Efficient SPARQL-to-SQL with R2RML mappings

Mariano Rodriguez-Muro, Martin Rezk

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 optimisation techniques from the logic programming and SQL optimisation 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 [9], 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 optimised 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 optimisation problem. There exist a number of systems and techniques related to this problem, such as the ones described in [10, 8, 29]. 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 optimisations on the Datalog program, including optimisation with respect to database metadata. We do this by adapting well known techniques for optimisation of logic programs and SQL query optimisation. Once the program has been optimised 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 optimisation. This is the technique implemented in the -ontop- system for OBDA, a mature open source system that is currently being used in a number of projects and that currently

http://ontop.inf.unibz.it/
outperforms other similar systems, sometimes by several orders of magnitude. -ontop- is available as a SPARQL endpoint, as an 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 optimisation techniques from logic programming the query optimisation; (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 6 by applying techniques from logic programming and SQL query optimisation. 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 [21] and Virtuoso 6.1 [12], 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 optimisations 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.

Virtuoso 7 also provides column-wise compressed storage[13], which may be much faster than traditional row stores. -ontop- (and any other general SPARQL-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. [10 8 29]. In addition one can also include here translations from SPARQL to Datalog [24 23 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 [28]. The work in [24] extends and improve the ones in [23 2] by modelling SPARQL 1.1 (under bag semantics, where duplicates are allowed), modeling non-safe queries [27] and modeling the W3C standard semantics to the SPARQL Optional. Since [23] 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 [22 23 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 [10 8] 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 [9]. Moreover, these approaches assume that the underlying relational DB is denormalized, and stores RDF terms. Also, the SPARQL generation technique presented in [10] 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. In [10 8] they present an ad-hoc evaluation with a much smaller dataset. The work described in [29] 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 [27]; however the optimisation 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.

3. Preliminaries

In this section we review the material required for the presentation of the core SPARQL-to-SQL technique and the opti-
Relational Algebra

3.1. Logic Programs

We start by reviewing basic notions from standard logic programming [20] needed in following sections.

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 \( \forall, \exists, \land, \rightarrow, \lnot \) are also called positive literals. The symbol \( \lnot \) will be used for default negation. A literal is either an atom or its negation. Literals that do not mention \( \lnot \) are said to be \( \lnot \)-free. Otherwise we say they are \( \lnot \)-literals.

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

\[
\forall \bar{x}: (l_0 \leftarrow l_1 \land \ldots \land l_m \land \lnot l_{m+1} \land \ldots \land \lnot l_n) \tag{1}
\]

where each \( l_i \) is a literal, \( l_0 \) is \( \lnot \)-free, and \( \bar{x} \) are all the variables mentioned in \( l_0 \ldots l_n \). \( l_0 \) is called the head of the rule. The set of literals \( \{l_1, \ldots, l_m\} \) is called the body of the rule. If the body is empty, then \( l_0 \leftarrow \), 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 (1), the sets \( \{l_0\}, \{l_1, \ldots, l_m\}, \{l_{m+1}, \ldots, l_n\} \) are referred to as head(\( r \)), pos(\( r \)) and neg(\( r \)) respectively. The set \( \text{lit}(\bar{t}) \) stands for \( \text{head}(\bar{t}) \cup \text{pos}(\bar{t}) \cup \text{neg}(\bar{t}) \).

As standard convention, (1) will be simply written as:

\[
l_0 \leftarrow l_1, \ldots, l_m, \lnot l_{m+1}, \ldots, \lnot l_n \tag{2}
\]

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

\[
\exists \bar{x}: l_1 \land \ldots \land l_m \tag{3}
\]

where \( l_1, \ldots, l_m \) are literals and \( \bar{x} \) are all the variables mentioned in \( l_1, \ldots, l_m \). The existential quantifier is usually omitted and comma is used often in lieu of the conjunction symbol \( \land \).

Stable model semantics for Logic Programs. In this section, we review the main concepts of stable model semantics [14].

For the sake of simplicity, we assume that logic rules are of the form (2) 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, \( M \), is consistent a subset of the Herbrand base. \( \square \)

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

- \( M \models l \), where \( l \) is a \( \lnot \)-free literal, iff \( l \in M \).
- \( M \models \phi_1 \land \phi_2 \), iff \( M \models \phi_1 \) and \( M \models \phi_2 \).
- \( M \models \lnot \phi \), iff it is not the case that \( M \models \phi \).
- \( M \models r \), where \( r \) is a ground rule of the form (2), iff \( l_0 \in M \) whenever \( M \models l_1 \land \ldots \land l_m \) and \( M \models \lnot (l_{m+1} \land \ldots l_n) \).

Given a \( \lnot \)-free program \( \Pi \), we write \( M \models \Pi \) if \( M \models r \) for every rule \( r \in \Pi \). In this case we say that \( M \) is a stable model (a.k.a. answer set) of \( \Pi \). It is known that every \( \lnot \)-free program \( \Pi \) has a unique least model [3]—a model \( M_0 \) such that for any other model \( N \) of \( \Pi, l \in M_0 \) implies \( l \in N \) for any \( l \in \mathcal{B} \).

To extend the definition of stable model (answer set) to arbitrary programs, take any program \( \Pi \), and let \( \bar{r} \) be a Herbrand interpretation in \( \mathcal{L} \). The reduct, \( \Pi(\bar{r}) \), of \( \Pi \) relative to a subset of the Herbrand Base \( \mathcal{S} \) is obtained from \( \Pi \) by first dropping every rule of the form (2) such that \( \{l_{m+1}, \ldots, l_n\} \cap \mathcal{S} = \emptyset \); and then dropping the \( l_{m+1}, \ldots, l_n \) literals from the bodies of all remaining rules. Thus \( \Pi(\Pi(\mathcal{S})) \) is a program without default negation.

Definition 2 (Stable Model). A Herbrand interpretation \( M \) is an answer set for \( \Pi \) if \( M \) is an answer set for \( \Pi(\mathcal{S}) \). \( \square \)

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

\[
p \leftarrow \lnot \ l \tag{4}
\]

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

Let \( \Pi \) be a program and \( q \) a query. (for simplicity we assume that \( q \) is a \( \lnot \)-free literal), we say that the program \( \Pi \)’s answer
Partial Evaluation. Partial evaluation is a logic programs technique in which, given a logic program \( \Pi \), one computes a new program \( \Pi' \) that represents the partial execution of \( \Pi \). This technique is used to either iteratively simplify the program to either compute the model of \( \Pi \) 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 [3]. Given a program \( \Pi \), intuitively the partial evaluation of \( \Pi \) produces a new program \( \Pi' \) that represents a pre-computation of \( \Pi \) needed to answer the goal \( G \). Observe that \( \Pi' \) should still provide sound and complete answers with respect to \( \Pi \), and that \( G \) should run more efficiently for \( \Pi' \) than for \( \Pi \).

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 [19] [16].

Definition 4 (substitution). A substitution \( \theta \) is a finite set of the form \( \{ x_i/t_i \} \), 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 \( \theta = \{ x_i/t_i \} \) be a substitution and \( E \) be an expression. Then \( E\theta \), the instance of \( E \) by \( \theta \), 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 \( S \) be a nonempty set of expressions (terms, atoms or a literals). A substitution \( \theta \) is called unifier of \( S \) if for every pair of expressions \( E_1, E_2 \in S \), it holds that \( E_1\theta = E_2\theta \).

Definition 7 (most general unifier). A unifier \( \theta \) of a set of expressions \( S \) is called most general unifier (mgu) of \( S \) if for every unifier \( \tau \) of \( S \), every binding in \( \theta \) is also in \( \tau \).

We note that computing a mgu for a set of expressions can be done in linear time [19]. Now we introduce the important notions of partial evaluation the we exploit later. All these notions where introduced in [18].

