Optique

Project N°: **FP7-318338**

Project Acronym: **Optique**

Project Title: **Scalable End-user Access to Big Data**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

Deliverable D6.2
Transformation System Configuration Techniques

Due date of deliverable: (T0+24)

Actual submission date: November 3, 2014

Start date of the project: **1st November 2012** Duration: **48 months**

Lead contractor for this deliverable: **FUB**

Dissemination level: **PU** – Public

Final version

Executive Summary:
Transformation System Configuration Techniques

This document constitutes deliverable D6.2 of project FP7-318338 (Optique), an Integrated Project supported by the 7th Framework Programme of the EC. Full information on this project, including the contents of this deliverable, is available online at `http://www.optique-project.eu/`.

More specifically, the present deliverables describes the activities carried out and the results obtained in Task 6.1 of Optique. This task is concerned with the techniques for configuring the query transformation system that constitutes the core component of the Optique architecture, towards meeting user requirements in terms of scalable answering of SPARQL queries formulated over an ontology.

From the foundational point of view, we have obtained results regarding the various phases of the translation of user queries. We first concentrate on the role of mappings in the query answering process, and study what affects the efficient translation of SPARQL queries formulated over (virtual) RDF data, to SQL queries over the relational data sources to which the RDF data is mapped. We then address the challenges posed by the additional presence of an ontology formulated in the standard OWL 2 QL fragment.

From the implementation point of view, we discuss how the user requirements, and specifically those coming from the Optique use cases have shaped the features with which the query transformation system has been extended. Several of these features are tightly related to the integration in the Optique Platform. We have then optimized the reasoning tasks at the level of the TBox, and have added functionalities for checking the consistency of the ontology, and for verifying the emptiness of classes and properties. A further extension of *Ontop* with spatial features has been carried out and is reported.

We observe that the above results reported in this deliverable, are complemented by those already reported in D6.1 (*WP6 Year 1 Progress Report*), which also refers to activities carried out in Task 6.1.

List of Authors

Konstantina Bereta (UoA)
Elena Botoeva (FUB)
Diego Calvanese (FUB)
Benjamin Cogrel (FUB)
Davide Lanti (FUB)
Martin Rezk (FUB)
Sarah Komla-Ebri (FUB)
Guohui Xiao (FUB)

Contents

1 Introduction **4**

2 Foundational Results on OBDA **7**
2.1 Efficient SPARQL-to-SQL with R2RML Mappings . . . 7
2.1.1 Experiments . . . 8
2.2 Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime . . . 9
2.2.1 Evaluation . . . 10

3 Implementation Development **12**
3.1 Integration with the Optique Platform . . . 12
3.1.1 Integration of the Optique R2RML API . . . 13
3.1.2 SQL Support in the Mapping Language . . . 13
3.1.3 SQL Multi-schema Support . . . 14
3.1.4 SPARQL Support Extension . . . 15
3.1.5 Sesame API Upgrade . . . 16
3.2 Novel Reasoning Functionalities . . . 16
3.2.1 Efficient TBox Reasoning via DAG Manipulation . . . 16
3.2.2 Consistency Checking . . . 17
3.2.3 Checking for Empty Classes and Properties . . . 17
3.3 Extending *Ontop* with Spatial Features . . . 19
3.3.1 Supported Spatial Features . . . 20
3.3.2 Future work . . . 22
3.4 Releases . . . 22
3.4.1 Maven . . . 22
3.4.2 Released Versions . . . 23

Bibliography **23**

A Efficient SPARQL-to-SQL with R2RML Mappings (JWS Paper) **26**

B Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime (ISWC 2014 Paper) **63**

Chapter 1

Introduction

We start by recalling the main ideas behind the Query Transformation module in Optique, which is being developed in WP6, exploiting state-of-the-art query translation techniques for query answering in Ontology-Based Data Access (OBDA) systems. In OBDA, the aim is to query data sources, which typically are databases managed by a relational database management system (but more in general might be also graph structured data sources or triple stores), through an ontology over which queries are formulated, and which provides a conceptual high-level representation of the domain of interest and of the information stored in the data source. The relationship between the information maintained by the ontology and the data stored at the sources is specified in a declarative way by means of a set of mappings, where in general each of these mappings relates a query over the data sources to a query over the ontology. The OBDA system makes use of the logical axioms that constitute the ontology and of the mappings between the ontology and the data sources to *transform* queries that users express in terms of the ontology into queries that can be directly handled by the system managing the data layer. The ides is to compile into the resulting query both the ontology axioms and the mapping assertions, so that it suffices to evaluate such a query over the data sources to obtain the answer to the original query formulated by the user.

OBDA has been investigated extensively in the past years under the assumption that ontologies are expressed in fragments of OWL 2 (which is the standard language for which Description Logics (DLs) [2] provide the formal counterpart). Specifically, in a context like the one of the Optique project, where one needs to access large amounts of data, it has been shown that the OWL 2 QL fragment of OWL 2 (corresponding to the *DL-Lite* family of DLs [7, 1]) provides a very good tradeoff between expressive power in the ontology language and computational complexity of query answering, specifically when measured in terms of the size of the data (i.e., for *data complexity*). Moreover, proposals for using mappings to overcome the impedance mismatch between the (relational) data sources storing values and the ontology maintaining abstract objects have been devised, and techniques have been studied for compiling the mapping assertions into the query over the data sources [17, 6].

However, transferring the good theoretical results about (complexity of) query answering to a practical setting like the one of Optique, where ontologies are potentially very large, mappings are numerous and have a complex structure, and we are in the presence of big data, turned out to be rather challenging. In Optique wP6, we are concerned with the development of techniques for query transformation in OBDA, that on the one hand make the whole transformation process efficient, and on the other hand (and even more importantly) produce SQL queries that can be evaluated efficiently over the underlying data sources by the query execution layer (cf. WP7). In addition, such techniques are to be implemented in a highly efficient query transformation engine, that constitutes the core of the Optique Platform.

In Task 6.1, the aim is to configure the various components of the query transformation system so as to make efficient query transformation possible. This involves several specific design choices for the various levels of the query transformation system, most importantly the features supported in the mapping language, the expressive power of the user query language, and the inference done at the level of the ontology. The choices are guided by the requirements coming from users of the OBDA system, and specifically from the

Optique use cases. In Task 6.1, we proceeded in parallel with Task 6.2 "*Runtime Query Rewriting*", about which we will report in Deliverable D6.3 at the end of month 36 of the Optique project.

In this deliverable, we describe the main contributions that have been provided in Task 6.1 in Year 2 of Optique towards the above objectives, and that we overview in the remaining part of this chapter. We observe that the results reported here, are to be complemented with those already reported in D6.1 (*WP6 Year 1 Progress Report*), which also refer in part to the activities that have been carried out in Task 6.1 in Year 1 of Optique, and that are not repeated here.

Overview of Contributions

The contributions we have provided are of two main kinds:

1. foundational results related to the design and configuration of the query transformation component, which improve the state-of-the-art in the area.

2. implementation of novel features in the *Ontop* OBDA system.

As for item 1, we have obtained results regarding the two phases of the translation of user queries. We first concentrate on the role of mappings in the query answering process, and study what affects the efficient translation of SPARQL queries formulated over (virtual) RDF data, to SQL queries over the relational data sources to which the RDF data is mapped (see Section 2.1 and Appendix A). Then we address the challenges posed by the additional presence of an ontology formulated in the standard OWL 2 QL fragment (see Section 2.2 and Appendix B).

As for item 2, we recall that the core of the query transformation component in the Optique Platform (cf. WP2) is provided by the *Ontop* system developed in the last years by FUB, and that has been brought by FUB as background to Optique. While *Ontop* implements state-of-the-art technology for OBDA, it was still not able to effectively cope with the challenges posed by the Optique use-cases. The main development effort, effectively carried out in Task 6.4 of WP6 in Year 2, continuing the work started in Year 1, has been directed towards the implementation of the techniques devised in Task 6.1 (and also Task 6.2), and towards a better integration in the general Optique Platform.

In WP6, we have worked in close interaction with Statoil and UiO. Indeed, a major effort has been the extension of the *Ontop* system with features that on the one hand were driven by the Optique use cases, and on the other hand were required to improve the conformance to W3C recommendations and/or to industrial standards. In particular, the features of the system have been configured in such a way that it is able to deal with the requirements that came up during the experimentation within the Statoil use case, both with the NPD Factpages and with the EDPS database. We describe below the main extensions that have been carried out, referring to Chapter 3 in the report where different activities and accomplishments related to this effort are described in mode detail.

Integration with the Optique Platform. Several adaptations were done to *Ontop* to better integrate it in the Optique platform. Moreover to improve also the support for standard languages and libraries, various extensions of the system functionalities have been implemented (see Section 3.1). Specifically, we carried out the following improvements of the system:

- support for the R2RML API of the standard R2RML mapping language (Section 3.1.1);

- support for extended SQL in the mapping language, to be able to accommodate, e.g., nested subqueries, and queries that carry out complex navigation over the data by means of regular expressions, both of which are required to capture the Statoil use case. (Sections 3.1.2);

- support for queries over multiple schemas (Section 3.1.3), which was required due to the fact that EPDS contains multiple schemas, each with several tables;

- improved support for the SPARQL query language (Section 3.1.4), and
- support for the OpenRDF Sesame API, and migration to the newest version of Sesame (Section 3.1.5).

Novel Reasoning Functionalities. We have optimized TBox reasoning and realized novel ontology reasoning functionalities that support the diagnosis of incorrect elements in the ontology and in the mappings, thus improving the design phase (cf. also Optique WP4) (Section 3.2). Specifically, we have provided a new efficient implementation of the TBox reasoner (Section 3.2.1), and we have implemented functionalities to check the consistency of the ontology with respect to the database and mappings (Section 3.2.2), and to check for empty classes and properties (Section 3.2.3). We have also developed an extension of *Ontop* with spatial features (Section 3.3).

Chapter 2

Foundational Results on OBDA

In this chapter, we describe some foundational results on OBDA that have been obtained in Optique in the context of WP6. These results have been published in prestigious international venues, and the corresponding publications are included as appendixes in this report. In this chapter, we provide a brief overview of the obtained results in terms of short summaries of the publications, and refer to the publications in the appendix for a comprehensive treatment of the presented results.

- Martin Rezk, Mariano Rodriguez-Muro.
  Efficient SPARQL-to-SQL with R2RML mappings. To appear in *Journal of Web Semantics* (JWS), 2014.

  The main results of this publication (which is included as Appendix A), are summarized in Section 2.1.

- Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyaschev.
  Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime. In *Proc. of the 13th Int. Semantic Web Conference* (ISWC), 2014.

  The main results of this publication (which is included as Appendix B), are summarized in Section 2.2.

2.1 Efficient SPARQL-to-SQL with R2RML Mappings

One of the most promising approaches for on-the-fly query answering over virtual RDF is query answering by query rewriting. That is, answer the queries posed by the user (e.g., SPARQL queries) by translating them into queries over the database (e.g., SQL). This kind of technique has several desirable features; notably, since all data remain in the original source there is no redundancy, the system immediately reflects any changes in the data, and well-known optimizations for relational databases can be used.

To be efficient in practice, the query rewriting technique must produce "reasonable" SQL queries, that is, queries that are not excessively large or too complex to be optimized by the DB engine. Thus, the query rewriting technique needs to tackle two different issues: *(i)* a query translation problem that involves RDB-to-RDF mappings over arbitrary relational schemas, and *(ii)* a query optimization problem.

There exist a number of techniques and systems that address the problem of SPARQL to SQL translation, such as the ones described in [10, 8, 19]. However, each of these approaches has limitations that affect critical aspects of query answering over virtual RDF. These limitations include the generation of inefficient or even incorrect SQL queries, lack of formal background, and poor implementations.

In order to optimize the queries and generate efficient SQL, we exploit datasource metadata such as primary and foreign keys to eliminate redundant joins. This redundancy arises often because the RDF data model (over which SPARQL operates) is a ternary model (s p o) while the relational model is n-ary. Hence, the SPARQL equivalent of `SELECT * FROM t` on an n-ary table t requires exactly n triple patterns. When translating each of these triple patterns, a SPARQL-to-SQL technique will generate an SQL query with exactly $n-1$ self-join operations. It is well known that keeping these redundant joins is detrimental for

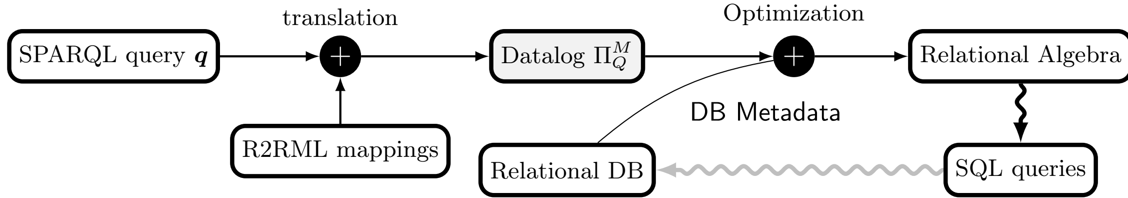


Figure 2.1: Proposed approach for translation of SPARQL to optimized SQL through Datalog with R2RML mappings

performance and a lot of research has been devoted to optimizing SQL queries in these cases. The most prominent area that investigates this subject is Semantic Query Optimization (SQO), from which we borrow techniques to optimize SPARQL translations.

The approach presented here, and depicted in Figure 2.1, deals with all the aforementioned issues. First, the SPARQL query and the R2RML mappings are translated into a Datalog program; the Datalog program is not meant to be executed, but instead we view this program as a formal representation of the query and the mappings that we can manipulate and transform into SQL. Second, we perform a number of structural and semantic optimizations on the Datalog program, including *optimization with respect to database metadata*. We do this by adapting well known techniques for optimization of logic programs and SQL query optimization. Once the program has been optimized, the final step is to translate it to relational algebra/SQL, and to execute it over the relational database. The technique is able to deal with all aspects of the translation, including URI and RDF Literal construction, RDF typing, and SQL optimization. It is implemented by the *Ontop* system, which provides the core query answering engine in the Optique architecture.

2.1.1 Experiments

We provide an evaluation of our SPARQL-to-SQL technique implemented in *Ontop* using DB2 and MySQL as backends. We compared *Ontop* with two systems that offer similar functionality to *Ontop* (i.e., SPARQL through SQL and mappings): Virtuoso RDF Views 6.1 (open source edition) and D2RQ 0.8.1 Server over MySQL. We also compared *Ontop* with three well known triple stores: OWLIM 5.3, Stardog 1.2, and Virtuoso RDF 6.1 (open source).

We considered the following benchmarks:

*BSBM***.** The Berlin SPARQL Benchmark (BSBM) [5] evaluates the performance of query engines utilizing use cases from the e-commerce domain.

*FishMark***.** The FishMark benchmark [3] is a benchmark for RDB-to-RDF systems that is based on a fragment of the FishBase DB, a publicly available database about fish species.

We observed that for BSBM, the query rewriting step takes around 10 ms in average. This is around 20%- 40% of the execution time. The queries had very high selectivity, therefore the execution time is small. For instance, *Ontop* requires 4ms to rewrite Query 1, and 17ms to perform the whole execution (including rewriting). The harder is the execution in the database, the smaller is the impact of the query rewriting step on the execution time.

We also compared the execution time of the SPARQL queries with the original BSBM SQL queries. We run these queries directly over the database engine, therefore the execution time includes neither the rewriting time, nor the time to post-process the SQL result set to generate an RDF result set. The performance obtained by MySQL is clearly much better than the one obtained by all the other Q&A systems, although the gap gets smaller as the dataset increases. It is worth noting that these queries are not SQL translation of SPARQL queries, thus they are intrinsically simpler, for instance, by not considering URIs.

Next, we can see is that for BSBM in almost every case, the performance obtained with *Ontop*'s queries executed by MySQL or DB2 outperforms all other Q&A systems by a large margin. The only cases in which

this doesn't hold are when the number of clients is less than 16 and the dataset is small (BSBM 25). This can be explained as follows: *Ontop*'s performance can be divided in three parts, *(i)* the cost of generating the SQL query , *(ii)* the cost of execution over the RDBMs and *(iii)* cost of fetching and transforming the SQL results into RDF terms. When the queries are cached, *(i)* is absent, and if the scenario includes little data (i.e., BSBM 25), the cost of *(ii)*, both for MySQL and DB2, is very low and hence *(iii)* dominates. We attribute the performance difference to a poor implementation of *(iii)* in *Ontop*, and the fact triple stores do not need to perform this step. However, when the evaluation considers 16 parallel clients, executing *Ontop*'s SQL queries with MySQL or DB2 outperforms other systems by a large margin. We attribute this to DB2's and MySQL's better handling of parallel execution (i.e., better transaction handling, table locking, I/O, caching, etc.). When the datasets are larger, e.g., BSBM 100/200, *Ontop (i)* stays the same. In these cases, *(ii)* dominates *(iii)*, since in both benchmarks queries return few results.

Regarding the FishMark benchmark, *Ontop* outperforms the rest almost in every case even from 1 single client. In FishMark, the original tables are structured in such a way that many of the SPARQL JOINs can be simplified dramatically when expressed as optimized SQL. For example, a FishMark query with 16 Join operations, when translated into SQL, *Ontop* is able to generate a query with only 5 joins.

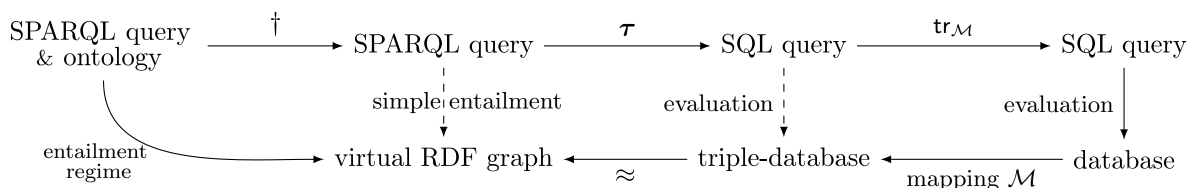
2.2 Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime

The SPARQL 1.1 query language, a W3C recommendation since 2013, has been designed to support various entailment regimes. As in the case of answering conjunctive queries over ontologies, these regimes are meant to provide more answers to SPARQL queries over RDF graphs by completing the knowledge by means of ontologies. The OWL 2 direct semantics entailment regime allows SPARQL queries over OWL 2 DL ontologies and RDF graphs, however query answering under this regime is intractable (coNP-hard for data complexity). Therefore, we investigate answering SPARQL queries under a less expressive entailment regime, which corresponds to OWL 2 QL, a profile of OWL 2 designed for efficient query answering. Moreover, we assume that data is stored in relational databases, and its relational schema is linked to the vocabulary of SPARQL queries by means of R2RML mappings.

We show how, given a SPARQL query, an OWL 2 QL ontology, an R2RML mapping and a database instance, to obtain an equivalent SQL query that can be evaluated over the database only. This is possible due to the following intermediate transformations:

1. First, answering SPARQL queries under the OWL 2 QL direct semantics entailment regime is reducible to answering queries under simple entailment. That is, for each SPARQL query q and OWL 2 QL ontology, we can construct a SPARQL query $q^\dagger$ that can be evaluated on any dataset directly. More precisely, $q^\dagger$ is evaluated over the virtual RDF graph obtained from a given relational database instance through the R2RML mappings.
2. Second, the SPARQL query $q^\dagger$ can be translated to an equivalent SQL $\boldsymbol{\tau}(q^\dagger)$ query over a relational representation of the virtual RDF graph as a 3-column table.
3. Finally, the resulting SQL query can be unfolded, using the R2RML mapping $\mathcal{M}$, to a SQL query $\mathsf{tr}_{\mathcal{M}}(\boldsymbol{\tau}(q^\dagger))$ over the original database.

These consecutive translations can be represented graphically as follows:



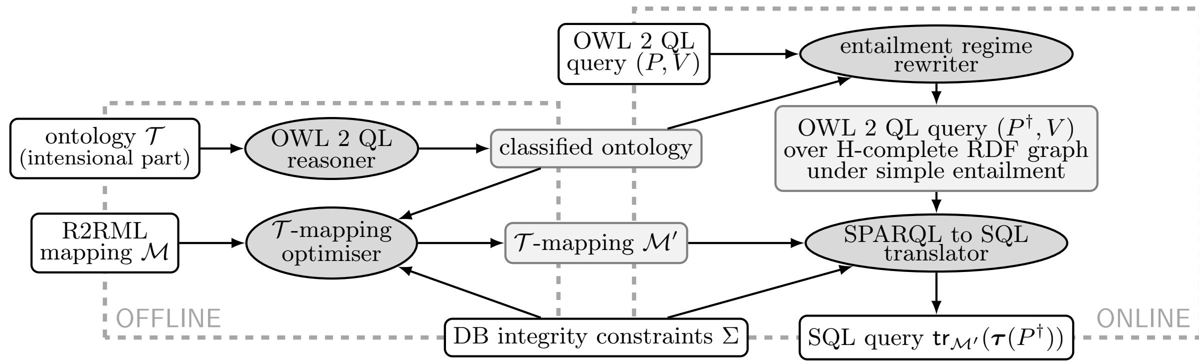
As in the more traditional OBDA setting, rewriting a SPARQL query into an SQL query that can be evaluated over the database by a relational engine has many advantages. However, for efficient query answering, the produced SQL query should be of a reasonable size and shape so that the DB engine is able to process it in an optimal way. Unfortunately, each of the three transformations may involve an exponential blowup. This problem is tackled in *Ontop* using the following optimization techniques.

(i) The mapping is compiled with the ontology into a $\mathcal{T}$-mapping and optimized by database dependencies (e.g., primary, candidate and foreign keys) and SQL disjunctions.

(ii) The SPARQL-to-SQL translation is optimised using null join elimination.

(iii) The unfolding is optimised by eliminating joins with mismatching R2RML IRI templates, de-IRIing the join conditions and using database dependencies.

These optimization techniques give rise to the following architecture to support answering SPARQL queries under the OWL 2 QL entailment regime with data instances stored in a database. As input we have an ontology $\mathcal{T}$, a database D over a schema Σ, and an R2RML mapping $\mathcal{M}$ connecting the languages of Σ and $\mathcal{T}$. The process of answering a given OWL 2 QL query (P, V) involves two stages, off-line and on-line.



The *off-line* stage takes $\mathcal{T}$, $\mathcal{M}$ and Σ and proceeds via the following steps:

(1) An OWL 2 QL reasoner is used to obtain a complete class / property hierarchy in $\mathcal{T}$.

(2) The composition $\mathcal{M}^{\mathcal{T}}$ of $\mathcal{M}$ with the class and property hierarchy in $\mathcal{T}$ is taken as an initial $\mathcal{T}$-mapping, and then optimised by (i) eliminating redundant triple maps detected by query containment with inclusion dependencies in Σ, (ii) eliminating redundant joins in logical tables using the functional dependencies in Σ, and (iii) merging sets of triple maps by means of interval expressions or disjunctions in logical tables. Let $\mathcal{M}'$ be the resulting $\mathcal{T}$-mapping over Σ.

The *on-line* stage takes an OWL 2 QL query (P, V) as an input and proceeds as follows:

(3) The graph pattern P and $\mathcal{T}$ are rewritten to the OWL 2 QL graph pattern $P^{\dagger}$ over the H-complete virtual RDF graph $G_{D,\mathcal{M}'}$ *under simple entailment* by applying the classified ontology of step (1) to instantiate class and property variables and then using a query rewriting algorithm.

(4) The graph pattern $P^{\dagger}$ is transformed to the SQL query $\boldsymbol{\tau}(P^{\dagger})$ over the 3-column representation *triple* of the RDF graph. Next, the query $\boldsymbol{\tau}(P^{\dagger})$ is unfolded into the SQL query $\mathsf{tr}_{\mathcal{M}'}(\boldsymbol{\tau}(P^{\dagger}))$ over the original database D. The unfolded query is optimised using the techniques similar to the ones employed in step (2).

(5) The optimised query is executed by the database.

2.2.1 Evaluation

The architecture described above has been implemented in *Ontop*. We evaluated its performance using the LUBM Benchmark extended with queries containing class and property variables (a total of 21 queries), and compared it with two other systems, OWL-BGP r123 and Pellet 2.3.1. We can summarize the performance of *Ontop* as follows:

- Generally, *Ontop* requires less time to start up, as it does not perform costly pre-computations as OWL-BGP and Pellet do.
- For first-order queries, due to the optimizations, the SQL queries produced by *Ontop* are simple, so the database engine is able to process them efficiently.
- As for queries with second-order variables, *Ontop* performs not as good as with the other queries, however still considerably well.
- While Pellet outperforms *Ontop* on small datasets, only *Ontop* is able to provide answers for very large datasets.

Chapter 3

Implementation Development

In this chapter, we present the major changes that have been implemented in *Ontop*, that were directly driven by the Statoil use case to tackle a variety of issues that came up during the experimentation both with the NPD Factpages and with the EDPS database.

In Section 3.1, we describe the adaptations done to *Ontop* to better integrate it in the Optique platform, and to improve the support for standard languages and libraries. In Section 3.2, we discuss the new more efficient TBox reasoning implementation, and the novel ontology reasoning services that exploit it, namely consistency checking, and checking for empty classes and properties. We describe also the development of an extension of *Ontop* with spatial features. Finally, in Section 3.4, we describe the publication of *Ontop* on the central Maven repository, and report on the last stable releases of the system.

3.1 Integration with the Optique Platform

The Optique consortium agreed to adopt the following languages and libraries for the Optique platform:

- The mapping language should be W3C R2RML [9] and the mappings should be handled using the corresponding R2RML API.
- SQL dialects of major database systems should be supported as R2RML source queries.
- End-user queries should be expressed in SPARQL.
- The software interface for RDF, SPARQL, and repository management should be the Sesame API[1].

During the second year of the Optique project, a significant part of the development has been dedicated to support the aforementioned standards and libraries. More specifically:

- We have provided support for R2RML by integrating the new Optique R2RML API (Section 3.1.1).
- We have extended the SQL support by integrating the JSQLParser library (Section 3.1.2) and dealing with multi-schema queries (Section 3.1.3).
- We have improved the support of SPARQL by accepting BIND expressions and UNIONs inside OPTIONAL blocks (Section 3.1.4).
- We have upgraded the Sesame API in *Ontop* to the latest 2.7.x version (Section 3.1.5).

[1] `http://www.openrdf.org/`

3.1.1 Integration of the Optique R2RML API

The mapping language R2RML [9] developed by the W3C RDB2RDF Working Group[2] has been chosen as the mapping language in the Optique platform since the beginning of the Optique project. At that time, no Java API for parsing and representing R2RML was available, therefore, at FUB, we had to develop from scratch the first version of the R2RML implementation, using the Sesame library to parse the R2RML input as an RDF graph.

Based on this first implementation, the new Optique R2RML API was developed as a standalone library by UiO [18] and released under the Apache License[3]. The core of the new API is designed to be "independent of the dependencies" [18] and several bridge extensions have been provided for the OWL, Jena, and Sesame APIs. The new API also fixed several bugs of the former version.

To integrate the Optique R2RML API, we first deployed the new R2RML API to the Bolzano Maven repository, since it is not available on the official Maven repository yet. The core of the API is:

```
<dependency>
    <groupId>org.optique-project</groupId>
    <artifactId>r2rml-api</artifactId>
    <version>0.1.3</version>
</dependency>
```

We also need the Sesame bridge artifact to use the new API with the Sesame library:

```
<dependency>
    <groupId>org.optique-project</groupId>
    <artifactId>r2rml-api-sesame-bridge</artifactId>
    <version>0.1.3</version>
</dependency>
```

The next step was to carefully replace the old ad-hoc code by the new API, which makes the bi-directional translation from R2RML to the native mapping syntax of *Ontop* easier. The *Ontop* system is now plainly accessible to users who are familiar with the R2RML recommendation.

After integrating the new R2RML API, *Ontop* is able to cover almost all R2RML mappings present in the Optique use cases. Moreover, it passes 90% (80/87) of the W3C R2RML Compliance Tests[4]; see the dedicated Wiki page[5] for a detailed report.

3.1.2 SQL Support in the Mapping Language

Mapping assertions contain the correspondence between the predicates of the ontology and the appropriate SQL queries over the relational data source. The *Ontop* system needs to process the SQL source queries from the mapping assertions to transform them into an internal Datalog representation. A critical component of the SQL query analysis is the SQL parser. The *Ontop* system was using an ad-hoc SQL parser that was only supporting a small fragment of the SQL standard.

We decided to integrate a new SQL parser, namely the *JSQLParser*[6], which is an open source library that has proven its ability to support special features and different syntax of the most common databases. In particular this API was chosen because it supports the main databases required by *Ontop* (Oracle, MySQL, PostgreSQL, MS SQLServer, H2 and DB2) and it is currently still under active development to integrate new features for the less common SQL features. JSQLParser allows one to parse more general forms of SQL queries (including subqueries in SQL) present in the mappings. It is structured to parse an SQL statement

[2] http://www.w3.org/2001/sw/rdb2rdf/
[3] https://github.com/R2RML-api/R2RML-api
[4] http://www.w3.org/2001/sw/rdb2rdf/test-cases/
[5] https://github.com/ontop/ontop/wiki/W3C-R2RML-Compliance
[6] https://github.com/JSQLParser/JSqlParser

and translate it into a hierarchy of Java classes; then the generated Java class hierarchy can be navigated according to the visitor pattern.

JSQLParser has been easily integrated with *Ontop* as a Maven dependency. We implemented new Visitor classes to visit the statement (the parsed query) and retrieve information about tables, projections, selections, join conditions, and alias mappings. The visitor pattern is convenient for incrementally adding new features. When our parser does not know how to parse the query correctly, we replace the SQL query by a view. However, such views penalize the performance at evaluation time, so it is recommended to use supported terms in the mappings as much as possible.

We describe briefly the new supported features. For instance, using a `SubSelectionVisitor` class, we have now the possiblity to support simple subqueries of the following form:

```
select * from
 (select * from tb_books) as CHILD, (select * from tb_authors) as PARENT
WHERE CHILD.bk_code = PARENT.bk_code
```

These subqueries are considered simple subselects because they do not have joins or WHERE conditions. It is important for *Ontop* to recognize them and parse them accordingly because they appear frequently in mappings and they are required to support R2RML mappings. Additionally more SQL functions can be handled:

- LIKE was introduced by adding a new LIKE expression for Datalog conversion.
- IN is handled using equality and OR expressions.
- BETWEEN is handled using OR and AND expressions.
- Regular expressions are not part of standard SQL and each database supports them differently. DB2 and MS SQL Server do not provide an operator for regular expressions, while the other databases use different syntaxes. In *Ontop*, we support REGEX in Oracle, MySQL, PostgresSQL and H2.

  The supported operators for regular expressions are:

  - in MySQL REGEXP [BINARY]
  - in H2 and Postgres $\sim$, $\sim *$, $! \sim$, $! \sim *$
  - in Oracle `REGEXP_LIKE`

The new SQL parser is also used for supporting multi-schema queries, as presented in the coming subsection.

3.1.3 SQL Multi-schema Support

Some databases have *schemas* that can be used to separate a database into different namespaces. The schema is used as a prefix to the object name. For example, in PostgresSQL, the table *animals* in schema *zoo* can be accessed by the identifier *zoo.animals*. The prefix is unnecessary when accessing the current/default schema.

In *Ontop*, we added the multi-schema support in order to allow users to query tables from different schemas. To achieve this, we improved the *Ontop* Mapping-to-Datalog converter. It is now able to recognize the database system from the JDBC connection and to adjust itself to its specificities. Indeed, each database support schemas differently:

- In PostgreSQL, the default schema is *public*.
- In Oracle, the default schema is the username.
- In MySQL, schema and database are equivalent.
- In Microsoft SQL Server, the default schema is *dbo*.

- In H2, the default schema is *PUBLIC*.
- In DB2, the object to be created is assigned to the default schema with the value of the session authorisation ID.

In the multi-schema settings, the local name of a table is no longer unique. Therefore, the table is identified by its full prefixed name in *Ontop* and in the database metadata.

Metadata

The schema information is taken from the database metadata. In *Ontop* virtual mode, we can fetch metadata in three different ways:

Constructor. The metadata is provided by the user. In the database metadata, we are able to insert a list of data definitions. The table definitions should respect the *schema.table* naming convention.

Full metadata. The data definitions are added to the metadata, and table and column names are retrieved from the JDBC connection to the database.

Parsed mappings. Table names are extracted from the mappings while column names are retrieved from the JDBC connection.

In *Ontop*, multi-schema queries are supported in the constructor and parsed mappinga modes only.

Particular attention must be given to the choice of quotes and cases in the table and column names. The case handling of identifiers is database-specific. As a general rule, if quotes are consistently not used, *Ontop* will always support the identifiers. Unquoted table and view names are translated internally by the *Ontop* system to the default case of the database engine. With respect to this, please note that:

- Oracle and H2 change unquoted identifiers to uppercase automatically.
- DB2 names are not case sensitive. Unquoted object names are converted to uppercase. If a name is enclosed in quotation marks, the name becomes case sensitive. However, the schema name is case sensitive, and must be specified in uppercase characters.
- PostgreSQL changes unquoted identifiers (both column and alias names) to lowercase.
- MySQL changes the unquoted columns to lowercase. Tables are stored as files in the running server, so the case sensitivity of database and table names depends on the host operating system.
- Microsoft SQL Server is not case sensitive by default.

3.1.4 SPARQL Support Extension

The SPARQL support has been improved: BIND expressions are now accepted and OPTIONAL blocks are better handled.

BIND

By supporting the BIND construct, values can be assigned to variables. For instance, in the following query, the discounted price is computed by the expression `BIND (?p*(1-?discount) AS ?price)`:

```
PREFIX : <http://it.unibz.krdb/obda/test/simple#>
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title ?price WHERE
{  ?x ns:hasPrice ?p .
```

```
    ?x ns:hasDiscount ?discount .
    BIND (?p*(1-?discount) AS ?price) .
    FILTER(?price < 20) .
    ?x dc:title ?title .
}
```

OPTIONAL

We developed techniques for supporting the SPARQL OPTIONAL construct combined with UNION in the second argument. Observe that the OPTIONAL construct (unlike JOIN) is not distributive with respect to UNIONs, and since the process of unfolding the query with respect to the mappings introduces UNIONs, correctly dealing with OPTIONAL and UNION together required quite significant changes in the unfolding process and the SQL generation step. Essentially, queries with OPTIONAL cannot always be unfolded to Union of Conjunctive Queries UCQs, therefore we implemented a new Datalog Unfolder using a bottom-up approach. In addition, the old implementation of SQL Generator can only handle UCQs. We extended the SQL Generator so as to handle acyclic Datalog programs. The new SQL generator is also essential for the new future features in *Ontop*, like SWRL rules.

3.1.5 Sesame API Upgrade

The OpenRDF Sesame API[7] is the Java API used in the Optique platform for integrating *Ontop*. This integration is made possible by the fact that in, the Optique platform, *Ontop* acts as a Sesame Repository by implementing the Sesame Repository API.

During the reporting period, we have upgraded *Ontop* to support Sesame 2.7.10, which at the time of the migration was the newest available version. The migration itself required substantial effort, because Sesame packages themselves underwent a lot of structural changes with respect to the previous version that was adopted in *Ontop* (see the release note of Sesame 2.7.10[8]). Given the substantial changes that had to be implemented to carry out the migration, this was also the right occasion to refactor the *Ontop* code, fully committing package names, artifact ids, and other component names to the name "`ontop`".

3.2 Novel Reasoning Functionalities

The TBox reasoner, which is responsible for classifying the concepts and properties of the OWL 2 QL ontology, is a crucial component of *Ontop*. Classification provides the basis for all other inference tasks, notably query answering, to which also consistency checking is then reduced (see Section 3.2.3). Note that, in the design phase of the OBDA system, where users operating with the Optique Platform make additions and changes both to the ontology and to the mappings, classification and consistency checking are operations that need to be carried out repeatedly while interacting with users. Therefore efficiency of classification (and hence of consistency checking) crucially affects overall usability of the system.

3.2.1 Efficient TBox Reasoning via DAG Manipulation

Ontop supports the OWL 2 QL profile of OWL 2, and for such ontology language, TBox reasoning is NLogSpace-complete [1]. This means that it relies essentially on reachability in directed acyclic graphs (DAGs). The previous implementation of *Ontop* was relying on an ad-hoc non-optimized implementation of operations for the manipulation of DAGs, and thus was suffering from performance limitations. DAGs were built directly without paying attention to the presence or introduction of redundant edges, i.e., those edges that can be derived by transitivity from other ones in the graph.

[7] `http://www.openrdf.org/`

[8] `https://openrdf.atlassian.net/secure/ReleaseNote.jspa?projectId=10000&version=10060`

We have devised a new implementation, in which DAGS are not built directly anymore. Instead, we start creating graphs that already contain all the information present in the ontology but still admit cycles and redundant edges. Optimized DAGs are built later, giving more flexibility to the system, by using the JGraphT library[9] as a tool for graph manipulation. We put special attention to choose optimal algorithms for reachability checking. In particular the implemented algorithm for strongly connected components follows the Cheriyan-Mehlhorn/Gabow approach [11] and we provide new algorithms to iterate and traverse a DAG for the TBox reasoner. Optimization based on *DL-Lite* TBox reasoning using DAGs [13] makes reasoning efficient for query answering. Optimized DAGs built in our implementation are used to index information and to create a mapping with the database. The TBox reasoner is used for traversing the ontology.

More concretely, starting from the TBox provided by the user, two graphs are generated for properties and classes, respectively. The graph representation of property inclusions in the ontology contains also implicit inclusions. For example, it contains an inclusion between the inverses of properties R and S if R is declared as sub-property of S in the ontology. Graphs are later transformed into optimized DAGs in such a way that entailment checking in the TBox is reduced to reachability checking in the corresponding graph. Each DAG vertex represents a set of equivalent classes or properties, while edges form a minimal set, whose transitive and reflexive closure coincides with the transitive and reflexive closure of the ontology graph. In this way, the TBox reasoning services, including those that are exploited by the query rewriting algorithm, can be performed efficiently.

3.2.2 Consistency Checking

Consistency checking is one of the basic DL reasoning tasks. In $DL\text{-}Lite_{\mathcal{R}}$ (i.e., OWL 2 QL), there are three types of TBox axioms that can cause inconsistency with respect to the data retrieved via the mappings from the underlying data source: disjoint class axioms, disjoint properties, and functional properties. Following the approach described in [7], we have implemented consistency checking by reducing it to answering suitable SPARQL ASK queries over the given ontology. Such queries correspond to Boolean queries, i.e., queries that return either true or false. Specifically, the queries are as follows:

- For each disjoint class axiom `owl:DisjointClasses`(A, B) , we create a query:
  `ASK { ?x a A ; a B }`
- For each disjoint object property axiom `owl:DisjointObjectProperties`(P, Q) or disjoint data property axiom `owl:DisjointDataProperties`(P, Q), we create a query:
  `ASK { ?x P ?y; Q ?y }`
- For each functional property axiom `owl:FunctionalObjectProperty`(P) or
  `owl:FunctionalDataProperty`(P), we create a query:
  `ASK { ?x P ?y; P ?z. FILTER (?z != ?y) }`

If one of these ASK queries is evaluated to true, the ontology is inconsistent; otherwise no violations could be detected and the ontology is consistent.

The consistency checking feature is available in the Java API of *Ontop* and also in its Protégé Plugin, as shown in Figure 3.1.

3.2.3 Checking for Empty Classes and Properties

When a class or a property is not mapped to the data sources, i.e., it does not appear in any of the mapping assertions, *Ontop* will not be able to retrieve any instances of that class/property from the underlying data. The same might also happen when the mapping for a class/property is specified incorrectly. In such cases, the class/property appears to be *empty*. Hence, checking for empty classes and properties is useful to diagnose some common mapping problems, and correct them by (properly) defining mappings. We added a new

[9] https://github.com/jgrapht/jgrapht

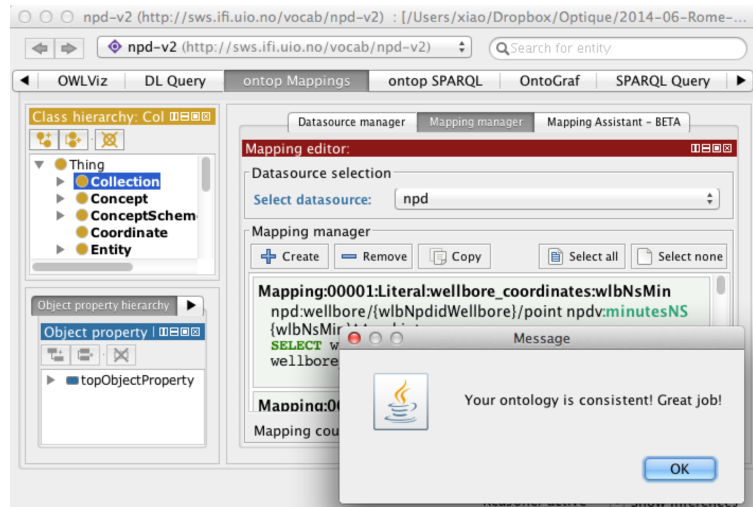


Figure 3.1: *Ontop* Inconsistency checking feature for Protégé plugin

feature to the *Ontop* Protégé plugin that allows the user to find empty classes and properties, and to check if some of them correspond to missing data.

In Protégé, the user can select in the *ontop* menu the action *Check for empties...*. A frame is then displayed that counts and lists the empty classes and properties (see Figure 3.2). This feature is also available in the Java API.

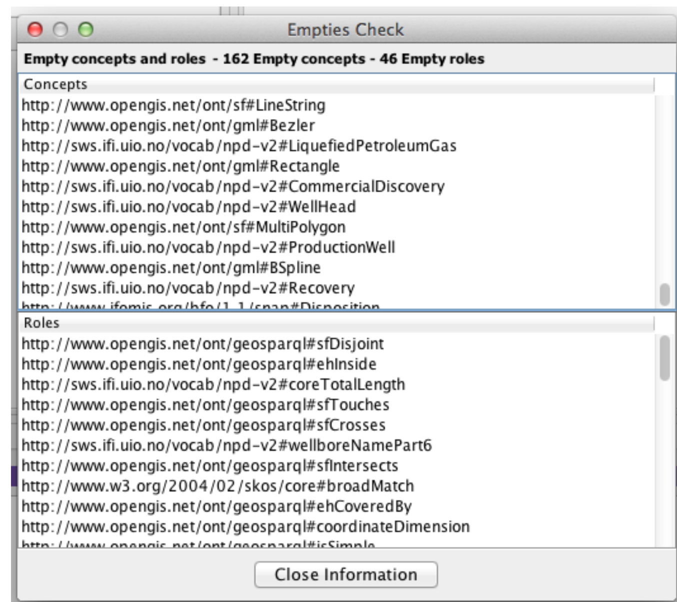


Figure 3.2: *Ontop* Emptiness checking feature for Protégé plugin

In order to achieve this, the respective SPARQL queries generated for each property and concept are:

```
SELECT * WHERE {?x <property> ?y. }
SELECT * WHERE {?x a <class>. }
```

Each SPARQL query is translated by *Ontop* into a SQL query before being evaluated by the database. If the query provide no result, then the corresponding property or concept is declared to be empty.

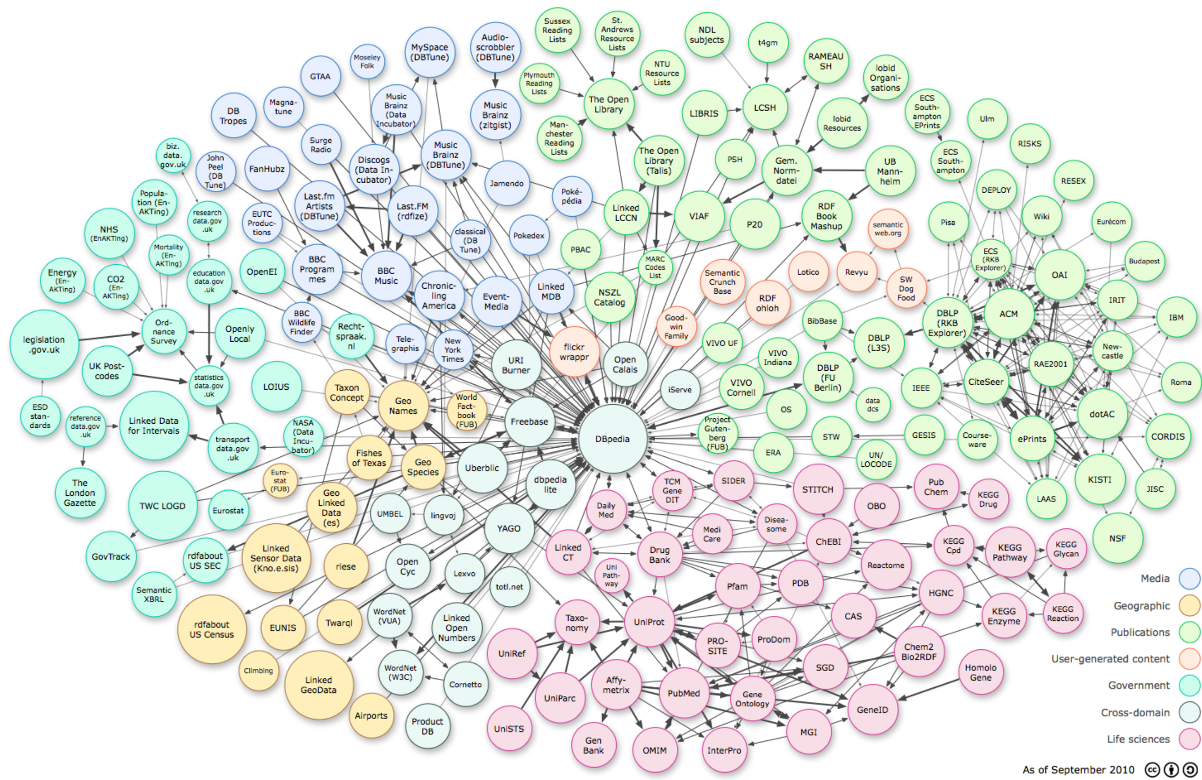


Figure 3.3: The Linked Open Data cloud

3.3 Extending *Ontop* with Spatial Features

We describe now the development, carried out at UoA, of an extension of *Ontop* with spatial features, named *Ontop*-spatial. The motivation behind the development of *Ontop*-spatial is to increase the expressive power of the queries that can be posed against the Optique platform, by adopting features of GeoSPARQL [15], especially for the Statoil use case, where geospatial data are handled. With this spatial extension in place, the geospatial dimension of this data can be fully exploited, by enabling users to pose geospatially rich queries such as "*Find wellbores that are spatially contained in a specific discovery area*".

