UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Doctoral Programme in
Information Engineering and Computer Science

Thesis Title

Practical Rewriting Techniques for Warded Ontology-Mediated Queries

Hebatalla Mohamed Wagih

Advisor
Prof. Marco Calautti
Università di Milano

October 2025

Dedication

To an angel in heaven,
My beloved mother, whose love, strength, and guidance continue to inspire me every day. Though you are no longer here to share this moment with me, your spirit has been beside me through every step of this journey. I promised you I would be strong and here I am, standing tall, with this work as proof of that promise. This work is a tribute to your memory and everything you believed I could become.
For all you gave, for all you were, and for all you still are to me, thank you, Mama.

Acknowledgment

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Marco Calautti, whose continuous support, guidance, and kindness have been invaluable throughout this journey. You have been not only a mentor and role model, but also a true friend. You were always patient and encouraging, always there for me with your wisdom, your time and your belief in my abilities. I will always remember that, before I even met you, I saw you and my late mother together in a dream. she was smiling peacefully, as if reassuring me that you would be the one to guide me. That dream filled me with calm and confidence, and I have always felt it was her way of blessing this journey under your supervision. I will forever be grateful that life made it a reality.

My heartfelt thanks go to my family, especially my father and sister, who have been my pillars of strength. This PhD is not only my dream, but also my father's, a dream he envisioned for me as a tribute to my late mother. Your love, prayers and unwavering faith in me gave me the courage to keep moving forward, even in the most difficult times.

To my sister, you have always been more than a sister, you have been my greatest support, sharing every emotion of my journey. I will always cherish your love and care. I will never forget how happy we were during your visit to Italy, celebrating the small moments that made this path so meaningful. I am endlessly grateful for everything you have done for me.

I am deeply grateful to my two dear friends and sisters, Basma Hathout and Sara Abdelhamid, whose love and friendship have been a constant source of comfort and strength. Your homes became my second homes, always open, always warm, always welcoming. I also extend my heartfelt thanks to your families for their kindness and support throughout my journey. Thank you for being part of my life and for standing by me through every step of this long road.

I would also like to express my sincere gratitude to Professor Emanuel Sallinger, under whose guidance I had the opportunity to work at the Knowledge Graph Lab at TU Wien during my period abroad. His insights and mentorship greatly enriched my research experience.

Finally, to my beloved late mother, the truest source of my strength and the reason behind every step I take. Your love still guides me, your voice still comforts me, and your memory lives in everything I do. I hope I have made you proud, Mama. You will forever be my light, my peace, and my greatest inspiration.

Abstract

Existential rules, a.k.a. tuple-generating dependencies (TGDs), form a well-established formalism for specifying ontologies, i.e., logical descriptions of a domain of interest. One of the main tasks in this context is Ontology-Mediated Query (OMQ) Answering. That is, given an Ontology-Mediated Query $\mathcal{O} = (q, \Sigma)$, comprising a TGD-based ontology Σ and a query q (usually a conjunctive query), find all the answers to q that can be entailed from the logical theory $D \wedge \Sigma$, where D is a given database encoded as a conjunction of facts. It is well-known that even checking if a tuple is an answer to an OMQ $\mathcal{O}$ is undecidable, due to the high expressiveness of TGDs. Hence, different classes of TGD-based ontologies have been introduced in the literature with the goal of restoring decidability. Such classes can be categorized in two main groups, depending on the strategy employed to obtain decidability. The first group comprises ontologies Σ for which an explicit and finite *materialization of the facts that can be derived using the database and the TGDs in Σ can be constructed; such a materialization is usually achieved by means of the well-known chase procedure. The second group comprises ontologies Σ guaranteeing that any OMQ $\mathcal{O} = (q, \Sigma)$ can be rewritten to a query q' in some other (decidable) query language such that for every database D, the answers of $\mathcal{O}$ over D coincide with the answers of q' over D.*

From the second such a group, warded *ontologies recently emerged as a well-behaved class of TGD-based ontologies, striking a good balance between expressive power and computational complexity of Ontology-Mediated Query Answering. The theoretical foundations of OMQ Answering over warded ontologies are by now well-understood, where the problem is known to be solvable in polynomial time, in data complexity, i.e., when only the database is considered as part of the input. In particular, it is well-known that OMQ Answering under warded ontologies is Datalog-rewritable, i.e., given an OMQ over a warded ontology, it is possible to construct a query written in Datalog, a well known database query language introduced in the 80's, such that for every database, the answers to the OMQ over the database coincide with the answers of the Datalog query over the database alone. Unfortunately, very few efforts exist in the literature that exploit such a rich theory for building practical query answering algorithms under warded ontologies.*

The main contribution of this thesis is to fill the above gap by designing a novel Dat-

alog rewriting algorithm for OMQs over warded ontologies which is more amenable to practical implementations, as well as providing an implementation and an experimental evaluation, with the aim of understanding which key input parameters affect the performance of this approach, and what are its limits. Furthermore, the thesis will show how to use the above analysis to draw key insights on the practical applicability of a rewriting-based approach for query answering under warded ontologies, when using off-the-shelf Datalog-based engines.

Keywords

[Tuple-generating dependencies, Ontology-mediated queries, Datalog rewriting, Warded ontologies]

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

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.

Example 1.1.1. Consider the TGD-based ontology $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$, where the TGDs therein are defined as follows:

$$
\begin{aligned}
\sigma_1 :&\ \forall x \forall y & & Follows(x, y) \wedge Follows(y, x) \rightarrow Friends(x, y),\\
\sigma_2 :&\ \forall x \forall y & & Friends(x, y) \rightarrow Friends(y, x),\\
\sigma_3 :&\ \forall x \forall y \forall z & & Friends(x, y) \wedge Friends(y, z) \rightarrow Friends(x, z),\\
\sigma_4 :&\ \forall x \forall y & & Friends(x, y) \rightarrow \exists z\, Likes(x, z) \wedge Likes(y, z).
\end{aligned}
$$

The TGD σ_1 states that if a person x follows another person y, and vice versa, then x and y must be friends, while σ_2 and σ_3 specify that friendship is a symmetric and transitive relationship. Finally, σ_4 states that if two people x and y are friends, then

there must be some topic z which both like.

Assume now that the database D is the following set, which contains facts about the system:

$$\{Follows(\text{alice}, \text{bob}), Follows(\text{bob}, \text{alice}),$$
$$Follows(\text{caterine}, \text{david}), Friends(\text{bob}, \text{caterine})\},$$

and assume we want to find all people x that have some interest z in common with some other person y. We can encode this query q as a conjunctive query (CQ):

$$Ans(x) \leftarrow \exists y, z\, Likes(x, z) \wedge Likes(y, z),$$

where the predicate Ans is the output predicate of the query, which will collect the answers.

Computing the answers of q means computing the answers of the OMQ $\mathcal{O} = (q, \Sigma)$ over the database D. That is, the semantics are given in terms of models of the knowledge base (D, Σ), i.e., sets of atomic statements that contain D and satisfy the TGDs in Σ. Then, the answers of $\mathcal{O}$ over D are all the tuples that are answers to q in every model of (D, Σ). Such answers are known as *certain answers* of $\mathcal{O}$ over D, which, in the case of this example, coincide with the set:

$$\{(\text{alice}), (\text{bob}), (\text{caterine})\}.$$

It is well-known that the problem of OMQ Answering is uncomputable, in general (e.g., see [30]). In other words, there is no general procedure that, given a database D and an OMQ $\mathcal{O}$, is able to compute the certain answers of $\mathcal{O}$ over D. Such a negative result holds even if the input database contains a single fact, and the CQ in the OMQ is atomic, i.e., it mentions a single atom in its right-hand side. Hence, it is widely recognized that the main source of complexity is the high expressive power of TGDs for defining ontologies.

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 [39, 48, 65, 25, 35, 21, 29, 22], as well as UCQ-rewritable languages [31, 33]. More expressive languages are the ones based on the notion of guardedness [31, 11, 19], and beyond [30]. We provide more details in Chapters 2 and 5. Furthermore, we point out that some of the above works

have been influenced by earlier works on Description Logics (DL), e.g., see [34, 61, 5] for UCQ-rewritable DL ontologies, and [7, 56] for guarded-like DL ontologies. We refer to [8] for a comprehensive overview.

1.2 Research Problem

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. Hence, most of the TGD-based ontology languages existing in the literature either provide efficient certain answers computation, but lack expressive power, or they are very expressive, but do not provide concrete means for developing real-world systems, or further, they do not come with publicly available implementations. To remedy the above situation, *warded* TGDs have been introduced in [43], 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 [43] by means of specialized alternating algorithms, while more recent works have shown that warded ontologies are actually Datalog-rewritable [20]. 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. Thus giving hope for an alternative means to implement OMQ Answering with warded ontologies in practice.

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.[1] One of the main reasons is that the results of [43, 20] are mostly theoretical in nature. In particular, the Datalog rewriting algorithm of [20] 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 [17, 16, 15], which supports a simpler version of OMQ Answering, where the CQ is atomic, and

[1] This is in contrast with other TGD-based ontology languages with similar goals to warded, such as *shy ontologies* [55], for which concrete implementations exist.

uses a procedure different from the ones of [43, 20]. The latter, together with the fact that Vadalog is available only after acquiring a license, severely limits the use of warded TGDs in practice.

1.3 Contributions

The goal of this thesis is to rectify the above state of affairs by providing a new Datalog rewriting algorithm for OMQs over warded ontologies, dubbed 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 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 WardedRewrite.

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

Published Results. We want to mention that the results presented in this thesis are mainly based on the following two works which have been accepted for publication:

> [18] Davide Benedetto, Marco Calautti, Hebatalla Hammad, Emanuel Sallinger, Adriano Vlad-Starrabba. A Datalog Rewriting Algorithm for Warded Ontologies. In the 34th International Joint Conference on Artificial Intelligence (IJCAI), 2025.
>
> [68] Heba M. Wagih and Marco Calautti. Enhancing ontological query-rewriting via parallelization. In SEBD, volume 3478, pages 401–409, 2023.

1.4 Roadmap

This thesis is organized as follows. Chapter 2 discusses the state of the art and related work on Ontology-Mediated Query Answering and ontological reasoning in general. Then, Chapter 3 introduces all the preliminaries that are used in our research study, including the basic notions of relational databases, ontologies, and OMQ Answering. Chapter 4 introduces the well-known Chase procedure as a canonical mean to produce universal models, a key notion that will be useful for the technical development of the

thesis. Then, tuple generating dependencies (TGDs) and well-known decidable classes of TGD-based ontologies are presented in Chapter 5, followed by a complete overview of warded ontologies in Chapter 6. In Chapter 7, our novel Datalog rewriting algorithm for warded ontologies is introduced, while detailed proofs of its correctness are presented in Chapter 8. Then, Chapter 9 presents a concrete implementation of the rewriting algorithm, discussing the key implementation choices that have been made to make the algorithm perform well in practice. Then, Chapter 10 discusses a comprehensive experimental evaluation of the implementation, and we finally draw conclusions and directions for future work in Chapter 11.

Chapter 2

State of the Art

As already stated in the Introduction, the problem of OMQ Answering, i.e., the problem of constructing the certain answers of an OMQ $\mathcal{O} = (q, \Sigma)$ over a given database is uncomputable, in general [30], and thus a long stream of literature has been developed in the last decades, devoted to identify fragments of TGD-based ontologies that not only guarantee computability of certain answers, but also strike a good balance between expressiveness of the ontology language, and the computational complexity of computing certain answers. In what follows, we provide an overview of the most notable attempts from the literature in restoring the computability of OMQ Answering by means of restricted ontology languages.

Acyclicity-based Languages. It is well known that the certain answers of an OMQ $\mathcal{O} = (q, \Sigma)$ over a database D can be equivalently defined as the set of tuples of constants that are the answers of the query q over a special kind of models of the knowledge base (D, Σ), known as *universal models*, which act as representatives of all other models of (D, Σ). A well-known procedure for constructing a universal model of (D, Σ) is the *chase*, which takes as input D and Σ, and outputs the universal model denoted by $\mathsf{chase}(D, \Sigma)$. Clearly, since OMQ Answering is uncomputable, in general, the instance $\mathsf{chase}(D, \Sigma)$ is not necessarily finite.

Example 2.0.1. Consider the database $D = \{Person(john)\}$ and the TGD-based ontology $\Sigma = \{\sigma_1, \sigma_2\}$ with:

$$\begin{aligned} \sigma_1 : \ & \forall x & & Person(x) \rightarrow \exists y \ HasFather(x, y), \\ \sigma_2 : \ & \forall x \forall y & & HasFather(x, y) \rightarrow Person(y). \end{aligned}$$

The chase procedure iteratively "triggers" the rules in Σ that are not currently satisfied by the facts from the database D and/or the facts produced in previous steps, in order

to produce new facts that together with previous facts satisfy the rule. When a triggered TGD has existential variables, then the chase "invents" a new value as witness for that variable, called a *labeled null*. Hence, denoting with $\mathsf{chase}^i(D, \Sigma)$, for some $i \geq 0$, the set of facts produced by the chase procedure at step i, with $\mathsf{chase}^0(D, \Sigma) = D$, the chase procedure will perform the following steps:

1. The TGD σ_1 is not satisfied by the facts in D, because no fact of the form $HasFather(john, \cdot)$ exists in D. Hence, the chase procedure triggers the TGD σ_1 using the fact $Person(john)$ from D, and produces the new instance $\mathsf{chase}^1(D, \Sigma) = D \cup \{HasFather(john, \bot_1)\}$, where $\bot_1$ is a labeled null.

2. Now, σ_2 is not satisfied by the facts in $\mathsf{chase}^1(D, \Sigma)$, and the chase triggers σ_2, producing $\mathsf{chase}^2(D, \Sigma) = \mathsf{chase}^1(D, \Sigma) \cup \{Person(\bot_1)\}$.

3. However, the presence of $Person(\bot_1)$ in $\mathsf{chase}^2(D, \Sigma)$ forces the chase procedure to trigger σ_1 again, producing the fact $HasFather(\bot_1, \bot_2)$, which will force triggering σ_2 again, producing the fact $Person(\bot_2)$, and so on.

It should be clear from the above example that the chase procedure does not terminate when executed over the above database D and the TGD-based ontology Σ.

Given the above observations, a first attempt at restoring computability of the OMQ Answering problem is to define TGD-based languages for which the chase procedure is always guaranteed to terminate, i.e., for every database D, there always exists some $i \geq 0$ such that $\mathsf{chase}^i(D, \Sigma) = \mathsf{chase}^{i+1}(D, \Sigma)$. We call this family of languages *acyclicity-based*, since they are all defined by restricting in some form the degree of cyclic triggering of TGDs during the execution of the chase procedure. Among these languages, the simplest is the one based on weakly-acyclic TGDs, which were originally defined in the context of Data Exchange [39]. More advanced and expressive acyclicity-based languages are based on stratification techniques [48], as well as rewriting-based approaches such as the ones of [65]. More recent contributions in this context are acyclicity-based languages that employ TGDs together with other ontological constructs, such as equality-generating dependencies, or extended forms of TGDs [25, 35], as well as languages that guarantee the termination of the chase, assuming that the underlying ontology already satisfies some well-behaved property, e.g., see [21, 29, 22, 54, 28]. A comprehensive overview of the chase can also be found in [48, 46].

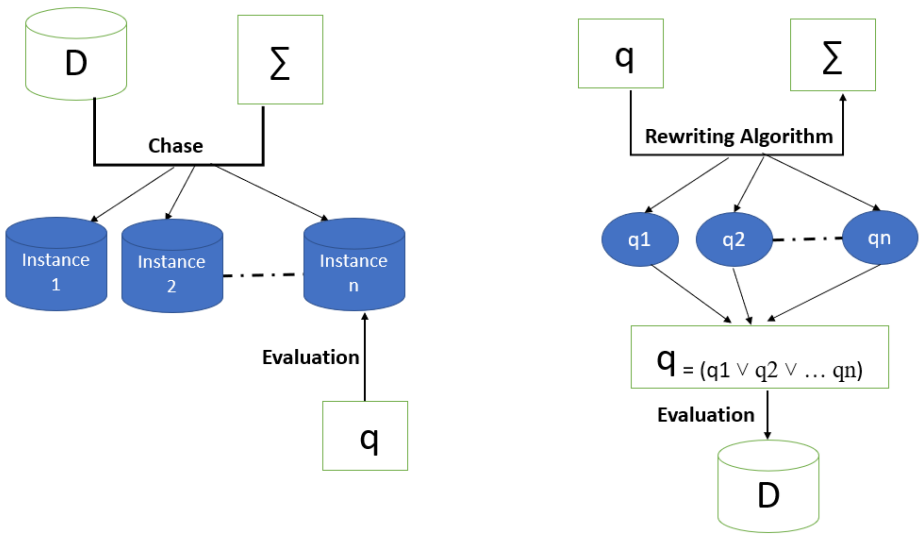

Figure 2.1: Chase vs. query rewriting for OMQ Answering.

