Optimizing SPARQL-to-SQL Rewriting

Jörg Unbehauen
AKSW Research Group
University of Leipzig
Leipzig, Germany
unbehauen@informatik.uni-leipzig.de

Claus Stadler
AKSW Research Group
University of Leipzig
Leipzig, Germany
cstadler@informatik.uni-leipzig.de

Sören Auer
AKSW Research Group
University of Leipzig
Leipzig, Germany
auer@informatik.uni-leipzig.de

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 map relational data to the RDF data model and make Linked Data interfaces to the data available. We can distinguish two main approaches: First, the database can be transformed into RDF row by row and the resulting knowledge base can be exposed using a triple store. Second, a mapper exposes a virtual graph, based on the relational database and performs SPARQL-to-SQL rewriting. The key challenge of such a SPARQL-to-SQL rewriting is to create an SQL query which can be efficiently executed by the optimizer of underlying relational database. In this article we discuss and evaluate the impact of different optimizations on query execution time using SparqlMap, a R2RML compliant SPARQL-to-SQL rewriter. We also compare the overall performance with other state-of-the-art systems.**

I. Introduction

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

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 RDBMS are still a magnitude faster than RDF triple 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 describe the optimizations required for achieving performance on par with native RDF stores. We implemented these optimizations in *SparqlMap*[1], a SPARQL-to-SQL rewriter based on the specifications of the *W3C R2RML working group*[2]. 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

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

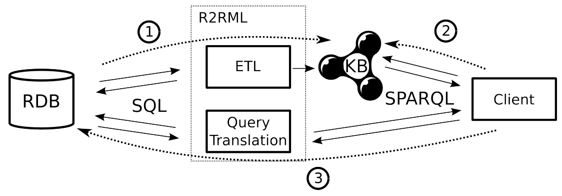


Fig. 1: Two models for mapping relational data to RDF: query rewriting and RDF extraction.

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 SPARQL query into a corresponding SQL query, which renders exactly the same results as the SPARQL query being executed (3).

Through a wide range of SPARQL-to-SQL query translation optimizations we are able achieve performance close to the native SQL case. Our library of optimizations includes filter optimization; self-join, optional-self-join and self-union elimination as well as filter and projection pushing. Our extensive evaluation demonstrates the impact of these optimizations and shows that SparqlMap outperforms the state-of-the-art in RDB2RDF mapping tools. Our implementation is mature, tested in a variety of real-world use cases and available for download and extension.

The remainder of this article is structured as follows. First, we present in Section II the basic concepts for mapping relational databases to RDF. Consecutively, we revisit the query translation process of SparqlMap in Section III. The optimizations are then presented in Section IV and evaluated in Section V. Then, we present the related work in Section VI. Finally, we draw a conclusion in Section VII.

II. Mapping Concepts

In this section we introduce the formalisms which are required to describe the mapping and translation process. These concepts are presented in greater detail in [12]. For a more general overview over the semantics of SPARQL we refer to [9]. For representing relational concepts use the notions of [2]. In general, we rely on the concepts defined in the *R2RML specification*[3].

Term map A *term map* is a tuple $tm = (A, ve)$ consisting of a set of relational attributes U from a single relation R and a value expression ve that describes the translation of U 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.

Term maps are the base element of a mapping. An example for such a *term map* is the template `http://comp.com/emp{id}` which creates URIs by concatenating the string "http://comp.com/emp" with the content of the attribute "id".

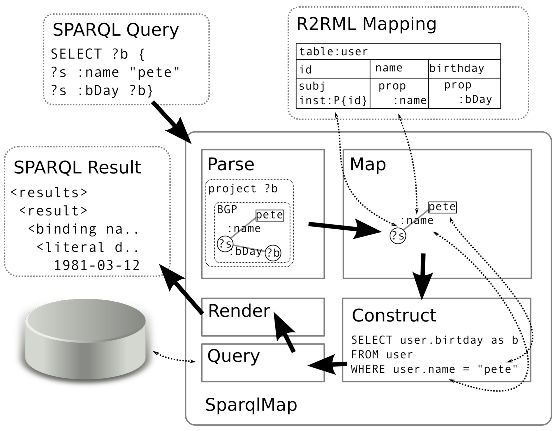
Triple map A triple map *trm* is the triple $(\text{tm}_S, \text{tm}_P, \text{tm}_O)$ of three *term maps* for generating the subject (position s), predicate (position p) and object (position o) of a triple. All attributes of the three *term maps* must originate from the same relation R.