We were also motivated by the emerging interest of scientific communities from various domains (e.g., earth scientists) that produce and process geospatial data to publish them as linked geospatial data in order to be used combined with other sources. Figure 3.3 illustrates the Linked Open Data cloud (LOD), in which the datasets that have been published as linked geospatial data appear in yellow colour.

Following to this trend, the Semantic Web community has been very active in the previous years in the geospatial domain, proposing data models, query languages, systems, and applications for the representation, querying, and management of geospatial data in the Semantic Web. The development of solutions for the production, publishing, and use of geospatial data have played central role in EU FP7 projects as well, such as TELEIOS[10], LEO[11], MELODIES[12], and GeoKnow[13]. On the other hand, research on relational geospatial databases counts decades, leading to several efficient geospatial DBMS.

Despite the extensive research in the fields of the relational databases and the Semantic Web on the development of solutions for handling geospatial data efficiently, there is no -to the best of our knowlege-OBDA system that enables the creation of virtual, *geospatial* RDF graphs on top of geospatial databases. This would be of major importance for scientists that produce and process geospatial data, as they mainly

[10] http://www.earthobservatory.eu/
[11] http://www.linkedeodata.eu/
[12] http://www.melodiesproject.eu/
[13] http://geoknow.eu/

store them in traditional geospatial databases (e.g., PostGIS, SpatiaLite, etc.). With the existing solutions in place, these scientists are forced to transform their data to RDF in order to publish them and/or use them in combination with other sources. This is the case for most of the use cases in the EU projects mentioned above. For example, in the project LEO, a tool named GeoTriples[14] has been developed. GeoTriples extends D2RQ in order to transform data that reside in geospatial databases (e.g., PostGIS), or other geospatial formats (e.g., Shapefiles) in RDF. In the work described in this section, we follow a different direction: Users do not have to transform their data and then store them in a geospatial RDF store in order to process them and combine them with other RDF data. They will be able to access them transparently, based on R2RML mappings.

The need for representing and querying geospatial data in the Semantic Web led to the development of geospatial extensions in RDF and SPARQL, such as the ones presented in [14, 16, 15]. The data model stRDF and the query language stSPARQL is an extension of RDF and SPARQL 1.1, respectively, developed for the representation and querying of spatial [14] and temporal data (i.e., the valid time of triples [4]). Our partner UoA developed this framework at the same period when GeoSPARQL was being developed. Another framework that has been developed for the representation and querying of geospatial data on the Semantic Web is GeoSPARQL, which is an OGC standard. GeoSPARQL and stSPARQL were developed independently, but they have a lot of features in common, the most important of which are that they both adopt the OGC standards serializations WKT and GML for representing geometries, and that they both support spatial analysis functions as extension functions. Their main differences derive from the fact that stSPARQL extends SPARQL 1.1., so it inherits and extends important features of SPARQL 1.1., providing support for spatial updates and spatial aggregates.

3.3.1 Supported Spatial Features

In the direction of extending *Ontop* with spatial features, the first issue that had to be addressed was selecting which spatially-enhanced query language should be supported, of the ones described above. We selected GeoSPARQL for the following reasons:

- It is a widely adopted OGC standard
- The topological extension of GeoSPARQL allows binary topological relations to be used as RDF properties. This could be useful for the Statoil use case
- The additional features offered by stSPARQL (spatial updates, spatial aggregates) do not seem necessary for the use cases of the project.

However, given the fact that GeoSPARQL and stSPARQL are very similar, both languages could be supported in the future.

By the reporting period, the following features are supported by *Ontop*-spatial:

- *The WKT Datatype.* There are two standard ways of expressing geometry serializations in RDF defined by OGC: using literals of the datatypes *WKT* and *GML*. Using standard WKT and GML literals is also compliant with R2RML, which already supports the use of custom datatypes by using the `rr:datatype` primitive. *Ontop*-spatial extends *Ontop* with the capability to support standard WKT literals and GML literals will also be supported in the future.
- *Spatial filters.* Spatial relations can appear as SPARQL extension functions in the filter clause of the query, as defined in GeoSPARQL. This is a collection of topological query funcitons that operate on geometry literals defined in the Geometry Topology Extension component of GeoSPARQL. More specifically, the current version of *Ontop*-spatial supports the Simple Features relation family (Requirement 22 of the GeoSPARQL specification), the Egenhofer relation family (Requirement 23 of the GeoSPARQL specification), and the RCC8 Relation family (Requirement 24 of the GeoSPARQL

[14] `http://sourceforge.net/projects/geotriples/`

specification). These functions are implemented as SPARQL extension functions, as specified in the standard, and can be placed in the `FILTER` clause of a query, in order to express either *spatial selections* or *spatial joins*. According to the specification, these functions must appear in the `SELECT` clause of the query as well, and this feature is not yet supported in this initial version of *Ontop*-spatial, but it will be supported in the future.

Examples of geospatial queries that are currently supported in *Ontop*-spatial are given below.
Example 1. Select the location of wellbores that are contained in field areas.

```
SELECT distinct ?w2
WHERE { ?x1 :hasGeometry ?g1 .
        ?g1 geo:asWKT ?w1 .
        ?x1 rdf:type :fieldArea .
        ?x2 :hasGeometry ?g2 .
        ?x2 rdf:type :wellborePoint .
        ?g2 geo:asWKT ?w2 .
        FILTER(geo:contains(?g2,?g1))
      }
```

The query described above uses the boolean extension function `geo:contains` to express a spatial filter. This filter is internally translated into the respective spatial operator of the underlying geospatial database, performing a *spatial join*.
Example 2. Select the field areas that contain the wellbore located in (2.497514,56.847728).

```
SELECT distinct ?w
WHERE { ?x rdf:type :fieldArea .
        ?x :hasGeometry ?g .
        ?g geo:asWKT ?w .
        FILTER(geo:contains(w,"POINT(2.497514 56.847728")))
      }
```

Similarly, this filter is internally translated into the underlying SQL operator, performing a *spatial selection.*

Implementation. The current implementation of *Ontop*-spatial extends *Ontop* release 1.12. A lot of extensions had to be performed in the *Ontop* components (some of them in the core), the most significant of which are as follows.

- *The SPARQL parser. Ontop* uses the Sesame[15] library to parse SPARQL queries. Sesame in turn generates its SPARQL parser using javacc[16]. To support the GeoSPARQL features described above, such as the spatial extension functions in the `FILTER` clause of the query, a new javacc parser was generated for Sesame, and that was used in turn for *Ontop*-spatial.
- *SPARQL-to-Datalog Translator.* The boolean extension functions described above are translated into a Datalog predicate.
- *Datalog-to-SQL Translator.* Datalog spatial predicates are translated into the respective SQL spatial function, which is supported by the underlying *geospatial* DBMS. Note here that, as there is no standardized syntax for the spatial functions in relational databases, the SQL adapters should be adjusted accordingly. Currently, PostGIS is supported as an underlying geospatial relational database.

[15] `http://www.openrdf.org/`
[16] `https://javacc.java.net/`

- *SQL generator*. The SQL generator of *Ontop* was extended so that geometry columns are identified. In spatial relational databases, geometries are mostly stored in the Well Known Binary (WKB) format, which is not a standard SQL datatype. Information in WKB is exported as text (WKT) in *Ontop*.

Theoretical formulation of the translations described above is on-going work, which will be presented in the future.

3.3.2 Future work

We have presented our work on extending the system *Ontop* with the ability to represent and query geospatial information stored in a geospatial back-end, by supporting a big subset of the OGC standard GeoSPARQL. In the future, we plan to continue our work in the following directions:

- Formalize the translation from GeoSPARQL into "spatial" SQL
- Carry out an experimental evaluation with systems that offer similar functionality (e.g., geospatial RDF stores). The benchmark presented in [12] could be used in this case.
- Extend our implementation into the direction of supporting more GeoSPARQL features (e.g., spatial features in the select clause of queries, spatial aggregates, etc.)

The issues described above is on-going work and a publication based on them is under preparation. Also, we plan to integrate *Ontop*-spatial into the Optique platform.

3.4 Releases

In agreement with the Optique consortium, in year 2 of Optique we have started releasing *Ontop* under the Open Source Apache 2 license. Like many Java libraries, *Ontop* has been published on the central Maven repository, so it can be easily integrated as a dependency in any other Java-based system. We keep releasing new stable versions with bug fixes and new features every three to four months. An internal log reports around 420 new registrations in this period.

3.4.1 Maven

Apache Maven is a popular software project management and comprehension tool adopted by most of the open source Java projects. In contrast with the former approach of downloading jar files manually, Maven allows users to simply declare the dependencies in a dedicated XML file.

All the artifacts of *Ontop* share the groupId `it.unibz.inf.ontop`. The artifactIds are `ontop-obdalib-core`, `ontop-obdalib-owlapi3`, `ontop-obdalib-protege4`, `ontop-obdalib-sesame`, `ontop-obdalib-r2rml`, `ontop-quest-db`, `ontop-quest-owlapi3`, `ontop-quest-sesame`, `ontop-reformulation-core`.

Since version 1.10, *Ontop* has been deployed to the central Maven repository. For instance, if a user wants to use the OWL API interface of *Ontop*, she can simply declare a suitable dependency by adding the following code to the file `pom.xml`:

```
<dependency>
    <groupId>it.unibz.inf.ontop</groupId>
    <artifactId>ontop-quest-owlapi3</artifactId>
    <version>1.12.0</version>
</dependency>
```

For the Optique platform, the most important *Ontop* dependency is its Sesame API interface, which is available under the `ontop-quest-sesame` artifact:

```
<dependency>
    <groupId>it.unibz.inf.ontop</groupId>
    <artifactId>ontop-quest-sesame</artifactId>
    <version>1.12.0</version>
</dependency>
```

3.4.2 Released Versions

In the second year of Optique, we have released four stable versions of *Ontop*. We provide here a brief summary of each release. A complete change log is available on a dedicated wiki[17].

Version 1.10, released on 06/12/2013. This is the first version released under the Apache 2 license and published on the central Maven repository (cf. Section 3.4.1).

Version 1.11, released on 19/02/2014. This release integrates the JSQLParser (cf. Section 3.1.2). It introduces support for consistency checking (cf. Section 3.2.2), multi-schema queries (cf. Section 3.1.3), the SQL terms IN, BETWEEN, and LIKE, and the SPARQL term BIND (cf. Section 3.1.4). The Sesame API has been upgraded to the version 2.7.10 (cf. Section 3.1.5).

Version 1.12, released on 26/06/2014. This version integrates the new Optique R2RML API (cf. Section 3.1.1). It features a faster TBox reasoner implementation (cf. Section 3.2.1), and emptiness checking for classes and properties (cf. Section 3.2.3).

Version 1.13, released on 29/09/2014. This version features support for regular expressions in mappings (cf. Section 3.1.2), proper handling of datatypes in ontologies and mappings, stream output in the Protégé plugin, and support for the HSQL database.

[17] `https://github.com/ontop/ontop/wiki/ObdalibPluginChangeLog`

Bibliography

[1] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. The *DL-Lite* family and relations. *J. of Artificial Intelligence Research*, 36:1–69, 2009.

[2] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2nd edition, 2007.

[3] Samantha Bail, Sandra Alkiviadous, Bijan Parsia, David Workman, Mark van Harmelen, Rafael S. Goncalves, and Cristina Garilao. FishMark: A linked data application benchmark. In *Proc. of the Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+HPCSW 2012)*, volume 943, pages 1–15. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/`, 2012.

[4] Konstantina Bereta, Panayiotis Smeros, and Manolis Koubarakis. Representation and querying of valid time of triples in linked geospatial data. In *Proc. of the 10th Extended Semantic Web Conf. (ESWC)*, volume 7882 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2013.

[5] Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *Int. J. on Semantic Web and Information Systems*, 5(2):1–24, 2009.

[6] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodríguez-Muro, and Riccardo Rosati. Ontologies and databases: The *DL-Lite* approach. In Sergio Tessaris and Enrico Franconi, editors, *Reasoning Web. Semantic Technologies for Informations Systems – 5th Int. Summer School Tutorial Lectures (RW)*, volume 5689 of *Lecture Notes in Computer Science*, pages 255–356. Springer, 2009.

[7] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. of Automated Reasoning*, 39(3):385–429, 2007.

[8] Artem Chebotko, Shiyong Lu, , and Farshad Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data and Knowledge Engineering*, 68(10):973–1000, 2009.

[9] Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF mapping language. W3C Recommendation, World Wide Web Consortium, September 2012. Available at `http://www.w3.org/TR/r2rml/`.

[10] Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, and Z. Meral Ozsoyoglu. A complete translation from SPARQL into efficient SQL. In *Proc. of the 2009 Int. Database Engineering & Applications Symposium (IDEAS)*, pages 31–42. ACM Press, 2009.

[11] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Lett.*, 74, 2000.

[12] George Garbis, Kostis Kyzirakos, and Manolis Koubarakis. Geographica: A benchmark for geospatial RDF stores (long version). In *Proc. of the 12th Int. Semantic Web Conf. (ISWC)*, volume 8219 of *Lecture Notes in Computer Science*, pages 343–359. Springer, 2013.

[13] Sarah Seyenam Adjoa Komla-Ebri. DL-Lite reasoning using directed acyclic graphs. Master's thesis, Free University of Bozen-Bolzano, Italy, 2013.

[14] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. Strabon: A semantic geospatial DBMS. In *Proc. of the 11th Int. Semantic Web Conf. (ISWC)*, volume 7649 of *Lecture Notes in Computer Science*, pages 295–311. Springer, 2012.

[15] Open Geospatial Consortium (OGC). GeoSPARQL – a geographic query language for RDF data. OGC Candidate Implementation Standard, 02 2012.

[16] Matthew S. Perry. *A Framework to Support Spatial, Temporal and Thematic Analytics over Semantic Web Data*. PhD thesis, Wright State University, Dayton, OH, USA, 2008. AAI3324256.

[17] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *J. on Data Semantics*, X:133–173, 2008.

[18] Marius Strandhaug. An R2RML mapping management API in java – making an API independent of its dependencies. Master's thesis, Department of Informatics, University of Oslo, 2014.

[19] Fred Zemke. Converting SPARQL to SQL. Technical report, Oracle Corporation, 2006. Available at `http://lists.w3.org/Archives/Public/public-rdf-dawg/2006OctDec/att-0058/sparql-to-sql.pdf`.

Appendix A

Efficient SPARQL-to-SQL with R2RML Mappings

This appendix reports the paper:

> Martin Rezk, Mariano Rodriguez-Muro:
> Efficient SPARQL-to-SQL with R2RML mappings. To appear in *Journal of Web Semantics* (JWS), 2014.

Efficient SPARQL-to-SQL with R2RML mappings

Mariano Rodríguez-Muro[1], Martin Rezk[1]

[1] *KRDB Research Centre, Free University of Bozen-Bolzano*

Abstract

Existing SPARQL-to-SQL translation techniques have limitations that reduce their robustness, efficiency and dependability. These limitations include the generation of inefficient or even incorrect SQL queries, lack of formal background, and poor implementations. Moreover, some of these techniques cannot be used over arbitrary DB schemas due to the lack of support for RDB to RDF mapping languages, such as R2RML. In this paper we present a technique (implemented in the -ontop- system) that tackles all these issues. We propose a formal approach for SPARQL-to-SQL translation that *(i)* generates efficient SQL by combining optimization techniques from the logic programming and SQL optimization fields; *(ii)* provides a well-defined specification of the SPARQL semantics used in the translation; and *(iii)* supports R2RML mappings over general relational schemas. We provide extensive benchmarks using the -ontop- system for Ontology Based Data Access (OBDA) and show that by using these techniques -ontop- is able to outperform well known SPARQL-to-SQL systems, as well as commercial triple stores, by several orders of magnitude.

Keywords: OBDA, -ontop-, SPARQL, Datalog, SQL, R2RML, RDF, RDB-to-RDF, RDBMS

1. Introduction

In an Ontology-Based Data Access (OBDA) framework, queries are posed over a conceptual layer and then translated into queries over the data layer. The conceptual layer is given in the form of an ontology that defines a shared vocabulary, and the data layer is in the form of one or more existing data sources. In this context, the most widespread data model for the conceptual layer and its matching query language are RDF (the Resource Description Framework) and SPARQL. Today, most enterprise data (data layer) is stored in relational databases, thus it is crucial that OBDA frameworks support RDB-to-RDF mappings. The new W3C standard for RDB-to-RDF mappings, R2RML [10], was created towards this goal.

R2RML mappings are used to expose relational databases as virtual RDF graphs. These virtual graphs can be *materialized*, generating RDF triples that can be used with RDF triple stores, or they can also be kept virtual and queried only during query execution. The virtual approach avoids the cost of materialization and (may) allow to profit from the more than 30 years maturity of relational systems (e.g., efficient query answering, security, robust transaction support, etc.). One of the most promising approaches for on-the-fly query answering over virtual RDF is query answering by query rewriting, that is, translating the original SPARQL query into an equivalent SQL query. This SQL query is then delegated to the DBMS for execution. In order to use these advantages provided by the DBMS, the query rewriting technique must produce "reasonable" SQL queries, that is, not excessively large or too complex to be efficiently optimized by the DB engine. Thus, the query rewriting technique needs to tackle two different issues: *(i)* a query translation problem that involves RDB-to-RDF mappings over arbitrary relational schemas, and *(ii)* a query optimization problem. There exist a number of systems and techniques related to this problem, such as the ones described in [12, 9, 34], However, each of these approaches has limitations that affect critical aspects of query answering over virtual RDF. These limitations include the generation of inefficient or even incorrect SQL queries, lack of formal background, and poor implementations. Moreover, some of them lack support for arbitrary DB schemas. since they do not support RDB to RDF mapping languages, such as, R2RML.

The approach presented in this paper, and depicted in Figure 1, deals with all the aforementioned issues. First, the SPARQL query and the R2RML mappings are translated into a Datalog program; the Datalog program is not meant to be executed, instead we view this program as a formal representation of the query and the mappings that we can manipulate and then transform into SQL. Second, we perform a number of structural and semantic optimizations on the Datalog program, including optimization with respect to database metadata. We do this by adapting well known techniques for optimization of logic programs and SQL query optimization. Once the program has been optimized the final step is to translate it to relational algebra/SQL, and to execute it over the relational database. The technique is able to deal with all aspects of the translation, including URI and RDF Literal construction, RDF typing, and SQL optimization. This is the technique implemented in the -ontop- system for OBDA, a mature open source system that is

Email addresses: mrodrig@us.ibm.com (Mariano Rodríguez-Muro[1]), mrezk@inf.unibz.it (Martin Rezk[1])

Mariano Rodríguez-Muro is currently working at IBM T.J. Watson Research Center.

http://ontop.inf.unibz.it/

currently being used in a number of projects and that currently outperforms other similar systems, sometimes by several orders of magnitude. -ontop- is available as a SPARQL endpoint, as a OWLAPI and Sesame query engine and as a Protege 4 plugin.

The contributions of this paper are four: *(i)* a formal approach for SPARQL-to-SQL translation that generates efficient SQL by adapting and combining optimization techniques from logic programming the query optimization; *(ii)* a rule based formalisation of R2RML mappings that can be integrated into our technique to support mappings to arbitrary database schemas; *(iii)* a discussion of the SQL features that are relevant in the context of SPARQL-to-SQL systems and that should be avoided to guarantee good performance in today's relational engines, together with experiments that validate these observations; *(iv)* an extensive evaluation comparing -ontop- with well known RDB2RDF systems and triple stores, showing that using the techniques presented here -ontop- can outperform them.

The rest of the paper is organized as follows: In Section 2 we briefly survey other works related to SPARQL-SQL translation. In Section 3 we introduce the necessary background. In Section 4 we present the core technique for translation of SPARQL to SQL. In Section 5 we show how to incorporate R2RML mappings into our approach. In Section 6 we provide a discussion on the SQL features that degrade performance of query execution. In Section 7 we describe how to optimise our technique with respect to the issues discussed in Section 7 by applying techniques from logic programming and SQL query optimization. In Section 8 we provide an evaluation of the performance of the technique. In Section 9 we conclude the paper. All proofs are given in the appendix.

2. Related Work

In this section we briefly survey related works regarding SPARQL query answering. We focus on two different but closely related topics: RDF stores and SPARQL to SQL translations.

RDF stores. Several RDF stores, such as RStar [23] and Virtuoso 6.1 [13], use a single table to store triples. This approach has the advantage that it is intuitive, flexible, and the mappings between the conceptual and data layer (if needed) are trivial. On the other hand such approach cannot use the known optimizations developed for normalized relational DBs—many of which are currently used in -ontop-. Our approach uses existing relational databases together with R2RML mappings to obtain a virtual representation of the RDF graph. In addition to the RDF stores mentioned above, we explore the commercial RDF stores Stardog and OWLIM more in detail in Section 8.

Stardog is a commercial RDF database developed by Clark&Parsia that supports SPARQL 1.1. Although it is a triplestore, this system uses query-rewriting techniques [25], but they do not translate SPARQL into SQL. Observe that the optimizations presented here and that are needed to produce an efficient SQL, might not be relevant for Stardog (and other triplestores) since the underlying backend is not a relational database.

Virtuoso 7 also provides column-wise compressed storage[14], which may be much faster than traditional row stores. -ontop- (and any other general SPARLQ-to-SQL techniques) may also benefit from the performance of column-stores that support SQL, e.g., MonetDB.

SPARQL-to-SQL. Regarding SPARQL-to-SQL translations, there have been several approaches in the literature, cf. [12, 9, 34]. In addition one can also include here translations from SPARQL to Datalog [27, 26, 2] given that: *(i)* SPARQL (under set semantics) has the same expressive power of non-recursive safe Datalog with default negation [2]; and *(ii)* any recursion-free safe Datalog program is equivalent to a SQL query [33]. The work in [27] extends and improve the ones in [26, 2] by modeling SPARQL 1.1 (under bag semantics, where duplicates are allowed), modeling non-safe queries, and modeling the W3C standard semantics to the SPARQL Optional. Since [26] was published before the publication of the SPARQL standard specification, the semantics presented there was not the same as in the standard. To keep the presentation simple, in this paper we will use the "academic" set semantics of SPARQL, as in [24, 26, 2]. We build and re-use several results from the works mentioned above, however we extend this line of research in several ways. First, we include R2RML mappings in the picture; second, we provide a (concrete) SQL translation from the Datalog program obtained from the input query; and third, we optimize and evaluate the performance of this approach. It is worth noticing that not any SQL query correctly translated from the Datalog program is acceptable, since *(i)* one has to deal the mismatch between types in SPARQL and SQL; and *(ii)* the syntactic form of SQL queries can severely affect their performance.

In [12, 9] the authors propose a translation function that takes a query and two many-to-one mappings: *(i)* a mapping between the triples and the tables, and *(ii)* a mapping between pairs of the form *(triple, pattern, position)* and relational attributes. Compared to that approach, -ontop- allows much richer mappings, in particular the W3C standard R2RML [10]. Moreover, these approaches assume that the underlying relational DB is denormalized, and stores RDF terms. The SPARQL generation technique presented in [12] lacks formal semantics. Another distinguishing feature of the work presented here is that it includes an extensive evaluation based on well-known benchmarks and large volumes of data.

The work in [9] was published before the publication of the SPARQL standard specification, thus the semantics it uses in the translation is not the same as in the standard document. The main difference is in the definition of the Optional algebra operator that in SPARQL 1.1 it has a non-compositional semantics with respect to the filter expression in the second argument. The work in [9] was recently extended in [28] to include R2RML

http://stardog.com/

Non-safe queries: queries with filter expressions that mention variables that do not occur in the graph pattern being filtered.

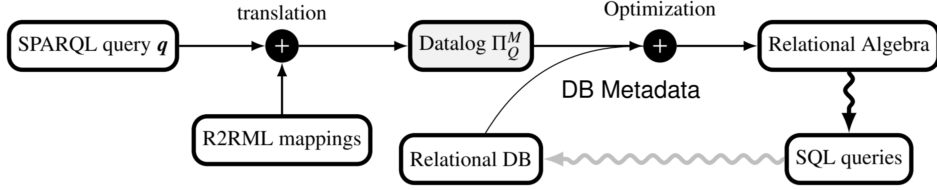


Figure 1: Proposed approach for SPARQL to optimized SQL through Datalog with R2RML mappings.

mappings in the picture. However, [28] did not update the semantic of the Optional algebra operator to make it compositional.

In [12, 9] they present an ad-hoc evaluation with a much smaller dataset. The work described in [34] also proposes a SPARQL-SQL translation, but exhibits several differences compared to our approach: it uses non-standard SQL constructs, it does not formally prove that the translation is correct, and it lacks in empirical results testing the proposed approach with real size DBs. Ultrawrap uses a view based technique for translating SPARQL to SQL [32]; however the optimization techniques used in the system appear to fail in several scenarios (see Section 4). We are also aware of other (non-published) techniques used in commercial and open source SPARQL-to-SQL engines, such as, D2RQ and VirtuosoRDF Views. We empirically show in Section 8 that the translation provided in this paper is more efficient than the ones implemented in those systems.

Several of these approaches have also discussed different optimizations to obtain more efficient SQL queries. We summarized the most relevant ones to this work in Figure 2. In Section 6 we described in detail the different optimizations performed by -ontop-. It is worth noticing that although when we say that a system do not perform a given optimization, we mean that such optimization has not been published as part of the system (to the best of our knowledge). In addition, when we say that a system (beside -ontop-) performs an optimization, it means that there is a publication that mentions it. However, often articles such as [32] [28] cover very briefly these topics omitting important details and issues that are critical for performance. For instance, [28] does not tackle URI templates in the optimization section, and [32] presents a technique that can not remove string concatenation from JOIN conditions when keys are missing or when the involved URI templates have different arity.

3. Preliminaries

In this section we review the material required for the presentation of the core SPARQL-to-SQL technique and the optimization techniques.

3.1. Logic Programs

We start by reviewing basic notions from standard logic programming [22] needed in following sections. Intuitively, a rule is a logic statement of the form:

$$\textbf{If} \text{ condition A holds } \textbf{then} \text{ conclude B} \tag{1}$$

Next we present the formal definitions.

Syntax. The language $\mathcal{L}$ in traditional logic programming consists of:

- A countably infinite set of variables $\mathcal{V}$.
- A countably infinite set of function symbols $\mathcal{F}$, where constants are treated as 0-arity function symbols.
- A countably infinite set of predicates $\mathcal{P}$.
- The symbols $\{\forall, \exists, \wedge, \rightarrow, \textbf{not}\}$

Terms and atoms are defined as usual in first order logic. We denote the set of all atoms in the language as $\mathbf{A}$. Atoms will be also called positive literals. The symbol **not** will be used for *default* negation. A literal is either an atom or its negation. Literals that do not mention **not** are said to be **not**-free. Otherwise we say they are **not**-literals.

A ***logic program*** is a collection of statements (called ***rules***) of the form

$$\forall \vec{x}\colon (l_0 \leftarrow l_1 \wedge \cdots \wedge l_m \wedge \textbf{not}\, l_{m+1} \wedge \cdots \wedge \textbf{not}\, l_n) \tag{2}$$

where each l_i is a literal, l_0 is **not**-free, and $\vec{x}$ are all the variables mentioned in $l_0 \dots l_n$. The literal l_0 is called the *head* of the rule. The set of literals $\{l_1, \dots, l_n\}$ is called the *body* of the rule. If the body is empty, then $\leftarrow$ can be dropped, and the rule is called a *fact*. We may also use the term *clause* to refer to a rule or a fact.

Given a rule r of the form (2), the sets $\{l_0\}$, $\{l_1 \dots l_m\}$, and $\{l_{m+1} \dots l_n\}$ are referred to as $head(r)$, $pos(r)$ and $neg(r)$ respectively. The set $lit(r)$ stands for $head(r) \cup pos(r) \cup neg(r)$.

As standard convention, (2) will be simply written as:

$$l_0 \leftarrow l_1, \dots, l_m, \textbf{not}\, l_{m+1}, \dots, \textbf{not}\, l_n \tag{3}$$

We may also replace the symbol $\leftarrow$ by :- . An expression—a rule, a program, or a literal—is called *ground* if it does not contain any variable.

Queries are statements of the form

$$\exists \bar{X}\colon l_1 \wedge ... \wedge l_m \tag{4}$$

where $l_1, ..., l_m$ are literals and $\bar{X}$ are all the variables mentioned in $l_1, ..., l_m$. The existential quantifier is usually omitted and comma is used often in lieu of the conjunction symbol $\wedge$.

Stable model semantics for Logic Programs. In this section, we review the main concepts of *stable model semantics* [15]. Intuitively, a stable model of a program is a set I of atoms such that: *(i)* for every rule in the program, if the condition of the rule is satisfied by I, then the conclusion of the rule is in the model; and *(ii)* I is minimal in the sense that removing an atom from I would violate some rule in the program.

Name	-ontop-	[12]	[9]	Morph[28]	Ultrawrap
Self-Join Elimination	Yes	No	No	Yes	Yes
Push Join into Union	Yes	No	No	No	No
Detection Unsat. Conditions	Yes	No	No	Yes	Yes
Remove URIs from Join	Yes	No	No	Yes	Yes

Figure 2: Optimisations presented in the OBDA literature

For the sake of simplicity, we assume that logic rules are of the form (3) and ground. Lifting to the non-ground case is done in a standard way.

The *Herbrand universe*, $\mathcal{U}$, is just the set of all constants in the language $\mathcal{L}$. The *Herbrand base*, $\mathcal{B}$, is a set of all ground literals in the language. Note that the Herbrand universe and Herbrand base are infinite, fixed, and depend only on the language $\mathcal{L}$.

Definition 1 (Herbrand interpretation). *A* **Herbrand interpretation***, $\mathcal{M}$, is consistent a subset of the Herbrand base.* □

Observe that under stable model semantics, interpretations are 2-valued. Satisfaction of a formula ϕ by Herbrand interpretation, $\mathcal{M}$, denoted $\mathcal{M} \models \phi$, is defined as follows:

- $\mathcal{M} \models l$, where l is a (**not** -free) literal, iff $l \in \mathcal{M}$.
- $\mathcal{M} \models \phi_1 \land \phi_2$, iff $\mathcal{M} \models \phi_1$ and $\mathcal{M} \models \phi_2$.
- $\mathcal{M} \models \textbf{not}\, \phi$, iff it is not the case that $\mathcal{M} \models \phi$.
- $\mathcal{M} \models r$, where r is a ground rule of the form (3), iff $l_0 \in \mathcal{M}$ whenever $\mathcal{M} \models l_1 \land \ldots l_m$ and $\mathcal{M} \models \textbf{not}\, (l_{m+1} \land \ldots l_n)$.

Given a **not** -free program Π, we write $\mathcal{M} \models \Pi$ if $\mathcal{M} \models r$ for every rule $r \in \Pi$. In this case we say that $\mathcal{M}$ is a ***stable model*** (a.k.a. answer set) of Π. It is known that every **not** -free program Π has a unique ***least model*** [3]—a model $\mathcal{M}_0$ such that for any other model $\mathbf{N}$ of Π, $l \in \mathcal{M}_0$ implies $l \in \mathbf{N}$ for any $l \in \mathcal{B}$.

To extend the definition of stable model (answer set) to arbitrary programs, take any program Π, and let $\mathcal{I}$ be a Herbrand interpretation in $\mathcal{L}$. The **reduct**, $\Pi(\mathcal{S})$, of Π relative to a subset of the Herbrand Base $\mathcal{S}$ is obtained from Π by first dropping every rule of the form (3) such that $\{l_{m+1}, \ldots, l_n\} \cap \mathcal{M} \neq \emptyset$; and then dropping the $\{l_{m+1}, \ldots, l_n\}$ literals from the bodies of all remaining rules. Thus $\Pi(\mathcal{M})$ is a program without default negation.

Definition 2 (Stable Model). *A Herbrand interpretation $\mathcal{M}$ is a **stable model** for Π if $\mathcal{M}$ is an answer set for $\Pi(\mathcal{M})$.* □

Observe that not every program has stable models, for instance, the following rule has not stable model.

$$p \leftarrow \textbf{not}\, p$$

Definition 3 (Entailment). *A program Π entails a ground literal l, written $\Pi \models l$, if l is satisfied by every stable model of Π.* □

Let Π be a program and q a query. (for simplicity we assume that q is a **not** -free literal), we say that the program Π's answer to q is *yes* if $\Pi \models l_i$ for every $i = 1 \ldots m$, *no* if $\Pi \models \textbf{not}\, l_i$ for some $i = 1 \ldots m$, and *unknown* otherwise.

Partial Evaluation. Partial evaluation is a logic programs technique in which, given a logic program Π, one computes a new program Π' that represents the *partial execution* of Π. This technique is used to either iteratively simplify the program to either compute the model of Π or to obtain a more efficient representation of the program. In this paper we use two notions of partial evaluation, i.e., partial evaluation with respect to a set of facts (used in the proofs of soundness and completeness) and partial evaluation with respect to a *goal* (i.e., query) which we use to optimize our Datalog programs for SQL efficiency. We now elaborate on both notions.

Partial Evaluation w.r.t. a goal. Let G be a query (a.k.a. *goal*) as defined in above (c.f. Equation 4). Given a program Π, intuitively the partial evaluation of Π produces a new program Π' that represents a pre-computation of Π needed to answer the goal G. Observe that Π' should still provide sound and complete answers with respect to Π, and that G should run more efficiently for Π' than for Π.

The notion of partial evaluation with respect to a goal is built on top of several other logic programming notions. We now start recalling the basic ones. All of these can be found in [21, 18].

Definition 4 (substitution). *A* substitution *θ is a finite set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$, where for each $i = 1, \ldots, n$:*

1. *x_i is a variable,*
2. *t_i is a term distinct from x_i*
3. *for each x_j ($j \neq i$) it holds that $x_i \neq x_j$*

Each element x_i/t_i is called a binding *for x_i.*

Definition 5 (instance). *Let $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ be a substitution and E be an expression. Then $E\theta$, the* instance *of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable x_i ($i = 1, \ldots, n$) in E by the term t_i.*

Definition 6 (unifier). *Let $\mathcal{S}$ be a nonempty set of expressions (terms, atoms or a literals). A substitution θ is called unifier of $\mathcal{S}$ if for every pair of expressions $E_1, E_2 \in \mathcal{S}$, it holds that $E_1\theta = E_2\theta$.*

Definition 7 (most general unifier). *A unifier θ of a set of expressions $\mathcal{S}$ is called* most general unifier *(mgu) of $\mathcal{S}$ if for every unifier τ of $\mathcal{S}$, every binding in θ is also in τ.*

We note that computing a mgu for a set of expressions can be done in linear time [21]. Now we introduce the important notions of partial evaluation the we exploit later. All these notions where introduced in [20].

Now define how new rules are computed from existing rules, a core step in SLD-resolution and the base for the computation of a partial evaluation.

Definition 8 (goal derivation). *Let G be the goal* $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ *and C be a* **not**-*free rule of the form*

$$A \leftarrow B_1, \ldots, B_q$$

Then G' is derived from G and C using the most general unifier (mgu) θ if the following conditions hold:

- A_m *is an atom in G, called the selected atom,*
- θ *is a mgu of* A_m *and A, and*
- G' *is the goal*

$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$$

where $(A_1, \ldots, A_n)\theta = A_1\theta, \ldots, A_n\theta$ *and* $A\theta$ *is the atom obtained from A applying the substitution θ*

That is, a goal derivation is obtained by computing a mgu θ between the selected atom and the head of another rule C, replacing the selected atom with the body of C, and applying θ to the resulting goal.

Now we introduce a key concept, SLD-Tree's. That is, the structure that represents a full SLDNF-resolution computation for a program. This structure is crucial since it allows us to manipulate the computation in an abstract way and describe its properties. The SLD-Tree nodes are *resultants*, which are defined next:

Definition 9 (resultant). *A* resultant *is an expression of the form*

$$Q_1 \leftarrow Q_2$$

where Q_i $(i = 1, 2)$ *is either absent or a conjunction of literals. All variables in* Q_1 *and* Q_2 *are assumed to be universally quantified.*

Definition 10 (SLD-tree). *Let* Π *be a program and let G be a goal. Then, a (partial) SLD-Tree of* $\Pi \cup \{G\}$ *is a tree satisfying the following conditions:*

- *Each node of the tree is a resultant,*
- *The root node is* $G\theta_0 \leftarrow G_0$, *where* $G\theta_0 = G_0 = G$ *(i.e.,* θ_0 *is the empty substitution),*
- *Let* $G\theta_0 \ldots \theta_i \leftarrow G_i$ *be a node at depth* $i \geq 0$ *such that* G_i *has the form* $A_i, \ldots, A_m, \ldots, A_k$, *and suppose that* A_m *is the selected atom. Then, for each input* **not**-*free rule* $A \leftarrow B_1, \ldots, B_q$ *such that* A_m *and A are unifiable with mgu* θ_{i+1}, *the node has a child*

$$G\theta_1\theta_2 \ldots \theta_{i+1} \leftarrow G_{i+1}$$

where G_{i+1} *is derived from* G_i *and* A_m *by using* θ_{i+1}, *i.e.,* G_{i+1} *has the form*

$$(A_1, \ldots, B_1, \ldots, B_q, \ldots, A_k)\theta_{i+1}$$

- *Nodes that are the empty* **not**-*free rule have no children.*

Given a branch of the tree, we say that it is a ***failing branch*** *if it ends in a node such that the selected atom does not unify with the head of any* **not**-*free rule. Moreover, we say that a* ***SLD-tree is complete*** *if all non-failing branches end in the empty* **not**-*free rule.*

Finally, given a node $Q\theta \leftarrow Q_n$ *at depth i, we say that the derivation of* Q_i *has length i with computed answer θ, where θ is the restriction of* $\theta_0, \ldots, \theta_i$ *to the variables in G, i.e., θ is the subset substitutions in* $\theta_0, \ldots, \theta_i$ *such that for each substitution domain is a variable in G.*

Now we define the notion of the *partial evaluation* (PE) of an atom and the partial evaluation of a query.

Definition 11 (partial evaluation (PE) of A in P). *Let* Π *be a program, A an atom, and* $\mathcal{T}$ *a SLD-tree for* $\Pi \cup \{\leftarrow A\}$. *Let* $G_1, \ldots, G_r$ *be a set of (non-root) goals in* $\mathcal{T}$ *such that each non-failed branch of* $\mathcal{T}$ *contains exactly one of them. Let* $R_i(i = 1, \ldots, r)$ *be the resultant of the derivation from* $\leftarrow A$ *down to* G_i *associated with the branch leading to* G_i. *Then*

- *the set of resultants* $\pi = \{R_1, \ldots, R_r\}$ *is a PE of A in* Π. *These resultants have the following form*

$$R_i = A\theta_1 \leftarrow Q_i (i = 1, \ldots, r)$$

where we have assumed $G_i = \leftarrow Q_i$

Definition 12 (partial evaluation of P w.r.t. A). *Let* Π *be a* **not**-*free program and A atom, a* partial evaluation of Π with respect to *A is a program* Π' *obtained by replacing the set of* **not**-*free rules in* Π *whose head contains A (called the* partially evaluated predicate*) with a partial evaluation of A in* Π.

For an example of this process see Section 7.

Partial Evaluation w.r.t. a set of literals. Next we will explain how partial evaluation is used to iteratively compute the intended model of a program. In the following, we will work with *stratified* programs; these have certain properties that we will use through out the paper and we now introduce. Intuitively, a program Π is stratified if it can be partitioned or split into disjoint strata $\Pi_0 \ldots \Pi_n$ such that: *(i)* $\Pi = \Pi_0 \cup \cdots \cup \Pi_n$, *(ii)* Π_0 is **not**-free, and *(iii)* all the negative literals in Π_i $(0 < i \leq n)$ are only allowed to refer to predicates that are already defined in Π_{i-1}. Intuitively, in a *stratified* program Π, the *intended model* is obtained via a sequence of bottom-up derivation steps. In the first step, Π is split into stratums. The first stratum is a bottom part that does not contain negation as failure. Since this sub-program is positive, it has a unique stable model. Having substituted the values of the bottom predicates in the bodies of the remaining rules, Π is reduced to a program with fewer strata.

By applying the *splitting* step several times, and computing every time the unique stable model of a positive bottom, we will arrive at the *intended model* of Π. Further details can be found in [29].

Definition 13 (Splitting Set [19]). *A **splitting set** for a program Π is any set U of literals such that for every rule $r \in \Pi$, if $head(r) \cap U \neq \emptyset$ then $lit(r) \subset U$. If U is a splitting set for Π, we also say that U **splits** P. The set of rules $r \in \Pi$ such that $lit(r) \subset U$ is called the **bottom** of Π relative to the splitting set U and denoted by $b_U(\Pi)$. The subprogram $\Pi \setminus b_U(\Pi)$ is called the **top** of Π relative to U.* □

Definition 14 (Partial Evaluation w.r.t. a set of literals). *The* partial evaluation of a program Π with splitting set U with respect to a set of literals X, *is the program $e_U(\Pi, X)$ defined as follows. For each rule $r \in \Pi$ such that*

$$(pos(r) \cap U) \subset X \text{ and } (neg(r) \cap U) \cap X = \emptyset$$

put in $e_U(\Pi, X)$ all the rules r' that satisfy the following property

$$\begin{array}{lcl} head(r') & = & head(r) \\ pos(r') & = & pos(r) \setminus U \\ neg(r') & = & neg(r) \setminus U \end{array}$$

□

Definition 15 (Solution). *Let U be a splitting set for a program Π. A **solution** to Π with respect to U is a pair (X, Y) of literals such that*

- *X is an stable model for $b_U(\Pi)$*
- *Y is an stable model for $e_U(\Pi \setminus b_U(\Pi), X)$*
- *$X \cup Y$ is consistent.* □

Example 1. *[5] Consider the following program Π :*

$$\begin{array}{l} a \leftarrow b, \textbf{not}\, c \\ b \leftarrow c, \textbf{not}\, a \\ c \leftarrow \end{array}$$

The set $U = \{c\}$ splits Π; the last rule of Π belongs to the bottom and the first two rules from the top. Clearly, the unique stable model for the bottom of Π is $\{c\}$. The partial evaluation of the top part of Π consists in dropping its first rule, because the negated subgoal c makes it useless, and in dropping the trivial positive subgoal c in the second rule. The result of simplification is the program consisting of one rule

$$b \leftarrow \textbf{not}\, a \tag{5}$$

The only stable model for P can be obtained by adding the only stable model for (5), which is $\{b\}$, to the stable model for the bottom used in the evaluation process, $\{c\}$. □

Proposition 1. *[19] Let U be a splitting set for a program Π. A set S of literals is a consistent stable model for Π if and only if $S = X \cup Y$ for some solution (X, Y) of Π with respect to U.*

3.2. Relational Algebra and SQL

Relational Algebra is a formalism for manipulating relations (e.g., sets of tuples). It's mainly used as the formal grounds for SQL and we will use it to represent SQL queries (there is a direct correspondence from one to the other). We now introduce the basic notions of relational algebra. The operations in relational algebra that takes one or two relations as inputs and produce a new relation as a result. These operations enable users to specify basic retrieval request.

In this section we use the following notation: r, r_1, r_2 denote relational tables, t, t_2 denote tuples, $c1, c2$ denote attributes, $v_1 \ldots v_n$ denote domain elements, p denotes a filter condition, jn denotes join condition of the form $r_1.c_i = r_2.c_j \ldots r_1.c'_i = r_2.c_2{}'$ and the function $col(r)$ returns the set of attributes of r.

