Formalisation and Experiences of R2RML-based SPARQL to SQL query translation using Morph

Freddy Priyatna
Ontology Engineering Group
Facultad de Informática, UPM
Madrid, Spain
fpriyatna@fi.upm.es

Oscar Corcho
Ontology Engineering Group
Facultad de Informática, UPM
Madrid, Spain
ocorcho@fi.upm.es

Juan Sequeda
Dept. of Computer Science
University of Texas at Austin
Austin, USA
jsequeda@cs.utexas.edu

ABSTRACT

R2RML is used to specify transformations of data available in relational databases into materialised or virtual RDF datasets. SPARQL queries evaluated against virtual datasets are translated into SQL queries according to the R2RML mappings, so that they can be evaluated over the underlying relational database engines. In this paper we describe an extension of a well-known algorithm for SPARQL to SQL translation, originally formalised for RDBMS-backed triple stores, that takes into account R2RML mappings. We present the result of our implementation using queries from a synthetic benchmark and from three real use cases, and show that SPARQL queries can be in general evaluated as fast as the SQL queries that would have been generated by SQL experts if no R2RML mappings had been used.

1. INTRODUCTION

Making relational database content available as RDF has played a fundamental role in the emergence of the Web of Data. Several approaches have been proposed in this direction since the early 2000s, focusing on the creation of mapping languages, models and supporting technology to enable such transformations (e.g., R_2O [2], D2R [3], Triplify [1]). In September 2012, the W3C RDB2RDF Working Group released the R2RML W3C Recommendation [9], a language to specify transformations of data stored in relational databases into materialised or virtual RDF datasets, so that relational databases can be queried using SPARQL, and several R2RML-aware implementations have been reported [19].

The R2RML specification does not provide any formalisation of the SPARQL to SQL query translation process that needs to be followed by R2RML-aware query translators, and the aforementioned implementations do not describe the formalizations of their query translation algorithms. In any case, some attemps have been done in the past to provide such a formalisation. For instance, Garrote and colleagues [11] provide a draft of the transformation of SPARQL SELECT queries to SQL based on the query translation algorithm defined in [7]. As their focus is to provide an R2RML-based RESTful writable API, the query translation algorithm that they describe is limited. For example, projections and conditions for those queries are built only for variable components of the triple pattern. However, a triple pattern may involve URIs and constants in their components, and these have to be taken into account when translating SPARQL queries. Rodríguez-Muro and colleagues have recently presented Quest/Ontop [15], which translates R2RML mappings and SPARQL queries into a set of Datalog rules, where optimizations based on query containment and Semantic Query Optimisation are applied, before transforming them into SQL queries. However, the process of how R2RML mappings are translated into Datalog rules has not been described at the moment of writing. Unbehauen and colleagues [18] define the process of binding triple patterns to the mappings and the process of generating column groups (a set of columns) for every RDF term involved in the graph pattern. Using the calculated bindings and column groups, they define how to translate each of the SPARQL operators (AND, OPTIONAL, FILTER, and UNION) into SQL queries. However, the function *joinCond(s1,s2)*, in which two patterns are joined, is not clearly defined. In fact, one fundamental aspect that distinguishes SPARQL queries from SQL queries is the semantics of the joins, as discussed in [8, 7], which corresponds to the treatment of NULL values in the join conditions. None of the aforementioned systems explain how to deal with `RefObjectMap` mappings, where a triples map is joined with another triples map (parent triples map), and consequently the generated SQL query has to take into account the logical source of the parent triples map and join condition specified in the mapping.

Furthermore, the performance of virtual RDF datasets based on RDB2RDF mapping languages has not always been satisfactory, as reported by Gray et. al. [12]. This experience has also been confirmed empirically by us in projects like Répener [17], BizkaiSense[1], or Integrate[2], for which we have had access to the data sources and mappings in languages like D2R. In all cases, some of the SQL queries produced by the translation algorithm could not be evaluated (e.g., too many joins are involved) or their evaluation takes too much time to complete, what makes their use in a virtual RDF dataset context unfeasible. The main reason for this is that the resulting queries are not sufficiently optimised to be efficiently evaluated over the underlying database engines.

[1]http://www.tecnologico.deusto.es/projects/bizkaisense/
[2]http://www.fp7-integrate.eu

In the context of RDF data management, some works [7, 10] have focused on using relational databases as the backend for triple stores, with the idea of exploiting the performance provided by relational database systems. They normally work with schema-oblivious layouts (using a triple table layout containing columns corresponding to the triple elements) or vertical partitioned layouts (using different tables to store triples corresponding to the same predicate). They provide translation algorithms from SPARQL to SQL using the aforementioned layouts specifically designed to store triple instances. This is out of the scope of our work.

While R2RML specifies custom mappings defined by users, it comes with a companion standard called Direct Mapping [14], which defines the RDF representation of data in a relational database. The generated RDF instances will have their classes and properties reflecting the structure and contents of the relational database. Using Direct Mappings, Sequeda and colleagues [16] define the process of translating SPARQL into SQL efficiently, by removing self-joins and unnecessary conditions.

The contribution of our paper is threefold: first, we extend an existing SPARQL to SQL query rewriting algorithm [7], originally proposed for RDBMS-backed triple stores, by considering R2RML mappings. This implies allowing the algorithm to work with arbitrary layouts, such as the ones normally used in legacy systems. This work is complementary to [11] and provides an alternative to the one presented in [18] and [15]. Second, we implement and evaluate several versions of our algorithm (with different types of optimisations), using queries from the BSBM benchmark [4], a widely used synthetic benchmark for RDB2RDF systems, and show that the evaluation of such rewritten SPARQL queries is generally as fast as the corresponding native SQL queries specified in the benchmark. Third, we report our experience with queries from real projects, which show performance issues when using state-of-the-art RDB2RDF systems, and use our implementation to produce a better translation that allows them to be evaluated in an appropiate time.

The paper is structured as follows. In the following section, we review the R2RML language and Chebotko's approach to SPARQL-to-SQL query rewriting and data translation. In Section 3 we formalise our extension to consider R2RML mappings. In Section 4 we describe our evaluation using the BSBM synthetic benchmark, and three positive experiences of applying our approach in real case projects. Finally, we present our conclusions and future work in Section 5.

2. BACKGROUND: R2RML AND CHEBOTKO'S QUERY TRANSLATION

2.1 R2RML

As discussed in the introduction, R2RML [9] is a W3C Recommendation that allows expressing customized mappings from relational databases to RDF. An R2RML mapping document M, as shown in Figure 1, consists of a set of `rr:TriplesMap`[3] classes. The `rr:TriplesMap` class consists of three properties; `rr:logicalTable`, `rr:subjectMap`, and `rr:predicateObjectMap`.

- `rr:logicalTable` specifies the logical table to be mapped, whether it is a SQL table, a SQL view, or an R2RML view.

- `rr:subjectMap` specifies the target class and the URI generation form.

- `rr:predicateObjectMap` specifies the target property and the generation of the object via `rr:objectMap`, whose value can be obtained by constant (`rr:constant`), column (`rr:column`) or template (`rr:template`). Linking with another table is accomodated by the use of `rr:refObjectMap`, which specifies the parent triples map and the join conditions.

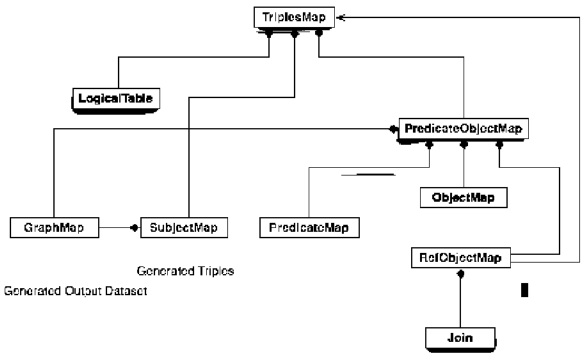


Figure 1: An overview of R2RML, based on [9]

Appendix 6 gives an example of a mapping document that maps Product and Offer tables to *ex:Product* and *ex:Offer* respectively, which we will use throughout this paper to illustrate our definitions and algorithms.

