BIROn - Birkbeck Institutional Research Online

STYPES: Nonrecursive Datalog Rewriter for Linear TGDs and Conjunctive Queries

Stanislav Kikot[1], Roman Kontchakov[2,✉], Salvatore Rapisarda[2], and Michael Zakharyaschev[2]

[1] University of Oxford, UK
staskikotx@gmail.com
[2] Birkbeck, University of London, UK
{roman,srapis01,michael}@dcs.bbk.ac.uk

Abstract. We present STYPES, a system that rewrites ontology-mediated queries with linear tuple-generating dependencies and conjunctive queries to equivalent nonrecursive datalog (NDL) queries. The main feature of STYPES is that it produces polynomial-size rewritings whenever the treewidth of the input conjunctive queries and the size of the chases for the ontology atoms as well as their arity are bounded; moreover, the rewritings can be constructed and executed in LOGCFL, indicating high parallelisability in theory. We show experimentally that Apache Flink on a cluster of machines with 20 virtual CPUs is indeed able to parallelise execution of a series of NDL-rewritings constructed by STYPES, with the time decreasing proportionally to the number of CPUs available.

1 Introduction

First-order (FO) query rewriting (or reformulation) lies in the core of ontology-based data access [24, 28] and data integration [22]. An abstract formulation of the problem is as follows: given a set $\mathcal{O}$ of tuple-generating dependencies (tgds), called here an *ontology*, and a conjunctive query $q(x)$ with a tuple x of answer variables, construct an FO-formula $q'(x)$, called an *FO-rewriting* of the ontology-mediated query (OMQ) $(\mathcal{O}, q(x))$, such that, for any data instance $\mathcal{D}$ and any tuple a of constants in it,

$$q(a) \text{ holds in every model of } \mathcal{O} \cup \mathcal{D} \quad \text{iff} \quad a \text{ is an answer to } q'(x) \text{ over } \mathcal{D}. \quad (1)$$

Thus, FO-rewriting is a reduction of the certain answer reasoning problem to database query evaluation, and so it can only be possible for OMQs given in carefully chosen languages. For ontology-based data access, the W3C standardised the *OWL 2 QL* profile of the Web Ontology Language *OWL 2* [23], which guarantees FO-rewritability of all OMQs with conjunctive queries (CQs) and ontologies in the *OWL 2 QL* profile [3]. Following the database tradition, more expressive, yet still ensuring FO-rewritability, languages have been suggested, including fragments of Datalog$^\pm$ such as linear tgds [11] (also known as atomic-body existential rules [5]) or sticky sets of tgds [12, 13]. We remind the reader that a *tuple-generating dependency* (*tgd*) is an FO-sentence of the form

$$\forall X \left(\gamma(X) \to \exists Y \, \gamma'(X', Y) \right), \quad (2)$$

where γ and γ' are conjunctions of atoms with variables $\boldsymbol{X}$ and $\boldsymbol{X}' \cup \boldsymbol{Y}$, respectively, and the variables of $\boldsymbol{X}'$ are contained in $\boldsymbol{X}$. A tgd is *linear* if $\gamma(\boldsymbol{X})$ is a single atom. As an illustration, we show how OMQs with linear tgds could be used in the system ETAP [10, 27] designed to answer natural language questions by translating them into SPARQL and executing—along with background knowledge—over RDF data extracted from texts.

Example 1. Suppose we have a data instance with atoms *purchased*(*john*, *BD51SMR*) and *Car*(*BD51SMR*) representing the sentence 'John purchased car *BD51SMR*'. To answer the question 'Which cars have been sold?' ETAP utilises the ontology rules

$$\forall xy \big[purchased(x,y) \rightarrow$$
$$\exists vz \, \big(Purchase(v) \wedge hasAgent_1(v,x) \wedge hasObject(v,y) \wedge hasAgent_2(v,z) \big) \big],$$

$$\forall vxyz \big[Purchase(v) \wedge hasAgent_1(v,x) \wedge hasObject(v,y) \wedge hasAgent_2(v,z) \rightarrow$$
$$\exists v' \, \big(Sale(v') \wedge hasAgent_1(v',z) \wedge hasObject(v',y) \wedge hasAgent_2(v',x) \big) \big],$$

where v and v' represent the acts of purchase and sale, respectively. These rules are beyond the limitations of *OWL 2 QL*; however, the knowledge they represent can also be captured by means of linear tgds with ternary predicates:

$$\forall xy \big[purchased(x,y) \rightarrow \exists z \, Purchase(x,y,z) \big],$$
$$\forall xyz \big[Purchase(x,y,z) \rightarrow Sale(z,y,x) \big],$$

which are sufficient for answering the CQ $\boldsymbol{q}(y) = \exists xz \, (Car(y) \wedge Sale(x,y,z))$ to obtain the answer *BD51SMR*. The resulting OMQ can be rewritten into the following FO-query (or equivalently, an SQL query), which can then be evaluated directly over the data:

$$\exists xz \, \big[Car(y) \wedge \big(Sale(x,y,z) \vee Purchase(z,y,x) \vee purchased(z,y) \big) \big].$$

FO-rewritability means, in particular, that OMQ answering is in the class AC^0 for *data complexity*, that is, as complex as standard database query evaluation. It has been discovered [21, 19], however, that the shortest FO-rewritings can be of superpolynomial size compared to the given CQ, which makes reduction (1) impractical. Further investigations [7–9] revealed that, by restricting the class of linear tgds to those of bounded arity and bounded existential depth and the class of CQs to those of bounded treewidth, one can achieve polynomial-size rewritings in the form of *nonrecursive datalog* (NDL) queries (rather than FO-formulas). In the context of Example 1, the following is a rewriting in the form of an NDL query with the goal predicate G:

$$G(y) \leftarrow Car(y) \wedge Sale'(x,y,z), \qquad Sale'(x,y,z) \leftarrow Sale(x,y,z),$$
$$Sale'(x,y,z) \leftarrow Purchase(z,y,x), \quad Sale'(x,y,z) \leftarrow purchased(z,y).$$

(NDL queries can also be thought of as SQL queries with view definitions.) The NDL-rewritings obtained in [8, 9] are *optimal* in the sense that the combined complexity of constructing and evaluating them is the same (LOGCFL) as the complexity of evaluating the underlying CQs [29, 15, 20]. (Note that the shortest rewritings into positive existential formulas in this case can still be of superpolynomial size, while polynomial-size

FO-rewritings exist iff $\text{LOGCFL}/\text{poly} \subseteq \text{NC}^1$, which is highly doubtful.) The experiments in [8] compared the size of the optimal NDL-rewritings constructed manually for a series of OMQs having the following fixed ontology:

$$\forall XY\big[P(X,Y) \to S(X,Y)\big], \qquad \forall X\big[A(X) \to \exists Y\, P(X,Y)\big],$$
$$\forall XY\big[P(X,Y) \to R(Y,X)\big], \qquad \forall X\big[B(X) \to \exists Y\, P(Y,X)\big],$$

with the rewritings produced by three known NDL-rewriters: Clipper [17], Presto [25] and Rapid [16], and established that, while the latter three grew exponentially, the former displayed linear growth.

The main distinguishing feature of the optimal NDL-rewritings from [9] is that, in theory, their evaluation can be performed by an efficient parallel algorithm because $\text{LOGCFL} \subseteq \text{NC}^2$ [26]. However, it has remained unclear whether such NDL-queries, which encode possibly exponentially large unions of CQs (UCQs), can be executed efficiently by a standard data management system, and whether the system can utilise the inherent parallelism of the NDL-rewritings.

The general aim of this paper is to give a positive answer to these questions. More specifically, we present a system STYPES that constructs an NDL-rewriting of any OMQ $(\mathcal{O}, \boldsymbol{q}(\boldsymbol{x}))$ with a set of linear tgds $\mathcal{O}$ and a CQ $\boldsymbol{q}(\boldsymbol{x})$. The rewritings are of polynomial size if the treewidth of CQs and the arity and the size of the chase for ontology atoms are bounded. Moreover, in this case, they can be constructed and executed in LOGCFL. Our rewriting algorithm takes a tree decomposition of the CQ as input in order to generate a plan for constructing an NDL-rewriting. Another input of the algorithm is a set of chases for the ontology atoms that occur in the rule bodies. In STYPES, the chases are constructed by the Graal library [4]. We use STYPES to produce NDL-rewritings for the OMQs from [8] mentioned above and then execute the rewritings by means of Apache Flink on a cluster of machines with 20 virtual CPUs. The experiments show that Flink is indeed able to parallelise execution of these NDL-rewritings, with the execution time decreasing proportionally to the number of CPUs available.