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 \( ← A_1, \ldots, A_m, A_k \) and \( C \) be a not-free rule of the form

\[ A ← B_1, \ldots, B_q \]

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

- \( A_m \) is an atom in \( G \), called the selected atom,
- \( \theta \) is a mgu of \( A_m \) and \( A \), and
- \( G' \) is the goal

\[ ← (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta \]

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 \( \theta \)

That is, a goal derivation is obtained by computing a mgu \( \theta \) between the selected atom and the head of another rule \( C \), replacing the selected atom with the body of \( C \), and applying \( \theta \) 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

\[ Q_1 ← Q_2 \]

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 \( \Pi \) 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 three is a resultant,
- The root node is \( G\theta_0 ← G_0 \), where \( G\theta_0 = G_0 = G \) (i.e., \( \theta_0 \) is the empty substitution),
- Let \( G\theta_0 \ldots \theta_i ← G_i \) be a node at depth \( i \geq 0 \) such that \( G_i \) has the form \( A_1, \ldots, A_m, \ldots, A_k \), and suppose that \( A_m \) is the selected atom. Then, for each input not-free rule \( A ← B_1, \ldots, B_q \) such that \( A_m \) and \( A \) are unifiable with mgu \( \theta_{i+1} \), the node has a child

\[ G\theta_0 \ldots \theta_{i+1} ← G_{i+1} \]

where \( G_{i+1} \) is derived from \( G_i \) and \( A_m \) by using \( \theta_{i+1} \), 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 $Q_0 \leftarrow Q_n$ at depth $i$, we say that the derivation of $Q_i$ has length $i$ with computed answer $\theta$, where $\theta$ is the restriction of $\theta_0, \ldots, \theta_i$ to the variables in $G$, i.e., $\theta$ 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 $\Pi$ be a program, $A$ an atom, and $T$ a SLD-tree for $\Pi \cup \{\leftarrow A\}$. Let $G_1, \ldots, G_r$ be a set of (non-root) goals in $T$ such that each non-failing branch of $T$ 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 $\Pi$. These resultants have the following form:
  \[ R_i = A\theta_i \leftarrow Q_i(i = 1, \ldots, r) \]
  where we have assumed $G_i \leftarrow Q_i$.

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

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 throughout the paper and we now introduce. Intuitively, a program $\Pi$ 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) $\Pi_0$ is not-free, and (iii) all the negative literals in $\Pi_i (0 < i \leq n)$ are only allowed to refer to predicates that are already defined in $\Pi_{i-1}$. Intuitively, in a stratified program $\Pi$, the intended model is obtained via a sequence of bottom-up derivation steps. In the first step, $\Pi$ is split into strata. The first stratum is a bottom part that does not contain negation as failure. Since this subprogram is positive, it has a unique stable model. Having substituted the values of the bottom predicates in the bodies of the remaining rules, $\Pi$ 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 $\Pi$. Further details can be found in [25].

**Definition 13 (Splitting Set [17]).** A splitting set for a program $\Pi$ is any set $U$ of literals such that for every rule $r \in \Pi$, if $\text{head}(r) \cap U \neq \emptyset$ then $\text{lit}(r) \subset U$. If $U$ is a splitting set for $\Pi$, we also say that $U$ splits $\Pi$. The set of rules $r \in \Pi$ such that $\text{lit}(r) \subset U$ is called the bottom of $\Pi$ relative to the splitting set $U$ and denoted by $b_U(\Pi)$. The subprogram $\Pi \setminus b_U(\Pi)$ is called the top of $\Pi$ relative to $U$.

**Definition 14 (Partial Evaluation w.r.t. a set of literals).** The partial evaluation of a program $\Pi$ 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
\[
(\text{pos}(r) \cap U) \subset X \text{ and } (\text{neg}(r) \cap U) \cap X = \emptyset
\]
put in $e_U(\Pi, X)$ all the rules $r'$ that satisfy the following property
\[
\begin{align*}
\text{head}(r') &= \text{head}(r) \\
\text{pos}(r') &= \text{pos}(r) \setminus U \\
\text{neg}(r') &= \text{neg}(r) \setminus U
\end{align*}
\]

**Definition 15 (Solution).** Let $U$ be a splitting set for a program $\Pi$. A solution to $\Pi$ 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.** Consider the following program $\Pi$:
\[
\begin{align*}
a &\leftarrow b, \text{not } c \\
b &\leftarrow c, \text{not } a \\
c &\leftarrow
\end{align*}
\]

The set $U = \{c\}$ splits $\Pi$; the last rule of $\Pi$ belongs to the bottom and the first two rules from the top. Clearly, the unique stable model for the bottom of $\Pi$ is $\{c\}$. The partial evaluation of the top part of $\Pi$ 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
\[
b \leftarrow \text{not } a
\]
(4)

The only stable model for $P$ can be obtained by adding the only stable model for (4), which is $\{b\}$, to the stable model for the bottom used in the evaluation process, $\{c\}$.

**Proposition 1.** Let $U$ be a splitting set for a program $\Pi$. A set $S$ of literals is a consistent stable model for $\Pi$ if and only if $S = X \cup Y$ for some solution $(X, Y)$ of $\Pi$ 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 form 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. \( t, t_1, t_2 \) denote tuples, \( c_1, c_2 \) denote attributes, \( v_1 \ldots v_n \) denote domain elements, \( p \) denotes a filter condition, \( \text{jn} \) denotes join condition of the form \( r_1 \cdot c_1 = r_2 \cdot c_2 \ldots r_3 \cdot c'_1 = r_2 \cdot c'_2 \) and the function \( \text{col}(r) \) 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 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 \( r_2 \).

\[
r_1 \times r_2 = \{ t_1, t_2 \mid t_1 \in r_1 \text{ and } t_2 \notin r_2 \}
\]

Difference (\( \setminus \)): This binary operator, written as, \( r_1 \setminus r_2 \), requires that the two relations involved must be union-compatible\( ^{\text{II}} \) 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 (\( \sigma \)): 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.

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

Natural join (\( \text{jn} \)): This is a binary operator written as, \( r_1 \text{jn} r_2 \), requires that the two relations involved must be union-compatible\( ^{\text{II}} \) that is, the two relations must have the same set of attributes. The result is the set of all combinations of tuples in \( r_1 \) and \( r_2 \) that are equal on their common attribute names.

\[
r_1 \text{jn} r_2 = \Pi_{\Delta}(\sigma_{\text{jn}}(r_1 \times r_2))
\]

Left join (\( \text{ acknow } \)): This is a binary operator written as, \( r_1 \text{ acknow} r_2 \), requires that the result is the set of all combinations of tuples in \( r_1 \) and \( r_2 \) that are equal on their common attribute names, in addition (loosely speaking) to tuples in \( r_1 \) that have no matching tuples in \( r_2 \).

\[
r_1 \text{ acknow} r_2 = (r_1 \setminus \text{know} r_2) \cup ((r_1 \times \Pi_{\text{col}(r)}(r_1 \setminus \text{know} r_2)) \times \{ \text{null}, \ldots, \text{null} \})
\]

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

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

Rename (\( \rho \)): 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.

Recall that every relational algebra expression is equivalent to a SQL query. Further details can be found in \( ^{\text{I}} \).

3.3. SPARQL

For formal purposes we will use the algebraic syntax of SPARQL similar to the ones in \( ^{23} \) and defined in the standard\( ^{4} \). 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: \( \text{I} \), denoting the IRIs, \( \text{B} \), denoting blank nodes, \( \text{L} \), denoting RDF literals; and \( \text{V} \), denoting variables.

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

\[
\text{BGP}(s, p, o)
\]

where \( s \in \text{I} \cup \text{B} \cup \text{V} \), \( p \in \text{I} \cup \text{V} \), and \( o \in \text{I} \cup \text{B} \cup \text{L} \cup \text{V} \). 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 \( ^{11} \), 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\( ^{3} \).

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 \( (V, P, G) \).

\[ \square \]

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

Semantics. Now we briefly introduce the formal set semantics of SPARQL as specified in \( ^{23} \) with the difference that updated the definition of the LeftJoin to match the published standard specifications. The result is a semantics which is more strict as the one in \( ^{24} \) 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.

\[ ^{3} \text{http://www.w3.org/TR/sparql-query/#sparqlAlgebra} \]

\[ ^{4} \text{http://www.w3.org/TR/rdf-sparql-query/#sparqlAlgebra} \]
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.

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 [23] 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 (I ∪ L ∪ {null}). We now provided the necessary definitions for this purpose.