The following are the relational algebra operators used in this paper:

Union ($\cup$): This binary operator, written as, $r_1 \cup r_2$, requires that the two relations involved must be union-compatible, that is, the two relations must have the same set of attributes. The result includes all tuples that are in r_1 or in r_2.

$$r_1 \cup r_2 = \{t \mid t \in r_1 \text{ or } t \in r_2\}$$

Cartesian Product ($\times$): This binary operator, written as, $r_1 \times r_2$, requires that the two relations involved must have disjoint set of attributes. The result includes all tuples that are in r_1 or in r_2.

$$r_1 \times r_2 = \{t_1, t_2 \mid t_1 \in r_1 \text{ and } t_2 \in r_2\}$$

Difference ($\setminus$): This binary operator, written as, $r_1 \setminus r_2$, requires that the two relations involved must be union-compatible, that is, the two relations must have the same set of attributes. The result includes all tuples that are in r_1 but not in r_2.

$$r_1 \setminus r_2 = \{t \mid t \in r_1 \text{ and } t \notin r_2\}$$

Selection (σ): This operator is used to choose a subset of the tuples (rows) from a relation that satisfies a selection condition, acting as a filter to retain only tuples that fulfills a qualifying requirement.

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

Rename (ρ): This is a unary operation written as, $\rho_{c_1/c_2}(r)$, where the result is identical to r except that the c_1 attribute in all tuples is renamed to a c_2 attribute.

Projection (Π): This operator is used to reorder, select and filter out attributes from a table.

$$\Pi_{c_1 \ldots c_k}(r) = \{v_1 \ldots v_k \mid v_k \ldots v_n \in r\}$$

In order to ease the presentation, we will often mimic SQL and include the renaming in the projection using *AS* statements. Thus, we write $\Pi_{c_1 \text{ } AS \text{ } c_2}(r)$ to denote $\rho_{c_1/c_2}(r)$. We will also overload the projection with statements of the form $\Pi_{\text{constant } AS \text{ } c_2}(r)$ where constant is null, or an string, or a concatenation of an string and an attribute. Observe that this second operation can be easily encoded in relational algebra using auxiliary tables. For instance, $\Pi_{\text{constant } AS \text{ } c_2}(r)$, can be encoded

as $\rho_{nullttr/c_2}\Pi_{attr(r)\setminus c_2} r \times$ NullTable where NullTable is a table with a single attribute *nullttr* and a single null record.

Natural join ($\bowtie$): This is a binary operator written as, $r_1 \bowtie r_2$, where the result is the set of all combinations of tuples in r_1 and r_2 that are equal on their common attribute names.

$$r_1 \bowtie_{jn} r_2 = \Pi_{sc}(\sigma_{jn}(r_1 \times r_2))$$

Left join ($⟕$): This is a binary operator written as, $r_1 ⟕ r_2$, where the result is the set of all combinations of tuples in R and S that are equal on their common attribute names, in addition (loosely speaking) to tuples in r_1 that have no matching tuples in r_2.

$$\begin{array}{rcl} r_1 ⟕_{jn} r_2 & = & (r_1 \bowtie_{jn} r_2) \cup \\ & & ((r_1 \setminus \Pi_{col(r_1)}(r_1 \bowtie_{jn} r_2)) \times \\ & & \text{NullTable}^{attr(r_2)\setminus attr(r_1)}) \end{array}$$

where $\text{NullTable}^{attr(r_2)\setminus attr(r_1)}$ is a table with a attributes $attr(r_2) \setminus attr(r_1)$ and a single record consisting only on null values.

Recall that every relational algebra expression is equivalent to a SQL query. Further details can be found in [1].

3.3. SPARQL.

For formal purposes we will use the algebraic syntax of SPARQL similar to the ones in [26, 2] and defined in the standard. However, to ease the understanding, we will often use graph patterns (the usual SPARQL syntax) in the examples. It is worth noticing, that although in this paper we restrict ourselves to SELECT queries, in -ontop- we also allow ASK, DESCRIBE and CONSTRUCT queries, which can be reduced or implemented using SELECT queries.

The SPARQL language that we consider contains the following pairwise disjoint countably infinite sets of symbols: **I**, denoting the IRIs, **B**, denoting blank nodes, **L**, denoting RDF literals; and **V**, denoting variables.

The *SPARQL algebra* is constituted by the following graph pattern operators (written using prefix notation): *BGP* (basic graph pattern), *Join*, *LeftJoin*, *Filter*, and *Union*. A *basic graph pattern* is a statement of the form:

$$BGP(s, p, o)$$

where $s \in \mathbf{I} \cup \mathbf{B} \cup \mathbf{V}$, $p \in \mathbf{I} \cup \mathbf{V}$, and $o \in \mathbf{I} \cup \mathbf{B} \cup \mathbf{L} \cup \mathbf{V}$. In the standard, a BGP can contain several triples, but since we include here the join operator, it suffices to view BGPs as the result of $\bowtie$ of its constituent triple patterns. Observe that the only difference between blank nodes and variables in BGPs, is that the former do not occur in solutions. So, to ease the presentation, we assume that BGPs contain no blank nodes. The remaining algebra operators are:

- Join(pattern,pattern)
- LeftJoin(pattern,pattern,expression)
- Union(pattern,pattern)
- Filter(pattern,expression)

and can be nested freely. Each of these operators returns the result of the sub-query it describes. Details on how to translate SPARQL queries into SPARQL algebra can be found in the W3C specification, and, in addition, several examples will be presented along the paper.

Note. Converting Graph Patterns. It is critical to notice that graph patterns are not translated straightforwardly into algebra expressions. There is a pre-processing of the graph patterns where filter expressions are either moved to the top of graph, or absorbed by LeftJoin expressions. Details can be found in the SPARQL 1.0 specification.

A SPARQL query is a graph pattern P with a solution modifier, which specifies the answer variables, that is, the variables in P whose values should be in the output. In this work we ignore this solution modifiers for simplicity.

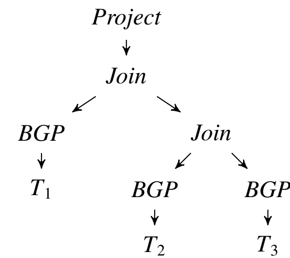
Definition 16 (SPARQL Query). *Let P be a SPARQL algebra expression, V a set of variables occurring in P, and G a set of RDF triples. Then a* query *is a triple of the form* (V, P, G). □

We will often omit specifying V and G when they are not relevant to the problem at hand.

Example 2. *Consider the following SPARQL query Q:*

```
SELECT ?x ?z ?w WHERE
{ ?x :knows ?y . ?y :email ?z . ?y :site ?w . }
```

This query is then translated into an SPARQL algebra expression that has the following tree shape:



where T_1, T_2 *and* T_3 *represent* (x,'knows',y), (x,'email',z), *and* (x,'site',w) *respectively.* □

Semantics. Now we briefly introduce the formal set semantics of SPARQL as specified in [26] with the difference that we updated the definition of the LeftJoin to match the published standard specifications. The result is a semantic which is more strict as the one in [27] and the standard W3C semantics in the sense that:

1. We do not allow joins through null values.
2. We work with set semantics opposed to bag semantics.
3. We do not actually model the "error" value of filter expressions. Observe that this is not a limitation in practice since, as specified by the standard, FILTERs eliminate any solutions that, when substituted into the expression, either result in an effective boolean value of false or an error.

http://www.w3.org/TR/rdf-sparql-query/#sparqlAlgebra

http://www.w3.org/TR/rdf-sparql-query#convertGraphPattern

It is worth noticing that constraints (1) and (2) can be actually modelled inside SPARQL by using Select Distinct and adding BIND filters to avoid null bindings in the variables occurring in joins and filter expressions. This means that we work with a fragment of all the possible SPARQL queries, but also implies we can still re-use the results in [27] regarding the SPARQL-Datalog translation.

Intuitively, when a query is evaluated, the result is a set of *substitutions* of the variables in the graph pattern for symbols in $(\mathbf{I} \cup \mathbf{L} \cup \{null\})$. We now provided the necessary definitions for this purpose.

Let T_{null} denote the following set $(\mathbf{I} \cup \mathbf{L} \cup \{null\})$.

Definition 17 (Substitution). *A **substitution**, θ, is a partial function*

$$\theta\colon \boldsymbol{V} \mapsto T_{null}$$

The domain of θ, denoted by dom(θ), is the subset of $\boldsymbol{V}$ *where θ is defined. Here we write substitutions using postfix notation.* □

Definition 18 (Union of Substitution). *Let θ_1 and θ_2 be substitutions, then $\theta_1 \cup \theta_2$ is the substitution obtained as follows:*

$$x(\theta_1 \cup \theta_2) = \begin{cases} x(\theta_1) \text{ if } x(\theta_1) \text{ is defined and } x(\theta_2) \in \{null, x(\theta_1)\} \\ \text{else: } x(\theta_2) \text{ if } x(\theta_2) \text{ is defined and } x(\theta_1) = null \\ \text{else: undefined} \end{cases}$$

□

Definition 19 (Compatibility). *Two substitutions θ_1 and θ_2 are **compatible** when*

1. *for all $x \in dom(\theta_1) \cap dom(\theta_2)$ it holds that $x(\theta_1 \cup \theta_2) \neq null$.*
2. *for all $x \in dom(\theta_1) \cap dom(\theta_2)$ it holds that $x(\theta_1) = x(\theta_2)$.*

□

Definition 20 (Evaluation of Filter Expressions). *Let R be a filter expression. Let v, u be variables, and $c \in B \cup \mathcal{I} \cup L$. The evaluation of R on a substitution θ returns one of three values $\{\top, \bot, \epsilon\}$ and it is defined in Figure 3.* □

In the following we describe the semantics of the SPARQL algebra.

Definition 21. *Let Ω_1 and Ω_2 be two sets of substitutions over domains D_1 and D_2 respectively. Then*

$$\begin{array}{lcl} \Omega_1 \bowtie \Omega_2 & = & \{\theta_1 \cup \theta_2 \mid \theta_1 \in \Omega_1, \theta_2 \in \Omega_2 \text{ are compatible}\} \\ \Omega_1 \cup \Omega_2 & = & \{\theta \mid \exists \theta_1 \in \Omega_1 \text{ with } \theta = \theta_1^{D_1 \cup D_2} \text{ or} \\ & & \quad \exists \theta_2 \in \Omega_2 \text{ with } \theta = \theta_2^{D_1 \cup D_2}\} \\ \Omega_1 -_R \Omega_2 & = & \{\theta \mid \theta \in \Omega_1 \text{ and for all } \theta_2 \in \Omega_2 \\ & & \text{either } \theta \text{ and } \theta_2 \text{ are not compatible} \\ & & \text{or } \theta \text{ and } \theta_2 \text{ are compatible and } R(\theta \cup \theta_2) = \bot\ \} \end{array}$$

□

The semantics of a algebra expression P over dataset G is defined next.

Definition 22 (Evaluation of Algebra Expressions).

$$\begin{array}{rcl} \| BGP(t) \| & = & \{\theta \mid dom(\theta) = vars(P) \text{ and } t\theta \in G\} \\ \| Join(P_1, P_2) \| & = & \| P_1 \| \bowtie \| P_2 \| \\ \| Union(P_1, P_2) \| & = & \| P_1 \| \cup \| P_2 \| \\ \| LeftJoin(P_1, P_2, R) \| & = & \| Filter(Join(P_1, P2), R) \| \cup \\ & & (\| P_1 \| -_R \| P2 \|) \\ \| Filter(R, P_1) \| & = & \{\theta \in \| P \| \mid R\theta = \top\} \end{array}$$

where R is a FILTER expression. □

Definition 23 (Evaluation of Queries). *Let $Q = (V, P, G)$ be a SPARQL query, and θ a substitution in $\| P \|$, then we call the tuple $V[(V \setminus vars(P)) \mapsto null]\theta$ a solution tuple of Q.* □

3.4. SPARQL to Executable Datalo

The following technique allows to translate SPARQL queries (and RDF data) into a Datalog program. The technique was introduced in [27] and intuitively, works as follows. We take the SPARQL algebra tree of a SPARQL query and generate rules for each node in the tree, starting from the root. The rules we generated for a given node encode the semantics of the operator (the node). The head of the rules project the bindings that the corresponding SPARQL algebra operator should return, the body of the rules implements the operation itself. A program is generated by recursively translating each node, and their children, into rules. The leafs of the SPARQL algebra trees are always access to the RDF data, hence, the corresponding Datalog rules must do the same. This is done by defining a ternary predicate named *triple* that serves as container for all the RDF facts. Once the query is translated, all RDF triples can be translated into fact rules (tuples for the *triple* relation) and the program can be delegated to a Datalog engine for execution.

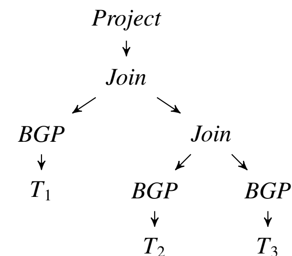
From this technique, in this paper we will reuse the technique to translate SPARQL queries into Datalog, hence, we introduce it here. We present a simplified version of the one in [27] since at the moment we tackle SPARQL 1.0 and not 1.1. Moreover, to ease the presentation, we kept some of the notation used in [26] instead of the one in [27].

Before introducing the formal definition, we will give an example to give the intuition behind.

Example 3. *Consider the following SPARQL query Q in Example 2. For convenient reference we reproduce it here:*

```
SELECT ?x ?z ?w WHERE
{ ?x :knows ?y . ?y :email ?z . ?y :site ?w . }
```

This query is then translated into an SPARQL algebra expression that has the following tree shape:

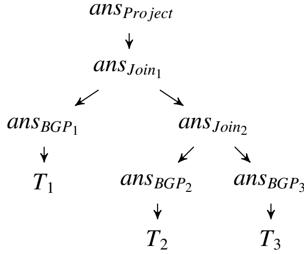


$$R = \begin{cases} isBLANK(v)\theta & = \begin{cases} \top \text{ if } v \in dom(\theta) \text{ and } v\theta \in B \\ \epsilon \text{ if } v \notin dom(\theta) \text{ or } v\theta = null \\ \bot \text{ otherwise} \end{cases} \\ isIRI(v)\theta & = \begin{cases} \top \text{ if } v \in dom(\theta) \text{ and } v\theta \in L \\ \epsilon \text{ if } v \notin dom(\theta) \text{ or } v\theta = null \\ \bot \text{ otherwise} \end{cases} \\ (v = c)\theta & = \begin{cases} \top \text{ if } v \in dom(\theta) \text{ and } v\theta = c \\ \epsilon \text{ if } v \notin dom(\theta) \text{ or } v\theta = null \\ \bot \text{ otherwise} \end{cases} \end{cases} \quad R = \begin{cases} (v = u)\theta & = \begin{cases} \top \text{ if } v \in dom(\theta) \text{ and } v\theta = u\theta \\ \epsilon \text{ if } v \text{ or } u \notin dom(\theta) \text{ or } v\theta \text{ or } u\theta = null \\ \bot \text{ otherwise} \end{cases} \\ (R_1 \wedge R_2)\theta & = \begin{cases} \top \text{ if } R_1\theta = \top \wedge R_2\theta = \top \\ \epsilon \text{ if } R_1\theta = \epsilon \vee R_2\theta = \epsilon \\ \bot \text{ otherwise} \end{cases} \\ (R_1 \vee R_2)\theta & = \begin{cases} \top \text{ if } R_1\theta = \top \vee R_2\theta = \top \\ \epsilon \text{ if } R_1\theta = \epsilon \wedge R_2\theta = \epsilon \text{ and } R_1\theta \neq \top \vee R_2\theta \neq \top \\ \bot \text{ otherwise} \end{cases} \end{cases}$$

Figure 3: Evaluation of R on a substitution θ

where T_1, T_2 and T_3 represent $(\texttt{x}, '\texttt{knows}', \texttt{y})$, $(\texttt{x}, '\texttt{email}', \texttt{z})$, *and* $(\texttt{x}, '\texttt{site}', \texttt{w})$ *respectively.*

To map this algebra expression into Datalog we will create one predicate symbol for each operator node in the tree, the predicate will have the form ans_{op} where ans stands for answer. *Now, if we map the dependency graph of these new predicate symbols, we would get a tree as follows:*



The technique introduced in [27] would then produce the following Datalog program.

$$\begin{aligned} \texttt{ans}_{\texttt{Project}}(\texttt{x},\texttt{w},\texttt{z}) &\;\text{:-}\; \texttt{ans}_{\texttt{Join}_1}(\texttt{x},\texttt{y},\texttt{z},\texttt{w}) \\ \texttt{ans}_{\texttt{Join}_1}(\texttt{x},\texttt{y},\texttt{w},\texttt{z}) &\;\text{:-}\; \texttt{ans}_{\texttt{BGP}_1}(\texttt{x},\texttt{y}), \texttt{ans}_{\texttt{Join}_2}(\texttt{y},\texttt{w},\texttt{z}) \\ \texttt{ans}_{\texttt{BGP}_1}(\texttt{x},\texttt{y}) &\;\text{:-}\; \texttt{triple}(\texttt{x},'\texttt{knows}',\texttt{y}) \\ \texttt{ans}_{\texttt{Join}_2}(\texttt{y},\texttt{w},\texttt{z}) &\;\text{:-}\; \texttt{ans}_{\texttt{BGP}_2}(\texttt{y},\texttt{z}), \texttt{ans}_{\texttt{BGP}_3}(\texttt{y},\texttt{w})) \\ \texttt{ans}_{\texttt{BGP}_2}(\texttt{y},\texttt{z}) &\;\text{:-}\; \texttt{triple}(\texttt{x},'\texttt{email}',\texttt{z}) \\ \texttt{ans}_{\texttt{BGP}_3}(\texttt{y},\texttt{w}) &\;\text{:-}\; \texttt{triple}(\texttt{x},'\texttt{site}',\texttt{w}) \end{aligned}$$

Note how we have one rule for each operator, and in the rule for each operator, we refer to the predicates over which the current node depends. For example, since our top Join operation depends on the bindings of the leftmost BGP and right most join, the rules for ans_{Join_1} reflect this by making reference to the predicates ans_{BGP_1} and ans_{Join_2}.

Now let us proceed with the formal definition. Recall that given two tuples of variables V and V', $V[V' \mapsto c]$ means that all the variables in $V \cap V'$ are replaced by c in V.

Definition 24 (SPARQL-Datalog). *Let $Q = (V, P, G)$, be a SPARQL query. The translation of this query to a logic program Π_Q is defined as follows:*

$$\Pi_Q = \{G\} \cup \tau(V, P)$$

Where $\{G\} = \{triples(s, p, o) \mid (s, p, o) \in G\}$. The first set of facts brings the data from the graph, the second is the actual translation of the graph SPARQL query that is defined recursively in Fig. 4. □

Intuitively, the $LT()$ operator disassembles complex filter expressions that includes boolean operators such as $\neg$, $\wedge$, $\vee$. The LT rewrite proceeds as follows: Complex filters involving $\neg$ are transformed by standard normal form transformations into negation normal form such that negation only occurs in front of atomic filter expressions. Conjunctions of filter expressions are simply disassembled to conjunctions of body literals, disjunctions are handled by splitting the respective rule for both alternatives in the standard way. Expressions are translated as follows:

- $E = Bound(v)$ is translated to **not** $v = null$,
- $E = \neg Bound(v)$ is translated to $v = null$,
- $E = isBlank(v)/isIRI(v)/isLiteral(v)$ are translated to their corresponding external atoms.

Observe that in rules (2) and (6) prevent null-bindings, and filter expressions involving null values.

As the original SPARQL-Datalog translations, we require *well-designed* queries [26]. This constraint imposes a restriction over the variables occurring in LeftJoin (Optional) and Unions operators.

Definition 25 (UNION-free well-designed graph pattern). *An UNION-free query Q is **well-designed** if for every occurrence of a sub-pattern $P' = LeftJoin(P_1, P_2)$ of P and for every variable v occurring in P, the following condition holds: if v occurs both in P_2 and outside P' then it also occurs in P_1.* □

Definition 26 (Well-designed). *A query Q is **well-designed** if the condition from Definition 25 holds and additionally for every occurrence of a sub-pattern $P' = Union(P_1, P_2)$ of P and for every variable v occurring in P' , the following condition holds: if v occurs outside P then it occurs in both P_1 and P_2.* □

Proposition 2 (Soundness and Completeness [27]). *Let Q be a well-designed SPARQL query and let Π_Q be the Datalog translation of Q. Then for each atom of the form $answer_P(\vec{s})$ in the unique answer set M of Π_Q, $\vec{s}$ is a solution tuple of the subquery P in Q. In addition, all solution tuples in Q are represented by the extension of the predicate $answer_Q$ in M.*

$$
\begin{array}{lcll}
\tau(V, \mathtt{BGP(s,p,o)}) & = & \mathtt{answer}_{\mathtt{BGP(s,p,o)}}(\overline{\mathtt{V}}) \text{ :- } \mathtt{triple(s,p,o)}. & (1)\\
\tau(V, \mathtt{Join(P_1,P_2)}) & = & \tau(vars(P_1), P_1) \cup \tau(vars(P_2), P_2) \cup & (2)\\
 & & \mathtt{answer}_{\mathtt{Join(P_1,P_2)}}(\overline{\mathtt{V}}) \text{ :- } (\mathtt{answer}_{\mathtt{P_1}}(\overline{\mathtt{vars(P_1)}}), \mathtt{answer}_{\mathtt{P_2}}(\overline{\mathtt{vars(P_2)}})) & \\
 & & \bigwedge_{v \in vars(P_1) \cap vars(P_2)} \mathbf{not}\ \mathtt{v = null}. & \\
\tau(V, \mathtt{Union(P_1,P_2)}) & = & \tau(vars(P_1), P_1) \cup \tau(vars(P_2), P_2) \cup & \\
 & & \mathtt{answer}_{\mathtt{Union(P_1,P_2)}}(\overline{\mathtt{V[V \setminus vars(P_1) \to null]}}) \text{ :- } \mathtt{answer}_{\mathtt{P_1}}(\overline{\mathtt{vars(P_1)}}). & (3)\\
 & & \mathtt{answer}_{\mathtt{Union(P_1,P_2)}}(\overline{\mathtt{V[V \setminus vars(P_2) \to null]}}) \text{ :- } \mathtt{answer}_{\mathtt{P_2}}(\overline{\mathtt{vars(P_2)}}). & \\
\tau(V, \mathtt{LeftJoin(P_1,P_2,E)}) & = & \tau(vars(P_1), P_1) \cup \tau(vars(P_2), P_2) \cup & (4)\\
 & & \mathtt{answer}_{\mathtt{LeftJoin(P_1,P_2)}}(\mathtt{V}) \text{ :- } \mathtt{answer}_{\mathtt{P_1}}(\mathtt{vars(P_1)}), \mathtt{answer}_{\mathtt{P_2}}(\mathtt{vars(P_2)}), \mathtt{E} & \\
 & & \mathtt{answer}_{\mathtt{LeftJoin(P_1,P_2)}}(\mathtt{V[vars(P_2) \setminus vars(P_1) \mapsto null]}) \text{ :- } \mathtt{answer}_{\mathtt{P_1}}(\mathtt{vars(P_1)}), & \\
 & & \qquad \mathtt{answer}_{\mathtt{P_2}}(\mathtt{vars(P_2)}), \mathbf{not}\ \mathtt{E} & \\
 & & \mathtt{answer}_{\mathtt{LeftJoin(P_1,P_2)}}(\mathtt{V[vars(P_2) \setminus vars(P_1) \mapsto null]}) \text{ :- } \mathtt{answer}_{\mathtt{P_1}}(\mathtt{vars(P_1)}), & \\
 & & \qquad \mathbf{not}\ \mathtt{answer}_{\mathtt{LJoin(P_1,P_2)}}(\mathtt{vars(P_1)}) & \\
 & & \mathtt{answer}_{\mathtt{LJoin(P_1,P_2)}}(\mathtt{vars(P_1)}) \text{ :- } \mathtt{answer}_{\mathtt{P_1}}(\mathtt{vars(P_1)}), \mathtt{answer}_{\mathtt{P_2}}(\mathtt{vars(P_2)}) & \\
\tau(V, \mathtt{Filter(E,P)}) & = & \tau(vars(P), P) \cup & (5)\\
 & & LT(\mathtt{answer}_{\mathtt{Filter(E,P)}}(\overline{\mathtt{V}}) \text{ :- } (\mathtt{answer}_{\mathtt{P_1}}(\overline{\mathtt{vars(P_1)}}, \mathtt{E})) & \\
 & & \bigwedge_{v \in vars(E)} \mathbf{not}\ \mathtt{v = null}. &
\end{array}
$$

Figure 4: Translation SPARQL-Datalog first presented in [26] and extended in [27]

3.5. Datalog to SQL

Recall that safe Datalog with negation and without recursion is equivalent to relational algebra [33]. From the previous section one can see that it is exactly the fragment of Datalog that we are working with. Therefore, it follows that any Datalog program obtained from the translation of a well-designed SPARQL query can translated to SQL.

3.6. R2RML

R2RML is a language that allows to specify mappings from relational databases to RDF data. The mappings allow to view the relational data in the RDF data model using a structure and vocabulary of the mapping author's choice .

An R2RML mapping is expressed as a RDF graph (in Turtle syntax). The graph is not arbitrary, a wellformed mapping consists of one or more trees called *triple maps* with a structure as shown in Figure 5. Each tree has a root node, called *triple map node*, which is connected to exactly one *logical table* node, one *subject map* node and one or more *predicate object map* nodes.

Example 4. *Let DB be a database composed by the table* `stud` *with columns* [<u>`id`</u>, `name`, `course`] *(primary keys are underlined) and the table* `course` *with columns* [<u>`id`</u>, `name`]. *Let DB contain the following data:*

id	*name*	*course*
20	*"John"*	*1*
21	*"Mary"*	*1*

id	*name*
1	*"SWT 101"*

Suppose that the desired RDF triples to be produced from these database are as follows:

```
:stud/20 rdf:type :Student ; :name "John" .
:stud/21 rdf:type :Student ; :name "Mary" .
:stud/20 :takes :course/1 .
:stud/21 :takes :course/1 .
:course/1 rdf:type :Course ; :name "SWT 101" .
```

http://www.w3.org/TR/r2rml/

The following R2RML mapping produces the desired triples:

```
_:m1 a rr:TripleMap;      # First triple map
 rr:logicalTable [ rr:tableName "stud" ] ;
 rr:subjectMap [ rr:template ":stud/{id}" ;
                rr:class :Student ] ;
 rr:predicateObjectMap [ rr:predicate :name ;
           rr:objectMap  [ rr:column "name"]].
 rr:predicateObjectMap [ rr:predicate :takes ;
   rr:objectMap  [ rr:parentTriplesMap _:m2 ;
   rr:joinCondition [ rr:child "ID"; rr:parent "ID" ]].

_:m2 a rr:TripleMap;      # Second triple map
 rr:logicalTable [ rr:tableName "course" ] ;
 rr:subjectMap [ rr:template ":course/{id}" ;
                rr:class :Course ] ;
 rr:predicateObjectMap [ rr:predicate :name ;
  rr:objectMap [ rr:column "name"]].
```

This R2RML mapping contains two triple maps. Intuitively, each triple map states how to construct a set of triples (subject, predicate, object) using i) the data from the logical table (which can be a table, view or SQL query), ii) the subject URI specified by the subject map node, and iii) the predicates and objects specified by each of the predicate object map nodes. In this particular case, the first 4 triples are entailed by the triple map that starts with node _:m1, and the last triple is entailed by _:m2.

Note that the mapping constructs URI's out of values from the DB using templates that get instantiated with the data from the columns of the logical table. Also, the mapping uses a custom vocabulary (:Student, :Course, :takes and :name).

Having introduced the core idea behind R2RML mappings we now introduce the core definitions and assumptions of R2RML that are relevant for the work presented in this paper. For further detail we refer the reader to the official R2RML specification.

The *logical table* of a triple map is a tabular SQL query result that is to be mapped to RDF triples. It may be either (i) an SQL base table or view, (i) or an R2RML view. A *logical table row* is a row in a logical table.

An *SQL base table or view* is a logical table containing SQL data from a database table or view in the input

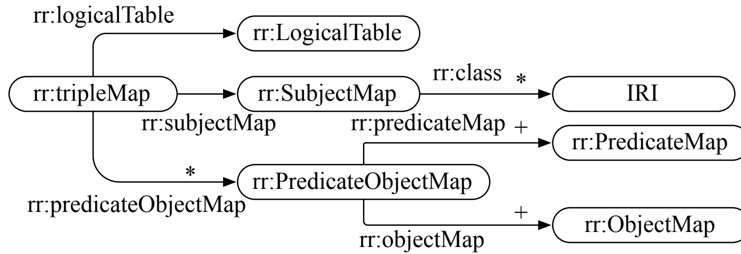


Figure 5: A well formed R2RML mapping node

database and is represented by a resource that has exactly one `rr:tableName` property with the string denoting the table or view name.

An *R2RML view* is a logical table whose contents are the results of executing a SQL query against the input database. It is represented by a resource with exactly one `rr:sqlQuery` property whose value is a SQL query string.

A *logical table* has an *effective SQL query* that produces the results of the logical table.

The *effective SQL query* of a table or view is `SELECT * FROM {table}` where *{table}* is replaced with the table or view name. The effective SQL query of an R2RML view is the value of its `rr:sqlQuery` property.

A *triple map* specifies how to translate each row of a logical table to zero or more RDF triples. Given a row, all triples generated from it share the same subject. The triple map has exactly one `rr:logicalTable` property and one *subject map* that specifies how to generate the subject for the triples generated by a row of the logical table. Last, a triple map may have zero or more *predicate object maps* specified with the `rr:predicateObjectMap` property. These specifies pairs of *predicate maps* and *object maps*, that together with the subject generated by the *subject map*, for the RDF triples for each row. Predicate, object and subject maps are constructed using *term maps*. A term map specifies what is the RDF term used for a subject, predicate or object and may be either a *RDF constant (i.e., a URI o RDF Value)*, a *column reference* or a *URI template* to indicate how to construct a URI using strings and column references. All referenced columns in a triple map element must be column names that exists in the term maps.

A subject map may specify one or more *class IRI's* represented by the `rr:class` property. The value of the property must be a valid IRI. A class IRI generates triples of the form `s rdf:type i` for each row in the logical table, where *s* is the IRI generated by the subject map and *i* is the class IRI specified by the `rr:class` property.

Triples are generated by a triple map per row, i.e., each row in the logical table *entails* a set of triples. All the triples entailed by a row share the same subject. Then for each row we generate the following triples (all share the same subject S, as is defined by the subjectMap).

- For each `rr:class C` connected to the subject, we generate the triple `S rdf:type C`
- For each predicate-object map of the triple map, we add the triples `S P O`, where P is the predicate as specified by the predicate map map and O is the object as specified by the object map.

For ease of exposition and due to space constraints, we will not deal here with RDF types, nor with referencing object maps. However, it is possible to extend our technique to deal with these features.

4. SPARQL to SQL through Datalog

We now describe the core technique for SPARQL to SQL. The translation consists of two steps: *(i)* translation of the SPARQL query into Datalog rules, and *(ii)* generation of a relational algebra expression from the Datalog program. Once the relational algebra expression has been obtained, we generate an SQL query by using the standard translation of the relational operators into the corresponding SQL operators [1]. We now describe steps *(i)* and *(ii)*.

4.1. SPARQL to Datalog Encoding

The first step of the translation process is generating a Datalog program that has equivalent semantics to the original SPARQL query. For this translation we use a syntactic variation of the translation proposed by [27] and introduced in Section 3.4. The original translation was developed in [26, 27] with the intention of using Datalog engines for SPARQL execution. Our intention is different, we will use the rule representation of the SPARQL query as means to manipulate and optimize the query before generating an SQL query. We will discuss the key differences in our translation during the presentation and we will use the same notation for readers familiar with [27]. To keep the grounding finite, we only allow functional terms that has nesting depth of at most 2.

The original translation takes the SPARQL algebra tree of a SPARQL query and translates each operator into an equivalent set of Datalog rules. To get an intuition of the process we advice the reader to see Example 3 in Section 3. Here, we *syntactically* modify the techniques output to obtain a more compact result that *(i)* uses built in predicates available in SQL and avoid the use of negation in rules and the exponential growth present in the original technique, *(ii)* provides a formal ground to ensure correctness of the transformation and clearly understand when this approach deviates from the official semantics. *(iii)* can be optimized using slight extensions to standard notions of logic programming and database theory. It is *critical* to notice that we

do not change the semantics of the Datalog translation, we only produce a more compact representation of the *same* program.

In this section, we assume that RDF facts are stored in a 3-ary relation named *triple* (which can also be seen as a DB table with 3 columns, s p o). This assumption is the same as in the original translation in [26], and we will remove this restriction in the next section, when we introduce R2RML mappings. Observe the `null` cannot occur in an RDF triple in the graph, and hence there are also no `null` values in the *triple* relation.

We now provide the translation.

Definition 27 (Π_Q). *Let $Q = (V, P, G)$ be a SPARQL query. The logic program of Q, denoted Π_Q, is defined as:*

$$\Pi_Q = \{G\} \cup \tau(V, P)$$

where $\{G\} = \{\mathtt{triple}(s, p, o) \mid (s, p, o) \in G\}$ and τ, which is defined inductively in Figure 6, stands for the actual translation of the SPARQL algebra query to rules. Expressions are translated as follows:

- *$E = \mathtt{Bound}(v)$ is translated into $\mathtt{isNotNull}(v)$,*
- *$E = \neg\mathtt{Bound}(v)$ is translated into $\mathtt{isNull}(v)$,*
- *We will not consider in this paper the expressions for typing ($\mathtt{isBlank}(v)$, $\mathtt{isIRI}(v)$, and $\mathtt{isLiteral}(v)$). However, these expressions can be handled easily using our approach.*
- *Every occurrence of $\neg$ in E is replaced by* **NOT**. □

The translation presented in Definition 27 deviates from [27] in that it exploits SQL built-ins in the generated rules. In particular, the differences are:

- Rule (4): In [27], the authors translate this operator (c.f. rule (4) in Section 3.4) using a set of rules. We encode this set using a single rule and the distinguished predicate `LeftJoin`.
- Rule (5): In [26], the authors translate filter boolean expressions into a set of rules that is exponential in the number of $\vee$ operators (c.f., LT operator). We encode this set in a single rule by leaving the boolean expression untouched. In [27] instead, the authors translate filter boolean expressions by adding facts that simulate filter evaluation, for instance, $equals(x, x, true)$. Clearly this is undesirable in our case.
- Boolean expressions: We encode the expressions `Bound`, ¬`Bound`, and ¬ using the distinguished predicates `isNull`, `isNotNull`, and **NOT**.

We highlight that the semantics of of rules (4) and (5) is exactly that of the corresponding rules in [27]. Hence, our variation is equivalent to the one in [27] and the soundness and completeness results for the SPARQL-Datalog translation still hold (see Appendix).

It is worth noticing that, intuitively, the resulting Datalog program can be seen as a tree, where ans_1 is the root, and the *triple* atoms and boolean expressions are the leaves. Moreover, the trees representing the SPARQL algebra and the Datalog translation have very similar structures. The following examples illustrate the concepts presented above.

Example 5. *Let Q be a SPARQL query asking for* the name of all students, and the grades of each student by year if such information exists*, as follows:*

```
SELECT * WHERE { {?x :a :Student;  :hasName ?y}
   OPTIONAL {?x :hasEnrolment ?z .
      ?z :hasYear ?w; :hasGrade ?u } }
```

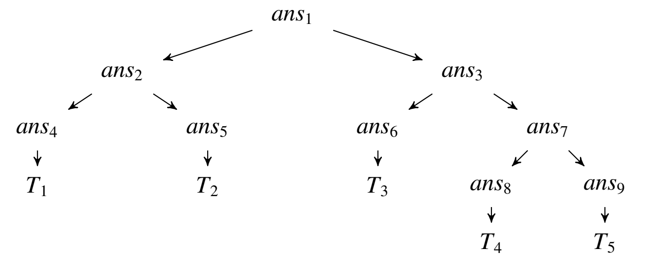
Using SPARQL algebra:

```
LeftJoin(
  JOIN(BGP(?x, :a, :Student),
       BGP(?x, :hasName, ?y)),
  JOIN(BGP(?x, :hasEnrolment, ?z) ,
       JOIN(BGP(?z, :hasYear, ?w),
            BGP(?z, :hasGrade, ?u))),
  TRUE )
```

The Datalog program, Π_Q, for this query is as follows (note, we use numeric subindexes instead of unique names due to space constrains):

$\mathtt{ans}_1(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{w},\mathtt{u})$	:-	$\mathtt{LeftJoin}(\mathtt{ans}_2(\mathtt{x},\mathtt{y}), \mathtt{ans}_3(\mathtt{x},\mathtt{z},\mathtt{w},\mathtt{u}), \mathtt{true})$
$\mathtt{ans}_2(\mathtt{x},\mathtt{y})$	:-	$\mathtt{ans}_4(\mathtt{x}), \mathtt{ans}_5(\mathtt{x},\mathtt{y})$
$\mathtt{ans}_3(\mathtt{x},\mathtt{z},\mathtt{w},\mathtt{u})$	:-	$\mathtt{ans}_6(\mathtt{x},\mathtt{z}), \mathtt{ans}_7(\mathtt{u},\mathtt{w},\mathtt{z}))$
$\mathtt{ans}_4(\mathtt{x})$	:-	$\mathtt{triple}(\mathtt{x}, '\mathtt{rdf:type}', \mathtt{Student})$
$\mathtt{ans}_5(\mathtt{x},\mathtt{y})$	:-	$\mathtt{triple}(\mathtt{x}, '\mathtt{hasName}', \mathtt{y})$
$\mathtt{ans}_6(\mathtt{x},\mathtt{z})$	:-	$\mathtt{triple}(\mathtt{x}, '\mathtt{hasEnrolment}', \mathtt{z})$
$\mathtt{ans}_7(\mathtt{z},\mathtt{u},\mathtt{w})$	:-	$\mathtt{ans}_8(\mathtt{w},\mathtt{z}), \mathtt{ans}_9(\mathtt{u},\mathtt{z})$
$\mathtt{ans}_8(\mathtt{w},\mathtt{z})$	:-	$\mathtt{triple}(\mathtt{z}, '\mathtt{hasYear}', \mathtt{w})$
$\mathtt{ans}_9(\mathtt{u},\mathtt{z})$	:-	$\mathtt{triple}(\mathtt{z}, '\mathtt{hasGrade}', \mathtt{u})$ □

As with the original translation, the dependency graph of the predicates in the Datalog program corresponds to the dependency graph of the SPARQL algebra operators, as can be seen in the following graph.



where each T_i represents $\mathtt{triple}(\mathtt{x}, '\mathtt{rdf:type}', \mathtt{Student})$, $\mathtt{triple}(\mathtt{x}, '\mathtt{hasName}', \mathtt{y})$, $\mathtt{triple}(\mathtt{x}, '\mathtt{hasEnrolment}', \mathtt{z})$, $\mathtt{triple}(\mathtt{z}, '\mathtt{hasYear}', \mathtt{w})$ *and* $\mathtt{triple}(\mathtt{z}, '\mathtt{hasGrade}', \mathtt{u})$, *respectively.* □

4.2. Datalog to SQL.

Next we show how to generate a relational algebra expression that corresponds to the Datalog program presented above. This is possible because the program we obtain is stratified and does not contain any recursion. From this relational algebra expression we generate the SQL query as usual. Since Datalog is position-based (uses variables) while relational algebra

$$
\begin{array}{lcll}
\tau(V, \mathtt{BGP(s,p,o)}) & = & \{\ \mathtt{ans_{BGP}}(\overline{\mathtt{V}}) \text{ :- } \mathtt{triple(s,p,o)}.\ \} & (1)\\
\tau(V, \mathtt{Join(P_1,P_2)}) & = & \tau(vars(P_1), P_1) \cup \tau(vars(P_2), P_2) \cup & \\
 & & \{\ \mathtt{ans_{Join(P_1,P_2)}}(\overline{\mathtt{V}}) \text{ :- } (\mathtt{ans_{P_1}}(\overline{\mathtt{vars(P_1)}}), \mathtt{ans_{P_2}}(\overline{\mathtt{vars(P_2)}})) & (2)\\
 & & \quad \bigwedge_{v \in vars(P_1) \cap vars(P_2)} \mathtt{isNotNull(v)}.\ \} & \\
\tau(V, \mathtt{Union(P_1,P_2)}) & = & \tau(vars(P_1), P_1) \cup \tau(vars(P_2), P_2) \cup & \\
 & & \{\ \mathtt{ans_{Union(P_1,P_2)}}(\overline{\mathtt{V[V \setminus vars(P_1) \to null]}}) \text{ :- } \mathtt{ans_{P_1}}(\overline{\mathtt{vars(P_1)}}).\ \} \cup & \\
 & & \{\ \mathtt{ans_{Union(P_1,P_2)}}(\overline{\mathtt{V[V \setminus vars(P_2) \to null]}}) \text{ :- } \mathtt{ans_{P_2}}(\overline{\mathtt{vars(P_2)}}).\ \} & (3)\\
\tau(V, \mathtt{LeftJoin(P_1,P_2,E)}) & = & \tau(vars(P_1), P_1) \cup \tau(vars(P_2), P_2) \cup & \\
 & & \{\ \mathtt{ans_{LeftJoin(P_1,P_2,E)}}(\overline{\mathtt{V}}) \text{ :- } (\mathtt{LeftJoin}((\mathtt{ans_{P_1}}(\overline{\mathtt{vars(P_1)}}), & \\
 & & \quad \mathtt{ans_{P_2}}(\overline{\mathtt{vars(P_2)}})), \mathtt{E})).\ \} & (4)\\
\tau(V, \mathtt{Filter(E,P)}) & = & \tau(vars(P), P) \cup & \\
 & & \{\ \mathtt{ans_{Filter(E,P)}}(\overline{\mathtt{V}}) \text{ :- } (\mathtt{ans_{P}}(\overline{\mathtt{vars(P_1)}}) \wedge \mathtt{E}) & (5)\\
 & & \quad \bigwedge_{v \in vars(E)} \mathtt{isNotNull(v)}.\ \} &
\end{array}
$$

Figure 6: Translation Π_Q from SPARQL algebra to rules.

SQL is usually name-based (use column names) we apply standard ([1]/Section 4.4) syntactic transformations to the program to go from one paradigm to the other. These transformations are not particularly interesting, but for the sake of completeness we describe them in Appendix C. After this transformation, each rule body consists of a single atom since joins are made explicit with a distinguished atom `Join` and every boolean expression is added to a `Filter` atom. Now, we are ready to provide the relational algebra translation for a program.

Given a Datalog program Π_Q, the operator $[\![\cdot]\!]_{\Pi_Q}$ takes an atom and returns a relational algebra expression. We drop the subindex Π_Q whenever it is clear from the context. We first define how to translate `ans` atoms. Then we define how to translate `triple` atoms and the distinguished `Join`, `Filter` and `LeftJoin` atoms. Intuitively, the *ans* atoms are the SQL projections, whereas Join, LeftJoin, and Filter are $\bowtie$, $⟕$ and σ respectively.

Definition 28 (Π_Q to SQL). *Let Q be a query,* ans *be defined predicate symbol in Π_Q, P_1, P_2 any predicates in Π_Q, $\vec{x}, \vec{z}$ vectors of terms, and E a filter expression. Then:*

$$[\![ans(\vec{x})]\!] = \Pi_{\vec{x}}([\![body^1_{ans}(\vec{z}_1)]\!] \cup \cdots \cup [\![body^n_{ans}(\vec{z}_n)]\!])$$

where $body^j_{ans}$ is the atom in the body of the j-th rule defining `ans`.

$$
\begin{array}{rcl}
[\![triple(\vec{z})]\!] & = & \Pi_{\vec{z}}(triple)\\
[\![Join(P_1(\vec{x_1}), P_2(\vec{x_2}), jn)]\!] & = & \bowtie_{jn}(([\![P_1(\vec{x_1})]\!],\\
 & & [\![P_2(\vec{x_2})]\!]))\\
[\![LeftJoin(P_1(\vec{x_1}), P_2(\vec{x_2}), ljn)]\!] & = & ⟕_{ljn}(([\![P_1(\vec{x_1})]\!],\\
 & & [\![P_2(\vec{x_2})]\!]))\\
[\![Filter(P_1(\vec{x_1}), E)]\!] & = & \sigma_E([\![P_1(\vec{x_1})]\!])
\end{array}
$$

□

Observe that in the previous definition, if there are *null* constants (or any other constant) in the *ans* atoms, they are translated as statements of the form "*null* AS *x*" in the projections. Recall that we are overloading the projection algebra operator to ease the presentation.

Example 6. *Let Π_Q be the Datalog program presented in Example 5. Then $[\![ans_1(\vec{x})]\!]$ is as follows:*

$$
\begin{array}{l}
\Pi_{as_1} ⟕_{ljn} (\\
\quad \Pi_{as_2} \bowtie_{jn_1} (\Pi_{as_4}\sigma_{fc_1}(triple), \Pi_{as_5}\sigma_{fc_2}(triple)),\\
\quad \Pi_{as_3} \bowtie_{jn_2} (\Pi_{as_6}\sigma_{fc_3}(triple),\\
\qquad\qquad \Pi_{as_7} \bowtie_{jn_3} (\Pi_{as_8}\sigma_{fc_4}(triple), \Pi_{as_9}\sigma_{fc_5}(triple))))
\end{array}
$$

Here we omit the definitions of the join, leftjoin, and filter conditions, and the projections to avoid distracting the reader from the core of the translation. The full definitions is given in Example 13 in Appendix C. □

SQL-compatibility: It is well known that there are some important differences between SQL and SPARQL with respect to the scope of results. This differences require restrictions on the SPARQL queries to guarantee a correct SQL translation. We now elaborate on this. We start with an illustrating example.