The plan of the remaining part of the paper is as follows. Section 2 provides definitions of the main notions we use and illustrates them with examples. Section 3 is the main technical contribution of this paper describing the NDL-rewriting algorithm implemented in STYPES. Section 4 presents and discusses our experimental results.
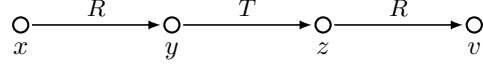
2 Preliminaries

Let Σ be a *relational schema*. By writing $P(\boldsymbol{x})$, for a predicate name P and an n-tuple $\boldsymbol{x}$ of variables (with possible repetitions), we mean that P is n-ary. Also, by writing $\gamma(\boldsymbol{x})$ we mean that $\boldsymbol{x}$ are the free variables of formula γ, where the tuple $\boldsymbol{x}$ contains no repetitions. When it is clear from the context, we use the set-theoretic notation for lists. In the series of examples below, we use relational schema Σ_0 with binary predicates R, S and T.

A *conjunctive query* (*CQ*) $\boldsymbol{q}(\boldsymbol{x})$ is an FO-formula of the form $\exists \boldsymbol{y}\, \varphi(\boldsymbol{x}, \boldsymbol{y})$, where φ is a conjunction of atoms $P(\boldsymbol{z})$ with predicate symbols from Σ and $\boldsymbol{z} \subseteq \boldsymbol{x} \cup \boldsymbol{y}$. The free variables, $\boldsymbol{x}$, are called the *answer variables* of the CQ, and a CQ without answer variables is called *Boolean*. We often regard CQs as *sets* of their atoms.

Example 2. For our running example, we use the Boolean CQ

$$\boldsymbol{q}_0 \;=\; \exists xyz \, \big(R(x,y) \wedge T(y,z) \wedge R(z,v) \big), \tag{3}$$

which can be depicted as follows:

$$
\begin{array}{ccccccc}
\circ & \xrightarrow{\;R\;} & \circ & \xrightarrow{\;T\;} & \circ & \xrightarrow{\;R\;} & \circ \\
x & & y & & z & & v
\end{array}
$$

An *ontology* $\mathcal{O}$ is a finite set of *linear tuple-generating dependencies* (*linear tgds*), that is, sentences of the form

$$\forall \boldsymbol{X} \, \big(\gamma(\boldsymbol{X}) \to \exists \boldsymbol{Y} \, \gamma'(\boldsymbol{X}', \boldsymbol{Y}) \big),$$

where γ is an atom and γ' a conjunction of atoms with predicate symbols from Σ and $\boldsymbol{X}' \subseteq \boldsymbol{X}$, for disjoint sets $\boldsymbol{X}$ and $\boldsymbol{Y}$ of variables ($\boldsymbol{Y}$ is possibly empty); as a convention, we will use capital letters for variables in tgds. When writing tgds, we omit both the universal and existential quantifiers. An *ontology-mediated query* (OMQ) $\boldsymbol{Q}(\boldsymbol{x})$ is a pair $(\mathcal{O}, \boldsymbol{q}(\boldsymbol{x}))$, where $\mathcal{O}$ is an ontology and $\boldsymbol{q}(\boldsymbol{x})$ a CQ. The variables $\boldsymbol{x}$ are called the *answer variables* of $\boldsymbol{Q}(\boldsymbol{x})$, and an OMQ without answer variables is called *Boolean*.

Example 3. For our running example, we use the Boolean OMQ $\boldsymbol{Q}_0 = (\mathcal{O}_0, \boldsymbol{q}_0)$, where the CQ $\boldsymbol{q}_0$ is given by (3) and $\mathcal{O}_0$ consists of the following linear tgds:

$$S(X, Z) \to R(X, Y) \wedge T(Y, Z), \tag{4}$$
$$T(X, Z) \to R(X, Y). \tag{5}$$

Note that X and Z are universally quantified and Y is existentially quantified in both tgds (4) and (5).

A *data instance* $\mathcal{D}$ over Σ is any finite set of ground atoms $P(\boldsymbol{a})$ with predicate symbols P from Σ. We denote by $\mathsf{ind}(\mathcal{D})$ the set of individual constants in $\mathcal{D}$. A tuple $\boldsymbol{a} \in \mathsf{ind}(\mathcal{D})^{|\boldsymbol{x}|}$ is a *certain answer* to an OMQ $\boldsymbol{Q}(\boldsymbol{x}) = (\mathcal{O}, \boldsymbol{q}(\boldsymbol{x}))$ over $\mathcal{D}$ if

$$\mathfrak{M} \models \boldsymbol{q}(\boldsymbol{a}), \qquad \text{for every model } \mathfrak{M} \text{ of } \mathcal{O} \cup \mathcal{D};$$

in this case we write $\mathcal{O}, \mathcal{D} \models \boldsymbol{q}(\boldsymbol{a})$. If $\boldsymbol{Q}$ is Boolean, then the *certain answer* to $\boldsymbol{Q}$ over $\mathcal{D}$ is 'yes' if $\mathfrak{M} \models \boldsymbol{q}$, for every model $\mathfrak{M}$ of $\mathcal{O} \cup \mathcal{D}$, and 'no' otherwise.

Canonical Models. An important property of tgds is the fact [1] that, for any $\mathcal{O}$ and $\mathcal{D}$, there is a (possibly infinite) *canonical* (or *universal*) *model* $\mathfrak{C}_{\mathcal{O},\mathcal{D}}$ such that

$$\mathcal{O}, \mathcal{D} \models \boldsymbol{q}(\boldsymbol{a}) \quad \text{iff} \quad \mathfrak{C}_{\mathcal{O},\mathcal{D}} \models \boldsymbol{q}(\boldsymbol{a}), \quad \text{for every CQ } \boldsymbol{q}(\boldsymbol{x}) \text{ and } \boldsymbol{a} \in \mathsf{ind}(\mathcal{D})^{|\boldsymbol{x}|}. \tag{6}$$

Such a canonical model can be constructed by a *chase procedure* that, intuitively, 'repairs' $\mathcal{D}$ with respect to $\mathcal{O}$ by extending the data instance with fresh *anonymous individuals* (labelled nulls) to witness existential quantifiers in tgds (though not necessarily in the most economical way).

Remark 1. There are variants of the chase procedure with various termination conditions. In our running examples, we follow a particular variant called the *Skolem chase*. However, STYPES can employ any type of chase as a black box. Obviously, the class of OMQs for which STYPES terminates depends on this choice. In particular, it would terminate for *weakly-acyclic linear* tgds if it were supplied with a restricted chase engine. Note that the ontology from Example 3.8 in [18], serving there as the main motivating example for introducing weakly-acyclic tgds, falls into this class. Moreover, since linear tgds satisfy the polynomial witness property [19], for each fixed OMQ, one could construct the chases only up to the depth that guarantees completeness of answers for the particular CQ, rather than for all CQs; cf. (6). Therefore, such a modification of our NDL rewriting algorithm would terminate on *all* OMQs with linear tgds.

Example 4. In the context of Example 3, consider a data instance $\mathcal{D}_0 = \{ S(a, b) \}$. Then, tgd (4) is *applicable* to $\mathcal{D}_0$ because h with $h\colon X \mapsto a$ and $h\colon Z \mapsto b$ is a homomorphism from the body $S(X, Z)$ of the tgd to $\mathcal{D}_0$. An *application* of the tgd produces a fresh anonymous individual e_0 for its existential quantifier and results in

$$\mathcal{D}_1 \; = \; \mathcal{D}_0 \, \cup \, \{ R(a, e_0), \; T(e_0, b) \}.$$

Next, tgd (5) is applicable to $\mathcal{D}_1$ via a homomorphism h with $h\colon X \mapsto e_0$ and $h\colon Z \mapsto b$ from the body $T(X, Z)$ of the tgd to $\mathcal{D}_1$. So, its application produces another fresh anonymous individual, e_1, and results in

$$\mathcal{D}_2 \; = \; \mathcal{D}_1 \, \cup \, \{ R(e_0, e_1) \}.$$

In this example, the chase terminates at step 2 because all tgds are satisfied in $\mathcal{D}_2$ (and so there are no defects to repair). In general, however, the chase does not have to terminate.

