each cond ∈ Where | cond is a isNotNull or equals condition do
14       <JSONPath>, <condition> ← cond
15       Q ← AND(Q, trans(<JSONPath>, <condition>))
16       end if
17    end for
18    Q_i' ← rewrite(Q)  // Q_i' is either a concrete query or a union of concrete queries
19    // Compute R_i, the set of documents matching Q_i'
20    if Q_i' is a concrete MongoDB query
21       R_i ← execute(Q_i', P_i, limit)
22    else // Q_i' is UNION(q_1, ..., q_n)
23       R_i ← execute(q_1, P_i, limit) ∪ ... ∪ execute(q_n, P_i, limit)
24    end if
25 end for
26 // Generate the RDF triples corresponding to XXX
27 Apply triples map bound to each Q_i to all documents of R_i
28 primaryGraph ← compute UNION, INNER JOIN, LEFT OUTER JOIN and LIMIT operators
29 // Late SPARQL query evaluation
30 resultGraph ← evaluate sparqlGraphPattern against primaryGraph
31 return resultGraph
```

At this point, we cannot guarantee that the RDF graph we have produced contains only RDF triples that match the SPARQL graph pattern. Let us remind why:

(i) The ambiguous semantics of the MongoDB query language (underlined in section 4.1) entails that a MongoDB query cannot be guaranteed to have the same semantics as the triple pattern it stands for. All documents matching the SPARQL query are returned (the certain answers), but in addition, non-matching documents may be returned too.

(ii) Yet, we proved in section 4.5.3 that the rewriting shall retrieve all matching documents. But again, non-matching documents may be retrieved too.

(iii) Lastly, at this stage, our method does not rewrite SPARQL filters, embedded in atomic abstract queries using *sparqlFilter* conditions, into appropriate MongoDB operators.

To work out those issues, introduces a final step called the *late SPARQL query evaluation*: the initial SPARQL query is evaluated against the primary RDF graph (line 30), which rules out all the non-matching triples that were generated due to issues above.

6 Conclusion, Discussion and Perspectives

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.

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 be 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.

6.1 Query optimization

Function $trans_m$ translates a SPARQL query into an abstract query containing INNER JOIN, LEFT OUTER JOIN, FILTER and UNION operators. With SQL or XQuery whose expressiveness is similar to that of SPARQL, the abstract query can be translated into a single SQL query, as shown in various approaches [5,18,8,20,16,15]. Conversely, 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 multiple independent queries, thereby delegating several steps to the query-processing engine. This is illustrated in Algorithm 3 that processes INNER JOIN, LEFT OUTER JOIN, FILTER and UNION operators between sets of JSON documents.

Evaluating concrete queries independently of each other can be the cause of performance issues. The problem of efficiently evaluating the abstract query amounts to a classical query plan optimization problem. Future works shall include the study of methods such as the bind join [10] 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 [17,9].

6.2 Limitations

At this point, several limitations must be highlighted, that we may consider in future works:

SPARQL filters are not tackled in the translation of an abstract query into the MongoDB query language. We plan to address this in the future, although it is likely that the support shall be 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. JavaScript functions can help in this matter, although we have to consider this option with reluctance due to the performance issues it entails (discussed below). Again, some filtering tasks shall be delegated to the query-processing engine to bridge the gap between SPARQL and MongoDB.

6.3 MongoDB find vs. aggregate queries

In a recent work, Botoeva et al. have proposed a generalization of the OBDA principles to support MongoDB [Botoeva et al., 2016b]. Their approach has similarities and discrepancies with ours, that we outline below.

Botoeva et al. derive a set of type constraints (literal, object, array) from the mapping assertions, called the MongoDB database *schema*. Then, a relational view over the database is defined with respect to that schema, notably by flattening array fields. A SPARQL query is rewritten into relational algebra (RA) query, and RA expressions over the relational view are translated into MongoDB *aggregate* queries. Similarly, we translate a SPARQL query into an abstract representation (that is not the relational algebra) under xR2RML mappings. To deal with the tree form of JSON documents we use JSONPath expressions. This avoids the definition of a relational view over the database, but this also comes with additional complexity in the translation process, as translating conditions on JSONPath expressions is not straightforward.

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

Finally, Botoeva et al. produce MongoDB *aggregate* queries: the major advantage is that a SPARQL 1.0 query is translated into a single semantics-preserving target query, thus delegating the whole processing to MongoDB. Yet, in practice, *aggregate* queries model processing pipelines. In some cases, they may perform extremely poorly in terms of memory and CPU consumption. This issue has been identified by the authors. Hence, they suggest that it might be necessary to decompose RA queries into smaller sub-queries, and finally perform the remaining steps in the query-processing engine. Our approach produces *find* queries that are indeed less expressive, but whose performance is easier to anticipate. This puts a higher burden on the query-processing engine (joins, unions and filtering), but having the job done outside of the database engine allows to leverage extensive works about query plan optimizations [Haas et al., 1997; Schwarte et al., 2011; Görlitz & Staab, 2011; Macina et al., 2016], whereas this is not possible when the database performs an aggregate query in a black-box manner.

In the future, it would be interesting to see whether we could characterize mappings with respect to the type of query that shall perform best: single vs. multiple separate queries, find vs. aggregate, and figure out a balance between the two approaches.

6.4 Dealing with the MongoDB $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, 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.

7 Appendix A

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

7.1 Functions genProjection and genProjectionParent

We first describe function *getReferences*, a utility function used in subsequent functions.

Algorithm 5: 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 ← termMap.constant
  end case
  return termVal
```