2.2 Chebotko and colleagues's approach

Our approach is based on the formalisation and algorithm from Chebotko and colleagues [7]. Other RDB2RDF query translation approaches lack a clear formalisation of the approach (e.g., [2], [3]) or they are not focused on RDF generation and/or SPARQL querying (e.g., [6]).

Chebotko's approach proposes the use of a set of mappings (α and β) and functions ($genCondSQL$, $genPRSQL$, and $name$) to translate SPARQL into SQL. Its formal definition is available at [7]. We now give an example of how those mappings and functions work in the absence of R2RML mappings. In Section 3 we provide the formal definition of these mappings and functions with R2RML mappings.

Example 1. Without R2RML Mappings. Consider a triple table `Triples(s, p, o)` that uses its columns to store the subject, predicate, and object of RDF triples. A user poses a SPARQL query with the triple pattern *tp1 = :Product1 rdfs:label ?plabel*.

- Mapping α returns the table that holds the triples that correspond to this triple pattern $tp1$. That is, $\alpha(tp1) =$ `Triples`.

- Mapping β returns the column that corresponds to each triple pattern position (sub, pre or obj). That is, $\beta(tp1, sub) =$ `s`, $\beta(tp1, pre) =$ `p`, and $\beta(tp1, obj) =$ `o`.

[3]From now on, we use prefix rr to denote R2RML namespace http://www.w3.org/ns/r2rml

- Function *genCondSQL* filters the table `Triples` returned by $\alpha(tp1)$ so that only those records that match the triple pattern *tp* are returned. $genCondSQL(tp1, \beta)$ = {`s = :Product1 AND p = rdfs:label`}.

- Function *name* generates alias for each triple pattern element. That is, *name*(*ex:Product1*) = `iri_Product1`, *name*(*rdfs:label*) = `iri_rdfs_label`, and *name*(*plabel*) = `var_plabel`.

- Function *genPRSQL* projects the β column as the value returned by function *name* for each triple pattern position. That is, $genPRSQL(tp1, \beta, name)$ = { `s AS iri_Product1, p AS iri_rdfs_label, o AS var_plabel`}.

- Function $trans(tp1, \alpha, \beta)$ finally returns the SQL query using the values returned by the previous mappings and functions. That is, $trans(tp1, \alpha, \beta)$ =
  ```
  SELECT s AS iri_Product1, p AS iri_rdfs_label,
  o AS var_plabel FROM TRIPLES
  WHERE s = :Product1 AND p = rdfs:label.
  ```
  [4]

3. AN R2RML-BASED EXTENSION OF CHEBOTKO'S APPROACH

We have seen from the previous example how the algorithm defined in [7] is used to translate SPARQL queries into SQL queries for RDBMS-backed triples stores. As mentioned in the Introduction section, this approach has to be extended so that all the functions and mappings used in the algorithm take into account user-defined R2RML mappings. Before we present the detail of how to extend the algorithm, we give an illustrative example that is comparable to the previous example.

3.1 Illustrative Example

Example 2. With R2RML Mappings. Consider a table `Product(nr, label)` that is mapped into an ontology concept `bsbm:Product`. The column `nr` is mapped as part of the URI of the `bsbm:Product` instances and column `label` is mapped to the property `rdfs:label`. The same triple pattern *tp1* = *:Product1 rdfs:label ?plabel* in the previous example produces the following mappings/functions:

- **Mapping** α returns the table that holds the triples that correspond to this triple pattern $tp1$. $\alpha(tp1)$ = `Product`.

- **Mapping** β returns the column/constant that corresponds to each triple pattern position (*sub*, *pre* or *obj*). $\beta(tp1, sub)$ = `nr`, $\beta(tp1, pre)$ = `'rdfs:label'`, and $\beta(tp1, obj)$ = `label`.

- **Function** *genCondSQL* filters the table `Product` returned by $\alpha(tp1)$ so that only the records that match the triple pattern $tp1$ are returned. That is, $genCondSQL(tp1, \beta)$ = {`nr = 1 AND label IS NOT NULL`}.

- **Function** *name* generates alias for the triple pattern element. In this case, *name*(*:Product1*) = `iri_Product1`, $name(rdfs:label)$ = `iri_rdfs_label`, and *name*(*plabel*) = `var_plabel`.

- **Function** *genPRSQL* projects the β column as the alias *name* of each triple pattern element. That is, $genPRSQL(tp1, \beta, name)$ = {`nr AS iri_Product1, 'rdfs:label' AS iri_rdfs_label, label AS var_plabel`}.

- **Function** $trans(tp1)$ finally returns the SQL query using the values returned by the previous mappings and functions. That is, $trans(tp1, \alpha, \beta)$ =
  ```
  SELECT nr AS iri_Product1,
  'rdfs:label' AS iri_rdfs_label,
  label AS var_plabel FROM Product
  WHERE nr = 1 AND label IS NOT NULL.
  ```

Figure 2 illustrates our two examples on using the Chebotko's algorithm in the context of triple stores and R2RML-based query translation.

Next, describe how we extend the translation function of a triple pattern in Chebotko's approach taking into account R2RML mappings. We then go into more detail explaining the mappings/functions used in the translation algorithm[5].

3.2 Function *trans* under *M*

Definition 1. Given the set of all possible triple patterns $TP = (IV) \times (I) \times (IVL)$ and $tp \in TP$, mappings α and β, functions *name*, *genCondSQL*, and *genPRSQL*, $trans(tp)$ generates a SQL query that can be executed by the underlying RDBMS to return the answer for *gp*.

Listing 1: $trans(tp)$ under M

```
1  trans^m(TP, α, β, name, Queries) :−
2    getTriplesMaps(TP, M, TMList),
3    transTMList(TP, α, β, name, TMList, Queries).
4
5  transTMList(TP, α, β, name, [], []).
6  transTMList(TP, α, β, name, [TM|TMListTail], Queries) :−
7    transTM(TP, α, β, name, TM, QHead),
8    transTMList(TP, α, β, name, TMListTail, QTail),
9    createUnionQuery(QHead, QTail, Queries).
10
11 transTM(TP, α, β, name, TM, Queries) :−
12   getGroundedPredicates(TP, TM, PreURIList),
13   transPreURIList(TP, α, β, name, TM, PreURIList, Queries).
14
15 transPreURIList(TP, α, β, name, TM, [], []).
16 transPreURIList(TP, α, β, name, TM, PreURIList, Queries) :−
17   PreURIList = [PreURIsHead | PreURIsTail],
18   trans(TP, α, β, name, TM, PreURIsHead, QHead),
19   transPreURIList(TP, α, β, name, TM, PreURIsTail, QTail),
20   createUnionQuery(QHead, QTail, Queries).
21
22 trans(TP, α, β, name, TM, PreURI, Query) :−
23   genPRSQL(TP, α, β, name, TM, PreURI, SelectPart),
24   α(TP, TM, PreURI, FromPart),
25   genCondSQL(TP, α, β, TM, PreURI, WherePart),
26   buildQuery(SelectPart, FromPart, WherePart, Query).
```

The algorithm (see Listing 1[6]) for translating a triple pattern $TP = (IV) \times (IV) \times (IVL)$[7] can be explained as follows:

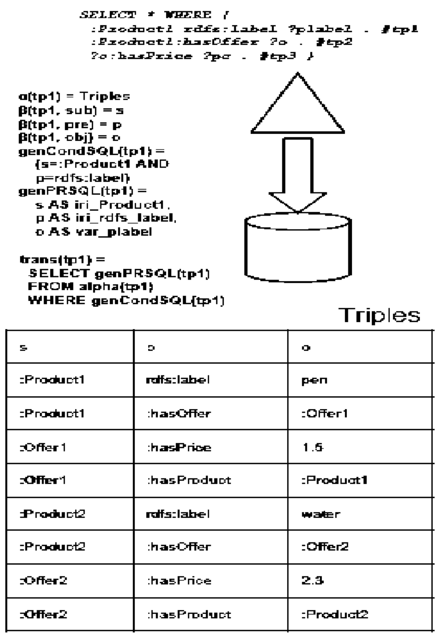
- A triple pattern can be mapped to several triples maps in the mapping document, thus all results from the possible mappings will be put in a single UNION query (line 1-7).

[4]For the sake of simplicity, we omit the aliases that have to be prefixed on each column name.

[5]While the order of presenting the mappings/functions is not important, we start by explaining $trans(tp)$, so as to facilitate understanding.

