Accessing Relational Data on the Web with SparqlMap

Jörg Unbehauen, Claus Stadler, and Sören Auer

Universität Leipzig, Postfach 100920, 04009 Leipzig, Germany
{unbehauen,cstadler,auer}@informatik.uni-leipzig.de
http://aksw.org

Abstract. The vast majority of the structured data of our age is stored in relational databases. In order to link and integrate this data on the Web, it is of paramount importance to make relational data available according to the RDF data model and associated serializations. In this article we present *SparqlMap*, a SPARQL-to-SQL rewriter based on the specifications of the W3C R2RML working group. The rationale is to enable SPARQL querying on existing relational databases by rewriting a SPARQL query to exactly one corresponding SQL query based on mapping definitions expressed in R2RML. The SparqlMap process of rewriting a query on a mapping comprises the three steps (1) mapping candidate selection, (2) query translation, and (3) query execution. We showcase our SparqlMap implementation and benchmark data that demonstrates that SparqlMap outperforms the current state-of-the-art.

Keywords: Triplification, SPARQL, RDB2RDF, R2RML.

1 Introduction

The vast majority of the structured data of our age is stored in relational databases. In order to link and integrate this data on the Web, it is of paramount importance to make relational data available according to the RDF data model and associated serializations.

Also, for several reasons, a complete transition from relational data to RDF may not be feasible: Relational databases commonly provide rich functionality, such as integrity constraints; data management according to the relational data model often requires less space and may be simpler than with RDF, such as in cases which would require reification or n-ary relations; the cost of porting existing applications or maintaining actual datasets in both formats may be prohibitive; and RDBs are still a magnitude faster than RDF stores. As it can not be expected that these gaps will close soon, relational data management will be prevalent in the next years. Hence, for facilitating data exchange and integration it is crucial to provide RDF and SPARQL interfaces to RDBMS.

In this article we present *SparqlMap*[1], a SPARQL-to-SQL rewriter based on the specifications of the *W3C R2RML working group*[2]. The rationale is to

[1] http://aksw.org/Projects/SparqlMap
[2] http://www.w3.org/TR/r2rml/

H. Takeda et al. (Eds.): JIST 2012, LNCS 7774, pp. 65–80, 2013.

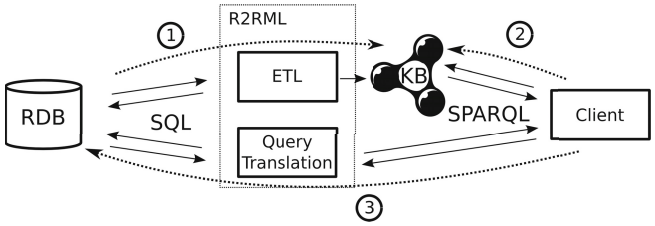

Fig. 1. Two models for mapping relational data to RDF: query rewriting and RDF extraction

enable SPARQL querying on existing relational databases by rewriting a SPARQL query to *exactly one* corresponding SQL query based on mapping definitions expressed in R2RML. The R2RML standard defines a language for expressing how a relational database can be transformed into RDF data by means of term maps and triple maps. In essence, implementations of the standard may use two process models, which are depicted in Figure 1: Either, the resulting RDF knowledge base is materialized in a triple store (1) and subsequently queried using SPARQL (2), or the materialization step is avoided by dynamically mapping an input SPAQRL query into a corresponding SQL query, which renders exactly the same results as the SPARQL query being executed against the materialized RDF dump (3).

SparqlMap is in terms of functionality comparable with *D2R* [3] or *Virtuoso RDF Views* [1] with the focus on performing all query operators in the relational database in a single unified query. This strategy ensures scalability since expensive round trips between the RDBMS and the mapper are reduced and the query optimization and execution of the RDBMS are leveraged. SparqlMap is designed as a standalone application facilitating light-weight integration into existing enterprise data landscapes.

In the following we give a short overview over our approach. The prequisite for rewriting SPARQL queries to SQL is a set of mapping definitions over which queries should be answered. In Section 3 we formalize the mapping and query syntax. The process of rewriting a query on a mapping is performed in the following three steps:

Mapping candidate selection. The initial step, described in Section 4.1, identifies candidate mappings. These are mappings that potentially contribute to the query's result set: Informally, this is the set of mappings that yield triples that could match the triple patterns of the query, as shown in Figure 4. The relation between the candidate mappings and the triple patterns is called a binding.

Query translation. The identified candidate mappings and the obtained bindings enable us to rewrite a SPARQL query to an SQL query. This process is described in Section 4.2.

Query execution. Finally, we show in Section 4.3 how the SPARQL result set is constructed from the SQL result set of the executed SQL query.

We finally evaluate our approach using the BSBM benchmark in Section 5 and show that SparqlMap is on average an order of magnitude faster than state-of-the-art techniques.

The contributions of our work described in this article include in particular (1) the formal description of mapping candidate selection, (2) an approach for efficient query translation and (3) showcasing the implementation and demonstrate its efficiency.

