A Datalog Rewriting Algorithm for Warded Ontologies

Davide Benedetto[1] , **Marco Calautti**[2] , **Hebatalla Hammad**[3] , **Emanuel Sallinger**[4] , **Adriano Vlad-Starrabba**[1]
[1]Prometheux
[2]University of Milan
[3]University of Trento
[4]TU Wien, Austria and University of Oxford, UK
davben@prometheux.co.uk, marco.calautti@unimi.it, hebatalla.hammad@unitn.it
sallinger@dbai.tuwien.ac.at, adriano@prometheux.co.uk

Abstract

Existential rules, a.k.a. tuple-generating dependencies (TGDs), form a well-established formalism for specifying ontologies. In particular, the *warded* language is a well-behaved fragment of TGD-based ontologies, striking a good balance between expressive power and computational complexity of answering Ontology-Mediated Queries (OMQs). The theoretical foundations of answering OMQs over warded ontologies are by now well-understood, but to the best of our knowledge, very few efforts exist that exploit such a rich theory for building practical query answering algorithms. Our goal is to fill the above gap by designing a novel Datalog rewriting algorithm for OMQs over warded ontologies which is amenable to practical implementations, as well as providing an implementation and an experimental evaluation, with the aim of understanding how key input parameters affect the performance of this approach, and what are its limits when combined with off-the-shelf Datalog-based engines.

1 Introduction

Ontological reasoning is a fundamental task in Knowledge Representation and Reasoning (KRR) and Artificial Intelligence in general, as it enables the development of intelligent data management systems, where data is enriched with additional knowledge that can be derived by means of an ontology, i.e., a logic-based formalization of the domain of interest. A prominent formalism for encoding ontologies is the one of *existential rules*, a.k.a. *tuple-generating dependencies (TGDs)*, which allow to encode knowledge by means of implication-like formulas, specifying how new knowledge can be derived from existing knowledge known about the system. In this context, the key task is *Ontology-Mediated Query (OMQ) Answering*. That is, given a database D, containing known facts about the system, and an OMQ $\mathcal{O} = (q, \Sigma)$, where Σ is a TGD-based ontology, and q a conjunctive query (CQ), OMQ Answering is the task of finding the so-called *certain answers* to $\mathcal{O}$ over the database D. It is well-known that the problem of OMQ Answering is uncomputable, in general (e.g., see [Calì *et al.*, 2013]).

Thus, a long stream of literature has been developed in the last decades, devoted to identify fragments of TGD-based ontologies that guarantee the computability of certain answers. See for example acyclicity-based TGD languages [Fagin *et al.*, 2005; Greco *et al.*, 2011; Spezzano and Greco, 2010; Calautti *et al.*, 2016; Carral *et al.*, 2017; Calautti *et al.*, 2015; Calautti and Pieris, 2021; Calautti *et al.*, 2022], as well as UCQ-rewritable languages [Calì *et al.*, 2012a; Calì *et al.*, 2012b]. More expressive languages are the ones based on the notion of guardedness [Calì *et al.*, 2012a; Baget *et al.*, 2011; Benedikt *et al.*, 2022], and beyond [Calì *et al.*, 2013]. We point out that some of the above works have been influenced by earlier works on Description Logics (DL), e.g., see [Calvanese *et al.*, 2007; Poggi *et al.*, 2008; Artale *et al.*, 2009] for UCQ-rewritable DL ontologies, and [Baader *et al.*, 2005; Lutz and Sabellek, 2022] for guarded-like DL ontologies. We refer to [Baader *et al.*, 2003] for a comprehensive overview.

Despite all the above efforts, each of the above TGD-based languages has at least one of two downsides: they either have limited expressive power (e.g., UCQ-rewritable or guarded-based ones), or they are highly expressive but OMQ Answering becomes computationally challenging. To remedy the above situation, *warded* TGDs have been introduced in [Gottlob and Pieris, 2015], striking a good balance between expressive power and computational complexity. In particular, warded ontologies are expressive enough to capture the full power of Datalog, meaning they can express important properties of data, such as the transitive closure of graphs, but at the same time keep certain answers computation feasible in polynomial time in data complexity, i.e., w.r.t. the size of the input database. The above complexity result has been first shown in [Gottlob and Pieris, 2015] by means of specialized alternating algorithms, while more recent works have shown that warded ontologies are actually Datalog-rewritable [Berger *et al.*, 2022]. That is, every OMQ $\mathcal{O} = (q, \Sigma)$ with Σ warded can be rewritten to a Datalog query $\mathcal{D}_{\mathcal{O}}$ such that for every database D, the certain answers of $\mathcal{O}$ over D coincide with the answers of $\mathcal{D}_{\mathcal{O}}$ over D. Although wardedness can be considered as the sweet spot between expressiveness and complexity, we are not aware of any efforts from the literature that exploit such a rich theory to develop practical implementations of OMQ Answering over warded TGDs. One of the main reasons is that the results of [Gottlob and Pieris, 2015; Berger *et al.*, 2022] are mostly theoretical in nature. In particular, the Datalog rewriting algorithm of [Berger *et al.*, 2022], constructs an equivalent Datalog query via a naive brute force search over the (exponentially large) space of so-called proof trees of the OMQ at hand. In fact, the only real-world system we are aware of that supports, to some extent, warded ontologies, is the Vadalog system [Bellomarini *et al.*, 2018; Bellomarini *et al.*, 2022; Bellomarini *et al.*, 2024], which supports a simpler version of OMQ Answering, where the CQ is atomic, and uses a procedure different from the ones of [Gottlob and Pieris, 2015; Berger *et al.*, 2022]. The latter, together with the fact that Vadalog is available only after acquiring a license, severely limits the use of warded TGDs in practice.

Contributions. The goal of our work is to rectify the above state of affairs by providing a new Datalog rewriting algorithm for OMQs over warded ontologies, dubbed $\mathsf{WardedRewrite}$, that is more amenable to practical implementations, supporting OMQs $\mathcal{O} = (q, \Sigma)$, where Σ is a warded TGD-based ontology, and q an arbitrary CQ, thus going beyond the capability of existing systems that support the warded language. We then implement $\mathsf{WardedRewrite}$, and perform an experimental analysis that highlights the applicability of our implementation, and provides key insights on the capability of different off-the-shelf Datalog-based engines in computing certain answers over OMQs with warded ontologies, via the evaluation of the Datalog query obtained using our algorithm $\mathsf{WardedRewrite}$.

The full source code and the benchmark data are available at: https://gitlab.com/mcalautti/warded-rewriting-paper.

2 Preliminaries

We consider the disjoint countably infinite sets $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ of *constants*, *(labeled) nulls*, and *variables*, respectively. We refer to constants, nulls and variables as *terms*. For an integer $n > 0$, we write $[n]$ for the set of integers $\{1, \ldots, n\}$.