[6]We use Prolog-syntax to present our listings

[7]For compactness, we write IRI as I, Variable as V, Literal as L, and combinations of those letters.



```
SELECT * WHERE {
  :Product1 rdfs:label ?plabel . #tp1
  :Product1:hasOffer ?o . #tp2
  ?o:hasPrice ?pc . #tp3 }
```

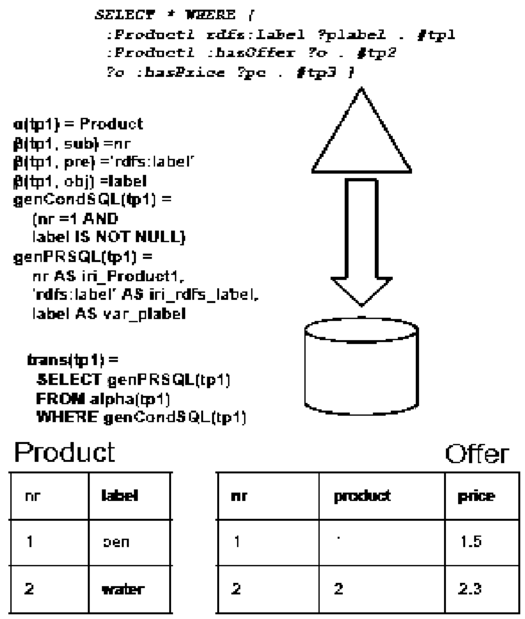
```
α(tp1) = Triples
β(tp1, sub) = s
β(tp1, pre) = p
β(tp1, obj) = o
genCondSQL(tp1) =
   {s=:Product1 AND
   p=rdfs:label}
genPRSQL(tp1) =
   s AS iri_Product1,
   p AS iri_rdfs_label,
   o AS var_plabel

trans(tp1) =
  SELECT genPRSQL(tp1)
  FROM alpha(tp1)
  WHERE genCondSQL(tp1)
```

Triples

s	p	o
:Product1	rdfs:label	pen
:Product1	:hasOffer	:Offer1
:Offer1	:hasPrice	1.5
:Offer1	:hasProduct	:Product1
:Product2	rdfs:label	water
:Product2	:hasOffer	:Offer2
:Offer2	:hasPrice	2.3
:Offer2	:hasProduct	:Product2

(a) Chebotko's in RDBMS-backed Triple Store



```
SELECT * WHERE {
  :Product1 rdfs:Label ?plabel . #tp1
  :Product1 :hasOffer ?o . #tp2
  ?o :hasPrice ?pc . #tp3 }
```

```
α(tp1) = Product
β(tp1, sub) =nr
β(tp1, pre) ='rdfs:label'
β(tp1, obj) =label
genCondSQL(tp1) =
   {nr =1 AND
   label IS NOT NULL}
genPRSQL(tp1) =
   nr AS iri_Product1,
   'rdfs:label' AS iri_rdfs_label,
   label AS var_plabel

 trans(tp1) =
  SELECT genPRSQL(tp1)
  FROM alpha(tp1)
  WHERE genCondSQL(tp1)
```

Product

nr	label
1	pen
2	water

Offer

nr	product	price
1	·	1.5
2	2	2.3

(b) Chebotko's in R2RML-based Query Translation

Figure 2: Chebotko's algorithm in Database-backed Triples Store and R2RML-based Query Translation

- The same process occurs if the predicate part of the triple map is unbounded. Then the variable of the predicate part will be grounded according to available mappings, and results will be merged as a UNION query (line 11-20).

- In the case where the triples map is defined and the predicate of the triple pattern is bounded, $trans(tp)$ builds a SQL query using the result of auxiliary mappings (α, β) and functions ($genCondSQL$, $genPRSQL$) (line 22-26). Those mappings and functions are explained in the following subsections.

The translation function $trans$ for other patterns (AND, OPTIONAL, UNION, FILTER) follows the algorithm described in [7].

3.3 Mapping α under M

Definition 2. Given the set of all possible triple patterns $TP = (IV) \times (I) \times (IVL)$, a set of relations REL, and a set of all possible mappings M defined in R2RML, a mapping α is a many-to-many mapping $\alpha : TP \times M \to REL$, where given a triple pattern $tp \in TP$ and a mapping $m \in M$, $\alpha^m(tp)$ returns a set of relations that generate all the triples that match tp.[8]

Listing 2: α under M

```
1  $\alpha^m(TP, TM, PreURI, [AlphaSub, AlphaPreObj])$ :−
2    $\alpha^m_{sub}(TM, AlphaSub)$,
3    $\alpha^m_{preobj}(TP, TM, PreURI, AlphaPreObj)$.
4
5  $\alpha^m_{sub}(TM, TM.logicalTable)$.
6
7  $\alpha^m_{preobj}(TP, TM, PreURI, AlphaPreObj)$ :−
8    $getPredicateObjectMap(TM, PreURI, POMap)$,
9    $ROMap = POMap.refObjectMap$,
10   $\alpha^m_{preobj}(TM, PreURI, [ROMap], AlphaPreObj)$.
11
12 $\alpha^m_{preobj}(TM, PreURI, [], [])$.
13 $\alpha^m_{preobj}(TM, PreURI, [ROMap], ROMap.parentTriplesMap.logicalTable)$.
```

The algorithm for computing mapping α under a set of R2RML mappings is provided in Listing 2. The output of this algorithm is used as the FROM part of the generated SQL query. This algorithm is divided into two main parts; calculating α for the subject ($\alpha^m_{sub}(TM)$) (line 5), and for the predicate-object part $\alpha^m_{preobj}(POMap)$ (line 7-13).

- The function $\alpha^m_{sub}(TM)$ returns the logical table property *logicalTable* of a triples map TM (line 5).

- A logical table from a triple may be joined with another logical table through the `refObjectMap` property. This case is handled by an auxiliary function $\alpha^m_{preobj}(POMap)$ that retrieves the `predicateObjectMap` property $POMap$ of TM that corresponds to the predicate of the triple pattern tp, and then returns the parent logical table of `refObjectMap` property $ROMap$ (line 7-13).

The output of mapping α is a set of logical tables with the result from the two auxiliary functions (line 1-3). Table 1 presents some examples of mapping α results.

3.4 Mapping β under M

Definition 3. Given a set of all possible triple patterns $TP = (IV) \times (I) \times (IVL)$, a set of positions in a triple pattern $POS = \{sub, pre, obj\}$, a set of relational attributes $ATTR$, and a set of all possible R2RML mappings M, a mapping β is a many-to-one mapping $\beta^m : TP \times POS \times M \to ATTR$, if

[8]If the set REL contains more than one relation, the join conditions are specified in the genCondSQL() function, described in Section 3.5

given a triple pattern $tp \in TP$, a position $pos \in POS$, and $m \in M$, $ATTR$ is a relational attribute whose value may match the triple pattern tp at position pos.

The algorithm for computing the mapping β under M is provided in Listing 3 and some examples of β results can be seen on Table 1. The output of mapping β is used in the functions *genPRSQL* and *genCondSQL* to select the relational attribute corresponding to the triple pattern position:

- If the position *pos* is subject, then the auxiliary function β^m_{sub} returns the corresponding relational attributes attached to the subject map SM of *TriplesMap* (line 1-2, 8-9).
- If the position *pos* is predicate, then URI of *tp.predicate* is returned as a constant. (line 3-4, 11).
- If the position *pos* is object, then the function β^m_{obj} (line 13-16) checks whether the argument *POMap* contains a Reference Object Map *ROMap*.
  - If *POMap* contains *ROMap*, the parent triple map *ParentTriplesMap* of *ROMap* is retrieved and the relational attributes corresponding to the subject map of *ParentTriplesMap* are returned (line 20-21).
  - Otherwise, the corresponding relational attributes of object map *objectMap* are returned (line 18-19).