2 Related Work

We identified two related areas of research. First, as many native triple stores are based on relational databases there is considerable research on efficiently storing RDF data in relational schema. Exemplary are both [7] and [5], discussing the translation of a SPARQL into a single SQL query. The translations presented there are however targeted towards database backed triple stores and need to be extended and adopted for usage in a database mapping scenario. Also notable is [8], describing SQL structures to represent facets of RDF data in relational databases. Second, the mapping of relational databases into RDF is a way of leveraging existing data into the Semantic Web. We can differentiate between tools that either expose their data as RDF, Linked Data or expose a SPARQL endpoint for interactive querying of the data. An example for RDF and Linked Data exposition is *Triplify* [2]. Exposing data via a SPARQL endpoints either requires loading transformed data into a SPARQL-enabled triple store or rewriting SPARQL queries into SQL. The answering of SPARQL queries over relational data is the goal of several concepts and implementations. *D2R Server* [3] is a standalone web application, answering SPARQL queries by querying the mapped database. D2R mixes in-database and out-of-database operations. Operators of an incoming SPARQL queries like joins and some filters are performed in the mapped database directly. Other operators are then later executed on the intermediate results directly by D2R. OpenLink's *Virtuoso RDF Views* [1] allows the mapping of relational data into RDF. RDF Views are integrated into the Virtuoso query execution engine, consequently allowing SPARQL query over native RDF and relational data. A further SPARQL-to-SQL tool is *Ultrawrap* [11], which integrates closely with the database and utilizes views for answering SPARQL queries over relational data. However, to the best of our knowledge, there is no approach describing in detail the mapping and translation process for generating a single, unified SQL query.

3 Definitions

In this section we define the syntax of RDF, SPARQL and the mapping. The RDF and SPARQL formalization is closely following [9].

Definition 1 (RDF definition). *Assume there are pairwise disjoint infinite sets I, B, and L (IRIs, blank nodes, and RDF literals, respectively). A triple $(v_s, v_p, v_o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple. In this tuple, v_s is the subject, v_p the predicate and v_p the object. We denote the union $I \cup B \cup L$ as by T called RDF terms.*

Using the notion $t.i$ for $i \in \{s, p, o\}$ we refer to the RDF term in the respective position. In the following, the same notion is applied to *triple patterns* and *triple maps*. An RDF *graph* is a set of RDF triples (also called RDF dataset, or simply a dataset). Additionally, we assume the existence of an infinite set V of variables which is disjoint from the above sets. The W3C recommendation SPARQL[3] is a query language for RDF. By using *graph patterns*, information can be retrieved from SPARQL-enabled RDF stores. This retrieved information can be further modified by a query's solution modifiers, such as sorting or ordering of the query result. Finally the presentation of the query result is determined by the *query type*, return either a set of triples, a table or a boolean value. The graph pattern of a query is the base concept of SPARQL and as it defines the part of the RDF graph used for generating the query result, therefore *graph patterns* are the focus of this discussion. We use the same graph pattern syntax definition as [9].

Definition 2 (SPARQL graph pattern syntax). *The syntax of a SPARQL graph pattern expression is defined recursively as follows:*

1. *A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern (a triple pattern).*
2. *The expressions (P_1 AND P_2), (P_1 OPT P_2) and (P_1 UNION P_2) are graph patterns, if P_1 and P_2 are graph patterns.*
3. *The expression (P FILTER R) is a graph pattern, if P is a graph pattern and R is a SPARQL constraint.*

Further the function $var(P)$ returns the set of variables used in the graph pattern P. SPARQL constraints are composed of functions and logical expressions, and are supposed to evaluate to boolean values. Additionally, we assume that the query pattern is well-defined according to [9].

We now define the terms and concepts used to describe the SPARQL-to-SQL rewriting process. The basic concepts are the relational database schema and a mapping for this schema m. The schema ishas a set of relations R and each relation is composed of attributes, denoted as $A_r = (r.a_0, r.a_1, ..., r.a_l)$. A mapping m defines how the data contained in tables or views in the relational database schema s is mapped into an RDF graph g. Our mapping definitions are loosely based on R2RML. An example of such a mapping is depicted in Figure 2 and used further in this section to illustrate the translation process.

Definition 3 (Term map). *A* term map *is a tuple $tm = (A, ve)$ consisting of a set of relational attributes A from a single relation r and a value expression ve that describes the translation of A into RDF terms (e.g. R2RML templates for generating IRIs). We denote by the range $range(tm)$ the set of all possible RDF terms that can be generated using this term map.*

[3] `http://www.w3.org/TR/rdf-sparql-query/`

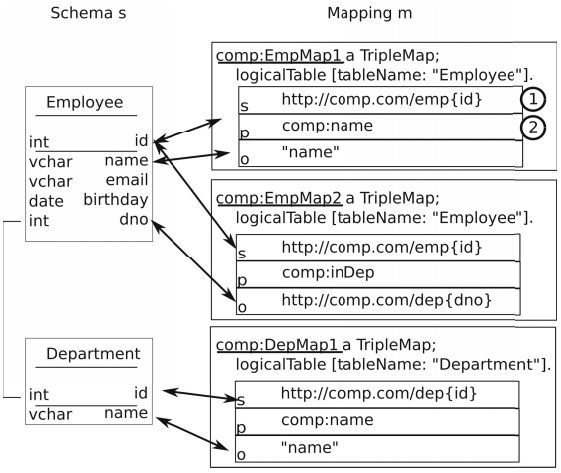

Fig. 2. Exemplary mapping of parts of two relations using three triple maps. R2RML's construct `logicalTable` specifies the source relation of a triple map.