Relational Databases. A *schema* $\mathbf{S}$ is a finite set of relation symbols (or predicates) with associated arity. We write R/n to denote that R has arity $n > 0$; we may also write $\mathsf{ar}(R)$ for the integer n. A *(predicate) position* of $\mathbf{S}$ is a pair (R, i), where $R/n \in \mathbf{S}$ and $i \in [n]$, that essentially identifies the i-th argument of R. We write $\mathsf{pos}(\mathbf{S})$ for the set of positions of $\mathbf{S}$, that is, the set $\{(R, i) \mid R/n \in \mathbf{S} \text{ and } i \in [n]\}$. An *atom* over $\mathbf{S}$ is an expression of the form $R(\bar{t})$, where $R/n \in \mathbf{S}$ and $\bar{t}$ is an n-tuple of terms. A *fact* is an atom whose arguments consist only of constants. We write $\mathsf{var}(R(\bar{t}))$ for the set of variables in $\bar{t}$. The notation $\mathsf{var}(\cdot)$ extends naturally to other objects mentioning variables. An *instance* over $\mathbf{S}$ is a (possibly infinite) set of atoms over $\mathbf{S}$ with constants and nulls. A *database* D over $\mathbf{S}$ is a finite set of facts, i.e., a finite instance mentioning only constants. The *active domain* of an instance I, denoted $\mathsf{dom}(I)$, is the set of terms occurring in I.

Homomorphisms and Conjunctive Queries. A *substitution* from a set of terms T to a set of terms T' is a function $h : T \to T'$. A *homomorphism* from a set of atoms A to a set of atoms B is a substitution h from the set of terms in A to the set of terms in B such that h is the identity on $\mathbf{C}$, and $R(t_1, \ldots, t_n) \in A$ implies $h(R(t_1, \ldots, t_n)) = R(h(t_1), \ldots, h(t_n)) \in B$. For an integer $k \geq 0$, a k-ary *conjunctive query (CQ)* q over a schema $\mathbf{S}$ with *output variables* $\bar{x}$, denoted $q(\bar{x})$, is an expression of the form $Q(\bar{x}) \leftarrow \exists \bar{y}\, R_1(\bar{x}_1) \wedge \ldots \wedge R_n(\bar{x}_n)$, where, for each $i \in [n]$, $\bar{x}_i$ is a tuple of variables, and $\bar{x}$, $\bar{y}$ form a partition of the set of variables $\bigcup_{i \in [n]} \bar{x}_i$; by abuse of notation, we treat a tuple of variables as a set of variables. Moreover, $\bar{x}$ is a tuple of k (not necessarily distinct) variables, and $Q(\bar{x})$ and $R_i(\bar{x}_i)$, for $i \in [n]$, are atoms with $R_i \in \mathbf{S}$, while $Q \notin \mathbf{S}$. For convenience, we may use comma in place of the symbol $\wedge$, and we may omit the existential quantifier. The *body* (resp., *head*) of q, denoted $\mathsf{body}(q)$ (resp., $\mathsf{head}(q)$) is the *set* of atoms $\{R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)\}$ (resp., the atom $Q(\bar{x})$). We use $\mathsf{headPred}(q)$ to denote the predicate Q. For an instance I over $\mathbf{S}$, we write $q(I)$ for the *answers of* q *over* I, defined as $\{\bar{t} \in (\mathsf{dom}(I) \cap \mathbf{C})^k \mid \exists \text{ a homomorphism } h \text{ from } \mathsf{body}(q) \text{ to } I \text{ with } h(\bar{x}) = \bar{t}\}$.

Tuple-Generating Dependencies. A *tuple-generating dependency* (TGD) σ over a schema $\mathbf{S}$ is a (constant-free) expression of the form $\forall \bar{x} \forall \bar{y}\, (\phi(\bar{x}, \bar{y}) \to \exists \bar{z}\, \psi(\bar{x}, \bar{z}))$, where $\bar{x}, \bar{y}$ and $\bar{z}$ are tuples of variables of $\mathbf{V}$, and $\phi(\bar{x}, \bar{y})$ and $\psi(\bar{x}, \bar{z})$ are non-empty conjunctions of atoms over $\mathbf{S}$ that only mention variables from $\bar{x} \cup \bar{y}$ and $\bar{x} \cup \bar{z}$, respectively. We may write σ as $\phi(\bar{x}, \bar{y}) \to \exists \bar{z}\, \psi(\bar{x}, \bar{z})$, and use comma instead of $\wedge$ for joining atoms. We refer to $\phi(\bar{x}, \bar{y})$ and $\psi(\bar{x}, \bar{z})$ as the *body* and *head* of σ, denoted $\mathsf{body}(\sigma)$ and $\mathsf{head}(\sigma)$, respectively. We say that σ is *full* if $\bar{z}$ is empty, and we say it is *single-head* if $\psi(\bar{x}, \bar{z})$ contains a single atom. The *frontier* of the TGD σ, denoted $\mathsf{fr}(\sigma)$, is the set of variables $\bar{x}$, i.e., the variables that appear both in the body and the head of σ. We use $\mathsf{exvar}(\sigma)$ and $\mathsf{expos}(\sigma)$ to denote the set of all existential variables of σ, and the set of all positions in which an existential variable of σ occurs. An *ontology* Σ over a schema $\mathbf{S}$ is a finite set of TGDs over $\mathbf{S}$. The *schema* of an ontology Σ, denoted $\mathsf{sch}(\Sigma)$, is the set of predicates occurring in Σ. Moreover, we assume w.l.o.g. that no two TGDs in Σ share a variable. We use TGD to denote the class of all ontologies.

An instance I satisfies a TGD σ as the one above, written $I \models \sigma$, if whenever there exists a homomorphism h from $\phi(\bar{x}, \bar{y})$ to I, then there is an extension of h that is a homomorphism from $\psi(\bar{x}, \bar{z})$ to I; we may treat a conjunction of atoms as a set of atoms. The instance I satisfies an ontology Σ, written $I \models \Sigma$, if $I \models \sigma$ for each $\sigma \in \Sigma$.

Ontology-Mediated Queries. The main task we are interested in this paper is query answering under TGDs. Consider a database D over a schema $\mathbf{S}$ and an ontology Σ over $\mathbf{S}$. A *model* of D and Σ is an instance I over $\mathbf{S}$ such that $D \subseteq I$ and such that $I \models \Sigma$. We use $\mathsf{models}(D, \Sigma)$ to denote the set of all models of D and Σ. For an integer $k \geq 0$, a k-ary *ontology-mediated query (OMQ)* over a schema $\mathbf{S}$ is a pair $\mathcal{O} = (q, \Sigma)$, where q is a k-ary CQ over $\mathbf{S}$, and Σ is an ontology over $\mathbf{S}$. For a database D over $\mathbf{S}$, the *certain answers of* $\mathcal{O}$ *over* D is the set of tuples $\mathsf{ans}(\mathcal{O}, D)$ defined as

$$\{\bar{t} \in \mathsf{dom}(D)^k \mid \bar{t} \in q(I) \text{ for all } I \in \mathsf{models}(D, \Sigma)\}.$$

The Chase Procedure. The chase procedure is an algorithm that can be used to produce a so-called *universal model* of a database D and ontology Σ, which in turn can be used to provide an alternative definition of certain answers of OMQs. Consider an instance I and an ontology Σ over a schema $\mathbf{S}$. A *trigger* for Σ on I is a pair (σ, h), where $\sigma \in \Sigma$ and h is a homomorphism from $\mathsf{body}(\sigma)$ to I. The *result* of (σ, h), denoted $\mathsf{result}(\sigma, h)$, is the set $\mu(\mathsf{head}(\sigma))$, where $\mu : \mathsf{var}(\mathsf{head}(\sigma)) \to \mathbf{C} \cup \mathbf{N}$ is such that $\mu(x) = h(x)$ if $x \in \mathsf{fr}(\sigma)$, and $\mu(x) = \perp^x_{\sigma,h}$, with $\perp^x_{\sigma,h} \in \mathbf{N}$, otherwise.