Algorithm 6: Generate the list of xR2RML references that must be projected in the abstract 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 | getReferences(TM.subjectMap) AS tp.sub
  end if

  if type(tp.pred) is VARIABLE then
    refList ← refList | getReferences(TM.predicateObjectMap.predicateMap) AS tp.pred
  end if

  OM ← TM.predicateObjectMap.objectMap
  if OM is a ReferencingObjectMap then
    // Since we do not know the target database, the join may have to be done by the query processing engine.
    // Hence, the joined fields are always projected, whether tp.obj is an IRI or a variable:
    refList ← refList | getReferences(OM.joinCondition.child)

  else if type(tp.obj) is VARIABLE then
    refList ← refList | getReferences(OM) AS tp.obj
  end if
  return refList
```

Algorithm 7: Generates the list of xR2RML references from a parent triples map that must be projected in the abstract query

Input: tp is a triple pattern, TM is an xR2RML triples map bound to tp, its object map is a referencing object map (it refers to a parent triples map).

Function genProjectionParent(tp, TM):

```
refList ← <empty list>
ROM ← TM.predicateObjectMap.objectMap    // Referencing Object Map

// Joined fields are always projected, whether tp.obj is an IRI or a variable:
refList ← refList | getReferences(ROM.joinCondition.parent)

// If tp.obj is a variable, the subject of the parent TM is projected too
if type(tp.obj) is VARIABLE then
   refList ← refList | getReferences(ROM.parentTriplesMap.subjectMap) AS tp.obj
end if
return refList
```

7.2 Functions genCond and genCondParent

We first describe function *getValue* that is used in subsequent functions.

Algorithm 8: 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 9: Generate the conditions to match a triple pattern with a triples map

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
            cond ← cond | isNotNull(OM.joinCondition.child)
        else if type(OM) is reference-valued or template-valued then
            cond ← cond | equals(getValue(tp.obj, OM), getReferences(OM))
        end if

    VARIABLE:
        if OM is a ReferencingObjectMap then
            cond ← cond | isNotNull(OM.joinCondition.child)
        else 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 case
```

Algorithm 10: Generate the conditions to match the object of a triple pattern with a referencing object map