Example 7. *Consider the following SPARQL algebra expression:*

$$LeftJoin(A(x,z), R(x,y), z > 0)$$

Following the translation presented in [27], we would obtain a Datalog program of the following (simplified) form:

$$
\begin{array}{lll}
\mathtt{answer_1(x)} & \text{:-} & \mathtt{A(x,z), R(x,y), z > 0}\\
\mathtt{answer_1(x)} & \text{:-} & \mathtt{A(x,z)}, \textbf{not}\ \mathtt{answer_2(x,y,z)}\\
\mathtt{answer_2(x,y,z)} & \text{:-} & \mathtt{R(x,y), z > 0}
\end{array}
$$

Although the semantics of this program is correct as far as Datalog is concerned, the fact that the variable z does not occur in any non-boolean atom in the body of $answer_2$ is a problem when one tries to translate that into SQL. □

As the original SPARQL-Datalog translations, we require *well-designed* queries [27]. This constraint imposes a restriction over the variables occurring in Optional and Unions operators. However, in order to produce a sound translation to SQL we need to require a further restriction—not present in [27]—as shown by the following example:

Example 8. *Consider the following SPARQL query:*

```
SELECT * WHERE {
  {?x1 :A ?x3} OPTIONAL {
     {?x1 :a :B} OPTIONAL {
       {?x1 :a :C} FILTER ?x3 < 1 } } }
```

Next, we show the relevant fragment of Π_Q*:*

$$
\begin{array}{lll}
\mathtt{ans_3(x_1)} & :- & \mathtt{LeftJoin(ans_4(x_1), ans_5(x_1), x_3 < 1)} \\
\mathtt{ans_4(x_1)} & :- & \mathtt{triple(x_1, 'rdf{:}type', ':B')} \\
\mathtt{ans_5(x_1)} & :- & \mathtt{triple(x_1, 'rdf{:}type', ':C')}
\end{array}
$$

Observe that using Datalog semantics, the boolean expression $x_3 < 1$ *works almost as an assignment, since as the program is grounded,* x_3 *will be replaced for all the values in the Herbrand Base smaller than 1. However, it does not work in the same way in relational algebra.*

$$
\begin{array}{l}
\Pi_{as_1} ⟕_{ljn_1} (\\
\quad \Pi_{as_2}(\sigma_{fc_1} triple), \\
\quad ⟕_{ljn, T_1.o<1}(\Pi_{as_6}(\sigma_{fc_2} triple), \\
\qquad\qquad \Pi_{as_7}(\sigma_{fc_3} triple)))
\end{array}
$$

Here we omit the definitions of as_i*,* fc_i *and* ljn_i *since they are not relevant for this example. Clearly the relational algebra expression shown above is incorrect since* $T_1.s$ *is not defined inside the scope of the second left join. This issue arises from the boolean operation in Datalog rules involving variables that do not occur in atoms with non built-in predicate symbols.* □

To avoid this issue we require SPARQL queries to be *SQL-compatible*, which is defined as follows.

Definition 29 (SQL-Compatible). *Let Q be a query. We say that Q is* **SQL-compatible** *if Q is well-designed, and in addition,* Π_Q *does not contain any rule r where there is a variable v in a boolean expression in the body of r such that there is neither a defined atom, nor an extensional atom in the body of r where v occurs.*

Observe that the previous constraint does not have a major impact in real-life queries. Queries in well-known benchmarks such as BSBM [6], LUBM [16], FishMark [4], NPD [7], etc. hold in this category.

Theorem 1. *Let Q be an SQL-compatible SPARQL query,* Π_Q *the Datalog encoding of Q, and* $[\![ans_Q(\vec{x})]\!]$ *the relational algebra statement of* Π_Q*. Then it holds:*

$$\vec{t} \in [\![ans_Q(\vec{x})]\!] \leftrightarrow \Pi_Q \models ans_Q(\vec{t})$$

PROOF. See Appendix B. □

5. Integrating R2RML mappings

In this section we present a translation of R2RML [10] mappings into Datalog rules. This translation allows to generalize our SPARQL-to-SQL technique to be able to deal with any relational schema. The necessary background needed to understand this section can be found in Section 3.6.

In the previous section we used the ternary predicate *triple* as an extensional predicate, that is, a DB relation in which all the data is stored (in that particular case, RDF terms). Now when we introduce R2RML mappings, *triple* becomes a defined predicate. The rules that define the *triple* predicate will be generated from the R2RML mapping (adding one more strata to the Datalog program). The Datalog rules coming from the mappings will also encode the operations needed to generate URI's and RDF Literals from the relational data.

The objective of our translation R2RML-to-Datalog is to generate a set of rules that reflect the semantics of every triple map in the R2RML mapping. Intuitively, for each triple map, we generate a set of rules whose bodies refer to the "effective SQL query" of the triple map, and whose heads entail the triples required by the R2RML semantics (see Section 3.6).

Before providing the formal definition we present an example illustrating the idea.

Example 9. *Consider the R2RML mapping from Example 4 minus the mappings related to courses, the following Datalog rules would capture their semantics:*

```
triple(cc(':stud/', id),' rdf:type',' :Student'):-
      stud(id, name, course), NotNull(id)

triple(cc(':stud/', id),' :name', name) : -
      stud(id, name, course), NotNull(id), NotNull(name)
```

where `cc` *stands for a built-in function that is interpreted as the* string concatenation operator. □

First, the following definitions assume that the R2RML mapping has been normalized so that:

- All shortcuts for constants are expanded,
- All SQL shortcuts have been expanded,
- All rr:class definitions are replaced by predicate object maps,
- All predicate-object maps with multiple predicate or object definitions are expanded into predicate-object maps with a single predicate and a single object definition,
- All referencing predicate-object maps have been replaced by a new triple map equivalent to the predicate-object map,
- All triple maps with multiple predicate-object maps have been replaced by a set of equivalent triple maps.

These transformations are described in Appendix Appendix A.

Next, we define a function that allows to obtain Datalog terms and atoms, from R2RML nodes. First, given a *term map node t* (those nodes that indicate how to construct RDF terms), we use $tr(t)$ to denote i) a constant c if t is an constant node with value c, ii) a variable v if t is a column reference with name v, iii) $cc(\vec{x})$ if t is a *URI template* where cc is a built-in predicate interpreted as the string concatenation function and $\vec{x}$ are the components of the template to be concatenated (constant strings and column references denoted by variables) .

Definition 30 (Triple map node translation). *Let m be a triple map node, let* V_t *the set of variables occurring in the term map nodes in m, let* `prop` *the property map node of the mapping m, and* `obj` *the object map node of prop. Then the* mapping rule *for m, denoted* $\rho(m)$*, is*

$$\mathtt{triple}(tr(\mathtt{subject}_m), tr(\mathtt{prop}), tr(\mathtt{obj})) \text{ :-}$$
$$\qquad \mathtt{translated_logical_table}, \mathtt{NN}$$

and where NN *is a conjunctions of atoms of the form* $NotNull(x_i)$ *for each variable* x_i *appearing in the head of the corresponding rule, and translated_logical_table is i)* $A(x_1, \ldots, x_n)$ *if the logic table is a base table or view with name A and arity n, or ii)* $A_1(\vec{x_1}), \ldots, A_n(\vec{x_n}), B(\vec{y})$*, that is, a conjunction of table or view atoms, and boolean condition* $B(\vec{y})$ *with* $\vec{y} \in \bigcup_1^n \vec{x_i}$ *if the logic table is an SQL query whose semantics can be captured by the body of a data log rule, iii) otherwise* $Aux(x_1, \ldots, x_n)$*, if the logical table is an SQL query of arity n whose semantics cannot be captured in Datalog.* □

Note that the previous definition avoids the generation of RDF triples in which null values are involved (as required by the R2RML standard). Also, it is important to highlight that processing of *translated_local_table* in the case of arbitrary SQL queries requires a system to perform SQL parsing of the SQL queries in the R2RML mapping. Providing full details on how to do this goes beyond the scope of this paper. However, recall that the semantics of a large fragment of SQL can be captured by Datalog rules. When the query cannot be translated into Datalog, the translation uses auxiliary predicates *Aux*, that captures the semantics of the logical table. By keeping a map between these auxiliary predicates and the corresponding SQL query, the SQL generator can then use SQL in-line subqueries to generate an appropriate translation.

Last, we also note that a system implementing this technique should aim at implementing an SQL parser that is as complete as possible, as to be able to avoid the generation of SQL queries with subqueries. Since these, as we will discuss in the next section, are detrimental to performance.

Now we continue by extending the definitions so that we can integrate the SPARQL Datalog translation with the translation of the R2RML mappings.

Definition 31 (R2RML mapping program). *Let M be an R2RML mapping. Then the* mapping program *for M, denoted* $\rho(M)$ *is the set of rules*

$$\Pi_M = \{\, \rho(m) \mid \textit{for each } \text{triple map node } m \in M\}$$

□

Now, we can introduce *SQL query programs*, that is, the program that allows us to obtain a full SQL rewriting for a given SPARQL query through the R2RML mappings.

Definition 32 (SQL query program). *Given a SPARQL query Q, and an R2RML mapping M, an* SQL query program *is defined as* $\Pi_Q^M = \Pi_Q \cup \Pi_M$.

In order to show the preservation of soundness and completeness as stated by Proposition 2 it is suffices to prove the following:

Lemma 1. *Let DB be a database, M be a R2RML mapping for DB and G be the RDF graph entailed by M and a DB. Then*

$$(s, p, o) \in G \textit{ iff } \Pi_m \models \mathtt{triple(s,p,o)}{:}-$$

PROOF. See Appendix. □

Last, we extend the translation from Datalog to relational algebra to be able deal with the rules introduced by the mappings..

Definition 33 (Π_Q to SQL). *Let Q be a query, M an R2RML mapping, and let* triple *be defined in* Π_Q^M*. Then* $[\![triple(s,p,o)]\!]$ *is defined next:*

$$[\![triple(s,p,o)]\!] = \begin{cases} [\![body^1_{triple}(\vec{z}_1)]\!] \cup \ldots \cup [\![body^n_{triple}(\vec{z}_n)]\!] & \textit{If triple has } n \geq 1 \textit{ rules defining it in } \Pi_Q \\ [\![NULL_table]\!] & \textit{If triple has 0 definitions} \end{cases}$$

where $body^j_{triple}$ *is the body of the j-th rule defining it; and* $NULL_table$ *is a DB table that contains the same number of columns as the number of variables in triple, and 1 single row with null values in every column.* □

Theorem 2. *Let Q be an SQL-compatible SPARQL query, M an R2RML mapping,* Π_Q *the Datalog encoding of Q and M, and* $[\![\Pi_Q]\!]$ *the relational algebra statement of* Π_Q*. Then it holds:*

$$\vec{t} \in [\![ans_1(\vec{x})]\!] \leftrightarrow \Pi_Q \models ans_1(\vec{t})$$

PROOF. The proof follows immediately from Theorem 1 and Lemma 1. □

Some features of R2RML that are not discussed here due to space constraints, but can easily be added to the presented technique are: RDF typing, data errors, default mappings, percent encoding. In particular, an important feature of R2RML mappings which is not addressed here is *inverse mappings*. In the -ontop- (the system that implements our technique) we have implemented a form of *default inverse mapping* that allows to transform URI's that appear in SPARQL queries into the corresponding functional terms (e.g., $cc(\vec{x})$). An in-depth extension of our technique to cover inverse R2RML mappings will be done on a follow up paper.

6. On SQL Performance

We have presented the core of our SPARQL to SQL translation, however, as is, the technique produces SQL queries which are suboptimal with respect to performance.

Example 10. *Consider the following SPARQL query asking for the URI and name of all the students.*

```
SELECT * WHERE { ?x a :Student; :name ?y }
```

together with the following R2RML mappings (in Datalog syntax)

```
triple(cc("stud1/id", id), "rdf:type", ":Student") :-
                                        stud1(id, name)
triple(cc("stud1/id", id), ":name", name) :-
                                        stud1(id, name)
triple(cc("stud2/id", id), "rdf:type", ":Student") :-
                                        stud2(id, name)
triple(cc("stud2/id", id), ":name", name) :-
                                        stud2(id, name)
```

a direct translation of the SPARQL query to SQL would render a query similar to query (a) in Figure 7. This is in fact the kind of queries that our technique, as far as it has been presented up to now, will generate.

We have observed that queries structured in this way perform poorly in most databases (see experiments *in this section). However, it is very often possible to obtain equivalent queries that perform much better, e.g., query (d) in our example.*

To arrive to query (d), -ontop- performs a procedure called *partial evaluation* (c.f. Section 3.1), extended with *semantic query optimization*.

The process transforms the query, roughly, through the steps shown in Figure 7. However, this transformations are done to the Datalog program, which is easier to manipulate than the algebra expression or the SQL query. We will describe this process formally in Section 7. Next we will isolate the features of these queries that trigger this bad performance, propose optimized alternatives, and evaluate their impact on the query execution time.

6.1. Structural analysis of direct SPARQL-to-SQL translations

Consider query (a) in Figure 7. There are 4 features of this query that we highlight as triggers for bad performance:

- *JOINs over URI templates*,
- *JOINs of UNION-subqueries*,
- *unsatisfiable conditions*, and
- *redundant JOINs* with respect to keys.

We will now elaborate on each point, describing why these are present and what are the alternatives.

JOINs over URI templates. URI templates in R2RML are usually translated as *functional terms*. We refer as *functional terms*—in the context of SQL— as values that result from a computation, e.g., string manipulation, arithmetic operation, etc. Functional terms in the context of SPARQL to SQL appear whenever a query involves URI templates, and they may appear in SELECT clauses to construct URIs, or as terms in boolean conditions of JOIN or WHERE clauses. For example,

```
SELECT name as Y FROM student
WHERE ':stud/'||ID = ":stud/22"
```

or

```
SELECT ':stud/'||v1.ID as X, name as Y
FROM student v1 JOIN student v2
ON 'stud/'||v1.ID = ':stud/'||v2.ID
```

Our technique as is produce this kind of terms.

Example 11. *Consider the from Example 10 and let cc be a function to compute string concatenation and suppose we have the following R2RML mappings that includes a URI template in the subject position (in Datalog syntax) .*

$\texttt{triple}(\texttt{cc}(\text{“stud/id”}, \texttt{id}), \text{“rdf:type”}, \text{“:Student”}) \text{ :- } \texttt{stud}(\texttt{id}, \texttt{name})$
$\texttt{triple}(\texttt{cc}(\text{“stud/id”}, \texttt{id}), \text{“:name”}, \texttt{name}) \text{ :- } \texttt{stud}(\texttt{id}, \texttt{name})$

our translation of this SPARQL query to SQL with these mappings would produce a query similar to the second SQL query presented in this section. In particular, this SQL query is produced form the following relational expression:

$$
\begin{array}{l}
\Pi_{T_1.s \text{ as } X, T_2.o \text{ as } Y} (\\
\quad \bowtie_{T_1.s = T_2.s} (\\
\quad\quad \Pi_{\text{“stud/id”} \,||\, ID \text{ AS } T_1.s,\ \text{“rdf:type” AS } T_1.p,\ \text{“:Student” AS } T_1.o} (stud), \\
\quad\quad \Pi_{\text{“stud/id”} \,||\, ID \text{ AS } T_2.s,\ \text{“:name” AS } T_2.p,\ \text{“:name” AS } T_2.o} (stud) \quad) \quad)
\end{array}
$$

□

Evidence that queries with this feature perform poorly has been reported for all major databases engines [30, 32] and in our new experiments (see next section). The reason for the bad performance is that query planners are not able to use indexes to evaluate the conditions.

At the same time, in the context of SPARQL-to-SQL systems, it is often possible to transform a query with expressions over functional terms to a semantically equivalent query where the function does not appear. For example, the previous query can be rewritten as:

```
SELECT ':stud/'||v1.ID as X, name as Y
FROM stud v1 JOIN stud v2
ON v1.ID = v2.ID
```

This alternative queries usually perform significantly better, as shown in the next section. However, performing these transformations is not trivial, e.g., to arrive to query (c) in Figure 7 first we need to transform a JOIN of UNIONs into a UNION of JOINs. In some cases it is impossible to perform such transformation (see our comment on OPTIONAL/LEFT JOIN with UNIONs on their right argument).

It is critical to notice that often such optimization is only correct in the context of R2RML mappings, for example, the ON expression in the following SQL query:

```
SELECT *
FROM courseGrade v1 JOIN courseGrade v2
ON
'http://course'||v1.studid||'/'||v1.courseid =
'http://course'||v2.studid||'/'||v2.courseid
```

can be simplified to the expression

```
ON v1.studid   = v2.studid AND
   v1.courseid = v2.courseid
```

only if we know that the slash character '/' does not appear in the columns studid and courseid. However, the R2RML standard does have a restriction on the form of the URI templates that guarantee this (see *safe separator* in Section 7.3 in `http://www.w3.org/TR/r2rml/`).

Recall that we are overloading the projection algebra operator to ease the presentation.

```
 SELECT v1.s as X, v2.o as Y FROM (
   SELECT 'stud1/id'||id as s, 'rdf:type' as p, ':Student' as o FROM stud1 UNION ALL
   SELECT 'stud1/id'||id as s, ':name'    as p, name       as o FROM stud1 UNION ALL
   SELECT 'stud2/id'||id as s, 'rdf:type' as p, ':Student' as o FROM stud2 UNION ALL
   SELECT 'stud2/id'||id as s, ':name'    as p, name       as o FROM stud2 ) v1
JOIN (
   SELECT 'stud1/id'||id as s, 'rdf:type' as p, ':Student' as o FROM stud1 UNION ALL
   SELECT 'stud1/id'||id as s, ':name'    as p, name       as o FROM stud1 UNION ALL
   SELECT 'stud2/id'||id as s, 'rdf:type' as p, ':Student' as o FROM stud2 UNION ALL
   SELECT 'stud2/id'||id as s, ':name'    as p, name       as o FROM stud2 ) v2
 ON v1.p='rdf:type' AND v1.o=':Student' AND v2.p=':name' AND v1.s=v2.s
```

(a)

```
SELECT v1.s as X, v2.o as Y FROM (
  SELECT 'stud1/id'||id as s, 'rdf:type' as p, ':Student' as o FROM stud1 UNION ALL
  SELECT 'stud2/id'||id as s, 'rdf:type' as p, ':Student' as o FROM stud2 ) v1
JOIN (
  SELECT 'stud1/id'||id as s, ':name'    as p, name       as o FROM stud1 UNION ALL
  SELECT 'stud2/id'||id as s, ':name'    as p, name       as o FROM stud2 ) v2
ON v1.s=v2.s
```

(b)

```
SELECT 'stud1/id'||v1.id as X, v2.name as Y FROM stud1 v1 JOIN stud1 v2 ON v1.id=v2.id
                                            UNION ALL
SELECT 'stud2/id'||v1.id as X, v2.name as Y FROM stud2 v1 JOIN stud2 v2 ON v1.id=v2.id
```

(c)

```
SELECT 'stud1/id'||v1.id as X, v2.name as Y FROM stud1 v1
                          UNION ALL
SELECT 'stud2/id'||v1.id as X, v2.name as Y FROM stud2 v1
```

(d)

Figure 7: SQL for the SPARQL query and mappings in Example 10. Variations are: (a) SQL queries with UNION-subqueries (b) simplified UNION-subqueries (c) equivalent query without subqueries (d) optimal query.

UNION-subqueries. We refer as *UNION-subqueries*—in the context of SQL—to in-line subqueries with UNION or UNION ALL operators. These may appear in any location where a table reference is allowed, e.g., in FROM clauses, JOIN operators, IN, etc. As with any table reference, columns from the subquery can be referenced to impose conditions in WHERE clauses or in ON conditions of JOIN operations.

UNION-subqueries in the context of SPARQL-to-SQL techniques appear in SQL queries due to several reasons.

The first possibility is because the original SPARQL query may have UNION operators, which is usually translated as an SQL UNION of the translations of the graph patters involved in the SPARQL UNION. This is the case with our basic technique.

The second possibility is the SPARQL-to-SQL technique itself, which often translate BGPs in the SPARQL query into a UNION of SQL queries. Recall that, intuitively, each BGP gets replaced by a subquery that is the union of *all* the SQL queries derived from the R2RML mappings. See for instance query (a) in Example 10. Although the technique takes the union of all SQL queries, in practice one takes only the SQL queries relevant for the given BGP. Finding out the relevant mappings is in general a simple task, just by looking at the constants in the SPARQL query and the mappings. For example, query (a) in Example 10 can be simplified to query (b) by using only mappings *for rdf:type :Student* and mappings *for :name* to construct the subquery for the first and second BGP of the SPARQL query (respectively). This kind of simplification is done by most OBDA systems.

However, in the case where there exist multiple mappings for a given predicate/class, this simplification can not avoid the UNION. This is a common scenario that appears constantly, particularly in scenarios for data integration where multiple tables/queries provide relevant data for any given property or class . An example of this, again, is Example 10 where we have two mappings for the class *Student* and two mappings for the property *name*. In this cases, it is not possible to simplify the query beyond query (b).

This kind of query performs worse than the simplified alternative in the form of a UNION of JOIN queries (query (c)), mostly because in the second form it is possible to simplify the JOIN expressions (as we will show in our experiments).

This transformation is in general non-trivial, especially in the presence of *functional terms*. In particular, UNIONs can *never* be eliminated from the right side of a OPTIONAL operator in SPARQL since the OPTIONAL operator is non distributable with respect to UNION.

Unsatisfiable conditions. We distinguish two types of unsatisfiable conditions. The first type, already mentioned in the previous section, is related to URI constants in Predicate and Object (Class) maps in R2RML mappings. These URIs interact with URI constants in Predicate and Object (Class) positions in BGPs of SPARQL queries.

The second type is related to functional terms, in particular, those used for URI templates.

The first type yields unsatisfiable conditions as those in query (a) in Figure 7 in which there is a condition that requires that `v1.p = 'rdf:type'`, `v1.o= ':Student'` and `v2.p=':name'` holds, but the nested UNIONs include subqueries which do not satisfy those conditions.

When detected, it allows to go from query (a) to query (b) in the same example. As mentioned before, this situation is easy to avoid and almost all OBDA system for SPARQL and OWL support it (e.g., -ontop-, Ultrawrap, Mastro, Quonto, Requiem).

Here, we focus on detecting the second kind of unsatisfiable conditions. Such conditions involve URI templates that participate in JOINs. For example:

```
'http://example/data1/' || v1.u1 || '/' || v1.u2 ==
'http://example/data2/' || v2.u1 || '/' || v2.u2
```

or, the more complex

```
'http://example/' || v1.u1 || '/1/' || v1.u2 ==
'http://example/' || v2.u1 || '/2/' || v2.u2
```

This kind of conditions arise in data integration scenarios in which classes and properties are mapped to multiple tables and hence, URI templates are often different to reflect the fact that the tables contain object of a different nature.

To detect such unsatisfiable expressions, one needs to go from UNION of JOINs to JOIN of UNIONs as in query (c) in the example.

Redundant JOINs w.r.t. KEYs. This situation arises when a SPARQL query is translated into a union of JOIN queries in which some of these JOINs are redundant w.r.t. primary or foreign keys. The situation is relevant only after transforming JOIN of UNIONs into UNIONs of JOINs (since keys are defined only over tables), as in query (c) from Figure 7, which can be simplified into query (d) in the presence of a primary key over the column ID.

This redundancy arises often because the RDF data model (over which SPARQL operates) is a ternary model (s p o) while the relational model is n-ary, hence, the SPARQL equivalent of `SELECT * FROM t` on an n-ary table t requires exactly n triple patterns. When translating each of these triple patterns, a SPARQL-to-SQL technique will generate an SQL query with exactly n-1 self-JOIN operations.

It is well known that keeping those redundant JOINs is detrimental for performance and a lot of research has been devoted to optimizing SQL queries in these cases. The most prominent area that investigates this subject is Semantic Query Optimization (SQO), from which we borrow techniques to optimize SPARQL translations.

6.2. Experiments

In this section we present a series of experiments that demonstrate that: *(i)* the SQL features discussed in Section 6 are detrimental to performance; *(ii)* the alternatives discussed in the same section—and obtiained by -ontop- through partial evaluation— perfom better.

As a side note beyond the scope of this paper, this is a recurrent issue when query rewriting is used to support RDFS or OWL entailment regimes in SPARQL 1.1

We setup an environment based on the Wisconsin Benchmark [11]. This benchmark was designed so that performance of a database could be tested in a controlled way. To do this, the designers created a schema and data generator that has clear semantics and whose data distribution is well understood. This allows to produce queries that isolate the features that need to be tested.

The benchmark defines a single table definition (which can be used to instantiate multiple tables). The table, which we now call simple T, contains 16 attributes as shown in Figure 8.

```
CREATE TABLE t
(
    unique1 integer NOT NULL,
    unique2 integer NOT NULL PRIMARY KEY,
    two integer NOT NULL,
    four integer NOT NULL,
    ten integer NOT NULL,
    twenty integer NOT NULL,
    onePercent integer NOT NULL,
    tenPercent integer NOT NULL,
    twentyPercent integer NOT NULL,
    fiftyPercent integer NOT NULL,
    unique3 integer NOT NULL,
    evenOnePercent integer NOT NULL,
    oddOnePercent integer NOT NULL,
    stringu1 char(52) NOT NULL,
    stringu2 char(52) NOT NULL,
    string4 char(52) NOT NULL
)
```

Figure 8: Wisconsin Benchmark schema description

The data for each attribute is generated with a distribution that allows to easily compute the degree of selectivity of SQL operations (in JOINs with conditions, WHERE clauses, cross products, unions). In particular, given a target number of rows N, the data for each attribute is generated as follows:

- `unique1`. Integers from 0 to N (sequential).
- `unique2`. Integers from 0 to N (randomly ordered)
- `stringu1`. A 6 character alphabetic string generated from unique1.
- `stringu2`. A 6 character alphabetic string generated from unique2.

We refer the reader to [11] for details on the algorithm that generates `stringu1` and `stringu2`. Using these definitions we created databases with 5 tables and 100,000 rows per table. In addition we defined unique indexes on each of the four attributes above.

In the following sub-sections we describe the queries used in the different experiments. These queries do not belong to the Wisconsin benchmark, and were crafted to evaluated different forms of SQL that are *relevant in the context of SPARQL-to-SQL translations*. All the queries and their respective execution times are available online

All experiments were conducted on a HP Proliant server with 24 Intel Xeon CPUs (144 cores @3.47GHz), 106GB of RAM and a 1TB 15K RPM HD. The OS is Ubuntu 12.04 64-bit edition. Note that all queries where ran sequentially, and hence only one core was used at a time. This was a conscious choice meant to isolate performance of the SQL queries at hand from the capacities of the underlying OS and DB engine w.r.t. to parallelism.

The database engines used where MySQL 5.6, Postgres 9.1 and DB2 Express-C 10.5, chosen as representatives of two classes of engines, open source and industrial. All the query execution times presented here refer to *cold* execution (as in, a freshly started DB engine and flushed caches).

Combined case. In this experiment we show that indeed, SPARQL translations that combine nested-subqueries, JOINs over URI templates and unsatisfiable conditions are significantly slower than the equivalent optimized queries. That is, queries expressed as UNION of JOINs, where JOIN conditions are over table columns and which avoid unsatisfiable conditions. For instance:

```
SELECT v1.subj FROM (
 SELECT 'http://example/data1'||'/'||unique1 as s,
         unique1 FROM t1
 UNION ALL
 SELECT 'http://example/data2'||'/'||unique1 as s,
         unique1 FROM t2
) v1
JOIN (
 SELECT 'http://example/data1'||'/'||unique1 as s,
         unique1 FROM t1
 UNION ALL
 SELECT 'http://example/data2'||'/'||unique1 as s,
         unique1 FROM t2
) v2
ON v1.s = v2.s
WHERE v2.unique1 = 666
```

against queries of the form

```
SELECT 'http://example/data1/'||v1.unique1 as s
 FROM t1 v1 JOIN t1 v2 ON
  v1.unique1 = v2.unique1 WHERE v2.unique1 = 666
UNION ALL
SELECT 'http://example/data2/'||v1.unique1 as s
 FROM t2 v1 JOIN t2 v2 ON
  v1.unique1 = v2.unique1 WHERE v2.unique1 = 666 ;
```

-ontop- generates these optimised queries.

In each experiment we evaluated 2 sets of 4 series of queries. In one set we evaluated JOINs of UNION queries (JU) where the JOIN conditions involve URI templates. In the second set we evaluated UNION of JOINs (UJ) where JOIN conditions are expressed over plain columns. In each set, each of the 4 series stands for a form of JOIN condition. We considered the following 4 cases:

- **u1** JOIN on `unique1`
- **u1,u2** JOIN on `unique1` and `unique2`
- **st1** JOIN on `stringu1`
- **st1,st2** JOIN on `stringu1` and `stringu2`

In the cases of JU queries, the JOIN conditions are always of the form v1.s = v2.s. However, the inner subqueries construct values for s as URIs using string concat operations as it would be done for a URI template, i.e., with a URI prefix, and series of of values separated by the slash character '/'. The values

https://github.com/marianormuro/onto-wisbench

being concatenated in each series depend on the kind of JOIN we want to evaluate. For example, if we are evaluating (st1,st2), the concatenation expression is of the form:

```
'http://example/' || v1.stringu1 ||
                          '/' || v2.stringu2
```

To introduce unsatisfiable conditions we used two different prefixes for the URI templates in the inner subqueries, that is

```
http://example.org/data1/
```

and

```
http://example.org/data2/
```

To verify the behavior of the queries w.r.t. to the volume of data returned by the query, each series contains 7 queries with decreasing degree of selectivity. This was accomplished through constraints over the column `unique1` in the WHERE clause of the query. In particular we used the following (decreasingly selective) constraints:

1. `v1.unique1 = 666`
2. `v1.unique1 > 5000 AND v1.unique1 < 6000`
3. `v1.unique1 BETWEEN 5000 AND 6000`
4. `v1.unique1 > 20000 AND v1.unique1 < 30000`
5. `v1.unique1 BETWEEN 20000 AND 30000`
6. `v1.unique1 > 10000 AND v1.unique1 < 30000`
7. `v1.unique1 BETWEEN 10000 AND 30000`

This experiment evaluates a total of 56 queries.

The results of this experiment are shown in Figure 9. In all cases, our *optimized* queries (dashed lines) outperform the corresponding non-optimized queries, often by several orders of magnitude. While selectivity of the query does affect the performance of each query, it does not seem to affect the difference in performance between optimized and not optimized queries. Also, in some cases, the type of column being JOINed appears to affect the performance of the queries. This is particularly visible in the non-optimized queries in MySQL, where queries that JOIN URIs constructed with integer values (square and triangle marks) are considerable slower than queries that concatenate only strings (circle and crossed circle marks). Last, an interesting side observation is that indeed, in all databases we see different performance for the queries in which BETWEEN is used to express ranges (3,5,7) instead of inequalities (2,4,6). In most cases BETWEEN seems to offer better performance.

Having confirmed the general claim, we now present the experiments that allow us to understand this performance differences. In particular, we will evaluate the effect of each of the features we discussed in the previous section in isolation.

JOIN over URI templates. In this experiment we compare the performance of JOIN queries in which the ON condition is expressed over plain attributes against that of JOIN queries in which the condition is expressed over a string concatenation operation as those generated by URI templates. That is, queries of the form:

```
SELECT * FROM t1 v1 JOIN t2 v2
ON t1.stringu1 = t2.stringu1
```

against queries of the form

```
SELECT * FROM t1 JOIN t2
ON 'http://example.org/' || t1.stringu1 =
   'http://example.org/' || t2.stringu1
```

As before, we evaluated 8 series of 7 queries each. Four of these series are queries that join on original columns, while half of them join on CONCAT operations for URI templates. In particular. The 8 join conditions we consider are:

u1
```
v1.unique1 = v2.unique1
```
u1, u2
```
v1.unique1 = v2.unique1 AND v1.unique2 =
v2.unique2
```
st1
```
v1.stringu1 = v2.stringu1
```
st1, st2
```
v1.stringu1 = v2.stringu1 AND v1.stringu2 =
v2.stringu2
```
|| u1
```
'http://example.org/' || v1.unique1 =
'http://example.org/' || v2.unique1
```
|| u1, u2
```
'http://example.org/' || v1.unique1 || '/' ||
v1.unique2 = 'http://example.org/' || v2.unique1
|| '/' || v2.unique2
```
|| st1
```
'http://example.org/' || v1.stringu1 =
'http://example.org/' || v2.stringu1
```
|| st1, st2
```
'http://example.org/' || v1.stringu1 || '/'
|| v2.stringu2 = 'http://example.org/' ||
v2.stringu1 || '/' || v2.stringu2
```

As before, the seven queries decreasing degrees of selectivity and we use inequalities or BETWEEN to describe intervals (see previous section). We evaluated a total of 56 queries.

The results of this experiment is summarized in Figure 10. In general, the results confirm our expectation, i.e., JOINs over plain columns outperforms the equivalent query with concat. The results also show that it does not make a difference if the join involves one or two columns, or integers vs strings columns. The only exception being DB2 in which JOINs over integer columns involved concat operators perform poorly w.r.t. to the rest, and where joins with or without concats for the rest of the series appear to behave similarly. Note that this good performance in concat operations involving only strings is anomalous, We will investigate this situation further in a follow-up paper.

UNION-subqueries. In this experiment we compare the performance of queries in the form of JOIN of UNIONs (JU) against the performance of queries that are a UNION of JOINs (UJ). In this case we will not mix join conditions (i.e., plain columns vs. concatenation of strings) nor we will involve unsatisfiable conditions. That is, we will compare queries of the form

```
SELECT * FROM
   (SELECT * FROM t1 UNION ALL SELECT * FROM t2) v1
JOIN
   (SELECT * FROM t1 UNION ALL SELECT * FROM t2) v2
ON  v1.unique1 = v2.unique1
WHERE v2.unique1 > 5000 AND v2.unique1 < 6000
```

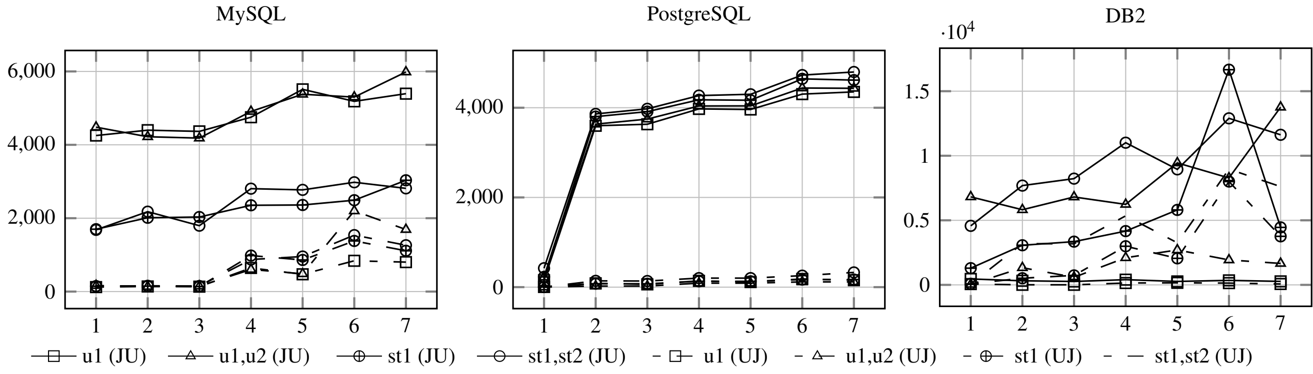


Figure 9: Full summary, non-optimal vs. optimal queries. Execution time in milliseconds

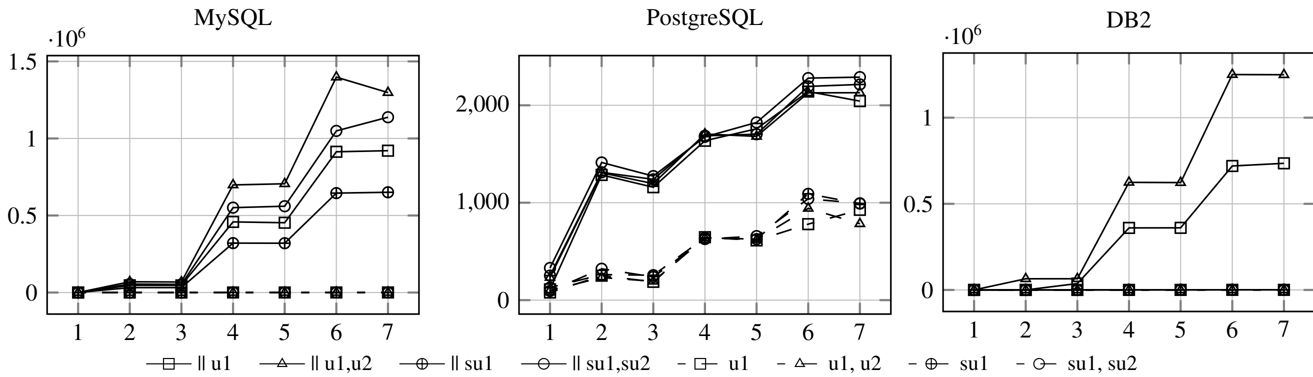


Figure 10: Performance of JOIN on *concat* expressions vs simple columns.

against queries of the form

```
SELECT * FROM t1 v1 JOIN t1 v2 ON
      v1.unique1 = v2.unique1
      WHERE v2.unique1 > 5000 AND v2.unique1 < 6000
UNION ALL
SELECT * FROM t1 v1 JOIN t2 v2 ON
      v1.unique1 = v2.unique1
      WHERE v2.unique1 > 5000 AND v2.unique1 < 6000
UNION ALL
SELECT * FROM t2 v1 JOIN t1 v2 ON
      v1.unique1 = v2.unique1
      WHERE v2.unique1 > 5000 AND v2.unique1 < 6000
UNION ALL
SELECT * FROM t2 v1 JOIN t2 v2 ON
      v1.unique1 = v2.unique1
      WHERE v2.unique1 > 5000 AND v2.unique1 < 6000
```

As before, we will have 2 sets of series, one for JUs and one for UJs. In each of these sets we will have 4 types of joins (u1), (u1,u2), (str1) and (str1,str2). The selectivity of the queries will also be decreased from query 1 to query 7, as before.

The results of this experiment are summarized in Figure 11. As expected, these results indicate that UNIONs of JOINs outperform JOINs of UNIONs.

To further understand the behavior of these queries we repeated the experiment now expressing JOIN conditions only using concatenation operations as described before. That is, we compare queries of the form

```
SELECT * FROM (
 SELECT 'http://example/'||u1 as s, u1 FROM t1
 UNION ALL
 SELECT 'http://example/'||u1 as s, u1 FROM t2
) v1
JOIN (
 SELECT 'http://example/'||u1 as s, u1 FROM t1
 UNION ALL
 SELECT 'http://example/'||u1 as s, u1 FROM t2
) v2 ON v1.s = v2.s WHERE v2.u1 = 555
```

against the equivalent of the form

```
SELECT * FROM t1 v1 JOIN t1 v2 ON
 'http://example/'||v1.u1 = 'http://example/'||v2.u1
 WHERE v2.u1 = 555
UNION ALL
SELECT * FROM t1 v1 JOIN t1 v2 ON
 'http://example/'||v1.u1 = 'http://example/'||v2.u1
 WHERE v2.u1 = 555
UNION ALL
SELECT * FROM t1 v1 JOIN t1 v2 ON
 'http://example/'||v1.u1 = 'http://example/'||v2.u1
 WHERE v2.u1 = 555
UNION ALL
SELECT * FROM t1 v1 JOIN t1 v2 ON
 'http://example/'||v1.u1 = 'http://example/'||v2.u1
 WHERE v2.u1 = 555
```

Surprisingly, as we can see from Figure 12, the results show that queries expressed as JOINs of UNIONs tend to outperform queries expressed as UNIONs of JOINs when JOIN conditions contain string concatenation. This indicates that when JOIN are

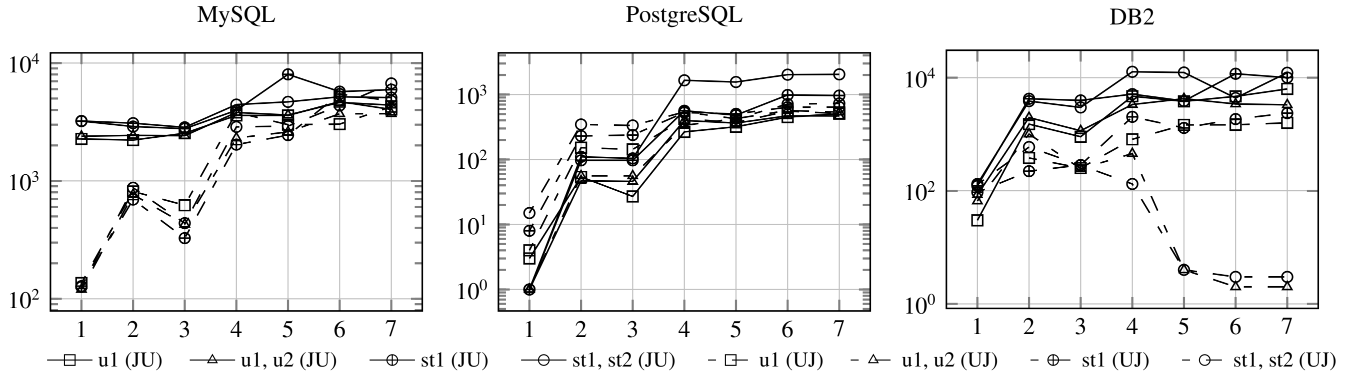


Figure 11: UNION of JOINs vs. JOIN of UNIONs. JOINs over table columns. Time in milliseconds, log scale.

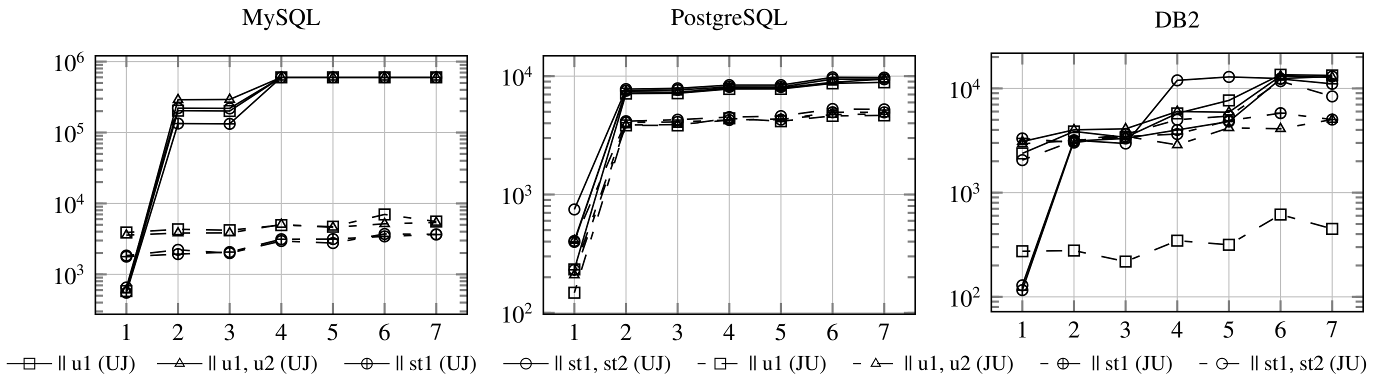


Figure 12: UNION of JOINs vs. JOIN of UNIONs. JOINs over string concatenation. Time in milliseconds, log scale.

expensive (due to string concatenation) it is better not to push the JOIN inside the unions.

Unsatisfiable conditions. In this experiment we isolate the effect in performance unsatisfiable conditions in queries. We focus on unsatisfiable conditions on string concatenation, of the form used to create URIs as specified by URI templates.

In the first experiment we study the effect of unsatisfiable conditions in JOINs of UNIONs. As we mentioned in the previous section, unsatisfiable conditions in JOINs on UNIONs often only make sense once we consider the query plans of these queries in which the DB actually *expands* the JOIN of UNIONs to a JOIN of UNIONs. For example, in the query

```
SELECT v1.s FROM (
 SELECT 'http://example/d1/'||u1 as s, u1 FROM t1
 UNION ALL
 SELECT 'http://example/d2/'||u1 as s, u1 FROM t2
) v1
JOIN (
 SELECT 'http://example/d1/'||u1 as s, u1 FROM t1
 UNION ALL
 SELECT 'http://example/d2/'||u1 as s, u1 FROM t2
) v2
ON v1.s = v2.s
WHERE v2.u1 = 666
```

the condition `v1.s = v2.s` is only unsatisfiable for half of the rows resulting from the cross product involved in the query. In particular, it is only unsatisfiable in rows that try to match strings of the form `'http://example/data1'||u1` with strings of the form `'http://example/data2'||u1`. In other words, if we transform the query into a UNION of JOINs, we obtain the query

```
SELECT * FROM t1 v1 JOIN t1 v2 ON
 'http://example/d1/'||v1.u1 =
 'http://example/d2/'||v2.u1
WHERE v2.u1 = 666
UNION ALL
SELECT * FROM t1 v1 JOIN t2 v2 ON
 'http://example/d1/'||v1.u1 =
 'http://example/d2/'||v2.u1
WHERE v2.u1 = 666
UNION ALL
SELECT * FROM t2 v1 JOIN t1 v2 ON
 'http://example/d1/'||v1.u1 =
 'http://example/d2/'||v2.u1
WHERE v2.u1 = 666
UNION ALL
SELECT * FROM t2 v1 JOIN t2 v2 ON
 'http://example/d1/'||v1.u1 =
 'http://example/d2/'||v2.u1
WHERE v2.u1 = 666
```

Comparing these queries is very similar to comparing JOIN of UNIONs against UNION of JOINs, however, in this case the strings created by concatenation do not always match, and hence, the DB engine could detect this and optimize the queries accordingly. As before, we tested with the 4 different types of join (with concat), and decreased the selectivity as usual (7 variations) for a total of 56 queries.