For a database D and an ontology Σ over a schema $\mathbf{S}$, the *chase of D w.r.t.* Σ is the set $\mathsf{chase}(D, \Sigma)$ of atoms inductively defined as follows. We define $\mathsf{chase}^0(D, \Sigma) = D$, and for each $i > 0$, $\mathsf{chase}^i(D, \Sigma) = \mathsf{chase}^{i-1}(D, \Sigma) \cup \{\mathsf{result}(\sigma, h) \mid (\sigma, h) \text{ is a trigger for } \Sigma \text{ on } \mathsf{chase}^{i-1}(D, \Sigma)\}$. Finally, $\mathsf{chase}(D, \Sigma) = \bigcup_{i \geq 0} \mathsf{chase}^i(D, \Sigma)$.

The following is well-known [Grahne and Onet, 2018]:

Theorem 1. *For a k-ary OMQ $\mathcal{O} = (q, \Sigma)$ and a database D over a schema $\mathbf{S}$, $\mathsf{ans}(\mathcal{O}, D) = \{\bar{t} \in \mathsf{dom}(D)^k \mid \bar{t} \in q(\mathsf{chase}(D, \Sigma))\}$.*

3 Wardedness and Datalog Rewritability

At the high-level, a warded ontology Σ restricts how some of the variables of its TGDs, called ”dangerous” variables, are used. We now proceed to formally define the above notions; in what follows, fix an ontology Σ.

Affected Positions. First, we need the notion of affected positions. For a position $(R, i) \in \mathsf{pos}(\mathsf{sch}(\Sigma))$, we inductively say that (R, i) is an *affected position of* Σ as follows: there exists a TGD $\sigma \in \Sigma$ with $(R, i) \in \mathsf{expos}(\sigma)$, or there exists a TGD $\sigma \in \Sigma$ and a variable $x \in \mathsf{fr}(\sigma)$ occurring at (R, i) in the head of σ such that x occurs in $\mathsf{body}(\sigma)$ only at affected positions. We use $\mathsf{aff}(\Sigma)$ to denote the set of all affected positions of Σ, and $\mathsf{notaff}(\Sigma) = \mathsf{pos}(\mathsf{sch}(\Sigma)) \setminus \mathsf{aff}(\Sigma)$ to denote set of all position of Σ that are not affected. Intuitively, when a position (R, i) of Σ is *not* affected, it is guaranteed that for every database D, every atom of the form $R(t_1, \ldots, t_n) \in \mathsf{chase}(D, \Sigma)$ is such that t_i is a constant.

Wardedness. Consider a variable x occurring in the body of some TGD σ of Σ. We say that x is *harmless* if it occurs in $\mathsf{body}(\sigma)$ at some position of $\mathsf{notaff}(\Sigma)$; *harmful* if it is not harmless, i.e., it occurs in $\mathsf{body}(\sigma)$ *only* at positions of $\mathsf{aff}(\Sigma)$; *dangerous* if it is harmful and it is a frontier variable of σ. We say that Σ is *warded* if for each TGD $\sigma \in \Sigma$ there exists an atom $\alpha \in \mathsf{body}(\sigma)$, called the *ward of* σ, such that (i) all dangerous variables of σ occur in α, and (ii) each variable in $\mathsf{var}(\alpha) \cap \mathsf{var}(\mathsf{body}(\sigma) \setminus \{\alpha\})$ is harmless. We use WARDED to denote the set of all warded ontologies.

Datalog Rewritability and Wardedness. A *Datalog program* is an ontology Σ containing only full, single-head TGDs, which we also call (Datalog) *rules*. A k-ary *Datalog query* is a pair $\mathcal{D} = (R, \Sigma)$, where Σ is a Datalog program, and $R/k \in \mathsf{sch}(\Sigma)$. A predicate occurring in Σ is *intensional* if it occurs in the head of some rule of Σ, otherwise, it is *extensional*. We use $\mathsf{idb}(\mathcal{D})$ and $\mathsf{edb}(\mathcal{D})$ to denote the set of all intensional and extensional predicates of $\mathcal{D}$, respectively. For a Datalog query $\mathcal{D} = (R, \Sigma)$, and a database D over $\mathsf{edb}(\mathcal{D})$, the *answers of $\mathcal{D}$ over D* is the set $\mathcal{D}(D) = \{\bar{t} \in \mathsf{dom}(D)^k \mid R(\bar{t}) \in \mathsf{chase}(D, \Sigma)\}$. A class C of ontologies is *Datalog-rewritable* if for every OMQ $\mathcal{O} = (q, \Sigma)$ over a schema $\mathbf{S}$, with $\Sigma \in \mathsf{C}$, there exists a Datalog query $\mathcal{D}_{\mathcal{O}} = (R_{\mathcal{O}}, \Sigma_{\mathcal{O}})$ with $\mathsf{edb}(\mathcal{D}_{\mathcal{O}}) \subseteq \mathbf{S}$ and $\mathsf{idb}(\mathcal{D}_{\mathcal{O}}) \cap \mathbf{S} = \emptyset$, called a *Datalog rewriting of* $\mathcal{O}$, such that for every database D over $\mathbf{S}$, $\mathsf{ans}(\mathcal{O}, D) = \mathcal{D}_{\mathcal{O}}(D')$, with $D' = \{R(\bar{t}) \in D \mid R \in \mathsf{edb}(\mathcal{D}_{\mathcal{O}})\}$. It is known that WARDED is Datalog-rewritable [Berger *et al.*, 2022].

4 A Datalog Rewriting Algorithm for Warded

We present a new algorithm that is able to produce a Datalog rewriting of a given OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in$ WARDED that is more amenable to be implemented in practice.

At the high-level, given an OMQ $\mathcal{O} = (q, \Sigma)$, with $\Sigma \in$ WARDED, our algorithm exhaustively applies, starting from the query q, one of the following two basic steps to the current query being processed: decomposition and resolution. The decomposition step is in charge of decomposing the query being processed into the smallest possible, independently processable queries. The resolution step is in charge of unfolding the smaller queries using the TGDs of the ontology. The latter step is the same employed by other algorithms from the literature that deal with UCQ-rewritable OMQs. We recall that a union of conjunctive queries (UCQ) is a Datalog query (R, Σ) containing only rules having R as their head predicate, and R does not occur in their body. However, the decomposition step is what distinguishes our algorithm from ones for UCQ-rewritable languages, as it guarantees the construction of a *finite* Datalog query, when focusing on warded ontologies. We now introduce the basic notions needed to define the above two steps.