A triple map defines how triples are actually generated from the attributes of a relation (i.e. rows of a table). In R2RML besides relations views defined by a SQL query can be used. In R2RML terms we refer to both as *logical table*.

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.

III. Query Translation with SparqlMap

In the following we give a short overview of our approach. The prerequisite for rewriting SPARQL queries to SQL is a set of mapping definitions m over which queries should be answered.



```
SELECT ?b {
?s :name "pete"
?s :bDay ?b}
```

table:user		
id	name	birthday
subj inst:P{id}	prop :name	prop :bDay

```
<results>
 <result>
  <binding na..
   <literal d..
    1981-03-12
```

```
SELECT user.birtday as b
FROM user
WHERE user.name = "pete"
```

Fig. 2: The different steps in translating a simple SPARQL query into an SQL query.

The process of rewriting a query on a mapping is performed in the following steps:

Parsing The initial step, the SPARQL query is transformed into an algebraic representation that allows further optimizations and transformations, which is depicted in Figure 2.

Mapping candidate selection This step 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. The relation between the candidate mappings and the triple patterns is called a binding. Initially, all triples of a query are bound to all triple maps of m. By examining how the triple patterns are processed in the query, for example if they are joined or filtered, the bindings are reduced to the triple maps, that can fullfill these operations. For example, by performing a so called *binding merge* [12] the join operation can be pre-evaluated, ensuring that only triple maps that can contribute to the SPARQL query result are used for the translation process. Here the $range(tm)$ of a term map is evaluated to check if the term maps are compatible, i.e. if they can at least theoretically produce the same RDF terms.

Construction of a query translation The identified candidate mappings and the obtained bindings enable us to rewrite a SPARQL query to an SQL query. In *binding merge* [12] we devise a recursive approach to query generation based on the query evaluation presented in [9]. Central for better understanding the optimizations presented later in the article is 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.

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.

A simple SPARQL query and its recursive translation is depicted in Figure 3.

Query execution an rendering Finally, the SPARQL result set is constructed from the SQL result set of the executed SQL query.

IV. SQL and Optimizations

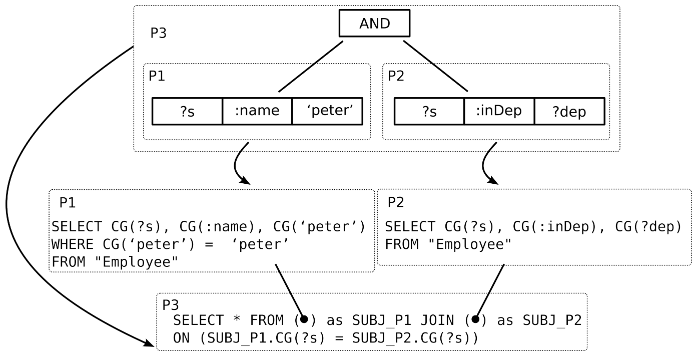
In this section we give an overview of the the optimizations implemented in SparqlMap. We focus on the optimizations for the generation of efficient SQL. SparqlMap utilizes *ARQ*[4] for SPARQL parsing and Java servlets for exposing a SPARQL endpoint. Our implementation is available under an open-source license on the project page[5].

In a previous evaluation we concluded that the general overhead introduced by SparqlMap, like HTTP handling,

[3]http://www.w3.org/2001/sw/rdb2rdf/r2rml/

[4]http://jena.apache.org/documentation/query/

[5]http://aksw.org/projects/SparqlMap



```
P1
SELECT CG(?s), CG(:name), CG('peter')
WHERE CG('peter') =  'peter'
FROM "Employee"
```

```
P2
SELECT CG(?s), CG(:inDep), CG(?dep)
FROM "Employee"
```

```
P3
 SELECT * FROM (●) as SUBJ_P1 JOIN (●) as SUBJ_P2
 ON (SUBJ_P1.CG(?s) = SUBJ_P2.CG(?s))
```

Fig. 3: Translation of a simple SPARQL query into a nested SQL query.

query parsing or connection management by far outweighs the processing needs of the mapping component. While with further enhancements the overhead of SparqlMap could certainly be lowered, especially for long running queries, the query execution time is determined by the efficiency of the generated SQL query.

In the following, we list several optimizations for which we explain the motivation, the approach and possible limitations.

In SQL representations of queries we refer with `cg_o` to the column group created by translating the term maps of the binding for `o`.