Term maps are the base element of a mapping. In Figure 2 an example for such a *term map* is (1). With *ve* being the template `http://comp.com/emp{id}` and $A = \{Employee.id\}$ it is possible to produce resource IRIs for employees. The RDF term (2) in Figure 2 creates a constant value, in this case a property. Consequently, for this RDF term $A = \emptyset$ holds.

Definition 4 (Triple map). *A triple map* trm *is the triple* (tm_S, tm_P, tm_O) *of three* term maps *for generating the subject (position s), predicate (position p) and object (position o) of a triple. All attributes of the three* term maps *must originate from the same relation* r*.*

A triple map defines how triples are actually generated from the attributes of a relation (i.e. rows of a table). This definition differs slightly from the R2RML specification, as R2RML allows multiple predicate-object pairs for a subject. These two notions, however, are convertible into each other without loss of generality. In Figure 2 the triple map `comp:EmpMap1` defines how triples describing the name of an employee resource can be created for the relation `Employee`.

A mapping definition $m = (R, TRM)$ is a tuple consisting of a set of relations R and a set of triple maps TRM. It holds all information necessary for the translation of a SPARQL query into a corresponding SQL query. We assume in this context that all data is stored according to the schema s is mapped into a single RDF graph and likewise that all queries and operations are performed on this graph[4].

[4] Note, that support for named graphs can be easily added by slightly extending the notion of triple map with an additional term map denoting the named graph.

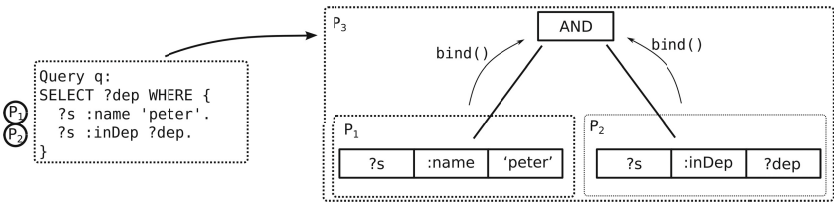

Fig. 3. Mapping candidate selection overview. The patterns of a query parsed into a tree. The bind function recurses over that tree.

4 The SparqlMap Approach

The SparqlMap approach is based on the three steps of mapping candidate selection, query translation and query execution that are discussed in this section.

4.1 Mapping Candidate Selection

Mapping selection is the process of identifying the parts of a mapped graph that can contribute to the solution of a query q. This selection process is executed for every query and forms the basis for the following step – the translation into SQL. The parts of a query that are used for matching the parts of a graph examined are the graph patterns. The graph of a mapped database is the set of triples defined by the *triple maps*. Consequently, we propose the selection of candidate *triple maps* to match the *graph pattern*. The general approach described here aims at first binding each *triple pattern* of q to a set of candidate triple maps, and then to reduce the amount of bindings by determining the unsatisfiability of constraints (e.g join conditions) based on the structure of the SPARQL query.

Before we formally introduce the operators used, we give a brief overview of the process in Figure 3. The simple query q depicted here represents a tree-like structure according to Definition 2. In a bottom-up traversal we first search for mapping candidates for the triple patterns P_1 and P_2. In the next step, indicated by the *bind*() function, these mapping candidates are examined on the next higher level of the tree. Based on the semantics of P_3 the *bind*() function reduces the mapping candidates for P_1 and P_2. Before we formally describe this process we define the notion of a binding.

Definition 5 (Triple Pattern Binding). *Let q be a query, with TP_q being its set of triple patterns. Let m be the mapping of a database, with TRM_m being its set of all triple maps. A triple pattern binding tpb is the tuple (tp, TRM), where $tp \in TP_q$ and $TRM \subseteq TRM_m$. We further denote by the set QPB_q for a query q the set of triple pattern bindings TPB, such that there exists for every $tp \in TP_q$ exactly one tpb.*

In this context we assume that in case q contains a triple pattern more than once, for each occurrence there exists in QPB_q a separate *tpb*. The set *TRM* for

a triple pattern *tp* is also called the set of mapping candidates for *tp*. We now define successively the basic terms and functions on the triple pattern bindings and illustrate them using the sample query introduced in Figure 3. In Figure 4 the result of the process is depicted. The dotted squares indicate the triple pattern bindings *tpb* with their patterns and triple maps.

Definition 6 (Term map compatibility). *We consider two term maps tm_1 and tm_2 to be compatible, if $range(tm_1) \cap range(tm_2) \neq \emptyset$. We further consider a term map tm compatible with an RDF term t, if the term $t \in range(tm)$. A variable v is always considered compatible with a term map.*

With the boolean function $compatible(t_1, t_2)$ we denote the check for compatibility. This function allows us to check, if two term maps can potentially produce the same RDF term and to pre-check constraints of the SPARQL query. Mapping candidates that cannot fulfill these constraints are removed from the binding. Further it allows to check, if a triple map is compatible with a triple pattern of a query. In the example given in Figure 4 term map compatibility is used to bind *triple maps* to *term maps*. At position (1) in Figure 4 the triple pattern P_2 is bound to the term map `:EmpMap2` because the resource IRI at the predicate position of P_2 is compatible with the constant value term map of `:EmpMap2` in the same position. The notion of term compatibility can be extended towards checking bindings for compatibility by the functions *join* and *reduce*.