Chunk-Based Resolution. Let A and B be non-empty sets of atoms that mention only variables. The sets A and B *unify* if there is a substitution γ, called *unifier for A and B*, such that $\gamma(A) = \gamma(B)$. A *most general unifier (MGU)* for A and B, denoted $\mathsf{mgu}(A, B)$, is a unifier for A and B such that for each unifier γ for A and B, $\gamma = \gamma' \circ \mathsf{mgu}(A, B)$, for some substitution γ'. It is well-known that if two sets of atoms unify, then there is always a MGU, which is unique (modulo variable renaming). Given a CQ $q(\bar{x})$ and a set of atoms $S \subseteq \mathsf{body}(q)$, we say that a variable $y \in \mathsf{var}(S)$ is *shared in q (w.r.t. S)*, if $y \in \bar{x}$ or $y \in \mathsf{var}(\mathsf{body}(q) \setminus S)$. We use $\mathsf{shared}(q, S)$ to denote the set of all variables in S that are shared in q w.r.t. S. We can now recall chunk unifiers.[1]

Definition 2. Consider a CQ $q(\bar{x})$ and a TGD σ having no variables in common with q. A *chunk unifier* of q with σ is a triple (S_1, S_2, γ), where

- $\emptyset \subsetneq S_1 \subseteq \mathsf{body}(q)$ and $\emptyset \subsetneq S_2 \subseteq \mathsf{head}(\sigma)$;
- γ is a unifier for S_1 and S_2;
- for each $x \in \mathsf{var}(S_2) \cap \mathsf{exvar}(\sigma)$, and for each variable $y \neq x$, $\gamma(x) = \gamma(y)$ implies that y occurs in S_1 (and thus $y \notin \mathsf{var}(S_2)$) and that $y \notin \mathsf{shared}(q, S_1)$.

We say that a chunk unifier (S_1, S_2, γ) is *most general (MGCU)* if γ is an MGU for S_1 and S_2. ■

[1] Chunk-unifiers, employed in most works related to wardedness, are equivalent to the *piece-unifiers* from [König *et al.*, 2015].

In other words, an MGCU faithfully describes how the chase would trigger σ producing a set of atoms to which the atoms in S_1 would homomorphically map.

The notion of chunk-based resolution follows naturally.

Definition 3. Consider a CQ $q(\bar{x})$, with $\mathsf{headPred}(q) = Q$, a TGD σ, and a MGCU $M = (S_1, S_2, \gamma)$ of q with σ. The *σ-resolvent of q (via M)*, denoted by $\mathsf{resolv}_{\sigma,M}(q)$, is the CQ q' with $\mathsf{head}(q') = Q(\gamma(\bar{x}))$, and $\mathsf{body}(q') = \gamma((\mathsf{body}(q) \setminus S_1) \cup \mathsf{body}(\sigma))$. ∎

As we are going to see in Section 4.1, a resolution step exhaustively applies chunk-based resolution on a given CQ.

Query Decomposition. We now move to the notions needed to implement the other step of our algorithm: the decomposition step. We start with the notion of existential-join decomposition, employed for other tasks in [Gottlob *et al.*, 2014].

Definition 4. Consider a CQ $q(\bar{x})$ with $\mathsf{headPred}(q) = Q$, and an ontology Σ. An *existential-join decomposition of q w.r.t. Σ*, or simply decomposition, is a set of $n > 1$ CQs $\mathcal{S} = \{q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)\}$ where $\{\mathsf{body}(q_1), \ldots, \mathsf{body}(q_n)\}$ is a partition of $\mathsf{body}(q)$, $\mathsf{headPred}(q_i) = Q_i$ for $i \in [n]$, and for each $i \in [n]$:

1. $\bar{x}_i = \mathsf{shared}(q, \mathsf{body}(q_i))$, and
2. for every two atoms $\alpha, \beta \in \mathsf{body}(q)$, if $\alpha \in \mathsf{body}(q_i)$ and α and β mention a variable $y \notin \bar{x}$ occurring only in affected positions of $\mathsf{body}(q)$, then $\beta \in \mathsf{body}(q_i)$.

We say that $\mathcal{S}$ is *optimal* if for each $i \in [n]$, there is no existential-join decomposition of q_i w.r.t. Σ. ∎

The main idea behind an optimal decomposition $\mathcal{S} = \{q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)\}$ of a CQ $q(\bar{x})$ w.r.t. Σ is to factorize the body of q into the smallest subqueries, guaranteeing that the certain answers of (q, Σ) can be retrieved by combining the certain answers of each OMQ (q_i, Σ), for $i \in [n]$. In particular, assuming $\mathsf{headPred}(q) = Q$, and $\mathsf{headPred}(q_i) = Q_i$, for $i \in [n]$, the *reconciliation rule* of $\mathcal{S}$ and q is the (full) TGD of the form $Q_1(\bar{x}_1), \ldots, Q_n(\bar{x}_n) \rightarrow Q(\bar{x})$. In other words, the reconciliation rule of $\mathcal{S}$ and q performs the inverse of the decomposition, i.e., it combines the certain answers of each member of $\mathcal{S}$, so to restore the certain answers of the CQ q.

4.1 The Algorithm WardedRewrite

In the following, for a CQ $q(\bar{x})$ of the form $Q(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$, we use $\mathsf{rule}(q)$ to denote the (full) TGD of the form $R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n) \rightarrow Q(\bar{x})$. Moreover, for a full TGD σ of the form $R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n) \rightarrow Q(\bar{x})$, with Q different from each R_i, we use $\mathsf{cq}(\sigma)$ to denote the CQ of the form $Q(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$. Moreover, for a CQ q, we use $\mathsf{can}(q)$ to denote the *canonical form* of q, i.e., the CQ obtained from q where each variable x of q that appears, from left to right in q, as the i-th variable, is replaced with the fresh variable w_i. For example, if q is the CQ $Q(x, y, y) \leftarrow R(y, z, x), S(x, z)$, then $\mathsf{can}(q)$ is the CQ $Q(w_1, w_2, w_2) \leftarrow R(w_2, w_3, w_1), S(w_1, w_3)$. Clearly, two CQs q_1 and q_2 which are the same, up to variable renaming, are such that $\mathsf{can}(q_1) = \mathsf{can}(q_2)$. We extend the above notations to sets in the natural way. Finally, given two CQs q_1 and q_2, we write $q_1 \approx q_2$ iff $\mathsf{can}(q_1)$ and $\mathsf{can}(q_2)$ are the same up to head predicate renaming. We now have all the notions we need to present our rewriting algorithm, dubbed WardedRewrite, which we report in Algorithm 1.