NDL-rewritings. A *datalog program*, Π, is a finite set of Horn clauses of the form

$$\forall \boldsymbol{z} \, (\gamma_0 \leftarrow \gamma_1 \wedge \cdots \wedge \gamma_m),$$

where each γ_i is an atom $P(\boldsymbol{y})$ with $\boldsymbol{y} \subseteq \boldsymbol{z}$ or an equality $(z = z')$ with $z, z' \in \boldsymbol{z}$. (As usual, we omit the universal quantifiers from clauses.) The atom γ_0 is the *head* of the clause, and $\gamma_1, \ldots, \gamma_m$ its *body*. All variables in the head must occur in the body, and $=$ can only occur in the body. The predicates in the heads of clauses in Π are *IDB predicates*, the rest (including $=$) *EDB predicates*. A predicate Q *depends* on P in Π if Π has a clause with Q in the head and P in the body. A program Π is a *nonrecursive datalog* (NDL) *program* if the (directed) *dependence graph* of the dependence relation is acyclic. The size $|\Pi|$ of Π is the number of symbols in it.

An *NDL query* is a pair $(\Pi, G(\boldsymbol{x}))$, where Π is an NDL program and G a predicate. A tuple $\boldsymbol{a} \in \mathsf{ind}(\mathcal{D})^{|\boldsymbol{x}|}$ is an *answer to* $(\Pi, G(\boldsymbol{x}))$ *over* a data instance $\mathcal{D}$ if $G(\boldsymbol{a})$ holds in the first-order structure with domain $\mathsf{ind}(\mathcal{D})$ obtained by closing $\mathcal{D}$ under the clauses in Π; in this case we write $\Pi, \mathcal{D} \models G(\boldsymbol{a})$. The problem of checking whether $\boldsymbol{a}$ is an answer to $(\Pi, G(\boldsymbol{x}))$ over $\mathcal{D}$ is called the *query evaluation problem*. It is known to be P-complete for combined complexity provided that the arity of predicates is bounded.

An NDL query $(\Pi, G(\boldsymbol{x}))$ is an NDL-*rewriting of an OMQ* $\boldsymbol{Q}(\boldsymbol{x}) = (\mathcal{O}, \boldsymbol{q}(\boldsymbol{x}))$ in case

$$\mathcal{O}, \mathcal{D} \models \boldsymbol{q}(\boldsymbol{a}) \quad \text{iff} \quad \Pi, \mathcal{D} \models G(\boldsymbol{a}), \qquad \text{for any } \mathcal{D} \text{ and any } \boldsymbol{a} \in \mathsf{ind}(\mathcal{D})^{|\boldsymbol{x}|}.$$

Every OMQ is known to have an NDL-rewriting [6, 11].

Example 5. For the OMQ $\boldsymbol{Q}_0$ in Example 3, the following program with the nullary goal predicate P_1 is an NDL-rewriting:

$$P_1 \leftarrow R(x, y) \wedge T(y, z) \wedge P_2(z), \tag{7}$$
$$P_1 \leftarrow S(x, z) \wedge P_2(z), \tag{8}$$
$$P_2(z) \leftarrow R(z, v), \tag{9}$$
$$P_2(z) \leftarrow S(z, Z), \tag{10}$$
$$P_2(z) \leftarrow T(z, Z). \tag{11}$$

In Section 3, we describe an algorithm for computing such NDL-rewritings.

H-completeness. A data instance $\mathcal{D}$ is said to be *H-complete* with respect to an ontology $\mathcal{O}$ if $\mathcal{D}$ validates all *full tgds* τ such that $\mathcal{O} \models \tau$ (full tgds have no existential variables). By default, STYPES produces an NDL-rewriting that is correct only for H-complete data instances (rather than over all data instances). An essentially NDL-rewriting that is correct for all data instances can be obtained by adding to it all full tgds τ with $\mathcal{O} \models \tau$. A proof of these statements is given in the extended version of [9].

3 NDL Rewriting

Our query rewriting algorithm takes two inputs, an ontology $\mathcal{O}$, which is a set of linear tgds, and a tree decomposition of a CQ $\boldsymbol{q}(\boldsymbol{x})$.

Tree Decompositions. A *bag* β for a CQ $\boldsymbol{q}(\boldsymbol{x})$ is a pair $(\boldsymbol{\nu}, \boldsymbol{\alpha})$, where $\boldsymbol{\nu}$ is a subset of the variables in the CQ and $\boldsymbol{\alpha}$ is a subset of query atoms with variables from $\boldsymbol{\nu}$; we refer to the two components as $\boldsymbol{\nu}(\beta)$ and $\boldsymbol{\alpha}(\beta)$, respectively.

A *tree decomposition* of a CQ $\boldsymbol{q}(\boldsymbol{x})$ is a pair (T, λ) consisting of a (rooted directed) tree $T = (V, E)$ and a map λ associating a bag to each vertex in T such that the following conditions hold:

- for any atom $R(\boldsymbol{z})$ in $\boldsymbol{q}$, there is a vertex $v \in V$ with $R(\boldsymbol{z}) \in \boldsymbol{\alpha}(\lambda(v))$;
- for any variable z in $\boldsymbol{q}$, the set $\{\, v \in V \mid z \in \boldsymbol{\nu}(\lambda(v)) \,\}$ is connected in T.

The *width* of the tree decomposition (T, λ) is $\max_{v \in V} |\boldsymbol{\nu}(\lambda(v))| - 1$. The *treewidth of* $\boldsymbol{q}$ is the minimum width over all tree decompositions of $\boldsymbol{q}$. Tree decompositions can be computed by *htd* [2], which is an open-source application written in C++[3].

---

[3] http://github.com/mabseher/htd

Example 6. A tree decomposition for q_0 in Example 2 can look as follows:
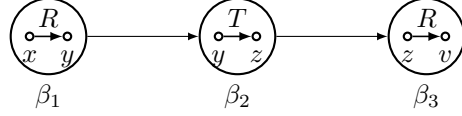
Atomic Canonical Models. The algorithm begins by extracting from the ontology $\mathcal{O}$ a set of atomic canonical models. Given a linear tgd $\gamma(\boldsymbol{X}) \to \gamma'(\boldsymbol{X'}, \boldsymbol{Y})$, we call $\gamma(\boldsymbol{X})$ a *generating atom* and the canonical model $\mathfrak{C}_{\mathcal{O}, \mathcal{D}}$ for $\mathcal{D} = \{\gamma(\boldsymbol{X})\}$ an *atomic canonical model* for $\mathcal{O}$, where (somewhat abusing notation) we treat $\boldsymbol{X}$ as a tuple of individual constants. We assume that $\mathcal{O}$ is fixed and denote the atomic canonical model by $\mathfrak{C}_{\gamma(\boldsymbol{X})}$. The implementation of the algorithm in STYPES uses the *Graal* [4][4] library to construct the atomic canonical models for $\mathcal{O}$ by chasing its generating atoms.

Example 7. The ontology $\mathcal{O}_0$ in Example 3 has two generating atoms, $S(X, Z)$ and $T(X, Z)$, and the following atomic canonical models:

generating atom	atomic canonical model
$S(X, Z)$	$T(e_0, Z), R(e_0, e_1), R(X, e_0), S(X, Z)$
$T(X, Z)$	$R(X, e_0), T(X, Z)$

where e_0 and e_1 are the anonymous individuals; see Example 4.

3.1 Bag Types

We use *term types* to indicate how query variables can be mapped into canonical models $\mathfrak{C}_{\mathcal{O}, \mathcal{D}}$ for possible data instances $\mathcal{D}$. The *non-anonymous* term type ε is used when a variable is mapped to an individual constant from $\mathcal{D}$. An *anonymous* term type is a pair of the form $(\gamma(\boldsymbol{X}), e)$, where $\gamma(\boldsymbol{X})$ is a generating atom and e an anonymous individual from the atomic canonical model $\mathfrak{C}_{\gamma(\boldsymbol{X})}$ for $\gamma(\boldsymbol{X})$. For a given $\gamma(\boldsymbol{X})$, we denote by $\mathfrak{T}_{\gamma(\boldsymbol{X})}$ the set of all term types of the form $(\gamma(\boldsymbol{X}), e)$ and ε. In the sequel, we silently assume that term types of all answer variables are always ε.