A. Filter Optimization

Problem Translating SPARQL operators in an automatic, naive approach leads to ineffective SQL operators. For example, comparing the equality between a template based term map and a given URI would, simplified, result in SQL equals expression as in Figure 4b. This expression would require the database to first materialize the string and then compare it, thus preventing the usage of exiting indexes.

Approach By decomposing URIs into fragments according to the template of the term map, the column values can be used directly. For the given sample this is depicted in Figure 4c.

Limitations An unsolved problem is the comparison of IRIs that require different modes of URL encoding. Given an IRI that is derived from an attribute without encoding, for example, an attribute "homepage", and comparing it for equality to an template based IRI, the evaluation requires URL-encoding of the column values used in the template. Database specific solutions for URL encoding can be used for alleviating this problem as well as implementing string replacement for the most prevalent characters. However, a generic solution for this problem has yet to be found.

B. Self-Join Elimination

Problem Following the approach described in Section III for a SPARQL query a nested SQL query is generated. SPARQL queries over mapped databases are often star-shaped, reference multiple columns of an underlying table and the generated SQL query is likely to contain self-joins. An example of such a query is shown in Figure 5a.

Approach As described in [7] we flatten out the nested SQL structure described in Section III with the goal of minimizing self-joins. For this example we assume that the query maps to a single table and would therefore generate an SQL query as in Figure 5b. This query, however, is not optimal because of a superfluous self-join. For a star shaped query, we can merge the bindings of triple patterns that belong to the same logical table, instead of nesting subselects, these can be transformed into a single join as depicted in Figure 7c.

Limitations We currently perform self-join elimination as part of the candidate mapping selection. While this approach eases implementation, it does not work for all possible cases where self-joins occur. For example, this approach lacks optimizations in the face of multiple mappings of a single table. In the future we will add self-join elimination on the SQL algebra level for a truly generic solution.

C. Optional-self-join Elimination

Problem `OPTIONAL` triple patterns are transformed into a left-joined subquery, for example the query in Figure 6a is translated into that of Figure 6b. This increases the amount of self-joins, in this case left-self-joins.

Approach Similar to self-join elimination, some left-joins can be equivalently transformed into an extension of the query's projection, as shown in Figure 6c.

Limitations This optimization can only be applied to `OPTIONAL` patterns containing a single triple pattern for which there exists only a single candidate triple map.

D. Self-Union Elimination

Problem Translating triple patterns that bind to more than one mapping require the construction of a union. Depending on the capabilities of the database, this requires the planner to perform multiple select on the same table and union them together. A sample SPARQL query is depicted in Figure 7a and its naive translation in Figure 7b.

Approach We group all select statements of a logical table together into a single select statement, project the requested columns and remove the `NOT NULL` condition normally associated. The resulting query Figure 7c therefore lacks any union operations, while producing the data.

Limitations In contrast to the previously described optimizations the result set structure is changed. Consequently, this optimization can only be used, if result size, ordering of elements and or aggregate functions are not being used.

E. Projection Pushing

Problem Consider the SPARQL query in Figure 8a over the exemplary mapping. Using a naive approach, this query will be translated into a series of unions, as denoted in Figure 8b. The resulting query's execution plan makes use of full-table scans and therefore only offers lackluster performance. This SPARQL query, however, is prototypical for the exploration of schemas in RDF data. For this reason, efficient support of such types of queries is crucial for the usability of SPARQL-to-SQL rewriting systems.

(a) SPARQL query

```
SELECT * {
x:Person1 :p1 ?o.
}
```

(b) Query with filter on concatenated expression

```
SELECT
  cg_o
FROM view1 v1 WHERE ( ('x:Person' || v1.a1)
      = 'x:Person1')
```

(c) Query with decomposed filter expressions.

```
SELECT
  cg_o
FROM view1 v1 WHERE ( ('x:Person' =  'x:
     Person') AND (v1.a1 = '1'))
```

Fig. 4: Filter optimization queries

(a) SPARQL query

```
SELECT * {
?s :p1 ?v1.
?s :p2 ?v2.
}
```

(b) Nested queries

```
SELECT
  cg_s, cg_p, cg_o
FROM view1 v1
JOIN (
  SELECT cg_s, cg_p2, cg_o
  FROM view1 ) as s1 on(s1.cg_s = v1.cg_s)
```

(c) Query without self-join

```
SELECT
 cg_s, cg_p, cg_o, cg_p2, cg_o2
FROM view1
```