Algorithm 1: WardedRewrite

```
Input: An OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in$ WARDED
Output: A Datalog rewriting of $\mathcal{O}$
1  $\mathcal{Q}_{\text{current}} := \{\mathsf{can}(q)\}$; $\mathcal{Q}_{\text{explored}} := \emptyset$; $\mathcal{Q}_{\text{dec}} := \emptyset$; $\mathcal{R} := \emptyset$;
2  while $\mathcal{Q}_{\text{current}} \neq \emptyset$ do
3      $\mathcal{Q}_{\text{new}} := \emptyset$;
4      foreach $q' \in \mathcal{Q}_{\text{current}}$ do
5          if a decomposition of $q'$ w.r.t. $\Sigma$ exists then
               // Decomposition step
6              $\mathcal{S} := \mathsf{decomposeOptimal}(q', \Sigma)$;
7              $\rho := \mathsf{reconciliationRule}(q', \mathcal{S})$;
8              foreach $q'' \in \mathcal{S}$ do
9                  if there is $q_{\text{alt}} \in \mathcal{Q}_{\text{dec}}$ with $q_{\text{alt}} \approx q''$ then
10                     Replace $\mathsf{headPred}(q'')$ with
                         $\mathsf{headPred}(q_{\text{alt}})$ in $\mathsf{body}(\rho)$;
11                 else
12                     $\mathcal{Q}_{\text{dec}} := \mathcal{Q}_{\text{dec}} \cup \{\mathsf{can}(q'')\}$;
13                     $\mathcal{Q}_{\text{new}} := \mathcal{Q}_{\text{new}} \cup \{\mathsf{can}(q'')\}$;
14             $\mathcal{R} := \mathcal{R} \cup \{\rho\}$;
15         else
               // Resolution step
16             foreach $\sigma \in \Sigma$ do
17                 foreach MGCU $M$ of $q'$ with $\sigma$ do
18                     $q'' := \mathsf{resolv}_{\sigma,M}(q')$;
19                     $\mathcal{Q}_{\text{new}} := \mathcal{Q}_{\text{new}} \cup \{\mathsf{can}(q'')\}$;
20             $\mathcal{R} := \mathcal{R} \cup \{\mathsf{rule}(q')\}$;
21         $\mathcal{Q}_{\text{explored}} := \mathcal{Q}_{\text{explored}} \cup \{q'\}$;
22     $\mathcal{Q}_{\text{current}} := \mathcal{Q}_{\text{new}} \setminus \mathcal{Q}_{\text{explored}}$;
23 return $(\mathsf{headPred}(q), \mathcal{R})$;
```

Consider an OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in$ WARDED. The main idea behind the algorithm WardedRewrite is to iteratively apply one of two steps: decomposition and resolution. The set $\mathcal{Q}_{\text{current}}$ collects the CQs that need to be processed, and it initially contains the (canonical version of) the input query q, while all other structures are initially empty.

For each query in $q' \in \mathcal{Q}_{\text{current}}$, the goal is to try first to decompose q' into smaller subqueries (line 5). If this is possible, then the algorithm constructs the decomposition $\mathcal{S}$ of q' w.r.t. Σ, and the corresponding reconciliation rule ρ (lines 6-7). Ideally, all queries in $\mathcal{S}$ should then be added to the set $\mathcal{Q}_{\text{new}}$ for later processing, and ρ added to the set $\mathcal{R}$ of rules of the final Datalog rewriting. However, to guarantee termination, the algorithm first checks, for each $q'' \in \mathcal{S}$, whether any previous decomposition step has already produced a query q_{alt} with $q_{\text{alt}} \approx q''$ (line 9). If this is the case, there is no need to add q'' to $\mathcal{Q}_{\text{new}}$, as an identical version has already been processed. Hence, it is enough to replace the occurrence of $\mathsf{headPred}(q'')$ in the body of ρ with $\mathsf{headPred}(q_{\text{alt}})$ (line 10). If no such a query q_{alt} is found, then the CQ q'' is really new and its canonical version is added to the set $\mathcal{Q}_{\text{new}}$ for later processing, as well as to the set $\mathcal{Q}_{\text{dec}}$, in order to remember that q'' has been produced by a decomposition step (lines 12-13).

Then, the (modified) rule ρ is added to the set $\mathcal{R}$ of all rules of the Datalog rewriting (line 14).

If q' cannot be decomposed, it is instead unfolded by constructing all possible σ-resolvents of q', for all TGDs $\sigma \in \Sigma$; this is a single resolution step (lines 16-19), which produces a set of new queries that are also added to $\mathcal{Q}_{\text{new}}$ for later processing. The (rule version of) q' is then added to the set of rules $\mathcal{R}$ of the final Datalog rewriting (line 20).

After q' has gone through a decomposition or a resolution step, it is marked as explored in line 21 by adding q' to the set $\mathcal{Q}_{\text{explored}}$ of processed queries. After all queries in $\mathcal{Q}_{\text{current}}$ have been explored, $\mathcal{Q}_{\text{current}}$ is updated in line 22 with all new queries in $\mathcal{Q}_{\text{new}}$ that have not been explored yet (i.e., that are not in $\mathcal{Q}_{\text{explored}}$), and the process repeats.[2] The algorithm terminates when no new queries (up to variable renaming) have been produced. The final Datalog rewriting is $(\mathsf{headPred}(q), \mathcal{R})$ and it is returned in line 23.

We can show the following result.

Theorem 5. *For every OMQ $\mathcal{O} = (q, \Sigma)$, with $\Sigma \in$ WARDED, WardedRewrite terminates with input $\mathcal{O}$, and outputs a Datalog rewriting of $\mathcal{O}$.*

5 Implementation and Experiments

In this section we discuss how we implemented the algorithm WardedRewrite. Moreover, we present an experimental analysis assessing two aspects: how fast is of our implementation of WardedRewrite to produce the Datalog rewriting, and how efficient existing Datalog-based engines are at evaluating the Datalog query produced by our algorithm.

Implementation

Regarding the implementation, the main challenge is keeping the set of rules $\mathcal{R}$ of the final Datalog rewriting as small as possible. This is in order to not hinder the performance of answering the query using off-the-shelf Datalog engines. As an optimization, each time a rule is added to the set $\mathcal{R}$, we consider all predicates $Q_1, \ldots, Q_n$ occurring in the head of some rule of $\mathcal{R}$, and for each $i \in [n]$, consider the maximal subset $\mathcal{R}_{Q_i}$ of TGDs with head predicate Q_i occurring in $\mathcal{R}$ such that $(Q_i, \mathcal{R}_{Q_i})$ is a UCQ. Then, we minimize each set $\mathcal{R}_{Q_i}$ independently by constructing, for each $i \in [n]$, the smallest subset $\mathcal{R}'_{Q_i}$ of $\mathcal{R}_{Q_i}$ such that $(Q_i, \mathcal{R}'_{Q_i})$ and $(Q_i, \mathcal{R}_{Q_i})$ are equivalent; the latter boils down to checking equivalence between UCQs, which is decidable [Abiteboul *et al.*, 1995].

Experimental Analysis

We now move to our experimental analysis. In particular, our goal is to experimentally evaluate different aspects of our implementation of WardedRewrite. In particular, we are interested to answer two key questions.

Question 1: *Given an OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in$ WARDED as input to our implementation of* WardedRewrite*, how do key parameters of $\mathcal{O}$ affect its running time and scalability?*

Answering the above will allow us to provide insights on the practical applicability of our implementation and its limits, as well as providing conclusions on the expected behaviour of the algorithm, depending on the input OMQ.