Let \( T_{null} \) denote the following set (I ∪ L ∪ {null}).

**Definition 17 (Substitution).** A substitution, \( \theta \), is a partial function \( \theta : V \mapsto T_{null} \)

The domain of \( \theta \), denoted by \( \text{dom}(\theta) \), is the subset of \( V \) where \( \theta \) is defined. Here we write substitutions using postfix notation.

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

\[
x(\theta_1 \cup \theta_2) = \begin{cases} 
  x(\theta_1) & \text{if } x(\theta_1) \text{ is defined and } x(\theta_2) \in \text{null}, x(\theta_1) \in \text{null} \\
  x(\theta_2) & \text{if } x(\theta_2) \text{ is defined and } x(\theta_1) = \text{null} \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

**Definition 19 (Compatibility).** Two substitutions \( \theta_1 \) and \( \theta_2 \) are compatible when

1. for all \( x \in \text{dom}(\theta_1) \cap \text{dom}(\theta_2) \) it holds that \( x(\theta_1 \cup \theta_2) \neq \text{null} \).
2. for all \( x \in \text{dom}(\theta_1) \cap \text{dom}(\theta_2) \) 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 \( c \in B \cup I \cup L \). The valuation of \( R \) on a substitution \( \theta \) returns one of three values \( \{T, \bot, \epsilon\} \) and it is defined in Figure [2]

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

**Definition 21.** Let \( \Omega_1 \) and \( \Omega_2 \) be two sets of substitutions over domains \( D_1 \) and \( D_2 \) respectively. Then

\[
\begin{align*}
\Omega_1 \bowtie \Omega_2 & = \{ \theta_1 \cup \theta_2 \mid \theta_1 \in \Omega_1, \theta_2 \in \Omega_2 \text{ are compatible} \} \\
\Omega_1 \cup \Omega_2 & = \{ \theta \mid \exists \theta_1 \in \Omega_1 \text{ with } \theta = \theta_1^{\Omega_1 \cup \Omega_2} \text{ or } \\
& \hspace{1cm} \exists \theta_2 \in \Omega_2 \text{ with } \theta = \theta_2^{\Omega_1 \cup \Omega_2} \} \\
\Omega_1 \setminus \Omega_2 & = \{ \theta \mid \theta \in \Omega_1 \text{ and for all } \theta_2 \in \Omega_2 \text{, either } \theta \text{ and } \theta_2 \text{ are not compatible} \} \\
& \hspace{1cm} \text{or } \theta \text{ and } \theta_2 \text{ are compatible and } R(\theta \cup \theta_2) = \bot \}
\end{align*}
\]

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

**Definition 22 (Evaluation of Algebra Expressions).**

\[
\begin{align*}
\| \text{BGP}(t) \| & = \{ \theta \mid \text{dom}(\theta) = \text{vars}(P) \text{ and } t0 \in G \} \\
\| \text{Join}(P_1, P_2) \| & = \| P_1 \| \Join \| P_2 \| \\
\| \text{Union}(P_1, P_2) \| & = \| P_1 \| \cup \| P_2 \| \\
\| \text{LeftJoin}(P_1, P_2, R) \| & = \| \text{Filter}((P_1 \Join P_2, R)) \cup \| (P_1 \Join \neg R \Join P_2) \| \\
\| \text{Filter}(R, P) \| & = \{ \theta || P || \mid R\theta = T \}
\end{align*}
\]

where \( R \) is a FILTER expression.

**Definition 23 (Evaluation of Queries).** Let \( Q = (V, P, G) \) be a SPARQL query, and \( \theta \) a substitution in \( || P || \), then we call the tuple \( V(\{V \mid \text{vars}(P)\} \mapsto \text{null}) \) a solution tuple of \( Q \).

3.4. SPARQL to Datalog

The following technique allows to translate SPARQL queries (and RDF data) into a Datalog program. The technique was introduced in [23] 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 [23] 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 [23] instead of the one in [24].

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

**Example 2.** Consider the following SPARQL query \( Q \):

```
SELECT ?x ?z ?w WHERE 
```

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

```
  SELECT
  
  \[ T_1 \rightarrow BGP \]

  Join

  \[ T_2 \rightarrow BGP \]

  Join

  \[ T_3 \rightarrow BGP \]
```

Project
where $T_1$, $T_2$ and $T_3$ represent $(x,’knows’,y)$, $(x,’hasEmail’,z)$, and $(x,’hasWebpage’,w)$ 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 $\text{ans}_\text{stop}$ where ans stands for answer. Now, if we map the dependency graph of these new predicate symbols, we would get a tree as follows:

$$
\begin{align*}
\text{ans}_{\text{Project}} & \downarrow \\
\text{ans}_{\text{Join}_1} & \downarrow \\
\text{ans}_{\text{BGP}_1} & \downarrow \\
T_1 & \downarrow \\
\text{ans}_{\text{Join}_2} & \downarrow \\
\text{ans}_{\text{BGP}_2} & \downarrow \\
T_2 & \downarrow \\
\text{ans}_{\text{BGP}_3} & \downarrow \\
T_3 & \downarrow \\
\end{align*}
$$

The technique introduced in [24] would then produce the following Datalog program.

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

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 $\text{ans}_{\text{Join}_1}$ reflect this by making reference to the predicates $\text{ans}_{\text{BGP}_1}$, and $\text{ans}_{\text{Join}_2}$.

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 $\Pi_Q$ is defined as follows:

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

Where $\{ G \} = \{ \text{triple}(o, p, s) \mid (o, p, s) \in G \}$. 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. 3.

Intuitively, the $LT()$ operator disassembles complex filter expressions that includes boolean operators such as $\neg$, $\land$, $\lor$. 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:

- $E = \text{Bound}(v)$ is translated to $\text{not} v = \text{null}$,
- $E = \neg \text{Bound}(v)$ is translated to $v = \text{null}$,
- $E = \text{isBlank}(v)/\text{isIRI}(v)/\text{isLiteral}(v)$ 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 [23]. 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 $P’ = \text{LeftJoin}(P_1, P_2)$ of $P$ and for every variable $v$ occurring in $P$, the following condition holds: $v$ 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 $P’ = \text{Union}(P_1, P_2)$ of $P$ and for every variable $v$ occurring in $P’$, the following condition holds: $v$ occurs outside $P$ then if it occurs in both $P_1$ and $P_2$.

Proposition 2 (Soundness and Completeness [24]). Let $Q$ be a well-designed SPARQL query and let $\Pi_Q$ be the Datalog translation of $Q$. Then for each atom of the form $\text{answer}_Q(\delta)$ in the unique answer set $M$ of $\Pi_Q$, $\delta$ 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 $\text{answer}_Q$ in $M$. 

\[ \text{Figure 2: Valuation of R on a substitution } \theta \]
3.5. Datalog to SQL

Recall that safe Datalog with negation and without recursion is equivalent to relational algebra [28]. 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 be 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 3. 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 3. Let DB be a database composed by the table student with columns [id, name, course] (primary keys are underlined) and the table course with columns [id, name]. Let DB contain the following data:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>&quot;John&quot;</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>&quot;Mary&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

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

```sql
stud/28 rdf:type :Student ; :name "John" .
stud/21 rdf:type :Student ; :name "Mary" .
stud/28 :takes :course/1 .
stud/21 : takes : course/1 .
course/1 rdf:type :Course ; :name "SWT 101" .
```

The following R2RML mapping produces the desired triples:

```xml
_: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 :takes ;
    rr:objectMap [ rr:parentTriplesMap _:m2 ;
      rr:template { "stud/" + $_id } ;
      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 :takes ;
    rr:objectMap [ rr:parentTriplesMap _:m1 ;
      rr:template { "course/" + $_id } ;
      rr:joinCondition [ rr:child "ID" ;
      rr:parent "ID" ]].
```

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, (ii) 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 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.