The results of this experiment where the same (i.e., negligible variation) those in our tests of UNIONs of JOINs against JOIN of UNIONs in the presence of string concatenation. This indicates that indeed, this kind of unsatisfiable conditions cannot be detected and by the database engine.

Redundant JOINs w.r.t. KEYs. It is well-known that removing redundant JOINs improve the performance of SQL queries, therefore, we did not evaluate the this scenario.

Conclusions. The previous experiments confirmed that the SQL queries that we call *optimized* perform considerably better than their corresponding non-optimized version. All the features we highlighted contribute to this performance difference. From the four features we listed, the ones that appear to be more critical are the presence of conditions over string concatenation functions as well as pushing the JOINs into the UNIONs.

Given that the answer to a SPARQL query should be a set of RDF terms (that contain URIs) removing boolean condition and concatenation operations from the SQL UNIONs is non-trivial. This task becomes particularly complex in presence of multiple and incompatible forms of templates

In the following section we present a technique that is able to apply all these transformations in a sound way, generating optimized queries when it is possible to do so.

7. Optimization through Partial Evaluation and SQO

Partial evaluation is an optimization technique from logic programming. The intuitive idea behind partial evaluation is that, given a fixed goal, it is possible to compute variations of a logic program which are more efficient to execute. Semantic Query Optimization (SQO) is a research area from the fields of Deductive Databases and SQL query optimization where the main objective is to optimize queries by performing static and semantic analysis of queries. In the current section we will show how to use both to optimize the SQL query programs produced by our technique to avoid the issues discussed in the previous section.

7.1. Partial Evaluation for SPARQL-to-SQL

In this Section we show how to use partial evaluation with respect to a goal (from now on simply *partial evaluation*) to optimize the *SPARQL query programs* (see Definition 32) to avoid conditions on functional terms, UNION-subqueries and conditions on UNION-subqueries. We show how to extend the original definitions of partial evaluation as 1. to deal with the nested expressions used in our SPARQL-to-Datalog technique, 2. to deal with partial evaluations of rules involving LeftJoin, and 3. to stop partial evaluations from evaluating rules that are ready to be translated into SQL queries. For the original definitions see *partial evaluation with respect to a goal* in Section 3.1.

It is worth noticing that negation only occurs in filter atoms, and we use **NOT** instead of **not**. Therefore, the program we obtain is **not**-free.

We start by showing that even without extensions, applying partial evaluation allows to eliminate all the aforementioned issues when the original SPARQL query does not involve OPTIONAL clauses.

Given a SPARQL query Q, an R2RML mapping M and it's SQL query program Π_Q^M, we compute a partial evaluation of the atom ans_1 in Π_Q^M and stop when all non-failing branches are formed by resultants whose bodies are composed only by database atoms (i.e., their predicates stand for database relations) or boolean atoms (as in boolean conditions of the query).

Example 12. *Consider the SPARQL query and R2RML mappings from Example 10. The partial evaluation would start with a root resultant of the form*

$$ans1(x, y) \text{ :- } ans1(x, y)$$

and would iteratively resolve against Π_Q^M *(see Figure 13a). The progression of the partial evaluation is shown in Figure 13b.*

Note how in the end of the partial evaluation we only two successful branches ending in the resultants $r1_8$ *and* $r1_9$*, which constitute the result of the partial evaluation. Also note that when translated into SQL by our technique, these rules generate exactly the optimal SQL queries shown in Example 7 (d).*

The first thing to note is that the partial evaluation process *is a query answering procedure* and filters out options that would not generate valid answers. For example, this is what happens with $r1_3$ in our example when it generates $r1_4$ and $r1_5$. Note that there are 4 candidate rules for the `triple` in $r1_3$, however, only two of them unify with the atom due to the presence of the constants `rdf:type` and `:Student`. We have a similar, but stronger situation for the `triple` atoms in $r1_6$ and $r1_7$. Again, there we have 4 rules that could potentially unify, however, in each case, only one rule can actually unify with the atom. In the case of $r1_6$, we have that only $r6$ unifies due to the presence of the constant *:name* and the functional symbol `cc(": stud1/", id)` (similar for $r1_7$). This makes it so that only rules that can produce satisfactory answers are used during the translation process, strongly simplifying the output.

The second thing to note is that partial evaluation deals with multiple choices by distribution, eliminating unions in the process. That is, whenever there are more than one possibility to partially evaluate an atom, the definition of partial evaluation forces the partial evaluation engine to branch. Hence, all unions are *expanded*. For an example see again the situation with the `triple` atom in in $r1_3$ where there are 2 rules that unify with the atom, and that branch $r1_3$ into $r1_4$ and $r1_4$. This affects not only SPARQL queries with UNION, but any situation in which multiple choices may be involved, like in our example in which multiple mappings exist for a given predicate (common situation in data integration scenarios) or when the OBDA system supports entailment regimes for RDFS or OWL 2 QL (query rewriting techniques usually introduce new rules in the program to cope with entailments).

Last, note that the functional terms introduced by mappings (e.g., the concatenation operators) are *moved* during the derivations from the rules introduced by the mappings, to the head and

body of the resultants and step by step, all non-database atoms are replaced by database atoms. For example, this is what happens in $r1_3$ in our example when it generates $r1_4$ and $r1_5$, in both cases the *triple* atom is replaced by a database atom and the derivation process moves the functional terms to locations in the query in which those functional terms participate. Moreover, the derivation process gets rid of these functional terms in the end, as can be seen from the triple atoms in $r1_6$ and $r1_7$. This has the strong effect that in the end of the process, no conditions are expressed over functional terms and these only appear in the head of the queries, where they do not affect performance of query execution.

Extensions for LeftJoin. The original definitions for partial evaluations were not capable of dealing with Join or LeftJoin atoms, as required by our translation, since this distinguished atoms have a semantics not encoded in the program itself. Recall that LeftJoin, for instance, encodes a set of rules. LeftJoin atoms must be handled in a way that maintains the correct semantics of the system, in particular, we cannot allow the right component of a LeftJoin atom to branch into more than one branch (since LeftJoin is a non-distributable operation on the right side relation) and we must ensure that if no rules unify with the right side of a LeftJoin, then the appropriate bindings to the `null` constant are generated. Now we provide the extensions to the definitions that allow for this to happen. We start by extending the definition of goal derivation to be able to deal with nested expressions (Join and LeftJoin).

Definition 34 (goal derivation if nested atoms). *Let G be the goal* $\leftarrow A_1, \ldots, Join(\ldots, A_m, \ldots), \ldots, A_k$ *and C be a program clause of the form* $A \leftarrow B_1, \ldots, B_q$*. Then G' is derived from G and C using the most general unifier (mgu)* θ *if the following conditions hold:*

- A_m *is an atom in G, called the selected atom,*
- θ *is a mgu of* A_m *and A, and*
- G' *is the goal*

$$\leftarrow (A_1, \ldots, Join(\ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots), \ldots, A_k)\theta$$

where $(A_1, \ldots, A_n)\theta = A_1\theta, \ldots, A_n\theta$ *and* $A\theta$ *is the atom obtained from A applying the substitution* θ

Goal derivation of atoms nested in LeftJoins is defined analogously.

Now we extend the definition of SLD-tree as to a) avoid branching on the right side of a left join, b) stop the derivation when all atoms are database atoms (extensional atoms).

Definition 35 (partial SLD-tree with Join and LeftJoin). *Let* Π *be a program and let G be a goal. Then, a (partial) SLD-Tree of* $\Pi \cup \{G\}$ *is a tree satisfying the following conditions:*

- *Each node of the tree is a resultant,*
- *The root node is* $G\theta_0 \leftarrow G_0$*, where* $G\theta_0 = G_0 = G$ *(i.e.,* θ_0 *is the empty substitution),*
- *Let* $G\theta_0 \ldots \theta_i \leftarrow G_i$ *be a node at depth* $i \geq 0$ *such that* G_i *and* A_m *be the selected atom in* G_i*. Then, for each input clause* $A \leftarrow B_1, \ldots, B_q$ *such that* A_m *and A are unifiable with mgu* θ_{i+1}*, the node has a child*

$$G\theta_1 \cdot \theta_2 \ldots \theta_{i+1} \leftarrow G_{i+1}$$

where G_{i+1} *is derived from* G_i *and* A_m *by using* θ_{i+1}*, except when* A_m *is the second argument in a LeftJoin atom,*
- *Nodes that are the empty clause have no children.*

Given a branch of the tree, we say that it is a ***failing branch*** *if it ends in a node such that the selected atom does not unify with the head of any program clause. Moreover, we say that a* ***SLD-tree for a SPARQL-to-SQL translation is complete*** *if all non-failing branches end in resultants in which only defined atoms (non-database atoms) appear only on the right side of LeftJoin atoms.*

Correctness of this definition of this extension of partial evaluation in the context of SPARQL-to-SQL translations follows from Theorem 4.3 in [20] (which states soundness and completeness of partial evaluations) and the following observations:

1. the definition of goal derivation is equivalent to the original one if we consider the way in which we interpret Join and LeftJoin atoms (i.e., syntactic shortcuts for a set of rules)
2. extensional atoms are the only atoms that may be grounded in SPARQL-to-SQL programs, and only through the data in the DB.

7.2. Semantic Query Optimization.

Semantic Query Optimization (SQO) [8, 17] refers to techniques that do semantic analysis of SQL/Datalog queries to transform them into a more efficient form, for example, by removing redundant JOINs, to detecting unsatisfiable or trivially satisfiable conditions, etc. SQO techniques often focus on the exploitation of database dependencies (e.g., SQL constraints) to do this analysis, and rely heavily on query containment theory. Most work on SQO was developed in the context of rule based formalisms (e.g., Datalog) and can be directly applied to our framework.

We will highlight two optimizations that we found critical in obtaining the best performance in most situations and that are implemented in the -ontop- system. In particular, optimization of queries with respect to keys and primary keys, and optimization of queries with respect to boolean conditions. In both cases, these optimizations are applied during or after the partial evaluation procedure.

Keys and Primary Keys. Recall that keys and primary keys are a form of equality generating dependencies (egd) [1]. That is, a (primary) key over a relation *r* defines a set of dependencies of the form

$$y_1^i = y_2^i \leftarrow r(\vec{x}, \vec{y_1}), r(\vec{x}, \vec{y_2})$$

for each y_1^i, y_2^i in $\vec{y_1}$ and $\vec{y_2}$, respectively.

For example, given a table *stud*1 as in our previous example, the primary key on the first column, *id* would generate the

```
r1 : ans1(x, y) :- ans2(x), ans3(x, y)
r2 : ans2(x) :- triple(x, "rdf:type", ":Student")
r3 : ans3(x, y) :- triple(x, ":name", y)
r4 : triple(cc(":stud1/", id), "rdf:type", " : Student") :- stud1(id, name)
r5 : triple(cc(":stud2/", id), "rdf:type", " : Student") :- stud2(id, name)
r6 : triple(cc(":stud1/", id), ":name", name) :- stud1(id, name)
r7 : triple(cc(":stud2/", id), ":name", name) :- stud2(id, name)
```

(a) SQL Query Program Π_Q^M for the SPARQL query and mappings in Example 10

```
r1₁ : ans1(x, y) :- ans1(x, y)
      by r1₁, r1 with θ = {x/x1, y/y1}
r1₂ : ans1(x1, y1) :- ans2(x1), ans3(x1, y1)
      by r1₂, r2 with θ = {x/x1}
r1₃ : ans1(x1, y1) :- triple(x1, "rdf:type", ":Student"), ans3(x1, y1)
      by r1₃, r4 with θ1 = {x1/cc(":stud1/", id1)} and r1₃, r5 with θ2 = {x1/cc(":stud2/", id1)}
r1₄ : ans1(cc(":stud1/", id1), y1) :- stud1(id1, name1), ans3(cc(":stud1/", id1), y1)
r1₅ : ans1(cc(":stud2/", id1), y1) :- stud2(id1, name1), ans3(cc(":stud2/", id1), y1)
      by r1₄, r3 with θ1 = {x/cc(":stud1/", id1), y1/y} and r1₅, r3 with θ2 = {x/cc(":stud2/", id1), y1/y}
r1₆ : ans1(cc(":stud1/", id1), y) :- stud1(id1, name1), triple(cc(":stud1/", id1), ":name", y)
r1₇ : ans1(cc(":stud2/", id1), y) :- stud2(id1, name1), triple(cc(":stud2/", id1), ":name", y)
      by r1₆, r6 with θ1 = {id/id1, y/name} and r1₇, r7 with θ2 = {id/id1, y/name}
r1₈ : ans1(cc(":stud1/", id1), name) :- stud1(id1, name1), stud1(id1, name)
r1₉ : ans1(cc(":stud2/", id1), name) :- stud2(id1, name1), stud2(id1, name)
```

(b) Progression of the partial evaluation w.r.t. the goal ans1

Figure 13: Partial Evaluation of ans_1 in Π_Q^M. Each block shows the leafs of the non-failing branches of the SLD-Tree.

following equality generating dependencies:

$$x = x \leftarrow stud1(x, y1), stud(x, y2) \quad (6)$$

$$y1 = y2 \leftarrow stud1(x, y1), stud(x, y2) \quad (7)$$

By chasing the dependencies (e.g., applying the equalities to the query) we will be able to either detect unsatisfiable queries, i.e., when two different constants are equated, or generate duplicated atoms which can be safely eliminated since a query with a duplicated atom is always equivalent to the query in which this atoms is removed (with respect to the standard query containment notions [1]). For example, given the query

$$ans1(id1, name) \text{ :- } stud1(id1, name1), stud1(id1, name)$$

by chasing egd 7 one obtains the query

$$ans1(id1, name) \text{ :- } stud1(id1, name), stud1(id1, name)$$

which can be simplified to

$$ans1(id1, name) \text{ :- } stud1(id1, name)$$

Boolean conditions. Another kind of optimization that can have a strong impact on performance is the detection of unsatisfiable boolean conditions, or the simplification of queries with respect to trivially satisfiable conditions.

Unsatisfiable conditions often arise from the partial evaluation process, in particular, when queries contain FILTER expressions and mappings involve constants. For example, consider the following SPARQL query asking for all people that attend either NYC or Columbia universities

```
SELECT ?x WHERE {
       ?x :attends ?y.
       FILTER (?y = :NYC || ?y = :Columbia)
}
```

Now consider a mapping for the *attends* property that states that all people from the table *stud1* attend Stanford university, as follows

```
triple(cc("stud1/id", id), ":attends", ":Stanford") :-
                                        stud1(id, name)
```

then we would have that after the partial evaluation process we would end up with the following partial evaluation:

```
triple(cc("stud1/id", id)) :- stud1(id, name),
  OR(":Stanford" = ":NYC", ":Stanford" = ":Columbia")
```

Clearly, this query is empty. However, when translated into SQL and executed, few relational DBs would be able to realize this since the analysis of arbitrary boolean expressions is beyond the scope of most query optimizers.

A similar situation arises with trivially satisfied conditions such as $1 = 1$, $x = x$, etc, which can be simplified or eliminated.

Implementing this kind of optimization requires a partial evaluation engine that is aware of boolean logic, as well as the semantics of some built-in operators such as the *concat* operator, the SPARQL built-in functions and all the arithmetic operators. While this kind of optimization is not theoretically very

interesting, we have found that in -ontop-, this functionality enables the system to deal effectively with complex situations in which mappings are not trivial and which otherwise would generate slow queries, even in commercial database engines.

8. Evaluation

This section provides an evaluation of our SPARQL-to-SQL technique implemented in -ontop- and using DB2 and MySQL as backends. We compared -ontop- with two systems that offer similar functionality to -ontop- (i.e., SPARQL trough SQL and mappings): Virtuoso RDF Views 6.1 (open source edition) and D2RQ 0.8.1 Server over MySQL. We also compare -ontop- with three well known triple stores: OWLIM 5.3, Stardog 1.2 and Virtuoso RDF 6.1 (Open Source). Another system that is relevant in this experiments is Ultrawrap [32], however the system is commercial and we were not able to obtain a copy for evaluation.

Systems. Next, we provide a brief description of each of the systems we benchmarked:

-ontop- is an open source framework developed in the Free University of Bozen-Bolzano to query databases as Virtual RDF Graphs using SPARQL 1.0. It relies on the query rewriting techniques presented in Section 4; and supports RDFS, OWL 2 QL reasoning, and all major DBMS (open-source and commercial) as backends. Regarding the mappings languages, -ontop- supports both, R2RML and its own mapping language. -ontop- comes in three different flavors: as an API, as a plugin for Protege 4—that provides a mapping editor—and as a SPARQL end-point based in Sesame.

D2RQ is a system for accessing relational databases as virtual RDF graphs. It is developed and maintained by the Free University of Berlin, and DERI. It offers SPARQL 1.1 query-answering to the content of relational databases without having to replicate it into an RDF store. It provides its own mapping language, and currently it does not support neither R2RML nor reasoning (with OWL ontologies).

OWLIM is a commercial semantic repository developed by Ontotext that allows to query, reason, and update RDF data. It relies on top of Sesame API; supports several query languages, such as, SeRQL and SPARQL 1.1; and several fragments of OWL, such as, OWL 2 RL and OWL 2 QL.

Virtuoso 6.1 Open Source is a hybrid server developed by OpenLink Software that allows relational, RDF, and XML data management. For this reason it can handle SPARQL and SQL queries. In this paper we work with the Open-Source edition. Although the Open-Source edition does not include some of the features available in the commercial edition such as, clustering, database federation, etc.; such features are not relevant for this evaluation.

This section extends the work presented in [31].
http://protege.stanford.edu
http://www.openrdf.org

Virtuoso Views is developed by OpenLink Software, and allows the RDF representation of the relational data. It supports a proprietary mapping language that is equivalent in expressivity to R2RML (minus a few minor features). We used the free version of Virtuoso Views which can only be used in conjunction with open link's own relational DB (no MySQL or DB2).

Stardog is a commercial RDF database developed by Clart&Parsia that allows SPARQL 1.1 for queries; and OWL for reasoning. It provides a command line interface to query, load, and update the data.

Benchmarks. We considered the following benchmarks:

BSBM. The Berlin SPARQL Benchmark (BSBM) [6] evaluates the performance of query engines utilizing use cases from e-commerce domain. The benchmark comes with a suite of tools for data generation and query execution. The benchmark also includes a relational version of the data, for which mappings can be created (D2RQ mappings are included).

FishMark. The FishMark benchmark [4] is a benchmark for RDB-to-RDF systems that is based on a fragment of the FishBase DB, a publicly available database about fish species. The benchmark comes with an extract of the database (approx. 16 M triples in RDF and SQL version), and 22 SPARQL queries obtained from the logs of FishBase. The queries are substantially larger (max 25 atoms, mean 10) than those in BSBM. Also, they make extensive use of OPTIONAL graph patterns.

These benchmarks use a total of 36 queries and 350+ million triples.

Experiment setup. The basic setup for the experiment is as follows: BSBM provides 12 query templates (i.e., queries with place holders for constant values). A predefined sequence of 25 of these templates constitutes a *Query Mix*. A BSBM run is the instantiation of a query mix with random constants and execution of the resulting queries. Performance is then measured in Query Mixes per Hour (QMpH). To compute QMpH we ran 150 query mixes, out of which 50 are considered warm up runs and their statistics are discarded. The collected statistics for QMpH over BSBM instances with 25, 100 and 200 million triples (or the equivalent in relational form). In the case of FishMark, the 22 queries are already instantiated and they constitute the query mix. We ran 150 query mixes, discarding the initial 50. We tested with 1, 4, 8, 16 and 64 simultaneous clients.

We exploited -ontop-'s simple SQL caching mechanism which stores SQL queries generated for any SPARQL query that has been rewritten previously. This allows to avoid the rewritten process completely and hence, the cost of query execution of a cached query is only the cost of evaluating the SQL query over the DBMS. To force the use of this cache, we re-ran the BSBM benchmark 5 more times (and averaged the results). For FishMark, this was not necessary since the queries are always the same. All experiments were conducted on a HP Proliant server with 24 Intel Xeon CPUs (144 cores @3.47GHz), 106GB of RAM and a 1TB 15K RPM HD. The OS is Ubuntu 12.04 64-bit edition. All the systems run as SPARQL end-

points. All configuration files are available online.

Discussion. The results are summarized in Figures 15, 16 and 17. In Figure 15 we show the QMpH for the 12 BSBM queries. Query 6 is not included since it contains a regex function that was not supported by -ontop- when this evaluation was done. We also included in this table the query rewriting time (SPARQL to SQL). Observe that the time required for this step is independent of the database, and of the selectivity of the query, and usually takes few milliseconds. It is not possible to give a precise general statement about the impact of the query rewriting time in whole query execution time given that BSBM queries have place holders that are filled in each execution, changing the answer given by the query. However, given an instantiation of a query, we can give the percentage of the query execution time taken by the query rewriting step. We run an institution of the query mix in BSBM 200 (MySql), and we observed that the query rewriting step takes around 20%- 40% of the execution time. The queries had very high selectivity, therefore the execution time is small. For instance, -ontop- requires 4ms to rewrite Query 1, and 17ms to perform the whole execution (including rewriting). The harder is the execution in the database, the smaller is the impact of the query rewriting step.

In Figures 16 and 17, first we note that the D2RQ server always ran out of memory, timed out in some queries or crashed. This is why it doesn't appear in our summary table. D2RQ's SPARQL-to-SQL technique is not well documented, however, by monitoring the queries being sent by D2RQ to MySQL, it appears that D2RQ doesn't translate the SPARQL query into a single SQL query, instead it computes multiple queries and retrieves part of the data from the database. We conjecture that D2RQ then uses this data to compute the results. Such approach is, in general, limited in scalability and prone to large memory consumption, being the last point the reason for the observed behavior. Also, Virtuoso Views is not included in the FishMark benchmark because it provided wrong results, we reported this to the developers which confirmed the issue. Also, we did not run -ontop- with DB2 for FishMark due to errors during data loading.

In Figures 16 we also included the original BSBM SQL queries. We run these queries directly over the database engine, therefore the execution time includes neither the rewriting time, nor the time to post-process the SQL result set to generate an RDF result set. The performance obtained by MySQL is clearly much better than the one obtained by all the other Q&A systems, although the gap gets smaller as the dataset increases. It is worth noting that these queries are not SQL translation of SPARQL queries, thus they are intrinsically simpler, for instance, by not considering URIs.

Next, we can see is that for BSBM in almost every case, the performance obtained with -ontop-'s queries executed by MySQL or DB2 outperforms all other Q&A systems by a large margin. The only cases in which this doesn't hold are when the number of clients is less than 16 and the dataset is small (BSBM 25). This can be explained as follows: -ontop-'s performance can be divided in three parts, *(i)* the cost of generating the SQL query , *(ii)* the cost of execution over the RDBMs and *(iii)* cost of fetching and transforming the SQL results into RDF terms. When the queries are cached, *(i)* is absent, and if the scenario includes little data (i.e., BSBM 25), the cost of *(ii)*, both for MySQL and DB2, is very low and hence *(iii)* dominates. We attribute the performance difference to a poor implementation of *(iii)* in -ontop-, and the fact triple stores do not need to perform this step. However, when the evaluation considers 16 parallel clients, executing -ontop-'s SQL queries with MySQL or DB2 outperforms other systems by a large margin. We attribute this to DB2's and MySQL's better handling of parallel execution (i.e., better transaction handling, table locking, I/O, caching, etc.). When the datasets are larger, e.g., BSBM 100/200, -ontop- *(i)* stays the same. In these cases, *(ii)* dominates *(iii)*, since in both benchmarks queries return few results. The conjunction of good planning, caching, and I/O mechanisms provided by MySQL and DB2, and the efficient -ontop-'s SQL, allows our system to outperform the rest already at 1 single client for BSBM 100 and BSBM 200.

We can see the strong effect of SELF JOIN elimination by Primary Keys. Consider the FishMark benchmark that has little data, only 16M triples, but in which in almost all queries -ontop-'s SQL executed over MySQL (we didn't run DB2 in this case) outperforms the rest almost in every case even from 1 single client. In this setting, 1 client and little data, the cost of *(ii)* falls in the cost of planning and executing JOINs and LEFT JOINs by the DBMS or triple store. At the same time, in FishMark, the original tables are structured in such a way that many of the SPARQL JOINs can be simplified dramatically when expressed as optimized SQL. For example, consider the FishMark query in Figure 14a. This query expresses a total of 16 Join operations. When translated into SQL, -ontop- is able to generate the query in Figure 14b.

A simple and flat SQL query (easy to execute) with a total of 3 Joins. Note that the use of a large number of JOIN operations is intrinsic to SPARQL since the RDF data model is ternary. However, if the data is stored in a n-ary schema (as usual in RDBMs), -ontop- can use semantic query optimization w.r.t. primary keys to construct the optimal query over the n-ary tables. Triple stores has no means to do this since data is de-normalized once it is transformed into RDF.

9. Conclusion and Future Work

The main contribution of this paper is a formal approach for SPARQL-to-SQL translation that generates efficient SQL by adapting and combining standard techniques from logic programming. In this context, we discussed several SQL features that affects performance, and showed how to avoid them. In addition, we presented a rule based formalization of R2RML mappings that can be integrated into our technique to support mappings to arbitrary database schemas.

To evaluate our approach, we provided compared the -ontop- system with well known RDB2RDF systems and triple stores,

https://babbage.inf.unibz.it/trac/obdapublic/wiki/BSBMFISH13aBench

```
SELECT ?forder ?family ?genus ?species ?occ ?name ?gameref ?game
WHERE {  ?ID fd:cComName ?name; fd:coC_Code ?ccode; fd:cSpecCode ?x.
            ?x  fd:sGenus ?genus; fd:sSpecies ?species; fd:sGameFish ?game;
                fd:sGameRef ?gameref; fd:sFamCode ?f .
            ?f  fd:fFamily ?family; fd:fOrder ?forder .
            ?c  fd:cSpecCode ?x; fd:cStatus ?occ; fd:cC_Code ?cf;
                fd:cGame 1 . ?cf fd:cPAESE "Indonesia" . }
```

(a) FishMark query

```
SELECT V3.FamilyOrder AS forder, V3.Family AS family, V1.Genus AS genus, V1.Species AS species,
  V4.Status AS occ, V1.ComName AS name, V1.GameRef AS gameref, V1.GameFish AS game
FROM species V1, comnames V2, families V3, country V4, countref V5
WHERE V1.SpecCode = V2.SpecCode AND V4.Game = 1 AND V5.PAESE = 'Indonesia' AND V4.C_Code = V5.C_Code
      AND V1.Genus = V8.Genus
```

(b) SQL translation by -ontop-

Figure 14: Optimization by -ontop-

	Rw Time (ms)	-ontop--MySQL	-ontop--DB2	OWLIM	Stardog	V. RDF	V. Views
Q1	4	3,22k	1,28 k	1,43k	1,42	23	45
Q2	12	1,40k	928	276	790	123	315
Q3	15	1,92k	1,12k	111	543	155	341
Q4	12	1,53k	955	295	669	140	265
Q5	15	27	72	2	29	14	48
Q6	-	-	-	-	-	-	-
Q7	9	9,78k	1,59k	541	400	101	316
Q8	4	13,99k	1,68k	3,22k	648	96	180
Q9	15	13,62k	1,01k	3,7k	1,99k	59	188
Q10	5	20,22k	2,04k	3,9k	610	228	305
Q11	18	20,75k	2,04k	3,52k	1,89k	1,66k	1,13k
Q12	19	11,34k	1,64k	5,99k	1,4k	1,25k	1,19k
QueryMix	-	44,19k	76,96k	2,91k	35,79k	9,21k	25,11k

Figure 15: Summary of results for BSBM-200 with 64 clients. Per query units are in *queries per second*, totals are in *query mixes per hour*

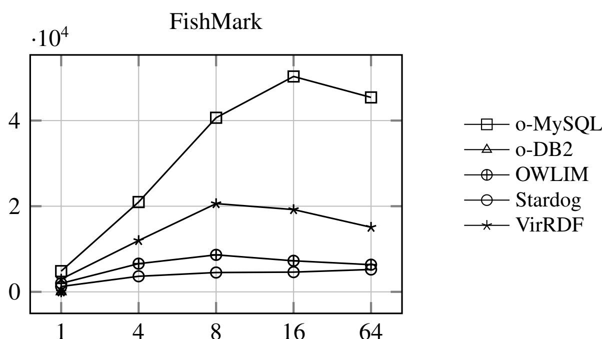


Figure 17: Query performance for FishMark. X axis = parallel clients, Y axis = Query Mixes per Hour (QMpH, higher is better)

showing that using the techniques presented here allows -ontop- to outperform all other systems.

One of the key benefits of our framework, is the possibility of extending it in many directions. For example, manipulating the Datalog programs to support inference for RDFS, OWL and SWRL (OWL 2 QL inference is already supported in -ontop-), as well as easy integration of more advanced semantic query optimization. We will work in these directions in the near future.

Acknowledgements

We thank the -ontop- development team (J. Hardi, T. Bagosi, M. Slusnys) for the help with the experiments. This work was supported by the EU FP7 project Optique (grant 318338).

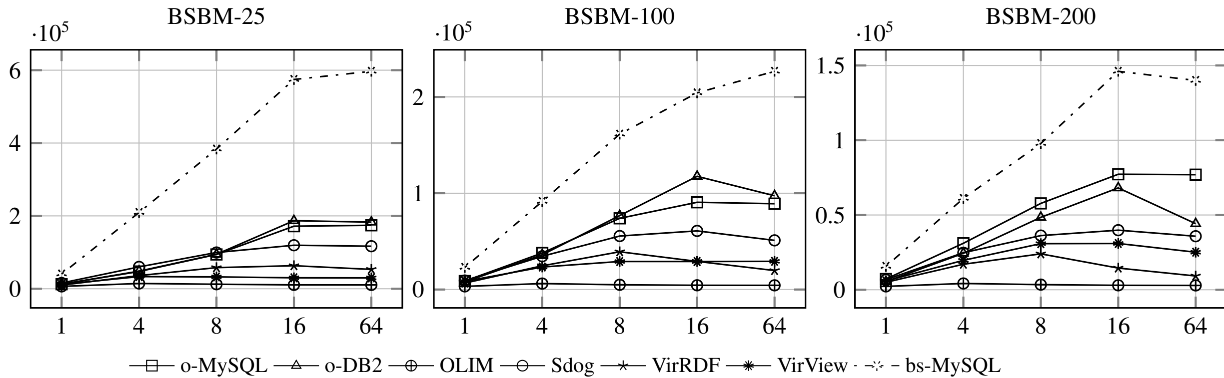


Figure 16: Query performance for BSBM. X axis = parallel clients, Y axis = Query Mixes per Hour (QMpH, higher is better)

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, pages 114–129, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29:841–862, July 1982.

[4] Samantha Bail, Sandra Alkiviadous, Bijan Parsia, David Workman, Mark van Harmelen, Rafael S. GonÃSalves, and Cristina Garilao. FishMark: A linked data application benchmark. In *Proc. of the Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+HPCSW 2012)*, volume 943, pages 1–15. 2012.

[5] C. Baral, M. Gelfond, and A. Provetti. Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming*, 1997.

[6] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal On Semantic Web and Information Systems*, 2009.

[7] Diego Calvanese, Davide Lanti, Martin Rezk, Mindaugas Slusnys, and Guohui Xiao. The NPD benchmark for OBDA systems. In *ORE*, 2014.

[8] Upen S Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, 1990.

[9] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data Knowl. Eng.*, 68(10):973–1000, October 2009.

[10] Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF Mapping Language. http://www.w3.org/TR/r2rml/, September 2012.

[11] David J. DeWitt. The wisconsin benchmark: Past, present, and future. In Jim Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann.

[12] Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, and Z. Meral Ozsoyoglu. A complete translation from sparql into efficient sql. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, IDEAS '09, pages 31–42, New York, NY, USA, 2009. ACM.

[13] Orri Erling. Implementing a sparql compliant rdf triple store using a sql-ordbms. Technical report, OpenLink Software, 2001. Available at `http://www.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSRDFWP`.

[14] Orri Erling. Virtuoso, a hybrid RDBMS/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[15] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceeding of the Fifth Logic Programming Symposium*, pages 1070–1080, 1988.

[16] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. 3(2–3):158–182, 2005.

[17] Jonathan Jay King. Query Optimization by Semantic Reasoning. 1981.

[18] Alexander Leitsch. *The resolution calculus*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[19] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *ICLP*, pages 23–37, 1994.

[20] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Log. Program.*, 11(3-4):217–242, 1991.

[21] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1993.

[22] J.W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer-Verlag, 1987.

[23] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. RStar: an RDF storage and query system for enterprise resource management. In *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13*, pages 484–491, 2004.

[24] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[25] Hector Perez-Urbina, Edgar RodrÃguez-Diaz, Michael Grove, George Konstantinidis, and Evren Sirin. Evaluation of query rewriting approaches for OWL 2. In *In Proc. of the Joint Workshop on Scalable and High-Performance Semantic Web Systems, volume 943 of CEUR-WS*, 2012.

[26] Axel Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 787–796, New York, NY, USA, 2007. ACM.

[27] Axel Polleres and Johannes Peter Wallner. On the relation between SPARQL1.1 and Answer Set Programming. *Journal of Applied Non-Classical Logics*, 23(1-2):159–212, 2013.

[28] Freddy Priyatna, Oscar Corcho, and Juan Sequeda. Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 479–490, Republic and Canton of Geneva, Switzerland, 2014. International World Wide Web Conferences Steering Committee.

[29] T. C. Przymusinski. *On the declarative semantics of deductive databases and logic programs*, pages 193–216. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[30] M Rodríguez-Muro. *Tools and Techniques for Ontology Based Data Access in Lightweight Description Logics*. PhD thesis, Free Univ. of Bozen-Bolzano, 2010.

[31] Mariano Rodriguez-Muro, Martín Rezk, Josef Hardi, Mindaugas Slusnys, Timea Bagosi, and Diego Calvanese. Evaluating sparql-to-sql translation in ontop. In *ORE*, pages 94–100, 2013.

[32] Juan F. Sequeda and Daniel P. Miranker. Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 22(0):19 – 39, 2013.

[33] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[34] F. Zemke. Converting sparql to sql. Technical report, Oracle Corporation, 2006. Available at `http://lists.w3.org/Archives/Public/public-rdf-dawg/2006OctDec/att-0058/sparql-to-sql.pdf`.

Benchmark	Size	System	Number of clients					Loading
			1	4	8	16	64	time
BSBM	25M	onTop-MySQL	12,437	47,325	94,094	171,702	174,153	00:01:57
		onTop-DB2	12,506	48,299	94,593	186,837	182,862	00:02:12
		OWLIM	6,455	14,439	12,338	10,617	10,509	00:12:21
		Stardog	15,751	59,969	99,471	119,390	116,726	00:02:54
		Virtuoso RDF	9,823	36,112	57,966	63,253	53,124	00:18:41
		Virtuoso Views	11,992	33,354	32,491	30,054	29,786	00:36:43
	100M	onTop-MySQL	8,771	37,923	73,920	90,587	89,095	00:11:33
		onTop-DB2	7,873	36,070	76,767	117,564	97,366	00:15:20
		OWLIM	3,238	6,173	4,907	4,384	4,329	00:49:45
		Stardog	9.311	34.523	55,527	60,876	51,038	00:11:47
		Virtuoso RDF	6,665	24,618	39,183	29,198	19,702	01:13:04
		Virtuoso Views	8,449	23,347	29,098	29,093	29,245	02:01:05
	200M	onTop-MySQL	7,171	31,080	57,753	77,250	76,962	00:25:13
		onTop-DB2	5,442	24,389	48,416	68,122	44,193	00:24:38
		OWLIM	2,249	4,196	3,429	2,959	2,905	01:43:10
		Stardog	6,719	24,769	36,222	39,842	35,790	00:22:59
		Virtuoso RDF	4,970	17,060	23,997	14,499	9,213	02:21:46
		Virtuoso Views	5,888	19,673	30,833	30,946	25,108	03:52:14
FishMark	16.5M	onTop-MySQL	4,835	20,975	40,657	50,295	45,405	00:01:04
		OWLIM	1,984	6,583	8,639	7,249	6,341	00:06:32
		Stardog	3,409	3,637	4,514	4,611	5,222	00:02:34
		Virtuoso RDF	2,973	11,986	20,603	19,214	15,085	00:12:21

Table 1: Result summary for all systems and datasets

Appendix A. Normalization of R2RML mappings

In the following we describe the normalization we apply to R2RML mappings that guarantee that R2RML document includes only triple maps with one single object predicate map. All these transformations are applied in the order listed here.

Appendix A.1. Expansion of rr:constants

Any shortcut for URI constants of the form

- `rr:subject C`
- `rr:predicate C`
- `rr:object C`

is replaced by

- `rr:subjectMap [ rr:constant C]`
- `rr:predicateMap [ rr:constant C]`
- `rr:objectMap [ rr:constant C]`

Appendix A.2. Expansion of SQL shortcuts

We remove all shortcuts for SQL tables and replace them with SQL queries that correspond to the *effective SQL query* defined by that shortcuts as indicated in Section 5 of the R2RML specification. For example, given the R2RML mapping:

```
_:m1 a rr:TripleMap;
 rr:logicalTable [ rr:tableName "stud" ] ;
 ...
```

we produce

```
_:m1 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM stud" ] ;
 ...
```

Appendix A.3. Expansion of rr:class shortcuts

We remove all shortcuts that generate triples of the form `s rdf:type C` and replace them with explicit predicate-object maps. For example:

```
_:m1 a rr:TripleMap;
 ...
 rr:subjectMap   [ rr:template ":stud/{id}" ;
                   rr:class C ] ;
 ...
```

we rewrite it as:

```
_:m1 a rr:TripleMap;
  ...
  rr:subjectMap   [ rr:template ":stud/{id}" ]
  rr:predicateObjectMap [
   rr:predicateMap [ rr:constant rdf:type ];
   rr:objectMap [ rr:constant ex:Employee ]
  ]
```

Appendix A.4. Expansion of predicate-object maps with multiple predicates/objects

We remove any predicate-object map m that has multiple predicate or object maps and replace it with a set of predicate-object maps that created by combining each predicate map in m with every object map in m, as to generate the triples described in Section 11.1 of the R2RML specification. For example, from the following mapping,

```
_:m1 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM stud" ] ;
 rr:subjectMap [ rr:template ":stud/{id}" ] ;
 rr:predicateObjectMap [
         rr:predicateMap [ rr:constant :name ] ;
         rr:predicateMap [ rr:constant :info ] ;
     rr:objectMap  [ rr:column "name"]].
```

we obtain

```
_:m1 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM stud" ] ;
 rr:subjectMap [ rr:template ":stud/{id}" ] ;
 rr:predicateObjectMap [
         rr:predicateMap [ rr:constant :name ] ;
     rr:objectMap  [ rr:column "name"]] ;
 rr:predicateObjectMap [
         rr:predicateMap [ rr:constant :info ]
     rr:objectMap  [ rr:column "name"]] .
```

Appendix A.5. Expansion of referencing predicate object maps

We remove all shortcuts defined by referencing object maps and replace them by a new triple map with a single predicate-object map that generates triples as specified in Section 11.1 in the R2RML specification. That is, the logical table is the *joint SQL query* of the referencing map, the subject map is the subject map of the child, and in the predicate-object, the predicate is the predicate map of the referencing object map and the object is as specified the map of the parent. Section 11.1. For example, given the following R2RML mapping:

```
_:m1 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM stud" ] ;
 rr:subjectMap [ rr:template ":stud/{id}" ] ;
 rr:predicateObjectMap [ rr:predicate :takes ;
   rr:objectMap  [ rr:parentTriplesMap _:m2 ;
   rr:joinCondition [ rr:child "ID"; rr:parent "ID" ]].

_:m2 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM course" ] ;
 rr:subjectMap [ rr:template ":course/{id}" ;
                rr:class :Course ] ;
...
```

we produce a reference-less version as follows:

```
_:m1 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM stud" ] ;
 rr:subjectMap [ rr:template ":stud/{id}" ] ;

_:m2 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM course" ] ;
 rr:subjectMap [ rr:template ":course/{id}" ;
                rr:class :Course ] ;
...

_:m3 a rr:TripleMap ;
  rr:logicalTable [ rr:sqlQuery "
  SELECT * FROM (SELECT * from stud) AS child,
                (SELECT * from course}) AS parent
          WHERE child.ID=parent.ID"] ;
 rr:subjectMap [ rr:template ":stud/{child.id}" ] ;
 rr:predicateObjectMap [
   rr:predicateMap [ rr:constant rdf:takes ];
   rr:objectMap  [ rr:template ":course/{parent.id}"  ] ]
   .
```

Appendix A.6. Splitting of triple maps with multiple predicate-object maps

Each triple map m that has n predicate-object maps where $n > 1$ we split into n fresh triple maps. Each i fresh triple map uses the logical table and subject map of the original triple map m and contains exactly one predicate-object map, the i-th predicate-object map found in m. For example, from the following triple map,

```
_:m1 a rr:TripleMap;      # First triple map
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM stud" ] ;
 rr:subjectMap [ rr:template ":stud/{id}" ] ;
 rr:predicateObjectMap [
         rr:predicateMap [ rr:constant :name ] ;
     rr:objectMap  [ rr:column "name"]].
 rr:predicateObjectMap [
         rr:predicateMap [ rr:constant :info ]
     rr:objectMap  [ rr:column "name"]] .
```

we obtain the following triple maps;

```
_:m1 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM stud" ] ;
 rr:subjectMap [ rr:template ":stud/{id}" ] ;
 rr:predicateObjectMap [
         rr:predicateMap [ rr:constant :name ] ;
     rr:objectMap  [ rr:column "name"]] .

_:m2 a rr:TripleMap;
 rr:logicalTable [ rr:sqlQuery "SELECT * FROM stud" ] ;
 rr:subjectMap [ rr:template ":stud/{id}" ] ;
 rr:predicateObjectMap [
         rr:predicateMap [ rr:constant :info ]
     rr:objectMap  [ rr:column "name"]] .
```

Appendix B. Proofs

Theorem 1. *Let Q be an SQL-compatible SPARQL query, Π_Q the Datalog encoding of Q, and $[\![ans_Q(\vec{x})]\!]$ the relational algebra statement of Π_Q. Then it holds:*

$$\vec{t} \in [\![ans_Q(\vec{x})]\!]^{\emptyset} \leftrightarrow \Pi_Q \models ans_Q(\vec{t})$$

Proof. From the definition of Π_Q it is clear that it is a stratified program of the form:

$$\Pi_Q = \Pi_0 \dots \Pi_n$$

Therefore it has a unique Herbrand model, and moreover, such model is the union of the unique models of each stratum Π_i. Recall that Π_0 is a bottom part that does not contain negation as failure (c.f. Section 3.1). The proof will be by induction on the number of splits needed to calculate the model of Π_Q. Therefore, it is enough to show that for every step k:

Note 1 (Inductive Claim:). *For every triple or defined predicate A such that for every $\vec{t}$, $\Pi_k \models A(\vec{t})$ iff $\Pi_Q \models A(\vec{t})$—that means that A must be entirely* computed *inside Π_k– it holds that:*

$$\Pi_k \models A(\vec{t}) \text{ if and only if } \vec{t} \text{ is a tuple in } [\![A(\vec{x})]\!]$$

The additional restriction that the predicate A must be entirely computed inside Π_k is to handle LeftJoin. Recall that the LeftJoin predicate is a syntactic sugar for the set of rules in Figure B.18. Observe that these rules contain **not** -atoms. Here we assume that we replace the syntactic sugar **NOT** by the original **not** in filter atoms.

Therefore, the LeftJoin as a whole is defined in the union of rules that belong to different stratums.

Observe that since the graph is finite, and the set of predicates is finite, the grounding of the program will also be finite and therefore we will eventually cover every possible ground atom $A(\vec{t})$. Recall that we only allow functional terms that has nesting depth of at most 2.

Base Case (Π_0): Recall that Π_0 is **not** -free and therefore it has a unique least Herbrand model, **M**. This implies that $\Pi_0 \models A(\vec{t})$ if and only if $A(\vec{t}) \in \mathbf{M}$. This model is computed via a sequence of bottom-up derivation steps, which apply the rules of Π_0 to the facts in Π_0 and then repeatedly to the newly derived facts. Our proof will proceed by induction on the number N of such steps.