The next natural question is whether the Datalog rewriting produced by our implementation can be actually used for query answering purposes by exploiting existing Datalog-based reasoning systems.

Question 2: *Given a database D, an OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in$ WARDED, and the Datalog rewriting $\mathcal{D}_{\mathcal{O}}$ produced by* WardedRewrite *with input $\mathcal{O}$, how efficient existing Datalog-based reasoning engines are in evaluating $\mathcal{D}_{\mathcal{O}}$ over D?*

Towards answering our key experimental questions, we need a way to stress test both our rewriting algorithm WardedRewrite, as well as the Datalog-based reasoning engines we consider. For this, we generate a large pool of synthetic database-OMQ pairs $(D, \mathcal{O})$, which we call *scenarios*, where $\mathcal{O} = (q, \Sigma)$ and $\Sigma \in$ WARDED, using the IWARDED benchmark generator from [Atzeni *et al.*, 2022]. IWARDED allows to generate scenarios $(D, \mathcal{O})$, with $\mathcal{O} = (q, \Sigma)$ and $\Sigma \in$ WARDED, by considering different parameters. In order to keep our experimental evaluation manageble, we will focus on a few, key parameters that mostly affect either the size of the database, or the main structure of the ontology Σ.

Ontology Parameters. IWARDED generates OMQs (q, Σ), where Σ is warded, and the TGDs have a single head atom and at most two atoms in the body. Moreover, TGDs are generated so to induce multiple sequences of TGDs of the form $\sigma_1, \ldots, \sigma_n$, where the atom produced during the construction of the chase by the TGD σ_i contributes to trigger the TGD σ_{i+1}, for each $i \in [n-1]$. We can control the shape of these sequences by choosing two main parameters.

The *average recursion length*, denoted `avg_recursion`, specifies the average length of *recursive TGD sequences*, i.e., sequences of TGDs $\sigma_1, \ldots, \sigma_n$ in Σ where the TGD σ_n can further trigger σ_1. The length n of the sequence is chosen using a Guassian distribution with `avg_recursion` as the mean, and variance 1.0. The *number of left-right-join recursions*, denoted `num_lrj`, specifies the number of TGDs that IWARDED will add to Σ that mention two predicates in their body that both belong to a recursive sequence.

Database Parameters. As for the database parameters, we consider the *number of facts per predicate*, denoted `db_facts`: IWARDED will add `db_facts` random facts to the database D for each predicate of D; the number of predicates of D is chosen automatically by IWARDED so to preserve the values of the ontology-related parameters above.

Query Parameters. There are no parameters we can control regarding the construction of the CQ q. In particular, IWARDED only supports the construction of CQs q of the form $Q(\bar{x}) \leftarrow R(\bar{x})$, i.e., CQs with a single body atom whose tuples are propagated as output of the query; IWARDED automatically adds auxiliary TGDs to Σ so that all the TGD sequences added to Σ "feed" the predicate R with tuples. This guarantees that the certain answers of q are never empty.

Test Scenarios. In generating our scenarios, we consider 21 different values for the number of left-right-join recursions `num_lrj` and the average recursion length `avg_recursion`, spanning from 0 to 400 with steps of

[2]Note that by considering the canonical version of the CQs, performing the set-theoretic difference is enough to exclude from $\mathcal{Q}_{\text{new}}$ all queries that have already been considered.

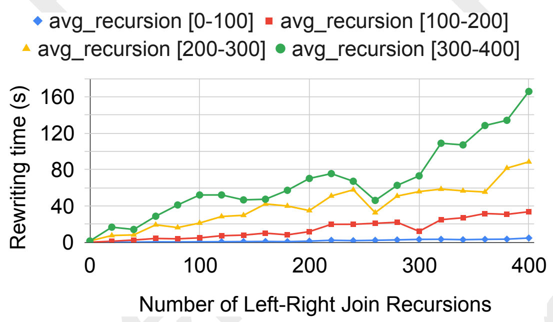

Figure 1: Rewriting time vs. number of left-right join recursions.

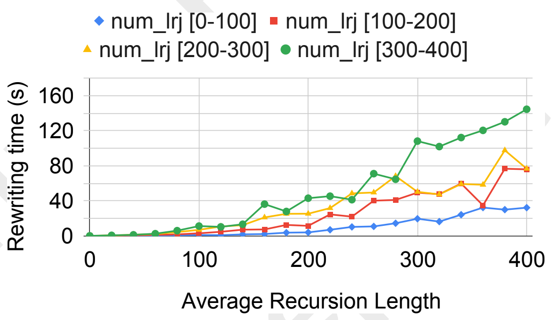

Figure 2: Rewriting time vs. average recursion length.

20 each. That is, we consider the values in the set $\{0, 20, 40, \ldots, 400\}$ for the above two parameters. We point out that the largest values in this set are quite extreme for an average real-world ontology, and we consider such a large interval in order to strees test our rewriting algorithm, and assess its practical applicability. Regarding the database generation, we consider a fixed value of $100k$ facts per predicate (db_facts). Then, for each combination $i, j \in \{0, 20, 40, \ldots, 400\}$, we generated a scenario, denoted $S[i, j]$, of the form $(D, \mathcal{O})$ with $\mathcal{O} = (q, \Sigma)$, by running IWARDED with input num_lrj $= i$, avg_recursion $= j$, and db_facts $= 100k$, obtaining a total of 441 scenarios.

Experimental Setup. For the experiments, we used an Amazon Elastic Compute Cloud (EC2) instance with an Intel(R) Xeon(TM) Platinum 8000-series @3.6 GHz, 16 GB of RAM, running Amazon Linux 2 LTS Candidate. We implemented the algorithm WardedRewrite in Java 22, and used openJDK 22 for its execution, while the Datalog-based reasoning engines have been executed in their default configuration.

5.1 Evaluating the Rewriting Algorithm

In this section we address our first experimental question. For each of the 441 scenarios $S[i, j] = (D, \mathcal{O})$ with $\mathcal{O} = (q, \Sigma)$ and with $i, j \in \{0, 20, 40, \ldots, 400\}$, we executed our implementation of WardedRewrite with input the OMQ $\mathcal{O}$. For each execution we collected the total running time.

Figure 1 (resp., Figure 2) shows the average running time of our implementation of WardedRewrite, for each value of the parameter num_lrj (resp., avg_recursion), where the average is computed across all test scenarios having avg_recursion (resp., num_lrj) in one of 4 intervals, i.e., [0-100], [100-200], [200-300], [300-400].

As we can see from Figure 1, the number of left-right-join recursions indeed has an impact on the running time of our implementation of WardedRewrite. We can see that for lower intervals of avg_recursion, even the highest value of 400 left-right-join recursions (num_lrj) leads to quite fast executions on average; e.g., 5 seconds for the average recursion length interval [0-100], and 33 seconds for [100-200]; the highest interval [300-400] leads to some minutes with the higher values of num_lrj. The trend is somehow expected, since the deeper TGDs can chain, and the more queries WardedRewrite needs to produce via resolution steps.

However, we can see that the increase in running time is mostly linear as num_lrj increases, on average, suggesting that the number of left-right-join recursions does not severely impact the efficiency of our implementation.