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

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 [23] and introduced in Section 3.4. The original translation was developed in [23, 24] 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 optimise 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 [24]. To keep the grounding finite, we only allow functional terms that have 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 advise the reader to see Example 2 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 optimised 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 [23], 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 (ΠQ).** Let Q = (V, P, G) be a SPARQL query. The logic program of Q, denoted ΠQ, is defined as:

\[ ΠQ = \{ G \} \cup \tau(V, P) \]

where \( \{ G \} = \{ \text{triple}(s, p, o) \mid (s, p, o) \in G \} \) and \( \tau \), which is defined inductively in Figure 5, stands for the actual translation of the SPARQL algebra query to rules. Expressions are translated as follows:

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

The translation presented in Definition 27 deviates from [24] in that it exploits SQL built-ins in the generated rules. In particular, the differences are:

- Rule (4): In [24], 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 [24], the authors translate filter boolean expressions into a set of rules that is exponential in the number of \( \lor \) operators (c.f., LT operator). We encode this set in a single rule by leaving the boolean expression untouched. In [24] instead, the authors translate filter boolean expressions by adding facts that simulate filter evaluation, for instance, \( \text{equals}(x, x, \text{true}) \). 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 [24]. Hence, our variation is equivalent to the one in [24] 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 \( \text{ans}_1 \) 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 4.** 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 .  
  ?x :hasYear ?w; :hasGrade ?u } }
```

Using SPARQL algebra:

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

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

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

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.