Definition 7 (Join of triple pattern bindings). *Let $tpb_1 = (tp_1, TRM_1)$ and $tpb_2 = (tp_2, TRM_2)$ be triple pattern bindings. Further, let $V = var(tp1) \cap var(tp2)$ be the set of shared variables.*

We define $join(tpb_1, tpb_2) : \{(trm_a, trm_b) \in TRM_1 \times TRM_2 |$ for each variable $v \in V$ the union of the corresponding sets of term maps of trm_a and trm_b is either empty or its elements are pairwise compatible.[5] *}*

Definition 8 (Reduction of triple pattern bindings). *The function $reduce(tpb_1, tpb_2)$ is defined as $proj(join(tpb_1, tpb_2), 1)$, i.e. the projection of the first component of the tuples obtained by the join operation.*

Reduction is the base operation for minimizing the set of *triple maps* associated with every *triple pattern*. It rules out all candidate tuples that would eventually yield unsatisfiable SQL join conditions. In Figure 4 the reduction process follows the dotted line indicated by (2). The triple patterns P_1 and P_2 share the variable $?s$ which is in both cases in the subject position of the triple pattern. Consequently, each triple map in TPB_1 is compared at the subject position with all subject term maps of TPB_2 for compatibility. If no compatible triple map is found, the triple map is removed from the candidate set. The term map `:DepMap1` in Figure 4 is therefore not included in TPB_3, as the subject of `:DepMap1` is not compatible with the subject of `:EmpMap2`. The reduction function now allows

[5] Note, that the same variable may occur multiple times in a triple pattern and therefore map to multiple term maps.

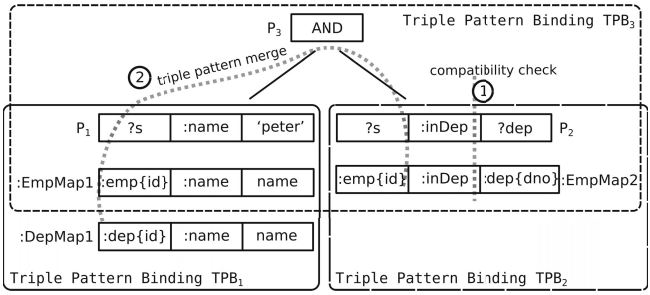

Fig. 4. Binding operations on a sample query. In (1) a simple check for compatibility at the predicate position is performed, in (2) the triples maps are merged between two triple pattern bindings, checking compatibility at the subject position.

(a) $merge(TPB_1, TPB_2)$

$TPB \leftarrow TPB_1 \cup TPB_2$
$TPB' \leftarrow \emptyset$
for $tpb_1 in TPB$ **do**
  for $tpb_2 in TPB$ **do**
    $tpb_1 \leftarrow reduce(tpb_1, tpb_2)$
  end for
  $TPB' \leftarrow TPB' \cup \{tpb_1\}$
end for
return TPB'

(b) $optmerge(TPB_1, TPB_2)$

$TPB' \leftarrow TPB_1$
for $tpb_1 in TPB_1$ **do**
  for $tpb_2 in TPB_2$ **do**
    $tpb_2 \leftarrow reduce(tpb_2, tpb_1)$
  end for
  $TPB' \leftarrow TPB' \cup \{tpb_2\}$
end for
return TPB'

Fig. 5. The merge and optmerge algorithms

the definition of two operators that perform a reduction of mapping candidates along the syntax of a SPARQL query.

For the two sets of *triple pattern bindings* TPB_1 and TPB_2 we define two merge operations for the triple pattern bindings as follows:

Binding merge $merge(TPB_1, TPB_2)$ reduces all triple pattern bindings with each other, as illustrated in Figure 5a.

Binding opt merge $optmerge(TPB_1, TPB_2)$ reduces all triple pattern bindings of TPB_2 with the all triple pattern bindings of TPB_1, as illustrated in Figure 5b.

Both merge operations preevaluate the join conditions of the later SQL execution. The compatibility check for the shared variables of two triple patterns rule out unfulfillable join or respectively left join conditions.

We can use these operators to define the recursive function $bind_m(P)$, which computes for a mapping m and the graph pattern P the set of triple pattern bindings TPB_p, similar to the recursive evaluation function defined in [9].

Definition 9. *Let TRM_m be the set of all triple maps in m, P_1 and P_2 be graph patterns and tp be a triple pattern of a query. The function $bind_m(P)$*

is the recursive binding of the TRM_m to the triple patterns of a query for the following cases:

1. *If P is a triple pattern tp, $bind(P) = \{(tp, TRM_{tp}) | TRM_{tp} = \{trm | trm \in TRM_m \wedge compatible(trm.s, tp.s) \wedge compatible(trm.p, tp.p) \wedge compatible(trm.o, tp.o)\}\}$.*
2. *If P is (P_1 AND P_2), $bind(P) = merge(bind_m(P_1), bind_m(P_2))$*
3. *If P is (P_1 OPT P_2), $bind(P) = optmerge(bind_m(P_1), bind_m(P_2))$*
4. *If P is (P_1 UNION P_2), $bind(P) = (bind_m(P_1) \cup bind_m(P_2)$*
5. *If P is (P_1 FILTER R), $bind(P) = \{tpb | tpb \in bind_m(P_1) \wedge$ if tpb is sharing variables with R, the constraint is pre-evaluated. If the filter is always false, the term map is not included.$\}$*