Moving to the analysis w.r.t. the parameter avg_recursion, we observe steeper trends in Figure 2. The reason is because, assuming we focus on a certain value of the parameter num_lrj, at each resolution step, there is a certain chance that a new CQ q' with two body atoms, both having a recursive predicate, is produced. The number of times this happens is proportional to avg_recursion. Hence, the runtime of WardedRewrite increases exponentially w.r.t. avg_recursion.

Nonetheless, the above analysis shows that even the most demanding scenarios (e.g., $S[400, 400]$), which are rather extreme and unlikely to occur in practice, can be solved in the order of some minutes. This is quite encouraging, considering that computing the rewriting is usually done ”offline”, since this is a database-independent task.

Validation with Other Benchmarks. To place our analysis above in perspective, we also executed our implementation of WardedRewrite on another benchmark of OMQs from the literature that uses warded ontologies from [Atzeni *et al.*, 2022], called *structural scenarios*. These scenarios were meant to stress test different aspects of a Datalog-based reasoning engine, and we are not aware of other existing benchmarks collecting OMQs whose ontologies are (strictly) warded. Executing WardedRewrite over the OMQs of the structural scenarios revealed that the Datalog rewriting can be constructed very efficiently, with the worst case requiring about 250 ms.

5.2 Impact of the Rewriting on Reasoning

In this section we move to our second experimental question, i.e., how off-the-shelf Datalog-based reasoning engines behave when they evaluate the Datalog query produced by our implementation of WardedRewrite.

Engines. We consider different engines from the literature. In particular, (the parallel version of) **Vadalog** [Bellomarini *et al.*, 2024],**VLog** [Urbani *et al.*, 2018], **DLV** [Adrian *et al.*, 2018] and its extension **DLV-E** [Leone *et al.*, 2019], as well as **InteGraal** [Baget *et al.*, 2015]. All the above Datalog-based reasoning engines support the evaluation of standard Datalog queries. They also support OMQ Answering, but to a certain degree. For example, Vadalog supports OMQs over warded ontologies, but only with atomic CQs, using a specialized procedure rather than a Datalog rewriting approach. On the other hand, for example, DLV-E works on OMQs over

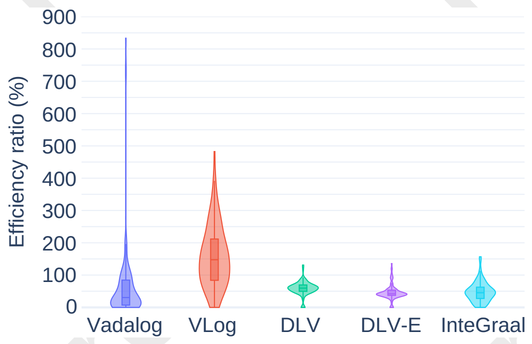

Figure 3: Efficiency ratio over the test scenarios $S[i,j]$, with $i,j \in \{0,20,40,\dots,200\}$.

shy ontologies, uncomparable to warded, and relies on a refined version of the chase, called parsimonious. Nonetheless, all engines above support OMQ Answering when the ontology Σ is such that, for every database D, $\mathsf{chase}(D,\Sigma)$ is finite. Since all our test scenarios turned out to have an ontology with this property, every engine we consider not only is able to evaluate Datalog queries, but it is also able to take directly as input any of our test scenarios $S[i,j] = (D,\mathcal{O})$, with $i,j \in \{0,20,40,\dots,400\}$, and effectively compute the certain answers $\mathsf{ans}(\mathcal{O},D)$.

Answering Question 2. For each Datalog-based reasoning engine E we consider and each pair $i,j \in \{0,20,40,\dots,200\}$, we asked E to compute the certain answers $\mathsf{ans}(\mathcal{O},D)$, and collected the time t_{certain} required by the engine. Then, we asked E to compute the answers $\mathcal{D}_{\mathcal{O}}(D)$ of the Datalog rewriting $\mathcal{D}_{\mathcal{O}}$ of $\mathcal{O}$ we previously computed using WardedRewrite in Section 5.1, and collected the time t_{rew} required. Note that we only considered the test scenarios with `num_lrj` and `avg_recursion` in the range $\{0,20,40,\dots,200\}$, since for higher values most of the engines run out of memory. Moreover, each time an engine was executed, it was given a timeout of 5 minutes.

Then, we measure the efficiency of an engine E over a scenario $S[i,j]$ via the *efficiency ratio* $\mathcal{R}_{E,S[i,j]} = \frac{t_{\text{rew}}}{t_{\text{certain}}}$. The reason is that our goal is not a comparison of the above engines, but rather asses for each engine, whether the Datalog rewriting produced by our implementation of WardedRewrite can be evaluated efficiently, *relative* to the capabilities and the limits of the engine itself; $\mathcal{R}_{E,S[i,j]} < 1$ means that it is "relatively" efficient for the engine to evaluate the Datalog query produced by WardedRewrite, while $\mathcal{R}_{E,S[i,j]} > 1$ means it is faster for the engine to evaluate the original OMQ directly.

Figure 3 reports, for each engine E, a violin plot collecting the efficiency ratios $\mathcal{R}_{E,S[i,j]}$, for all $i,j \in \{0,20,40,\dots,200\}$, expressed as a percentage—e.g. 50% corresponds to an efficiency ratio equal to 0.5. In each violin plot, the bottom and the top borders of the inner box denote the first and third quartile; the line inside the box denotes the median efficiency ratio; the area surrounding the box denotes the distribution of all the efficiency ratios.

Regarding DLV, DLV-E, and InteGraal, we can see that most of the efficiency ratios are below 100%, with DLV-E and InteGraal having most of them around 50%. This indicates that the Datalog rewriting produced by our implementation of WardedRewrite can be evaluated efficiency by these systems, and in most cases, the rewriting is even desirable w.r.t. evaluating the original OMQ itself. The main reason for this trend is that all these engines implement certain answering by means of an explicit construction of the chase instance (or a variant of it). This means that when considering the original OMQ $\mathcal{O}$ of a test scenario, the presence of existential variables in $\mathcal{O}$ forces the engine to introduce a large number of nulls, while constructing the chase, and thus results in a large chase instance. On the other hand, the Datalog rewriting avoids the construction of all such intermediate atoms, reducing the size of the chase instance considerably.

On the opposite side, VLog is not very efficient, in most cases, when evaluating the Datalog rewriting w.r.t. evaluating the original OMQ directly, with almost 75% of the scenarios having an efficiency ratio over 100%, with the worst efficiency ratio around 480%. The reason for this is that, differently from DLV, DLV-E, and InteGraal, VLog is less affected by the size of the chase instance, when evaluating the original OMQ, due to a custom column-based representation of the produced atoms, and thus it does not usually gain much in computing the certain answers via the Datalog rewriting.

Finally, Vadalog is in a middle ground, with roughly 80% of the test scenarios having an efficiency ratio below 50%, while for the remaining 20% of the test scenarios, the Datalog rewriting performs poorly, compared to the original OMQ, with high efficiency ratios up to 830%. This could be due to the fact that for some of the scenarios, when Vadalog evaluates the original OMQ of the scenario, it is able to stop much earlier in the contruction of the chase instance, since it internally constructs only an isomorphim-closed portion of the chase (see [Bellomarini *et al.*, 2018] for more details).