```
   ans2  ans1
     ↓     ↓
   T1     T2
   \text{ans4} \text{ans5} \text{ans6} \text{ans7}
   \text{ans8} \text{ans9}
   \text{T3} \text{T4} \text{T5}
```

where each \( T_i \) represents \( \text{triple}(x, \text{rdf} : \text{type}, \text{Student}), \text{triple}(x, \text{hasName}, y), \text{triple}(x, \text{hasEnrolment}, z), \text{triple}(z, \text{hasYear}, w) \) and \( \text{triple}(z, \text{hasGrade}, u) \), 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 SQL is usually name-based (use column names) we apply standard ([11]/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 \[\text{Appendix B}\]. After this transformation, each rule body consists of a single atom since joins are made explicit with a distinguished atom \text{Join} and every boolean expression is added to a \text{Filter} atom. Now, we are ready to provide the relational algebra translation for a program.

Given a Datalog program \(\Pi_Q\), the operator \(\llbracket \cdot \rrbracket\) takes an atom and returns a relational algebra expression. We drop the subindex \(\Pi\) whenever it is clear from the context. We first define how to translate \text{ans} atoms. Then we define how to translate \text{triple} atoms and the distinguished \text{Join}, \text{Filter} and \text{LeftJoin} atoms. Intuitively, the \text{ans} atoms are the SQL projections, whereas \text{Join}, \text{LeftJoin}, and \text{Filter} are \(\Join\), \(\Join\Join\) and \(\sigma\) respectively.

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

\[
\llbracket\text{ans}(\bar{x})\rrbracket = \Pi_{\bar{z}}(\text{body}_{\text{ans}}(\bar{z})) \cup \cdots \cup \text{body}_{\text{ans}}(\bar{z}))
\]

where \(\text{body}_{\text{ans}}\) is the atom in the body of the \(j\)-th rule defining \text{ans}.

\[
\llbracket\text{triple}(\bar{z})\rrbracket = \Pi_z(\text{triple})
\]

\[
\llbracket\text{Join}(P_1(x_1), P_2(x_2), jn)\rrbracket = \llbracket P_1(x_1)\rrbracket \cap \llbracket P_2(x_2)\rrbracket
\]

\[
\llbracket\text{LeftJoin}(P_1(x_1), P_2(x_2), jn)\rrbracket = \llbracket P_1(x_1)\rrbracket \cup \llbracket P_2(x_2)\rrbracket - \llbracket \text{join}(\bar{z})\rrbracket
\]

\[
\llbracket\text{Filter}(P_1(x_1), E)\rrbracket = \sigma_E(\llbracket P_1(x_1)\rrbracket)
\]

\[\Box\]

Observe that in the previous definition, if there are \textit{null} constants (or any other constant) in the \text{ans} atoms, they are translated as statements of the form “null AS \(\bar{x}\)” in the projections.

**Example 5.** Let \(\Pi_Q\) be the Datalog program presented in Example 4. Then \(\llbracket\text{ans}_5(\bar{x})\rrbracket\) is as follows:

\[
\Pi_{\bar{z}}; \text{JOIN} J \ldots \text{JOIN} J \ldots \text{JOIN} J
\]

\[
\llbracket\text{Join}(\text{ans}_5(x_1), \text{ans}_1(x_2), \text{ans}_2(z_3))\rrbracket
\]

\[
\llbracket\text{LeftJoin}(\text{ans}_5(x_1), \text{ans}_1(x_2), \text{ans}_2(z_3))\rrbracket
\]

\[
\llbracket\text{Filter}(\text{ans}_5(x_1))\rrbracket
\]

Here we omit the definitions of the join, \text{leftjoin}, and \text{filter} conditions, and the projections to avoid distracting the reader from the core of the translation. The full definitions is given in Example 12 in \[\text{Appendix B}\].

\[\Box\]

**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 6.** Consider the following SPARQL algebra expression:

\[
\text{LeftJoin}(A(x,z), R(x,y), z > 0)
\]

Following the translation presented in [24], we would obtain a Datalog program of the following (simplified) form:

\[
\text{answer}_1(x) :- A(x,z), R(x,y), z > 0
\]

\[
\text{answer}_1(x) :- A(x,z), \text{not} \text{answer}_2(x,y,z)
\]

\[
\text{answer}_2(x,y,z) :- R(x,y), z > 0
\]

Although the semantics of this program is correct as far as Datalog is concern, the fact that the variable \(z\) does not occur in any non-boolean atom in the body of \text{answer}_2 is a problem when one tries to translate that into SQL.

\[\Box\]

As the original SPARQL-Datalog translations, we require well-designed queries [24]. 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 [24]—as shown by the following example:

**Example 7.** Consider the following SPARQL query:

\[
\text{SELECT} * \text{ WHERE} / \quad ( ?x1:a \ ?x3 ) \text{ OPTIONAL} / \quad ( ?x1:a :B ) \text{ OPTIONAL} / \quad ( ?x1:a :C ) \text{ FILTER} ?x3 < 1 \}
\]

Next, we show the relevant fragment of \(\Pi_Q\):

\[
\text{ans}_1(x_1) :- \text{LeftJoin}(\text{ans}_5(x_1), \text{ans}_1(x_2), \text{ans}_2(z_3), x_1 < 1)
\]

\[
\text{ans}_4(x_1) :- \text{triple}(x_1, \text{rdfs}\text{-type}', \text{'B}')
\]

\[
\text{ans}_5(x_1) :- \text{triple}(x_1, \text{rdfs}\text{-type}', \text{'C}')
\]
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.

\[
\Pi_{x_3} \exists y \exists z \exists a \exists b (\rho_{a,b,c}(x_3, y, z, a, b))
\]

\[
\Pi_{x_3} \exists y \exists z (\rho_{a,b,c}(x_3, y, z) \wedge c(z)
\]

Here we omit the definitions of \( a, f, c, \) and \( l \) 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, \( \Pi_Q \) 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.

**Theorem 1.** Let \( Q \) be an SQL-compatible SPARQL query, \( \Pi_Q \) the Datalog encoding of \( Q \), and \( \llbracket \Pi_Q(\vec{x}) \rrbracket \) the relational algebra statement of \( \Pi_Q \). Then it holds:

\[
\vec{r} \in \llbracket \Pi_Q(\vec{x}) \rrbracket \iff \Pi_Q \models \Pi_Q(\vec{r})
\]

**Proof.** See Appendix A.

### 5. Integrating R2RML mappings

In this section we present a translation of R2RML [9] 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 \( \text{triple} \) 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, \( \text{triple} \) becomes a defined predicate. The rules that define the \( \text{triple} \) 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 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 triple required by the R2RML semantics (see Section [3.6]).

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

**Example 8.** Consider the R2RML mapping from Example 3 minus the mappings related to courses, the following Datalog rules would capture their semantics:

\[
\text{triple}(\text{cc}(\text{stud}', \text{id}), \text{id}, \text{<Student>):} -
\]

\[
\text{stud(id, name, course), NotNull(id)}
\]

\[
\text{triple}(\text{cc}(\text{stud}', \text{id}), \text{name, name}): -
\]

\[
\text{stud(id, name, course), NotNull(id), NotNull(name)}
\]

where \( \text{cc} \) stands for a built-in function that is interpreted as the string concatenation operator.

Next, we define a function that allows us 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 \( \text{tr}(t) \) to denote i) a constant \( c \) if \( t \) is an constant node with value \( c \), ii) a variable \( t \) if \( t \) is a column reference with name \( v \), iii) \( \text{cc}(\vec{x}) \) if \( t \) is a URI template where \( \text{cc} \) 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, and let \( V_i \) the set of variables occurring in the term map nodes in \( m \). Then the mapping rules for \( m \), denoted \( \rho(m) \), are the following set of rules:

- For each class, \( \text{clss} \), of the subject map node, \( \text{subject}_m \), in \( m \)

\[
\text{triple}(\text{tr(subject}_m), \text{"rdf:type"}, \text{tr(clss)}):- 
\]

\[
\text{translated}_m, \text{translated}_m, \text{translated}_m \text{logical}_m, \text{NN}
\]

is in \( \rho(m) \)

- For each property (map node), \( \text{prop} \), in the mapping \( m \), and each object node, \( \text{obj} \), of \( \text{prop} \)

\[
\text{triple}(\text{tr(subject}_m), \text{tr(prop), tr(obj)}):- 
\]

\[
\text{translated}_m, \text{translated}_m, \text{translated}_m \text{logical}_m, \text{NN}
\]

is in \( \rho(m) \)

where \( \text{NN} \) is a conjunctions of atoms of the form \( \text{NotNull}(<x>) \) for each variable \( x \) appearing in the head of the corresponding rule, and \( \text{translated}_m \text{logical}_m \) is

i) \( A_1(x_1), \ldots, A_n(x_n) \) if the logical table is a base table or view with name \( A \) and arity \( n \), or

ii) \( A_1(x_1), \ldots, A_n(x_n), B(y) \), that is, a conjunction of table or view atoms, and boolean condition \( B(\vec{y}) \) if the logical table is an SQL query whose semantics can be captured by the body of a data log rule, or

iii) otherwise \( \text{Aux}(x_1, \ldots, x_n) \), 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 \( \text{translated}_m \text{logical}_m \) 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 $\text{Aux}$, 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

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

Now, we can now 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 sufficient 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

$$(s, p, o) \in G \text{ iff } \Pi_M \vdash \text{triple}(s, p, o):-$$

**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 ($\Pi_Q^M$ to SQL).** Let $Q$ be a query, $M$ an R2RML mapping, and let triple be defined in $\Pi_Q^M$. Then $\text{triple}(s, p, o)$ is defined next:

$$\text{triple}(s, p, o) = \begin{cases} [\text{body}_{\text{triple}}(z)]_j \cup \ldots \text{ If triple has} \\ n \geq 1 \text{ rules} \\ \text{U} [\text{body}_{\text{triple}}(z)]_a \text{ defining it in } \Pi_Q \\ 0 \text{ definitions} \end{cases}$$

where $\text{body}_{\text{triple}}$ is the body of the $j$-th rule defining it; and \text{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, $\Pi_Q$ the Datalog encoding of $Q$ and $M$, and $[\Pi_Q]$ the relational algebra statement of $\Pi_Q$. Then it holds:

$$i \in [\text{ans}_1(i)] \leftrightarrow \Pi_Q \models \text{ans}_1(i)$$