All the above languages not only guarantee the computation of the certain answers, but also guarantee that the chase instance can be constructed efficiently, i.e., in polynomial time w.r.t. the size of the input database, and thus OMQ Answering can be solved efficiently. Some acyclicity-based languages in the literature provide more expressive power at the expense of efficiency. For example, the work of [49] provides a hierarchy of languages with increasing expressive power, such that computing the certain answers of OMQs using the language at level k of the hierarchy is in k-EXPTIME, w.r.t. the size of the input database.

We point out that this line of research on acyclicity-based languages is also related to the definition of well-behaved languages in *logic programming*, e.g., see [27, 23, 26, 24]. A key limitation of the above languages is that they do not allow reasoning over infinite structures, which is a key feature of ontological modeling.

UCQ-rewritable Languages. To overcome the limitations of the acyclicity-based languages discussed before, different contributions from the literature have focused on designing so-called UCQ-rewritable ontology languages. That is, for a given OMQ $\mathcal{O} = (q, \Sigma)$, if the ontology Σ falls in the language, the guarantee is that the OMQ $\mathcal{O}$ can be rewritten to a first-order query q', and in particular, a union of conjunctive queries (UCQ), such that for *every* database D, the certain answers of $\mathcal{O}$ over D can be retrieved by simply evaluating q' over D, without involving the ontology Σ. UCQs encode the fragment of first-order logic corresponding to the select-from-where-union fragment of SQL (more details on UCQs will be given in Chapter 3). Hence, the query

rewriting approach allows the implementation of OMQ Answering by exploiting existing relational database systems, and thus achieve efficient evaluations, without requiring the finiteness of the chase instance. In particular, the certain answers of OMQs over a UCQ-rewritable ontology can be computed using no more than logarithmic space, w.r.t. the size of the input database, i.e., in the *data complexity*. Figure 2.1 highlights the difference in strategy between answering an OMQ (q, Σ) over a database D via the chase procedure, and a query rewriting approach.

The technique of query rewriting has its roots in answering OMQs over Description Logics-based ontologies [60, 34, 3, 66, 62]. Description Logics are an ontology formalism based on first-order logic that allow to reason about abstract concepts such as People, Animals, etc, and their relationships (roles). The work of [34] is one of the first to propose a query rewriting algorithm for one of the simplest and most widespread Description Logics: DL-Lite. DL-Lite is a lightweight Description Logic that forms the logical underpinning of the OWL2 QL profile of the W3C. The output of the algorithm is a UCQ that can be evaluated alone on the original database, and retrieves the certain answers of the original OMQ. In [60], the authors provide an overview of different query rewriting algorithms for different Description Logic-based OMQs, and also introduce their own query rewriting algorithm, dubbed REQUIEM. The difference with other rewriting approaches is that, although the considered Description Logics are UCQ-rewritable, REQUIEM uses Datalog as the target query language in order to build small queries (more details on Datalog queries will be given in Chapter 3). Another promising approach for query rewriting under Description Logics was introduced in [3]. The authors proposed a versatile approach that can deal with both Description Logic-based languages and TGD-based languages.

Inspired by the above results on Description Logics, similar results have been proved for OMQs over TGD-based ontologies. Current approaches to tackle this issue focused on identifying fragments of TGD-based ontologies that are UCQ-rewritable. Recall that an ontology Σ is *UCQ-rewritable* if for any conjunctive query q, it is possible to rewrite the OMQ (q, Σ) to a UCQ q' such that, for *every* database D, the answers of (q, Σ) over D coincide with the answers of q' over D alone. Different fragments of TGD-based ontologies have been introduced in the literature that guarantee the above property. Among the main UCQ-rewritable languages from the literature, we recall the *linear* [31] and *sticky* [32] ontology languages. Implementations of the rewriting techniques for the above languages have been developed in the literature and have proved to behave well in practice, e.g., see [41, 70] for linear and its variations, while see [41] for sticky. Finally, generic algorithms have also been developed that can construct a UCQ-rewriting of

a given OMQ $\mathcal{O} = (q, \Sigma)$, whenever Σ is UCQ-rewritable [52], i.e., the algorithms do not rely on any specific syntax of Σ, but only require Σ to be UCQ-rewritable. Implementations of these generic approaches also exist, which have also been shown to behave well in practice [10].

Guarded-based Languages. The high efficiency of UCQ-rewritable languages comes at the cost of limited expressive power. In particular, all UCQ-rewritable languages lack the ability to express basic data properties that are crucial for effective ontological modeling. Hence, more expressive, computable TGD-based languages emerged that try to overcome these limitations. Examples of these languages are the *guarded* language [31], and its extension *frontier-guarded* [11]. These languages are proper extensions of linear, that still guarantee data-efficient computation of certain answers, i.e., certain answers can be computed in polynomial time w.r.t. the size of the database. One way of obtaining the above complexity results is to show, as done in [44, 4], that (frontier-)guarded ontologies are Datalog-rewritable, i.e., every OMQ $\mathcal{O} = (q, \Sigma)$, with Σ (frontier-)guarded, 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 can be retrieved by simply evaluating $\mathcal{D}_{\mathcal{O}}$ over D, without involving the ontology Σ. It is well-known that Datalog queries can be evaluated in polynomial time w.r.t. the size of the database, e.g. see [37], and different Datalog-based engines exist in the literature that have proven to be quite efficient at evaluating Datalog queries in practice, e.g. see [67, 2, 51]. Some implementations exist of the Datalog rewriting algorithm for OMQs $\mathcal{O} = (q, \Sigma)$, where Σ is guarded, assuming that q is an atomic query [19].

Languages Beyond Guardedness and UCQ-rewritability. Although guarded-based languages provide higher expressive power, while preserving tractability of certain answers, they still lack critical features for expressing important properties of data. Among such features, the main ones are transitive closure-based properties over graphs, such as finding all pairs of nodes in a graph that are connected by a path. Expressive TGD-based ontology languages with this capability have been defined in the literature, which also preserve computability of the certain answers, such as the language of weakly-guarded TGD-based ontologies [30]. However, such languages increase expressiveness to the point that computing certain answers is not tractable anymore. In the case of weakly-guarded ontologies, computing certain answers is EXPTIME-hard, in data complexity. Such a high data complexity implies that no Datalog rewriting algorithms exist for weakly-guarded ontologies, and thus specialized query answering procedures are required. However, we are not aware of any practical implementations of

these procedures. More efficient TGD-based languages that support the above features include shy [55, 13, 40] and warded [43, 20]. While these two languages are uncomparable from the expressiveness point of view [12], they both guarantee that the certain answers can be computed in polynomial time w.r.t. the size of the input database. However, only for shy there exist procedures that go beyond theoretical complexity results, and that have been implemented in practice.

Chapter 3

Preliminaries

In what follows, we recall notions of relational databases, ontologies and the OMQ Answering problem.

3.1 Relational Databases

A schema $\mathbf{S}$ is considered the blueprint that defines concrete databases. It consists of a finite set of predicates R/n (relation names), each with a specific arity 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 $\{(R, i) \mid R/n \in \mathbf{S} \text{ and } i \in [n]\}$. We use $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ to denote countably infinite sets of *constants*, *nulls* and *variables*. A *term* is a constant, null, or variable. An atomic formula (or simply atom) over a schema $\mathbf{S}$ is an expression of the form $R(\bar{t})$, where R is a predicate of $\mathbf{S}$ with arity n, and $\bar{t} = t_1, \ldots, t_n$ is a tuple of n terms. An atom with only constants is called a *fact*. In a sequence of atoms $\alpha_1, \ldots, \alpha_n$, a variable is a *join* variable if it appears more than once in the sequence. An *instance* is a (possibly infinite) set of atoms over 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.

Example 3.1.1. Consider the database schema $\mathbf{S}$ containing the relations shown below. For clarity, we write a relation R/n of $\mathbf{S}$ as $R(A_1, \ldots, A_n)$ where each A_i is a name we give to the i-th position of R in order to describe its meaning in R.

$$Employee(EmployeeID, Name, Department, Project, PlaceOfBirth)$$
$$Department(DepartmentNO, DepartmentName)$$
$$Project(ProjectID, ProjectName)$$
$$City(CityID, CityName, Country)$$

The schema consists of four relations: Employee, Department, Project, and City, where Employee has arity 5, Department and Project have arity 2, and City has arity 3. The following depicts a database D over the schema $\mathbf{S}$, which we graphically represent as a set of tables, one for each relation of $\mathbf{S}$, where each row (tuple) $\bar{t}$ in the table of a relation R corresponds to the fact $R(\bar{t})$ of D.

Employee	EmployeeID	Name	Department	Project	PlaceOfBirth
	101	Alice	1	V22	VR
	102	Bob	2	V22	RM
	103	Andrea	1	AI	TN
	104	Martha	1	HZ	TN

Department	DepartmentNO	DepartmentName
	1	Computer Science
	2	Engineering
	3	Finance

Project	ProjectID	ProjectName
	V22	Vision22
	HZ	Horizon
	AI	CoolAI

City	CityID	CityName	Country
	MI	Milan	Italy
	TN	Trento	Italy
	RM	Rome	Italy
	BZ	Bolzano	Italy
	VR	Verona	Italy

3.2 (Union of) 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 $\bigcup_{i \in [n]} \bar{x}_i$. 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 $Q \notin \mathbf{S}$. Note that, by abuse of notation, we treat a tuple of variables as a set of variables. 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*, which is defined as the set of tuples

$$\{\bar{t} \in (\mathsf{dom}(I) \cap \mathbf{C})^k \mid \text{there is a homomorphism } h \text{ from } \mathsf{body}(q) \text{ to } I \text{ with } h(\bar{x}) = \bar{t}\}.$$

Example 3.2.1. Consider the schema $\mathbf{S}$ and the database D of Example 3.1.1 above. A conjunctive query q over $\mathbf{S}$ can be used to ask different questions about the database D. For example, to find all the pairs of employees that were born in the same country, we can write the following CQ; we use _ to denote a variable whose name is not important and it is different from all the other variables occurring in the query.

$$Ans(x, x') :- Employee(x, \_, \_, \_, y), Employee(x', \_, \_, \_, y'), \\ City(y, \_, z), City(y', \_, z).$$

A *union of conjunctive queries (UCQ)* is a generalization of a CQ that allows to return the union of the answers of more than one CQ. Formally, for an integer $k \geq 0$, a k-ary UCQ q is a set of k-ary conjunctive queries $q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)$ all sharing the same head predicate. The set of answers of q over an instance I is $q(I) = \bigcup_{i=1}^{n} q_i(I)$.

3.3 Datalog and TGD-based Ontologies

Datalog is a prominent rule-based language that was originally introduced as a more expressive query language than simple UCQs [42]. The higher expressiveness of Datalog is due to the ability to express recursive constructs. In recent years, there has been a renowned interest in using Datalog for other data management tasks, including modeling ontologies.

Consider a schema $\mathbf{S}$. A Datalog program Σ over $\mathbf{S}$ is a set of (Datalog) rules over $\mathbf{S}$. A Datalog rule ρ over $\mathbf{S}$ is an expression of the form:

$$\forall \bar{x}_1, \ldots, \bar{x}_n \, R_1(\bar{x}_1) \wedge \ldots, \wedge R_n(\bar{x}_n) \rightarrow R_0(\bar{x}_0)$$

where each $R_i(\bar{x}_i)$ is an atom over $\mathbf{S}$, $\bar{x}_i$ is a tuple of variables, and each variable of $\bar{x}_0$ occurs in some $\bar{x}_i$.

Example 3.3.1. Consider the Datalog program Σ consisting of the following rules:

$$\begin{aligned} \rho_1 : &\quad \forall x, y \; Edge(x, y) \rightarrow Path(x, y), \\ \rho_2 : &\quad \forall x, y, z \; Edge(x, y) \wedge Path(y, z) \rightarrow Path(x, z). \end{aligned}$$

The above rules encode the definition of path in directed graphs. Such a property can be expressed thanks to recursive statements, and in this particular case, via rule ρ_2, stating that if there exists an edge from a node x to a node y in the graph, and we know there is a path from y to another node z, we can conclude that a path from x to z must also exist.

Although Datalog provides a good basis for an ontological language, its main limitation in this context is the inability to reason about entities, i.e., terms, that are explicitly mentioned in a given database. This leaves a critical challenge in ontology modeling, where ontologies are required to represent data that might not exist in the database. For this reason, extensions of Datalog have been considered in the literature as more expressive ontology languages. The prominent example are *tuple generating dependencies (TGDs)*, that add existential quantification to the head of Datalog rules, increasing its expressiveness.[1]

[1]Historically, TGDs have been employed as relational database integrity constraints, while in our setting, they are employed as logical rules for ontology specification.

3.3.1 Tuple-Generating Dependencies

A *tuple-generating dependency* (TGD) σ over a schema $\mathbf{S}$ is a (constant-free) expression of the form $\forall\bar{x}\forall\bar{y}\,(\phi(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x},\bar{z}))$, 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 $\phi(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x},\bar{z})$, 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. Note that a full, single-head TGD corresponds to a Datalog rule. 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 $\sigma \in \Sigma$.

Example 3.3.2. The set $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ presented in Example 1.1.1 is an ontology containing 4 TGDs, where the TGD σ_1 encodes when two persons are friends, while σ_2 and σ_3 specify that friendship is a symmetric and transitive relationship. Finally, σ_4 states that if two people x and y are friends, then there must be some topic z which both like. Note how σ_4 employes existential quantification to talk about an unknown topic that connects two friends.

3.4 Ontology-Mediated Queries

The main task we are interested in 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.

Example 3.4.1. Consider the database $D = \{Person(a)\}$ and the ontology $\Sigma = \{Person(x) \rightarrow \exists y\ Ancestor(y,x)\}$, stating that every person must have some ancestor. Several (actually infinitely many) models of D and Σ exist, For example,

$M_1 = \{Person(a), Ancestor(b, a)\}$ is a model, since it contains D, and satisfies the only TGD in Σ. Another example is $M_2 = \{Person(a), Ancestor(b, a), Ancestor(c, a)\}$, where the presence of $Ancestor(c, a)$ indicates that there can be more ancestors to a. Another model is $M_3 = \{Person(a), Ancestor(\bot, a)\}$ where $\bot$ is a null from $\mathbf{N}$, which we interpret as an unknown value.

Hence, the models of D and Σ encode all possible states in which the real-world system that Σ is describing can be, that also agree on the ground knowledge stated in D.

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

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

In other words, the certain answers of $\mathcal{O}$ over D collect only the answers that are true in every model, and thus are true, no matter the actual state of the underlying real-world system encoded by Σ.

Example 3.4.2. Consider the database D and the ontology Σ of Example 3.4.1, and consider the OMQ $\mathcal{O} = (q, \Sigma)$, with q of the form:

$$Ans(x) \leftarrow \exists y\ Ancestor(y, x)$$

returning all people with an ancestor. We have that $\mathsf{ans}(\mathcal{O}, D)$ contains the tuple (a), while for the OMQ $\mathcal{O}' = (q', \Sigma)$, with q' of the form:

$$Ans(y) \leftarrow \exists x\ Ancestor(x, y)$$

finding all the ancestors, we have that $\mathsf{ans}(\mathcal{O}', D)$ is empty, since we cannot be certain who are the ancestors of a.

3.5 Ontology-Mediated Query Answering

For a class C of ontologies, we define $\mathsf{CQAns}[\mathsf{C}]$ as the (decision) problem that, given a database D, a k-ary OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in \mathsf{C}$, and a tuple $\bar{t} \in \mathsf{dom}(D)^k$, asks whether $\bar{t} \in \mathsf{ans}(\mathcal{O}, D)$.

Since TGD-based ontologies are characterized by the addition of existential quantifiers in the head of the rules, which allow to express knowledge about unknown individuals, that with standard Datalog would not be possible, this on the other hand causes computational issues. In particular, the problem $\mathsf{CQAns}[\mathsf{TGD}]$, where we recall TGD denotes the class of all ontologies, is known to be undecidable. That is, given a database D, a k-ary OMQ $\mathcal{O} = (q, \Sigma)$, with $\Sigma \in \mathsf{TGD}$, and a tuple $\bar{t} \in \mathsf{dom}(D)^k$, there is no algorithm that is always able to check whether $\bar{t} \in \mathsf{ans}(\mathcal{O}, D)$, for example, see [41]. The latter implies that it is not possible to devise a generic algorithm that can always construct the set of certain answers of a given OMQ over a given database, in a finite amount of time.