1. $N = 0$: Then A has to be a triple predicate. Then the claim follows trivially from the definition of Π_Q, Π_0, and Definition 28.
2. $N = k + 1$: Suppose that A was derived in the $k + 1$ step. It follows that it is a defined predicate $ans_P(\vec{x})$. Then P can be a Union, a Join, or a Filter.
   - Suppose P is a Union. Then, it is defined by a set of rules of the form:

$$\begin{array}{lcl} ans_{Union}(\vec{x_1}) & :- & ans_{P_1}(\vec{z_1}) \\ & \vdots & \\ ans_{Union}(\vec{x_n}) & :- & ans_{P_n}(\vec{z_n}) \end{array}$$

     Recall that Union may add null constants to some x_i, and these are translated as AS statements in the projections in the relational algebra expression. Then, by Definition 28, it follows that $[\![ans_{Union}(\vec{x})]\!]$ is defined as follows:

$$\Pi_{\vec{x}}[\![ans_{P_1}(\vec{z_1})]\!] \cup \cdots \cup \Pi_{\vec{x}}[\![ans_{P_n}(\vec{z_n})]\!]$$

     By Inductive Hypothesis we know that ($i = 1 \ldots n$)

$$\vec{c_i} \in [\![ans_{P_i}(\vec{z_i})]\!] \text{ iff } \mathbf{M} \models ans_{P_i}(\vec{c_i})$$

     It follows that

$$\vec{c} \in [\![ans_{Union}(\vec{z_i})]\!] \text{ iff } \mathbf{M} \models ans_{Union}(\vec{c})$$

   - Suppose that P has the form

$$Filter(ans_{P_1}(\vec{w}), E(\vec{w}))$$

     where $ans_{P_1}(\vec{w})$ has been computed in k steps and E is a filter condition. The proof remains the same if we consider a *triple* atom instead. The atom $Filter(ans_{P_1}(\vec{w}), E)$ represents the body

$$ans_{P_1}(\vec{w}), E(\vec{w})$$

     By inductive hypothesis,

$$\Pi_0 \models ans_{P_1}(\vec{t}), E(\vec{t}) \text{ iff } t \in [\![ans_{P_1}(\vec{w}), E(\vec{w})]\!]$$

     It follows that

$$\Pi_0 \models ans_{Filter}(\vec{t}) \text{ iff } t \in \Pi_{\vec{x}}([\![ans_{Filter}(\vec{x})]\!])$$

   - Now suppose that P has the form

$$Join(ans_{P_1}(\vec{w_1}), ans_{P_2}(\vec{w_2}), jn)$$

     where $ans_{P_1}(\vec{w_1})$ and $ans_{P_2}(\vec{w_2})$ hav been computed in k steps and jn is the join condition. The proof remains the same if we consider a *triple* atom instead, and follows applying inductive hypothesis as above.

Inductive Case (Step $k + 1$): Notice that by Proposition 1 and the definition of splitting set (c.f. Section 3.1) we can conclude that

- The strata Π_k has a unique answer set $\mathbf{M}_k$
- The set $U_k = \{lit(r) \mid head(r) \in \mathbf{M}_k\}$ forms a splitting set for Π_Q
- By Proposition 1, it follows that **M** is an answer set of Π_Q iff

$$\mathbf{M} = \mathbf{M}_k \cup \mathbf{M}_{top}$$

  where $\mathbf{M}_{top}$ is an answer set of the partial evaluation $e_{U_k}(\Pi_Q \setminus b_{U_k}, \mathbf{M}_k)$.

Using the same proposition, it follows that $\mathbf{M}_{top}$ can be computed iteratively in the same way, that is, computing the model of the positive part of $e_{U_k}(\Pi_Q \setminus b_{U_k}, \mathbf{M}_k)$, and the continue splitting and computing the partial evaluations. Let $\mathbf{M}_e$ be the unique model of the positive part of $e_{U_k}(\Pi_Q \setminus b_{U_k}, \mathbf{M}_k)$.

Suppose that $ans_P(\vec{t}) \in \mathbf{M}_e$ and ans_P is completely defined in $\mathbf{M}_e$ in the sense specified above.

Here we assume that we replace the syntactic sugar LeftJoin by the original set of rules. We have several cases:

- ans_P defines a Join
- ans_P defines an Union
- ans_P defines an LeftJoin (Optional operator)
- ans_P defined a Filter

We will prove the case where ans_P defines a LeftJoin. The rest of the case are analogous and simpler.

Since $ans_{LeftJoin(P_1,P_2)}$ is a Datalog translation of the fragment of a query of the form:

$$LeftJoin(P_1, P_2, E)$$

we can conclude Π_Q contains rules of the form shown in Figure B.18.

These rules intuitively represent: τ of the join (rule (1)), and τ of the difference of P_1, P_2 (rules 2 and 3). Recall that after the splitting process, the rules in (B.18) lose several literals, since literals of the form $A(\vec{t})$ (including negative ones) in U_k have been removed from the rules. For every atoms A removed in the successive splittings of Π_Q, we can use inductive hypotheses to conclude that:

$$\mathbf{M}_e \models A(\vec{t}) \leftrightarrow \mathbf{M} \models A(\vec{t}) \leftrightarrow \vec{t} \in [\![A(\vec{x})]\!]$$

1	$\mathtt{answer}_{\mathtt{LeftJoin(P_1,P_2)}}(\mathtt{V})$	:-	$\mathtt{answer}_{\mathtt{P_1}}(\mathtt{vars(P_1)}), \mathtt{answer}_{\mathtt{P_2}}(\mathtt{vars(P_2)}), \mathtt{E}(\mathtt{vars(P_1 \cup P_2)})$
2	$\mathtt{answer}_{\mathtt{LeftJoin(P_1,P_2)}}(\mathtt{V}[\mathtt{vars(P_2)} \setminus \mathtt{vars(P_1)} \mapsto \mathtt{null}])$	:-	$\mathtt{answer}_{\mathtt{P_1}}(\mathtt{vars(P_1)}), \mathtt{answer}_{\mathtt{P_2}}(\mathtt{vars(P_2)}), \mathbf{not}\ \mathtt{E}(\mathtt{vars(P_1 \cup P_2)})$
3	$\mathtt{answer}_{\mathtt{LeftJoin(P_1,P_2)}}(\mathtt{V}[\mathtt{vars(P_2)} \setminus \mathtt{vars(P_1)} \mapsto \mathtt{null}])$	:-	$\mathtt{answer}_{\mathtt{P_1}}(\mathtt{vars(P_1)}), \mathbf{not}\ \mathtt{answer}_{\mathtt{LJoin(P_1,P_2)}}(\mathtt{vars(P_1)})$
4	$\mathtt{answer}_{\mathtt{LJoin(P_1,P_2)}}(\mathtt{vars(P_1)})$	:-	$\mathtt{answer}_{\mathtt{P_1}}(\mathtt{vars(P_1)}), \mathtt{answer}_{\mathtt{P_2}}(\mathtt{vars(P_2)})$

Figure B.18: Translation SPARQL-Datalog first presented in [26] and extended in [27]

for positive atoms, and analogously

$$\mathbf{M}_e \models \mathbf{not}\, A(\vec{t}) \leftrightarrow \mathbf{M} \models \mathbf{not}\, A(\vec{t}) \leftrightarrow \vec{t} \notin [\![A(\vec{x})]\!]$$

- Let $body^1_{ans_{LeftJoin(P_1,P_2)}}$ be the body of the **not**-free rule (1) defining $answer_{LeftJoin(P_1,P_2)}$ above.
- Let $body^2_{ans_{LeftJoin(P_1,P_2)}}$ be the positive part of the body in rule (2) defining $answer_{LeftJoin(P_1,P_2)}$ above.
- Let $body^3_{ans_{LeftJoin(P_1,P_2)}}$ be the positive part of the body in rule (3) defining $answer_{LeftJoin(P_1,P_2)}$ above.
- Let $body^4_{ans_{LeftJoin(P_1,P_2)}}$ be the positive part of the body in rule (4) defining $answer_{LJoin(P_1,P_2)}$ above.

Thus from the previous facts, it follows that for every $\vec{c}$ satisfying rule (1) encoding the join part of the left join, there exist $\vec{c_1}\vec{c_2}$ such that

$$\mathbf{M}_e \models ans_{LeftJoin(P_1,P_2)}(\vec{c}) \text{ iff } \mathbf{M}_e \models body^1_{ans_{LeftJoin(P_1,P_2)}}(\vec{c_1}\vec{c_2})$$

Then, from the definition of translation we know that:
$\mathbf{M}_e \models ans_{LeftJoin(P_1,P_2)}(\vec{c})$
iff $\mathbf{M}_e \models body^1_{ans_{LeftJoin(P_1,P_2)}}(c_1\vec{c}_2)$
iff $\mathbf{M}_e \models Join(P_1(\vec{c_1}), P_2(\vec{c_2}), E(c_1 c_2))$
iff $c_1\vec{c}_2 \in [\![P_1(\vec{x_1})]\!] \bowtie_E [\![P_2(\vec{x_2})]\!]$
iff $c_1\vec{c}_2 \in [\![P_1(\vec{x_1})]\!] ⟕_E [\![P_2(\vec{x_2})]\!]$
iff $\vec{c} \in \Pi_{\vec{x}}([\![body^1_{ans_{LeftJoin(P_1,P_2)}}(x_1\vec{x}_2)]\!])$ iff
iff $\vec{c} \in [\![ans_{LeftJoin(P_1,P_2)}(\vec{x})]\!]$

And for every $\vec{c}$ satisfying either rule (2) encoding the difference part of the left join, there exist $\vec{c_1}$ such that
$\mathbf{M}_e \models ans_{LeftJoin(P_1,P_2)}(\vec{c})$
iff $\mathbf{M}_e \models body^2_{ans_{LeftJoin(P_1,P_2)}}(\vec{c_1})$ and $\vec{c} = \vec{c_1}\vec{null}$
iff $c_1 \in [\![P_1(\vec{x_1})]\!]$ and there is no c_2 such that coincides with $\vec{c_1}$ in the join positions and $\mathbf{M}_e \models ans_{P_1}(\vec{c_2}), ans_{P_2}(\vec{c_2})$ and therefore $\mathbf{M}_e \models \mathtt{answer}_{\mathtt{LJoin(P_1,P_2)}}(\mathtt{c_1})$. iff $\vec{c_1} \in [\![P_1(\vec{x_1})]\!] \setminus_E [\![P_2(\vec{x_2})]\!]]$
iff $\vec{c} \in \Pi_{\vec{x}}[\![P_1(\vec{x_1})]\!] ⟕_E [\![P_2(\vec{x_2})]\!]$
iff $\vec{c} \in [\![ans_{LeftJoin(P_1,P_2)}(\vec{x})]\!]$

And for every $\vec{c}$ satisfying either rule (3) encoding the difference part of the left join, there exist $\vec{c_1}$ such that
$\mathbf{M}_e \models ans_{LeftJoin(P_1,P_2)}(\vec{c})$
iff $\mathbf{M}_e \models body^3_{ans_{LeftJoin(P_1,P_2)}}(\vec{c_1})$ and $\vec{c} = \vec{c_1}\vec{null}$
iff $c_1 \in [\![P_1(\vec{x_1})]\!]$ and there is a c_2 such that coincides with $\vec{c_1}$ in the join positions and $c_2 \in [\![P_2(\vec{x_2})]\!]$ but $\mathbf{M}_e \not\models E(c_1, c_2)$
iff $\vec{c_1} \in [\![P_1(\vec{x_1})]\!] \setminus_E [\![P_2(\vec{x_2})]\!]]$
iff $\vec{c} \in \Pi_{\vec{x}}[\![P_1(\vec{x_1})]\!] ⟕_E [\![P_2(\vec{x_2})]\!]$
iff $\vec{c} \in [\![ans_{LeftJoin(P_1,P_2)}(\vec{x})]\!]$

This concludes the proof for the Left Join case, and the proof this theorem. □

Lemma 1. *Let M be a R2RML mapping. Let G be a RDF graph defined by M. Then*

$$(s, p, o) \in G \textit{ iff } \Pi_m \models triple(tr(s), tr(p), tr(o))$$

PROOF. Since the definition of $\rho(m)$ changes depending on o, we need to consider each case in turn. Suppose that o is an object which is not a class. The case where it is a class is analogous. By definition we know that $(s, p, o) \in G$ if and only if there is a mapping M with

1. a triple map node n;
2. a subject map node s hanging from m;
3. a property map node p and an object map node o hanging from the PropertyObjectMap of m;
4. and a logic table lt from where s, p, o are extracted.

From the previous facts it follows that Π_M contains:

$$\mathtt{triple}(tr(\mathtt{s}), tr(\mathtt{p}), tr(\mathtt{o})) \text{ :- } translated_logic_table$$

where *translated_logic_table* is the Datalog translation of lt. For the sake of simplicity assume that lt is a table T with columns $c_1 \dots c_n$. Then Π_m has the following rule:

$$\mathtt{triple}(tr(\mathtt{x_s}), tr(\mathtt{x_p}), tr(\mathtt{x_o})) \text{ :- } T(\vec{x})$$

where x_s, x_p, x_o correspond to the columns in T as specified in m. We know that there is a row in T where s, p, o are projected from it. It immediately follows that

$$\Pi_M \models triple(tr(s), tr(p), tr(o))$$

□

Appendix C. Datalog Normalization

Before performing the translation into Datalog, we need to deal with a number of issues that arise from the different nature of the formalisms at hand. For instance, Datalog is position-based (uses variables) while relational algebra and SQL are name-based (use column names). To cope with these issues while keeping the representation simple, we apply the following syntactic transformations to the program in this specific order:

- **Constants:** For every atom of the form $triple(t_1, t_2, t_3)$ in Π_Q where t_i is a constant symbol, add a new boolean atom of the form $v_i = t_i$ to the rule, and replace t_i by the fresh variable v_i in the *triple* atom. For instance:

  $$triple(x, a, \text{:Student})$$

  is replaced by

  $$triple(x, y, z),\ y = a,\ z = \text{:Student}$$

  This set of atoms will be denoted as fc, and the whole set is replaced by a single *Filter* atom. In our example above, this would be:

  $$Filter(triple(x, y, z), fc)$$

  where $fc = y = :a, z = :Student$.

- **Shared variables:** For every rule r, and every variable x such that the variable occurs in two different positions (in the same or different atoms in r), replace the variables with two new fresh variables x_1, x_2, and add them to the body of the query in the form $x_1 = x_2$, e.g., $\mathtt{ans}_1(\mathtt{x})$:- $\mathtt{ans}_2(\mathtt{x}, \mathtt{y}), \mathtt{ans}_3(\mathtt{x}, \mathtt{z})$ becomes:

  $\mathtt{ans}_1(\mathtt{x}_1)$:- $\mathtt{ans}_2(\mathtt{x}_1, \mathtt{y}), \mathtt{ans}_3(\mathtt{x}_2, \mathtt{z}), \mathtt{x}_1 = \mathtt{x}_2$

  These sets of join conditions together with any other boolean atom in the rule will be denoted jn. If there are several atoms in the body of r, the atoms will be renamed to a single *Join* atom, for instance: $\mathtt{Join}(\mathtt{ans}_2(x, y), \mathtt{ans}_3(x, z), jn)$.

- **Variable Names:** Recall that Π_Q can be seen as a tree where the root is $\mathtt{ans}_1(\vec{x})$ and the leaves are either `triple` atoms or boolean expressions. Then we:

  1. Enumerate the `triple` atoms in the leaves from left to right: $1 \ldots n$.
  2. For each of these `triple` leaves T, enumerated as j, and each variable x in the position $i = 1, 2, 3$ replace x by $T_j.s$ (if $i = 1$) or $T_j.p$ (if $i = 2$) or $T_j.o$ (if $i = 3$).
  3. Spread this change to the boolean expressions and the inner nodes of the tree.

In Figure C.19 we show the Datalog program from Example 5 after the transformation explained above.

Example 13. *Let Π_Q be the Datalog program presented in Example 5. Then $[\![ans_1(\vec{x})]\!]$ is as follows:*

$$\Pi_{as_1} ⟗_{\mathtt{1jn}} (\ \Pi_{as_2} \bowtie_{jn_1} (\Pi_{\mathtt{as_4}}\sigma_{\mathtt{fc_1}}(triple), \Pi_{\mathtt{as_5}}\sigma_{\mathtt{fc_2}}(triple)),$$
$$\Pi_{as_3} \bowtie_{jn_2} (\Pi_{\mathtt{as_6}}\sigma_{\mathtt{fc_3}}(triple),$$
$$\Pi_{as_7} \bowtie_{jn_3} (\Pi_{\mathtt{as_8}}\sigma_{\mathtt{fc_4}}(triple), \Pi_{\mathtt{as_9}}\sigma_{\mathtt{fc_5}}(triple))))$$

where

- $as_1 = [T_2.s, T_2.o, T_3.o, T_5.s, T_5.o]$
- $as_2 = [T_2.s, T_2.o]$
- $as_3 = [T_3.s, T_3.o, T_4.s, T_5.o]$
- as_4 = [*s AS* $T_1.s$, *p AS* $T_1.p$, *o AS* $T_1.o$]
- as_5 = [*s AS* $T_2.s$, *p AS* $T_2.p$, *o AS* $T_2.o$]
- as_6 = [*s AS* $T_3.s$, *p AS* $T_3.p$, *o AS* $T_3.o$]
- $as_7 = [T_4.s, T_5.o, T_4.o]$
- as_8 = [*s AS* $T_4.s$, *p AS* $T_4.p$, *o AS* $T_4.o$]
- as_9 = [*s AS* $T_5.s$, *p AS* $T_5.p$, *o AS* $T_5.o$]

- $ljn = [T_2.s = T_3.s]$
- $jn_1 = [T1.s = T2.s]$
- $jn_2 = [T3.o = T4.o]$
- $jn_3 = [T4.s = T5.s]$
- $fc_1 = [T_1.p = rdf{:}type, T_1.o = \textit{:Student}]$
- $fc_2 = [T_2.p = \textit{:hasName}]$
- $fc_3 = [T_3.p = \textit{:hasEnrolment}]$
- $fc_4 = [T_4.p = \textit{:hasYear}]$
- $fc_5 = [T_5.p = \textit{:hasGrade}]$

Observe that there is a very close relation between the ans predicates and the as_i statements, as well as Join, LeftJoin, and Filter conditions in both Datalog and SQL. □

$$
\begin{array}{lcl}
\mathtt{ans}_1(\mathtt{T}_2.\mathtt{s}, \mathtt{T}_2.\mathtt{o}, \mathtt{T}_3.\mathtt{o}, \mathtt{T}_5.\mathtt{s}, \mathtt{T}_5.\mathtt{o}) & \text{:-} & \mathtt{LeftJoin}(\mathtt{ans}_2(\mathtt{T}_2.\mathtt{s}, \mathtt{T}_2.\mathtt{o}), \\
 & & \mathtt{ans}_3(\mathtt{T}_3.\mathtt{s}, \mathtt{T}_3.\mathtt{o}, \mathtt{T}_4.\mathtt{s}, \mathtt{T}_5.\mathtt{o}), \mathtt{T}_2.\mathtt{s} = \mathtt{T}_3.\mathtt{s}) \\
\mathtt{ans}_2(\mathtt{T}_2.\mathtt{s}, \mathtt{T}_2.\mathtt{o}) & \text{:-} & \mathtt{Join}(\mathtt{ans}_4(\mathtt{T}_1.\mathtt{s}), \mathtt{ans}_5(\mathtt{T}_2.\mathtt{s}, \mathtt{T}_2.\mathtt{o}), \mathtt{T}_1.\mathtt{s} = \mathtt{T}_2.\mathtt{s}) \\
\mathtt{ans}_3(\mathtt{T}_3.\mathtt{s}, \mathtt{T}_3.\mathtt{o}, \mathtt{T}_4.\mathtt{s}, \mathtt{T}_5.\mathtt{o}) & \text{:-} & \mathtt{Join}(\mathtt{ans}_6(\mathtt{T}_3.\mathtt{s}, \mathtt{T}_3.\mathtt{o}), \mathtt{ans}_7(\mathtt{T}_4.\mathtt{s}, \mathtt{T}_5.\mathtt{o}, \mathtt{T}_4.\mathtt{o}), \mathtt{T}_3.\mathtt{o} = \mathtt{T}_4.\mathtt{o}) \\
\mathtt{ans}_4(\mathtt{T}_1.\mathtt{s}) & \text{:-} & \mathtt{Filter}(\mathtt{triple}(\mathtt{T}_1.\mathtt{s}, \mathtt{T}_1.\mathtt{p}, \mathtt{T}_1.\mathtt{o}), \mathtt{T}_1.\mathtt{p} = \mathtt{a}, \mathtt{T}_1.\mathtt{o} = \mathtt{Student}) \\
\mathtt{ans}_5(\mathtt{T}_2.\mathtt{s}, \mathtt{T}_2.\mathtt{o}) & \text{:-} & \mathtt{Filter}(\mathtt{triple}(\mathtt{T}_2.\mathtt{s}, \mathtt{T}_2.\mathtt{p}, \mathtt{T}_2.\mathtt{o}), \mathtt{T}_2.\mathtt{p} = \mathtt{hasNAme}) \\
\mathtt{ans}_6(\mathtt{T}_3.\mathtt{s}, \mathtt{T}_3.\mathtt{o}) & \text{:-} & \mathtt{Filter}(\mathtt{triple}(\mathtt{T}_3.\mathtt{s}, \mathtt{T}_3.\mathtt{p}, \mathtt{T}_3.\mathtt{o}), \mathtt{T}_3.\mathtt{p} = \mathtt{hasEnrolment}) \\
\mathtt{ans}_7(\mathtt{T}_4.\mathtt{s}, \mathtt{T}_5.\mathtt{o}, \mathtt{T}_4.\mathtt{o}) & \text{:-} & \mathtt{Join}(\mathtt{ans}_{14}(\mathtt{T}_4.\mathtt{s}, \mathtt{T}_4.\mathtt{o}), \mathtt{ans}_{15}(\mathtt{T}_5.\mathtt{s}, \mathtt{T}_5.\mathtt{o}), \mathtt{T}_4.\mathtt{s} = \mathtt{T}_5.\mathtt{s}) \\
\mathtt{ans}_{14}(\mathtt{T}_4.\mathtt{s}, \mathtt{T}_4.\mathtt{o}) & \text{:-} & \mathtt{Filter}(\mathtt{triple}(\mathtt{T}_4.\mathtt{s}, \mathtt{T}_4.\mathtt{p}, \mathtt{T}_4.\mathtt{o}), \mathtt{T}_4.\mathtt{p} = \mathtt{hasYear}) \\
\mathtt{ans}_{15}(\mathtt{T}_5.\mathtt{s}, \mathtt{T}_5.\mathtt{o}) & \text{:-} & \mathtt{Filter}(\mathtt{triple}(\mathtt{T}_5.\mathtt{s}, \mathtt{T}_5.\mathtt{p}, \mathtt{T}_5.\mathtt{o}), \mathtt{T}_5.\mathtt{p} = \mathtt{hasGrade}) \quad \square
\end{array}
$$

Figure C.19: Modified Π_Q.

Appendix B

Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime

This appendix reports the paper:

> Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyaschev: Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime. In *Proc. of the 13th Int. Semantic Web Conference* (ISWC), 2014.

Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime

Roman Kontchakov[1], Martin Rezk[2], Mariano Rodríguez-Muro[3], Guohui Xiao[2], and Michael Zakharyaschev[1]

[1] Department of Computer Science and Information Systems, Birkbeck, University of London, U.K.
[2] Faculty of Computer Science, Free University of Bozen-Bolzano, Italy
[3] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

Abstract. We present an extension of the ontology-based data access platform *Ontop* that supports answering SPARQL queries under the OWL 2 QL direct semantics entailment regime for data instances stored in relational databases. On the theoretical side, we show how any input SPARQL query, OWL 2 QL ontology and R2RML mappings can be rewritten to an equivalent SQL query solely over the data. On the practical side, we present initial experimental results demonstrating that by applying the *Ontop* technologies—the tree-witness query rewriting, $\mathcal{T}$-mappings compiling R2RML mappings with ontology hierarchies, and $\mathcal{T}$-mapping optimisations using SQL expressivity and database integrity constraints—the system produces scalable SQL queries.

1 Introduction

Ontology-based data access and management (OBDA) is a popular paradigm of organising access to various types of data sources that has been developed since the mid 2000s [11,17,24]. In a nutshell, OBDA separates the user from the data sources (relational databases, triple stores, etc.) by means of an ontology which provides the user with a convenient query vocabulary, hides the structure of the data sources, and can enrich incomplete data with background knowledge. About a dozen OBDA systems have been implemented in both academia and industry; e.g., [27,30,24,4,23,15,12,8,20,22]. Most of them support conjunctive queries and the OWL 2 QL profile of OWL 2 as the ontology language (or its generalisations to existential datalog rules). Thus, the OBDA platform *Ontop* [29] was designed to query data instances stored in relational databases, with the vocabularies of the data and OWL 2 QL ontologies linked by means of global-as-view (GAV) mappings. Given a conjunctive query in the vocabulary of such an ontology, *Ontop* rewrites it to an SQL query in the vocabulary of the data, optimises the rewriting and delegates its evaluation to the database system.

One of the main aims behind the newly designed query language SPARQL 1.1—a W3C recommendation since 2013—has been to support various entailment regimes, which can be regarded as variants of OBDA. Thus, the OWL 2 direct semantics entailment regime allows SPARQL queries over OWL 2 DL ontologies and RDF graphs (which can be thought of as 3-column database tables). SPARQL queries are in many aspects more expressive than conjunctive queries as they offer more complex query

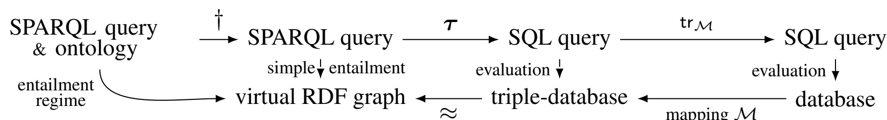
constructs and can retrieve not only domain elements but also class and property names using second-order variables. (Note, however, that SPARQL 1.1 does not cover all conjunctive queries.) OWL 2 DL is also vastly superior to OWL 2 QL, but this makes query answering under the OWL 2 direct semantics entailment regime intractable (CONP-hard for data complexity). For example, the query evaluation algorithm of [19] calls an OWL 2 DL reasoner for each possible assignment to the variables in a given query, and therefore cannot cope with large data instances.

In this paper, we investigate answering SPARQL queries under a less expressive entailment regime, which corresponds to OWL 2 QL, assuming that data is stored in relational databases. It is to be noted that the W3C specification[1] of SPARQL 1.1 defines entailment regimes for the profiles of OWL 2 by restricting the general definition to the profile constructs that can be used in the queries. However, in the case of OWL 2 QL, this generic approach leads to a sub-optimal, almost trivial query language, which is essentially subsumed by the OWL 2 RL entailment regime.

The first aim of this paper is to give an optimal definition of the OWL 2 QL direct semantics entailment regime and prove that—similarly to OBDA with OWL 2 QL and conjunctive queries—answering SPARQL queries under this regime is reducible to answering queries under *simple entailment*. More precisely, in Theorem 4 we construct a rewriting $\cdot^\dagger$ of any given SPARQL query and ontology under the OWL 2 QL entailment regime to a SPARQL query that can be evaluated on any dataset directly.

In a typical *Ontop* scenario, data is stored in a relational database whose schema is linked to the vocabulary of the given OWL 2 QL ontology via a GAV mapping in the language R2RML. The mapping allows one to transform the relational data instance into an RDF representation, called the virtual RDF graph (which is not materialised in our scenario). The rewriting $\cdot^\dagger$ constructs a SPARQL query over this virtual graph.

Our second aim is to show how such a SPARQL query can be translated to an equivalent SQL query over a relational representation of the virtual RDF graph as a 3-column table (translation $\boldsymbol{\tau}$ in Theorem 7). The third aim is to show that the resulting SQL query can be unfolded, using a given R2RML mapping $\mathcal{M}$, to an SQL query over the original database ($\mathsf{tr}_\mathcal{M}$ in Theorem 12), which is evaluated by the database system.



Unfortunately, each of these three transformations may involve an exponential blowup. We tackle this problem in *Ontop* using the following optimisation techniques. (*i*) The mapping is compiled with the ontology into a $\mathcal{T}$-mapping [29] and optimised by database dependencies (e.g., primary, candidate and foreign keys) and SQL disjunctions. (*ii*) The SPARQL-to-SQL translation is optimised using null join elimination (Theorem 8). (*iii*) The unfolding is optimised by eliminating joins with mismatching R2RML IRI templates, de-IRIing the join conditions (Section 3.3) and using database dependencies.

Our contributions (Theorems 4, 7, 8 and 12 and optimisations in Section 3.3) make *Ontop* the first system to support the W3C recommendations OWL 2 QL, R2RML, SPARQL and the OWL 2 QL direct semantics entailment regime; its architecture is out-

[1] `http://www.w3.org/TR/sparql11-entailment`

lined in Section 4. We evaluate the performance of *Ontop* using the LUBM Benchmark [16] extended with queries containing class and property variables, and compare it with two other systems that support the OWL 2 entailment regime by calling OWL DL reasoners (Section 5). Our experiments show that *Ontop* outperforms the reasoner-based systems for most of the queries over small datasets; over larger datasets the difference becomes dramatic, with *Ontop* demonstrating a solid performance even on 69 million triples in LUBM$_{500}$. Finally, we note that, although Ontop was designed to work with existing relational databases, it is also applicable in the context of RDF triple stores, in which case approaches such as the one from [3] can be used to generate suitable relational schemas. Omitted proofs and evaluation details can be found in the full version at `http://www.dcs.bbk.ac.uk/~michael/ISWC-14-v2.pdf`.

2 SPARQL Queries under OWL 2 QL Entailment Regime

SPARQL is a W3C standard language designed to query RDF graphs. Its vocabulary contains four pairwise disjoint and countably infinite sets of symbols: I for *IRIs*, B for *blank nodes*, L for *RDF literals*, and V for *variables*. The elements of $\mathsf{C} = \mathsf{I} \cup \mathsf{B} \cup \mathsf{L}$ are called *RDF terms*. A *triple pattern* is an element of $(\mathsf{C} \cup \mathsf{V}) \times (\mathsf{I} \cup \mathsf{V}) \times (\mathsf{C} \cup \mathsf{V})$. A *basic graph pattern* (*BGP*) is a finite set of triple patterns. Finally, a *graph pattern*, P, is an expression defined by the grammar

$$P \ ::= \ \mathrm{BGP} \mid \textsc{Filter}(P, F) \mid \textsc{Bind}(P, v, c) \mid \textsc{Union}(P_1, P_2) \mid \textsc{Join}(P_1, P_2) \mid \textsc{Opt}(P_1, P_2, F),$$

where F, a *filter*, is a formula constructed from atoms of the form *bound*(v), $(v = c)$, $(v = v')$, for $v, v' \in \mathsf{V}$, $c \in \mathsf{C}$, and possibly other built-in predicates using the logical connectives $\land$ and $\neg$. The set of variables in P is denoted by *var*(P).

A *SPARQL query* is a graph pattern P with a *solution modifier*, which specifies the *answer variables*—the variables in P whose values we are interested in—and the form of the output (we ignore other solution modifiers for simplicity). The values to variables are given by *solution mappings*, which are *partial* maps $s\colon \mathsf{V} \to \mathsf{C}$ with (possibly empty) domain *dom*(s). In this paper, we use the set-based (rather than bag-based, as in the specification) semantics for SPARQL. For sets S_1 and S_2 of solution mappings, a filter F, a variable $v \in \mathsf{V}$ and a term $c \in \mathsf{C}$, let

- $\textsc{Filter}(S, F) = \{s \in S \mid F^s = \top\}$;
- $\textsc{Bind}(S, v, c) = \{s \oplus \{v \mapsto c\} \mid s \in S\}$ (provided that $v \notin dom(s)$, for $s \in S$);
- $\textsc{Union}(S_1, S_2) = \{s \mid s \in S_1 \text{ or } s \in S_2\}$;
- $\textsc{Join}(S_1, S_2) = \{s_1 \oplus s_2 \mid s_1 \in S_1 \text{ and } s_2 \in S_2 \text{ are compatible}\}$;
- $\textsc{Opt}(S_1, S_2, F) = \textsc{Filter}(\textsc{Join}(S_1, S_2), F) \ \cup \ \{s_1 \in S_1 \mid \text{ for all } s_2 \in S_2,$ either s_1, s_2 are incompatible or $F^{s_1 \oplus s_2} \neq \top\}$.

Here, s_1 and s_2 are *compatible* if $s_1(v) = s_2(v)$, for any $v \in dom(s_1) \cap dom(s_2)$, in which case $s_1 \oplus s_2$ is a solution mapping with $s_1 \oplus s_2 \colon v \mapsto s_1(v)$, for $v \in dom(s_1)$, $s_1 \oplus s_2 \colon v \mapsto s_2(v)$, for $v \in dom(s_2)$, and domain $dom(s_1) \cup dom(s_2)$. The *truth-value* $F^s \in \{\top, \bot, \varepsilon\}$ *of a filter F under a solution mapping s* is defined inductively:

– $(\mathit{bound}(v))^s$ is $\top$ if $v \in \mathit{dom}(s)$ and $\bot$ otherwise;
– $(v = c)^s = \varepsilon$ if $v \notin \mathit{dom}(s)$; otherwise, $(v = c)^s$ is the classical truth-value of the predicate $s(v) = c$; similarly, $(v = v')^s = \varepsilon$ if either v or $v' \notin \mathit{dom}(s)$; otherwise, $(v = v')^s$ is the classical truth-value of the predicate $s(v) = s(v')$;
– $(\neg F)^s = \begin{cases} \varepsilon, & \text{if } F^s = \varepsilon, \\ \neg F^s, & \text{otherwise,} \end{cases}$ and $(F_1 \land F_2)^s = \begin{cases} \bot, & \text{if } F_1^s = \bot \text{ or } F_2^s = \bot, \\ \top, & \text{if } F_1^s = F_2^s = \top, \\ \varepsilon, & \text{otherwise.} \end{cases}$

Finally, given an RDF graph G, the *answer to a graph pattern P over G* is the set $[\![P]\!]_G$ of solution mappings defined by induction using the operations above and starting from the following base case: for a basic graph pattern B,

$$[\![B]\!]_G = \{s \colon \mathit{var}(B) \to \mathsf{C} \mid s(B) \subseteq G\}, \tag{1}$$

where $s(B)$ is the set of triples resulting from substituting each variable u in B by $s(u)$. This semantics is known as *simple entailment*.

Remark 1. The condition '$F^{s_1 \oplus s_2}$ *is not true*' in the definition of OPT is different from '$F^{s_1 \oplus s_2}$ has an effective Boolean value of false' given by the W3C specification:[2] the effective Boolean value can be undefined (type error) if a variable in F is not bound by $s_1 \oplus s_2$. As we shall see in Section 3.1, our reading corresponds to LEFT JOIN in SQL. (Note also that the informal explanation of OPT in the W3C specification is inconsistent with the definition of DIFF; see the full version for details.)

Under the *OWL 2 QL direct semantics entailment regime*, one can query an RDF graph G that consist of two parts: an *extensional* sub-graph $\mathcal{A}$ representing the *data* as OWL 2 QL class and property assertions, and the *intensional* sub-graph $\mathcal{T}$ representing the background *knowledge* as OWL 2 QL class and property axioms. We write $(\mathcal{T}, \mathcal{A})$ in place of G to emphasise the partitioning. To illustrate, we give a simple example.

Example 2. Consider the following two axioms from the LUBM ontology $(\mathcal{T}, \mathcal{A})$ (see Section 5), which are given here in the functional-style syntax (FSS):

SubClassOf(ub:UGStudent, ub:Student), SubClassOf(ub:GradStudent, ub:Student).

Under the entailment regime, we can write a query that retrieves all named *subclasses* of students in $(\mathcal{T}, \mathcal{A})$ and all *instances* of each of these subclasses (cf. q_9' in Section 5):

SELECT $?x$ $?C$ WHERE { $?C$ rdfs:subClassOf ub:Student. $?x$ rdf:type $?C$. }.

Here $?C$ ranges over the class names (IRIs) in $(\mathcal{T}, \mathcal{A})$ and $?x$ over the IRIs of individuals. If, for example, $\mathcal{A}$ consists of the two assertions on the left-hand side, then the answer to the query over $(\mathcal{T}, \mathcal{A})$ is on the right-hand side:

$\mathcal{A}$
ClassAssertion(ub:UGStudent, ub:jim)
ClassAssertion(ub:Student, ub:bob)

$?x$	$?C$
ub:jim	ub:UGStudent
ub:jim	ub:Student
ub:bob	ub:Student

[2] http://www.w3.org/TR/sparql11-query/#sparqlAlgebra

To formally define SPARQL queries that can be used under the OWL 2 QL direct semantics entailment regime, we assume that the set I of IRIs is partitioned into disjoint and countably infinite sets of *class names* I_C, *object property names* I_R and *individual names* I_I. Similarly, the variables V are also assumed to be a disjoint union of countably infinite sets V_C, V_R, V_I. Now, we define an *OWL 2 QL BGP* as a finite set of triple patterns representing OWL 2 QL axiom and assertion templates in the FSS such as:[3]

$$\begin{array}{ll}
\mathsf{SubClassOf}(\mathit{SubC}, \mathit{SuperC}), & \mathsf{DisjointClasses}(\mathit{SubC}_1, \dots, \mathit{SubC}_n),\\
\mathsf{ObjectPropertyDomain}(\mathit{OP}, \mathit{SuperC}), & \mathsf{ObjectPropertyRange}(\mathit{OP}, \mathit{SuperC}),\\
\mathsf{SubObjectPropertyOf}(\mathit{OP}, \mathit{OP}), & \mathsf{DisjointObjectProperties}(\mathit{OP}_1, \dots, \mathit{OP}_n),\\
\mathsf{ClassAssertion}(\mathit{SuperC}, I), & \mathsf{ObjectPropertyAssertion}(\mathit{OP}, I, I),
\end{array}$$

where $I \in \mathsf{I}_I \cup \mathsf{V}_I$ and *OP*, *SubC* and *SuperC* are defined by the following grammar with $C \in \mathsf{I}_C \cup \mathsf{V}_C$ and $R \in \mathsf{I}_R \cup \mathsf{V}_R$:

$$\begin{array}{rcl}
\mathit{OP} & ::= & R \mid \mathsf{ObjectInverseOf}(R),\\
\mathit{SubC} & ::= & C \mid \mathsf{ObjectSomeValuesFrom}(\mathit{OP}, \mathsf{owl{:}Thing}),\\
\mathit{SuperC} & ::= & C \mid \mathsf{ObjectIntersectionOf}(\mathit{SuperC}_1, \dots, \mathit{SuperC}_n) \mid\\
& & \qquad \mathsf{ObjectSomeValuesFrom}(\mathit{OP}, \mathit{SuperC}).
\end{array}$$

OWL 2 QL graph patterns are constructed from OWL 2 QL BGPs using the SPARQL operators. Finally, an *OWL 2 QL query* is a pair (P, V), where P is an OWL 2 QL graph pattern and $V \subseteq \mathit{var}(P)$. To define the answer to such a query (P, V) over an RDF graph $(\mathcal{T}, \mathcal{A})$, we fix a *finite* vocabulary $\mathsf{I}_{\mathcal{T},\mathcal{A}} \subseteq \mathsf{I}$ that includes all names (IRIs) in $\mathcal{T}$ and $\mathcal{A}$ as well as the required finite part of the OWL 2 RDF-based vocabulary (e.g., owl:Thing but not the infinite number of the rdf:_n). To ensure finiteness of the answers and proper typing of variables, in the following definition we only consider solution mappings $s\colon \mathit{var}(P) \to \mathsf{I}_{\mathcal{T},\mathcal{A}}$ such that $s^{-1}(\mathsf{I}_\alpha) \subseteq \mathsf{V}_\alpha$, for $\alpha \in \{C, R, I\}$. For each BGP B, we define the *answer* $[\![B]\!]_{\mathcal{T},\mathcal{A}}$ *to* B *over* $(\mathcal{T}, \mathcal{A})$ by taking

$$[\![B]\!]_{\mathcal{T},\mathcal{A}} = \{s\colon \mathit{var}(B) \to \mathsf{I}_{\mathcal{T},\mathcal{A}} \mid (\mathcal{T}, \mathcal{A}) \models s(B)\},$$

where $\models$ is the entailment relation given by the OWL 2 direct semantics. Starting from the $[\![B]\!]_{\mathcal{T},\mathcal{A}}$ and applying the SPARQL operators in P, we compute the set $[\![P]\!]_{\mathcal{T},\mathcal{A}}$ of *solution mappings*. The *answer to* (P, V) *over* $(\mathcal{T}, \mathcal{A})$ is the restriction $[\![P]\!]_{\mathcal{T},\mathcal{A}}|_V$ of the solution mappings in $[\![P]\!]_{\mathcal{T},\mathcal{A}}$ to the variables in V.

Example 3. Suppose $\mathcal{T}$ contains

$$\begin{array}{l}
\mathsf{SubClassOf}(\mathsf{:A}, \mathsf{ObjectSomeValuesFrom}(\mathsf{:P}, \mathsf{owl{:}Thing})),\\
\mathsf{SubObjectPropertyOf}(\mathsf{:P}, \mathsf{:R}), \quad \mathsf{SubObjectPropertyOf}(\mathsf{:P}, \mathsf{ObjectInverseOf}(\mathsf{:S})).
\end{array}$$

Consider the following OWL 2 QL BGP B:

$$\begin{array}{l}
\mathsf{ClassAssertion}(\mathsf{ObjectSomeValuesFrom}(\mathsf{:R}, \mathsf{ObjectSomeValuesFrom}(\mathsf{:S},\\
\qquad\qquad \mathsf{ObjectSomeValuesFrom}(\mathsf{:T}, \mathsf{owl{:}Thing}))), ?x).
\end{array}$$

[3] The official specification of legal queries under the OWL 2 QL entailment regime only allows ClassAssertion(C, I) rather than ClassAssertion(SuperC, I), which makes the OWL 2 QL entailment regime trivial and essentially subsumed by the OWL 2 RL entailment regime.

Assuming that $\mathcal{A} = \{\mathsf{ClassAssertion}(\mathsf{:A}, \mathsf{:a}), \mathsf{ObjectPropertyAssertion}(\mathsf{:T}, \mathsf{:a}, \mathsf{:b})\}$, it is not hard to see that $[\![B]\!]_{\mathcal{T},\mathcal{A}} = \{?x \mapsto \mathsf{:a}\}$. Indeed, by the first assertion of $\mathcal{A}$ and the first two axioms of $\mathcal{T}$, any model of $(\mathcal{T}, \mathcal{A})$ contains a domain element w (not necessarily among the individuals in $\mathcal{A}$) such that $\mathsf{ObjectPropertyAssertion}(\mathsf{:R}, \mathsf{:a}, w)$ holds. In addition, the third axiom of $\mathcal{T}$ implies $\mathsf{ObjectPropertyAssertion}(\mathsf{:S}, w, \mathsf{:a})$, which together with the second assertion of $\mathcal{A}$ mean that $\{?x \mapsto \mathsf{:a}\}$ is an answer.

The following theorem shows that answering OWL 2 QL queries under the direct semantics entailment regime can be reduced to answering OWL 2 QL queries under simple entailment, which are evaluated only on the extensional part of the RDF graph:

Theorem 4. *Given any intensional graph $\mathcal{T}$ and OWL 2 QL query (P, V), one can construct an OWL 2 QL query $(P^\dagger, V)$ such that, for any extensional graph $\mathcal{A}$ (in some fixed finite vocabulary), $[\![P]\!]_{\mathcal{T},\mathcal{A}}|_V = [\![P^\dagger]\!]_{\mathcal{A}}|_V$.*

Proof sketch. By the definition of the entailment regime, it suffices to construct $B^\dagger$, for any *BGP* B; the rewriting $P^\dagger$ is obtained then by replacing each BGP B in P with $B^\dagger$. First, we instantiate the class and property variables in B by all possible class and property names in the given vocabulary and add the respective BIND operations. In each of the resulting BGPs, we remove the class and property axioms if they are entailed by $\mathcal{T}$; otherwise we replace the BGP with an empty one. The obtained BGPs are (SPARQL representations of) conjunctive queries (with non-distinguished variables in complex concepts *SuperC* of the assertions $\mathsf{ClassAssertion}(\mathit{SuperC}, \mathit{I})$). The second step is to rewrite these conjunctive queries together with $\mathcal{T}$ into unions of conjunctive queries (BGPs) that can be evaluated over any extensional graph $\mathcal{A}$ [5,21]. (We emphasise that the SPARQL algebra operations, including difference and OPT, are applied to BGPs and do not interact with the two steps of our rewriting.) ❑

We illustrate the proof of Theorem 4 using the queries from Examples 2 and 3.