Input: tp is a triple pattern, TM is an xR2RML triples map bound to tp and its object map is a referencing object map (it refers to a parent triples map), f is a SPARQL filter.

```
Function genCondParent(tp, TM, f):
  cond ← <empty list>
  OM ← TM.predicateObjectMap.objectMap
  case type(tp.obj)
    IRI:
        // 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
        // And in any case add a non null condition to satisfy the join
        cond ← cond | isNotNull(OM.joinCondition.parent)
    VARIABLE:
        // 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
        // And in any case add a non null condition to satisfy the join
        cond ← cond | isNotNull(OM.joinCondition.parent)
  end case
```

8 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 tp_1 and tp_2:

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

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

$$trans_m(bgp, true)$$
$$= trans_m(tp_1, true) \text{ INNER JOIN } trans_m(tp_2, true) \text{ ON } var(tp_1) \cap var(tp_2)$$
$$= transTP_m(tp_1, true) \text{ INNER JOIN } transTP_m(tp_2, true) \text{ ON } \{?dept\}$$

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

$$bind_m(bgp) = \{ (tp_1, \{<\#Staff>\}) , (tp_2, \{<\#Departments>)\} \}$$

In the subsequent sections we describe the execution of the *transTP$_m$* function for each triple pattern, starting with tp$_2$; 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";
      ] ] ].
```

8.1 Translation of tp$_2$ into an abstract query

Triple pattern tp$_2$: `?dept ex:hasSeniorMember ?senior.`

`getBoundTMs`$_m$`(gp, tp`$_2$`)` returns triples map `<#Departments>`.

```
transTP_m(tp_2, true) =
    From    ← {[xrr:query "db.departments.find({})"]}
    Project ← genProjection(tp_2, <#Departments>)
    Where   ← genCond(tp_2, <#Departments>, true)
```

Let us detail the calculation of *Project* part (Algorithm 6) and *Where* part (Algorithm 9):

Project:

`genProjection(tp`$_2$`, <#Departments>) = ($.code` **`AS`** `?dept, $.members[?(@.age>=40)].name` **`AS`** `?senior)`

Note that 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", we can only project field "members". Consequently, when translated to the MongoDB query language, the *Project* part shall only project fields "code" and "members": `{"code":1, "members":1}`.

Where ← `genCond(tp`$_2$`, <#Departments>, true)`:

- The subject of tp$_2$ 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 tp_2 is constant, hence no condition is entailed.
- The object of tp_2 is again a variable, this entails a second non-null condition:

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

Finally, $transTP_m(tp_2, true)$ =

```
From    ← {[xrr:query "db.departments.find({})"]}
Project ← {$.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
Where   ← {isNotNull($.code), isNotNull($.members[?(@.age >= 40)].name)}
```

8.2 Translation of tp_1 into an abstract query

Triple pattern tp_1: `<http://example.org/staff/Dunbar> ex:manages ?dept.`

$getBoundTMs_m(gp, tp_1)$ returns triples map `<#Staff>`.

$transTP_m(tp_1, true)$ =

```
{ From    ← {[xrr:query "db.staff.find({})"]}
  Project ← genProjection(tp1, <#Staff>)
  Where   ← genCond(tp1, <#Staff>, true)
} AS child
INNER JOIN
{ From    ← {[xrr:query "db.departments.find({})"]}
  Project ← genProjectionParent(tp1, <#Staff>)
  Where   ← genCondParent(tp1, <#Staff>, true)
} AS parent
ON child/$.manages.* = parent/$.dept
```

Project (Algorithm 6):

As the subject of tp_1 is a constant, the reference in the subject map of triples map `<#Staff>` is not projected. Since the object map of `<#Staff>` is a referencing object map with parent triples map `<#Departments>`, the references in the join condition must be projected: this is achieved by *genProjection* on the side of `<#Staff>`, and by *genProjectionParent* on the side of `<#Departments>`. The object of tp_1 is a variable, thus the reference of the corresponding term map must be projected too: this is the subject map of triples map `<#Departments>` projected by *genProjectionParent*:

```
genProjection(tp1, <#Staff>) = {$.manages.*}
genProjectionParent(tp1, <#Staff>) = {$.dept ,$.code AS ?dept}
```

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

Child query: `{"manages":1}`

Parent query: `{"dept":1, "code":1}`

Where part of the child query (Algorithm 9):

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

- The subject of tp_1 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", $['lastname','familyname'])
```