Fig. 5: Self-join elimination queries

(a) SPARQL query

```
SELECT * {
?s :p1 ?v1.
OPTIONAL {
  ?s :p2 ?v2.
  }
}
```

(b) Nested queries

```
SELECT
  cg_s, cg_p1, cg_o1, cg_p2, cg_o2
FROM view1 v1
LEFT JOIN (
  SELECT cg_s, cg_p2, cg_o2
  FROM view1 ) as s1 on(s1.cg_s = v1.cg_s)
WHERE cg_s NOT NULL AND  cg_p1 NOT NULL AND
       cg_o1 NOT NULL
```

(c) Query without self-join

```
SELECT
 cg_s, cg_p1, cg_o1, cg_p2, cg_o2
FROM view1
WHERE cg_s NOT NULL AND  cg_p1 NOT NULL AND
      cg_o1 NOT NULL
```

Fig. 6: Self-join elimination queries

(a) SPARQL query

```
SELECT * {
:Person1 ?p ?o.
}
```

(b) Stacked unions

```
SELECT *
FROM (
    (SELECT cg(v1.c1) as s, cg(:p1) as p, cg
         (v1.c2) as o FROM view1 v1)
   UNION ALL
    (SELECT cg(v1.c1) as s, cg(:p2) as p ,
         cg(v1.c3) as o FROM view1 v1)
    ........
) as u1
```

(c) Union flatted into join

```
SELECT cg(v1.c1) as s, cg(:p1) as p_1, cg(
     v1.c2) as o_1, cg(:p2) as p_2 , cg(v1.
     c3) as o_2 FROM view1 as v1
```

Fig. 7: Self-union elimination

Approach For the only projected variable ?p, solely constant-value term maps are found. Due to the DISTINCT modifier, we know that it is sufficient to only collect the distinct constants without actually scanning the logical table. Yet, we also have to consider the case that no constants are produced from empty tables. Our approach is to push the projection into the subqueries and prevent a table scan by adding a LIMIT clause to the SQL query, resulting in a query as shown in Figure 8c.

Limitations This optimization can only be applied to DISTINCT queries with only constant-valued term maps bound to projected variables.

F. Filter Pushing

Problem During a bottom up evaluation of a SPARQL query's algebra tree representation, filter operations on higher levels to do not contribute to the reduction of the selectivity at lower levels. For this reason, query evaluation may encounter unnecessarily large intermediate results.

Approach After parsing the query filter expressions are pushed into the graph patterns as described in [10]. This is an optimization directly performed on the algebraic tree representation of the SPARQL query.

Limitations No limitations apply.

(a) SPARQL query

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

(b) Union with full projection subqueries

```
SELECT DISTINCT
  cg_p
FROM (
   (SELECT cg(v1.c1) as s , cg(:p1) as p,
       cg(v1.c2) as o FROM view1 v1)
   UNION ALL
   (SELECT cg(v1.c1) as s, cg(:p2) as p ,
       cg(v1.c3) as o FROM view1 v1)
   ........
) as u1
```

(c) Subqueries with pushed projection and added limit

```
SELECT DISTINCT
  cg_p
FROM (
   (SELECT cg(:p1) as p FROM view1 v1 LIMIT
       1)
   UNION ALL
   (SELECT cg(:p2) as p FROM view1 v1 LIMIT
       1)
   ........
) as u1
```

Fig. 8: Pushing projections into subselects

(a) Filter on UNION

```
SELECT * {
  {?x :p1 ?z}
   UNION
  {?x :p2 ?z}
  FILTER (?x = :Person1)
}
```

(b) Filter pushed into UNION

```
SELECT * {
  {?x :p1 ?z
  FILTER (?x = :Person1)
       }
   UNION
  {?x :p2 ?z
  FILTER (?x = :Person1)
       }
}
```

Fig. 9: Filter pushing

V. Evaluation

We evaluated SparqlMap using the *Berlin SPARQL Benchmark* (BSBM) [5]. We selected BSBM because of its widespread use and the provision of both RDF and relational data with corresponding queries in SPARQL and SQL. BSBM actually provides two benchmarks, namely *Explore* and *Business Intelligence*, however only the former offers the resources needed to benchmark SPARQL SQL rewriters. As a baseline tool *D2R* [4] was chosen because its popularity and still ongoing development efforts.

As outlined in Section VI, 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.