The complete binding process can now be illustrated using the example in Figure 4. Starting from the bottom, $bind(P_1)$ evaluates to $TPB_1 = \{(P_1, \{$ `:empMap1`,`:depMap1`$\})\}$ and $bind(P_2)$ to $TPB_2 = \{(P2, \{$`:empMap2`$\})\}$. For P_1 the triple map `:empMap2` is not bound, because $compatible(P1.p,$ `:empMap2.p`$) = false$. In the next step of the recursion, the pattern binding merge is evaluated between the two sets, creating TPB_3. The sets of triple maps of TPB_1 and of TPB_2 are reduced on the shared variable s. The term map at the subject position of `:depMap1` is not compatible with the subject from another triple map of TPB_1 and is not included in TPB_3. Here the recursion halts, the set obtained in the last step represents the full mapping of the query $QPB = TPB_3$. QBP is a set of two triple pattern bindings, each with one triple map, and is ready to be used for creating the SQL query in the next step of the process.

The approach described in Definition 9 has some limitations. Variables used in both sub-patterns of UNIONs are not exploited for reducing the candidate triple maps. This for example could be overcome by using algebraic equivalence transformations which are described in [10]. Another limitation is that not all candidate reductions are applied to triple patterns that are not directly connected by shared variables. The modifications to the query binding strategy dealing with this limitation are part of the future work on SparqlMap.

4.2 Query Translation

Query translation is the process of generating a SQL query from the SPARQL query using the bindings determined in the previous step. We devise a recursive approach to query generation, similar to the one presented for mapping selection. The result of this translation is a nested SQL query reflecting the structure of the SPARQL query. We first describe the function $toCG(tm)$ that maps a term map tm into a set of column expressions CG, called a *column group*. The utilization of multiple columns is necessary for efficient filter and join processing and data type compatibility of the columns in a SQL union. In a CG the columns can be classified as:

Type columns. The RDF term type, i.e. resource, literal or blank node is encoded into these columns using constant value expressions. The column

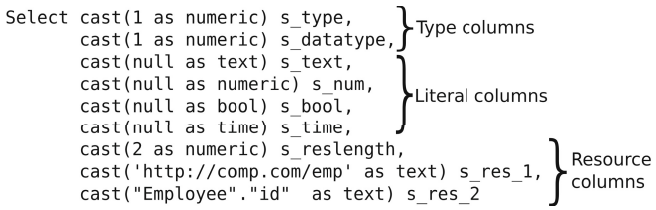

Fig. 6. Column group for variable `?s` of graph pattern P_1

expression `cast(1 as numeric) s_type` declares the RDF terms produced in this column group to be resources.

Resource columns. The IRI value expression VE is embedded into multiple columns. This allows the execution of relational operators directly on the columns and indexes.

Literal columns. Literals are cast into compatible types to allow SQL UNION over these columns.

In Figure 6 the column group created for the variable `?s` of triple pattern P_1 is depicted. The following aspects of query translation require the definition of additional functions.

Align. The alignment of of two select statements is a prerequisite for performing a SQL union as the column count equality and data type compatibility of the columns are mandatory. The function $\mathit{align}(s_1, s_2)$ for two SQL select statements s_1 and s_2 returns s_1' by adding adding columns to s_1 such that s_1' contains all columns defined in s_2. The columns added do not produce any RDF terms.

Join. Filter conditions are performed on column groups, not directly on columns. As already outlined in [6] this requires embedding the filter statements into a conditional check using `case` statements. This check allows the database to check for data types and consequently to select the correct columns of a column group for comparison. For two SQL queries s_1, s_2 the function $\mathit{joinCond}(s_1, s_2)$ calculates the join condition as an expression using `case` statements, checking the column groups bound to of shared variables.

Filter. For R being a filter expression according to Definition 2 (5), the function $\mathit{filter}_f(R)$ translates a filter expression into an equivalent SQL expression on column groups.

Triple Pattern. The RDF literals and resources used in a triple pattern tp are implicit predicates and need to be made explicit. The function $\mathit{filter}_p(tp)$ maps these triple patterns into a set of SQL predicates, similar to the definition of $\mathit{filter}_f(R)$.

Alias. The renaming of a *column group* CG is performed by the function $\mathit{alias}(CG, a)$ that aliases all column expressions of CG, by adding the prefix a. For the scope of this paper, we assume that proper alias handling is performed and is not further explicitly mentioned.

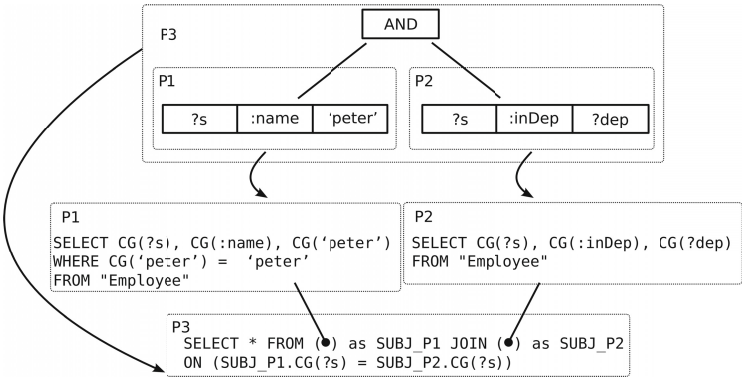

Fig. 7. Query nesting for a sample query

Using the previously defined function $toCG(tm)$ and the usage of the SQL operators *JOIN, LEFT JOIN and UNION* we can now devise a simple recursive translation function.