Listing 3: β under M

```
1  $\beta^m(TP, pos.sub, TM, PredicateURI, \alpha, BetaResult)$ :-
2    $\beta^m_{sub}(TM, \alpha, BetaResult)$.
3  $\beta^m(tp, pos.pre, tm, predicateURI, \alpha, BetaResult)$ :-
4    $\beta^m_{pre}(PredicateURI, BetaResult)$.
5  $\beta^m(tp, pos.obj, tm, predicateURI, \alpha, BetaResult)$ :-
6    $\beta^m_{obj}(TM, PredicateURI, \alpha, BetaResult)$.
7
8  $\beta^m_{sub}(TM, \alpha, BetaResult)$ :-
9    databaseColumn(TM.subjectMap, BetaResult).
10
11 $\beta^m_{pre}(PredicateURI, PredicateURI)$.
12
13 $\beta^m_{obj}(TM, PredicateURI, \alpha, BetaResult)$ :-
14   getPredicateObjectMap(TM, PredicateURI, POMap),
15   ROMap = POMap.refObjectMap,
16   $\beta^m_{obj}(TM, PredicateURI, \alpha, POMap, [ROMap], BetaResult)$.
17
18 $\beta^m_{obj}(TM, PredicateURI, \alpha, POMap, [], BetaResult)$ :-
19   databaseColumn(POMap.objectMap, BetaResult).
20 $\beta^m_{obj}(TM, PredicateURI, \alpha, POMap, [ROMap], BetaResult)$ :-
21   databaseColumn(ROMap.parentTriplesMap.subjectMap, BetaResult).
```

3.5 Function *genCondSQL* under M

Definition 4. Given a set of all possible triple patterns TP, a mapping β, and a set of all possible mappings M defined in R2RML mapping document, *genCondSQL* generates a SQL expression that is evaluated to true if and only if *tp* matches a tuple represented by relational attributes $\beta^m(tp, sub)$, $\beta^m(tp, pre)$, and $\beta^m(tp, obj)$ where $tp \in TP$ and $m \in M$.

The algorithm of function *genCondSQL* under M is provided in Listing 4. The output of this function is used as the WHERE part of the generated SQL query. This function calls and returns the result of two auxiliary functions: $genCondSQL^m_{sub}$ and $genCondSQL^m_{preobj}$.

Listing 4: *genCondSQL* under M

```
1  $genCondSQL^m(TP, \alpha, \beta, TM, PreURI, CondSub)$ :-
2    $genCondSQL^m_{sub}(TP, \alpha, \beta, TM, CondSub)$,
3    $genCondSQL^m_{preobj}(TP, \alpha, \beta, TM, PreURI, CondPreObj)$,
4    Conds = [CondSub, CondPreObj].
5
6  $genCondSQL^m_{sub}(TP, \alpha, \beta, TM, CondSub)$ :-
7    type(TP.subject, IRI),
8    SM = TM.subjectMap,
9    inverseExpr(TP.subject, SubjectId),
10   $\beta^m_{sub}(SM, \alpha, BetaSub)$,
11   genSQLExpr(equals, SubjectId, BetaSub, CondSub).
12 $genCondSQL^m_{sub}(TP, \alpha, \beta, TM, [])$ :- type(TP.subject, Type), $Type \neq IRI$.
13
14 $genCondSQL^m_{preobj}(TP, \alpha, \beta, TM, PreURI, Cond)$ :-
15   getPredicateObjectMap(TM, preURI, POMap),
16   OMap = POMap.objectMap,
17   ROMap = POMap.refObjectMap,
18   ObjType = type(TP.object),
19   $genCondSQL^m_{preobj}(TP, \alpha, \beta, TM, PreURI, ObjType, ROMap, Cond)$.
20
21 $genCondSQL^m_{preobj}(TP, \alpha, \beta, TM, PreURI, var, [], CondPreObj)$ :-
22   $\beta^m_{obj}(tp, preURI, \alpha, BetaObj)$,
23   genSQLExpr(isNotNull, BetaObj, CondPreObj).
24 $genCondSQL^m_{preobj}(TP, \alpha, \beta, TM, PreURI, var, ROMap, EqCond)$ :-
25   joinElements = ROMap.joinCond.child, ROMap.joinCond.parent,
26   genSQLExpr(equals, joinElements, EqCond).
27
28 $genCondSQL^m_{preobj}(TP, \alpha, \beta, TM, PreURI, literal, \_, CondPreObj)$ :-
29   $\beta^m_{obj}(TP, PreURI, \alpha, BetaObj)$,
30   genSQLExpr(equals, TP.object, BetaObj, CondPreObj).
31
32 $genCondSQL^m_{preobj}(TP, \alpha, \beta, TM, PreURI, iri, [], CondPreObj)$ :-
33   $\beta^m_{sub}(ROMap, \alpha, BetaObj)$,
34   inverseExpr(TP.object, ObjId),
35   genSQLExpr(equals, ObjId, BetaObj, CondPreObj).
36 $genCondSQL^m_{preobj}(TP, \alpha, \beta, TM, PreURI, iri, ROMap, CondPreObj)$ :-
37   inverseExpr(TP.object, ObjId),
38   joinElements = ROMap.joinCond.child, ROMap.joinCond.parent,
39   genSQLExpr(equals, joinElements, EqCond),
40   $\beta^m_{sub}(ROMap, \alpha, BetaSub)$,
41   genSQLExpr(and, EqCond, BetaSub, condPreObj).
```

The function $genCondSQL^m_{sub}$ checks the term type of *tp.subject*. If *tp.subject* is an IRI (line 6-11), then the identifier of this subject is obtained by a function *inverseExpression*, a function that takes an IRI and returns only the corresponding values that can be related to database values. For example, *inverseExpression* will return 1 or 3847 for IRIs :Product1 or :Offer3847, respectively. $genCondSQL^m_{sub}$ returns nothing if the subject is not an IRI (line 12).

The auxiliary function $genCondSQL^m_{preobj}(tp, POMap)$ checks the term type of *tp.object*.

- If *tp.object* is a variable, two cases are evaluated.
  - In the case that *PredicateObjectMap* doesn't have any Reference Object Map, then the SQL expression stating that the β of *PredicateObjectMap* IS NOT NULL is returned (line 21-23). This is to guarantee that we do not return NULL values. Only in case that the value is used in some filter expressions (e.g. *FILTER(!bound(?label))*) that we must remove the IS NOT NULL expression so that *genCondSQL* returns NULL, what will be interpreted as unbounded values.
  - If *RefObjectMap* of *PredicateObjectMap* is specified, then a SQL expression that is the *JoinCondition* property from *RefObjectMap* is returned (line 24-26).
- If *tp.object* is a literal, then a SQL equality expression of *tp.object* and the output of β for *PredicateObjectMap* is returned (line 28-30).

- If *tp.object* is an IRI, then $OBJID$ as the identifier of *tp.object* is evaluated using the inverse expression. Then two cases are evaluated.
  - If no Reference Object Map is specified, the function returns an SQL equality expression of $OBJID$ and β^m_{obj} mapping of $POMap$ (line 32-35).
  - If an instance of Reference Object Map is specified, then the function returns 1)$JoinCondition$ and 2)an SQL equality expression between $OBJID$ and the $\beta^m_{sub}(ROMap)$ mapping of $ROMap$ (line 36-41).

Table 1 illustrates the result of calculating $genCondSQL$ for our example.

3.6 Function $genPRSQL$ under M

Definition 5. Given a set of all possible triple pattern TP and $tp \in TP$, a mapping β, a function *name*, and a set of all possible mappings M defined in R2RML and $m \in M$, $genPRSQL$ generates SQL expression which projects only those relational attributes that correspond to distinct *tp.subject*, *tp.predicate*, *tp.object* and renames the projected attributes as $\beta^m(tp, sub) \rightarrow name(tp.subject)$, $\beta^m(tp, pre) \rightarrow name(tp.predicate)$, and $\beta^m(tp, obj) \rightarrow name(tp.object)$.