Validation and Scalability Analysis. We employed once again the structural scenarios from [Atzeni *et al.*, 2022] in order to validate our analysis performed above, and indeed the above trends are confirmed also in these scenarios. Moreover, it turned out that for most of the engines, the efficiency ratio even improves (i.e., decreases) as the database size increases.

End-to-end Reasoning. We point out that, besides VLog, for which we have already shown that evaluating the original OMQ is faster than evaluating the Datalog rewriting, in most cases, for all other engines, most of the efficiency ratios (88% on average) that are below 100% remain below 100% even if the time for building the rewriting is taken into account.

6 Conclusion

Our work is a first step in making OMQ Answering under warded ontologies viable in practice, without relying on closed systems, as we have shown that constructing the Datalog rewriting is quite efficient, and most off-the-shelf Datalog-based engines are very capable in evaluating it.

A possible direction for improvement would be parallelizing WardedRewrite by exploiting the fact that all intermediate queries produced by the algorithm can be processed indepependently from each other, thanks to the optimal decomposition employed by the algorithm.

Ethical Statement

There are no ethical issues.

Acknowledgments

We thank the anonymous referees for their feedback. This work was funded by the European Union - Next Generation EU under the MUR PRIN-PNRR grant P2022KHTX7 "DISTORT", by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013, 10.47379/NXT22018], and by the Austrian Science Fund (FWF) 10.55776/COE12. The Vadalog Parallel system as presented here is the intellectual property of Prometheux Limited.

References

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

[Adrian *et al.*, 2018] Weronika T. Adrian, Mario Alviano, Francesco Calimeri, Bernardo Cuteri, Carmine Dodaro, Wolfgang Faber, Davide Fuscà, Nicola Leone, Marco Manna, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV: advancements and applications. *Künstliche Intell.*, 32(2-3):177–179, 2018.

[Artale *et al.*, 2009] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The dl-lite family and relations. *Journal of Artificial Intelligence Research*, 36:1–69, 2009.

[Atzeni *et al.*, 2022] Paolo Atzeni, Teodoro Baldazzi, Luigi Bellomarini, and Emanuel Sallinger. iwarded: A versatile generator to benchmark warded datalog+/- reasoning. In *RuleML*, pages 113–129, 2022.

[Baader *et al.*, 2003] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[Baader *et al.*, 2005] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL envelope. In *IJCAI*, pages 364–369, 2005.

[Baget *et al.*, 2011] Jean-François Baget, Marie-Laure Mugnier, Sebastian Rudolph, and Michaël Thomazo. Walking the complexity lines for generalized guarded existential rules. In *IJCAI*, pages 712–717, 2011.

[Baget *et al.*, 2015] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. Graal: A toolkit for query answering with existential rules. In *RuleML*, pages 328–344, 2015.

[Bellomarini *et al.*, 2018] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The vadalog system: datalog-based reasoning for knowledge graphs. *VLDB*, 11(9):975–987, 2018.

[Bellomarini *et al.*, 2022] Luigi Bellomarini, Davide Benedetto, Georg Gottlob, and Emanuel Sallinger. Vadalog: A modern architecture for automated reasoning with large knowledge graphs. *Inf. Syst.*, 105(C), 2022.

[Bellomarini *et al.*, 2024] Luigi Bellomarini, Davide Benedetto, Matteo Brandetti, Emanuel Sallinger, and Adriano Vlad. The vadalog parallel system: Distributed reasoning with datalog+/-. *Proc. VLDB Endow.*, 17(13):4614–4626, 2024.

[Benedikt *et al.*, 2022] Michael Benedikt, Maxime Buron, Stefano Germano, Kevin Kappelmann, and Boris Motik. Rewriting the infinite chase. *VLDB*, 15(11):3045–3057, 2022.

[Berger *et al.*, 2022] Gerald Berger, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. The space-efficient core of vadalog. *ACM Trans. Database Syst.*, 47(1), 2022.

[Calautti and Pieris, 2021] Marco Calautti and Andreas Pieris. Semi-oblivious chase termination: The sticky case. *Theory Comput. Syst.*, 65(1):84–121, 2021.

[Calautti *et al.*, 2015] Marco Calautti, Georg Gottlob, and Andreas Pieris. Chase termination for guarded existential rules. In *PODS*, pages 91–103, 2015.

[Calautti *et al.*, 2016] Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Exploiting equality generating dependencies in checking chase termination. *VLDB*, 9(5):396–407, 2016.

[Calautti *et al.*, 2022] Marco Calautti, Georg Gottlob, and Andreas Pieris. Non-uniformly terminating chase: Size and complexity. In *PODS*, pages 369–378, 2022.

[Calì *et al.*, 2012a] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general Datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83, 2012.

[Calì *et al.*, 2012b] Andrea Calì, Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.*, 193:87–128, 2012.

[Calì *et al.*, 2013] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res.*, 48:115–174, 2013.

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

[Carral *et al.*, 2017] David Carral, Irina Dragoste, and Markus Krötzsch. Restricted chase (non)termination for existential rules with disjunctions. In Carles Sierra, editor, *IJCAI*, pages 922–928, 2017.

[Fagin *et al.*, 2005] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[Gottlob and Pieris, 2015] Georg Gottlob and Andreas Pieris. Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue. In *IJCAI*, pages 2999–3007, 2015.

[Gottlob *et al.*, 2014] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Query rewriting and optimization for ontological databases. *ACM Trans. Database Syst.*, 39(3):25:1–25:46, 2014.

[Grahne and Onet, 2018] Gösta Grahne and Adrian Onet. Anatomy of the chase. *Fundam. Inf.*, 157(3):221–270, 2018.

[Greco *et al.*, 2011] Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Stratification criteria and rewriting techniques for checking chase termination. *VLDB*, 4(11):1158–1168, 2011.

[König *et al.*, 2015] Mélanie König, Michel Leclère, Marie-Laure Mugnier, and Michaël Thomazo. Sound, complete and minimal ucq-rewriting for existential rules. *Semantic Web*, 6(5):451–475, 2015.

[Leone *et al.*, 2019] Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. Fast query answering over existential rules. *ACM Trans. Comput. Logic*, 20(2), 2019.

[Lutz and Sabellek, 2022] Carsten Lutz and Leif Sabellek. A complete classification of the complexity and rewritability of ontology-mediated queries based on the description logic el. *Artificial Intelligence*, 308:103709, 2022.

[Poggi *et al.*, 2008] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. *Linking Data to Ontologies*, page 133–173. Springer Berlin Heidelberg, 2008.

[Spezzano and Greco, 2010] Francesca Spezzano and Sergio Greco. Chase termination: A constraints rewriting approach. *VLDB*, 3(1):93–104, 2010.

[Urbani *et al.*, 2018] Jacopo Urbani, Markus Krötzsch, Ceriel J. H. Jacobs, Irina Dragoste, and David Carral. Efficient model construction for horn logic with vlog. In *IJCAR*, pages 680–688, 2018.