Definition 10 (Query translation). *Let QPB be a query pattern binding, P_1 and P_2 be graph patterns and tp be a triple pattern and $tpb = (tp, TRM)$ be triple pattern binding for tp with the set of term maps TRM. The relation for each $trm \in TRM$ is denoted r. The translation of a graph pattern P into a SQL query $Q_{sql} = t_{QPB}(P)$ is performed as follows.*

1. *If P is a triple pattern:* $t_{QPB}(P) =$ `UNION ALL` $\{\forall trm \in TRM$: `SELECT` $toCG(trm.s), toCG(trm.p), toCG(trm.o)$ `FROM` r `WHERE` $filter_p(P)\}$
2. *If P is (P_1 AND P_2):* $t_{QPB}(P) =$ `SELECT * FROM (` $t_{QPB}(P_1)$ `) p1 JOIN (` $t_{QPB}(P_2)$ `) p2 ON(` $joinCond(p1, p2)$ `)`
3. *If P is (P_1 OPT P_2):* $t_{QPB}(P) =$ `SELECT * FROM (` $t_{QPB}(P_1)$ `) p1 LEFT JOIN (` $t_{QPB}(P_2)$ `) p2 ON(` $joinCond(p1, p2)$ `)`
4. *If P is (P_1 UNION P_2):* $t_{QPB}(P) =$ `(` $align(t_{QPB}(P_1), t_{QPB}(P_2))$ `) UNION (` $align(t_{QPB}(P_2), t_{QPB}(P_1))$ `)`
5. *If P is (P_1 FILTER R):* $t_{QPB}(P) =$ `SELECT * FROM` $t_{QPB}(P)$ `WHERE` $filter_f(R)$

The translation of the example of Figure 4 is depicted in Figure 7. The column groups are indicated here by the notion `(CG(t))`, where `t` is the RDF term or variable the column group was created for.

4.3 Query Execution

The SQL query created as outlined in the previous section can now be executed on the mapped database. The result set of this SQL query is then mapped into a SPARQL result set depending on the query type of the SPARQL query. Each row of a SQL result set produces for every column group an RDF term which

then can be used to create the SPARQL result set. In the case of an SPARQL SELECT query, for each projected variable the corresponding column group is used to generate the RDF terms. We use the result set for the query initially described in Figure 3 to illustrate the result set translation process. The following listing presents a result set snippet for the column group of variable ?*dep*. For brevity, literal columns are collapsed into to `dep_lits`.

```
dep_type|dep_datatype|dep_lits|dep_reslength|     dep_res_1     |dep_res_2
--------|------------|--------|-------------|-------------------|---------
       1|           1|    null|            2|http://comp.com/dep|        1
--------|------------|--------|-------------|-------------------|---------
```