Listing 5: $genPRSQL$ under M

```
1  $genPRSQL^m(TP, \alpha, \beta, name, TM, PreURI, Result)$ :−
2    $genPRSQL^m_{sub}(TP, \alpha, \beta, name, TM, PreURI, RSub)$,
3    $genPRSQL^m_{pre}(TP, \alpha, \beta, name, TM, PreURI, RPre)$,
4    $genPRSQL^m_{obj}(TP, \alpha, \beta, name, TM, PreURI, RObj)$,
5    $Result = [RSub, RPre, RObj]$.
6
7  $genPRSQL^m_{sub}(TP, \alpha, \beta, name, TM, PRSQLSub)$ :−
8    $\beta^m_{sub}(TM, \alpha, BetaSub)$, $name(TP.subject, SubjectAlias)$,
9    $generateSQLExpression(as, BetaSub, SubjectAlias, PRSQLSub)$.
10
11 $genPRSQL^m_{pre}(TP, \alpha, \beta, name, TM, PreURI, [])$ :−
12   $TP.subject = TP.predicate$.
13 $genPRSQL^m_{pre}(TP, \alpha, \beta, name, TM, PreURI, PRSQLPre)$ :−
14   $TP.subject \neq TP.predicate$, $\beta^m_{pre}(PreURI, BetaPre)$,
15   $name(TP.predicate, PredicateAlias)$,
16   $generateSQLExpression(as, BetaPre, PredicateAlias, PRSQLPre)$.
17
18 $genPRSQL^m_{obj}(tp, \alpha, \beta, name, TM, PreURI, [])$ :−
19   $TP.object = TP.subject$.
20 $genPRSQL^m_{obj}(tp, \alpha, \beta, name, TM, PreURI, PRSQLObj)$ :−
21   $TP.object \neq TP.subject$,
22   $TP.object \neq TP.predicate$,
23   $\beta^m_{obj}(TM, PreURI, \alpha, BetaObj)$,
24   $name(TP.object, ObjectAlias)$,
25   $generateSQLExpression(as, BetaObj, ObjectAlias, PRSQLObj)$.
```

The algorithm for computing $genPRSQL$ under M is provided in Listing 5. The outputs of this function are used in the SELECT part of the generated SQL as the select items of the SQL queries together with its aliases, which are generated by function *name*. As defined in [7], given a term in IVL, a function *name* generates a unique name, such that the generated name conforms to the SQL syntax for relational attribute names. In our examples, *name(t)* generates term type appended by the name of the term. For example, the result of *name(?lbl)* is `var_lbl` and the result of *name(hasOffer)* is `iri_hasOffer`. The function $genPRSQL$ projects all the β attributes and assigns them unique aliases generated from the *name* function (line 9, 16, and 25). Some examples of $genPRSQL()$ results can be seen in Table 1.

3.7 Query Rewriting Optimizations

The translation mappings/functions that we have presented so far can be evaluated directly over relational database systems (RDBMS). Taking into account that many of these RDBMS have already implemented a good number of optimisations for their query evaluation, most of the resulting queries can be evaluated at the same speed as the native SQL queries what would have been generated using SQL directly. However, there are RDBMS that need special attention (e.g., MySQL, PostgreSQL), because the resulting queries contain some properties that affect the capabilities of the database optimizer. For this reason, whenever it is possible, we remove non-correlated subqueries from the resulting queries by pushing down projections and selections (as advocated by [10]). We also remove self-joins that occur in the rewritten queries such as when the graph patterns contain alpha mappings coming from the same table(s). Those translations are illustrated in Listing 6.

Listing 6: Examples of translation queries

```
#SPARQL
SELECT DISTINCT ?pr ?productLabel ?productComment ?productProducer
WHERE {
   ?pr hasLabel ?label .
   ?pr hasComment ?comment .
   ?pr hasProducer ?producer .
}

-- naive translation (C)
SELECT DISTINCT pr, productLabel, productComment, productProducer
FROM (
  SELECT p_1.pr, p_1.productLabel,
    p_2_3.productComment, p_2_3.productProducer
  FROM
   (SELECT nr AS pr, label AS productLabel FROM product
     WHERE nr IS NOT NULL AND label IS NOT NULL) p_1,
   (SELECT p_2.pr, p_2.productComment, p_3.productProducer FROM
     (SELECT nr AS pr, comment AS productComment FROM product
        WHERE nr IS NOT NULL AND comment IS NOT NULL) p_2,
   (SELECT nr AS pr, producer AS productProducer FROM product
     WHERE nr IS NOT NULL AND producer IS NOT NULL) p_3
  WHERE p_2.pr= p_3.pr) p_2_3
WHERE p_1.pr = p_2_3.pr
) gp

-- subquery elimination (SQE)
SELECT p1.nr AS pr, p1.label AS productLabel,
  p2.comment AS productComment, p3.producer AS productProducer
FROM product p1, product p2, product p3
WHERE p1.nr = p2.nr AND p2.nr = p3.nr
  AND p1.nr IS NOT NULL AND p2.nr IS NOT NULL
  AND p3.nr IS NOT NULL AND p1.label IS NOT NULL
  AND p2.comment IS NOT NULL AND p3.producer IS NOT NULL

--self-join elimination (SJE)
SELECT pr, productLabel, productComment, productProducer
FROM
   (SELECT p.nr AS pr, p.label AS productLabel
     , p.comment as productComment, p.producer AS productProducer
   FROM product p
   WHERE p.nr IS NOT NULL AND p.label IS NOT NULL
   AND p.comment IS NOT NULL AND p.producer IS NOT NULL
   ) gp

--subquery and self-join elimination (SQE + SJE)
SELECT p.nr AS pr, p.label AS productLabel,
  p.comment AS productComment, p.producer AS productProducer
FROM product p
WHERE p.nr IS NOT NULL AND p.label IS NOT NULL
  AND p.comment IS NOT NULL AND p.producer IS NOT NULL
```

4. EVALUATION

We separate our evaluations into two categories, using a synthetic benchmark and using three real projects. Their details are available in the following sub-sections. Our query translation algorithm has been implemented in our latest version of Morph, which is available as a Java/Scala open-source project in Github[9]. The query translation types supported by Morph are the naïve translation queries (C), which

[9]https://github.com/fpriyatna/morph

tp	*TriplesMap*	α^m	β^m	$genCondSQL^m$	$genPRSQL^m$
?product rdfs:label ?label	TMProduct	{PRODUCT}	sub:{PRODUCT.nr}, pre:{'rdfs:label'}, obj:{PRODUCT.label}	{PRODUCT.label IS NOT NULL}	{PRODUCT.nr AS var_product, 'rdfs:label' AS iri_rdfs_label, PRODUCT.label AS var_label}
:Product1 hasOffer :Offer3847	TMProduct	{PRODUCT, OFFER}	sub:{PRODUCT.nr}, pre:{'hasOffer'}, obj:{OFFER.nr}	{PRODUCT.nr = 1 AND PRODUCT.nr = OFFER.product AND OFFER.nr = 3847}	{PRODUCT.nr AS iri_Product1, 'hasOffer' AS iri_has_offer, OFFER.nr AS iri_Offer2847}

Table 1: Example of mappings and functions results under R2RML document m

are the result of the query rewriting algorithm described in Section 3, together with three variants of it; with subquery elimination (SQE), self-join elimination (SJE), and both types of eliminations (SQE+SJE).

4.1 Synthetic Benchmark Evaluation

The BSBM benchmark [5] focuses on the e-commerce domain and provides a data generation tool and a set of twelve SPARQL queries together with their corresponding SQL queries generated by hand. The data generator is able to generate datasets with different sizes containing entities normally involved in the domain (e.g., products, vendors, offers, reviews, etc). For the purpose of our benchmark, we work with the 100-million triple dataset configuration.

All these queries have been evaluated on the same machine, with the following configuration: Pentium E5200 2.5GHz processor, 4GB RAM, 320 GB HDD, and Ubuntu 13.04. The database server used for the synthetic benchmark queries is PostgreSQL 9.1.9. We normalize the evaluation time over the native evaluation time. We have run all queries with 20 times with different parameters, in warm mode run.The resulting sets of queries together with query plans generated by PostgreSQL9.1.9, and the resulting query evaluation time are available at `http://bit.ly/15XSdDM`.

The BSBM SPARQL queries are designed in such a way that they contain different types of queries and operators, including SELECT/CONTRUCT/DESCRIBE, OPTIONAL, UNION. In the same spirit, the corresponding SQL queries also consider various properties such as low selectivity, high selectivity, inner join, left outer join, and union among many others. Out of the 12 BSBM queries, we focus on all of the 10 SELECT queries (that is, we leave out DESCRIBE query Q09 and CONSTRUCT query Q12). We compare the native SQL queries (N), which are specified in the BSBM benchmark with the ones resulting from the translation of SPARQL queries generated by Morph. Although not included here, we also evaluated those queries using D2R 0.8.1 with the –fast option enabled. The reason why we do not include here the results from these evaluations is because in many queries (such as in Q2, Q3, Q4, Q7, Q8, and Q11), D2R produces multiple SQL queries and then the join/union operations are performed at the application level, rather than in the database engine, what prevents us from doing direct comparisons. Nevertheless, this approach is clearly not scalable (e.g., in Q07 and Q08 the system returned an error while performing the operations, while the native and the translation queries could be evaluated over the database system).