**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., $\text{cc}(\bar{v})$). 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. This is so because of the presence of several structural features in these SQL queries that are detrimental to performance. Given that these features are relevant to most SPARQL-to-SQL techniques and SQL query rewriters in general, and that they affect all modern DBMS engines, we will devote this section to their discussion.

#### 6.1. Conditions over 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 may appear in any location where SQL values are allowed, e.g., SELECT clauses or terms in boolean conditions of JOIN or WHERE clauses. For example,

**Example 9.** Consider the following SPARQL query asking for the URI and name of all the students.

\[\text{SELECT id FROM student} \]
\[\text{WHERE ':stud/'||ID = ':stud/22''} \]

or

\[\text{SELECT ':stud/'||v1.ID as X, name as Y} \]
\[\text{FROM student v1 JOIN student v2} \]
\[\text{ON ':stud/'||v1.ID = ':stud/'||v2.ID} \]

Techniques that translate SPARQL into SQL require such feature since in relational DBs objects are identified with SQL values, while in RDF objects are identified by URIs. Because of this, RDB to RDF mappings (such as R2RML) need to express how to construct these URIs through templates, which get translated into string concatenation operators in SQL. The issue arises in the most basic R2RML use cases, as illustrated in the example below:

**Example 9.** Consider the following SPARQL query asking for the URI and name of all the students.
SELECT * WHERE { ?x a :Student; :name ?y }

Let cc be a function to compute string concatenation and suppose we have the following R2RML mappings (in Datalog syntax).

triple(cc("stud/id",id),"rdf:type",":Student") :-
  stud(id, name)
triple(cc("stud/id",id),":name",name) :-
  stud(id, name)

A translation of this SPARQL query to SQL with these mappings would produce a query similar to the second SQL query presented in this section. This is also the case for our core technique, which would produce something similar to the following relational algebra expression:

\[ \Pi_{T_1, s \times T_2, o} \text{ as } Y (\forall_{T_1, s \times T_2, o} (\Pi_{\text{stud/id}} \ll A_{T_1, s}, \ "\text{rdf:type}" \ A_{T_1, p, \ "\text{Student}" \ A_{T_1, o} (\text{stud}),
\Pi_{\text{stud/id}} \ll A_{T_2, s}, \ "\text{name}" \ A_{T_2, p, \ "\text{name}" \ A_{T_2, o} (\text{stud})))) \]

Evidence that queries with this feature perform poorly has been reported for all major databases engines \[26, 27\] and in our own experiences. The reason for the bad performance is that query planners are not able to use indexes to evaluate the conditions over these functional terms and have to resource to expensive computations to evaluate the expressions.

At the same time, in the context of SPARQL-to-SQL system, almost any query with expressions over functional terms may be re-expressed in a semantically equivalent query where this feature does not appear. For example, the previous query can be re-expressed as:

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

This alternative queries are usually orders of magnitude more efficient \[26, 27\].

However, perform this transformations is not trivial in particular, when this feature appears in combination the rest of the issues discussed in this section. The systems/techniques we have investigated \[27, 13\] and Morpl \[1\] often fall back to SQL queries with conditions over functional terms that provide poor performance or return incorrect answers.

### 6.2. SQL Conditions over 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 appear in the SQL queries generated by SPARQL-to-SQL techniques for several reasons. The first possibility is simply because the original SPARQL query may have UNION operators. Second, due to the presence of multiple sources of data for a triple pattern in SPARQL query. The second reason is the most important one since in many techniques, e.g., our own technique and \[27\] (both without optimisations), can appear in trivial cases.

**Example 10.** Consider the SPARQL query from Example \[2\] 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 \[6\]

Some systems like Ultrawrap \[27\] reduce the number of queries in each UNION-subquery constructing subqueries for query patterns using only the mappings relevant for the property/class in the pattern, as in query (b) in Figure \[6\]. This simplification is called detection of unsatisfiable conditions in \[27\]. However, there are common scenarios where this kind of simplification is not enough, for example, when there exist multiple mappings for a given predicate/class (as in our example).

Multiple mappings per class/property are common in multiple scenarios. For example, when a property is used to associate different types of instances; for instance, a property like :name can be mapped over people, companies or many types of entities. Another case is in data integration, where two sources provide data for the same property, as in our example where two tables hold information about students. Last, a similar situation may appear if the rewriting system supports RDFS/OWL entailment regimes.

At the same time, simplifying these queries is in general non-trivial, partly because of the potential exponential brow that would be generated by simply expanding the unions, and partly because the issue is complicated by the presence of functional terms described in the previous section (as shown in Figure \[6\]). However, as with the previous feature, queries with UNION-subqueries can be transformed into equivalent SQL queries where there are no UNION-subqueries, for example in query (d) in Figure \[6\] which continues Example \[10\].

### 6.3. Redundant JOINs and SQO

A common feature of SQL queries generated from SPARQL is the presence of a large number of self-JOIN operations. This situation rises 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 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.p=':Student' AND v1.s=v2.s AND v2.p=':name'
)

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

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

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

Figure 6: 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.
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.

An example of this is the SQL query in Example [10] where we have two triple patterns, one for `rdf:type` and one for `:name`, which by the mappings get translated into two individual references the `stud` table.

Most of these JOIN operations could generally be detected to be redundant due to the presence of primary keys and foreign keys in the involved tables. Techniques to do this for SQL and logic programs are well known. However, performing this kind of optimisation in the presence of functional terms and subqueries is, again, non trivial and SQL query optimizers of modern DBMS are not able to generate efficient plans.

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 Optimisation (SQO) is a research area from the fields of Deductive Databases and SQL query optimisation where the main objective is to optimise queries by performing static and semantic analysis of queries. In the current section we will show how to use both to optimise 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 optimise 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 5.1.

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 \( \Pi^Q_M \), we compute a partial evaluation of the atom \( \text{ans}_1 \) in \( \Pi^Q_M \) 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 11. Consider the SPARQL query and R2RML mappings from Example [7a]. The partial evaluation would start with a root resultant of the form

\[
\text{ans}_1(x,y) :- \text{ans}_1(x,y)
\]

and would iteratively resolve resolve against \( \Pi^Q_M \) (see Figure [7a]). The progression of the partial evaluation is shown in Figure [7b].

Note how in the end of the partial evaluation we only two successful branches ending in the resultants \( r_{13} \) and \( r_{16} \), 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 [6](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 \( r_{13} \) in our example when it generates \( r_{14} \) and \( r_{15} \). Note that there are 4 candidate rules for the triple in \( r_{13} \), 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 \( r_{16} \) and \( r_{17} \). 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 \( r_{16} \), we have that only \( 6 \) unifies due to the presence of the constant `:name` and the functional symbol `cc`: `stud1"'/id)` (similar for \( r_{17} \)). 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 \( r_{13} \) where there are 2 rules that unify with the atom, and that branch \( r_{13} \) into \( r_{14} \) and \( r_{15} \). 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 \( r_{13} \) in our example when it generates \( r_{14} \) and \( r_{15} \), 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 \( r_{13} \) and \( r_{17} \). 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 semantic 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 $\leftarrow A_1, \ldots, \text{Join}(\ldots, A_m, \ldots), A_{k+1}$ 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) $\theta$ if the following conditions hold:

- $A_m$ is an atom in $G$, called the selected atom, $\theta$ is a mgu of $A_m$ and $A$, and
- $G'$ is the goal obtained from $G$ applying the substitution $\theta$.

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 $\Pi$ 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 $G\theta_0 = G$ (i.e., $\theta_0$ 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 $\text{mgu} \theta_{i+1}$, the node has a child $G\theta_i, \theta_2 \ldots, \theta_{i+1} \leftarrow G_{i+1}$