According to `dep_type` the RDF term is a resource. The IRI of the resource is generated from 2 columns, indicated by `dep_reslength`. The IRI is constructed by concatenating the prefix from `s_res_1` with the percent-encoded[6] value from `dep_res_2`. The SPARQL result set corresponding to the sample query is consequently:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head><variable name="dep"/></head>
  <results> <result>
      <binding name="dep"><uri>http://comp.com/dep1</uri></binding>
  </result> </results>
</sparql>
```

4.4 Implementation and Optimizations

We implemented the mapping candidate selection, query translation and query execution in SparqlMap, a standalone SPARQL-to-SQL rewriter. It utilizes ARQ[7] for SPARQL parsing and Java servlets for exposing a SPARQL endpoint. Our implementation is available under an open-source license on the project page[8]. We implemented several optimizations for achieving both better SQL query runtime and efficient mapping candidate selection. After parsing the query filter expressions are pushed into the graph patterns as describe in [10]. Further, when searching for mapping candidates, the patterns are grouped according to their subject for reducing search time. As described in [7] we flatten out the nested SQL structure described in Section 4.2 with the goal of minimizing self-joins.

5 Evaluation

We evaluated SparqlMap using the *Berlin Sparql Benchmark* (BSBM) [4]. We selected BSBM because of its widespread use and the provision of both RDF and relational data with corresponding queries in SPARQL and SQL. As a baseline

[6] as defined in `http://tools.ietf.org/html/rfc3986`
[7] `http://jena.apache.org/documentation/query/`
[8] `http://aksw.org/projects/SparqlMap`

tool *D2R* [3] was chosen since D2R is the most mature implementation and also implemented as standalone application, interacting with a remote database. As outlined in Section 2, D2R performs only a part of the operations of a SPARQL query in the database. Therefore multiple SQL queries will be issued for a single SPARQL query.

Setup. We used a virtualized machine with three AMD Opteron 4184 cores and 4 GB RAM allocated. Data resided in all cases in a *PostgreSQL* 9.1.3 database[9], with 1 GB of RAM. Additional indices to the BSBM relational schema were added, as described in the BSBM results document[10]. Both D2R and SparqlMap were allocated 2 GB of RAM. We utilized D2R version 0.8 with the mapping provided by BSBM[11]. All benchmarks employed the explore use case and were performed on a single thread with 50 warm-up and 500 measurement runs. In our evaluation we performed two different runs of BSBM. In the first run the SPARQL endpoints of SparqlMap, D2R and D2R with fast mode enabled (D2R-fast) were tested. Additionally, the PostgreSQL database was directly benchmarked. Due to D2R running out of memory when executing query 7 and 8[12] with 100M triples we additionally run the benchmark without Q7 and Q8. In a second run, we compare the efficiency of the generated SQL by recording the SQL generated by SparqlMap during a benchmark session and executing the SQL directly.

Overall runtimes. The results of the first run are presented for each query in Figure 8, the total runtime of this run is depicted in Figure 9 (b). A first observation is the performance advantage of SQL over SPARQL. Both D2R and SparqlMap are Java applications and therefore add overhead to the queries. Especially simple select queries like Q2 or Q11 with short execution times for BSBM-SQL show extreme differences compared to the SPARQL-to-SQL rewriters. We therefore focus on the discussion of the first benchmark run on a comparison between D2R and SparqlMap. For the 1M and 25M triple dataset SparqlMap outperforms D2R-fast in the vast majority of the queries. Comparing the total runtime, as presented in Figure 9 (b), for the 1M dataset SparqlMap is overall 5 times faster than D2R-fast and for the 25M dataset SparqlMap is overall 90 times faster than D2R-fast. It can be clearly seen, that SparqlMap outperforms D2R-fast by at least an order of magnitude and has superior scaling characteristics. For larger data sets (i.e. above 1M triples) SparqlMap's performance is also relatively close to the one of handcrafted SQL. It is noteworthy, that the huge difference in overall runtime can be attributed mainly to the slow performance of D2R-fast in Q7 and Q8.

[9] http://www.postgresql.org/

[10] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/index.html#d2r

[11] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/V2/results/store_config_files/d2r-mapping.n3

[12] Cf.: http://sourceforge.net/mailarchive/message.php?msg_id=28051074

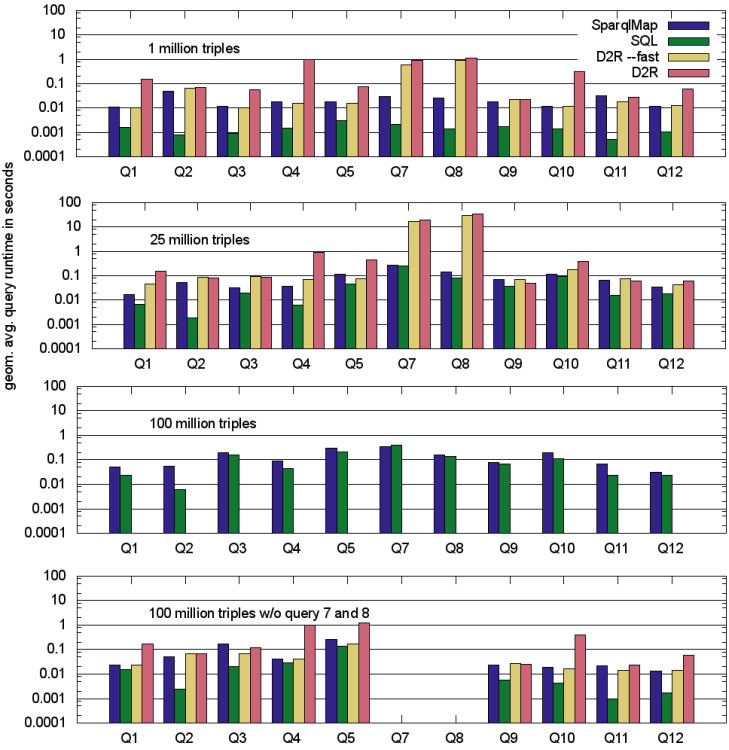

Fig. 8. Berlin SPARQL Benchmark evaluation with 1M, 25M and 100M triples, comparing SparqlMap, D2R with and without fast mode and native SQL. Values are average runtimes of queries in seconds for each query on a log scale.

Query specific performance. In general, for queries 1, 2, 3, 4, 5, 9, 10, 11 and 12 SparqlMap, D2R and D2R-fast show performance figures in the same order of magnitude. However, when examining the SQL generated by D2R and SparqlMap and relate it to their performance as presented in Figure 8 we can identify three categories.

1. Queries, which both mappers translate into a single SQL query with a similar runtime (Q1, Q5, Q10, Q12). These queries scale well for both mappers and the differences towards SQL decreases as the constant overhead imposed by the mapping contributes less to query execution time.