4.1.1 Discussion

We can observe that all translation types (native, C, SQE, SJE, SQE+SJE) have similar performance in most of BSBM queries, ranging from 0.67 to 2.60 when normalized according to the native SQL queries. To understand this behaviour better, we analyzed the query plans generated by the RDBMS. Our observation tells us that in many of the queries (Q01, Q02, Q03, Q05, Q06, Q10, and Q11), C produces identical query plans to SQE's, and SJE produces identical query plans to SQE+SJE's. Additionally, SQE also produces identical query plans to SQE+SJE in Q07 and Q08. The reason for this is the capability of the database's optimizer to eliminate non-correlated subqueries. The difference between the query plans produced by C/SQE and the ones produced by SJE/SQE+SJE is the number of joins to be performed. However, as the join conditions are normally perfomed on indexed columns, there is small overhead in terms of their performance. This explains why all the translation results have similar performance.

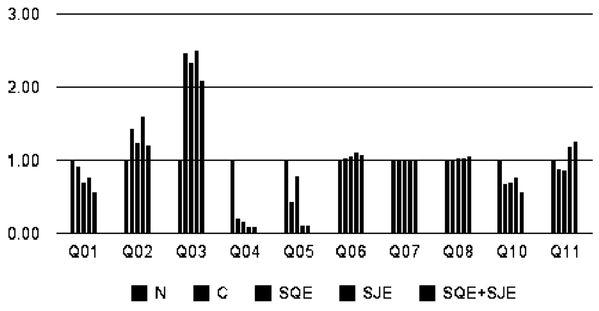


Figure 3: BSBM query evaluations (normalized time to native query)

However, in some queries the translation results show significant differences, such as in Q04 and Q05.

- BSBM SQL 4 contains a join between two tables (product and producttypeproduct) and three subqueries, two of them are used as OR operators. The SPARQL equivalent of this query is a UNION of two BGPs (a set of triple patterns). We note that the native query contains a correlated subquery and the generated query plan requires a table scan to find a specific row condition. The query plans generated by the translation algorithm, on the other hand, produce joins, instead of a correlated subquery, and the joins are able to exploit the indexes defined.

- BSBM SQL 5 is a join of four tables (product, product, productfeatureproduct, and productfeatureproduct). The size of table productfeatureproduct is significantly bigger than the table product (280K rows vs 5M rows). The generated query plan by the native query joins bigger tables (productfeatureproduct and productfeatureproduct) before joining the intermediate result with the smaller table (product and product). This join order is specified in the query itself. The join orders of the translation queries are different; C and SQE join based on the order of triple patterns in the graph, SJQ and SQE+SJE join based on the smaller tables first (which is an effect of the self-joins elimination process).

These explain why the translation queries perform better than the native queries in Q04 and Q05 and in fact show that the native queries proposesd in the benchmark should have been better optimised when proposed for the benchmark.

4.2 Real Cases Evaluation

While the BSBM benchmark is considered as a standard way of evaluating RDB2RDF approaches, given the fact that it is very comprehensive, we were also interested in analysing real-world queries from projects that we had access to, and where there were issues with respect to the performance of the SPARQL to SQL query rewriting approach. In all the cases, we compare the queries generated by D2R Server with –fast enabled with the queries generated by Morph with subquery and self-join elimination enabled. All the resulting queries together with their query plans are also available at `http://bit.ly/15XSdDM`.

4.2.1 BizkaiSense Project

The BizkaiSense project is an effort to measure various environmental properties coming from sensors that are deployed throughout Greater Bilbao, Spain. Some of the most common queries that are needed in this project are:

- Q01 obtains all observations coming from a particular weather or air quality station together with the time of the observation. We set 100, 1000, and 10000 as the maximum number of rows to be returned.
- Q02 extends Q01 by returning the sensor results generated from those observations obtained. We set 100, 1000, and 10000 as the maximum number of rows to be returned.
- Q03 returns the average measures by property for a given week in a given station (with the units of measure).
- Q04 returns the average measures by property for a given week in a given station (without the units of measure).
- Q05 returns the maximum measure in all the stations for each property in a given day (returning also the station in which it happened).
- Q06 returns the maximum measure in all the stations for each property in a given day (without the station in which it happened).
- Q07 returns the maximum measure in all the stations for a given property in a given day (returning the station in which it happened) -avoiding the use of the "MAX" clause.

The corresponding SPARQL queries can be seen in Listing 7. All queries generated by D2R and Morph are evaluated on a machine with the following specification: Intel(R) Xeon(R) CPU E5640 2.67GHz x 16, 48GB RAM, Ubuntu 12.04 LTS - 64 bits and MySQL 5.5 with a 10 minutes timeout. The evaluation time of those queries (in some cases with a limit in the number of returned bindings) is shown in Figure 4. We note that the database server failed to evaluate the queries Q03, Q05, Q06, and Q07 generated from D2R Server and in all cases, the queries generated by Morph take significantly less time to be evaluated compared to the ones generated by D2R Server.

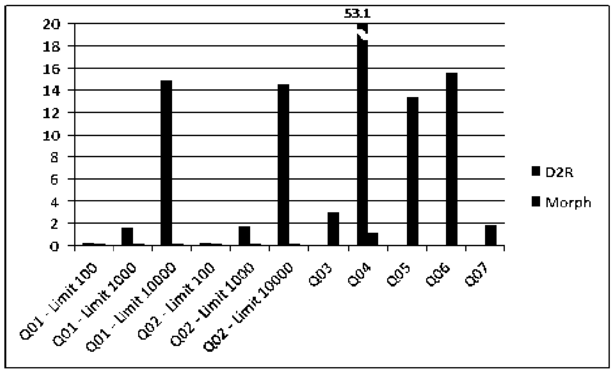


Figure 4: BizkaiSense - query evaluation time (in seconds)

4.2.2 RÉPENER Project

RÉPENER [17] is a Spanish national project that provides access to energy information using semantic technology called SEiS. Two of the most common queries that are consulted through SEiS are the following:

- Q01 retrieves all buildings and their climatezone and building life cycle phase
- Q02 retrieves all buildings and their climatezone, building life cycle phase, and conditioned floor area.

The corresponding SPARQL queries are shown in Listing 8. All queries generated by D2R and Morph are evaluated on a machine with the following specification: Intel Core 2 Quad Q9400 2.66 GHz, 4 GB RAM, Windows 7 Professional 32 bits, and MySQL 5.5 as the database server. As shown in Figure 5, all queries are successfully evaluated by the database server, with queries generated by Morph being 2-3 faster than those generated by D2R server.