We used an Intel i5-3570K quad-core with 4GB RAM, running CentOS 6.2. The RDBMS used is *PostgreSQL* 9.1.3 with 1GB of RAM allocated. Additional indexes to the BSBM relational schema were added, as described in the BSBM results document[6]. All testing was performed on a 50 million triple database. We utilized D2R version 0.8.1 with the mapping provided by on the BSBM website[7] with all optimizations enabled. For benchmarking D2R we omitted query 8, as with this configuration we faced out-of-memory problems[8]. Further, Virtuoso 6.1.6 was configured to use up to 2GB of RAM.

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

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

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

We evaluated SparqlMap and the optimizations in two steps. Firstly, we tested the optimizations individually, using the BSBM explore queries and a custom query. Secondly, we compared SparqlMap with other systems.

All benchmark runs based on BSBM explore were configured to use a single thread with 50 warm-up and 500 measurement runs. We additionally benchmarked one of our optimizations using a custom query, for which no adequate query exists in BSBM explore. This query, referred to as `?p`-query, was was run 10 times and the results were averaged.

A. *Individual Evaluation*

In this section we discuss the impact of the optimizations on the query run time. The benchmark was repeatedly run with different combinations of optimizations enabled in order to determine each combination's effect on the overall runtime performance. The results of this benchmark run are presented in Figure 10.

Naive translation The benchmark run with all optimizations disabled is labeled "none" in Figure 10.

Self-join elimination In Figure 10 the benchmark run for the examination of this optimization is labeled "w/ selfjoin" The effect of self-join elimination can most prominently be seen in queries that retrieve multiple attributes from an relation and therefore exhibit a star-shaped query pattern. In these queries (1, 2, 3, 7, 8, 10, 12) the gain depends on the count of attributes selected and can result in an performance gain up to an order of magnitude, or in the case of queries 8 and 10 enable the execution. Consequently, queries with little or none join operations (4,5,9,11) unsurprisingly only benefit marginally or not at all. At first glance query 1 may be expected to benefit stronger from self-join elimination as the rewritten SPARQL query references multiple attributes. However, the mapping of three of the SPARQL query's predicates require joining with other relations that are not subject to self-join elimination.

Filter In Figure 10 "w/ filter" denotes the results of the benchmark run with solely the filter optimization turned on. In general, this is the single most effective optimization when compared to the naive translation, as for all queries the filter operation is accelerated. Additionally, the effect of the optimization varies with the type of filter. In Figure 10 "w/ filter and selfjoin" shows the query runtimes for both optimizations enabled. Again, all queries benefit from combining

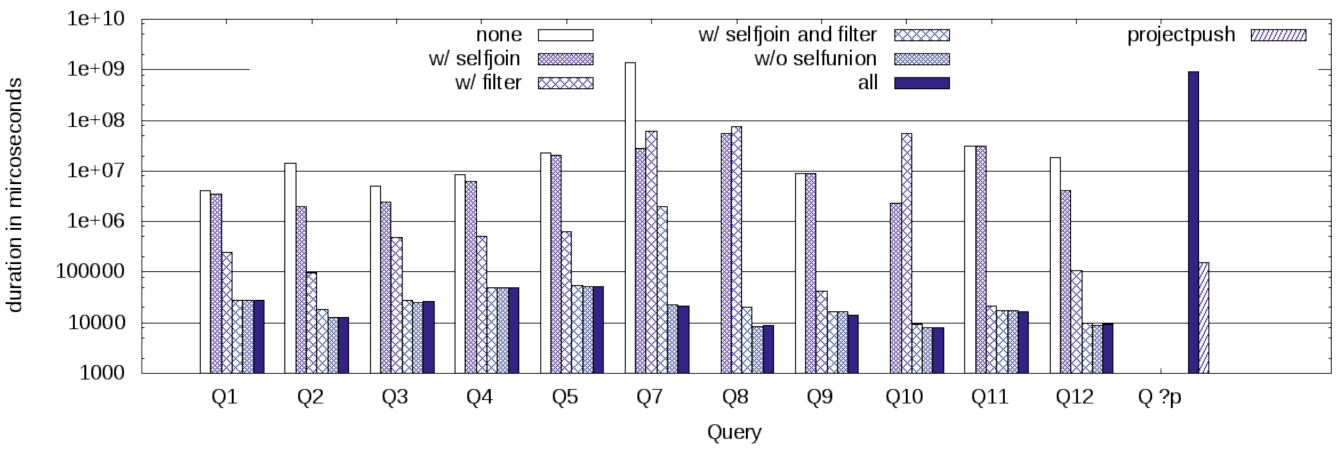


Fig. 10: Query run times in microseconds for SparqlMap with different optimization configurations on a logarithmic scale.

both optimizations, as it reduces the amount of joins and makes the join conditions faster to evaluate, albeit to a varying degree. For example, query 3 the gain in performance is less pronounced than in query 2. This shows that the elimination of ineffective self-joins is more efficient than the removal of optimized joins.