2. Queries that D2R maps into multiple SQL queries, that still yield a performance similar to SparqlMap (Q2, Q3, Q4, Q9, Q11). These queries however retrieve result sets of moderate sizes and may therefore in general scale worse than queries executed in a single SQL query. An example for this is Q3, where the ratio of the average execution time for SparqlMap and D2R-fast increases from 0.84 for the 1m dataset to a factor of 2.84 for the 25m dataset.
3. Queries mapped by D2R into multiple queries with low selectivity yielding huge intermediate result sets (Q7, Q8). In comparison with SparqlMap the

queries show poor scalability and performance. The average execution time ratio for Q7 between SparqlMap and D2R increases from 20 for the 1m dataset to 60 for the 25m dataset.

SQL performance comparison. The second benchmark run compares the SQL generated by SparqlMap (SparqlMap-SQL) with the SQL queries defined for BSBM (BSBM-SQL). This second run allows a direct comparison of efficiency between the handcrafted BSBM-SQL and the SQL generated by SparqlMap. The benchmark compares pure SQL efficiency, cutting out the overhead imposed by finding and translating the queries and is depicted in Figure 9 (a). In general we can observe that SparqlMap adds some overhead compared to the BSBM-SQL. SparqlMap generates for Q2 structurally similar SQL compared to the one of BSBM-SQL. The queries of SparqlMap-SQL, however, contain additional information required to generate RDF terms, such as the datatype, and therefore result in a twice as high average query runtime when compared to BSBM-SQL. A more drastic example is Q12, where BSBM-SQL is approximately 3.6 times faster than SparqlMap-SQL, or Q3 which takes on average 3.8 times longer in SparqlMap. Due to the large amount of tables used in this query, the overhead is more significant. In Q1 we observe that SparqlMap-SQL is slightly faster than BSBM-SQL (factor 1.1). Also in Q5 SparqlMap-SQL slightly outperforms BSBM-SQL by a factor of 1.2. We analyzed this difference and attribute it to the nested queries in BSBM-SQL. Especially the multiple nesting of subqueries in BSBM-SQL Q5 presents a challenge for the PostgreSQL query optimizer. The flat SQL structure generated by SparqlMap performs better here. Q11 is an example where SparqlMap-SQL performs significantly worse than BSBM-SQL (factor 34.0). The unbound predicate of Q11 leads SparqlMap to create a SQL query containing multiple unions of subselects. The lean structure of BSBM-SQL explains this gap.

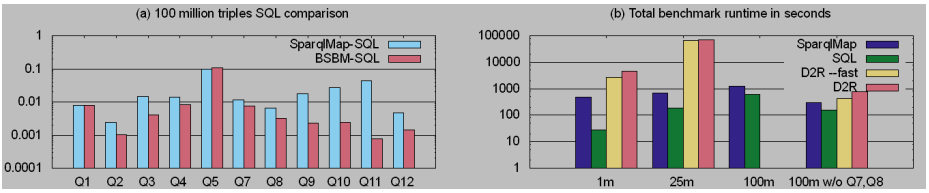

Fig. 9. (a) Comparison of BSBM-SQL with the SQL generated by SparqlMap for the 100M dataset. Average runtime in seconds on a log scale. (b) Total benchmark runtime in seconds on a log scale.

6 Conclusion and Outlook

In this paper we presented a formalization of the mapping from relational databases to RDF, which allows a translation of SPARQL queries into single unified

SQL queries. The benchmarking of our implementation shows clearly, that such an efficient mapping into a single query leveraging the SQL query optimizer and processor during query execution and results in significantly improved scalability when compared to non-single-SQL RDB2RDF implementations. In future work, we aim to improve the support for R2RML. We will also add further optimizations, such as support for the SPARQL `REDUCED` construct, which can boost the execution of certain queries. The query generation overhead can be substantially reduced by enabling prepared SPARQL queries, where a SPARQL query template is already precompiled into the corresponding SQL query template and subsequently reoccurring queries using the template do not have to be translated anymore. During our evaluation, we gained the impression, that BSBM is not best suited for benchmarking RDB2RDF mapping approaches. Certain features that are particularly challenging for an RDB2RDF tool (such as queries over the schema) are not part of BSBM. We plan to perform additional benchmarks and also to evaluate SparqlMap with large-scale real-world data.

References

1. Mapping relational data to rdf with virtuoso's rdf views, `http://virtuoso.openlinksw.com/whitepapers/relational%20rdf%20views%20mapping.html`
2. Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., Aumueller, D.: Triplify - lightweight linked data publication from relational databases. In: 18th International World Wide Web Conference, p. 621 (April 2009)
3. Bizer, C., Cyganiak, R.: D2r server – publishing relational databases on the semantic web. Poster at the 5th Int. Semantic Web Conf., ISWC 2006 (2006)
4. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. Int. J. Semantic Web Inf. Syst. 5(2), 1–24 (2009)
5. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving sparql-to-sql translation. Data and Knowledge Engineering 68(10), 973–1000 (2009)
6. Cyganiak, R.: A relational algebra for SPARQL. Technical report, Digital Media Systems Laboratory, HP Laboratories Bristol (2005)
7. Elliott, B., Cheng, E., Thomas-Ogbuji, C., Ozsoyoglu, Z.M.: A complete translation from sparql into efficient sql. In: Int. Database Engineering & Applications Symp., IDEAS 2009, pp. 31–42. ACM (2009)
8. Lu, J., Cao, F., Ma, L., Yu, Y., Pan, Y.: An Effective SPARQL Support over Relational Databases. In: Christophides, V., Collard, M., Gutierrez, C. (eds.) SWDB-ODBIS 2007. LNCS, vol. 5005, pp. 57–76. Springer, Heidelberg (2008)
9. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. ACM Trans. Database Syst. 34(3), 16:1–16:45 (2009)
10. Schmidt, M., Meier, M., Lausen, G.: Foundations of sparql query optimization. In: 13th Int. Conf. on Database Theory, ICDT 2010, pp. 4–33. ACM (2010)
11. Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL Execution on Relational Data. Poster at the 10th Int. Semantic Web Conf., ISWC 2011 (2011)