Example 5. The class variable $?C$ in the query from Example 2 can be instantiated, using BIND, by all possible values from $\mathsf{I}_C \cap \mathsf{I}_{\mathcal{T},\mathcal{A}}$, which gives the rewriting

```
SELECT ?x ?C WHERE {
    { ?x rdf:type ub:Student. BIND(ub:Student as ?C) } UNION
    { ?x rdf:type ub:GradStudent. BIND(ub:GradStudent as ?C) } UNION
    { ?x rdf:type ub:UGStudent. BIND(ub:UGStudent as ?C) } }.
```

The query from Example 3 is equivalent to a (tree-shaped) conjunctive query with three non-distinguished and one answer variable, which can be rewritten to

```
SELECT ?x WHERE { { ?x :R ?y. ?y :S ?z. ?z :T ?u. } UNION
                  { ?x rdf:type :A. ?x :T ?u. } }.
```

3 Translating SPARQL under Simple Entailment to SQL

A number of translations of SPARQL queries (under simple entailment) to SQL queries have already been suggested in the literature; see, e.g., [9,13,7,32,27]. However, none

of them is suitable for our aims because they do not take into account the three-valued logic used in the OPTIONAL and BOUND constructs of the current SPARQL 1.1 (the semantics of OPTIONAL was not compositional in SPARQL 1.0). Note also that SPARQL has been translated to Datalog [25,2,26].

We begin by recapping the basics of relational algebra and SQL (see e.g., [1]). Let U be a finite (possibly empty) set of *attributes*. A *tuple over* U is a map $t\colon U \to \Delta$, where Δ is the underlying domain, which always contains a distinguished element *null*. A ($|U|$*-ary*) *relation over* U is a finite set of tuples over U (again, we use the set-based rather than bag-based semantics). A *filter* F *over* U is a formula constructed from atoms *isNull*(U'), $(u = c)$ and $(u = u')$, where $U' \subseteq U$, $u, u' \in U$ and $c \in \Delta$, using the connectives $\land$ and $\neg$. Let F be a filter with variables U and let t be a tuple over U. The *truth-value* $F^t \in \{\top, \bot, \varepsilon\}$ *of* F *over* t is defined inductively:

- $(\mathit{isNull}(U'))^t$ is $\top$ if $t(u)$ is *null*, for all $u \in U'$, and $\bot$ otherwise;
- $(u = c)^t = \varepsilon$ if $t(u)$ is *null*; otherwise, $(u = c)^t$ is the classical truth-value of the predicate $t(u) = c$; similarly, $(u = u')^t = \varepsilon$ if either $t(u)$ or $t(u')$ is *null*; otherwise, $(u = u')^t$ is the classical truth-value of the predicate $t(u) = t(u')$;
- $(\neg F)^t = \begin{cases} \varepsilon, & \text{if } F^t = \varepsilon, \\ \neg F^t, & \text{otherwise,} \end{cases}$ and $(F_1 \land F_2)^t = \begin{cases} \bot, & \text{if } F_1^t = \bot \text{ or } F_2^t = \bot, \\ \top, & \text{if } F_1^t = F_2^t = \top, \\ \varepsilon, & \text{otherwise.} \end{cases}$

(Note that $\neg$ and $\land$ are interpreted in the same three-valued logic as in SPARQL.) We use standard relational algebra operations such as union, difference, projection, selection, renaming and natural (inner) join. Let R_i be a relation over U_i, $i = 1, 2$.

- If $U_1 = U_2$ then the standard $R_1 \cup R_2$ and $R_1 \setminus R_2$ are relations over U_1.
- If $U \subseteq U_1$ then $\pi_U R_1 = R_1|_U$ is a relation over U.
- If F is a filter over U_1 then $\sigma_F R_1 = \{t \in R_1 \mid F^t = \top\}$ is a relation over U_1.
- If $v \notin U_1$ and $u \in U_1$ then $\rho_{v/u} R_1 = \{t_{v/u} \mid t \in R_1\}$, where $t_{v/u}\colon v \mapsto t(u)$ and $t_{v/u}\colon u' \mapsto t(u')$, for $u' \in U_1 \setminus \{u\}$, is a relation over $(U_1 \setminus \{u\}) \cup \{v\}$.
- $R_1 \bowtie R_2 = \{t_1 \oplus t_2 \mid t_1 \in R_1 \text{ and } t_2 \in R_2 \text{ are compatible}\}$ is a relation over $U_1 \cup U_2$. Here, t_1 and t_2 are *compatible* if $t_1(u) = t_2(u) \neq \mathit{null}$, for all $u \in U_1 \cap U_2$, in which case a tuple $t_1 \oplus t_2$ over $U_1 \cup U_2$ is defined by taking $t_1 \oplus t_2\colon u \mapsto t_1(u)$, for $u \in U_1$, and $t_1 \oplus t_2\colon u \mapsto t_2(u)$, for $u \in U_2$ (note that if u is *null* in either of the tuples then they are incompatible).

To bridge the gap between partial functions (solution mappings) in SPARQL and total mappings (on attributes) in SQL, we require one more operation (expressible in SQL):

- If $U \cap U_1 = \emptyset$ then the *padding* $\mu_U R_1$ is $R_1 \bowtie \mathit{null}^U$, where null^U is the relation consisting of a single tuple t over U with $t\colon u \mapsto \mathit{null}$, for all $u \in U$.

By an *SQL query*, Q, we understand any expression constructed from relation symbols (each over a fixed set of attributes) and filters using the relational algebra operations given above (and complying with all restrictions on the structure). Suppose Q is an SQL query and D a data instance which, for any relation symbol in the schema under consideration, gives a concrete relation over the corresponding set of attributes. The

answer to Q over D is a relation $\|Q\|_D$ defined inductively in the obvious way starting from the base case: for a relation symbol Q, $\|Q\|_D$ is the corresponding relation in D.

We now define a translation, $\boldsymbol{\tau}$, which, given a graph pattern P, returns an SQL query $\boldsymbol{\tau}(P)$ with the same answers as P. More formally, for a set of variables V, let ext_V be a function transforming any solution mapping s with $\mathit{dom}(s) \subseteq V$ to a tuple over V by padding it with *null*s:

$$\mathit{ext}_V(s) \;=\; \{v \mapsto s(v) \mid v \in \mathit{dom}(s)\} \;\cup\; \{v \mapsto \mathit{null} \mid v \in V \setminus \mathit{dom}(s)\}.$$

The *relational answer to P over G* is $\|P\|_G = \{\mathit{ext}_{\mathit{var}(P)}(s) \mid s \in [\![P]\!]_G\}$. The SQL query $\boldsymbol{\tau}(P)$ will be such that, for any RDF graph G, the relational answer to P over G coincides with the answer to $\boldsymbol{\tau}(P)$ over $\mathit{triple}(G)$, the database instance storing G as a ternary relation *triple* with the attributes *subj*, *pred*, *obj*. First, we define the translation of a SPARQL filter F by taking $\boldsymbol{\tau}(F)$ to be the SQL filter obtained by replacing each $\mathit{bound}(v)$ with $\neg\mathit{isNull}(v)$ (other built-in predicates can be handled similarly).

Proposition 6. *Let F be a SPARQL filter and let V be the set of variables in F. Then $F^s = (\boldsymbol{\tau}(F))^{\mathit{ext}_V(s)}$, for any solution mapping s with $\mathit{dom}(s) \subseteq V$.*

The definition of $\boldsymbol{\tau}$ proceeds by induction on the construction of P. Note that we can always assume that graph patterns *under simple entailment* do not contain blank nodes because they can be replaced by fresh variables. It follows that a BGP $\{\mathit{tp}_1, \dots, \mathit{tp}_n\}$ is equivalent to $\textsc{Join}(\{\mathit{tp}_1\}, \textsc{Join}(\{\mathit{tp}_2\}, \dots))$. So, for the basis of induction we set

$$\boldsymbol{\tau}(\{\langle s,p,o\rangle\}) = \begin{cases} \pi_\emptyset \sigma_{(\mathit{subj}=s)\wedge(\mathit{pred}=p)\wedge(\mathit{obj}=o)}\, \mathit{triple}, & \text{if } s,p,o \in \mathsf{I} \cup \mathsf{L},\\ \pi_s \rho_{s/\mathit{subj}}\, \sigma_{(\mathit{pred}=p)\wedge(\mathit{obj}=o)}\, \mathit{triple}, & \text{if } s \in \mathsf{V} \text{ and } p,o \in \mathsf{I} \cup \mathsf{L},\\ \pi_{s,o} \rho_{s/\mathit{subj}}\, \rho_{o/\mathit{obj}}\, \sigma_{\mathit{pred}=p}\, \mathit{triple}, & \text{if } s,o \in \mathsf{V}, s \neq o, p \in \mathsf{I} \cup \mathsf{L},\\ \pi_s \rho_{s/\mathit{subj}}\, \sigma_{(\mathit{pred}=p)\wedge(\mathit{subj}=\mathit{obj})}\, \mathit{triple}, & \text{if } s,o \in \mathsf{V}, s = o, p \in \mathsf{I} \cup \mathsf{L},\\ \dots \end{cases}$$

(the remaining cases are similar). Now, if P_1 and P_2 are graph patterns and F_1 and F are filters containing only variables in $\mathit{var}(P_1)$ and $\mathit{var}(P_1) \cup \mathit{var}(P_2)$, respectively, then we set $U_i = \mathit{var}(P_i)$, $i = 1, 2$, and

$$\begin{aligned}
&\boldsymbol{\tau}(\textsc{Filter}(P_1, F_1)) = \sigma_{\boldsymbol{\tau}(F_1)} \boldsymbol{\tau}(P_1),\\
&\boldsymbol{\tau}(\textsc{Bind}(P_1, v, c)) = \boldsymbol{\tau}(P_1) \bowtie \{v \mapsto c\},\\
&\boldsymbol{\tau}(\textsc{Union}(P_1, P_2)) = \; \mu_{U_2 \setminus U_1} \boldsymbol{\tau}(P_1) \;\cup\; \mu_{U_1 \setminus U_2} \boldsymbol{\tau}(P_2),\\
&\boldsymbol{\tau}(\textsc{Join}(P_1, P_2)) = \bigcup_{\substack{V_1, V_2 \subseteq U_1 \cap U_2\\ V_1 \cap V_2 = \emptyset}} \mu_{V_1 \cup V_2} \big[(\pi_{U_1 \setminus V_1} \sigma_{\mathit{isNull}(V_1)} \boldsymbol{\tau}(P_1)) \bowtie (\pi_{U_2 \setminus V_2} \sigma_{\mathit{isNull}(V_2)} \boldsymbol{\tau}(P_2))\big],\\
&\boldsymbol{\tau}(\textsc{Opt}(P_1, P_2, F)) = \; \sigma_{\boldsymbol{\tau}(F)}(\boldsymbol{\tau}(\textsc{Join}(P_1, P_2))) \;\cup\\
&\qquad\qquad \mu_{U_2 \setminus U_1} \big(\boldsymbol{\tau}(P_1) \setminus \pi_{U_1} \sigma_{\boldsymbol{\tau}(F)}(\boldsymbol{\tau}(\textsc{Join}(P_1, P_2)))\big).
\end{aligned}$$

It is readily seen that any $\boldsymbol{\tau}(P)$ is a valid SQL query and defines a relation over $\mathit{var}(P)$.

Theorem 7. *For any RDF graph G and any graph pattern P, $\|P\|_G = \|\boldsymbol{\tau}(P)\|_{\mathit{triple}(G)}$.*

Proof. The proof is by induction on the structure of P. Here we only consider the induction step for $P = \text{JOIN}(P_1, P_2)$. Let $U_i = \mathit{var}(P_i)$, $i = 1, 2$, and $U = U_1 \cap U_2$.

If $t \in \|\text{JOIN}(P_1, P_2)\|_G$ then there is a solution mapping $s \in [\![\text{JOIN}(P_1, P_2)]\!]_G$ with $\mathit{ext}_{U_1 \cup U_2}(s) = t$, and so there are $s_i \in [\![P_i]\!]_G$ such that s_1 and s_2 are compatible and $s_1 \oplus s_2 = s$. Since, $\mathit{ext}_{U_i}(s_i) \in \|P_i\|_G$, by IH, $\mathit{ext}_{U_i}(s_i) \in \|\boldsymbol{\tau}(P_i)\|_{\mathit{triple}(G)}$. Let $V = \mathit{dom}(s_1) \cap \mathit{dom}(s_2)$ and $V_i = U \setminus \mathit{dom}(s_i)$. Then V_1, V_2 and V are disjoint and partition U. By definition, $\mathit{ext}_{U_i}(s_i)\colon v \mapsto \mathit{null}$, for each $v \in V_i$, and therefore $\mathit{ext}_{U_i}(s_i)$ is in $\|\sigma_{\mathit{isNull}(V_i)}\boldsymbol{\tau}(P_i)\|_{\mathit{triple}(G)}$. Let $t_i = \mathit{ext}_{U_i \setminus V_i}(s_i)$ and $Q_i = \pi_{U_i \setminus V_i}(\sigma_{\mathit{isNull}(V_i)}\boldsymbol{\tau}(P_i))$. We have $t_i \in \|Q_i\|_{\mathit{triple}(G)}$, and since s_1 and s_2 are compatible and V are the common non-null attributes of t_1 and t_2, we obtain $t_1 \oplus t_2 \in \|Q_1 \bowtie Q_2\|_{\mathit{triple}(G)}$. As t extends $t_1 \oplus t_2$ to $V_1 \cup V_2$ by *null*s, we have $t \in \|\boldsymbol{\tau}(\text{JOIN}(P_1, P_2))\|_{\mathit{triple}(G)}$.

If $t \in \|\boldsymbol{\tau}(\text{JOIN}(P_1, P_2))\|_{\mathit{triple}(G)}$ then there are disjoint $V_1, V_2 \subseteq U$ and compatible tuples t_1 and t_2 such that $t_i \in \|\pi_{U_i \setminus V_i}(\sigma_{\mathit{isNull}(V_i)}\boldsymbol{\tau}(P_i))\|_{\mathit{triple}(G)}$ and t extends $t_1 \oplus t_2$ to $V_1 \cup V_2$ by *null*s. Let $s_i = \{v \mapsto t(v) \mid v \in U_i \text{ and } t(v) \text{ is not } \mathit{null}\}$. Then s_1 and s_2 are compatible and $\mathit{ext}_{U_i}(s_i) \in \|\boldsymbol{\tau}(P_i)\|_{\mathit{triple}(G)}$. By IH, $\mathit{ext}_{U_i}(s_i) \in \|P_i\|_G$ and $s_i \in [\![P_i]\!]_G$. So, $s_1 \oplus s_2 \in [\![\text{JOIN}(P_1, P_2)]\!]_G$ and $\mathit{ext}_{U_1 \cup U_2}(s_1 \oplus s_2) = t \in \|\text{JOIN}(P_1, P_2)\|_G$. ❑

3.1 Optimising SPARQL JOIN and OPT

By definition, $\boldsymbol{\tau}(\text{JOIN}(P_1, P_2))$ is a union of exponentially many natural joins ($\bowtie$). Observe, however, that for any BGP $B = \{\mathit{tp}_1, \dots, \mathit{tp}_n\}$, none of the attributes in the $\boldsymbol{\tau}(\mathit{tp}_i)$ can be *null*. So, we can drastically simplify the definition of $\boldsymbol{\tau}(B)$ by taking

$$\boldsymbol{\tau}(\{\mathit{tp}_1, \dots, \mathit{tp}_n\}) = \boldsymbol{\tau}(\mathit{tp}_1) \bowtie \cdots \bowtie \boldsymbol{\tau}(\mathit{tp}_n).$$

Moreover, this observation can be generalised. First, we identify the variables in graph patterns that are not necessarily bound in solution mappings:

$$\begin{aligned}
\nu(B) &= \emptyset, \qquad B \text{ is a BGP},\\
\nu(\text{FILTER}(P_1, F)) &= \nu(P_1) \setminus \{v \mid \mathit{bound}(v) \text{ is a conjunct of } F\},\\
\nu(\text{BIND}(P_1, v, c)) &= \nu(P_1),\\
\nu(\text{UNION}(P_1, P_2)) &= (\mathit{var}(P_1) \setminus \mathit{var}(P_2)) \cup (\mathit{var}(P_2) \setminus \mathit{var}(P_1)) \cup \nu(P_1) \cup \nu(P_2),\\
\nu(\text{JOIN}(P_1, P_2)) &= \nu(P_1) \cup \nu(P_2),\\
\nu(\text{OPT}(P_1, P_2, F)) &= \nu(P_1) \cup \mathit{var}(P_2).
\end{aligned}$$

Thus, if a variable v in P does not belong to $\nu(P)$, then $v \in \mathit{dom}(s)$, for any solution mapping $s \in [\![P]\!]_G$ and RDF graph G (but not the other way round). Now, we observe that the union in the definition of $\boldsymbol{\tau}(\text{JOIN}(P_1, P_2))$ can be taken over those subsets of $\mathit{var}(P_1) \cap \mathit{var}(P_2)$ that only contain variables from $\nu(P_1) \cup \nu(P_2)$. This gives us:

Theorem 8. *If $\mathit{var}(P_1) \cap \mathit{var}(P_2) \cap (\nu(P_1) \cup \nu(P_2)) = \emptyset$ then we can define*

$$\boldsymbol{\tau}(\text{JOIN}(P_1, P_2)) = \boldsymbol{\tau}(P_1) \bowtie \boldsymbol{\tau}(P_2), \quad \boldsymbol{\tau}(\text{OPT}(P_1, P_2, F)) = \boldsymbol{\tau}(P_1) ⟕_{\boldsymbol{\tau}(F)} \boldsymbol{\tau}(P_2),$$

where $R_1 ⟕_F R_2 = \sigma_F(R_1 \bowtie R_2) \cup \mu_{U_2 \setminus U_1}(R_1 \setminus \pi_{U_1}(\sigma_F(R_1 \bowtie R_2)))$, for R_i over U_i.

(Note that the relational operation $⟕_F$ corresponds to LEFT JOIN in SQL with the condition F placed in its ON clause.)

Example 9. Consider the following BGP B taken from the official SPARQL specification ('find the names of people who do not know anyone'):

$$\textsc{Filter}(\textsc{Opt}(\{\, ?x \text{ foaf:givenName } ?n \,\},\ \{\, ?x \text{ foaf:knows } ?w \,\},\ \top), \neg \mathit{bound}(?w)).$$

By Theorem 8, $\boldsymbol{\tau}(B)$ is defined as $\sigma_{\mathit{isNull}(w)}(\pi_{x,n}Q_1 ⟕ \pi_{x,w}Q_2)$, where Q_1 and Q_2 are $\sigma_{\mathit{pred}=\text{foaf:givenName}}\rho_{x/\mathit{subj}}\rho_{n/\mathit{obj}}\ \mathit{triple}$ and $\sigma_{\mathit{pred}=\text{foaf:knows}}\rho_{x/\mathit{subj}}\rho_{w/\mathit{obj}}\ \mathit{triple}$, respectively (we note in passing that the projection on x is equivalent to $\pi_x Q_1 \setminus \pi_x Q_2$).

3.2 R2RML Mappings

The SQL translation of a SPARQL query constructed above has to be evaluated over the ternary relation *triple*(G) representing the virtual RDF graph G. Our aim now is to transform it to an SQL query over the actual database, which is related to G by means of an R2RML mapping [10]. A variant of such a transformation has been suggested in [27]. Here we develop the idea first presented in [28]. We begin with a simple example.

Example 10. The following R2RML mapping (in the Turtle syntax) populates an object property ub:UGDegreeFrom from a relational table students, whose attributes id and degreeuniid identify graduate students and their universities:

```
_:m1 a rr:TripleMap;
      rr:logicalTable [ rr:sqlQuery "SELECT * FROM students WHERE stype=1" ];
      rr:subjectMap [ rr:template "/GradStudent{id}" ] ;
      rr:predicateObjectMap [ rr:predicate ub:UGDegreeFrom ;
                              rr:objectMap [ rr:template "/Uni{degreeuniid}" ] ]
```

More specifically, for each tuple in the query, an R2RML processor generates an RDF triple with the predicate ub:UGDegreeFrom and the subject and object constructed from attributes id and degreeuniid, respectively, using IRI templates.

Our aim now is as follows: given an R2RML mapping $\mathcal{M}$, we are going to define an SQL query $\mathsf{tr}_{\mathcal{M}}(\mathit{triple})$ that constructs the relational representation $\mathit{triple}(G_{D,\mathcal{M}})$ of the virtual RDF graph $G_{D,\mathcal{M}}$ obtained by $\mathcal{M}$ from any given data instance D. Without loss of generality and to simplify presentation, we assume that each triple map has

– one logical table (rr:sqlQuery),
– one subject map (rr:subjectMap), which does not have resource typing (rr:class),
– and one predicate-object map with one rr:predicateMap and one rr:objectMap.

This normal form can be achieved by introducing predicate-object maps with rdf:type and splitting any triple map into a number of triple maps with the same logical table and subject. We also assume that triple maps contain no referencing object maps (rr:parentTriplesMap, etc.) since they can be eliminated using joint SQL queries [10]. Finally, we assume that the term maps (i.e., subject, predicate and object maps) contain no constant shortcuts and are of the form [rr:column v], [rr:constant c] or [rr:template s].

Given a triple map m with a logical table (SQL query) R, we construct a selection $\sigma_{\neg \mathit{isNull}(v_1)} \cdots \sigma_{\neg \mathit{isNull}(v_k)} R$, where $v_1, \ldots, v_k$ are the *referenced columns* of m (attributes of R in the term maps in m)—this is done to exclude tuples that contain *null* [10]. To construct tr_m, the selection filter is prefixed with projection $\pi_{\mathit{subj},\mathit{pred},\mathit{obj}}$

and, for each of the three term maps, either with renaming (e.g., with $\rho_{obj/v}$ if the object map is of the form [rr:column v]) or with value creation (if the term map is of the form [rr:constant c] or [rr:template s]; in the latter case, we use the built-in string concatenation function ||). For instance, the mapping _:m1 from Example 10 is converted to the SQL query

```
SELECT ('/GradStudent' || id) AS subj, 'ub:UGDegreeFrom' AS pred,
        ('/Uni' || degreeuniid) AS obj FROM students
WHERE (id IS NOT NULL) AND (degreeuniid IS NOT NULL) AND (stype=1).
```

Given an R2RML mapping $\mathcal{M}$, we set $\mathsf{tr}_{\mathcal{M}}(\mathit{triple}) = \bigcup_{m \in \mathcal{M}} \mathsf{tr}_m$.

Proposition 11. *For any R2RML mapping* $\mathcal{M}$ *and data instance* D*,* $t \in \|\mathsf{tr}_{\mathcal{M}}(\mathit{triple})\|_D$ *if and only if* $t \in \mathit{triple}(G_{D,\mathcal{M}})$*.*

Finally, given a graph pattern P and an R2RML mapping $\mathcal{M}$, we define $\mathsf{tr}_{\mathcal{M}}(\boldsymbol{\tau}(P))$ to be the result of replacing every occurrence of the relation *triple* in the query $\boldsymbol{\tau}(P)$, constructed in Section 3, with $\mathsf{tr}_{\mathcal{M}}(\mathit{triple})$. By Theorem 7 and Proposition 11, we obtain:

Theorem 12. *For any graph pattern* P*, R2RML mapping* $\mathcal{M}$ *and data instance* D*,* $\|P\|_{G_{D,\mathcal{M}}} = \|\mathsf{tr}_{\mathcal{M}}(\boldsymbol{\tau}(P))\|_D$*.*

3.3 Optimising SQL Translation

The straightforward application of $\mathsf{tr}_{\mathcal{M}}$ to $\boldsymbol{\tau}(P)$ can result in a very complex SQL query. We now show that such queries can be optimised by the following techniques:

- choosing matching tr_m from $\mathsf{tr}_{\mathcal{M}}(\mathit{triple})$, for each occurrence of *triple* in $\boldsymbol{\tau}(P)$;
- using the distributivity of $\bowtie$ over $\cup$ and removing sub-queries with *incompatible IRI templates* and *de-IRIing* join conditions;
- functional dependencies (e.g., primary keys) for self-join elimination [6,18,29,30].

To illustrate, suppose we are given a mapping $\mathcal{M}$ containing _:m1 from Example 10 and the following triple maps (which are a simplified version of those in Section 5):

```
_:m2 a rr:TripleMap;
     rr:logicalTable [ rr:sqlQuery "SELECT * FROM students WHERE stype=0" ];
     rr:subjectMap [ rr:template "/UGStudent{id}"; rr:class ub:Student ].
_:m3 a rr:TripleMap;
     rr:logicalTable [ rr:sqlQuery "SELECT * FROM students WHERE stype=1" ];
     rr:subjectMap [ rr:template "/GradStudent{id}"; rr:class ub:Student ].
```

which generate undergraduate and graduate students (both are instances of ub:Student, but their IRIs are constructed using different templates [16]). Consider the following query (a fragment of q_2^{obg} from Section 5):

SELECT $?x$ $?y$ WHERE $\{$ $?x$ rdf:type ub:Student. $?x$ ub:UGDegreeFrom $?y$ $\}$.

The translation $\boldsymbol{\tau}$ of its BGP (after the SPARQL JOIN optimisation of Section 3.1) is

$$(\pi_x \rho_{x/\mathit{subj}} \sigma_{(\mathit{pred}=\text{rdf:type}) \land (\mathit{obj}=\text{ub:Student})}\ \mathit{triple}) \bowtie \\ (\pi_{x,y} \rho_{x/\mathit{subj}} \rho_{y/\mathit{obj}} \sigma_{\mathit{pred}=\text{ub:UGDegreeFrom}}\ \mathit{triple})$$

First, since *triple* always occurs in the scope of some selection operation σ_F, we can choose only those elements in $\bigcup_{m\in\mathcal{M}} \mathsf{tr}_m$ that have matching values of *pred* and/or *obj*. In our example, the first occurrence of *triple* is replaced by $\mathtt{tr}_{\mathtt{_:m2}} \cup \mathtt{tr}_{\mathtt{_:m3}}$, and the second one by $\mathtt{tr}_{\mathtt{_:m1}}$. This results in the natural join of the following union, denoted A:

```
(SELECT DISTINCT '/UGStudent' || id AS x FROM students
 WHERE (id IS NOT NULL) AND (stype=0))
UNION (SELECT DISTINCT '/GradStudent' || id AS x FROM students
       WHERE (id IS NOT NULL) AND (stype=1))
```

and of the following query, denoted B:

```
SELECT DISTINCT '/GradStudent' || id AS x, '/Uni' || degreeuniid AS y FROM students
WHERE (id IS NOT NULL) AND (degreeuniid IS NOT NULL) AND (stype=1)
```

Second, observe that the IRI template in B is compatible only with the second component of A. Moreover, since the two compatible templates coincide, we can *de-IRI* the join, namely, replace the join over the constructed strings (A.x = B.x) by the join over the numerical attributes (A.id = B.id), which results in a more efficient query:

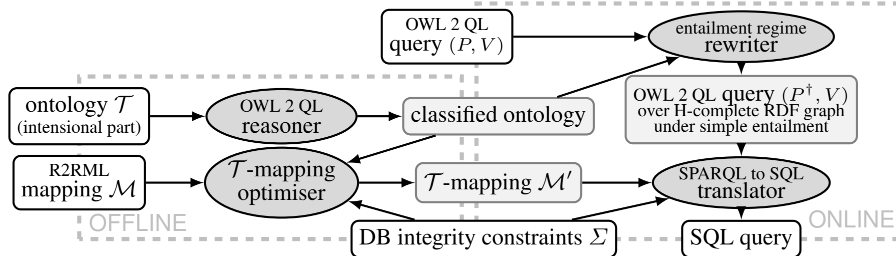
```
SELECT DISTINCT A.x, B.y FROM
   (SELECT id, '/GradStudent' || id AS x FROM students
    WHERE (id IS NOT NULL) AND (stype=1)) A
   JOIN
      (SELECT id, '/GradStudent' || id AS x, '/Uni' || degreeuniid AS y FROM students
       WHERE (id IS NOT NULL) AND (degreeuniid IS NOT NULL) AND (stype=1)) B
   ON A.id = B.id
```

Finally, by using self-join elimination and the fact that id and stype are the composite primary key in students, we obtain the query (without DISTINCT as x is unique)

```
SELECT '/GradStudent' || id AS x, '/Uni' || degreeuniid AS y FROM students
WHERE (degreeuniid IS NOT NULL) AND (stype=1)
```

4 Putting it all Together

The techniques introduced above suggest the following architecture to support answering SPARQL queries under the OWL 2 QL entailment regime with data instances stored in a database. Suppose we are given an ontology with an intensional part $\mathcal{T}$ and an extensional part stored in a database, D, over a schema Σ. Suppose also that the languages of Σ and $\mathcal{T}$ are connected by an R2RML mapping $\mathcal{M}$. The process of answering a given OWL 2 QL query (P, V) involves two stages, off-line and on-line.



The *off-line* stage takes $\mathcal{T}$, $\mathcal{M}$ and Σ and proceeds via the following steps:

❶ An OWL 2 QL reasoner is used to obtain a complete class / property hierarchy in $\mathcal{T}$.

❷ The composition $\mathcal{M}^{\mathcal{T}}$ of $\mathcal{M}$ with the class and property hierarchy in $\mathcal{T}$ is taken as an initial $\mathcal{T}$-mapping. Recall [29] that a mapping $\mathcal{M}'$ is a *$\mathcal{T}$-mapping over Σ* if, for any data instance D satisfying Σ, the *virtual* (not materialised) RDF graph $G_{D,\mathcal{M}'}$ obtained by applying $\mathcal{M}'$ to D contains all class and property assertions α with $(\mathcal{T}, G_{D,\mathcal{M}'}) \models \alpha$. As a result, $G_{D,\mathcal{M}'}$ is complete with respect to the class and property hierarchy in $\mathcal{T}$ (or H-complete), which allows us to avoid reasoning about class and property inclusions (in particular, inferences that involve property domains and ranges) at the query rewriting step ❹ and drastically simplify rewritings (see [29] for details).
❸ The initial $\mathcal{T}$-mapping $\mathcal{M}^{\mathcal{T}}$ is then optimised by (*i*) eliminating redundant triple maps detected by query containment with inclusion dependencies in Σ, (*ii*) eliminating redundant joins in logical tables using the functional dependencies in Σ, and (*iii*) merging sets of triple maps by means of interval expressions or disjunctions in logical tables (see [29] for details). Let $\mathcal{M}'$ be the resulting $\mathcal{T}$-mapping over Σ.

The *on-line* stage takes an OWL 2 QL query (P, V) as an input and proceeds as follows:
❹ The graph pattern P and $\mathcal{T}$ are rewritten to the OWL 2 QL graph pattern $P^\dagger$ over the H-complete virtual RDF graph $G_{D,\mathcal{M}'}$ *under simple entailment* by applying the classified ontology of step ❶ to instantiate class and property variables and then using a query rewriting algorithm (e.g., the tree-witness rewriter of [29]); see Theorem 4.
❺ The graph pattern $P^\dagger$ is transformed to the SQL query $\boldsymbol{\tau}(P^\dagger)$ over the 3-column representation *triple* of the RDF graph (Theorem 7). Next, the query $\boldsymbol{\tau}(P^\dagger)$ is unfolded into the SQL query $\mathsf{tr}_{\mathcal{M}'}(\boldsymbol{\tau}(P^\dagger))$ over the original database D (Theorem 12). The unfolded query is optimised using the techniques similar to the ones employed in step ❸.
❻ The optimised query is executed by the database.

As follows from Theorems 4, 7 and 12, the resulting query gives us all correct answers to the original OWL 2 QL query (P, V) over $\mathcal{T}$ and D with the R2RML mapping $\mathcal{M}$.

5 Evaluation

The architecture described above has been implemented in the open-source OBDA system *Ontop*[4]. We evaluated its performance using the OWL 2 QL version of the Lehigh University Benchmark LUBM [16]. The ontology contains 43 classes, 32 object and data properties and 243 axioms. The benchmark also includes a data generator and a set of 14 queries q_1–q_{14}. We added 7 queries with second-order variables ranging over class and property names: q_4', q_4'', q_9', q_9'' derived from q_4 and q_9, and $q_2^{obg}, q_4^{obg}, q_{10}^{obg}$ taken from [19]. The LUBM data generator produces an OWL file with class and property assertions. To store the assertions in a database, we created a database schema with 11 relations and an R2RML mapping with 89 predicate-object maps. For instance, the information about undergraduate and graduate students (id, name, etc.) from Example 10 is collected in the relation students, where the attribute stype distinguishes between the types of students (stype is known as a discriminant column in databases); more details including primary and foreign keys and indexes are provided in the full version.

We experimented with the data instances LUBM$_n$, $n = 1, 9, 20, 50, 100, 200, 500$ (where n specifies the number of universities; LUBM$_1$ and LUBM$_9$ were used in [19]).

[4] `http://ontop.inf.unibz.it`

Q	$LUBM_1$				$LUBM_9$			$LUBM_{100}$		$LUBM_{200}$	$LUBM_{500}$
	O	OB_H	OB_P	P	O	OB_H	P	O	P	O	O
q_1	2	8	29	1	3	97	1	3	1	3	2
q_2	2	25	11 137	19	3	2 531	256	16	30 593	36	88
q_3	1	6	86	9	2	78	158	2	2 087	63	12
q_4	13	7	19	14	15	44	164	27	2 093	24	22
q_5	16	12	4 451	10	22	98	158	32	2 182	28	23
q_6	455	27	32	21	5 076	411	317	58 968	10 781	123 578	434 349
q_7	5	21	34 005	10	6	429	157	8	2 171	8	9
q_8	726	195	95 875	80	760	917	192	796	2 131	820	855
q_9	60	972	168 978	78	668	189 126	857	7 466	12 125	15 227	44 598
q_{10}	2	6	126	9	3	97	158	2	2 134	3	2
q_{11}	4	5	58	10	6	43	160	11	2 093	18	44
q_{12}	3	4	19	15	4	70	236	3	2 114	5	5
q_{13}	6	4	67	8	7	40	157	14	2 657	38	58
q_{14}	91	20	24	15	1 168	329	287	13 524	4 457	29 512	92 376
q'_4	93	58	190	46	99	98	767	92	4 422	95	107
q''_4	108	21	35	63	122	72	719	115	9 179	108	127
q'_9	257	716	91 855	174	4 686	40 575	1 385	54 092	19 945	115 110	295 228
q''_9	557	951	65 916	102	6 093	178 401	1 214	67 123	19 705	151 376	356 176
q_2^{obg}	150	30	57 141	29	9 992	520	348	39 477	5 411	79 351	206 061
q_4^{obg}	6	7	241	25	31	40	273	7	3 969	7	494
q_{10}^{obg}	641	760	31 269	253	6 998	149 191	2 258	163 308	17 929	174 362	459 669
start up	3.1s	13.6s	7.7s	3.6s	3.1s	80m33s	18s	3.1s	3m23s	3.1s	3.1s
data load	10s	n/a	n/a	n/a	15s	n/a	n/a	1m56s	n/a	3m35s	10m17s

Table 1. Start up time, data loading time (in s) and query execution time (in ms): O is *Ontop*, OB_H and OB_P are OWL-BGP with Hermit and Pellet, respectively, and P is standalone Pellet.

Here we only show the results for $n = 1, 9, 100, 200, 500$ containing 103k, 1.2M, 14M, 28M and 69M triples, respectively; the complete table can be found in the full version. All the materials required for the experiments are available online[5]. We compared *Ontop* with two other systems, OWL-BGP r123 [19] and Pellet 2.3.1 [31] (Stardog and OWLIM are incomplete for the OWL 2 QL entailment regime). OWL-BGP requires an OWL 2 reasoner as a backend; as in [19], we employed HermiT 1.3.8 [14] and Pellet 2.3.1. The hardware was an HP Proliant Linux server with 144 cores @3.47GHz, 106GB of RAM and a 1TB 15k RPM HD. Each system used a single core and was given 20 GB of Java 7 heap memory. *Ontop* used MySQL 5.6 database engine.

The evaluation results are given in Table 1. OWL-BGP and Pellet used significantly more time to start up (last but one row) because they do not rely on query rewriting and require costly pre-computations. OWL-BGP failed to start on $LUBM_9$ with Pellet and on $LUBM_{20}$ with HermiT; Pellet ran out of memory after 10hrs loading $LUBM_{200}$. For *Ontop*, the start up is the off-line stage described in Section 4; it does not include the time of loading the data into MySQL, which is specified in the last row of Table 1 (note that the data is loaded only once, not every time *Ontop* starts; moreover, this could be improved with CSV loading and delayed indexing rather than SQL dumps we used).

On queries q_1–q_{14}, *Ontop* generally outperforms OWL-BGP and Pellet. Due to the optimisations, the SQL queries generated by *Ontop* are very simple, and MySQL is able to execute them efficiently. This is also the case for large datasets, where *Ontop* is able to maintain almost constant times for many of the queries. Notable exceptions are q_6, q_8 and q_{14} that return a very large number (hundreds of thousands) of results (low

[5] `https://github.com/ontop/iswc2014-benchmark`

selectivity). A closer inspection reveals that execution time is mostly spent on fetching the results from disk. On the queries with second-order variables, the picture is mixed. While indeed these queries are not the strongest point of *Ontop* at the moment, we see that in general the performance is good. Although Pellet outperforms *Ontop* on small datasets, only *Ontop* is able to provide answers for very large datasets. For second-order queries with high selectivity (e.g., q_4' and q_4'') and large datasets, the performance of *Ontop* is very good while the other systems fail to return answers.

6 Conclusions

In this paper, we gave both a theoretical background and a practical implementation of a procedure for answering SPARQL 1.1 queries under the OWL 2 QL direct semantics entailment regime in the scenario where data instances are stored in a relational database whose schema is connected to the language of the given OWL 2 QL ontology via an R2RML mapping. Our main contributions can be summarised as follows:

– We defined an entailment regime for SPARQL 1.1 corresponding to the OWL 2 QL profile of OWL 2 (which was specifically designed for ontology-based data access).
– We proved that answering SPARQL queries under this regime is reducible to answering SPARQL queries under simple entailment (where no reasoning is involved).
– We showed how to transform such SPARQL queries to equivalent SQL queries over an RDF representation of the data, and then unfold them, using R2RML mappings, into SQL queries over the original relational data.
– We developed optimisation techniques to substantially reduce the size and improve the quality of the resulting SQL queries.
– We implemented these rewriting and optimisation techniques in the OBDA system *Ontop*. Our initial experiments showed that *Ontop* generally outperforms reasoner-based systems, especially on large data instances.

Some aspects of SPARQL 1.1 (such as RDF types, property paths, aggregates) were not discussed here and are left for future work.

Acknowledgements. Our work was supported by EU project Optique. We thank S. Komla-Ebri for help with the experiments, and I. Kollia and B. Glimm for discussions.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Angles, R., Gutierrez, C.: The expressive power of SPARQL. In: Proc. of ISWC. LNCS, vol. 5318, pp. 114–129. Springer (2008)
3. Bornea, M., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., Bhattacharjee, B.: Building an efficient RDF store over a relational database. In: Proc. of SIGMOD 2013, pp. 121–132. ACM (2013)
4. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R., Ruzzi, M., Savo, D.F.: The MASTRO system for ontology-based data access. Semantic Web 2(1), 43–53 (2011)
5. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. J. of Automated Reasoning 39(3), 385–429 (2007)

6. Chakravarthy, U.S., Grant, J., Minker, J.: Logic-based approach to semantic query optimization. ACM Transactions on Database Systems 15(2), 162–207 (1990)
7. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. Data Knowl. Eng. 68(10), 973–1000 (2009)
8. Chortaras, A., Trivela, D., Stamou, G.: Optimized query rewriting for OWL 2 QL. In: Proc. of CADE-23. LNCS, vol. 6803, pp. 192–206. Springer (2011)
9. Cyganiak, R.: A relational algebra for SPARQL. Tech. Rep. HPL-2005-170, HP Labs (2005)
10. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF Mapping Language (September 2012), `http://www.w3.org/TR/r2rml`
11. Dolby, J., Fokoue, A., Kalyanpur, A., Ma, L., Schonberg, E., Srinivas, K., Sun, X.: Scalable grounded conjunctive query evaluation over large and expressive knowledge bases. In: Proc. of ISWC. LNCS, vol. 5318, pp. 403–418. Springer (2008)
12. Eiter, T., Ortiz, M., Šimkus, M., Tran, T.K., Xiao, G.: Query rewriting for Horn-SHIQ plus rules. In: Proc. of AAAI. AAAI Press (2012)
13. Elliott, B., Cheng, E., Thomas-Ogbuji, C., Özsoyoglu, Z.M.: A complete translation from SPARQL into efficient SQL. In: Proc. of IDEAS. pp. 31–42. ACM (2009)
14. Glimm, B., Horrocks, I., Motik, B., Stoilos, G.: Optimising ontology classification. In: Proc. of ISWC, part I. LNCS, vol. 6496, pp. 225–240. Springer (2010)
15. Gottlob, G., Orsi, G., Pieris, A.: Ontological queries: Rewriting and optimization. In: Proc. of ICDE. pp. 2–13. IEEE Computer Society (2011)
16. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. of Web Semantics 3(2–3), 158–182 (2005)
17. Heymans, S. *et al.*: Ontology reasoning with large data repositories. In: Ontology Management, Semantic Web, Semantic Web Services, and Business Applications. Springer (2008)
18. King, J.J.: Query Optimization by Semantic Reasoning. Ph.D. thesis, Stanford, USA (1981)
19. Kollia, I., Glimm, B.: Optimizing SPARQL query answering over OWL ontologies. J. of Artificial Intelligence Research 48, 253–303 (2013)
20. König, M., Leclère, M., Mugnier, M.-L., Thomazo, M.: On the exploration of the query rewriting space with existential rules. In: Proc. of RR. LNCS. pp. 123–137. Springer (2013)
21. Kontchakov, R., Rodríguez-Muro, M., Zakharyaschev, M.: Ontology-based data access with databases: A short course. In: Reasoning Web. LNCS, vol. 8067, pp. 194–229. Springer (2013)
22. Lutz, C., Seylan, I., Toman, D., Wolter, F.: The combined approach to OBDA: Taming role hierarchies using filters. In: Proc. of ISWC. LNCS, vol. 8218, pp. 314–330. Springer (2013)
23. Pérez-Urbina, H., Rodríguez-Díaz, E., Grove, M., Konstantinidis, G., Sirin, E.: Evaluation of query rewriting approaches for OWL 2. In: SSWS+HPCSW. CEUR-WS, vol. 943 (2012)
24. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. J. on Data Semantics X, 133–173 (2008)
25. Polleres, A.: From SPARQL to rules (and back). In: Proc. WWW. pp. 787–796. ACM (2007)
26. Polleres, A., Wallner, J.P.: On the relation between SPARQL 1.1 and Answer Set Programming. J. of Applied Non-Classical Logics 23(1–2), 159–212 (2013)
27. Priyatna, F., Corcho, O., Sequeda, J.: Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph. In: Proc. of WWW. pp. 479–490 (2014)
28. Rodríguez-Muro, M., Hardi, J., Calvanese, D.: Quest: Efficient SPARQL-to-SQL for RDF and OWL. In: Proc. of the ISWC 2012 P&D Track. vol. 914. CEUR-WS.org (2012)
29. Rodríguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-based data access: Ontop of databases. In: Proc. of ISWC. LNCS, vol. 8218, pp. 558–573. Springer (2013)
30. Sequeda, J.F., Miranker, D.P.: Ultrawrap: SPARQL execution on relational data. J. of Web Semantics 22, 19–39 (2013)
31. Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL Reasoner. J. of Web Semantics 5(2), 51–53 (2007)
32. Zemke, F.: Converting SPARQL to SQL. Tech. rep., Oracle Corp. (2006)

A On the Semantics of SPARQL

Remark 1. The condition '$F^{s_1 \oplus s_2}$ *is not true*' in our definition of OPT is slightly different from '$F^{s_1 \oplus s_2}$ has an effective Boolean value of false' given by the W3C specification[6]. The two definitions do not necessarily coincide because the effective Boolean value can be undefined (type error) if, for example, a variable in F is not bound by $s_1 \oplus s_2$. As we see from Section 3.1, our reading corresponds to LEFT JOIN in SQL.

We also find the informal explanation of the semantics for OPT in the W3C recommendation inconsistent with the definition of DIFF, which forms the second component of the union in the definition of OPT. It suggests that $\textsc{Diff}(S_1, S_2, F)$ is equivalent to

$$\begin{aligned}\textsc{Diff}'(S_1, S_2, F) \;=\; &\{s_1 \in S_1 \mid s_1 \text{ and } s_2 \text{ are incompatible, for all } s_2 \in S_2\}\\ \cup\;\; &\{s_1 \in S_1 \mid \text{there is } s_2 \in S_2 \text{ compatible with } s_1 \text{ such that } F^{s_1 \oplus s_2} = \bot\}.\end{aligned}$$

Observe that there may be $s_2, s_2' \in S_2$ that are both compatible with s_1, $F^{s_1 \oplus s_2} = \top$ and $F^{s_1 \oplus s_2'} = \bot$, in which case $s_1 \in \textsc{Diff}'(S_1, S_2, F) \setminus \textsc{Diff}(S_1, S_2, F)$.