Example 8. As follows from Example 7, $\mathcal{O}_0$ has four term types: ε, $(S(X, Z), e_0)$, $(S(X, Z), e_1)$ and $(T(X, Z), e_0)$.

A *partial type* $\boldsymbol{s}$ is a map that assigns term types to a subset of query variables. The domain of $\boldsymbol{s}$ is denoted by $\mathrm{dom}(\boldsymbol{s})$. Given a partial type $\boldsymbol{s}$, we denote by $var(\boldsymbol{s})$ the tuple of variables that contains, for $z \in \mathrm{dom}(\boldsymbol{s})$,

the variable z, if $\boldsymbol{s}(z) = \varepsilon$,   and   the variables $\boldsymbol{X}^z$, if $\boldsymbol{s}(z) \in \mathfrak{T}_{\gamma(\boldsymbol{X})} \setminus \{\varepsilon\}$,

where, for a tuple $\boldsymbol{X} = (X_1, \ldots, X_n)$ of variables and a decoration k, we denote by $\boldsymbol{X}^k$ the tuple $(X_1^k, \ldots, X_n^k)$ of variables in which every component is decorated by k.

Given a bag $\beta = (\boldsymbol{\nu}, \boldsymbol{\alpha})$ for $\boldsymbol{q}$, we say that a partial type $\boldsymbol{s}$ is a *bag type for* β if $\boldsymbol{\nu}$ is the domain of $\boldsymbol{s}$ and, for every atom $R(\boldsymbol{z}) \in \boldsymbol{\alpha}$, one of the following applies:

---

[4] `http://graphik-team.github.io/graal`

(d) $s(z) \subseteq \{\varepsilon\}$ (in which case the variables of the atom are mapped to the individuals in the data instance, and the atom itself is in the data instance);

(b) $s(z) \subseteq \mathfrak{T}_{\gamma(\boldsymbol{X})}$, for some (uniquely determined) $\gamma(\boldsymbol{X})$, with $\boldsymbol{z}_\varepsilon \neq \emptyset$ and $\boldsymbol{z} \setminus \boldsymbol{z}_\varepsilon \neq \emptyset$, and there is a *grounding function* $g \colon \boldsymbol{z}_\varepsilon \to \boldsymbol{X}$ such that $R(\boldsymbol{c}) \in \mathfrak{C}_{\gamma(\boldsymbol{X})}$, where

$$\boldsymbol{z}_\varepsilon = \{\, z \in \boldsymbol{z} \mid s(z) = \varepsilon \,\} \quad \text{and} \quad \boldsymbol{c}(z) = \begin{cases} g(z), & \text{if } s(z) = \varepsilon, \\ e, & \text{if } s(z) = (\gamma(\boldsymbol{X}), e) \end{cases}$$

(in which case the variables in $\boldsymbol{z}_\varepsilon$ are mapped to the individuals in the data instance, whereas $\boldsymbol{z} \setminus \boldsymbol{z}_\varepsilon$ are mapped to the anonymous individuals, that is, the atom is on the boundary of the data instance and the anonymous part of the chase);

(i) $s(z) \subseteq \mathfrak{T}_{\gamma(\boldsymbol{X})} \setminus \{\varepsilon\}$, for some (uniquely determined) $\gamma(\boldsymbol{X})$, and $R(\boldsymbol{c}) \in \mathfrak{C}_{\gamma(\boldsymbol{X})}$, where $\boldsymbol{c}(z) = e$ for z with $s(z) = (\gamma(\boldsymbol{X}), e)$ (in which case all variables are mapped to the anonymous individuals, that is, the atom is in the interior of the anonymous part of the chase).

Given a bag type s for a bag $\beta = (\boldsymbol{\nu}, \boldsymbol{\alpha})$, we denote by $\equiv_{\boldsymbol{\nu}}$ the smallest equivalence relation on $\boldsymbol{\nu}$ such that $z \equiv_{\boldsymbol{\nu}} z'$ if $s(z) \neq \varepsilon$, $s(z') \neq \varepsilon$ and z and z' occur in $R(\boldsymbol{z})$ and $R'(\boldsymbol{z}')$, respectively, such that the two atoms share a variable z'' with $s(z'') \neq \varepsilon$. For a variable z, we denote by $[z]$ its $\equiv_{\boldsymbol{\nu}}$-equivalence class; also, for a set $\boldsymbol{z}$ of variables occurring in an atom from $\boldsymbol{\alpha}$ with $s(\boldsymbol{z}) \subseteq \mathfrak{T}_{\gamma(\boldsymbol{X})} \setminus \{\varepsilon\}$, let $[\boldsymbol{z}]$ be the $\equiv_{\boldsymbol{\nu}}$-equivalence class of some (equivalently, any) $z \in \boldsymbol{z}$.

The `MakeAtoms` function in the code of STYPES produces the formula $\mathrm{At}^s(var(s))$ by mapping each atom $R(\boldsymbol{z})$ in $\boldsymbol{\alpha}$ to

(d′) $R(\boldsymbol{z})$ if $s(\boldsymbol{z}) \subseteq \{\varepsilon\}$;

(b′) $\gamma(\boldsymbol{X}^{[\boldsymbol{z} \setminus \boldsymbol{z}_\varepsilon]}) \wedge \left(\bigvee\limits_{\substack{g \colon \boldsymbol{z}_\varepsilon \to \boldsymbol{X} \\ \text{is a grounding function}}} \left[\bigwedge\limits_{z \in \boldsymbol{z}_\varepsilon \text{ and } g(z) = X} (z = X^{[\boldsymbol{z} \setminus \boldsymbol{z}_\varepsilon]}) \right] \right)$
        if $s(\boldsymbol{z}) \subseteq \mathfrak{T}_{\gamma(\boldsymbol{X})}$ but neither $s(\boldsymbol{z}) \subseteq \mathfrak{T}_{\gamma(\boldsymbol{X})} \setminus \{\varepsilon\}$ nor $s(\boldsymbol{z}) \subseteq \{\varepsilon\}$;

(i′) $\gamma(\boldsymbol{X}^{[\boldsymbol{z}]})$ if $s(\boldsymbol{z}) \subseteq \mathfrak{T}_{\gamma(\boldsymbol{X})} \setminus \{\varepsilon\}$.

We also add to the formula $\mathrm{At}^s(var(s))$ the equalities $X^{[z]} = X^z$, for all $z \in \mathrm{dom}(s)$ such that $s(z) \in \mathfrak{T}_{\gamma(\boldsymbol{X})} \setminus \{\varepsilon\}$ and $X \in \boldsymbol{X}$.

Example 9. There are four possible bag types for $\beta_3 = (\{z, v\}, \{R(z, v)\})$. The bag type $t_3 = \{z \mapsto \varepsilon, v \mapsto \varepsilon\}$ trivially gives rise to the following At-formula, see **(d′)**:

$$\mathrm{At}^{t_3}(z, v) = R(z, v).$$

For the bag type $t_4 = \{z \mapsto \varepsilon, v \mapsto (T(X, Z), e_0)\}$, we have neither $t_4(z, v) \subseteq \{\varepsilon\}$ nor $t_4(z, v) \subseteq \mathfrak{T}_{T(X, Z)} \setminus \{\varepsilon\}$, and the only grounding function is $g \colon z \mapsto X$. Thus, by **(b′)**, we obtain

$$\mathrm{At}^{t_4}(X^v, Z^v, z) = T(X^{[v]}, Z^{[v]}) \wedge (z = X^{[v]}) \wedge (X^{[v]} = X^v) \wedge (Z^{[v]} = Z^v).$$

For the bag type $\boldsymbol{t}_5 = \{z \mapsto \varepsilon,\, v \mapsto (S(X,Z), e_0)\}$, the At-formula is constructed similarly, with $S(X,Z)$ in place of $T(X,Z)$.