Optional join elimination The BSBM query mix contains several queries with *OPTIONAL*, namely the queries 2, 3, 7, 8. In Figure 10 “w/o deunion” the query runtimes for the combination of filter optimization, self-join elimination and optional join elimination is depicted. For query 2 and 8 minor improvements can be registered, as here 3, respectively 2 joins could be avoided. For query 3 no improvement could be registered, as the *OPTIONAL* could not be optimized. The by far greatest effect on the runtime can be observed for query 7 which features nested optional patterns. By eliminating one level of the nested optionals, PostgreSQL is able to perform a more efficient query execution that is about 90 times faster.

Self-Union Elimination In Figure 10 “all” and “w/o selfunion” we compare the effect of the self-union elimination. The only queries affected by this optimization are 9 and 11. For these queries, multiple candidate mappings of the same logical table apply to some of their triple patterns. The effect on query runtime however with a query execution time reduction of about 10% is smaller than expected. While the optimization certainly allows a more effective query plan, total execution time is however dominated by other operations in the query.

Projection Pushing As the BSBM query mix does not contain a query, that is affected by this optimization evaluated the query `SELECT DISTINCT ?p {?s ?p ?o}` against the BSBM data set. In Figure 10 the effect can be seen for “Q ?p”. Although this query is not found in the BSBM, this type of query is often used in real-world deployments, where for example a user wants to explore a dataset. With an average execution time of 130ms this query optimization allows the execution of such queries in a reasonable amount of time.

Filter pushing The filter pushing optimization had no measurable effect on the performance in BSBM. A closer examination of query execution plans reveals that PostgreSQL is performing this optimization in the query planner. Performing this optimizations has at least no effect on the queries of the BSBM, as the same execution plan is generated by PostgreSQL. We therefore refrained from depicted this benchmark run separately.

B. *Profiling*

In order to explain the discrepancy between native SQL and the translations from SPARQL we profiled a SparqlMap benchmark run by measuring the duration of function calls. The results are depicted in Figure 11. In an additional benchmark run we confirmed that switching off optimizations has only an impact on query execution time (i.e. does not affect other stages of the SPARQL SQL rewriting process), therefore we discuss only the profiling with all optimizations enabled here.

In general, the more query optimizations are enabled, the greater is the time consumed by this introduced overhead. This is particularly apparent for queries with a low overall response time, such as query 2. Here the mapping process consumes half of the total execution time. On the other hand, for the longer running queries like query 5, overhead and mapping only marginally affect the total execution duration. While focus of our current research is to find and optimal translation, for further accelerating already fast queries the technical platform becomes more important.

C. *Overall Evaluation*

In this section we compare SparqlMap with other systems. In summary, SparqlMap turned out to be fasted, being followed by Virtuoso and finally D2R. The total benchmark runtime is depicted in Figure 12. SparqlMap is still by a factor of 4 slower than PostgreSQL with Virtuoso requiring an additional 25% more time. D2R is by a factor of 80 slower than Virtuoso. We explain the benchmark result by further examining the results per query, as shown in Figure 13.