B Proof of Theorem 1

Theorem 4. *Given any intensional graph $\mathcal{T}$ and OWL 2 QL query (P, V), one can construct an OWL 2 QL query $(P^\dagger, V)$ such that, for any extensional graph $\mathcal{A}$ (in some fixed finite vocabulary),*

$$[\![P]\!]_{\mathcal{T},\mathcal{A}}|_V \;=\; [\![P^\dagger]\!]_{\mathcal{A}}|_V.$$

Proof. By the definition of the entailment regime, it suffices to construct a rewriting $B^\dagger$, for any *basic* graph pattern B, such that $[\![B]\!]_{\mathcal{T},\mathcal{A}} \;=\; [\![B^\dagger]\!]_{\mathcal{A}}$. A rewriting $P^\dagger$ of P is obtained by replacing every BGP B in it with $B^\dagger$.

Take any BGP B that occurs in P. Let $?c_1, \dots, ?c_m$ be all class variables in B, and let $?r_1, \dots, ?r_n$ be all property variables in B. For any m-tuple $\boldsymbol{C} = (C_1, \dots, C_m)$ of class names and n-tuple $\boldsymbol{R} = (R_1, \dots, R_n)$ of property names in the given vocabulary, we take the result $B'_{\boldsymbol{C},\boldsymbol{R}}$ of replacing every $?c_i$ in B with C_i and every $?r_j$ in B with R_j. By definition, $B'_{\boldsymbol{C},\boldsymbol{R}}$ contains no class or property variables and contains only OWL 2 QL class and property axioms (rather than templates) and assertions. For any class or property axiom in $B'_{\boldsymbol{C},\boldsymbol{R}}$, we check whether it is entailed by (logically follows from) $\mathcal{T}$. If at least one of the axioms is not entailed by $\mathcal{T}$, we set $B''_{\boldsymbol{C},\boldsymbol{R}}$ to be the empty BGP; otherwise, let $B''_{\boldsymbol{C},\boldsymbol{R}}$ be the result of removing all class or property axioms from $B'_{\boldsymbol{C},\boldsymbol{R}}$.

The constructed BGP $B''_{\boldsymbol{C},\boldsymbol{R}}$ contains only class and property assertions and can be regarded as (a SPARQL representations of) a conjunctive query. As is well-known [29], we can rewrite this query and $\mathcal{T}$ into a union of conjunctive queries over $\mathcal{A}$, which can be represented as a union $B^\dagger_{\boldsymbol{C},\boldsymbol{R}}$ of BGPs. Now, we take $B^\dagger$ to be the union (UNION) of

$$B^\dagger_{\boldsymbol{C},\boldsymbol{R}}[?c_1 \mapsto C_1, \dots, ?c_m \mapsto C_m, ?r_1 \mapsto R_1, \dots ?r_n \mapsto R_n],$$

[6] `http://www.w3.org/TR/sparql11-query/#sparqlAlgebra`

for all possible m-tuples $\boldsymbol{C}$ of class names and n-tuples $\boldsymbol{R}$ of property names in the given vocabulary such that $B''_{\boldsymbol{C},\boldsymbol{R}}$ is not empty (the empty union is, by definition, the empty BGP). Here

$$B[?v_1 \mapsto V_1, \dots, ?v_k \mapsto V_k] = \textsc{Bind}(\dots \textsc{Bind}(B, ?v_1, V_1), \dots, ?v_k, V_k).$$

It follows from the construction that $[\![B]\!]_{\mathcal{T},\mathcal{A}} = [\![B^\dagger]\!]_{\mathcal{A}}$. ❑

We emphasise that $[\![P^\dagger]\!]_{\mathcal{A}}$ is computed only on the extensional part of the RDF graph and under simple entailment.

C Proof of Proposition 6 and Theorem 7

Proposition 6. *Let V be a set of variables and F a SPARQL filter expression with variables in V. For each solution mapping s with $dom(s) \subseteq V$, we have $F^s = (\boldsymbol{\tau}(F))^{ext_V(s)}$.*

Proof. The proof is based on the observation that $ext_V(s)\colon v \mapsto null$ just in case $v \notin dom(s)$, for each $v \in V$. Thus, the claim holds for $F = bound(v)$. Also, due to this observation, the clauses in the definition of $v = c$ and $v = v'$ and the truth tables for $\neg$ and $\wedge$ coincide for SPARQL and SQL filter expressions. ❑

Theorem 7. *For any RDF graph G and any graph pattern P,*

$$\|P\|_G = \|\boldsymbol{\tau}(P)\|_{triple(G)}.$$

Proof. The proof is by induction on the structure of P.

For the basis of induction, let P be a triple pattern of the form $\langle s, p, o\rangle$. Since each of the components of the triple pattern is either a variable in V or an RDF term in $\mathsf{I} \cup \mathsf{L}$, there are 15 possible cases (recall that we have no blank nodes):

	$s, p, o \in \mathsf{I} \cup \mathsf{L}$	
$s \in \mathsf{V}$, $p, o \in \mathsf{I} \cup \mathsf{L}$	$p \in \mathsf{V}$, $s, o \in \mathsf{I} \cup \mathsf{L}$	$o \in \mathsf{V}$, $s, p \in \mathsf{I} \cup \mathsf{L}$
$s, p \in \mathsf{V}$, $s \neq p$, $o \in \mathsf{I} \cup \mathsf{L}$	$s, o \in \mathsf{V}$, $s \neq o$, $p \in \mathsf{I} \cup \mathsf{L}$	$p, o \in \mathsf{V}$, $p \neq o$, $s \in \mathsf{I} \cup \mathsf{L}$
$s, p \in \mathsf{V}$, $s = p$, $o \in \mathsf{I} \cup \mathsf{L}$	$s, o \in \mathsf{V}$, $s = o$, $p \in \mathsf{I} \cup \mathsf{L}$	$p, o \in \mathsf{V}$, $p = o$, $s \in \mathsf{I} \cup \mathsf{L}$
	$s, p, o \in \mathsf{V}$, $s \neq p, p \neq o, o \neq s$	
$s, p, o \in \mathsf{V}$, $s = p \neq o$	$s, p, o \in \mathsf{V}$, $p = o \neq s$	$s, p, o \in \mathsf{V}$, $o = s \neq p$
	$s, p, o \in \mathsf{V}$, $s = p = o$	

We consider just one case with $s, p \in \mathsf{V}$, $s \neq p$ and $o \in \mathsf{I} \cup \mathsf{L}$ and leave all other cases to the reader. By definition, we have $\boldsymbol{\tau}(\langle s, p, o\rangle) = \pi_{s,p}\rho_{s/subj}\rho_{p/pred}\sigma_{obj=o}\, triple$ and $var(\langle s, p, o\rangle) = \{s, p\}$. Then the following are equivalent:

- $\|\langle s, p, o\rangle\|_G$ contains tuple $\{s \mapsto a,\ p \mapsto b\}$;
- $[\![\langle s, p, o\rangle]\!]_G$ contains solution mapping $\{s \mapsto a,\ p \mapsto b\}$;
- G contains triple (a, b, o);
- $triple(G)$ contains tuple $\{subj \mapsto a,\ pred \mapsto b,\ obj \mapsto o\}$;
- $\|\pi_{s,p}\rho_{s/subj}\rho_{p/pred}\sigma_{obj=o}\, triple\|_{triple(G)}$ contains tuple $\{s \mapsto a,\ p \mapsto b\}$.

For the induction step, we consider the five cases of SPARQL algebra operations.

$P = \textsc{Filter}(P_1, F)$. Denote $U = var(P)$ (all variables of F are in U).

($\subseteq$) Let $t \in \|\text{FILTER}(P_1, F)\|_G$. Then there is $s \in [\![\text{FILTER}(P_1, F)]\!]_G$ such that $\mathit{ext}_U(s) = t$. By definition, we have $s \in [\![P_1]\!]_G$ and $F^s = \top$. Then $t \in \|P_1\|_G$, whence, by the induction hypothesis, $t \in \|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)}$ and, by Proposition 6, $(\boldsymbol{\tau}(F))^t = \top$. Thus, $t \in \|\sigma_{\boldsymbol{\tau}(F)}\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)}$, which coincides with $\|\boldsymbol{\tau}(\text{FILTER}(P_1, F))\|_{\mathit{triple}(G)}$.

($\subseteq$) Let $t \in \|\boldsymbol{\tau}(\text{FILTER}(P_1, F))\|_{\mathit{triple}(G)}$. By definition, $t \in \|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)}$ and $(\boldsymbol{\tau}(F))^t = \top$. By the induction hypothesis, $t \in \|P_1\|_G$. Then there is a solution mapping s such that $t = \mathit{ext}_U(s)$ and $s \in [\![P_1]\!]_G$. By Proposition 6, $F^s = \top$ and thus, we obtain $s \in [\![\text{FILTER}(P_1, F)]\!]_G$ and $t \in \|\text{FILTER}(P_1, F)\|_G$.

$P = \text{BIND}(P_1, v, c)$. (Recall that $v \notin \mathit{var}(P_1)$.)

($\subseteq$) Let $t \in \|\text{BIND}(P_1, v, c)\|_G$. Then there is $s \in [\![\text{BIND}(P_1, v, c)]\!]_G$ such that $\mathit{ext}_U(s) = t$. By definition, we have $s' \in [\![P_1]\!]_G$ for s' that coincides with s on $\mathit{dom}(s)\setminus\{v\}$ and is undefined on v. Then $s' \in \|P_1\|_G$, whence, by the induction hypothesis, $s' \in \|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)}$. Thus, $t \in \|\boldsymbol{\tau}(P_1) \times \{v \mapsto c\}\|_{\mathit{triple}(G)}$, which coincides with $\|\boldsymbol{\tau}(\text{BIND}(P_1, v, c))\|_{\mathit{triple}(G)}$.

($\subseteq$) Let $t \in \|\boldsymbol{\tau}(\text{BIND}(P_1, v, c))\|_{\mathit{triple}(G)}$. By definition, $t(v) = c$ and the restriction t' of t to $\mathit{var}(P) \setminus \{v\}$ is in $\|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)}$. By the induction hypothesis, $t' \in \|P_1\|_G$. So, there is a solution mapping s with $t' = \mathit{ext}_U(s)$ and $s \in [\![P_1]\!]_G$. Thus, $t \in [\![\text{BIND}(P_1, v, c)]\!]_G$.

$P = \text{UNION}(P_1, P_2)$. Denote $U_i = \mathit{var}(P_i)$, for $i = 1, 2$.

($\subseteq$) Let $t \in \|\text{UNION}(P_1, P_2)\|_G$. Then there is $s \in [\![\text{UNION}(P_1, P_2)]\!]_G$ such that $t = \mathit{ext}_{U_1 \cup U_2}(s)$. By definition, we have either $s \in [\![P_1]\!]_G$ or $s \in [\![P_2]\!]_G$ and so, either $\mathit{ext}_{U_1}(s) \in \|P_1\|_G$ or $\mathit{ext}_{U_2}(s) \in \|P_2\|_G$. By the induction hypothesis, either $\mathit{ext}_{U_1}(s) \in \|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)}$ or $\mathit{ext}_{U_2}(s) \in \|\boldsymbol{\tau}(P_2)\|_{\mathit{triple}(G)}$, which implies $\mathit{ext}_{U_1 \cup U_2}(s) = t \in \|\boldsymbol{\tau}(\text{UNION}(P_1, P_2))\|_{\mathit{triple}(G)}$.

($\supseteq$) Let $t \in \|\boldsymbol{\tau}(\text{UNION}(P_1, P_2))\|_{\mathit{triple}(G)}$. Let s be such that $t = \mathit{ext}_{U_1 \cup U_2}(s)$. By definition, either $\mathit{ext}_{U_1}(s) \in \|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)}$ or $\mathit{ext}_{U_2}(s) \in \|\boldsymbol{\tau}(P_2)\|_{\mathit{triple}(G)}$. By the induction hypothesis, either $\mathit{ext}_{U_1}(s) \in \|P_1\|_G$ or $\mathit{ext}_{U_2}(s) \in \|P_2\|_G$, which implies $s \in [\![P_1]\!]_G$ or $s \in [\![P_2]\!]_G$. Thus, $s \in [\![\text{UNION}(P_1, P_2)]\!]_G$ and thus, $t \in \|\text{UNION}(P_1, P_2)\|_G$.

$P = \text{JOIN}(P_1, P_2)$. Let $U_i = \mathit{var}(P_i)$, $i = 1, 2$, and $U = U_1 \cap U_2$.

($\subseteq$) If $t \in \|\text{JOIN}(P_1, P_2)\|_G$ then there is a solution mapping $s \in [\![\text{JOIN}(P_1, P_2)]\!]_G$ with $\mathit{ext}_{U_1 \cup U_2}(s) = t$, and so there are compatible s_1 and s_2 with $s_1 \oplus s_2 = s$ and $s_i \in [\![P_i]\!]_G$, for $i = 1, 2$. By definition, $\mathit{ext}_{U_i}(s_i) \in \|P_i\|_G$, $i = 1, 2$, from which, by the induction hypothesis, $\mathit{ext}_{U_i}(s_i) \in \|\boldsymbol{\tau}(P_i)\|_{\mathit{triple}(G)}$. Let $V = \mathit{dom}(s_1) \cap \mathit{dom}(s_2)$ and $V_i = U \setminus \mathit{dom}(s_i)$, for $i = 1, 2$. Then V_1, V_2 and V are disjoint and partition U. By definition, $\mathit{ext}_{U_i}(s_i)\colon v \mapsto \mathit{null}$, for each $v \in V_i$ and $i = 1, 2$, and therefore, $\mathit{ext}_{U_i}(s_i)$ is in $\|\sigma_{\mathit{isNull}(V_i)}\boldsymbol{\tau}(P_i)\|_{\mathit{triple}(G)}$. Thus, $\mathit{ext}_{U_i \setminus V_i}(s_i) \in \|\pi_{U_i \setminus V_i}(\sigma_{\mathit{isNull}(V_i)}\boldsymbol{\tau}(P_i))\|_{\mathit{triple}(G)}$. Since s_1 and s_2 are compatible and V are the common non-null attributes of $\mathit{ext}_{U_1 \setminus V_1}(s_1)$ and

$\mathit{ext}_{U_2\setminus V_2}(s_2)$, we obtain

$$\mathit{ext}_{U_1\setminus V_1}(s_1) \oplus \mathit{ext}_{U_2\setminus V_2}(s_2) \in \\ \big\|\big(\big(\pi_{U_1\setminus V_1}(\sigma_{\mathit{isNull}(V_1)}\boldsymbol{\tau}(P_1))\big) \bowtie \big(\pi_{U_2\setminus V_2}(\sigma_{\mathit{isNull}(V_2)}\boldsymbol{\tau}(P_2))\big)\big)\big\|_{\mathit{triple}(G)}.$$

As t extends this tuple to $V_1 \cup V_2$ by *null*s, $t \in \|\boldsymbol{\tau}(\mathrm{JOIN}(P_1, P_2))\|_{\mathit{triple}(G)}$.

($\supseteq$) If $t \in \|\boldsymbol{\tau}(\mathrm{JOIN}(P_1, P_2))\|_{\mathit{triple}(G)}$ then there are disjoint $V_1, V_2 \subseteq U_1 \cap U_2$ and tuples t_1 and t_2 with $t_i \in \|\pi_{U_i\setminus V_i}\sigma_{\mathit{isNull}(V_i)}\boldsymbol{\tau}(P_i)\|_{\mathit{triple}(G)}$ such that t_1 and t_2 are compatible and t extends $t_1 \oplus t_2$ to $V_1 \cup V_2$ by *null*s. We then define solution mappings s_i by taking $s_i = \{v \mapsto t(v) \mid v \in U_i \text{ and } t(v) \text{ is not } \mathit{null}\}$, $i = 1, 2$. It follows that s_1 and s_2 are compatible and $\mathit{ext}_{U_i}(s_i) \in \|\boldsymbol{\tau}(P_i)\|_{\mathit{triple}(G)}$, $i = 1, 2$. By induction hypothesis, $\mathit{ext}_{U_i}(s_i) \in \|P_i\|_G$ and $s_i \in [\![P_i]\!]_G$, from which $s_1 \oplus s_2 \in [\![\mathrm{JOIN}(P_1, P_2)]\!]_G$ and $\mathit{ext}_{U_1\cup U_2}(s_1 \oplus s_2) = t \in \|\mathrm{JOIN}(P_1, P_2)\|_G$.

$P = \mathrm{OPT}(P_1, P_2, F)$. Denote $U_i = \mathit{var}(P_i)$, $i = 1, 2$, and $U = U_1 \cap U_2$. Recall that we assumed that the variables of F are in $U_1 \cup U_2$.

($\subseteq$) Let $t \in \|\mathrm{OPT}(P_1, P_2, F)\|_G$. Then there is $s \in [\![\mathrm{OPT}(P_1, P_2, F)]\!]_G$ such that $\mathit{ext}_{U_1\cup U_2}(s) = t$. By definition, either $s \in [\![\mathrm{FILTER}(\mathrm{JOIN}(P_1, P_2), F)]\!]_G$ or $s \in [\![P_1]\!]_G$ and, for all $s_2 \in [\![P_2]\!]_G$, either s and s_2 are incompatible or $F^{s\oplus s_2} \neq \top$. In the former case, we have $t \in \|\mathrm{FILTER}(\mathrm{JOIN}(P_1, P_2), F)\|_G$ and, by the induction hypothesis, $t \in \|\boldsymbol{\tau}(\mathrm{FILTER}(\mathrm{JOIN}(P_1, P_2), F))\|_{\mathit{triple}(G)}$. In the latter case, $\mathit{ext}_{U_1}(s) \in \|P_1\|_G$ and there is no solution mapping s_2 such that $\mathit{ext}_{U_2}(s_2) \in \|P_2\|_G$, s and s_2 are compatible and $F^{s\oplus s_2} = \top$. By induction hypothesis and Proposition 6, $\mathit{ext}_{U_1}(s) \in \|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)}$ and there is no s_2 such that $\mathit{ext}_{U_2}(s_2) \in \|\boldsymbol{\tau}(P_2)\|_{\mathit{triple}(G)}$, s and s_2 are compatible and $(\boldsymbol{\tau}(F))^{\mathit{ext}_{U_1\cup U_2}(s\oplus s_2)} = \top$. It follows that there is no s_2 such that $\mathit{ext}_{U_1\cup U_2}(s \oplus s_2) \in \|\boldsymbol{\tau}(\mathrm{JOIN}(P_1, P_2))\|_{\mathit{triple}(G)}$ and $(\boldsymbol{\tau}(F))^{\mathit{ext}_{U_1\cup U_2}(s\oplus s_2)} = \top$, that is no s_2 with $\mathit{ext}_{U_1\cup U_2}(s \oplus s_2) \in \|\sigma_{\boldsymbol{\tau}(F)}\boldsymbol{\tau}(\mathrm{JOIN}(P_1, P_2))\|_{\mathit{triple}(G)}$. This means that

$$\mathit{ext}_{U_1}(s) \in \|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)} \setminus \|\pi_{U_1}\sigma_{\boldsymbol{\tau}(F)}\boldsymbol{\tau}(\mathrm{JOIN}(P_1, P_2))\|_{\mathit{triple}(G)}.$$

Therefore, in either case, $\mathit{ext}_{U_1\cup U_2}(s) = t \in \|\boldsymbol{\tau}(\mathrm{OPT}(P_1, P_2, F))\|_{\mathit{triple}(G)}$.

($\subseteq$) Let $t \in \|\boldsymbol{\tau}(\mathrm{OPT}(P_1, P_2, F))\|_{\mathit{triple}(G)}$. If $t \in \|\sigma_{\boldsymbol{\tau}(F)}\boldsymbol{\tau}(\mathrm{JOIN}(P_1, P_2))\|_{\mathit{triple}(G)}$ then, by the induction hypothesis, $t \in \|\boldsymbol{\tau}(\mathrm{FILTER}(\mathrm{JOIN}(P_1, P_2), F))\|_{\mathit{triple}(G)}$ and so, $t \in \|\mathrm{FILTER}(\mathrm{JOIN}(P_1, P_2), F)\|_G \subseteq \|\mathrm{OPT}(P_1, P_2, F)\|_G$. Otherwise, there is some $t' \in \|\boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)} \setminus \|\pi_{U_1}\boldsymbol{\tau}(\mathrm{FILTER}(\mathrm{JOIN}(P_1, P_2), F))\|_{\mathit{triple}(G)}$ such that t extends t' by *null*s; in particular, all non-*null* components in t are in U_1. By the induction hypothesis, $t' \in \|P_1\|_G$. Thus, there is $s \in [\![P_1]\!]_G$ such that $t = \mathit{ext}_{U_1\cup U_2}(s)$. On the other hand, by the induction hypothesis and Proposition 6, there is $s_2 \in [\![P_2]\!]_G$ such that s and s_2 are compatible and $F^{s\oplus s_2} = \top$. It follows then that $s \in [\![\mathrm{OPT}(P_1, P_2, F)]\!]_G$ and therefore, $t = \mathit{ext}_{U_1\cup U_2}(s) \in \|\mathrm{OPT}(P_1, P_2, F)\|_G$.

This completes the proof of Theorem 7. ❑

Example 9. Consider the following query taken from the official SPARQL document[7] ('find the names of people who do not know anyone'):

$$
\begin{array}{l}
\text{SELECT } ?name \text{ WHERE } \{ \\
\quad ?x \text{ foaf:givenName } ?name. \\
\quad \text{OPTIONAL } \{ \ ?x \text{ foaf:knows } ?who \ \}. \\
\quad \text{FILTER (! BOUND } (?who) \text{)} \\
\},
\end{array}
$$

which is represented by the following graph pattern

$$
\textsc{Filter}(\textsc{Opt}(\{\, ?x \text{ foaf:givenName } ?name \,\},\ \{\, ?x \text{ foaf:knows } ?who \,\},\ \top),\ \neg bound(?who)). \quad (2)
$$

For the translation of the OPT operator, we first require to translate

$$
\textsc{Join}(\{\, ?x \text{ foaf:givenName } ?name \,\},\ \{\, ?x \text{ foaf:knows } ?who \,\}),
$$

which results in the following relational expression (we remove trivial projections and filters):

$$
\begin{array}{l}
\pi_{x,name}\sigma_{pred=\text{foaf:givenName}}\rho_{x/subj}\rho_{name/obj}\, triple \bowtie \\
\qquad \pi_{x,who}\sigma_{pred=\text{foaf:knows}}\rho_{x/subj}\rho_{who/obj}\, triple \ \cup \\
\mu_x(\pi_{name}\sigma_{isNull(x)}\pi_{x,name}\sigma_{pred=\text{foaf:givenName}}\rho_{x/subj}\rho_{name/obj}\, triple \bowtie \\
\qquad \pi_{x,who}\sigma_{pred=\text{foaf:knows}}\rho_{x/subj}\rho_{who/obj}\, triple) \ \cup \\
\mu_x(\pi_{x,name}\sigma_{pred=\text{foaf:givenName}}\rho_{x/subj}\rho_{name/obj}\, triple \bowtie \\
\qquad \pi_{who}\sigma_{isNull(x)}\pi_{x,who}\sigma_{pred=\text{foaf:knows}}\rho_{x/subj}\rho_{who/obj}\, triple).
\end{array}
$$

Since *triple* contains no *null*s, the above relational expression is clearly equivalent to (cf. Theorem 8):

$$
Q = \pi_{x,name}\sigma_{pred=\text{foaf:givenName}}\rho_{x/subj}\rho_{name/obj}\, triple \bowtie \pi_{x,who}\sigma_{pred=\text{foaf:knows}}\rho_{x/subj}\rho_{who/obj}\, triple
$$

Then $\textsc{Opt}(\{\, ?x \text{ foaf:givenName } ?name \,\},\ \{\, ?x \text{ foaf:knows } ?who \,\},\ \top)$ is translated into the following relational expression:

$$
Q \ \cup \ \mu_{who}(\pi_{x,name}\sigma_{pred=\text{foaf:givenName}}\rho_{x/subj}\rho_{name/obj}\, triple \setminus \pi_{x,name}Q).
$$

It can be verified (cf. Theorem 8) that this expression is in fact equivalent to

$$
\pi_{x,name}\sigma_{pred=\text{foaf:givenName}}\rho_{x/subj}\rho_{name/obj}\, triple \ ⟕ \ \pi_{x,who}\sigma_{pred=\text{foaf:knows}}\rho_{x/subj}\rho_{who/obj}\, triple.
$$

[7] http://www.w3.org/TR/sparql-features

Finally, the filter expression in graph pattern (2) is translated into $\neg\neg\mathit{isNull}(who)$, that is $\mathit{isNull}(who)$, and the graph pattern itself to

$$\sigma_{\mathit{isNull}(who)}\big(\pi_{x,\mathit{name}}\sigma_{\mathit{pred}=\text{foaf:givenName}}\rho_{x/\mathit{subj}}\rho_{\mathit{name}/\mathit{obj}}\ \mathit{triple}\ \ \sqsupset\!\bowtie \\ \pi_{x,\mathit{who}}\sigma_{\mathit{pred}=\text{foaf:knows}}\rho_{x/\mathit{subj}}\rho_{\mathit{who}/\mathit{obj}}\ \mathit{triple}\big).$$

whose projection onto $\{x\}$ can also be expressed as follows:

$$\pi_x\rho_{x/\mathit{subj}}\sigma_{\mathit{pred}=\text{foaf:givenName}}\ \mathit{triple}\setminus\pi_x\rho_{x/\mathit{subj}}\sigma_{\mathit{pred}=\text{foaf:knows}}\ \mathit{triple}$$

(informally, find those individuals who do not know anyone).

D Proof of Theorem 8

We begin by formalising the intuition behind the definition of ν:

Proposition 13. *Let P be a graph pattern. Then, for any RDF graph G and any solution mapping $s \in [\![P]\!]_G$, we have $\mathit{var}(P) \setminus \nu(P) \subseteq \mathit{dom}(s)$.*

Proof. The proof is by induction on the structure of P. The basis of induction is immediate: all variables in any BGP P are in the domain of any $s \in [\![P]\!]_G$. For the induction step, we consider the cases of SPARQL algebra operations:

$P = \text{FILTER}(P_1, F)$. If $s \in [\![\text{FILTER}(P_1, F)]\!]_G$ then $s \in [\![P_1]\!]_G$ and so, by the induction hypothesis, $\mathit{var}(P_1) \setminus \nu(P_1) \subseteq \mathit{dom}(s)$, from which the claim follows.

$P = \text{BIND}(P_1, v, c)$. If $s \in [\![\text{BIND}(P_1, v, c)]\!]_G$ then $v \in \mathit{dom}(s)$ and the restriction s' of s onto $\mathit{dom}(s) \setminus \{v\}$ is in $[\![P_1]\!]_G$. By the induction hypothesis, $\mathit{var}(P_1) \setminus \nu(P_1) \subseteq \mathit{dom}(s')$, whence the claim: $(\mathit{var}(P_1) \cup \{v\}) \setminus \nu(P_1) \subseteq \mathit{dom}(s') \cup \{v\} = \mathit{dom}(s)$.

$P = \text{UNION}(P_1, P_2)$. If $s \in [\![\text{UNION}(P_1, P_2)]\!]_G$ then either $s \in [\![P_i]\!]_G$, for $i = 1$ or $i = 2$. By the induction hypothesis, $\mathit{var}(P_i) \setminus \nu(P_i) \subseteq \mathit{dom}(s)$, for $i = 1$ or $i = 2$. Let $v \in \mathit{var}(P_1)$ but $v \notin (\mathit{var}(P_1) \setminus \mathit{var}(P_2)) \cup (\mathit{var}(P_2) \setminus \mathit{var}(P_1)) \cup \nu(P_1) \cup \nu(P_2)$. It follows that $v \in \mathit{var}(P_2)$ but $v \notin \nu(P_1)$ and $v \notin \nu(P_2)$. So, $v \in \mathit{var}(P_i) \setminus \nu(P_i)$, for both $i = 1$ and $i = 2$. By the mirror image argument, if $v \in \mathit{var}(P_2)$ then $v \in \mathit{var}(P_i) \setminus \nu(P_i)$, for both $i = 1$ and $i = 2$. Thus, $v \in \mathit{dom}(s)$.

$P = \text{JOIN}(P_1, P_2)$. If $s \in [\![\text{JOIN}(P_1, P_2)]\!]_G$ then there are $s_1 \in [\![P_1]\!]_G$ and $s_2 \in [\![P_2]\!]_G$ such that s_1 and s_2 are compatible and $s = s_1 \oplus s_2$. By the induction hypothesis, $\mathit{var}(P_i) \setminus \nu(P_i) \subseteq \mathit{dom}(s_i)$. Let $v \in \mathit{var}(P_i)$, for either $i = 1$ or $i = 2$ but $v \notin \nu(P_1) \cup \nu(P_2)$. Clearly, $v \in \mathit{dom}(s_i)$ and so, in either case $v \in \mathit{dom}(s)$.

$P = \text{OPT}(P_1, P_2, F)$. If $s \in [\![\text{OPT}(P_1, P_2, F)]\!]_G$ then either there is $s_1 \in [\![P_1]\!]_G$ and $s_2 \in [\![P_2]\!]_G$ such that s_1 and s_2 are compatible, $s = s_1 \oplus s_2$ and $F^s = \top$ or $s \in [\![P_1]\!]_G$ and, for all $s_2 \in [\![P_2]\!]_G$ either s and s_2 are incompatible or $F^{s \oplus s_2} \neq \top$. Let $v \in \mathit{var}(P_1)$ but $v \notin \nu(P_1) \cup \mathit{var}(P_2)$. By the induction hypothesis, for the two options above, we have $v \in \mathit{dom}(s_1) \subseteq \mathit{dom}(s)$ and $v \in \mathit{dom}(s)$, respectively. The choice $v \in \mathit{var}(P_2)$ but $v \notin \nu(P_1) \cup \mathit{var}(P_2)$ is impossible.

This completes the proof of Proposition 13. ❑

Theorem 8. *If* $\mathit{var}(P_1) \cap \mathit{var}(P_2) \cap (\nu(P_1) \cup \nu(P_2)) = \emptyset$ *then we can define*

$$\begin{aligned} \boldsymbol{\tau}(\textsc{Join}(P_1, P_2)) &= \boldsymbol{\tau}(P_1) \bowtie \boldsymbol{\tau}(P_2), \\ \boldsymbol{\tau}(\textsc{Opt}(P_1, P_2, F)) &= \boldsymbol{\tau}(P_1) ⟕_{\boldsymbol{\tau}(F)} \boldsymbol{\tau}(P_2), \end{aligned}$$

where $R_1 ⟕_F R_2 = \sigma_F(R_1 \bowtie R_2) \cup \mu_{U_2 \setminus U_1}(R_1 \setminus \pi_{U_1}(\sigma_F(R_1 \bowtie R_2)))$*, for relations* R_1 *and* R_2 *over* U_1 *and* U_2*, respectively.*

Proof. We clearly have $\|\boldsymbol{\tau}(P_1) \bowtie \boldsymbol{\tau}(P_2)\|_{\mathit{triple}(G)} \subseteq \|\boldsymbol{\tau}(\textsc{Join}(P_1, P_2))\|_{\mathit{triple}(G)}$ because $\boldsymbol{\tau}(P_1) \bowtie \boldsymbol{\tau}(P_2)$ is a component of the union in the definition of $\boldsymbol{\tau}(\textsc{Join}(P_1, P_2))$, with $V_1 = V_2 = \emptyset$. For the converse inclusion, consider a component of the union in the definition of $\boldsymbol{\tau}(\textsc{Join}(P_1, P_2))$:

$$\big((\pi_{U_1 \setminus V_1}(\sigma_{\mathit{isNull}(V_1)} \boldsymbol{\tau}(P_1))\big) \bowtie \big(\pi_{U_2 \setminus V_2}(\sigma_{\mathit{isNull}(V_2)} \boldsymbol{\tau}(P_2))\big)\big).$$

By Proposition 13 and Theorem 7, if $V_1 \cap \nu(P_1) \neq \emptyset$ then P_1 contains a variable that is always bound and so, $\|\sigma_{\mathit{isNull}(V_1)} \boldsymbol{\tau}(P_1)\|_{\mathit{triple}(G)} = \emptyset$, for any RDF graph G. Therefore, in this case the component is empty and can be removed from the union. If the condition in the theorem is satisfied then only $V_1 = \emptyset$ and $V_2 = \emptyset$ will give rise to a possibly non-empty component. Thus, $\|\boldsymbol{\tau}(\textsc{Join}(P_1, P_2))\|_{\mathit{triple}(G)} \subseteq \|\boldsymbol{\tau}(P_1) \bowtie \boldsymbol{\tau}(P_2)\|_{\mathit{triple}(G)}$.

The claim for $\textsc{Opt}(P_1, P_2, F)$ is then immediate from the claim for $\textsc{Join}(P_1, P_2)$ and the definition of the left join relational operation. ❑

E Proof of Proposition 11 and Theorem 12

Proposition 11. *For any R2RML mapping* $\mathcal{M}$ *and data instance* D*,* $t \in \|\mathsf{tr}_{\mathcal{M}}(\mathit{triple})\|_D$ *if and only if* $t \in \mathit{triple}(G_{D,\mathcal{M}})$*.*

Proof. ($\Rightarrow$) Let $t \in \|\mathsf{tr}_{\mathcal{M}}(\mathit{triple})\|_D$ then there is m in $\mathcal{M}$ such that $t \in \|\mathsf{tr}_m\|_D$. That is, the logical table of m matches the selection of tr_m (minus the $\neg\mathit{isNull}(v_i)$ operators) and the term maps (subject, predicate and object) of m match the *subj*, *pred*, and *obj* projections of tr_m. It follows that, by the procedure described in [10, Section 11], t is part of the *generated triples* of m and, therefore, belongs to $\mathit{triple}(G_{D,\mathcal{M}})$.

($\Leftarrow$) If a triple t is produced by $\mathcal{M}$, then there is triple map m with a predicate object map po that produces it by the procedure in [10, Section 11]. If this is the case, the logical table of m returns a tuple s, for which the values of the referenced columns in the term maps of m are not *null* and that generates t. By construction, m gives rise to tr_m in $\mathsf{tr}_{\mathcal{M}}(\mathit{triple})$ whose selection is the logical table of m and its projection matches the term maps of m. Thus, s produces t in $\|\mathsf{tr}_m\|_D$ and so, in $\|\mathsf{tr}_{\mathcal{M}}(\mathit{triple})\|_D$.

Theorem 12. *For any graph pattern* P*, R2RML mapping* $\mathcal{M}$ *and data instance* D*,*

$$\|P\|_{G_{D,\mathcal{M}}} = \|\mathsf{tr}_{\mathcal{M}}(\boldsymbol{\tau}(P))\|_D.$$

Proof. Follows from Theorem 7 and Proposition 11. ❑

Q	$LUBM_1$				$LUBM_9$			$LUBM_{20}$		$LUBM_{50}$		$LUBM_{100}$		$LUBM_{200}$	$LUBM_{500}$
	O	OB_H	OB_P	P	O	OB_H	P	O	P	O	P	O	P	O	O
q_1	2	8	29	1	3	97	1	3	2	3	1	3	1	3	2
q_2	2	25	11 137	19	3	2 531	256	5	633	12	11 312	16	30 593	36	88
q_3	1	6	86	9	2	78	158	2	345	3	932	2	2 087	63	12
q_4	13	7	19	14	15	44	164	15	347	22	943	27	2 093	24	22
q_5	16	12	4 451	10	22	98	158	16	343	21	945	32	2 182	28	23
q_6	455	27	32	21	5 076	411	317	11 610	730	28 674	4 321	58 968	10 781	123 578	434 349
q_7	5	21	34 005	10	6	429	157	6	343	32	939	8	2 171	8	9
q_8	726	195	95 875	80	760	917	192	748	374	755	974	796	2 131	820	855
q_9	60	972	168 978	78	668	189 126	857	1 594	5 093	3 734	5 175	7 466	12 125	15 227	44 598
q_{10}	2	6	126	9	3	97	158	2	344	2	930	2	2 134	3	2
q_{11}	4	5	58	10	6	43	160	6	349	8	939	11	2 093	18	44
q_{12}	3	4	19	15	4	70	236	4	567	4	937	3	2 114	5	5
q_{13}	6	4	67	8	7	40	157	9	348	11	935	14	2 657	38	58
q_{14}	91	20	24	15	1 168	329	287	2 782	694	6 444	1 866	13 524	4 457	29 512	92 376
q'_4	93	58	190	46	99	98	767	98	715	91	2 672	92	4 422	95	107
q''_4	108	21	35	63	122	72	719	117	3 394	140	3 994	115	9 179	108	127
q'_9	257	716	91 855	174	4 686	40 575	1 385	11 659	3 415	26 418	8 932	54 092	19 945	115 110	295 228
q''_9	557	951	65 916	102	6 093	178 401	1 214	14 934	3 038	34 178	9 296	67 123	19 705	151 376	356 176
q_2^{obg}	150	30	57 141	29	9 992	520	348	8 232	939	19 486	2 607	39 477	5 411	79 351	206 061
q_4^{obg}	6	7	241	25	31	40	273	29	735	32	1 699	7	3 969	7	494
q_{10}^{obg}	641	760	31 269	253	6 998	149 191	2 258	17 106	5 440	43 006	8 642	163 308	17 929	174 362	459 669
start up time	3.1s	13.6s	7.7s	3.6s	3.1s	80m33s	18s	3.1s	39.8s	3.1s	1m38s	3.1s	3m23s	3.1s	3.1s
loading time	10s	n/a	n/a	n/a	15s	n/a	n/a	26s	n/a	1m3s	n/a	1m56s	n/a	3m35s	10m17s

Table 2. Start up time, data loading time (in s) and query execution time (in ms) for LUBM in *Ontop* (O), OWL-BGP +Hermit (OB_H), OWL-BGP +Pellet (OB_P) and Pellet (P); OWL-BGP +Pellet times out on $LUBM_9$, OWL-BGP +Hermit on $LUBM_{20}$ and Pellet on $LUBM_{200}$.

F SQL Schema for LUBM

We evaluated the performance of *Ontop* using the OWL 2 QL version of the Lehigh University Benchmark LUBM [16]. We experimented with the data instances LUBM$_n$, for $n = 1, 9, 20, 50, 100, 200, 500$ (where n specifies the number of universities; note that LUBM$_1$ and LUBM$_9$ were used for experiments in [19]). The results of the experiments are shown in Table 2.

The LUBM data generator creates an OWL file with class and property assertions. As *Ontop* has been designed to work with relational databases, we had to store the class and property assertions in a database. The simplest solution to this issue would to use a generic schema, as usually done when storing triples in RDBMS backends. The two most common examples of this are (*a*) a schema with a single relation containing three attributes: *subj*, *pred*, *obj*, or (*b*) a schema with one unary relation for each class and one binary relation for each property. Such generic schemas, however, do not allow for efficient SQL translations in *Ontop* (or any other SPARQL-to-SQL system) because, for example, they require multiple and expensive (self)-join operations (if multiple properties are needed for an individual) and cause exponential blowup (due to class and property hierarchies).

In order to obtain efficient SQL queries, it is necessary that the schema follows standard best practices in the DB schema design, for example, normalisation. Usually, a normalised schema is obtained through a top-down approach that starts with the design of a conceptual model and follows by a translation of the conceptual model into a relational model. In the case of LUBM, we were not able to do so because the data generator was fixed. Instead, we studied the specification of the generator, extracted a model out of it, and created a schema based on that model. In particular, we identified disjoint classes (such as ub:UndergraduateStudent and ub:GraduateStudent) and functional properties (such as ub:name). When creating the schema we used the following principles:

- Classes on the top levels of hierarchies (e.g., ub:Student) have their own relations (e.g., students).
- The class membership in hierarchies is encoded using discriminating attributes (e.g., instances of ub:UndergraduateStudent and ub:GraduateStudent are also stored in the relation students for their superclass ub:Student, but are distinguished by the value of the discriminating attribute stype).
- Each functional property is included in the relation for its domain (e.g., property ub:name is the attribute name in students).
- Each 1-to-N and N-to-N property together with its attributes has a separate relation (e.g., relation takescourses represents ub:takesCourse).

The resulting schema consists of eleven relations:

- coauthors (data about the authors of a publication),
- courses (data about courses and the teachers assigned to those courses),
- departments (university departments),
- heads (heads of departments),
- publications (publications and the main author of a publication),

– ra (research assistants),
– researchgroups (research groups),
– students (data about students including their degrees and supervisors),
– ta (data about teaching assistants and courses they teach),
– takescourses (data about students and courses they take),
– teachers (data about teachers and their departments).

Instead of storing complete URIs, which were generated automatically by the LUBM data generator following a certain pattern, we store their components separately. For example, URIs of instances of ub:GraduateStudent are of the form

http://www.Department12.University54.edu/GraduateStudent22

where 54 refers to the university, 12 to the department in the university, and 22 to the graduate student in that department. We extracted those IDs (54, 12, 22 for this instance) and stored them in the respective attributes of the relations in the database (uniid, depid, id of the relation students, respectively). The obtained attributes together with the class names uniquely identify individuals, and so they form *primary keys* of the respective relations (e.g., attributes depid, uniid, stype, id constitute the primary key for relation students;[8] note that the discriminating attribute stype encodes the class name). *Foreign keys* are defined when the relation contained attributes that referred to the IDs of entities stored in a different relation.

In addition to the indexes that are defined by default on primary keys, we added indexes on attributes that would likely participate in join operations between relations. These were, mostly, the attributes that store IDs of entities from different relations. In total, we defined 39 additional indexes out of which 7 are composite. Note that in DB tuning, it usual that indexes are defined with respect to query workload. Since we did not proceed in this way, there are indexes that could be added to obtain a better performance for some of the queries we evaluated, and some indexes that could be removed since they are not relevant for the workload of the evaluation.

Finally, R2RML mappings were defined so that the (virtual) RDF graph entailed by the mappings would consist of all the triples that were initially used to populate the database.

To illustrate our rationale in more detail, we consider the case of ub:Student and its two disjoint subclasses, ub:GraduateStudent and ub:UndergraduateStudent. The class ub:Student is what is known as an abstract class in ER modelling, that is, a class that has no instances; only ub:GraduateStudent and ub:UndergraduateStudent have instances. In addition, each ub:Student instance has exactly one value for the properties ub:name, ub:telephone ub:degreeFrom, ub:emailAddress and ub:advisor (note that these properties are identified as functional on the basis of the specification of the data generator rather than the ontology). So, we defined the relation students shown in Fig. 1. Indexes in this relation include the (composite) index of the primary key (depid, uniid, stype, id) as well as indexes on the attributes degreeuniid, advisortype, advisorid. In addition, it contains a composite foreign key on the pair depid, uniid referring the attributes id, universityid in the relation departments.

[8] In the simplified example in Section 3.3 we do not have depid and uniid.

```
CREATE TABLE `students` (
  `depid` smallint(6) NOT NULL,
  `uniid` smallint(6) NOT NULL,
  `stype` tinyint(4) NOT NULL,
  `id` smallint(6) NOT NULL,
  `name` varchar(45) DEFAULT NULL,
  `degreeuniid` smallint(6) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `phone` varchar(255) DEFAULT NULL,
  `advisortype` tinyint(4) DEFAULT NULL,
  `advisorid` smallint(6) DEFAULT NULL,
  PRIMARY KEY (`depid`,`uniid`,`stype`,`id`),
  INDEX `idx_stud_1` (`degreeuniid`),
  INDEX `idx_stud_2` (`advisortype`),
  INDEX `idx_stud_3` (`advisorid`),
  CONSTRAINT `fk_students_1` FOREIGN KEY (`depid`, `uniid`)
                REFERENCES `departments` (`departmentid`,`universityid`)
);
```

Fig. 1. Relations for the subclasses ub:UndergraduateStudent and ub:GraduateStudent of ub:Student.

The relation students is mapped to the LUBM classes and properties using R2RML triple maps such as the one presented in Fig. 2. The mapping generates all triples in which the subject is a URI of an undergraduate student (indicated by the value 0 of the discriminating attribute stype).

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#> .

[ a  rr:TriplesMap ;
  rr:logicalTable    [ a rr:R2RMLView ;
                       rr:sqlQuery  "select * from students where stype=0"
                     ] ;
  rr:subjectMap    [ a            rr:SubjectMap, rr:TermMap ;
                     rr:class     ub:UndergraduateStudent ;
                     rr:template
      "http://www.Department{depid}.University{uniid}.edu/UndergraduateStudent{id}" ;
                     rr:termType  rr:IRI
                   ] ;
  rr:predicateObjectMap  [ a rr:PredicateObjectMap ;
                           rr:predicate  ub:telephone ;
                           rr:objectMap  [ a rr:ObjectMap, rr:TermMap ;
                                           rr:column    "phone" ;
                                           rr:termType  rr:Literal
                                         ]
                         ] ;
  rr:predicateObjectMap  [ a  rr:PredicateObjectMap ;
                           rr:predicate  ub:memberOf ;
                           rr:objectMap  [ a rr:TermMap, rr:ObjectMap ;
                                           rr:template
                                  "http://www.Department{depid}.University{uniid}.edu" ;
                                           rr:termType  rr:IRI
                                         ]
                         ] ;
  rr:predicateObjectMap  [ a rr:PredicateObjectMap ;
                           rr:predicate  ub:emailAddress ;
                           rr:objectMap  [ a rr:TermMap, rr:ObjectMap ;
                                           rr:column    "email" ;
                                           rr:termType  rr:Literal
                                         ]
                         ]
] .
```

Fig. 2. R2RML mapping for instances of ub:UndergraduateStudent.