Finally, for the bag type $\boldsymbol{s} = \{z \mapsto (S(X,Z), e_0),\, v \mapsto (S(X,Z), e_1)\}$, we have $\boldsymbol{s}(z,v) \subseteq \mathfrak{T}_{S(X,Z)} \setminus \{\varepsilon\}$. So, by **(i′)**, we obtain the following At-formula:

$$\mathsf{At}^{\boldsymbol{s}}(X^v, X^z, Z^v, Z^z) = S(X^{[v,z]}, Z^{[v,z]}) \wedge$$
$$(X^{[v,z]} = X^v) \wedge (X^{[v,z]} = X^z) \wedge (Z^{[v,z]} = Z^v) \wedge (Z^{[v,z]} = Z^z).$$

3.2 Splitters

A *splitter* S is a (rooted) directed tree and a map that associates a bag to each node of the tree, which is constructed by the following recursive algorithm. The constructor receives a tree decomposition (T, λ) of the given CQ and proceeds as follows. First, we find the splitting vertex of T by computing, for every vertex v of T, the size $|T_v|$ of the subtree at v, and then recursively moving from the root r of T to the child of the maximal size until we reach a vertex v with $|T_v| \leq |T_r|/2 + 1$. Then the root of S is associated with the bag $\lambda(v)$, which is called the *splitting bag* of S. The vertex v splits T into subtrees, whose induced tree decompositions are then used to recursively construct children in S. We will often refer to subtrees of S also as splitters.

For each splitter S, we define a set of its *boundary variables* $bv(S)$ by induction on the tree structure: for the root splitter, $bv(S)$ is the set of all answer variables of the given CQ, and, if S' is a child of S and β is the splitting bag of S, then $bv(S')$ is the restriction of $bv(S) \cup \boldsymbol{\nu}(\beta)$ to the set of all variables in the bags of the nodes of S'. A *boundary type* for a splitter S is a partial type defined on its boundary variables $bv(S)$.

Example 10. In our running example, we have the following root splitter S_1 with two children, S_2 and S_3:
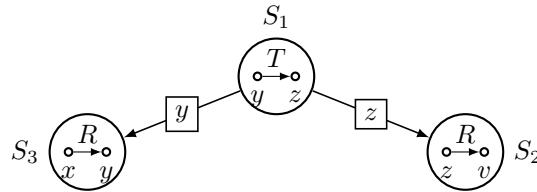
The sets of boundary variables for S_1, S_2 and S_3 are $\emptyset$, $\{y\}$ and $\{z\}$, respectively.

With any splitter S and a boundary type $\boldsymbol{w}$ for S, we associate a fresh IDB $P_{S,\boldsymbol{w}}$ with variables $var(\boldsymbol{w})$. In Example 10, we associate

- nullary predicate P_1 with S_1 and the empty boundary type $\{\}$;
- unary predicate $P_2(z)$ with S_2 and the boundary type $\{z \mapsto \varepsilon\}$;
- binary predicate $P_3(X^y, Z^y)$ with S_3 and the boundary type $\{y \mapsto (S(X,Z), e_0)\}$;
- and finally, unary predicate $P_4(y)$ with S_3 and the boundary type $\{y \mapsto \varepsilon\}$

(for simplicity, we use numerical subscripts rather than $S, \boldsymbol{w}$).

3.3 Bag Type Extender

A *splitting type* for a splitter S and its boundary type w is a type s for the splitting bag β of S that agrees with w on their common domain. `TypeExtender` produces all splitting types for a given bag $\beta = (\nu, \alpha)$ and a given partial type w by constructing a rooted type extender tree with a labelling function ℓ that assigns to each node v a partial type $\ell(v)$ such that the root is labelled with w, and $\ell(u)$ is an extension of $\ell(v)$ whenever u is a child of v. More precisely, the recursive algorithm maintains the following parameters when constructing the tree:

- the current node label, in other words, the partial type w' being extended
  (`currentType`);
- the subset α' of atoms α that are yet to be processed (`atomsToBeMapped`);
- the subset ν' of variables ν on which w' is yet to be defined (`varsToBeMapped`).

For the root of the type extender tree, we set $w' = w$, $\alpha' = \alpha$ and $\nu' = \nu \setminus \mathrm{dom}(w')$. Then, the constructor proceeds by recursion on decreasing sets α' and ν' using the following three rules in the given order:

(I) If ν' is empty, then, in function `filterThroughAtoms`, we check whether each atom in α' satisfies one of the three conditions, **(d)**, **(b)** or **(i)**, assuming that w' is a bag type for β. We mark the current leaf as *valid* if it is the case, and *invalid* otherwise.

(II) If α' contains an atom $R(z)$ such that w' is defined on some $z \in z$ and $w'(z)$ is an anonymous individual from $\mathfrak{T}_{\gamma(X)}$, then we say that $R(z)$ *connected to w'* and, using $\mathfrak{C}_{\gamma(X)}$, extend w' to all variables z on which it is not defined. More precisely, in function `extendToAnAtom`, we use *Graal* to execute the query $R(z)$ on $\mathfrak{C}_{\gamma(X)}$ and then extract all possible extensions from its output.

(III) If a connected atom cannot be found, then we pick a variable from ν' and, in function `ExtendToATerm`, assign all possible term types to it.

Thus, the extension tree is constructed using the following procedure:

```
1   /* Receives a node of the type extender tree under construction
2      Returns a tuple (status, extensions), where
3        - status is the status of the node, and
4        - extensions is the children of the current node.          */
5
6   getExtensions(currentType, varsToBeMapped, atomsToBeMapped) {
7     if (varsToBeMapped is Empty) {
8         return (filterThroughAtoms(currentType, atomsToBeMapped),
9                 EmptyList)
10    }
11    else if (atomsToBeMapped contains
12                        an atom connected to the currentType) {
13        return (true, ExtendToAnAtom(currentType, atom))
14    }
15    else {
16        return (true,
17                ExtendToATerm(currentType, varsToBeMapped.head))
18    }
19  }
```

When the type extender tree is fully constructed, all bag types s for β that extend w can be collected from the labels w' of valid tree leaves that satisfy $\mathrm{dom}(w') = \nu$.

Example 11. In the running example, initially, w' is empty, and so we apply rule **(III)** and create four children of the root for four possible term types for y. Then, we apply rule **(III)** to the node $\{y \mapsto \varepsilon\}$ and create four children for four possible term types for z. Then, by **(I)**, we apply the `FilterThroughAtoms` procedure to these four children and conclude that only $\{y \mapsto \varepsilon, z \mapsto \varepsilon\}$ is valid, while the other three siblings are not.

The three remaining children of the tree root have an atom $T(y, z)$ connected to its partial type, and so we apply rule **(II)** to them.

First, consider $\{y \mapsto (S(X, Z), e_0)\}$. We execute the query $T(y, z)$ on $\mathfrak{C}_{S(X,Z)}$, which yields $\{y \mapsto e_0, z \mapsto Z\}$. Since the obtained value for y matches the partial type $\{y \mapsto (S(X, Z), e_0)\}$, we create a single child $\{y \mapsto (S(X, Z), e_0), z \mapsto \varepsilon\}$, which is valid because α' is empty when `FilterThroughAtoms` is called.

Second, consider $\{y \mapsto (S(X, Z), e_1)\}$. The execution of the query $T(y, z)$ on $\mathfrak{C}_{S(X,Z)}$ gives the answer $\{y \mapsto e_0, z \mapsto Z\}$, which does not match the partial type, and so the node $\{y \mapsto (S(X, Z), e_1)\}$ has no children.

Third, the same happens with the remaining node $\{y \mapsto (T(X, Z), e_0)\}$.
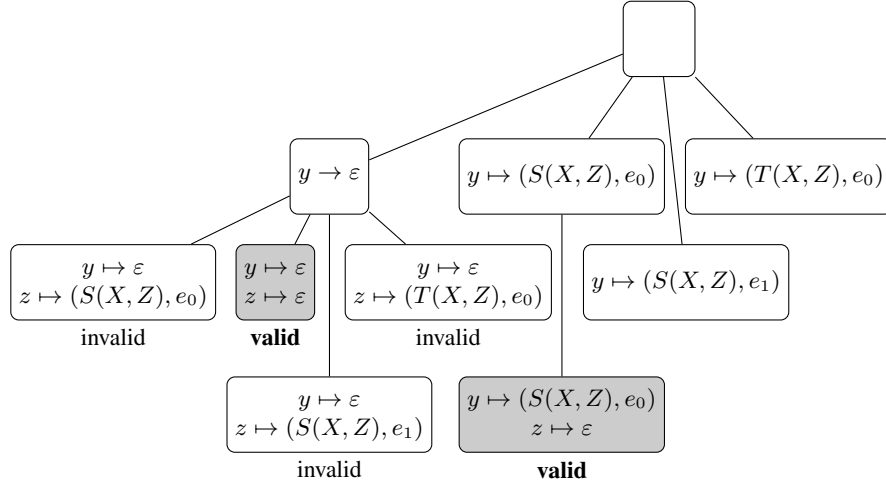
Finally, we collect the partial types from the valid leaves of the tree: note that the last two nodes considered do not give rise to any partial types because their labels w' do not satisfy $\mathrm{dom}(w') = \nu$. To sum up, there are two splitting types for S_1 and its empty boundary type that are compatible with $T(y, z)$:

$$\boldsymbol{t}_1 = \{\, y \mapsto \varepsilon,\ z \mapsto \varepsilon \,\} \qquad \text{and} \qquad \boldsymbol{t}_2 = \{\, y \mapsto (S(X, Z), e_0),\ z \mapsto \varepsilon \,\}.$$