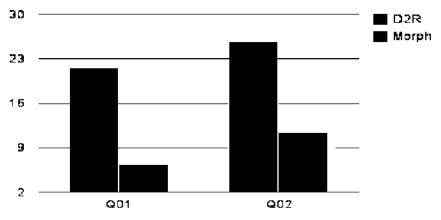


Figure 5: SEiS - query evaluation time (in seconds)

4.2.3 Integrate Project

Integrate is an FP7 project for sharing and integrating clinical data using HL7-RIM [13], a model that represents

Listing 7: BizkaiSense queries

```
#Q01 obtains all observations coming from a particular weather
# or air quality station together with the time of the observation.
SELECT DISTINCT ?medition ?date WHERE {
?medition ssn:observedBy <http://localhost:2020/resource/station/ANORGA> .
?medition dc:date ?date . } LIMIT 10 # 100, 1000

#Q02 extend Q01 by returning the sensor results generated
# from those observations obtained.
SELECT DISTINCT ?medition ?date ?res WHERE {
?medition ssn:observedBy <http://localhost:2020/resource/station/ANORGA>.
?medition dc:date ?date .
?medition ssn:observationResult ?res . } LIMIT 10 # 100, 1000

#Q03 returns the average measures by property for a given week
# in a given station (with the units of measure).
SELECT (AVG(?datavalue) AS ?avg_month) ?prop ?unit WHERE {
?medition ssn:observedBy <http://localhost:2020/resource/station/BETONO> ;
  dc:date ?date ;
  ssn:observedProperty ?prop ;
  ssn:observationResult ?obsres .
?obsres ssn:hasValue ?val .
?val dul:hasDataValue ?datavalue ;
  dul:isClassifiedBy ?unit .
FILTER ( ?date >= "2011-01-01T00:00:00"^^xsd:date
  && ?date <= "2011-01-07T00:00:00"^^xsd:date )
} GROUP BY ?prop ?unit

#Q04 returns the average measures by property for a given week
# in a given station (without the units of measure).
SELECT (AVG(?datavalue) AS ?avg_month) ?prop WHERE {
?medition ssn:observedBy <http://localhost:2020/resource/station/BETONO> ;
  dc:date ?date ;
  ssn:observedProperty ?prop ;
  ssn:observationResult ?obsres .
?obsres ssn:hasValue ?val .
  ?val dul:hasDataValue ?datavalue ;
  dul:isClassifiedBy ?unit .
FILTER ( ?date >= "2011-01-01T00:00:00"^^xsd:date
  && ?date <= "2011-01-07T00:00:00"^^xsd:date )
} GROUP BY ?prop

#Q05 returns the maximum measure in all the stations for each property
# in a given day (returning also the station in which have happened).
SELECT (MAX(?datavalue) AS ?max) ?prop ?station WHERE {
?medition a ssn:Observation ;
  dc:date ?date ;
  ssn:observedBy ?station ;
  ssn:observedProperty ?prop ;
  ssn:observationResult ?obsres .
?obsres ssn:hasValue ?val .
?val dul:hasDataValue ?datavalue .
FILTER ( ?date >= "2011-01-01T00:00:00"^^xsd:date
  && ?date < "2011-01-02T00:00:00"^^xsd:date )
} GROUP BY ?prop ?station

#Q06 returns the maximum measure in all the stations for each property
# in a given day (without the station in which have happened).
SELECT (MAX(?datavalue) AS ?max) ?prop WHERE {
?medition a ssn:Observation ;
  dc:date ?date ;
  ssn:observedProperty ?prop ;
  ssn:observationResult ?obsres .
?obsres ssn:hasValue ?val .
?val dul:hasDataValue ?datavalue .
FILTER ( ?date >= "2011-01-01T00:00:00"^^xsd:date
  && ?date < "2011-01-02T00:00:00"^^xsd:date ) }

#Q07 returns the maximum measure in all the stations for a given property
# in a given day (returning the station in which have happened)
# -avoiding the use of MAX clause.
SELECT ?datavalue ?station WHERE {
?medition a ssn:Observation ;
  dc:date ?date ;
  ssn:observedBy ?station ;
  ssn:observedProperty
        <http://sweet.jpl.nasa.gov/2.3/matrCompound.owl#NO>;
  ssn:observationResult ?obsres .
?obsres ssn:hasValue ?val .
?val dul:hasDataValue ?datavalue .
FILTER ( ?date >= "2011-01-01T00:00:00"^^xsd:date
  && ?date < "2011-01-02T00:00:00"^^xsd:date )
} ORDER BY DESC(?datavalue) LIMIT 1
```

Listing 8: SEiS queries

```
#Q01 Retrieve all buildings and their climatezone
# and building life cycle phase
SELECT DISTINCT * WHERE {
  ?a repener:hasBuilding ?building .
  ?a repener:value ?climatezone .
  ?building a sumo:Building .
  ?building repener:hasProjectData ?projectData .
  ?projectData repener:hasBuildingLifeCyclePhase ?buildingLifeCyclePhase .
  ?buildingLifeCyclePhase repener:value ?phase . }

#Q02. Retrieve all buildings and their building life cycle phase
# and conditioned floor area
SELECT DISTINCT * WHERE {
  ?b rdf:type sumo:Building .
  ?b repener:hasProjectData ?b1 .
  ?b1 repener:hasBuildingLifeCyclePhase ?b2 .
  ?b2 repener:value ?phase .
  ?b repener:hasBuildingProperties ?b3 .
  ?b3 repener:hasBuildingGeometry ?b4 .
  ?b4 repener:hasConditionedFloorArea ?b5 .
  ?b5 repener:conditionedFloorAreaValue ?conditionedFloorArea . }
```

entities and relationships commonly involved in clinical activities. Some commonly-used queries in this project are:

- Q01/Q02/Q03 are similarly structured, with a code that specifies whether to obtain tumor size, tumor stage, or pregnant women.
- Q04 obtains multiple participants who have been treated with Antracyclines
- Q05 obtains demographic information (people that are older than 30 years old)
- Q06 obtains images and information that a diagnosis is based on

The corresponding SPARQL queries are shown in Listing 9. All queries generated by D2R and Morph are evaluated on a machine with the following specification: Intel QuadCore processor 2 GHz, 4 GB of RAM, 126GB SSD HD, Ubuntu 64-bits operating system and MySQL 5.6. The evaluation time can be seen in Figure 6. For Q01/Q02/Q03, the queries generated by D2R could not be evaluated by the database server because they returned an error message saying that too many tables needed to be joined. For the other queries, the queries generated by Morph were evaluated in less time than the ones generated by D2R.

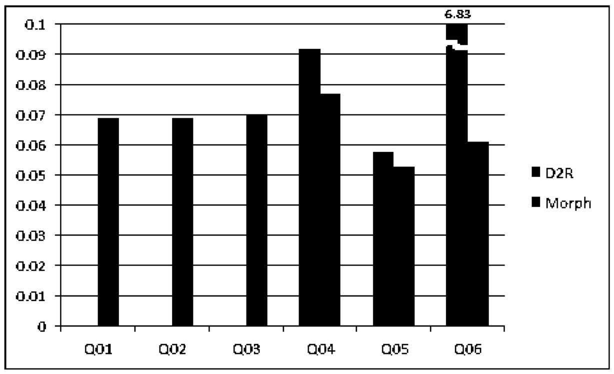


Figure 6: Integrate - query evaluation time (in seconds)

5. CONCLUSION AND FUTURE WORK

In this paper we have shown that the query translation approaches used so far for handling RDB2RDF mapping languages may be inefficient. For instance, systems like D2R normally produce multiple queries to be evaluated and then perform the join operations in memory rather than in the database engine, or generate SQL queries where the number of joins inside the query is so large that the underlying database engine cannot evaluate them, producing an error.

We have proposed an extension of one of the most detailed approaches for query rewriting, Chebotko's, which was not originally conceived for RDB2RDF query translation but for RDBMS-backed triple stores. Now we consider R2RML