where $G_{i+1}$ is derived from $G_i$ and $A_m$ by using $\theta_{i+1}$, 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 [13] (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.

**Semantic Query Optimisation.** Semantic Query Optimisation (SQO) [7, 15] 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 optimisations that we found critical in obtaining the best performance in most situations and that are implemented in the -ontop- system. In particular, optimisation of queries with respect to keys and primary keys, and optimisation of queries with respect to boolean conditions. In both cases, these optimisations 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) $\Pi$. That is, a (primary) key over a relation $r$ defines a set of dependencies of the form

$$y_{i1} = y_{i2} \leftarrow r(\vec{x}, y_1), r(\vec{x}, y_2)$$

for each $y_{i1}, y_{i2}$ in $y_1$ and $y_2$, respectively.

For example, given a table $\text{stud}$ as in our previous example, the primary key on the first column, $id$ would generate the following equality generating dependencies:

$$x = x \leftarrow \text{stud}(x, y_1), \text{stud}(x, y_2) \quad (5)$$

$$y_1 = y_2 \leftarrow \text{stud}(x, y_1), \text{stud}(x, y_2) \quad (6)$$

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 $\Pi$). For example, given the query

$$\text{ans1}(id, \text{name}) :- \text{stud}(id, \text{name1}), \text{stud}(id, \text{name})$$
by chasing eg\textsuperscript{D} one obtains the query

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

which can be simplified to

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

\subsection*{Boolean conditions.} Another kind of optimisation 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 \textsc{filter} expressions and mappings involve constants. For example, consider the following SPARQL query asking for all people that attend either \textsc{NYC} or Columbia universities

\begin{verbatim}
SELECT ?x WHERE {
  FILTER (?y = :NYC || ?y = :Columbia)
}
\end{verbatim}

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

\[
\text{triple(cc(\"stud1\"),id,\"\:attends\"),\"\:Stanford\")} :- \text{stud1(id,name)}
\]

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

\[
\text{triple(cc(\"stud1\"),id,\"\:attends\"),\"\:Stanford\")} :- \text{stud1(id,name)}
\]

\[
\text{OR("\:Stanford" = "\:NYC", "\:Stanford" = "\:Columbia")}
\]

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 optimisation requires a partial evaluation engine that is aware of boolean logic, as well as the semantics of some built-in operators such as the \textsc{concat} operator, the \textsc{sparql} built-in functions and all the arithmetic operators. While this kind of optimisation is not theoretically very interesting, we have found that in \textsc{-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.

\section{Evaluation}

This section provides an evaluation of our \textsc{sparql-to-sql} technique implemented in \textsc{-ontop} and using \textsc{db2} and \textsc{my sql} as backends. We compared \textsc{-ontop} with two systems that offer similar functionality to \textsc{-ontop} (i.e., \textsc{sparql} trough \textsc{sql} and mappings): \textsc{Virtuoso RDF Views 6.1} (open source edition) and \textsc{D2RQ 0.8.1} Server over \textsc{my sql}. We also compare \textsc{-ontop} with three well known triple stores: \textsc{owlim 5.3}, \textsc{star dog 1.2} and \textsc{Virtuoso RDF 6.1} (Open Source). Another system that is relevant in this experiments is \textsc{ultrawrap} [27], however the sys-
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 flavours: as an API, as a plugin for Protege—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 querying 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 On-Text 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.

**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) evaluates the performance of query engines utilising 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 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-run 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 endpoints. All configuration files are available online.

Discussion. The results are summarized in Figure 10. 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.

http://protege.stanford.edu
http://www.openrdf.org

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

20
Next, we can see is that for BSBM in almost every case, the performance obtained with -ontop-’s SQL queries executed by MySQL or DB2 outperforms all other 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.

These experiments do not allow to fully see the benefit of the optimizations on SQL that is performed by -ontop-; this would require to be able to enable and disable them, and in -ontop- this is not possible at the moment. However, some observations are possible, in particular 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[6a]. This query expresses a total of 16 Join operations. When translated into SQL, -ontop- is able to generate the query in Figure[6b].

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.

In BSBM this optimization is weaker since queries are smaller and have fewer joins, however a trend pointing to this same observation can be seen. Consider the results for Q2, Q3 and Q4 in Figure[3].

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 formalisation 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, 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 optimisation. We will work in these directions in the near future.

Acknowledgements

We thank the -ontop- development team (J. Harri, T. Bagosi, M. Slusnys) for the help with the experiments. This work was supported by the EU FP7 project Opique (grant 318338) and UK EPSRC project ExODA (EP/H05099X).

References


Figure 8: Optimization by -ontop-

<table>
<thead>
<tr>
<th>Query</th>
<th>-ontop--MySQL</th>
<th>-ontop--DB2</th>
<th>OWLIM</th>
<th>Stardog</th>
<th>V. RDF</th>
<th>V. Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>3.22k</td>
<td>1.28 k</td>
<td>1.43k</td>
<td>1.42</td>
<td>23</td>
<td>45</td>
</tr>
<tr>
<td>Q2</td>
<td>1.40k</td>
<td>928</td>
<td>276</td>
<td>790</td>
<td>123</td>
<td>315</td>
</tr>
<tr>
<td>Q3</td>
<td>1.92k</td>
<td>1.12k</td>
<td>111</td>
<td>543</td>
<td>155</td>
<td>341</td>
</tr>
<tr>
<td>Q4</td>
<td>1.53k</td>
<td>955</td>
<td>295</td>
<td>669</td>
<td>140</td>
<td>265</td>
</tr>
<tr>
<td>Q5</td>
<td>27</td>
<td>72</td>
<td>2</td>
<td>29</td>
<td>14</td>
<td>48</td>
</tr>
<tr>
<td>Q6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Q7</td>
<td>9.78k</td>
<td>1.59k</td>
<td>541</td>
<td>400</td>
<td>101</td>
<td>316</td>
</tr>
<tr>
<td>Q8</td>
<td>13.99k</td>
<td>1.68k</td>
<td>3.22k</td>
<td>648</td>
<td>96</td>
<td>180</td>
</tr>
<tr>
<td>Q9</td>
<td>13.62k</td>
<td>1.01k</td>
<td>3.7k</td>
<td>1.99k</td>
<td>59</td>
<td>188</td>
</tr>
<tr>
<td>Q10</td>
<td>20.22k</td>
<td>2.04k</td>
<td>3.9k</td>
<td>610</td>
<td>228</td>
<td>305</td>
</tr>
<tr>
<td>Q11</td>
<td>20.75k</td>
<td>2.04k</td>
<td>3.52k</td>
<td>1.89k</td>
<td>1.66k</td>
<td>1.13k</td>
</tr>
<tr>
<td>Q12</td>
<td>11.34k</td>
<td>1.64k</td>
<td>5.99k</td>
<td>1.4k</td>
<td>1.25k</td>
<td>1.19k</td>
</tr>
<tr>
<td>QueryMix</td>
<td>44.19k</td>
<td>76.96k</td>
<td>2.91k</td>
<td>35.79k</td>
<td>9.21k</td>
<td>25.11k</td>
</tr>
</tbody>
</table>

Figure 9: 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 10: Query performance comparison summary. X axis = parallel clients, Y axis = Query Mixes per Hour (QMpH, higher is better)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size</th>
<th>System</th>
<th>Number of clients</th>
<th>Loading time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSBM</td>
<td>25M</td>
<td>onTop-MySQL</td>
<td>12,437</td>
<td>00:01:57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>onTop-DB2</td>
<td>12,506</td>
<td>00:02:12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OWLIM</td>
<td>6,455</td>
<td>00:12:21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stardog</td>
<td>15,751</td>
<td>00:18:41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtuoso RDF</td>
<td>9,823</td>
<td>00:36:43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtuoso Views</td>
<td>11,992</td>
<td>00:11:33</td>
</tr>
<tr>
<td></td>
<td>100M</td>
<td>onTop-MySQL</td>
<td>8,771</td>
<td>00:15:20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>onTop-DB2</td>
<td>7,873</td>
<td>00:49:45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OWLIM</td>
<td>3,238</td>
<td>01:13:04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stardog</td>
<td>9,311</td>
<td>01:43:10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtuoso RDF</td>
<td>6,665</td>
<td>02:21:46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtuoso Views</td>
<td>8,449</td>
<td>03:52:14</td>
</tr>
<tr>
<td></td>
<td>200M</td>
<td>onTop-MySQL</td>
<td>7,171</td>
<td>00:25:13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>onTop-DB2</td>
<td>5,442</td>
<td>00:24:38</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OWLIM</td>
<td>2,249</td>
<td>01:43:10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stardog</td>
<td>6,719</td>
<td>02:22:59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtuoso RDF</td>
<td>4,970</td>
<td>02:21:46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtuoso Views</td>
<td>5,888</td>
<td>03:52:14</td>
</tr>
<tr>
<td>FishMark</td>
<td>16.5M</td>
<td>onTop-MySQL</td>
<td>4,835</td>
<td>00:01:04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OWLIM</td>
<td>1,984</td>
<td>00:06:32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stardog</td>
<td>3,409</td>
<td>00:02:34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtuoso RDF</td>
<td>2,973</td>
<td>00:12:21</td>
</tr>
</tbody>
</table>

Table 1: Result summary for all systems and datasets