3.4 Generating Rewriting

The `GenerateRewriting`$(S, \boldsymbol{w})$ function in STYPES receives a splitter S and its boundary type $\boldsymbol{w}$. It first calls `TypeExtender` to construct the splitting types for S and $\boldsymbol{w}$. Then, for each splitting type $\boldsymbol{s}$, the function creates a fresh IDB $P_{S,\boldsymbol{w}}(var(\boldsymbol{w}))$ for S and $\boldsymbol{w}$

(see Section 3.2) and produces a `RuleTemplate`, which describes clauses, whose head is $P_{S,\boldsymbol{w}}(var(\boldsymbol{w}))$ and whose body includes $\mathsf{At}^{\boldsymbol{s}}(var(\boldsymbol{s}))$ and the atoms $P_{S',\boldsymbol{w}'}(var(\boldsymbol{w}'))$ for children S' of S, where $\boldsymbol{w}'$ is the restriction of $\boldsymbol{w} \cup \boldsymbol{s}$ to the boundary variables $bv(S')$ of S'. The name `RuleTemplate` refers to the fact that the $\mathsf{At}^{\boldsymbol{s}}(var(\boldsymbol{s}))$ formulas in general contain disjunctions and therefore, each `RuleTemplate` may give rise to several clauses. Next, function `GenerateRewriting`$(S', \boldsymbol{w}')$ is called recursively for all children S' of S and their induced boundary types $\boldsymbol{w}'$.

Example 12. We continue Example 11. For S_1 and its empty boundary type, the splitting types $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$, give rise to the following:

$$P_1 \leftarrow T(y, z) \wedge P_2(z) \wedge P_4(y),$$
$$P_1 \leftarrow S(X^y, Z^y) \wedge P_2(z) \wedge P_3(X^y, Z^y) \wedge (z = Z^y).$$

For readability, we do not distinguish between variables V^u and $V^{[u]}$ and omit the respective equalities of the form $V^u = V^{[u]}$. Note that in these cases (and in the cases below) the $\mathsf{At}^{\boldsymbol{s}}(var(\boldsymbol{s}))$ formulas contain no disjunctions, and so we actually have clauses.

For S_2 and its boundary type $\{z \mapsto \varepsilon\}$, the splitting types $\boldsymbol{t}_3$, $\boldsymbol{t}_4$ and $\boldsymbol{t}_5$ (see Example 9) give rise to the clauses

$$P_2(z) \leftarrow R(z, v),$$
$$P_2(z) \leftarrow T(X^v, Z^v) \wedge (z = X^v),$$
$$P_2(z) \leftarrow S(X^v, Z^v) \wedge (z = X^v).$$

Next, for the splitter S_3 and its boundary type $\{y \mapsto (S(X, Z), e_0)\}$, the only splitting type $\{y \mapsto (S(X, Z), e_0),\ x \mapsto \varepsilon\}$ yields the clause

$$P_3(X^y, Z^y) \leftarrow S(X^y, Z^y) \wedge (x = X^y).$$

Finally, for S_3 and its other boundary type, $\{y \mapsto \varepsilon\}$, the only possible splitting type $\{y \mapsto \varepsilon,\ x \mapsto \varepsilon\}$ gives

$$P_4(y) \leftarrow R(x, y).$$

The produced list of `RuleTemplates` is passed to the `generateDatalog` function, which converts them into clauses and simplifies the resulting NDL program in the following way. First, it recursively removes predicates of the form $P_{S,\boldsymbol{w}}(var(\boldsymbol{w}))$ that are guaranteed to be empty because they do not occur in the head of any clause. Then, it recursively eliminates predicates with a single definition: each $P_{S,\boldsymbol{w}}(var(\boldsymbol{w}))$ that has a single clause with $P_{S,\boldsymbol{w}}(var(\boldsymbol{w}))$ in the head is replaced by the body of the clause (with the existentially quantified variables appropriately renamed). Finally, all equalities are removed from the program by repeatedly replacing one of the terms of an equality for all terms that are equivalent to it in the clause.

Example 13. In Example 12, P_3 and P_4 have single definitions, and so, they can be eliminated. This gives us the following NDL program:

$$P_1 \leftarrow T(y, z) \wedge P_2(z) \wedge R(x, y),$$
$$P_1 \leftarrow S(X^y, Z^y) \wedge P_2(z) \wedge (z = Z^y) \wedge (x = X),$$
$$P_2(z) \leftarrow R(z, v),$$
$$P_2(z) \leftarrow T(X^v, Z^v) \wedge (z = X^v),$$
$$P_2(z) \leftarrow S(X^v, Z^v) \wedge (z = X^v).$$

After removing equalities and replacing variables accordingly, we obtain the final NDL-rewriting given in Example 5.

4   Experiments

To understand whether the NDL-rewritings computed by STYPES are efficient in practice and whether a standard data management system is capable of taking advantage of their inherent parallelism, we conducted a few experiments with the ontology and CQs designed in [8]. The ontology was given in the introduction and the CQs are path queries with up to 15 atoms that correspond to words in the language $\{R, S\}^*$. For example, the CQ $q(x_0, x_7)$ for the word $RSRRSRR$ is shown below (with black nodes representing answer variables):
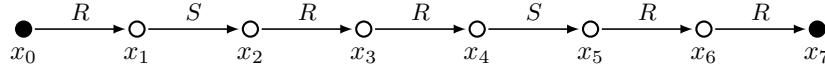
We used STYPES to compute NDL-rewritings for OMQ Q_{15} with a CQ of 15 atoms, Q_{22} with 7 atoms, and Q_{45} with 15 atoms. The queries are available at

http://github.com/srapisarda/stypes/tree/master/src/test/resources/ODBASE.

Their NDL-rewritings have, respectively, 25, 5 and 30 clauses. Thus, for Q_{15}, the constructed NDL-rewriting looks as follows, where p1 is the goal predicate:

```
1   p1(x0,x15) :- p35(x0,x7), r(x7,x8), p2(x8,x15).
2   p1(x0,x15) :- p3(x0,x8), a(x8), p2(x8,x15).
3   p3(x0,X) :- p19(x0,x3), r(x3,x4), p28(x4,X).
4   p3(x0,x2) :- r(x0,x1), r(x1,x2), a(x2), p28(x2,x2).
5   p3(x0,x6) :-  p19(x0,x6), b(x6), a(x6), r(x6,x6).
6   p28(x4,x4) :- a(x4).
7   p28(x4,x6) :- s(x4,x5), r(x5,x6), a(x6).
8   p2(x8,x15) :- p5(x10,x8), r(x10,x11), p14(x11,x15).
9   p2(x8,x15) :- r(x8,x9), a(x9), p14(x9,x15).
10  p14(x11,x15) :- r(x11,x12), s(x12,x13), p7(x13,x15).
11  p14(x11,x15) :- b(x11), p7(x11,x15).
12  p7(x15,x15) :- a(x15).
13  p7(x13,x15) :- s(x13,x14), r(x14,x15).
14  p5(x8,x8) :- b(x8).
15  p5(x10,x8) :- s(x9,x10), r(x8,x9).
16  p35(x0,x7) :- p43(x7,x2), r(x0,x1), r(x1,x2), a(x2).
```

```
17  p35(x0,x7) :- p19(x0,x3), r(x3,x4), p43(x7,x4).
18  p35(x0,x7) :- p40(x7,x3), p19(x0,x3),  b(x3).
19  p19(x0,x3) :- r(x0,x3), b(x3).
20  p19(x0,x3) :- r(x0,x1), r(x1,x2), s(x2,x3).
21  p43(x7,x4) :- s(x4,x5), r(x5,x6), s(x6,x7).
22  p43(x7,x4) :- a(x4), s(x4,x7).
23  p43(x7,x4) :- s(x4,x7), b(x7).
24  p40(x7,x5) :- b(x5), r(x5,x6), s(x6,x7).
25  p40(x5,x5) :- b(x5).
```

It is to be noted [8] that UCQ-rewritings of these OMQs are very large. STYPES encodes these UCQs as 'deep' yet polynomial-size NDL-queries. Thus, in the example above, p1 depends on p2, which depends on p14, which in turn depends on p7, with each of these predicates having at least two defining clauses. This makes STYPES different from all other existing rewriters. Neither Rapid nor Clipper terminate on this OMQ within 15 minutes, while Presto (in the NDL mode) produces an NDL-rewriting with 2723 clauses (see [8] for details). The aim of our experiments was to understand whether the optimal NDL-rewritings computed by STYPES are (i) executable and (ii) efficiently parallelisable.