The main research endeavour has then been to identify fragments of TGD-based ontology languages that guarantee the decidability of the OMQ Answering problem. This aspect will be discussed in the next two chapters.

Chapter 4

The Chase

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 defined as:

$$\mu(x) \;=\; \begin{cases} h(x) & \text{if } x \in \mathsf{fr}(\sigma) \\ \\ \perp^{x}_{\sigma,h} & \text{otherwise} \end{cases}$$

where $\perp^{x}_{\sigma,h}$ is a null value from $\mathbf{N}$. The trigger (σ, h) is *active* if $\mathsf{result}(\sigma, h) \not\subseteq I$.

Observe that in the definition of $\mathsf{result}(\sigma, h)$ above each existentially quantified variable x of $\mathsf{head}(\sigma)$ is mapped by μ to a null value of $\mathbf{N}$ whose name is uniquely determined by the trigger (σ, h) and x itself. This means that, given a trigger (σ, h), we can unambiguously write down the set of atoms $\mathsf{result}(\sigma, h)$. An *application* of (σ, h) on I is the expression $I\langle\sigma, h\rangle J$, where $J = I \cup \mathsf{result}(\sigma, h)$.

At the high level, the chase procedure performs exhaustive applications of triggers on a given database D, in order to construct a model of D and Σ. Since not all triggers are required to be applied in order to build such a model, different variations of this basic procedure exist, which only differ on the criterion used to decide when a trigger should be applied, i.e., when a trigger is considered *active*.

There are three main variations of this basic procedure: the standard (a.k.a. restricted) chase, the semi-oblivious (a.k.a. skolem) chase, and an oblivious chase. All such variations are defined in the same way, with the difference of what we mean by

active trigger, that we will clarify later.

Definition 1. *Consider a database D, and an ontology Σ of TGDs.*

- *A finite sequence $(I_i)_{0 \leq i \leq n}$ of instances, with $D = I_0$ and $n \geq 0$, is a* chase derivation *of D w.r.t. Σ if, for each $i \in \{0, \ldots, n-1\}$, there is an "active" trigger (σ, h) for Σ on I_i with $I_i \langle \sigma, h \rangle I_{i+1}$, and there is no active trigger for Σ on I_n. The* result *of such a chase derivation is the instance I_n.*

- *An infinite sequence $(I_i)_{i \geq 0}$ of instances, with $D = I_0$, is a* chase derivation *of D w.r.t. Σ if, for each $i \geq 0$, there is an "active" trigger (σ, h) for Σ on I_i such that $I_i \langle \sigma, h \rangle I_{i+1}$. Moreover, $(I_i)_{i \geq 0}$ is* fair *if, for each $i \geq 0$, and for every active trigger (σ, h) for Σ on I_i, there exists $j > i$ such that (σ, h) is not an active trigger for Σ on I_j. The* result *of such a chase derivation is the instance $\bigcup_{i \geq 0} I_i$.*

A chase derivation is valid *if it is finite, or infinite and fair.*

Let us stress that infinite but unfair chase derivations are not considered valid ones since they do not serve the main purpose of the chase, that is, to build a model of the database D and ontology Σ. Indeed, given the ontology Σ consisting of the TGDs

$$\sigma \;=\; R(x, y) \rightarrow \exists z\, R(y, z) \qquad \sigma' \;=\; R(x, y) \rightarrow P(x, y),$$

the result of the unfair chase derivation of $D = \{R(a, b)\}$ w.r.t. Σ that involves only triggers of the form $(\sigma, \cdot)$, i.e., only the TGD σ is used, does not satisfy σ', and thus does not satisfy Σ.

We now move to discuss the three different variations of chase mentioned before: the standard chase, semi-oblivious, and oblivious chase. Each variation has advantages and disadvantages depending on the application where they are involved. In what follows, fix an instance I, an ontology Σ, and a trigger (σ, h) for Σ on I.

4.1 Standard (Restricted) Chase

The emergence of the first chase procedure known as the restricted or standard chase goes back to the late 70s and early 80s [57, 14], when the database community was studying the implication problem for data dependencies. The restricted chase requires that triggers (σ, h) for Σ on I are applied only if the TGD σ is not satisfied by I. In other words, in the case of the standard chase, (σ, h) is *active* if $I \not\models \sigma$. Hence, the

standard chase is a very conservative version of the chase, where a TGD is applied only when necessary.

Example 4.1.1. Consider applying the restricted chase on a database $D = \{R(a)\}$ and the ontology Σ containing the following TGDs:

$$\sigma_1 : \quad R(x) \rightarrow \exists y \; S(x, y)$$
$$\sigma_2 : \quad S(x, y) \rightarrow T(y).$$

Then, we have that (σ_1, h_1), with $h_1 = \{x \mapsto a\}$ is a trigger for Σ on $I_0 = D$, and it is active, since $R(a) \in D$, but no atom of the form $S(a, \cdot)$ occurs in D. Hence, the instance $I_1 = D \cup \mathsf{result}(\sigma_1, h_1) = D \cup \{S(a, \perp^y_{\sigma_1,h_1}$ is constructed by the standard chase, where $I_0 \langle \sigma_1, h_1 \rangle I_1$. Then, (σ_2, h_2), with $h_2 = \{x \mapsto a, y \mapsto \perp^y_{\sigma_1,h_1}\}$ is a trigger for Σ on I_1, which is also active since no atom with relation T occurs in I_1, and then the instance $I_2 = I_1 \cup \mathsf{result}(\sigma_2, h_2) = I_1 \cup \{T(\perp^y_{\sigma_1,h_1}\}$ is constructed. No other active triggers exist, and I_2 is the result of the (valid) standard chase derivation I_0, I_1, I_2.

4.2 Oblivious Chase

The oblivious chase [39] is the simplest among the three variations we consider in this thesis, as it deems a trigger (σ, h) for Σ on I as active simply if its result $\mathsf{result}(\sigma, h)$ does not appear in I, thus disregarding whether the TGD σ is already satisfied by I or not. This makes the oblivious chase much simpler to implement in real-world systems, but at the same time, it incurs higher redundancy in the number of generated facts.

Example 4.2.1. Consider the database $D = \{R(a), S(a, b)\}$, and the ontology Σ of Example 4.1.1. The only trigger for Σ on $I_0 = D$ is (σ_1, h_1) with $h_1 = \{x \mapsto a\}$. However, although $I_0 \models \sigma_1$, according to the oblivious chase, the trigger is active, since $\mathsf{result}(\sigma_1, h_1) \not\subseteq I_0$. Hence, the oblivious chase applies the trigger, producing $I_1 = I_0 \cup \{S(a, \perp^y_{\sigma_1,h_1})\}$. Then, there are two triggers (σ_2, h_2) and (σ_2, h'_2) for Σ on I_1, where $h_2 = \{x \mapsto a, y \mapsto b\}$ and $h'_2 = \{x \mapsto a, y \mapsto \perp^y_{\sigma_1,h_1}\}$. Both are active, and assuming the oblivious chase applies (σ_2, h_2) first, obtaining I_2, we can see that (σ_2, h'_2) is still active, and it is also applied on I_2, producing I_3 of the form:

$$\{R(a), S(a, b), S(a, \perp^y_{\sigma_1,h_1}), T(b), T(\perp^y_{\sigma_1,h_1})\}.$$

Hence, I_0, I_1, I_2, I_3 is a valid oblivious chase derivation.

4.3 Semi-Oblivious Chase

The semi-oblivious chase is a refined version of the oblivious chase [46], where the results of two triggers (σ, h) and (σ, h') over the same TGD are indistinguishable as far as h and h' map the frontier variables of σ to the same terms. That is, their results $\mathsf{result}(\sigma, h)$ and $\mathsf{result}(\sigma, h')$ are considered to be the same, modulo renaming the fresh nulls used for the existential variables of σ. This is a more selective version of the oblivious chase and less restrictive than the standard chase, and it is the version of the chase that is mostly preferred for implementations.

Example 4.3.1. Consider the database $D = \{R(a, b), R(a, c)\}$ and the ontology $\Sigma = \{\sigma\}$ with σ being the TGD $R(x, y) \rightarrow \exists z\ S(x, z)$. There are two triggers (σ, h) and (σ, h') for Σ on $I_0 = D$, with $h = \{x \mapsto a, y \mapsto b\}$ and $h' = \{x \mapsto a, y \mapsto c\}$. Both triggers are active according to the semi-oblivious chase. However, if (σ, h) is applied, producing $I_1 = I_0 \cup \mathsf{result}(\sigma, h) = I_0 \cup \{S(a, \perp^z_{\sigma,h})\}$, the trigger (σ, h') for Σ on I_1 is not active anymore, since $\mathsf{result}(\sigma, h')$ and $\mathsf{result}(\sigma, h)$ produce the same set of atoms, modulo the name of the null assigned to the existential variable z. Hence, I_0, I_1 is a (valid) semi-oblivious chase derivation.

4.4 Universality of the Chase and the Variation Used in the Thesis

Regardless of the version of the chase considered, every valid chase derivation of a database D and an ontology Σ is guaranteed to produce as a result, an instance I that is a model of D and Σ. Moreover, I is guaranteed to be a *universal model* of D and Σ. That is, for every model $M \in \mathsf{models}(D, \Sigma)$, there exists a homomorphism from I to M. The above universality property implies the following well-known result (e.g., see [46], [46]):

Theorem 1. *For a k-ary OMQ $\mathcal{O} = (q, \Sigma)$ and a database D over a schema $\mathbf{S}$, it holds that*

$$\mathsf{ans}(\mathcal{O}, D) = \{\bar{t} \in \mathsf{dom}(D)^k \mid \bar{t} \in q(I) \text{ with } I \text{ the result of some (standard/oblivious/semi-oblivious) chase derivation of } D \text{ and } \Sigma\}.$$

Given that any chase variation can be used for OMQ Answering purposes, as stated by Theorem 1, for simplicity, in our technical analysis starting from Chapter 6, we will

employ the oblivious chase. In particular, it is well-known, and it is not difficult to verify, that for the oblivious chase, for every database D and ontology Σ, every valid oblivious chase derivation of D and Σ produces exactly the same instance, which can be equivalently defined as follows. For a database D and an ontology Σ over a schema $\mathbf{S}$, the *(oblivious) 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 $i > 0$, we define $\mathsf{chase}^i(D, \Sigma)$ as the set

$$\mathsf{chase}^{i-1}(D, \Sigma) \cup \{\mathsf{result}(\sigma, h) \mid (\sigma, h) \text{ is a trigger for } \Sigma \text{ on } \mathsf{chase}^{i-1}(D, \Sigma)\}.$$

Finally, $\mathsf{chase}(D, \Sigma) = \bigcup_{i \geq 0} \mathsf{chase}^i(D, \Sigma)$.

Chapter 5

TGD-based Ontology Languages

In this chapter, we provide more details on the main TGD-based ontology languages introduced in the literature to restore decidability of OMQ Answering. Tuple-generating dependencies (TGDs) have originally been employed in database theory as a formalism for enforcing constraints over the schema of a relational database, such as inclusion dependencies [1]. In recent years, TGDs have seen a resurgence of interest as a rich formalism for modeling domain knowledge in the form of ontologies. The rich expressive power of TGDs as an ontology language in turn allows to infer complex knowledge from a system by answering Ontology-Mediated Queries (OMQs).

As discussed in Chapters 1 and 2, the major challenge in leveraging TGDs for OMQ Answering is that the high expressive power of TGDs makes OMQ Answering undecidable in general. Hence, different attempts exist in the literature that try to identify TGD-based languages that restore computability.

We categorize such attempts in three large families: acyclicity-based, rewriting-based, and ad hoc algorithmic-based. We point out that although each of the languages we are going to discuss below was originally defined with the intent to employ only one of the above three main techniques, it later turned out that for some of them, OMQ Answering can be solved via different approaches, falling in more than one of the above categories. Hence, for historical reasons, we will include each language in the category it was originally designed for, but we will also give hints on how other techniques from the other categories can be applied to these languages.

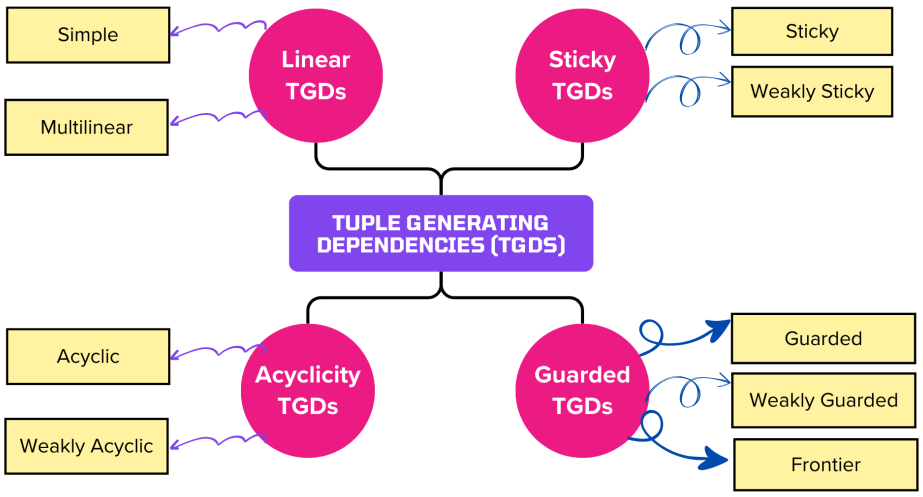

Figure 5.1: TGD-based Ontology Languages.

5.1 Acyclicity-based Ontology Languages

We start by focusing on the first family of languages that restores computability, i.e., the acyclicity-based ones, which, as discussed in Chapter 2, guarantees the termination of (some variation of) the chase procedure. In particular, when we say that a chase variation terminates with input a database D and ontology Σ, we mean that *every* valid derivation of D and Σ is finite. The latter allows the construction of a finite universal model that can be employed for OMQ Answering purposes, thanks to Theorem 1.

There are mainly two aspects that in combination can cause some variation of the chase to not terminate:

1. The ontology allows the construction of new values (nulls) during the evaluation of the chase procedure.

2. The set of TGDs creates a cyclic dependency between different predicates.

Example 5.1.1. Consider the database D and the ontology Σ of Example 2.0.1. It is not difficult to see that any variation of the chase discussed in Chapter 4 does not terminate with input D and Σ, because the first TGD σ_1 produces a fresh null due to the existential variable y, while the second TGD σ_2 propagates this null back to the body of σ_1.

The simplest way to prevent the chase to produce an infinite instance, no matter the input database, is to consider so-called *acyclic ontologies*. Acyclic ontologies, also

known as non-recursive ontologies, ensure that no cycles can ever be formed between the predicates of the ontology. Formally, the *predicate graph* of an ontology Σ is the directed graph (V, E), where the set of nodes $V = \mathsf{sch}(\Sigma)$ is the set of predicates in Σ, and there is an edge $(R, P) \in E$ iff there is a TGD $\sigma \in \Sigma$ with R occurring in $\mathsf{body}(\sigma)$ and P occurring in $\mathsf{head}(\sigma)$. We say that Σ is *acyclic* iff the predicate graph of Σ is acyclic.

It is easy to verify that for an acyclic ontology Σ, for every database D, every variation of the chase discussed in Chapter 4 terminates producing a finite universal model.

Example 5.1.2. Consider the ontology $\Sigma = \{\sigma\}$ where:

$$\sigma : \ \ Employee(x) \rightarrow \exists z \ Manager(x, z),$$

Stating that every employee must have a manager. The predicate graph of Σ consists of the single edge $Employee \rightarrow Manager$, which is acyclic. Hence, even if σ allows the construction of new nulls, there is no way for these nulls to trigger in turn the construction of infinitely many nulls.