Appendix A. Proofs

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

$$\vec{t} \in [ans_Q(\vec{x})]^0 \Rightarrow \Pi_Q \models ans_Q(\vec{t})$$

Proof. From the definition of $\Pi_Q$ it is clear that it is a stratified program of the form:

$$\Pi_Q = \Pi_0 \ldots \Pi_n$$

Therefore it has a unique Herbrand model, and moreover, such model is the union of the unique models of each stratum $\Pi_i$. Recall that $\Pi_0$ 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 $\Pi_Q$. 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 $\Pi_k$— 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 $\Pi_k$ is to handle LeftJoin. Recall that the LeftJoin predicate is a syntactic sugar for the set of rules in Figure A.11. 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 strata.

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 ($\Pi_0$): Recall that $\Pi_0$ 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 $A(\vec{t}) \in M$. This model is computed via a sequence of bottom-up derivation steps, which apply the rules of $\Pi_0$ to the facts in $\Pi_0$ 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 $\Pi_Q$, $\Pi_0$, 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:

     $$ans_{\text{Union}}(\vec{x}_1) := ans_P(\vec{z}_1)$$
     $$\vdots$$
     $$ans_{\text{Union}}(\vec{x}_n) := ans_P(\vec{z}_n)$$

     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 $[ans_{\text{Union}}(\vec{x})]$ is defined as follows:

     $$\Pi_\ell \cdot \Pi_i \cdot \Pi_{\text{Union}}(\vec{z}) \cup \ldots \cup \Pi_i \cdot \Pi_{\text{Join}}(\vec{z})$$

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

     $$\vec{c} \in \Pi_i \Leftrightarrow \Pi_i \models ans_P(\vec{c})$$

     It follows that

     $$\vec{c} \in \Pi_{\text{Union}}(\vec{z}) \Leftrightarrow \Pi_i \models ans_P(\vec{z})$$

   * Suppose that $P$ has the form

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

     where $ans_{\text{P}}(\vec{w})$ 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 $\text{Filter}(ans_{\text{P}}(\vec{w}), E)$ represents the body

     $$ans_{\text{P}}(\vec{w}), E(\vec{w})$$

     By inductive hypothesis,

     $$\Pi_0 \models ans_{\text{P}}(\vec{t}), E(\vec{t}) \text{ iff } t \in ans_{\text{P}}(\vec{w}), E(\vec{w})$$

     It follows that

     $$\Pi_0 \models \Pi_0 \cdot \Pi_{\text{Filter}}(\vec{t}) \text{ iff } \vec{t} \in \Pi_{\text{Filter}}(\vec{z})$$

   * Now suppose that $P$ has the form

     $$\text{Join}(ans_{\text{P}}(\vec{w}_1), ans_{\text{P}}(\vec{w}_2), \ldots, \ldots)$$

     where $ans_{\text{P}}(\vec{w}_1)$ and $ans_{\text{P}}(\vec{w}_2)$ has 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 $\Pi_k$ has a unique answer set $M_k$
   * The set $U_k = \{ \text{lit}(r) \mid \text{head}(r) \in M_k \}$ forms a splitting set for $\Pi_Q$
   * By Proposition[1] it follows that $M$ is an answer set of $\Pi_Q$ iff

     $$M = M_k \cup M_{\text{op}}$$

     where $M_{\text{op}}$ is an answer set of the partial evaluation $e_U(\Pi_Q \setminus U_k, M_k)$.
Using the same proposition, it follows that $M_{bop}$ can be computed iteratively in the same way, that is, computing the model of the positive part of $e_1(\Pi_Q \setminus b_{ij}, M_b)$, and the continue splitting and computing the partial evaluations.

Let $M_b$ be the unique model of the positive part of $e_1(\Pi_Q \setminus b_{ij}, M_b)$.

Suppose that $ans_p(i) \in M_b$ and $ans_p$ is completely defined in $M_b$ in the sense specified above.

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

- $ans_p$ defines a Join
- $ans_p$ defines an Union
- $ans_p$ defines a 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 $ans_{\text{LeftJoin}}(P_1, P_2)$ is a Datalog translation of the fragment of a query of the form:

$$\text{LeftJoin}(P_1, P_2, E)$$

we can conclude $\Pi_Q$ contains rules of the form shown in Figure A.11.

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

$$M_b \models A(i) \iff M \models A(i) \iff i \in \{A(\overline{c})\}$$

for positive atoms, and analogously

$$M_b \models \text{not } A(i) \iff M \models \text{not } A(i) \iff i \notin \{A(\overline{c})\}$$

- Let $\text{body}_1^{\text{notLeftJoin}(P_1, P_2)}$ be the body of the $\text{not}$ -free rule (1) defining $\text{answer}_{\text{LeftJoin}}(P_1, P_2)$ above.
- Let $\text{body}_2^{\text{notLeftJoin}(P_1, P_2)}$ be the positive part of the body in rule (2) defining $\text{answer}_{\text{LeftJoin}}(P_1, P_2)$ above.
- Let $\text{body}_3^{\text{notLeftJoin}(P_1, P_2)}$ be the positive part of the body in rule (3) defining $\text{answer}_{\text{LeftJoin}}(P_1, P_2)$ above.
- Let $\text{body}_4^{\text{notLeftJoin}(P_1, P_2)}$ be the positive part of the body in rule (4) defining $\text{answer}_{\text{LeftJoin}}(P_1, P_2)$ above.

Thus from the previous facts, it follows that for every $\overline{c}$ satisfying rule (1) encoding the join part of the left join, there exist $\overline{c}_1, \overline{c}_2$ such that

$$M_b \models \text{ans}_{\text{LeftJoin}}(P_1, P_2)(\overline{c}) \iff M_b \models \text{body}_1^{\text{LeftJoin}(P_1, P_2)}(\overline{c}_1, \overline{c}_2)$$

Then, from the definition of translation we know that:

$$M_b \models \text{ans}_{\text{LeftJoin}}(P_1, P_2)(\overline{c}) \iff \text{if } M_b \models \text{body}_1^{\text{LeftJoin}(P_1, P_2)}(\overline{c}_1, \overline{c}_2)$$

And for every $\overline{c}$ satisfying either rule (2) encoding the difference part of the left join, there exist $\overline{c}_1$ such that

$$M_b \models \text{ans}_{\text{LeftJoin}}(P_1, P_2)(\overline{c}) \iff \text{if } M_b \models \text{body}_2^{\text{LeftJoin}(P_1, P_2)}(\overline{c}_1)$$

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

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

**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 $\Pi_M$ contains:

$\text{triple}(\text{tr}(s), \text{tr}(p), \text{tr}(o)) \iff \text{translated\_logic\_table}$
where translated_logic_table is the Datalog translation of lt. For the sake of simplicity assume that it is a table T with columns \( c_1 \ldots c_m \). Then \( \Pi_m \) has the following rule:
\[
\text{triple}(tr(x_s), tr(x_p), tr(x_o)) \rightarrow T(\vec{x})
\]
where \( x_s, x_p, x_o \) correspond to the columns in \( T \) as specified in \( m \). We make the assumption that there is a row in \( T \) where \( s, p, o \) are projected from it. It immediately follows that
\[
\Pi_m \models \text{triple}(tr(s), tr(p), tr(o)) \]

Appendix B. Datalog Normalisation

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 \( \text{triple}(t_1, t_2, t_3) \) in \( \Pi \), 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 \( \text{triple} \) atom. For instance:

\[
\text{triple}(x, a, \text{ Student})
\]

is replaced by

\[
\text{triple}(x, y, z), y = a, z = \text{ Student}
\]

This set of atoms will be denoted as \( fc \), and the whole set is replaced by a single \( \text{Filter} \) atom. In our example above, this would be:

\[
\text{Filter}(\text{triple}(x, y, z), \text{fc})
\]

where \( fc = y = a, z = \text{ Student} \).

- **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., \( \text{ans}_1(x) \rightarrow \text{ans}_2(x_1, y), \text{ans}_3(y, z) \) becomes:

\[
\text{ans}_1(x_1) \rightarrow \text{ans}_2(x_1, y), \text{ans}_3(x_2, z), x_1 = x_2
\]

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 \( \text{Join} \) atom, for instance:

\[
\text{Join}(\text{ans}_2(x, y), \text{ans}_3(y, z), jn)
\]

- **Variable Names:** Recall that \( \Pi_\varphi \) can be seen as a tree where the root is \( \text{ans}_1(\vec{x}) \) and the leaves are either \( \text{triple} \) atoms or boolean expressions. Then we:

1. Enumerate the \( \text{triple} \) atoms in the leaves from left to right: \( 1 \ldots n \).
2. For each of these \( \text{triple} \) 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 B.12 we show the Datalog program from Example 4 after the transformation explained above.

**Example 12.** Let \( \Pi_0 \) be the Datalog program presented in Example 4 Then \( \text{ans}_1(\vec{x}) \) is as follows:

\[
\Pi_{\text{ans}_1} \models \text{triple}(\text{Student}, p = \text{ hasName}), \text{triple}(\text{Student}, p = \text{ hasEnrolment}), \text{triple}(\text{Student}, p = \text{ hasYear}), \text{triple}(\text{Student}, p = \text{ hasGrade})
\]

where

\[
\begin{align*}
\text{ans}_1(x) &\rightarrow \text{ln} = [T_3.s = T_3.s] \\
\text{ans}_2(x) &\rightarrow \text{ln} = [T_1.s = T_2.s] \\
\text{ans}_3(x) &\rightarrow \text{ln} = [T_2.s = T_2.s] \\
\text{ans}_4(x) &\rightarrow \text{ln} = [T_3.o = T_4.o] \\
\text{ans}_5(x) &\rightarrow \text{ln} = [T_3.o = T_4.o] \\
\text{ans}_6(x) &\rightarrow \text{ln} = [T_3.s = T_3.s] \\
\text{ans}_7(x) &\rightarrow \text{ln} = [T_3.o = T_3.o] \\
\text{ans}_8(x) &\rightarrow \text{ln} = [T_3.s = T_3.s] \\
\text{ans}_9(x) &\rightarrow \text{ln} = [T_3.s = T_3.s] \\
\end{align*}
\]

Observe that there is a very close relation between the ans predicates and the \( \text{ans} \) statements, as well as Join, LeftJoin, and Filter conditions in both Datalog and SQL.
ans₁(T₂ₛ,T₂ₒ,T₃ₒ,T₅ₛ,T₃ₒ) :- LeftJoin(ans₂(T₂ₛ,T₂ₒ),
                                   ans₃(T₃ₛ,T₄ₛ,T₃ₒ,T₂ₛ = T₃ₛ)
ans₂(T₂ₛ,T₂ₒ) :- Join(ans₄(T₁ₛ), ans₅(T₂ₛ,T₂ₒ), T₁ₛ = T₂ₛ)
ans₃(T₃ₛ,T₃ₒ,T₄ₛ,T₅ₒ) :- Join(ans₆(T₁ₛ,T₃ₒ), ans₇(T₄ₛ,T₅ₒ,T₄ₒ), T₃ₒ = T₄ₒ)
ans₄(T₁ₛ) :- Filter(triple(T₁ₛ,T₁ₚ,T₁ₒ), T₁ₚ = a, T₁ₒ = Student)
ans₅(T₂ₛ,T₂ₒ) :- Filter(triple(T₂ₛ,T₂ₚ,T₂ₒ), T₂ₚ = hasName)
ans₆(T₁ₛ,T₃ₒ) :- Filter(triple(T₁ₛ,T₃ₚ,T₃ₒ), T₃ₚ = hasEnrolment)
ans₇(T₄ₛ,T₅ₒ,T₄ₒ) :- Join(ans₁₄(T₄ₛ,T₄ₒ), ans₁₅(T₅ₛ,T₅ₒ), T₄ₛ = T₅ₛ)
ans₁₄(T₄ₛ,T₄ₒ) :- Filter(triple(T₄ₛ,T₄ₚ,T₄ₒ), T₄ₚ = hasYear)
ans₁₅(T₅ₛ,T₅ₒ) :- Filter(triple(T₃ₛ,T₅ₚ,T₅ₒ), T₅ₚ = hasGrade) □

Figure B.12: Modified Πₑₐ.