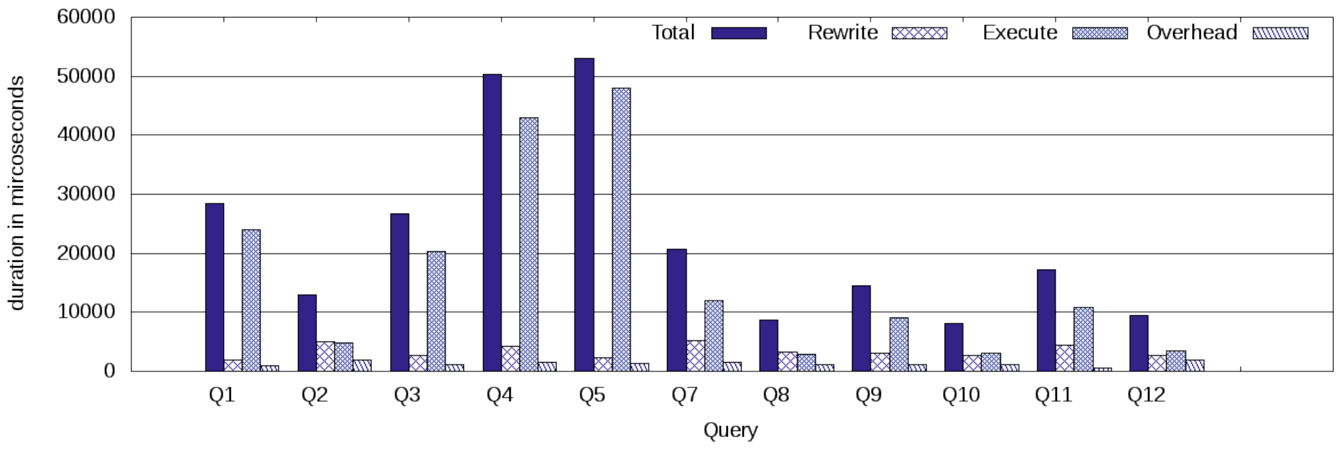


Fig. 11: Execution times of the query transformation steps in microseconds with all optimizations enabled. Rewrite includes the mapping and the query translation step. Overhead includes query parsing and resource management.

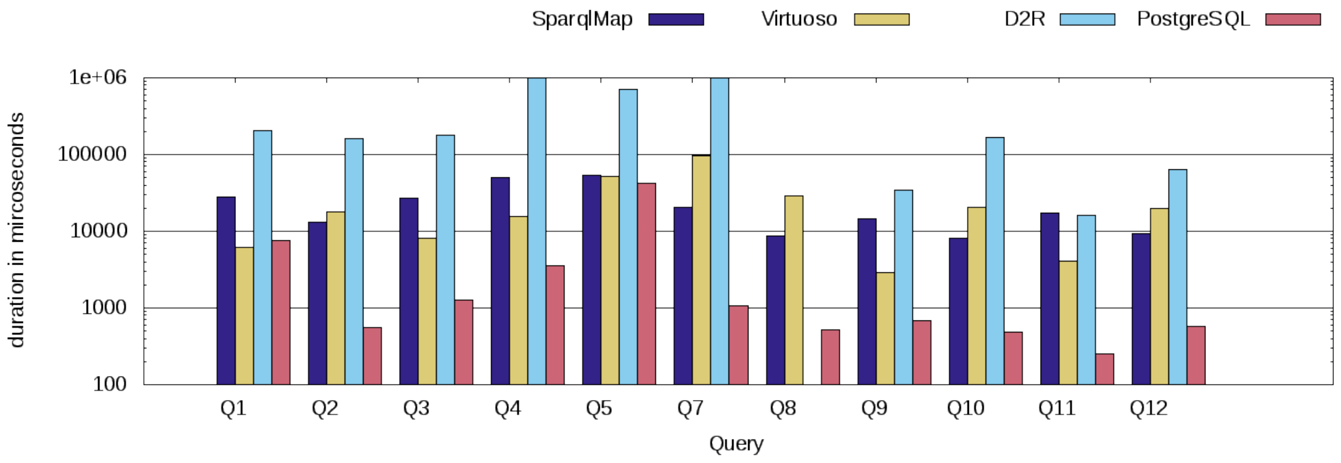


Fig. 13: Comparison of the query execution times in microseconds for different endpoints on a logarithmic scale using single threaded BSBM on a 50 million triples dataset.

With the exception of query 1 and 5 PostgreSQL is about an order of an magnitude faster than the SPARQL endpoints. With our previous profiling of SparqlMap, especially for fast queries this gap will remain open for as the overhead associated with SPARQL query handling begins to dominate query execution time here. Query 1 is interestingly close in terms of performance between PostgreSQL the other systems. The SQL representation of this query utilizes subqueries, the query translations however use flat joins. For query 1 this more effective query representation almost outweighs the penalties associated with SPARQL query processing. Query 5 on the other hand incorporates an expensive join, whose execution time dominates the total execution time of the query. Here the SPARQL overhead is negligible. Query 5 is for PostgreSQL the single most expensive query, contributing over 2/3 to total benchmark runtime.

Comparing SparqlMap and D2R shows that SparqlMap significantly outperforms D2R in all but the most basic queries. Especially queries that require D2R to performs operations with huge intermediate result set in memory are orders of magnitude slower than SparqlMap. Here the limitations of the hybrid in-database and out-of-database processing are clearly shown.

Further notable is that SparqlMap is able to outperform Virtuoso. A significant contribution to this is query 7, here the relational backed SparqlMap benefits from the reduced amount of joins needed.