Although acyclic ontologies guarantee the decidability of OMQ Answering, they form a very limited ontology language, as they completely disregard the use of recursion, which is an important feature in knowledge modeling. Hence, a more sensible strategy is to allow the combination of recursive rules and value invention but in a controlled way. This is what the class of *weakly-acyclic ontologies* achieves [39]. Weakly-acyclic ontologies is a more relaxed version of acyclic ontologies which allow the creation of recursive loops where the number of nulls that could be possibly created remains bounded, no matter the database. The definition of weakly-acyclic ontologies is heavily based on the concept of dependency graph [39]. Intuitively, the dependency graph of an ontology Σ, denoted $\mathsf{dg}(\Sigma)$, is a graph that describes how nulls are propagated via the positions of the predicates of Σ, and we call Σ weakly-acyclic if no cycle in $\mathsf{dg}(\Sigma)$ exists that witnesses the propagation of a null. We now formalize the above notion.

Definition 2 (Dependency Graph)**.** *Consider an ontology Σ. The* dependency graph $\mathsf{dg}(\Sigma)$ *of Σ, is a directed multigraph (V, E), where $V = \mathsf{pos}(\mathsf{sch}(\Sigma))$, i.e., V is the set of all positions of predicates in Σ, and E contains* only *the following edges. For each TGD $\sigma \in \Sigma$ with $\mathsf{head}(\sigma) = \{\alpha_1, \ldots, \alpha_k\}$, for each $x \in \mathsf{fr}(\sigma)$, and for each position π of $\mathsf{body}(\sigma)$ in which x occurs:*

- *For each $i \in [k]$, and for each position π' in α_i in which x occurs, there exists a* normal *edge* $(\pi, \pi') \in E$.

- *For each existentially quantified variable z in σ, $i \in [k]$, and position π' in α_i in which z occurs, there is a* special *edge* $(\pi, \pi') \in E$.

Roughly, if an edge is created between a universal variable in the body and an existential variable in the head of the same TGD, this edge, which we call special, witnesses the creation of a new null because of the presence of a term mapped to the universal variable. On the other hand, a normal edge simply denotes that a term is propagated from a position in the body to a position in the head. We say that an ontology Σ is *weakly-acyclic* if there is no cycle in $\mathsf{dg}(\Sigma)$ mentioning a special edge [39].

Example 5.1.3. Consider the schema

$$\mathbf{S} = \{Employee(Name, Manager), Department(DeptName, Manager)\}$$

mentioning employees with their manager, and departments with their manager, and consider the ontology $\Sigma = \{\sigma_1, \sigma_2\}$ with:

$$\begin{aligned} \sigma_1 : \quad & Employee(x, y) \rightarrow \exists z \; Department(z, y), \\ \sigma_1 : \quad & Department(x, y) \rightarrow \exists z \; Employee(z, y). \end{aligned}$$

The TGD σ_1 states that if an employee x exists with a manager y, then y must be the manager of some department, while σ_2 states that if a department x exists with a manager y, an employee must exist having y as the manager. The dependency graph of Σ contains the edges $Employee[2] \rightarrow Department[2]$, $Employee[2] \dashrightarrow Department[1]$, $Department[2] \rightarrow Employee[2]$, and $Department[2] \dashrightarrow Employee[1]$, where a dashed arrow denotes a special edge, and a non-dashed arrow denotes a normal edge. Although a cycle exists in the dependency graph of Σ, no such a cycle goes via a special edge, and thus Σ is weakly-acyclic.

It is known that for a weakly-acyclic ontology Σ, both the standard and the semi-oblivious chase terminate, for every input database [39, 36]. A variation of the dependency graph also exists which leads to so-called *richly-acyclic* ontologies, that guarantee that the oblivious chase terminates, for every input database [50].

Based on the above ideas, a portfolio of more refined acyclicity-based notions have been defined in the literature that guarantee the termination of one or more variants of the chase procedure. Among these notions, we recall stratification [38], safety [59],

joint acyclicity [53], super-weak acyclicity [58], model-faithful acyclicity and model-summarizing acyclicity [47].

Interestingly, all the above notions not only guarantee that (some variation of) the chase always terminates, for every input database, but such a chase procedure can construct a universal model in polynomial time w.r.t. the size of the input database, i.e., in data complexity. Hence, they form well-behaved TGD-based ontology languages that can potentially be used in practice for OMQ Answering.

5.2 Rewriting-based Ontology Languages

One of the key limitations of acyclicity-based ontologies is that they cannot reason about infinite structures, since by design, they only allow the introduction of a bounded number of new terms. In this section, we focus on a family of ontology languages that guarantee decidability of OMQ Answering by means of a different property; these languages usually do not force the chase to terminate, but instead limit the language in other ways. The general idea in defining a so-called rewriting-based ontology language is to guarantee that an ontology Σ falling in the language enjoys a property of the following form, for a query language $\mathcal{L}$.

For every CQ q, the OMQ $\mathcal{O} = (q, \Sigma)$ can be rewritten to a query q' over $\mathcal{L}$ such that, for every database D, $\mathsf{ans}(\mathcal{O}, D) = q'(D)$, where $q'(D)$ denotes the answers to q', when evaluated over D. The idea is that, generally, $\mathcal{L}$ is a query language for which $q'(D)$ is computable, for every database, and thus one reduces the problem of answering $\mathcal{O}$ over D to the problem of answering q' over D. We note that with the above approach, nothing is said on whether any variation of the chase is actually guaranteed to terminate, for every input database.

The most common instantiations of $\mathcal{L}$ include UCQs, which form a very efficient query language that can be evaluated over standard relational databases, and Datalog, which can be evaluated efficiently with state-of-the-art engines. In the rest of this section we recall some of the main rewriting-based ontology languages.

5.2.1 Linearity

The simplest rewriting-based ontology language is the class of *linear ontologies* [30, 31]. A TGD σ is *linear* if $\mathsf{body}(\sigma)$ contains only one atom. Then, an ontology Σ is *linear* if each TGD $\sigma \in \Sigma$ is linear. The main property of linear ontologies is that they are all UCQ-rewritable. That is, for every OMQ $\mathcal{O} = (q, \Sigma)$, where Σ is linear, then $\mathcal{O}$ can be

rewritten to a UCQ q' such that $\mathsf{ans}(\mathcal{O}, D) = q'(D)$, for every database D.

Linear ontologies are powerful enough to capture other ontology languages from the Description Logics literature, and in particular the Description Logic DL-Lite, which is the logical underpinning of one of the most popular profiles of the OWL2 specification by the W3C, i.e., OWL2 QL.

Simple-Linear Ontologies. Actually, linear ontologies generalize DL-Lite even when restricted to have each body variable of a TGD to appear only once in the body. We call linear TGDs of this form *simple-linear*, and ontologies containing only simple-linear TGDs are called *simple-linear ontologies*.

Example 5.2.1. The ontology $\Sigma = \{\sigma\}$, where:

$$\sigma : \ \mathit{Employee}(x, y, z) \rightarrow \exists k \ \mathit{Department}(z, k)$$

is simple-linear since its only TGD σ contains only one atom in its body, and each variable in $\mathsf{body}(\sigma)$ occurs only once in $\mathsf{body}(\sigma)$.

Multi-Linear Ontologies. Linear Ontologies have been further extended to the class of multi-linear ontologies, where the body of each TGD can now contain more than one atom, as far as all the body atoms share the exact same set of variables. The class of multi-linear ontologies has been defined to capture extensions of DL-Lite (i.e., DL-Lite with conjunctions) [31]. Also multi-linear ontologies are UCQ-rewritable.

Example 5.2.2. Consider the ontology $\Sigma = \{\sigma\}$, where:

$$\sigma : \ \mathit{WorksIn}(x, y), \mathit{Manages}(x, y) \rightarrow \exists z \mathit{Supervises}(x, y, z),$$

stating that if an employee x works in a department y that she also manages, then x must supervise some employee z in that department. The ontology Σ is not linear, but it is multi-linear.

Non-termination of the Chase. We remark again that (multi-)linear, or even simple-linear ontologies do not necessarily guarantee the termination of (any) variation of the chase.

Example 5.2.3. Consider the database $D = \{R(a, a)\}$, and the ontology $\Sigma = \{\sigma\}$, where:

$$\sigma : \ R(x, y) \rightarrow \exists z \ R(y, z).$$

The ontology Σ is linear (actually, simple-linear), however $\mathsf{chase}(D, \Sigma)$ is infinite.[1] In particular, $\mathsf{chase}(D, \Sigma)$ is the infinite instance:

$$\{R(a, a), R(a, \perp_1), R(\perp_1, \perp_2), R(\perp_2, \perp_3), \ldots\},$$

where $\perp_1, \perp_2, \ldots$ are (simplified) names for the labeled nulls produced via triggers.

5.2.2 Stickiness

A key limitation of (multi-)linear ontologies is that they highly restrict the ability to combine knowledge from multiple atoms in order to derive new facts. In other words, they mostly disallow the use of "joins". A novel class of ontologies, dubbed *sticky* have been defined in [32] that provides more freedom on how variables are joined in the body of TGDs. We point out that, however, also in this case, this "freedom" is controlled in a way to preserve decidability of OMQ Answering, and thus sticky ontologies are uncomparable to (multi-)linear ones, but they generalize simple-linear ontologies. Nonetheless, sticky ontologies are UCQ-rewritable, just like (multi-)linear ontologies.

The key syntactic constraint that sticky ontologies enforce is thus a special structural constraint on the "join" variables of the TGDs, i.e., variables that have multiple occurrences in the TGD bodies.

This constraint is defined on an ontology Σ by means of a recursive marking procedure defined as follows [32]:

- Base case: For each TGD $\sigma \in \Sigma$ and each variable x in the body of σ, if x does not occur in all the head atoms of σ, then we mark each occurrence of x in the body of σ.
- Inductive case: For each TGD $\sigma \in \Sigma$, if a variable x in the body of σ occurs at some position (R, i) in the body of σ and this occurrence is marked, then for every TGD $\sigma' \in \Sigma$, we mark each occurrence of the variables in the body of σ' that also appear in the head of σ' at position (R, i).

Then, a sticky ontology, roughly, prevents joins among variables whose occurrences are marked. Formally, an ontology Σ is *sticky* if, after applying the marking procedure above to Σ, a variable x in the body of some TGD $\sigma \in \Sigma$ that occurs as marked in σ occurs only once in the body of σ.

[1] Actually, any variation of the chase we considered in Chapter 4 does not produce a finite valid derivation.

Example 5.2.4. Consider the ontology $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$, where:

$$\begin{aligned}
\sigma_1 :&\ \ AssignsTask(x, y, z), AssignsTask(x', y', z) \rightarrow WorkTogether(y, y', z),\\
\sigma_2 :&\ \ Manages(x, y) \rightarrow \exists z\ AssignsTask(x, y, z),\\
\sigma_3 :&\ \ WorksOn(x, y, z) \rightarrow \exists w\ Manages(x, w), Monitors(z, w).
\end{aligned}$$

The TGD σ_1 states that if x assigns to y a task z, and somebody else x' assigns to somebody else y' the same task z, then y and y' work together on task z. Then, σ_2 states that if x manages employee y, then x must assign a task z to y. Finally, σ_3 states that if x works on task y in a project z, then there must be somebody w that manages the employee x, and monitors the project z.

After applying the base case of the marking procedure on Σ, we get:

$$\begin{aligned}
\sigma_1 :&\ \ AssignsTask(\mathbf{x}, y, z), AssignsTask(\mathbf{x'}, y', z) \rightarrow WorkTogether(y, y', z),\\
\sigma_2 :&\ \ Manages(x, y) \rightarrow \exists z\ AssignsTask(x, y, z),\\
\sigma_3 :&\ \ WorksOn(\mathbf{x}, \mathbf{y}, \mathbf{z}) \rightarrow \exists w\ Manages(x, w), Monitors(y, w),
\end{aligned}$$

where variables occurrences in boldface are marked. In particular, the (only) occurrences of x and x' in the body of σ_1 are marked, since they do not appear in the head of σ_1, while no variables occurrences are marked in the body of σ_2, and all variable occurrences in the body of σ_3 are marked.

Then, due to the inductive case, we find that the occurrence of x in the body of σ_2 is marked, because x appears in the head in position $(AssignsTask, 1)$, which contains a marked variable in the body of σ_1. No other variables are marked during this step, and no further variables can be marked, obtaining:

$$\begin{aligned}
\sigma_1 :&\ \ AssignsTask(\mathbf{x}, y, z), AssignsTask(\mathbf{x'}, y', z) \rightarrow WorkTogether(y, y', z),\\
\sigma_2 :&\ \ Manages(\mathbf{x}, y) \rightarrow \exists z\ AssignsTask(x, y, z),\\
\sigma_3 :&\ \ WorksOn(\mathbf{x}, \mathbf{y}, \mathbf{z}) \rightarrow \exists w\ Manages(x, w), Monitors(y, w),
\end{aligned}$$

Since each marked variable appears only once in the body of the TGD where it appears, the ontology Σ is sticky. Note that the ontology is not even multi-linear.

.

5.3 Ad hoc Algorithmic-based Ontologies

In this section, we discuss the last family of ontology languages defined in the literature that guarantee the decidability of OMQ Answering. Such languages have been introduced with the goal of increasing the expressive power of rewriting-based languages (in particular, UCQ-rewritable ones), and still overcome the limitations of acyclicity-based ones, by allowing to reason about infinite structures.

For most of these languages, the decidability of OMQ Answering is obtained by means of dedicated algorithms that directly exploit the key features of the language restrictions. We point out, however, that for some of such languages, such as guarded ontologies, it later turned out that a rewriting-based approach can also be employed, by using Datalog as the target language $\mathcal{L}$ for the rewriting.

5.3.1 Guardedness

Guarded ontologies [31] form one of the most studied fragments of TGD-based ontologies in the literature, and it is inspired by the guarded fragment of first-order logic. A TGD σ is *guarded* if there is an atom in the body of σ, called *the guard* of σ, that mentions all body variables.[2] An ontology Σ is *guarded* if every TGD in Σ is guarded.

Example 5.3.1. Consider the ontology $\Sigma = \{\sigma_1, \sigma_2\}$, where:

$$\begin{aligned} \sigma_1 :&\quad Manager(x) \rightarrow Employee(x), \\ \sigma_2 :&\quad Employee(x), Supervises(x, y) \rightarrow Manager(x). \end{aligned}$$

The TGD σ_1 states that every manager is an employee, and σ_2 states that every employee that supervises some project is a manager. The ontology Σ is guarded because the body of σ_1 contains a single atom, which is trivially the guard of σ_1, and the body of σ_2 contains the atom $Supervises(x, y)$ mentioning all the body variables x and y of σ_2, and thus $Supervises(x, y)$ is the guard of σ_2.

Clearly, by definition of guardedness, (multi-linear) ontologies are a special case of guarded ontologies, and thus guarded ontologies strictly generalize multi-linear ontologies. Guarded ontologies are also able to capture the Description Logic EL, the logical underpinning of the OWL2 EL profile of the W3C. However, guarded ontologies are incomparable to other languages such as sticky and acyclicity-based ones.

[2] If more than one such an atom exist, by convention, we pick left-most such an atom as the guard.

It was shown in [31] that OMQ Answering under guarded ontologies is decidable, and in particular, solvable in polynomial time, in data complexity. The above result heavily relies on the notion of guarded chase forest. Roughly, for a database D and a guarded ontology Σ, the *guarded chase forest* is a forest (i.e., a collection of trees), where nodes are the facts in $\mathsf{chase}(D, \Sigma)$, and an edge $\alpha \rightarrow \beta$ in the guarded chase forest exists if there is a trigger (σ, h) that was used during the construction of $\mathsf{chase}(D, \Sigma)$ such that h maps the guard of σ to α, and $\beta \in \mathsf{result}(\sigma, h,)$.

Then, it is shown that given a database D, and an OMQ $\mathcal{O} = (q, \Sigma)$, with Σ guarded, the answers to $\mathcal{O}$ over D can be retrieved by evaluating just q over a finite portion of the guarded chase forest of D and Σ, whose size only depends on q and Σ. Then, they show that this portion can be constructed in polynomial time in the size of D, and the claim follows.

Hence, in order to solve OMQ Answering for guarded ontologies, a specialized procedure that builds the finite portion of the guarded chase forest is needed.

We point out that in subsequent works [45], it has been shown that guarded ontologies are Datalog-rewritable, and thus one can reduce the OMQ Answering problem over guarded ontologies to the problem of evaluating a Datalog query.

Weakly-Guarded Ontologies. Although guarded ontologies extended the expressiveness of other languages such as multi-linear TGDs, they are not expressive enough to capture the full power of Datalog rules. Hence, similar to what has been done with sticky ontologies, a "weak" version of guarded ontologies has been introduced [30], to overcome the limitation. Roughly, a TGD σ is *weakly-guarded* if there is an atom in the body of σ that mentions all the body variables of σ that might unify with a null during the construction of $\mathsf{chase}(D, \Sigma)$. An ontology Σ is *weakly-guarded* if each TGD in Σ is weakly-guarded.