We executed the NDL-rewritings on Apache Flink [14], a highly scalable modern tool for parallel streaming and batch processing based on estimated cost-based choice of the optimal physical execution plan.

For our experiments, we created a Hadoop cluster with six nodes, where each virtual machine had four Intel(R) Xeon(R) E5-2640 v3 CPUs @ 2.60GHz and 16GB RAM. Each machine served as a data node equipped with Hadoop HDFS of 250GB on SDD. For Flink, we used a stand-alone cluster configuration.

We represented clauses of NDL-rewritings using the `join-where-equalTo-map` sequences of standard Flink functions. For example, for the clause

```
p(x,z) :- w(x,y), v(z,x,y).
```

we produced the following Flink script:

```
val p = w.join(v).where(0,1).equalTo(1,2).map(t => (t._1._1, t._2._1))
```

In Flink, the `join` of two relations consists of all composite tuples (w, v) constructed from their tuples w and v. In the example above, `w.join(v)` consists of all composite tuples $((w_0, w_1), (v_0, v_1, v_2))$ for tuples (w_0, w_1) in `w` and tuples (v_0, v_1, v_2) in `v`. The arguments of the `where` function specify the positions in the first relation that must have the values equal to the values in the respective positions in the second relation specified by the arguments of `equalTo` (the positions start from 0). In the example above, the `where(0,1)` selects the first and second components of tuples in `w` (i.e., the components at positions 0 and 1), which are then matched by `equalsTo(1,2)` with the second and third components of tuples in `v` (which are at positions 1 and 2). Finally, we use function `map` to keep only the positions occurring in the clause head: for instance, `t => (t._1._1, t._2._1)` maps each composite tuple `t` to the first component of its first half and the first component of its second half.
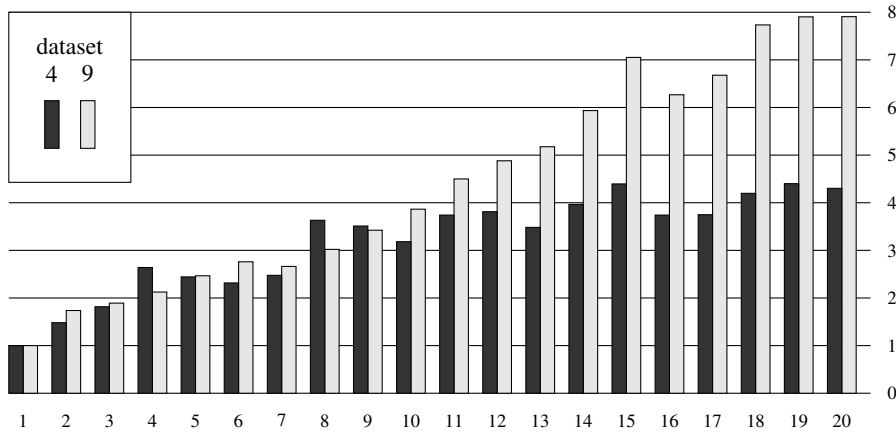
Fig. 1. Speed-up factor due to parallelisation for Q_{15}.

We used the following randomly generated datasets for the binary relation R and unary relations A and B:

dataset	$|V|$	p	q	avg. degree	#atoms	ttl size	csv size
1.ttl	1 000	0.050	0.050	50	61K	1MB	0.5MB
2.ttl	5 000	0.002	0.004	10	64K	1.2MB	0.7MB
3.ttl	10 000	0.002	0.004	20	257K	5MB	3MB
4.ttl	20 000	0.002	0.010	40	1M	20MB	12MB
5.ttl	30 000	0.002	0.010	60	2M	47MB	28MB
6.ttl	40 000	0.002	0.010	80	5M	84MB	51MB
7.ttl	50 000	0.002	0.010	100	6M	130MB	70MB
8.ttl	60 000	0.002	0.010	120	9M	190MB	100MB
9.ttl	70 000	0.002	0.010	140	13M	260MB	140MB

The parameter p is the probability of an R-edge between two points from V; q is the density of unary concepts A and B. The first four datasets come from [8], the last five were generated to create a significant load on the system. We intentionally decided to make the relation S empty in order to leave some margin for optimisation for the NDL-query planner, and also because empty relations are typical in the OBDA scenario.

We executed the constructed NDL-rewritings of the OMQs Q_{15}, Q_{22} and Q_{45} over the datasets 1.ttl–9.ttl on our cluster using Flink with the number of available virtual CPUs varying from 1 to 20. The run-times ranged from 3.91s to 1797s for Q_{15}, from 1.85s to 1398s for Q_{22}, and from 3.47s to 1769s for Q_{45}. Our main concern was the degree of parallelisability, which can be measured as the speed-up factor t_1/t_n, where t_i is the run-time on i-many CPUs over the same dataset. The results for two datasets, 4.ttl and 9.ttl, are presented in Figures 1–3: the horizontal axis indicates the number of CPUs available, while the vertical axis the speed-up factor. The charts show that the increase in the number of CPUs reduces query execution time, with the speed-up factor growing almost linearly with the number of CPUs, particularly on the larger dataset.
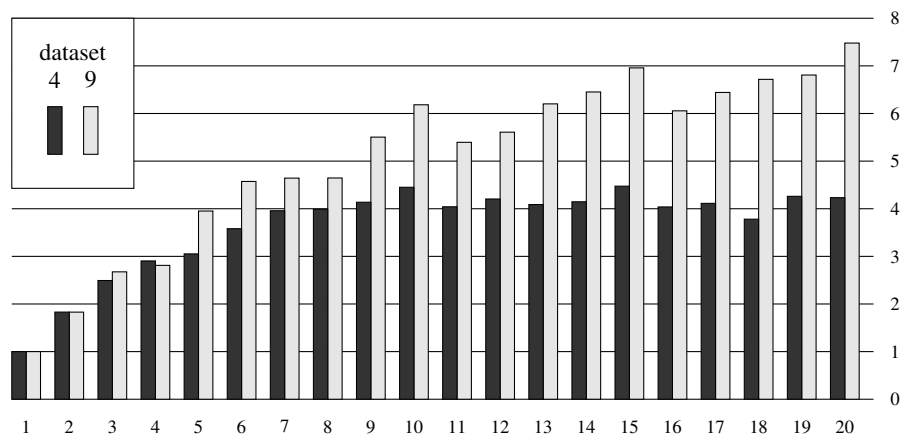
Fig. 2. Speed-up factor due to parallelisation for Q_{22}.