VI. 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 [6], discussing the translation of a SPARQL query 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

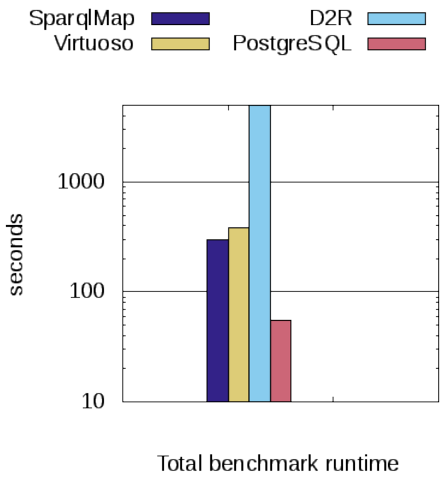


Fig. 12: Comparison of the total benchmark runtime in seconds on a logarithmic scale.

represent facets of RDF data in relational databases.

Second, the mapping of relational databases into RDF is a way of leveraging existing data on the Semantic Data Web. We can differentiate between RDB-to-RDF tools that simply aid one with the setup of a Linked Data interface and possibly with the creation dumps, and tools that expose a SPARQL endpoint for interactive querying of the data, which enables serving much more sophisticated use cases. An example for RDF and Linked Data exposition is *Triplify* [3]. 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* [4] 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 enabling SPARQL queries 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. While benchmark results are published and basic optimizations are described, Ultrawrap is at the time of writing not available for independent evaluation.

VII. Conclusion

In this paper we presented and evaluated optimizations in SPARQL-to-SQL rewriting that allow SparqlMap to compete or even surpass native triple stores in terms of SPARQL benchmark performance. We contributed by implementing and testing well-known optimizations like self-join elimination, but also introduce genuinely new optimizations like self-union-elimination. To the best of our knowledge this is the most thorough evaluation of a SPARQL-to-SQL rewriter published. For future work we aim at developing further optimizations. We currently investigate how to further reduce joins by examining logical tables that are backed by SQL statements. For example, in BSBM query 1 an unnecessarily high amount of joins is required, because the SQL query describing the logical table is used in form of an subselect. Also, as a consequence of our profiling efforts, performance optimizations will further not only target optimizing SQL. Query overhead can be reduced, first by simple engineering efforts. Secondly, precompiling recurring queries, as they are encountered while being deployed as a backend system for an application, can contribute to overhead reduction. Another open research question is to find the best integration scenario for SPARQL-to-SQL rewriters.

Acknowledgment

This work was partly supported by a grant from the European Union's 7th Framework Programme provided for the project LOD2 (GA no. 257943).

References

[1] Mapping relational data to rdf with virtuoso's rdf views. http://virtuoso.openlinksw.com/whitepapers/relational%20rdf%20views%20mapping.html.
[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[3] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify - light-weight linked data publication from relational databases. In *18th International World Wide Web Conference*, pages 621–621, April 2009.
[4] C. Bizer and R. Cyganiak. D2r server – publishing relational databases on the semantic web. Poster at the 5th Int. Semantic Web Conf. (ISWC2006), 2006.
[5] C. Bizer and A. Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst*, 5(2):1–24, 2009.
[6] A. Chebotko, S. Lu, and F. Fotouhi. Semantics preserving sparql-to-sql translation. *Data and Knowledge Engineering*, 68(10):973 – 1000, 2009.
[7] B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z. M. Ozsoyoglu. A complete translation from sparql into efficient sql. In *Int. Database Engineering & Applications Symp.*, IDEAS '09, pages 31–42. ACM, 2009.
[8] J. Lu, F. Cao, L. Ma, Y. Yu, and Y. Pan. Semantic web, ontologies and databases. chapter An Effective SPARQL Support over Relational Databases, pages 57–76. Springer-Verlag, Berlin, Heidelberg, 2008.
[9] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, Sept. 2009.
[10] M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *13th Int. Conf. on Database Theory*, ICDT '10, pages 4–33. ACM, 2010.
[11] J. F. Sequeda and D. P. Miranker. Ultrawrap: SPARQL Execution on Relational Data. Poster at the 10th Int. Semantic Web Conf. (ISWC2011), 2011.
[12] J. Unbehauen, C. Stadler, and S. Auer. Accessing relational data on the web with sparqlmap. In T. Hideaki et al., editors, *Proc. 2nd Joint International Semantic Technology Conference*, volume 7774 of *Lecture Notes in Computer Science*. Springer, 2012.