Clearly, weakly-guarded ontologies generalize guarded ones, and are even able to capture the full power of Datalog, since with Datalog rules, no variables can ever be unified with a null, and thus any body atom is the (weak) guard. However, the complexity of OMQ Answering increases dramatically. In particular, OMQ Answering under weakly-guarded ontologies is EXPTIME-complete in data complexity, making the language difficult to employ in practice. The above complexity result is obtained by means of a specialized procedure [30].

Frontier-Guarded Ontologies. A milder extension of guarded ontologies is the one of frontier-guarded [9]. A TGD σ is *frontier-guarded* if there is an atom in the body of σ that mentions all the *frontier* variables of σ, i.e., the body variables that also occur in

the head of σ. An ontology Σ is *frontier-guarded* if every TGD in Σ is frontier-guarded.

Example 5.3.2. Consider the ontology $\Sigma = \{\sigma\}$, where:

$$\sigma : \quad Directs(x, y), Collaborates(y, z) \rightarrow Manager(x, y).$$

The ontology Σ is frontier-guarded because the frontier variables of σ are x and y, and the body atom $Directs(x, y)$ mentions both.

Frontier guarded ontologies strictly generalize guarded ontologies by still preserving good computational properties. In particular, OMQ Answering under frontier-guarded ontologies remains polynomial-time solvable in data complexity [11]. The upper bound can be obtained by a simple extension of the specialized procedure introduced in [31] for guarded ontologies. However, frontier-guarded ontologies are still not general enough to capture the full power of Datalog. For example, the Datalog rule $R(x, y), P(y, z) \rightarrow T(x, z)$ is not frontier-guarded.

5.3.2 Strongly Parsimonious and Shy Ontologies

The last language we consider is the one of *shy* ontologies, defined in [55]. Shy ontologies are a syntactic subset of a more general class of ontologies called strongly parsimonious. Roughly, when an ontology Σ is strongly parsimonious, it means that a special variant of the chase, called parsimonious chase with resumption, can be employed for OMQ Answering purposes, in the same vein of Theorem 1. The parsimonious chase with resumption is a special variant of the chase procedure, where multiple executions of a particular chase procedure are performed (resumed). Although the parsimonious chase with resumption can be generally infinite, it is shown in [55] that for strongly parsimonious ontologies, it only needs to be resumed a finite number of times, and overall, it can be constructed in polynomial time, in the size of the database. Hence, OMQ Answering over strongly parsimonious ontologies can be solved in polynomial time in the size of the input database. We point out that the parsimonious chase with resumption might not be constructible in a finite number of steps in general. However, it always constructs a universal model [55], and thus, it can be employed whenever it is finite, thanks to Theorem 1.

Since deciding whether an ontology is strongly parsimonious is undecidable [55], a simpler syntactic fragment of strongly parsimonious ontologies, called shy, has also been defined in [55], which is easy to recognize. Shy ontologies are, to the best of our knowledge, one of the very few TGD-based ontology languages that provide a good

balance between expressiveness and complexity. In fact, shy ontologies fully capture Datalog and preserve polynomial time data complexity. The only other language that we are aware of with similar properties is warded, which is the main focus of this thesis.

5.3.3 Weakly-sticky Ontologies

An extension of sticky ontologies has been introduced in [32], called *weakly-sticky*. The idea of this extension is to further loosen the restriction on the join variables. In particular, in weakly-sticky ontologies multiple occurrences of a body variable are allowed, even when marked, as far as at least one such an occurrence can, roughly, only unify with a constant during the construction of the chase.

Weakly-sticky ontologies extend the expressive power of sticky ontologies making them capture the full power of Datalog, which standard sticky ontologies, or (multi-)linear ontologies are not able to do, while keeping the complexity of OMQ Answering tractable, i.e., polynomial time w.r.t. the size of the input database. However, the increase of expressive power makes the new language not UCQ-rewritable anymore and dedicated procedures are instead employed to solve OMQ Answering.

5.4 Conclusion

In this section, we have seen different TGD-based ontology languages that guarantee the decidability of OMQ Answering providing different levels of expressiveness and computational complexity. Most of the languages provide efficient means for OMQ Answering, being either acyclicity-based, or rewriting-based. However, their expressive power is generally limited. More expressive languages instead incur in a complexity increase that make them unsuitable for practical applications, with the exclusion of shy ontologies, which also have existing implementations of the underlying algorithmic solutions for OMQ Answering [55]. Table 5.1 provides a summary of the various OMQ answering methodologies, illustrating how each ontology language supports and performs the OMQ answering process.

In the rest of the thesis we will focus on the only other language we are aware of that strikes a very good balance between expressiveness and complexity, like what shy does, i.e., warded. However, we recall that warded ontologies effective algorithmic solutions for OMQ Answering that can be implemented in practice, and rectifying this state of affairs is the main goal of this thesis.

Ontology Language	**OMQ Answering Methodology**
Acyclic	Chase termination
Weakly-acyclic	Chase termination
Linear	UCQ rewriting
Sticky	UCQ rewriting
Weakly-sticky	Datalog rewriting
Guarded	Datalog rewriting / finite portion of the guarded chase forest
Shy	Parsimonious chase with resumption

Table 5.1: OMQ answering methodologies for different ontology languages.

Chapter 6

Warded Ontologies

As discussed in previous chapters, most of the TGD-based ontology languages existing in the literature either provide efficient certain answers computation, but lack expressive power, or they are very expressive, but do not provide concrete means for developing real-world systems, or further, they do not come with publicly available implementations. This is the case, in particular, for the TGD-based ontology language *warded*, which has been recently introduced in [43], 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. The above complexity result has been first shown in [43] by means of a specialized algorithm, while more recent works have shown that warded ontologies are actually Datalog-rewritable [20], thus giving hope for an alternative mean to implement OMQ answering with warded ontologies in practice.[1] As for existing implementations, a simpler version of the specialized query answering procedure defined in [43] for OMQs based on warded ontologies has been implemented in the *commercial* system Vadalog [17]. Vadalog is a system that is able to compute certain answers of OMQs $\mathcal{O} = (q, \Sigma)$, where Σ is a warded ontology, assuming that q is an atomic CQ, and extends the warded language with additional features, such as the use of arithmetic and aggregate operators, to make the language more ergonomic in practice; we refer the reader to [17, 16] for further details.

[1] We point out that the theoretical Datalog rewriting algorithm presented in [20] cannot be implemented as is which will be discussed later in the thesis, and thus further development is required.

6.1 Warded Ontologies and Datalog Rewritability

In the presented thesis, we are interested in the warded fragment of TGD-based ontologies. As we have already discussed, it is well-known that for arbitrary OMQs, OMQ Answering is undecidable, that is, the problem $\mathsf{CQAns}[\mathsf{TGD}]$ is undecidable; the latter implies that it is not possible to devise a generic algorithm that can always construct the set of certain answers of a given OMQ over a given database, in a finite amount of time.

Hence, different fragments of TGD-based ontologies have been considered in the past in order to restore decidability. One such a prominent language is warded.

At the high level, a warded ontology Σ applies restrictions on how some of the variables of its TGDs, called "dangerous" variables, are used. Roughly, a variable of a TGD in Σ is dangerous if it can be unified with a null during the construction of the chase and it is also propagated to the head of the TGD. We recall that from now on, we focus on the oblivious chase, which is the unique instance $\mathsf{chase}(D, \Sigma)$, for every database D and ontology Σ.

Example 6.1.1. Consider the following schema

$$\mathbf{S} = \{Employee(EmpName, Dept), Project(ProjId, Dept), \\ WorksOn(EmpName, ProjId, Task), Task(Name)\},$$

mentioning Employees with their name, the department they work in, as well as projects with the department they are run in, and assigning tasks to employees for a specific project, and collecting tasks in a Task predicate. Consider now the ontology $\Sigma = \{\sigma_1, \sigma_2\}$, with:

$$\sigma_1 : \ Employee(x, y), Project(z, y) \rightarrow \exists w \ WorksOn(x, z, w),$$
$$\sigma_2 : \ WorksOn(x, y, z) \rightarrow Task(z).$$

The TGD σ_1 in Σ states that if an employee x works in a department y in which a project z is running, then x must work at the project z in a certain task w. The TGD σ_2 that if an employee x works in a project y at some task z, then z is a task.

Intuitively, the variable z occurring in σ_2 is dangerous (in σ_2), because it might take a null value produced by σ_1, which is then propagated to the head of σ_2. No other variables are dangerous.

Then, roughly, an ontology Σ is warded if for each TGD σ of Σ, all dangerous

variables of σ occur together in a single body atom of σ, called the "ward" of σ, and all the variables that the ward of σ shares with other body atoms can only unify with a constant during the construction of the chase.

Example 6.1.2. The ontology Σ of Example 6.1.1 is warded, because no dangerous variables appear in the body of σ_1, and the the body of σ_2, which mentions the dangerous variable z contains a single atom, which acts as the ward of σ_2.

We now 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 the 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 (in σ)* 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[2], 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 of Warded Ontologies. It is well-known that the problem $\mathsf{CQAns}[\mathsf{WARDED}]$ is decidable. In particular, it is P-complete in data complexity, i.e., when assuming only the database D and the tuple $\bar{t}$ are part of the input, while the OMQ $\mathcal{O}$ is fixed, and EXPTIME-complete in combined complexity, i.e., when the database, the OMQ and the tuple are all part of the input.

The above decidability results were originally obtained in [43] via dedicated alternating procedures that exploit the key property of wardedness, and that run in logarithmic space in data complexity, and in polynomial space in combined complexity. An alter-

[2]If more than one such an atom exists in the body of the TGD, we consider the left-most one as the ward.

Feature	Warded	Guarded	Linear	Weakly-acyclic
Syntactic restriction	All dangerous variables of a TGD appear in a single body atom called ward, and all variables the ward shares with other body atoms are harmless	All body variables of a TGD appear in a single body atom called guard	Only one atom can exist in the body of a TGD	No cycles with a special edge in the dependency graph
OMQ Answering Methodology	Datalog-rewriting/ad hoc procedure	Datalog-rewriting/finite portion of the chase graph	UCQ-rewriting	Chase termination
Modeling features	Full power of Datalog + controlled reasoning on infinite structures	Full power of EL ontologies + part of Datalog (guarded Datalog rules)	Full power of DL-Lite + part of Datalog (linear Datalog rules)	Full power of Datalog + value invention without reasoning on infinite structures)

Table 6.1: Comparison between warded and the main ontology languages.

native way to prove the above results has been given in [20], where it is shown that warded OMQs are Datalog-rewritable.

Formally, 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 an *intensional predicate (of $\mathcal{D}$)* 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 $\mathcal{D}(D) = \{\bar{t} \in \mathsf{dom}(D)^k \mid R(\bar{t}) \in \mathsf{chase}(D, \Sigma)\}$. We say that 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 $\mathsf{edb}(\mathcal{D}_\mathcal{O}) = \mathbf{S}$, called a *Datalog rewriting of $\mathcal{O}$*, such that for every database D over $\mathbf{S}$, $\mathsf{ans}(\mathcal{O}, D) = \mathcal{D}_\mathcal{O}(D)$; WARDED is Datalog-rewritable [20].

We conclude this section by referring the reader to Table 6.2, where we highlight some of the key differences between warded ontologies and the main ontology languages discussed in Chapter 5. Furthermore, in Table 6.2 we provide examples of ontologies that are expressible only in the warded language and not in any of the other ontology classes.

Ontology	Example	Why Warded	Why not in other Class
Warded but not Shy	σ_1: P(x,y),U(y) → L(x) σ_2: T(x) → ∃y U(y) σ_3: U(x) → ∃y P(y,x)	Warded: The only dangerous variable x in σ_1 appears only inside the ward atom P(x,y) which controls null propagation.	Not Shy: (P,2) and (U,1) in σ_1 are invaded by the same existential variable originated from σ_2.
Warded but not Frontier-guarded	σ: A(x), B(y) → ∃z R(x,y,z)	Warded: No join on dangerous variables; ward trivially satisfied (no reuse of existential).	Not Frontier-guarded: Frontier variables x,y both appear in the head, but no single atom in the body contains them together → frontier not guarded.
Warded but not Weakly-guarded	σ_1: G(m,z) → ∃x T(x) σ_2: P(x) → ∃y R(x,y) σ_3: R(x,y), T(z) → S(x,y)	Warded: Dangerous variable y is contained in the ward R(x,y) in σ_3, so null propagation is controlled and finite.	Not Weakly-guarded: In σ_3, variables y and z are in affected positions however no single body atom that contains all affected variables of the rule.

Table 6.2: Expressiveness of the warded fragment: example ontologies

Chapter 7

A Practical Datalog Rewriting Algorithm for Warded Ontologies

In this chapter, we provide the high-level idea of how the theoretical rewriting algorithm to Datalog for warded ontologies was originally obtained in [20], and then discuss why directly implementing such an algorithm is not feasible in practice. Then, we introduce a new algorithm that is able to produce a Datalog rewriting of a given OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in \mathsf{WARDED}$ that is more amenable to practical implementations.

7.1 An Overview of the Theoretical Rewriting Algorithm and its Limitations

Consider a k-ary OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in \mathsf{WARDED}$. The main idea from [20] to construct a Datalog rewriting of $\mathcal{O}$ is to exploit the so-called proof trees of $\mathcal{O}$. Intuitively, a proof tree T of $\mathcal{O}$ is a finite node-labeled tree, whose nodes are labeled with CQs, and describes one of the possible sequences of CQs generated during the unfolding of the query q using the TGDs in Σ. Roughly, the root of T is labeled with (a slight modification of) the query q, and if there is a node u of T labeled with some query q' and with $n > 0$ children $u_1, \ldots, u_n$, where u_i is labeled with a CQ q_i, for $i \in [n]$, then it means there is a way to rewrite q' to a new CQ q'' using one of the TGDs in Σ, and $q_1, \ldots, q_n$ forms a decomposition of q'' into n smaller queries that can be later unfolded independently. The CQs labeling the leaves of T induce a k-ary CQ q_T which is shown in [20] to enjoy the following property: for every database D, and every tuple $\bar{t} \in \mathsf{dom}(D)^k$, $\bar{t} \in \mathsf{ans}(\mathcal{O}, D)$ iff there exists a proof tree T of $\mathcal{O}$ such that $\bar{t} \in q_T(D)$.

The key result from [20] that allows to show that WARDED is Datalog-rewritable is

that $\Sigma \in \mathsf{WARDED}$ implies that for every database D, and every tuple $\bar{t} \in \mathsf{dom}(D)^k$, $\bar{t} \in \mathsf{ans}(\mathcal{O}, D)$ iff there exists a proof tree T of $\mathcal{O}$ with $\bar{t} \in q_T(D)$ such that each node of T is labeled with a CQ whose body contains at most $f_{\text{WARD}}(q, \Sigma)$ atoms, where

$$f_{\text{WARD}}(q, \Sigma) = 2 \cdot \max\{|\mathsf{body}(q)|, \max_{\sigma \in \Sigma}\{|\mathsf{body}(\sigma)|\}\}.$$

The above result leads to the following rewriting algorithm, which we describe informally. Given a k-ary OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in \mathsf{WARDED}$, the algorithm constructs a Datalog rewriting $\mathcal{D}_\mathcal{O} = (R_\mathcal{O}, \Sigma_\mathcal{O})$ of $\mathcal{O}$ as follows: for each proof tree T of $\mathcal{O}$ whose labels have a body with at most $f_{\text{WARD}}(q, \Sigma)$ atoms, and for each node u of T with n children, add to $\Sigma_\mathcal{O}$ a full, single-head TGD σ_u with n body atoms, where the head of σ_u is obtained from the CQ labeling u, and the i-th atom in the body of σ_u is obtained from the CQ labeling the i-th child of u, for $i \in [n]$. Moreover, if u is the root of T, and R_T is the predicate in the head of σ_u, then also add the full, single-head TGD of the form $R_T(x_1, \ldots, x_k) \rightarrow R_\mathcal{O}(x_1, \ldots, x_k)$.

We identify two key practical limitations of the above theoretical rewriting algorithm:

- The theoretical algorithm naively explores every possible proof tree having nodes labeled with a CQ whose body has at most $f_{\text{WARD}}(q, \Sigma)$ atoms. However, not all such proof trees may need to be considered, since they could lead to a set of Datalog rules that are subsumed by the set of Datalog rules obtained from other proof trees.