Listing 9: HL7-RIM query

```
#Q01/Q02/Q03
SELECT DISTINCT ?id ?code ?entityId ?birthTime ?effectiveTime ?valueST
?valuePQ WHERE {
?instObs    hl7rim:observation_code ?code;
  hl7rim:observation_id ?id;
  hl7rim:observation_effectiveTime ?effectiveTime;
  hl7rim:observation_targetSiteCode ?tarSiteCode;
  hl7rim:observation_targetSiteCodeTitle ?tarSiteTitle;
  hl7rim:observation_targetSiteCodeVocId ?tarSiteVocId;
  hl7rim:observation_methodCode ?methodCode;
  hl7rim:observation_methodCodeTitle ?methodTitle;
  hl7rim:observation_methodCodeVocId ?methodVocId;
  hl7rim:observation_valueST ?valueST;
  hl7rim:observation_valuePQ ?valuePQ;
  hl7rim:observation_units ?units;
  hl7rim:observation_refRangeMin ?rangeMin;
  hl7rim:observation_refRangeMax ?rangeMax;
  hl7rim:observation_title ?title;
  hl7rim:observation_actionNegationInd ?actNegInd;
  hl7rim:observation_codeVocId ?codeVocId;
  hl7rim:observation_participation ?obs_part.
?obs_part   hl7rim:participation_entityId ?entityId;
  hl7rim:participation_role ?obs_role.
?livSubj    hl7rim:livingSubject_id ?entityId;
  hl7rim:livingSubject_birthTime ?birthTime.
?obs_role    hl7rim:role_entity ?obs_entity.
?obs_entity    hl7rim:entity_code ?entityCode.

#Q01to obtain tumor size of patients, code value is 263605001
# FILTER (?code IN (263605001))

#Q02:to obtain tumor stage of patients, code value is 58790005
# FILTER (?code IN (58790005))

#Q03:to obtain patients that are/have been pregnant, code value is 77386006
# FILTER (?code IN (77386006))
} LIMIT 100

#Q04 to retrieving multiple participants who have been treated
# with Antracyclines (Family of drugs)
SELECT * WHERE {
?substanceAdm a hl7rim:substanceAdministration .
?substanceAdm hl7rim:substanceAdministration_code ?substanceAdmCode .
?substanceAdm hl7rim:substanceAdministration_code "432102000" .
?substanceAdm hl7rim:substanceAdministration_participation ?partDevice .
?device a hl7rim:entity .
?device hl7rim:entity_id ?deviceId .
?device hl7rim:entity_code "417916005" .
?device hl7rim:entity_role ?deviceRole .
?deviceRole hl7rim:role_participation ?partDevice .
?substanceAdm hl7rim:substanceAdministration_participation ?partPatient .
?patient a hl7rim:livingSubject .
?patient hl7rim:livingSubject_id ?livingSubjectId .
?patient hl7rim:livingSubject_role ?patientRole .
?patient hl7rim:livingsubject_classCode "PSN" .
?patientRole hl7rim:role_classCode "PAT" .
?patientRole hl7rim:role_participation ?partPatient . }

#Q05 to obtain patient demographics information: people older than 30 years
SELECT DISTINCT ?entityId ?birthTime ?gender WHERE {
?subject hl7rim:livingSubject_id ?subjectId .
?subject hl7rim:livingSubject_birthTime ?birthTime .
?subject hl7rim:livingSubject_administrativeGenderCode ?gender .
?subject hl7rim:livingsubject_classCode "PSN" .
FILTER (?birthTime < "1983-10-02T00:00:00"^^xsd:date) }

#Q06 tobtain images and information where a diagnosis are based in
SELECT DISTINCT ?observationId ?code ?title ?imagId ?xml WHERE {
?obsInst hl7rim:observation_id "ee3606f9-9dd1-11e2-9bba-0155938d90a2";
  hl7rim:observation_id ?observationId;
  hl7rim:observation_code ?code;
  hl7rim:observation_title ?title;
  hl7rim:observation_actRelationship ?instRel.
?instRel hl7rim:actRelationship_typeCode "EXPL";
  hl7rim:actRelationship_actB ?obsInstB.
?obsInstB hl7rim:act_id ?imagId;
  hl7rim:act_text ?xml }
```

mappings in the query translation process. We have also shown through our empirical evaluation that in all of the BSBM queries and real cases queries, our approach behaves in general similarly to native queries, and better than other existing approaches.

There is still room for improvement in our work, which we will address in the near future. For instance, some of the optimised translated queries perform better than the native ones, due to the introduction of additional predicates, what is common in the area of Semantic Query Optimization (SQO). In this sense, we will deepen in the design of our algorithm so as to take into account other SQO techniques that can be useful in query translation. For instance, predicate introduction may speed up the query evaluation process by exploiting more indexes. Our evaluation setup will be also made available as a service for other researchers to use, so that they can evaluate their R2RML query rewriting implementations with low effort in a number of RDBMSs.

Acknowledgements. This research was supported by the Spanish project myBigData and by PlanetData (FP7-257641). Juan Sequeda was supported by the NSF Graduate Research Fellowship. We also thank Boris, Jean-Paul, and Jose-Mora for the valuable discussions and people responsible of the Integrate (Raul Alonso, David Perez del Rey), BizkaiSense (Jon Lázaro, Oscar Peña, and Mikel Emaldi) and Repener (Alvaro Sicilia) projects for providing us the RDB2RDF mappings and queries used in these projects.

6. MAPPING DOCUMENT EXAMPLE

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix bsbm: <http://localhost:2020/resource/vocab/> .

<TMProduct> a rr:TriplesMap;
  rr:logicalTable [ rr:tableName  "Product" ];
  rr:subjectMap [ a rr:Subject; rr:class bsbm:Product;
    rr:template "http://localhost:2020/resource/Product/{nr}"; ];
  rr:predicateObjectMap [
    rr:predicateMap [ rr:constant rdfs:label ];
    rr:objectMap [ rr:column "label"; ]; ];
  rr:predicateObjectMap [
    rr:predicateMap [ rr:constant rdfs:comment ];
    rr:objectMap [ rr:column "comment"; ]; ];
  rr:predicateObjectMap [
    rr:predicateMap [ rr:constant bsbm:productOffer ];
    rr:objectMap [ rr:parentTriplesMap <TriplesMapOffer>;
      rr:joinCondition [ rr:child "nr" ; rr:parent "product" ; ] ];
].

<TMOffer> a rr:TriplesMap;
  rr:logicalTable [ rr:tableName  "Offer" ];
  rr:subjectMap [ a rr:Subject; rr:class bsbm:Offer;
    rr:template "http://localhost:2020/resource/Offer/{nr}"; ];
  rr:predicateObjectMap [
    rr:predicateMap [ rr:constant bsbm:price ];
    rr:objectMap [ rr:column "price"; ]; ].
```

7. REFERENCES

[1] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify: lightweight Linked Data publication from relational databases (2009). In *18th International Conference on World Wide Web*, pages 621–630, 2009.

[2] J. Barrasa and A. Gómez-Pérez. Upgrading relational legacy data to the semantic web. In *15th International Conference on World Wide Web*, pages 1069–1070. ACM, 2006.

[3] C. Bizer and R. Cyganiak. D2R Server : Publishing relational databases on the web. In *The 5th International Semantic Web Conference*, 2006.

[4] C. Bizer and A. Schultz. The berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[5] C. Bizer and A. Schulz. Benchmarking the performance of storage systems that expose sparql endpoints. In *4th International Workshop on Scalable Semantic Web knowledge Base Systems (SSWS2008)*, 2008.

[6] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of Automated reasoning*, 39(3):385–429, 2007.

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

[8] R. Cyganiak. A relational algebra for sparql. Technical Report HPL-2005-170, Digital Media Systems Laboratory HP Laboratories Bristol. `https://www.hpl.hp.com/techreports/2005/HPL-2005-170.pdf`.

[9] S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF mapping language. W3C Recommendation, `http://http://www.w3.org/TR/r2rml/`.

[10] B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z. Ozsoyoglu. A complete translation from SPARQL into efficient SQL. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 31–42. ACM, 2009.

[11] A. Garrote and M. García. Restful writable APIs for the web of linked data using relational storage solutions. In *WWW 2011 Workshop: Linked Data on the Web (LDOW2011)*, 2011.

[12] A. Gray, N. Gray, and I. Ounis. Can RDB2RDF tools feasibily expose large science archives for data integration? In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, pages 491–505. Springer-Verlag, 2009.

[13] A. Hasman et al. HL7 RIM: an incoherent standard. In *Ubiquity: Technologies for Better Health in Aging Societies, Proceedings of Mie2006*, volume 124, page 133. IOS Press, 2006.

[14] E. P. Marcelo Arenas, Alexandre Bertails and J. Sequeda. A direct mapping of relational data to RDF. W3C Recommendation, `http://www.w3.org/TR/rdb-direct-mapping/`.

[15] M. Rodrıguez-Muro, J. Hardi, and D. Calvanese. Quest: Efficient SPARQL-to-SQL for RDF and OWL. Poster presented at International Semantic Web Conference, ISWC 2012.

[16] J. Sequeda and D. P. Miranker. Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2013.

[17] A. Sicilia, G. Nemirovskij, M. Massetti, and L. Madrazo. RÉPENER's linked dataset (in revision). *Semantic Web Journal*, 2013.

[18] J. Unbehauen, C. Stadler, and S. Auer. Accessing relational data on the web with sparqlmap. In *JIST*, pages 65–80. Springer, 2013.

[19] B. Villazóón-Terrazas and M. Hausenblas. RDB2RDF implementation report. W3C Editor's Draft, `http://www.w3.org/2001/sw/rdb2rdf/implementation-report/`.