- The predicate of tp_1 is constant, hence no condition is entailed.
- The object of tp_1 matched with the subject map of triples map <#Departments>, this will be managed by *genCondParent*. Nevertheless we have to add a not-null condition on the child joined reference:

  isNotNull($.manages.*)

Where part of the parent query:

Where ← genCondParent(tp_1, <#Staff>, true):

- The object of tp_1 is a variable, this entails a not-null condition. It is matched with the subject map of triples map <#Departments>. Hence:

  isNotNull(getReferences(<#Departments>.subjectMap)) = isNotNull($.code)

- We must also add a not-null condition on the parent joined reference:

  isNotNull($.dept)

Finally, $transTP_m(tp_1, true)$ =

```
{ From    ← {[xrr:query "db.staff.find({})"]}
  Project ← {$.manages.*}
  Where   ← {equals("Dunbar", $['lastname','familyname']), isNotNull($.manages.*)}
} AS child
INNER JOIN
{ From    ← {[xrr:query "db.departments.find({})"]}
  Project ← {$.dept, $.code AS ?dept}
  Where   ← {isNotNull($.code), isNotNull($.dept)}
}  AS parent
ON (child/$.manages.* = parent/$.dept)
```

8.3 Abstract query optimization

When we put the translation of tp_1 and tp_2 together we obtain the following abstract query:

```
transm(bgp, true) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"]}
      Project ← {$.dept, $.code AS ?dept}
      Where   ← {isNotNull($.code), isNotNull($.dept)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
    INNER JOIN
    { From    ← [xrr:query "db.departments.find({})"]
      Project ← {$.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← { isNotNull($.code), isNotNull($.members[?(@.age>=40)].name) }
    ON {?dept}
```

The 2nd and 3rd atomic queries have the same *From* part, thus entailing a self-join. To eliminate it we first rewrite the abstract query: we change the natural associative property of joins by embedding the 2nd and 3rd atomic queries in curly brackets.

```
transm(bgp, true) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
```

```
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    {
        { From    ← {[xrr:query "db.departments.find({})"]}
          Project ← {$.dept, $.code AS ?dept}
          Where   ← {isNotNull($.code), isNotNull($.dept)} }
        INNER JOIN
        { From    ← [xrr:query "db.departments.find({})"]
          Project ← {$.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
          Where   ← {isNotNull($.code), isNotNull($.members[?(@.age>=40)].name)}
        ON {?dept}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

Now we can perform a self-join elimination by merging the two queries together: we merge the *Project* parts on the one hand, and the *Where* parts on the other hand. We obtain the following optimized abstract query:

```
trans_m(bgp, true) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"]}
      Project ← {$.dept, $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← {isNotNull($.code), isNotNull($.dept), isNotNull($.members[?(@.age>=40)].name)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

8.4 Rewriting atomic queries to MongoDB queries

Child query

Each condition of the *Where* part in translated into an abstract MongoDB query (, lines 6-10). Below we detail the execution of the *trans* function (section 4.4) 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($.manages.*, isNotNull) =
   R0    trans(.manages.*, isNotNull) =
   R8,R7,R1 FIELD(manages) ELEMMATCH(COND(sNotNull))
```

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

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

`Q1'` is inserted in the MongoDB find request along with the *Project* part, for the child query:

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

The request returns one document (, lines 12-16):

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

Parent query

Each condition of the *Where* part is translated into an abstract MongoDB query (, lines 6-10). Below we detail the execution of the *trans* function (section 4.4) by indicating the rules matched at each step:

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

Q2 ← trans($.dept, isNotNull) = FIELD(dept) COND(isNotNull)

Q3 ← trans($.members[?(@.age >= 40)].name), isNotNull) =
   R0    trans(.members[?(@.age >= 40)].name, isNotNull) =
   R8    FIELD(members) trans([?(@.age >= 40)].name, isNotNull) =
   R4    FIELD(members) ELEMMATCH(trans(.name, isNotNull), transJS(?(@.age >= 40))) =
   R8,R1 FIELD(members) ELEMMATCH(FIELD(name) COND(isNotNull), transJS(@.age >= 40)) =
   J6    FIELD(members) ELEMMATCH(FIELD(name) COND(isNotNull), COMPARE(age, $gte, 40))
```