- Besides the bound on the size of the body of the CQs labeling the nodes of a proof tree, the theoretical algorithm does not exploit any further properties of warded ontologies. In particular, while constructing a proof tree T, the algorithm needs to make some arbitrary assumptions on which positions will only contain constants during the construction of the chase, in order to guide the decomposition of a CQ labeling a node to smaller independent queries; this is required for the completeness of the algorithm. However, considering all such combinations leads to a large number of proof trees to consider, which might instead be safely ignored, as they do not contribute in adding new rules to the final rewriting.

In the rest of this chapter, our goal is to define an alternative rewriting algorithm that does not explicitly rely on proof trees and further exploits the key properties of warded ontologies to both guarantee correctness and efficiency.

7.2 Setting the Ground

In this section, we introduce the basic build blocks of our rewriting algorithm. At the high-level, given an OMQ $\mathcal{O} = (q, \Sigma)$, with $\Sigma \in \mathsf{WARDED}$, 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. This is required for the termination of the algorithm, as we will later show that wardedness guarantees that such small queries never contain more than $f_{\text{WARD}}(q, \Sigma)$ atoms, and thus the algorithm will eventually terminate. We point out that the decomposition step we employ in our algorithm is different than the one employed in the theoretical algorithm of [20], in that our decomposition step constructs an "optimal" decomposition of a query, i.e., the subqueries obtained by this step cannot be decomposed any further. This has then an effect not only on termination, but also on efficiency, as it allows to minimize the time required by the algorithm to unfold the obtained subqueries by means of the resolution step. 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. It is well-known that warded ontologies are not UCQ-rewritable, since ontologies made of full TGDs are trivially warded, and OMQs using only full TGDs are not UCQ-rewritable (e.g., see [1]).

Remark: The fact that our algorithm WardedRewrite exhaustively applies resolution and (optimal) decomposition steps means that the queries produced during its execution essentially correspond to nodes of the proof trees used by the theoretical algorithm. Thus, our algorithm does not construct and evaluate every possible proof tree whose nodes are labeled with CQs of size at most $f_{\text{WARD}}(q, \Sigma)$ explicitly, but instead it encodes them compactly, by reusing the query labels of a proof tree for another proof tree. Furthermore, the use of an optimal decomposition step means that such query labels remain as small as possible. We now proceed to formally define the two main steps of our algorithm.

Chunk-based Resolution. We start discussing the notions needed to implement the resolution step of our algorithm. These notions are somehow standard in query rewriting algorithms under TGDs. For this, we first need to recall the notion of chunk-based resolution. We first illustrate the notion via an example borrowed from [20].

Example 7.2.1. Consider the following CQ q and TGD σ:

$$Q(x) \leftarrow R(x,y), S(y) \qquad \sigma : P(x') \rightarrow \exists y' R(x',y'), S(y'), T(y').$$

With a chunk-based resolution step we want to simulate the fact that during the construction of the chase of some database D w.r.t. $\Sigma = \{\sigma\}$, triggering the TGD σ produces some atoms to which a portion (chunk) of q can be homomorphically mapped. Indeed, it should be clear that $R(x,y), S(y)$ is such a chunk, since for every database D, if σ can be triggered during the construction of $\mathsf{chase}(D,\Sigma)$, three atoms of the form $R(t,\bot)$, $S(\bot)$, $T(\bot)$ can be produced, where $t \in \mathbf{C} \cup \mathbf{N}$, and $\bot$ is a fresh null that is different from t. Hence, one can map $R(x,y), S(y)$ to the atoms $R(t,\bot), S(\bot)$ via the homomorphism $h = \{x \mapsto t, y \mapsto \bot\}$. The above is reflected in the chunk-based resolution step of q with the TGD σ, where we consider the substitution (actually a unifier)

$$\mu = \{x \mapsto x', y \mapsto y', x' \mapsto x', y' \mapsto y'\},$$

which we can exploit to rewrite q to a new CQ q' where we first replace the atoms $R(x,y), S(y)$ in q with the body of σ, and then apply the unifier μ to the result, obtaining the CQ

$$Q(x') \leftarrow P(x').$$

On the other hand, the single atom $R(x,y)$ cannot be considered a chunk of q that can be mapped to some atoms produced by σ, even if $R(x,y)$ and $R(x',y')$ unify via the unifier μ. Indeed, if we were to replace $R(x,y)$ in q with the body of σ, and then apply μ, we would obtain the CQ

$$Q(x') \leftarrow P(x'), S(y')$$

losing the fact that the atoms $R(x,y)$ and $S(y)$ in q were in a join (they share the variable y). This happens because the variable y is mapped to y', which is an existential variable of σ. Hence, when a variable y of the query unifies with an existential variable y' of the TGD, then no atom of the query mentioning y must be left out of the chunk of q.

We now recall the formal definition of 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 are now ready to recall the notion of chunk unifier.

Definition 3 (Chunk unifier)**.** *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.*

In other words, an MGCU describes a way to match a portion of the query q, i.e., the set S_1, with a portion of the head of a TGD σ, i.e., the set S_2, in a "safe" way. That is, the MGCU takes into account the fact that a variable y of S_1 unifying with an existential variable x of σ must not occur anywhere else in q (i.e., no atoms of q mentioning y are left out of the chunk S_1), and that y unifies with no other variable of S_2, or an output variable of q. These conditions are required to take into account that the null witnessing the existential variable x in S_2 is always going to be different from a term witnessing a frontier variable of σ or from another null produced by σ, or from the term witnessing an output variable (as this is always going to be a constant), and thus x cannot unify with another existential variable, or a frontier variable of σ, or an output variable of q.

With the above notion, we are now finally ready to define the chunk-based resolution.

Definition 4 (Chunk-based resolution)**.** *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 $\mathsf{body}(q') = \gamma((\mathsf{body}(q) \setminus S_1) \cup \mathsf{body}(\sigma))$.*

The following, which is implicit in [20], is the key property of chunk-based resolution which we will exploit to prove the correctness of our rewriting algorithm later in the thesis.

Proposition 1 (Implicit in [20]). *Consider an ontology Σ, a CQ q, and a TGD $\sigma \in \Sigma$. For every σ-resolvent q' of q, and for every database D, the following holds:*

$$\mathsf{ans}((q', \Sigma), D) \subseteq \mathsf{ans}((q, \Sigma), D).$$

Query Decomposition. Now we define the notions we need to implement the other step of our rewriting algorithm: the decomposition step. We start with the notion of existential-join decomposition, which we adapt from [41].

Definition 5. *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 $\mathcal{S}$ is such that 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 both mention a variable $y \notin \bar{x}$ that occurs only in affected positions of $\mathsf{body}(q)$, then $\beta \in \mathsf{body}(q_i)$ as well.*

Among all existential-join decompositions, we are interested in "optimal" ones, i.e., decompositions whose queries cannot be decomposed further. Formally,

Definition 6 (Optimal decomposition). *Consider a CQ $q(\bar{x})$ and an ontology Σ. An* optimal decomposition of q w.r.t. Σ *is an existential-join decomposition of the form $\mathcal{S} = \{q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)\}$ of q w.r.t. Σ such that, for each $i \in [n]$, there is no existential-join decomposition of q_i w.r.t. Σ.*

Example 7.2.2. Consider again the CQ $q(x)$ and the TGD σ of Example 7.2.1. It is not difficult to verify that $(R, 2)$, $(S, 1)$, $(T, 1)$ are all the affected positions of $\Sigma = \{\sigma\}$, while $(P, 1)$ and $(R, 1)$ are not affected. By definition of existential-join decomposition, it is not possible to decompose the body of q in two subqueries, since $\mathsf{body}(q) = \{R(x, y), S(y)\}$, and the two atoms both mention a variable, i.e., y, that is not an output variable of q, and occurs only in affected positions, i.e., $(R, 2)$ and $(S, 1)$. Hence, no existential-join decomposition of q w.r.t. Σ exists.

If instead we consider the TGD $\sigma' = P(x') \rightarrow \exists y' R(x', y'), S(x'), T(y')$, the affected positions of $\Sigma' = \{\sigma'\}$ are only $(R, 2)$ and $(T, 1)$, while $(P, 1)$, $(R, 1)$, and $(S, 1)$ are non-affected. Hence, the atoms $R(x, y)$ and $S(y)$ in the body of q both mention a non-output variable, i.e., y, which occurs in at least one non-affected position, i.e., $(S, 1)$. Hence, an existential-join decomposition of q w.r.t. Σ' is the set $\{q_1(x, y), q_2(x)\}$, where q_1 is the CQ $Q_1(x, y) \leftarrow R(x, y)$ and q_2 is the CQ $Q_2(y) \leftarrow S(y)$. Clearly, the decomposition is optimal, since both q_1 and q_2 have only one atom in their body.

The main idea behind an optimal decomposition of a CQ $q(\bar{x})$ is to factorize the body of q into the smallest, independent subqueries, where by independent we mean that the certain answers of (q, Σ) can be equivalently constructed by combining the certain answers of each OMQ (q_i, Σ), for $i \in [n]$. We now formalize the above idea.

Consider a CQ $q(\bar{x})$, with $\mathsf{headPred}(q) = Q$, an ontology Σ, and assume $\mathcal{S}$ is an existential-join decomposition $\{q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)\}$ of q w.r.t. Σ, with $\mathsf{headPred}(q_i) = Q_i$, 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 answers of each member of $\mathcal{S}$ as the original CQ q does.

7.3 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 $Q(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$. We extend the above notations to sets in the natural way. We now have all the notions we need to present our rewriting algorithm, dubbed WardedRewrite, which we report in Algorithm 1.

Consider an OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in \mathsf{WARDED}$. We start with a general overview of the algorithm, and then proceed to describe it in more details. Recall that the main idea behind the algorithm WardedRewrite is to iteratively apply, starting from q, one of two steps: decomposition and resolution. The goal is to first decompose the query q into smaller subqueries, if possible, otherwise we keep q as it is; this is the decomposition step. Then, each query q' obtained at the decomposition step is unfolded by constructing all possible σ-resolvents of q', for all TGDs $\sigma \in \Sigma$; this is a single resolution step, which produces a set of new queries. Then, the process repeats where each new query is first decomposed optimally, if possible, and then its subqueries are unfolded again.

Algorithm 1: WardedRewrite

Input: An OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in$ WARDED
Output: A Datalog rewriting of $\mathcal{O}$

$\mathcal{Q}_{\text{current}} := \{\mathsf{can}(q)\}$; $\mathcal{Q}_{\text{explored}} := \emptyset$; $\mathcal{Q}_{\text{dec}} := \emptyset$; $\mathcal{R} := \emptyset$;
while $\mathcal{Q}_{\text{current}} \neq \emptyset$ **do**
  $\mathcal{Q}_{\text{new}} := \emptyset$;
  foreach $q' \in \mathcal{Q}_{\text{current}}$ **do**
    if *a decomposition of* q' *w.r.t.* Σ *exists* **then**
      // Decomposition step
      $\mathcal{S} := \mathsf{decomposeOptimal}(q', \Sigma)$;
      $\rho := \mathsf{reconciliationRule}(q', \mathcal{S})$;
      foreach $q'' \in \mathcal{S}$ **do**
        if *there is* $q_{\text{alt}} \in \mathcal{Q}_{\text{dec}}$ *with* $q_{\text{alt}} \approx q''$ **then**
          Replace $\mathsf{headPred}(q'')$ with $\mathsf{headPred}(q_{\text{alt}})$ in $\mathsf{body}(\rho)$;
        else
          $\mathcal{Q}_{\text{dec}} := \mathcal{Q}_{\text{dec}} \cup \{\mathsf{can}(q'')\}$;
          $\mathcal{Q}_{\text{new}} := \mathcal{Q}_{\text{new}} \cup \{\mathsf{can}(q'')\}$;
      $\mathcal{R} := \mathcal{R} \cup \{\rho\}$;
    else
      // Resolution step
      foreach $\sigma \in \Sigma$ **do**
        foreach *MGCU* M *of* q' *with* σ **do**
          $q'' := \mathsf{resolv}_{\sigma,M}(q')$;
          $\mathcal{Q}_{\text{new}} := \mathcal{Q}_{\text{new}} \cup \{\mathsf{can}(q'')\}$;
      $\mathcal{R} := \mathcal{R} \cup \{\mathsf{rule}(q')\}$;
    $\mathcal{Q}_{\text{explored}} := \mathcal{Q}_{\text{explored}} \cup \{q'\}$;
  $\mathcal{Q}_{\text{current}} := \mathcal{Q}_{\text{new}} \setminus \mathcal{Q}_{\text{explored}}$;
return $(\mathsf{headPred}(q), \mathcal{R})$;

Whenever a query can be decomposed, the connection between the queries produced by the decomposition step is kept by means of the corresponding reconciliation rule. The process stops when no new queries (up to variable renaming) can be produced. The final Datalog rewriting is made of all reconciliation rules together with all the unfoldings of the queries considered.

We now describe the algorithm WardedRewrite in more details. In what follows, 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 $\mathsf{can}(q_1) = \mathsf{can}(q_2)$. Furthermore, given two CQs q_1 and q_2 in canonical form, we write $q_1 \approx q_2$ iff q_1 and q_2 are the same up to head predicate renaming, i.e., assuming we replace the head predicate of q_1 and q_2 with the same predicate, we have $q_1 = q_2$.

Initialization. The algorithm WardedRewrite takes as input an OMQ $\mathcal{O} = (q, \Sigma)$ with $\Sigma \in$ WARDED. It first initializes (line 1) the sets $\mathcal{Q}_{\text{current}} = \{\mathsf{can}(q)\}$, $\mathcal{Q}_{\text{explored}} = \emptyset$, $\mathcal{Q}_{\text{dec}} = \emptyset$, $\mathcal{R} = \emptyset$. In particular, at the beginning of each iteration of the algorithm, $\mathcal{Q}_{\text{current}}$ contains the canonical version of all the queries that need to be processed at the current iteration, and thus it initially contains $\mathsf{can}(q)$; $\mathcal{Q}_{\text{explored}}$ contains the canonical version of all queries that have already been processed up to this iteration, and thus do not need to be considered again; $\mathcal{Q}_{\text{dec}}$ collects the canonical version of all subqueries constructed during the past decomposition steps, and we will give more details on its role later in the discussion; $\mathcal{R}$ contains all the rules of the final Datalog rewriting up to this iteration.

Main Loop. The while loop between lines 2-22 is the main loop of the algorithm, and at each iteration is in charge of processing (lines 4-21) all queries in $\mathcal{Q}_{\text{current}}$, if there are any. At the end of one iteration of the while loop, the set $\mathcal{Q}_{\text{new}}$ contains all the queries obtained by processing the queries in $\mathcal{Q}_{\text{current}}$ using either a decomposition or a resolution step. The set of current queries is updated with the queries in $\mathcal{Q}_{\text{new}}$ that have never been considered yet, i.e., $\mathcal{Q}_{\text{current}} := \mathcal{Q}_{\text{new}} \setminus \mathcal{Q}_{\text{explored}}$ (line 22).[1] If at the next iteration, no queries need to be processed, i.e., $\mathcal{Q}_{\text{current}} = \emptyset$, the main loop terminates, and the final Datalog rewriting is returned in line 23 (more details on this later).

Query Processing. We now discuss how each query $q' \in \mathcal{Q}_{\text{current}}$ is processed by the algorithm in lines 4-21

Decomposition Step. First, the algorithm verifies whether q' can be decomposed into smaller, independent subqueries (line 5). If this is the case, then the algorithm computes an optimal decomposition $\mathcal{S}$ of q', together with the corresponding reconciliation rule ρ, using procedures decomposeOptimal and reconciliationRule (lines 6-7). The reconciliation rule ρ, together with all queries in $\mathcal{S}$, play the role of the single query q', and thus, it is the queries in $\mathcal{S}$ that actually need to be unfolded at later iterations.