5   Conclusion

The contribution of this paper is twofold. First, it presents an OMQ rewriter STYPES that transforms conjunctive queries mediated by ontologies given as sets of linear tgds (in particular, *OWL 2 QL* concept and role inclusions) into equivalent nonrecursive datalog queries over the data. A distinctive feature of STYPES is that, if the treewidth of the input conjunctive queries and the size of the chases for the ontology atoms as well as their arity are bounded, then the resulting rewritings are theoretically optimal in the sense that they can be constructed and executed in LOGCFL. Second, the paper experimentally demonstrates that optimal NDL-rewritings computed by STYPES can be efficiently executed by Apache Flink, whereas other existing rewriters struggle even to produce rewritings for the same OMQs. It is also shown that Fink is capable of parallelising the execution of STYPES's NDL-rewritings proportionally to the number of available CPUs, although it remains to be seen how exactly Flink utilises the structure of the rewritings and whether further improvements are possible.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Abseher, M., Musliu, N., Woltran, S.: htd - A free, open-source framework for (customized) tree decompositions and beyond. In: Proc. of the 14th Int. Conf. on Integration of AI and OR Techniques in Constraint Programming, CPAIOR 2017. LNCS, vol. 10335, pp. 376–386. Springer (2017). https://doi.org/10.1007/978-3-319-59776-8_30
3. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The DL-Lite family and relations. Journal of Artificial Intelligence Research (JAIR) **36**, 1–69 (2009). https://doi.org/10.1613/jair.2820
4. Baget, J.F., Leclère, M., Mugnier, M.L., Rocher, S., Sipieter, C.: Graal: A toolkit for query answering with existential rules. In: Proc. of the 9th Int. Symposium on Rule Technologies: Foundations, Tools, and Applications, RuleML 2015. LNCS, vol. 9202, pp. 328–344. Springer (2015). https://doi.org/10.1007/978-3-319-21542-6_21
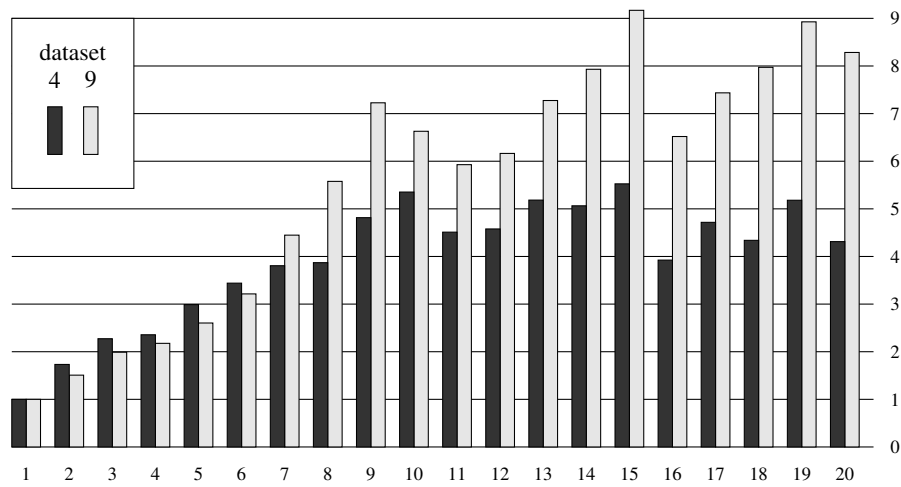
Fig. 3. Speed-up factor due to parallelisation for Q_{45}.

5. Baget, J.F., Leclère, M., Mugnier, M.L., Salvat, E.: Extending decidable cases for rules with existential variables. In: Proc. of the 21th Int. Joint Conf. on Artificial Intelligence (IJCAI 2009). pp. 677–682. IJCAI (2009)

6. Baget, J.F., Leclère, M., Mugnier, M.L., Salvat, E.: On rules with existential variables: Walking the decidability line. Artificial Intelligence **175**(9–10), 1620–1654 (2011). https://doi.org/10.1016/j.artint.2011.03.002

7. Bienvenu, M., Kikot, S., Kontchakov, R., Podolskii, V., Zakharyaschev, M.: Ontology-mediated queries: Combined complexity and succinctness of rewritings via circuit complexity. Journal of the ACM **65**(5), 28:1–28:51 (2018). https://doi.org/10.1145/3191832

8. Bienvenu, M., Kikot, S., Kontchakov, R., Podolskii, V.V., Ryzhikov, V., Zakharyaschev, M.: The complexity of ontology-based data access with OWL 2 QL and bounded treewidth queries. In: Proc. of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017. pp. 201–216. ACM (2017). https://doi.org/10.1145/3034786.3034791

9. Bienvenu, M., Kikot, S., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: Optimal non-recursive datalog rewritings of linear tgds and bounded (hyper)tree-width queries. In: Proc. of the 30th Int. Workshop on Description Logics, DL 2017. CEUR Workshop Proceedings, vol. 1879. CEUR-WS.org (2017)

10. Boguslavsky, I., Dikonov, V., Iomdin, L., Lazursky, A., Sizov, V., Timoshenko, S.: Semantic analysis and question answering: a system under development. In: Computational Linguistics and Intellectual Technologies (Papers from the Annual Int. Conf. Dialogue 2015, vol. 1), pp. 62–79. RSUH (2015)

11. Calì, A., Gottlob, G., Lukasiewicz, T.: A general datalog-based framework for tractable query answering over ontologies. Journal of Web Semantics **14**, 57–83 (2012). https://doi.org/10.1016/j.websem.2012.03.001

12. Calì, A., Gottlob, G., Pieris, A.: Advanced processing for ontological queries. PVLDB **3**(1), 554–565 (2010). https://doi.org/10.14778/1920841.1920912

13. Calì, A., Gottlob, G., Pieris, A.: Towards more expressive ontology languages: The query answering problem. Artificial Intelligence **193**, 87–128 (2012).

https://doi.org/10.1016/j.artint.2012.08.002

14. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)

15. Chekuri, C., Rajaraman, A.: Conjunctive query containment revisited. Theoretical Computer Science **239**(2), 211–229 (2000). https://doi.org/10.1016/S0304-3975(99)00220-0

16. Chortaras, A., Trivela, D., Stamou, G.: Optimized query rewriting for OWL 2 QL. In: Proc. of the 23rd Int. Conf. on Automated Deduction, CADE-23. LNCS, vol. 6803, pp. 192–206. Springer (2011). https://doi.org/10.1007/978-3-642-22438-6_16

17. Eiter, T., Ortiz, M., Šimkus, M., Tran, T.K., Xiao, G.: Query rewriting for Horn-SHIQ plus rules. In: Proc. of the 26th AAAI Conf. on Artificial Intelligence, AAAI 2012. pp. 726–733. AAAI Press (2012)

18. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theoretical Computer Science **336**(1), 89–124 (2005). https://doi.org/10.1016/j.tcs.2004.10.033

19. Gottlob, G., Kikot, S., Kontchakov, R., Podolskii, V.V., Schwentick, T., Zakharyaschev, M.: The price of query rewriting in ontology-based data access. Artificial Intelligence **213**, 42–59 (2014). https://doi.org/10.1016/j.artint.2014.04.004

20. Gottlob, G., Leone, N., Scarcello, F.: Computing LOGCFL certificates. In: Proc. of the 26th Int. Colloquium on Automata, Languages and Programming, ICALP-99. LNCS, vol. 1644, pp. 361–371. Springer (1999). https://doi.org/10.1007/3-540-48523-6_33

21. Kikot, S., Kontchakov, R., Podolskii, V.V., Zakharyaschev, M.: Exponential lower bounds and separation for query rewriting. In: Proc. of the 39th Int. Colloquium on Automata, Languages and Programming, ICALP 2012. LNCS, vol. 7392, pp. 263–274. Springer (2012). https://doi.org/10.1007/978-3-642-31585-5_26

22. Lenzerini, M.: Data integration: A theoretical perspective. In: Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'02. pp. 233–246. ACM (2002). https://doi.org/10.1145/543613.543644

23. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language Profiles. W3C Recommendation (2012), `http://www.w3.org/TR/owl2-profiles`

24. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. Journal on Data Semantics **10**, 133–173 (2008). https://doi.org/10.1007/978-3-540-77688-8_5

25. Rosati, R., Almatelli, A.: Improving query answering over DL-Lite ontologies. In: Proc. of the 12th Int. Conf. on Principles of Knowledge Representation and Reasoning, KR 2010. pp. 290–300. AAAI Press (2010)

26. Ruzzo, W.L.: Tree-size bounded alternation. Journal of Computer and System Sciences **21**(2), 218–235 (1980). https://doi.org/10.1016/0022-0000(80)90036-7

27. Rygaev, I.: Rule-based reasoning in semantic text analysis. In: Proc. of the Doctoral Consortium, Challenge, Industry Track, Tutorials and Posters @ RuleML+RR 2017. CEUR Workshop Proceedings, vol. 1875. CEUR-WS.org (2017)

28. Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: Ontology-based data access: A survey. In: Proc. of the 27th Int. Joint Conf. on Artificial Intelligence, IJCAI-ECAI 2018. pp. 5511–5519. IJCAI/AAAI (2018). https://doi.org/10.24963/ijcai.2018/777

29. Yannakakis, M.: Algorithms for acyclic database schemes. In: Proc. of the 7th Int. Conf. on Very Large Data Bases, VLDB'81. pp. 82–94. IEEE Computer Society (1981)