`Q1, Q2` and `Q3` are translated into either a concrete query or a union of concrete queries (, line 11):

```
Qi' ← rewrite(AND(AND(AND(true,Q1),Q2),Q3) =
        {"code": {$exists:true, $ne:null},
         "dept": {$exists:true, $ne:null},
         "members": {$elemMatch: {"name": {$exists:true, $ne:null}, "age": {$gte:40}}}}
```

`Qi'` is inserted in the MongoDB find request along with the *Project* part:

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

The request returns two documents (, lines 12-16):

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

8.5 Complete $trans_m$ processing

Now we rewrite the optimized abstract query obtained in section 8.3 by replacing each atomic abstract query with its respective results:

```
    {
     {"manages":["R&D", "Human Resources"]}
    } AS child
```

```
INNER JOIN
{
 {"dept":"Sales", "code":"sa",
  "members":[{"name":"P. Russo", "age":28}, {"name":"J. Mendez", "age":43}]}
 {"dept":"Human Resources", "code":"hr",
  "members": [{"name":"R. Posner", "age":46}, {"name":"D. Stamper", "age":38}]}
} AS parent
ON child/$.manages.* = parent/$.dept
```

We then compute the INNER JOIN operator, this returns only two documents:

```
 {"manages":["R&D", "Human Resources"]},
 {"dept":"Human Resources", "code":"hr",
  "members": [{"name":"R. Posner", "age":46}, {"name":"D. Stamper", "age":38}]}
```

Finally, applying the xR2RML triples maps to those results shall entail 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/staff/Dunbar> ex:manages <http://example.org/dept/rd>.
<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 (, line 23) will not rule out any result. The answer to the SELECT clause is the binding of variable `?senior` to value `"R. Posner"`.

9 References

[1] N. Bikakis, C. Tsinaraki, 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] E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk, G. Xiao, A formal presentation of MongoDB (Extended version), 2016.

[5] A. Chebotko, S. Lu, F. Fotouhi, Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. 68 (2009) 973–1000.

[6] S. Das, S. Sundara, R. Cyganiak, R2RML: RDB to RDF Mapping Language, (2012).

[7] A. Dimou, M.V. 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.

[8] 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.

[9] O. Görlitz, S. Staab, SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions, in: Proc. 2nd Int. Workshop Consum. Linked Data, Bonn, Germany, 2011.

[10] 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.

[11] A. Husson, Une sémantique statique pour MongoDB, in: 25th Journ. Francoph. Lang. Appl. JFLA, 2014: pp. 77–92.

[12] 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.

[13] F. Michel, L. Djimenou, C. Faron-Zucker, J. Montagnat, xR2RML: Non-Relational Databases to RDF Mapping Language, 2014.

[14] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Trans. Database Syst. 34 (2009) 1–45.

[15] 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.

[16] M. Rodríguez-Muro, M. Rezk, Efficient SPARQL-to-SQL with R2RML mappings, Web Semant. Sci. Serv. Agents World Wide Web. 33 (2015) 141–169.

[17] 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.

[18] J.F. Sequeda, D.P. Miranker, Ultrawrap: SPARQL execution on relational data, Web Semant. Sci. Serv. Agents World Wide Web. 22 (2013) 19–39.

[19] 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.

[20] J. Unbehauen, C. Stadler, S. Auer, Accessing relational data on the web with SparqlMap, in: Semantic Technol., Springer, 2013: pp. 65–80.

[21] J. Unbehauen, C. Stadler, S. Auer, Optimizing SPARQL-to-SQL Rewriting, in: Proc. IIWAS 13, ACM, 2013: p. 324.

[22] 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.