From the above discussion, one might expect that at this point, it is enough to add all the CQs in $\mathcal{S}$ to the set $\mathcal{Q}_{\text{new}}$ for later processing, and add ρ to $\mathcal{R}$. However, this might make the algorithm not terminate, since each time a query q' is decomposed,

[1] 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, up to variable renaming.

the algorithm produces a set $\mathcal{S}$ of new queries with a fresh new head predicate, and thus these CQs will necessarily be unfolded at later iterations of the while loop via a resolution step. Thus, when decomposing at later iterations any of the unfoldings of such queries, we might construct again new subqueries with fresh head predicates, and so on. To prevent this, the algorithm keeps (the canonical version of) all queries produced by all decomposition steps up to the current iteration in the set $\mathcal{Q}_{\text{dec}}$. Then, for each CQ q'' of the decomposition, the algorithm first checks whether any previous decomposition step has already produced a query $q_{\text{alt}} \in \mathcal{Q}_{\text{dec}}$ which, except for the head predicate, is identical to q'', i.e., $q_{\text{alt}} \approx q''$ (line 9). If this is the case, we do not need to unfold q'' later, and we can safely reuse the CQ q_{alt} in place of q'' by simply replacing the occurrence of $\mathsf{headPred}(q'')$ in the body of ρ with $\mathsf{headPred}(q_{\text{alt}})$ (line 10). Otherwise, 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 (line 12). Then, the (modified) rule ρ is added to the set $\mathcal{R}$ of all rules of the Datalog rewriting (line 14).

Resolution Step. If the query q' cannot be decomposed, then the algorithm proceeds with a resolution step, where it exhaustively unfolds q' by considering all TGDs of Σ via MGCUs (lines 16-19). That is, for each TGD $\sigma \in \Sigma$, and for each MGCU $M = (S_1, S_2, \mu)$ of q' with σ, it constructs the σ-resolvent q'' of q' via M, and then adds the canonical version of q'' to the set of new queries $\mathcal{Q}_{\text{new}}$. After the resolution step, the algorithm adds the full TGD corresponding to the current CQ q' to the rules of the final Datalog rewriting (line 20).

Mark Current Query as Explored. Once a decomposition or a resolution step is executed, the processing of q' concludes at line 21 by adding q' to the set $\mathcal{Q}_{\text{explored}}$ of processed queries.

Constructing the Datalog Rewriting. Once the main while loop of the algorithm terminates, the set $\mathcal{R}$ contains all the rules of the Datalog rewriting of $\mathcal{O}$. Then, the algorithm outputs the Datalog query $(\mathsf{headPred}(q), \mathcal{R})$, where the head predicate of the original query q will collect all the answers.

Chapter 8

Correctness of WardedRewrite

The goal of this chapter is to prove that, for a given OMQ $\mathcal{O} = (q, \Sigma)$, with $\Sigma \in$ WARDED, the algorithm WardedRewrite always terminates, and produces a Datalog rewriting $\mathcal{D}_{\mathcal{O}}$ of $\mathcal{O}$. That is, we prove the following result:

Theorem 2. *For every OMQ $\mathcal{O} = (q, \Sigma)$, with $\Sigma \in$ WARDED, WardedRewrite terminates with input $\mathcal{O}$, and outputs a Datalog rewriting of $\mathcal{O}$.*

The proof of the above theorem is somehow involved, and we split it into three main steps, where each step proves one key property of the Datalog query produced by the algorithm. We discuss each step separately in the following subsections. For the readers who would first like to move to the implementation details and experimental evaluation, they may choose to skip this chapter and proceed directly to Chapter 9, as the current chapter is dedicated solely to the proofs.

8.1 Termination of WardedRewrite

In this section, we prove that when considering warded ontologies, WardedRewrite always terminates. Towards this end, we first establish an auxiliary result.

Lemma 1. *Consider an ontology $\Sigma \in$ WARDED, a TGD $\sigma \in \Sigma$, and a CQ $q(\bar{x})$. For every MGCU M of q with σ, the σ-resolvent q' of q via M is such that either $|\mathsf{body}(q'')| \leq |\mathsf{body}(q)|$, or q' admits an existential-join decomposition $\mathcal{S}$ w.r.t. Σ such that, for each query $q'' \in \mathcal{S}$, $|\mathsf{body}(q'')| \leq \max\{|\mathsf{body}(q)|, |\mathsf{body}(\sigma)|\}$.*

Proof. It is enough to prove that when $|\mathsf{body}(q')| > |\mathsf{body}(q)|$, then q' admits an existential-join decomposition $\mathcal{S}$ w.r.t. Σ such that, for each query $q'' \in \mathcal{S}$, $|\mathsf{body}(q'')| \leq$

$\max\{|\mathsf{body}(q)|, |\mathsf{body}(\sigma)|\}$. Assume $|\mathsf{body}(q')| > |\mathsf{body}(q)|$, and let $M = (S_1, S_2, \gamma)$, with $S_1 \subseteq \mathsf{body}(q)$, and $S_2 \subseteq \mathsf{head}(\sigma)$, and assume $q(\bar{x})$ is of the form:

$$Q(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n).$$

Since $|\mathsf{body}(q')| > |\mathsf{body}(q)|$, it must be the case that $\mathsf{body}(\sigma)$ contains at least two atoms, one of which is the ward of σ. Hence, let $\mathsf{body}(\sigma) = \{P_1(\bar{y}_1), \ldots, P_m(\bar{y}_m)\}$, with $m > 1$, and assume, w.l.o.g., that $P_1(\bar{y}_1)$ is the ward of σ, and that S_1 contains the atoms $R_1(\bar{x}_1), \ldots, R_k(\bar{x}_k)$, for some $k \in [n]$. Then, the σ-resolvent q' of q via M is of the form:

$$Q(\gamma(\bar{x})) \leftarrow P_1(\gamma(y_1)), \ldots, P_m(\gamma(y_m)), R_{k+1}(\gamma(x_{k+1})), \ldots, R_n(\gamma(\bar{x}_n)).$$

Consider the following subsets of the body of q':

$$B_1 = \{P_1(\gamma(\bar{y}_1)), R_{k+1}(\gamma(x_{k+1})), \ldots, R_n(\gamma(\bar{x}_n))\},$$

$$B_2 = \{P_2(\gamma(\bar{y}_2)), \ldots, P_m(\gamma(\bar{y}_m))\}.$$

Assuming $\bar{x}_1 = \mathsf{shared}(q', B_1)$ and $\bar{x}_2 = \mathsf{shared}(q', B_2)$, we claim that the set $\mathcal{S} = \{Q_1(\bar{x}_1) \leftarrow B_1, Q_2(\bar{x}_2) \leftarrow B_2\}$ is an existential-join decomposition of q' w.r.t. Σ where the body of each query therein contains at most $\max\{|\mathsf{body}(q)|, |\mathsf{body}(\sigma)|\}$ atoms. Clearly, $|B_1|$ and $|B_2|$ are both smaller or equal than $\max\{|\mathsf{body}(q)|, |\mathsf{body}(\sigma)|\}$, by construction. Moreover, B_1 is non-empty by construction, while B_2 is not empty because $m > 1$. We are only left to prove that $\mathcal{S}$ is an existential-join decomposition of q' w.r.t. Σ. For the latter, it is enough to show that for every atom $\alpha \in B_1$, and every variable $z \in \mathsf{var}(\alpha)$ that also occurs in B_2, either $z \in \gamma(\bar{x})$, or z occurs in at least one non-affected position in $\mathsf{body}(q')$. We distinguish two cases, depending on the shape of the atom $\alpha \in B_1$.

Assume first that $\alpha = P_1(\gamma(\bar{y}_1))$, and assume, towards a contradiction, that there is a variable $z \notin \gamma(\bar{x})$ in α that also occurs in B_2, and occurs only in affected positions in $\mathsf{body}(q')$. Since $P(\bar{y}_1)$ is the ward of σ, by definition of warded TGD, we know that all the variables that $P(\bar{y}_1)$ shares with the atoms $P_2(\bar{y}_2), \ldots, P_m(\bar{y}_m)$ must appear in at least one non-affected position in $\mathsf{body}(\sigma)$. Hence, if α and B_2 share the variable z occurring only in affected positions of $\mathsf{body}(q')$, this is necessarily because $z = \gamma(z_1) = \gamma(z_2)$, where z_1 and z_2 are two distinct variables such that z_1 occurs in $P_1(\bar{y}_1)$, and z_2 is a harmful variable of σ that occurs in $P_j(\bar{y}_j)$, for some $j \in \{2, \ldots, m\}$. Moreover,

if γ unifies z_1 and z_2 to z, it must be the case that both z_1 and z_2 occur in the head of σ. The fact that z_2 is harmful in σ and that z_2 occurs in the head of σ implies, by wardedness of σ, that z_2 must also appear in $P(\bar{y}_1)$. However, this means that $P_1(\bar{y}_1)$ and $P_j(\bar{y}_j)$ share the harmful variable z_2. But since the ward cannot share harmful variables with the other atoms in the body of σ, we obtain a contradiction.

Assume now that $\alpha = R_j(\gamma(\bar{x}_j))$, for some $j \in \{k+1, \ldots, n\}$, and assume, towards a contradiction, that there exists a variable $z \notin \gamma(\bar{x})$ in α that also occurs in B_2, and occurs only in affected positions in $\mathsf{body}(q')$. Recall that we assume that CQs and TGDs do not share any variable, when constructing MGCUs. Hence, the atoms $P_2(\bar{y}_2), \ldots, P_m(\bar{y}_m)$ do not share any variable with the atoms $R_{k+1}(\bar{x}_{k+1}), \ldots, R_n(\bar{x}_n)$. Hence, if z occurs in both $\alpha = R_j(\gamma(\bar{x}_j))$ and in some atom of B_2, this is because there are two distinct variables z_1 and z_2, with $z = \gamma(z_1) = \gamma(z_2)$, such that z_1 occurs in $R_j(\bar{x}_j)$, and z_2 occurs in one or more atoms in $\{P_2(\bar{y}_2), \ldots, P_m(\bar{y}_m)\}$, and also in the head of σ. Since z occurs only in affected positions of $\mathsf{body}(q')$, by hypothesis, z_2 must be harmful in σ. The latter, together with the fact that z_2 occurs in the head of σ, and the wardedness of σ, implies that z_2 must occur in the ward of σ, i.e., $P_1(\bar{y}_1)$. However, by wardedness again, the ward cannot share harmful variables with the atoms in $\{P_2(\bar{y}_2), \ldots, P_m(\bar{y}_m)\}$, obtaining a contradiction. □

With the above lemma in place, we can prove that when we focus on warded ontologies, WardedRewrite always terminates.

Theorem 3. *For every input OMQ $\mathcal{O} = (q, \Sigma)$, with $\Sigma \in$ WARDED, WardedRewrite terminates.*

Proof. Assume $\mathcal{O} = (q, \Sigma)$ is the OMQ, with $\Sigma \in$ WARDED, given as input to the algorithm WardedRewrite, and consider now the i-th iteration of the while loop of WardedRewrite when executed with $\mathcal{O}$ as input, for some $i > 0$. We use $\mathcal{Q}^i_{\text{current}}$, $\mathcal{Q}^i_{\text{new}}$, and $\mathcal{Q}^i_{\text{explored}}$ to denote the content of the sets $\mathcal{Q}_{\text{current}}$, $\mathcal{Q}_{\text{new}}$, and $\mathcal{Q}_{\text{explored}}$ respectively, at the very end of iteration i of the while loop (i.e., just after line 22). We note that $\mathcal{Q}^1_{\text{explored}} = \{\mathsf{can}(q)\}$, and for each iteration $i > 1$,

$$\mathcal{Q}^i_{\text{explored}} = \mathcal{Q}^{i-1}_{\text{explored}} \cup \mathcal{Q}^{i-1}_{\text{new}}.$$

That is, the set of all queries explored by the algorithm up to iteration i contains the canonical version of the input query q, and all queries produced by the algorithm up to the previous iteration. Since $\mathcal{Q}^i_{\text{current}} = \mathcal{Q}^i_{\text{new}} \setminus \mathcal{Q}^i_{\text{explored}}$, and since WardedRewrite

terminates iff there exists an iteration i such that $\mathcal{Q}^i_{\text{current}} = \emptyset$, we can show that WardedRewrite terminates by showing the following lemma:

Lemma 2. *There exists an integer $k_{\mathcal{O}}$, which depends only on $\mathcal{O}$, such that for each iteration $i > 0$ of the while loop of* WardedRewrite, $|\mathcal{Q}^i_{\text{explored}}| \leq k_{\mathcal{O}}$.

Indeed, by the above lemma, and from the fact that $\mathcal{Q}^j_{\text{new}} \subseteq \mathcal{Q}^i_{\text{explored}}$, for each $1 \leq j \leq i-1$, there must be an iteration $\ell > k_{\mathcal{O}}$, such that $\mathcal{Q}^\ell_{\text{new}} \subseteq \mathcal{Q}^\ell_{\text{explored}}$, otherwise $|\mathcal{Q}^\ell_{\text{explored}}| > k_{\mathcal{O}}$. However, the fact that $\mathcal{Q}^\ell_{\text{new}} \subseteq \mathcal{Q}^\ell_{\text{explored}}$ implies that $\mathcal{Q}^\ell_{\text{current}} = \mathcal{Q}^\ell_{\text{new}} \setminus \mathcal{Q}^\ell_{\text{explored}} = \emptyset$, and thus WardedRewrite terminates after the ℓ-th iteration.

The rest of the proof is devoted to prove Lemma 2. Towards this end, we introduce some auxiliary notation. Consider an iteration $i > 0$ of the while loop of WardedRewrite. Note that at iteration i, a query can be added to the set $\mathcal{Q}^i_{\text{new}}$ only during a decomposition step in lines 12-13, or during a resolution step in line 19. We use $\mathcal{Q}^{i,d}_{\text{new}}$ and $\mathcal{Q}^{i,r}_{\text{new}}$ to denote the set of queries in $\mathcal{Q}^i_{\text{new}}$ obtained during a decomposition step and a resolution step, respectively. If a query q' is added more than once, at iteration i, to the set $\mathcal{Q}^i_{\text{new}}$, e.g., because a resolution step adds q' after a decomposition step of the same iteration already added q', then we assume the query q' belongs to the set corresponding to the step that added q' to $\mathcal{Q}^i_{\text{new}}$ first. Hence, the two sets $\mathcal{Q}^{i,d}_{\text{new}}$ and $\mathcal{Q}^{i,r}_{\text{new}}$ form a partition of $\mathcal{Q}^i_{\text{new}}$. We are now ready to prove Lemma 2. In what follows, we recall that $f_{\text{WARD}}(q, \Sigma) = 2 \cdot \max\{|\mathsf{body}(q)|, \max_{\sigma \in \Sigma}\{|\mathsf{body}(\sigma)|\}\}$, and we use $g_{\text{WARD}}(q, \Sigma)$ to denote the number $\max\{|\mathsf{body}(q)|, \max_{\sigma \in \Sigma}\{|\mathsf{body}(\sigma)|\}\}$. To prove the lemma, we prove the following two properties, for each iteration $i > 0$:

1. There are no two queries $q_1, q_2 \in \mathcal{Q}^i_{\text{explored}} \setminus \{\mathsf{can}(q)\}$ with $q_1 \approx q_2$.
2. For each query $q^* \in \mathcal{Q}^i_{\text{new}}$:
   (a) if $q^* \in \mathcal{Q}^{i,d}_{\text{new}}$, then $|\mathsf{body}(q^*)| \leq g_{\text{WARD}}(q, \Sigma)$ and q^* does not admit an existential-join decomposition w.r.t. Σ;
   (b) if $q^* \in \mathcal{Q}^{i,r}_{\text{new}}$, then $|\mathsf{body}(q^*)| \leq f_{\text{WARD}}(q, \Sigma)$ and q^* is such that either $|\mathsf{body}(q^*)| \leq g_{\text{WARD}}(q, \Sigma)$, or q^* admits an existential-join decomposition $\mathcal{S}$ w.r.t. Σ, where the body of each query in $\mathcal{S}$ has size at most $g_{\text{WARD}}(q, \Sigma)$.

Items (1) and (2), together with the fact that $g_{\text{WARD}}(q, \Sigma) \leq f_{\text{WARD}}(q, \Sigma)$, and that $\mathcal{Q}^i_{\text{explored}} = \mathcal{Q}^{i-1}_{\text{explored}} \cup \mathcal{Q}^{i-1}_{\text{new}}$, for $i > 1$, immediately imply that for each $i > 0$, $|\mathcal{Q}^i_{\text{explored}}|$ is bounded by $1 + k$, where k is the total number of sets of atoms with at most $f_{\text{WARD}}(q, \Sigma)$ atoms that can be formed using predicates from $\mathsf{sch}(\Sigma)$, up to variable

renaming. Hence, Lemma 2 holds with $k_{\mathcal{O}} = 1 + k$. We are only left to prove item (1) and item (2).

Item (1). This item follows immediately from the fact that a query q^* with a fresh head predicate becomes part of $\mathcal{Q}^i_{\text{explored}}$ only if $q^* = \mathsf{can}(q'')$ is added to $\mathcal{Q}^{i-1}_{\text{new}}$, at iteration $i-1$, where q'' is obtained at lines 12-13, i.e., $q^* \in \mathcal{Q}^{i,d}_{\text{new}}$. However, the latter is done only if no other query q_{alt} has already been considered with $q_{\text{alt}} \approx q^*$.

Item (2). We prove the claim by induction on the iteration i. Assume first that $i = 1$. Thus, at the beginning of iteration i, $\mathcal{Q}_{\text{current}}$ contains only the query $\mathsf{can}(q)$. Thus, the queries in $\mathcal{Q}^i_{\text{new}}$ are obtained either all via a decomposition step over $\mathsf{can}(q)$, or all via a resolution step over $\mathsf{can}(q)$. If $\mathsf{can}(q)$ admits an existential-join decomposition w.r.t. Σ, then, each query q^* in $\mathcal{Q}^i_{\text{new}}$ actually belongs to $\mathcal{Q}^{i,d}_{\text{new}}$, and $|\mathsf{body}(q^*)| \leq g_{\text{WARD}}(q, \Sigma)$ by the fact that an optimal decomposition of $\mathsf{can}(q)$ w.r.t. Σ can only produce queries that are smaller than $\mathsf{can}(q)$. The fact that q^* has no existential-join decomposition w.r.t. Σ follows from the optimality of the decomposition constructed by WardedRewrite, and the claim follows. If instead $\mathsf{can}(q)$ does not admit an existential-join decomposition w.r.t. Σ, then each query q^* in $\mathcal{Q}^i_{\text{new}}$ actually belongs to $\mathcal{Q}^{i,r}_{\text{new}}$. In particular, each query $q^* \in \mathcal{Q}^{i,r}_{\text{new}}$ is such that $q^* = \mathsf{can}(q'')$, where q'' is the σ-resolvent of $\mathsf{can}(q)$ via M, for some TGD $\sigma \in \Sigma$, and MGCU M of $\mathsf{can}(q)$ with σ. Hence, the body of q'' can contain at most a number of atoms equal to $|\mathsf{body}(q)| + |\mathsf{body}(\sigma)| - 1 \leq f_{\text{WARD}}(q, \Sigma)$. We are left to show that either $|\mathsf{body}(q'')| \leq g_{\text{WARD}}(q, \Sigma)$, or q'' admits an existential-join decomposition $\mathcal{S}$ w.r.t. Σ with each query in $\mathcal{S}$ having a body of size at most $g_{\text{WARD}}(q, \Sigma)$. The latter follows from the fact that q'' is the σ-resolvent of $\mathsf{can}(q)$ via M, and by Lemma 1.

Assume now that $i > 1$, and consider a query $q^* \in \mathcal{Q}^i_{\text{new}}$. We distinguish two cases, depending on whether $q^* \in \mathcal{Q}^{i,d}_{\text{new}}$ or $q^* \in \mathcal{Q}^{i,r}_{\text{new}}$. If $q^* \in \mathcal{Q}^{i,d}_{\text{new}}$, then $q^* = \mathsf{can}(q'')$, where q'' is a query of the optimal decomposition $\mathcal{S}$ of q' w.r.t. Σ, where q' is a query from Q^{i-1}_{current}. Since, $\mathcal{Q}^j_{\text{current}} = \mathcal{Q}^j_{\text{new}} \setminus \mathcal{Q}^j_{\text{explored}}$, for each iteration $j > 0$, we conclude that actually $q' \in \mathcal{Q}^{i-1}_{\text{new}}$. Since q' admits an existential-join decomposition w.r.t. Σ, and $q' \in \mathcal{Q}^{i-1}_{\text{new}}$, it must be the case, by inductive hypothesis, that $q' \in \mathcal{Q}^{i-1,r}_{\text{new}}$, and each query of the optimal decomposition of q' w.r.t. Σ has a body of size at most $g_{\text{WARD}}(q, \Sigma)$. The latter, together with the definition of optimal decomposition, implies that q^* does not admit an existential-join decomposition w.r.t. Σ, and $|\mathsf{body}(q^*)| \leq g_{\text{WARD}}(q, \Sigma)$, and the claim follows. If $q^* \in \mathcal{Q}^{i,r}_{\text{new}}$, then $q^* = \mathsf{can}(q'')$, where q'' is the σ-resolvent of q' via M, for a query $q' \in \mathcal{Q}^{i-1}_{\text{current}}$, TGD $\sigma \in \Sigma$, and MGCU M of q' with σ. We recall again that $\mathcal{Q}^j_{\text{current}} = \mathcal{Q}^j_{\text{new}} \setminus \mathcal{Q}^j_{\text{explored}}$, for each iteration $j > 0$, and thus $q' \in \mathcal{Q}^{i-1}_{\text{new}}$. Since q' does not admit an existential-join decomposition w.r.t. Σ, we conclude, by inductive

hypothesis, that either $q' \in \mathcal{Q}_{\text{new}}^{i-1,r}$ and $|\mathsf{body}(q')| \leq g_{\text{WARD}}(q, \Sigma)$, or $q' \in \mathcal{Q}_{\text{new}}^{i-1,d}$, and $|\mathsf{body}(q')| \leq g_{\text{WARD}}(q, \Sigma)$.

Hence, in both cases, the body of the query q^*, which is the canonical version of the σ-resolvent q'' of q' via M can contain at most $g_{\text{WARD}}(q, \Sigma) + |\mathsf{body}(\sigma)| - 1$ atoms, which is smaller than $f_{\text{WARD}}(q, \Sigma)$. We are left to prove that either $|\mathsf{body}(q^*)| \leq g_{\text{WARD}}(q, \Sigma)$, or q^* admits an existential-join decomposition $\mathcal{S}$ w.r.t. Σ where the body of each query in $\mathcal{S}$ contains at most $g_{\text{WARD}}(q, \Sigma)$ atoms. From the fact that q'' is the σ-resolvent of q' via M, and by Lemma 1, we conclude that either $|\mathsf{body}(q'')| \leq g_{\text{WARD}}(q', \Sigma) = \max\{|\mathsf{body}(q')|, \max_{\sigma \in \Sigma}\{|\mathsf{body}(\sigma)|\}\}$, or q'' admits an existential-join decomposition $\mathcal{S}$ w.r.t. Σ, where the body of each query in $\mathcal{S}$ contains at most $g_{\text{WARD}}(q', \Sigma)$ atoms. Since we have already shown that $|\mathsf{body}(q')| \leq g_{\text{WARD}}(q, \Sigma)$, we note that $g_{\text{WARD}}(q', \Sigma) \leq \max\{g_{\text{WARD}}(q, \Sigma), \max_{\sigma \in \Sigma}\{|\mathsf{body}(\sigma)|\}\} \leq g_{\text{WARD}}(q, \Sigma)$. Hence, we conclude that either $|\mathsf{body}(q'')| \leq g_{\text{WARD}}(q, \Sigma)$, or q'' admits an existential-join decomposition $\mathcal{S}$ w.r.t. Σ such that each query in $\mathcal{S}$ has no more than $g_{\text{WARD}}(q, \Sigma)$ atoms, and the claim follows.

With item (1) and item (2) in place, Lemma 2 follows, and from previous discussions, Theorem 3 follows as well. □

8.2 Soundness of WardedRewrite

We now focus on the soundness of the algorithm WardedRewrite. That is, given an OMQ $\mathcal{O}$, it always produces a Datalog query $\mathcal{D}_{\mathcal{O}}$ such that $\mathcal{D}_{\mathcal{O}}(D) \subseteq \mathsf{ans}(\mathcal{O}, D)$, for every database D. In order to prove the above result, we first need to introduce some notions.

Consider an OMQ $\mathcal{O} = (q, \Sigma)$, with $\mathsf{headPred}(q) = Q$, and for which WardedRewrite terminates and outputs the Datalog query $\mathcal{D}_{\mathcal{O}} = (Q, \mathcal{R})$. For each predicate P occurring in the head of some rule in $\mathcal{R}$, the *root rule of* P is the first rule ρ added to $\mathcal{R}$ by WardedRewrite with head predicate P. Moreover, we use $\mathsf{refq}(\rho)$ to denote the query q', which we call the *reference query of* ρ, that WardedRewrite has used to construct ρ. In particular, note that when the algorithm WardedRewrite adds ρ to $\mathcal{R}$ during its execution, it does so by taking a CQ q' from the set $\mathcal{Q}_{\text{current}}$, at line 4, and by either letting $\rho = \mathsf{rule}(q')$ (line 20), i.e., ρ is nothing else than q' in rule form, or by letting ρ be the reconciliation rule (after some modifications) of q' and the optimal decomposition of q' w.r.t. Σ (line 14). Regardless of how ρ has been obtained, we use $\mathsf{refq}(\rho)$ to denote the query q'. With an abuse of notation, for a rule $\rho' \in \mathcal{R}$ with head predicate P which is not necessarily a root rule of P, we use $\mathsf{refq}(\rho')$ to denote $\mathsf{refq}(\rho)$, where ρ is the root

rule of P. Finally, for a rule $\rho \in \mathcal{R}$ of the form $R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n) \rightarrow P(\bar{x})$, we use $\mathsf{freshcq}(\rho)$ to denote the CQ $P'(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$, where P' a fresh predicate of the same arity of P, which is different from R_i, for $i \in [n]$.

With the above notation in place, we start by proving the following auxiliary result.

Lemma 3. *Consider an OMQ $\mathcal{O} = (q, \Sigma)$, with $\mathsf{headPred}(q) = Q$, and such that* WardedRewrite *terminates with input $\mathcal{O}$, producing the Datalog query $\mathcal{D}_\mathcal{O} = (Q, \mathcal{R})$. Then, for every database D, and every rule $\rho \in \mathcal{R}$, the following holds:*

$$\mathsf{ans}((\mathsf{freshcq}(\rho), \Sigma \cup \mathcal{R}), D) \subseteq \mathsf{ans}((\mathsf{refq}(\rho), \Sigma), D).$$

Proof. Let D be a database, and let $\rho \in \mathcal{R}$. We prove the claim by distinguishing on how ρ has been produced by WardedRewrite. Observe that ρ can only be of two possible forms, where $\mathsf{body}(\rho)$ either mentions only predicates in $\mathsf{sch}(\Sigma)$, i.e., it has been produced in line 20, or mentions only predicates from $\mathsf{sch}(\mathcal{R}) \setminus \mathsf{sch}(\Sigma)$, i.e., it has been produced in line 14 as the result of computing the reconciliation rule of a CQ. We prove the claim by distinguishing the above two cases.

Case 1. Assume that ρ has been produced in line 20. Note that the body of ρ mentions only predicates in $\mathsf{sch}(\Sigma)$, and thus $\mathsf{freshcq}(\rho)$ and $\mathsf{cq}(\rho)$ are the same, up to head predicate renaming. We say that a rule $\rho' \in \mathcal{R}$, with head predicate P, is at *level 0* if ρ' is the root rule of P, otherwise, ρ' is at *level i* for some $i > 0$ if $\mathsf{cq}(\rho')$ is obtained in line 18, during some iteration of the while loop, as the resolvent of $\mathsf{cq}(\rho'')$, for some rule $\rho'' \in \mathcal{R}$ at level $i - 1$, and ρ' did not yet appear in $\mathcal{Q}_{\text{explored}}$, during that iteration. Note that all rules in $\mathcal{R}$ that are produced in line 20 have a well-defined level. We now prove the claim by induction on the level of ρ.

Assume that ρ is at level 0, and assume there exists a tuple $\bar{t} \in \mathsf{ans}((\mathsf{freshcq}(\rho), \Sigma \cup \mathcal{R}), D)$. Since ρ has been produced in line 20, the body of $\mathsf{freshcq}(\rho)$ mentions only predicates from $\mathsf{sch}(\Sigma)$. Moreover, since only the TGDs in Σ have predicates from $\mathsf{sch}(\Sigma)$ in their head, and no head predicates from $\mathcal{R}$ occur in Σ, we conclude that $\bar{t} \in \mathsf{ans}((\mathsf{freshcq}(\rho), \Sigma \cup \mathcal{R}), D)$ implies $\bar{t} \in \mathsf{ans}((\mathsf{freshcq}(\rho), \Sigma), D)$. However, since ρ is at level 0, we have that $\mathsf{freshcq}(\rho)$ is the same as $\mathsf{refq}(\rho)$, modulo head predicate renaming, and the claim follows.

Assume now that ρ is at some level $i > 0$, and assume there exists a tuple $\bar{t} \in \mathsf{ans}((\mathsf{freshcq}(\rho), \Sigma \cup \mathcal{R}), D)$. As done in the base case, since the body of $\mathsf{freshcq}(\rho)$ mentions only predicates from $\mathsf{sch}(\Sigma)$, we have that $\bar{t} \in \mathsf{ans}((\mathsf{freshcq}(\rho), \Sigma \cup \mathcal{R}), D)$ implies $\bar{t} \in \mathsf{ans}((\mathsf{freshcq}(\rho), \Sigma), D)$. Moreover, since ρ is at level $i > 0$, there must be

a rule $\rho' \in \mathcal{R}$ at level $i-1$ such that $\mathsf{cq}(\rho) = q''$ is the σ-resolvent of $\mathsf{cq}(\rho') = q'$ via some MGCU M of q' with σ. By Proposition 1, and from the fact that $\mathsf{freshcq}(\rho)$ is the same as $\mathsf{cq}(\rho)$, modulo head predicate renaming, we conclude that $\bar{t} \in \mathsf{ans}((q', \Sigma), D)$, which in turn implies that $\bar{t} \in \mathsf{ans}((q', \Sigma \cup \mathcal{R}), D)$, by monotonicity of certain answers. Since, again, q' is the same as $\mathsf{freshcq}(\rho')$ modulo head predicate renaming, by inductive hypothesis, we conclude that $\bar{t} \in \mathsf{ans}((\mathsf{refq}(\rho'), \Sigma), D)$. Since both ρ and ρ' have the same head predicate, $\mathsf{refq}(\rho) = \mathsf{refq}(\rho')$, and the claim follows.

Case 2. Assume now that ρ has been produced in line 14. Then, ρ is (a modification of) the reconciliation rule of some CQ q' and its optimal decomposition w.r.t. Σ. Note that the body of ρ mentions only predicates in $\mathsf{sch}(\mathcal{R}) \setminus \mathsf{sch}(\Sigma)$, which only occur in the head of rules in $\mathcal{R}$, and do not occur anywhere in Σ. Assuming $q_\rho = \mathsf{freshcq}(\rho)$, and $q_r = \mathsf{refq}(\rho)$, we show the claim by proving that for every tuple $\bar{t} \in \mathsf{dom}(D)^k$, with k the arity of q_ρ, and for every $i > 0$, $\bar{t} \in q_\rho(\mathsf{chase}^i(D, \Sigma \cup \mathcal{R}))$ implies $\bar{t} \in q_r(\mathsf{chase}^i(D, \Sigma))$, for each $i > 0$; the case $i = 0$ is trivial, as no facts in D mention predicates in the body of q_ρ. The proof is by induction on i. In what follows, assume q_ρ is of the form:

$$Q_\rho(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n).$$

Assume $i = 1$, and assume that $\bar{t} \in q_\rho(\mathsf{chase}^i(D, \Sigma \cup \mathcal{R}))$. Then, there exists a homomorphism h from $\mathsf{body}(q_\rho)$ to $\mathsf{chase}^i(D, \Sigma \cup \mathcal{R})$ with $h(\bar{x}) = \bar{t}$. We conclude that $R_j(h(\bar{x}_j)) \in \mathsf{chase}^i(D, \Sigma \cup \mathcal{R})$, for each $j \in [n]$. The latter means that there must be rules $\rho_1, \ldots, \rho_n$, with R_j the head predicate of ρ_j, and such that there exist a homomorphism h_j from $\mathsf{body}(\rho_j)$ to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$, with $\bar{t}_j = h_j(\bar{x}_j) = h(\bar{x}_j)$, for each $j \in [n]$. We note that by definition of optimal decomposition, $\bar{t}_j$ necessarily mentions only constants, and since $i = 1$, ρ_j can only be a rule produced in line 20, whose body only mentions predicates from $\mathsf{sch}(\Sigma)$, i.e., ρ_j falls in **Case 1**, for which have already proved the claim. Hence, we conclude that for each $j \in [n]$, $\bar{t}_j \in \mathsf{ans}((\mathsf{refq}(\rho_j), \Sigma), D)$. By construction of ρ, and by definition of optimal decomposition, the fact that $\bar{t}_j \in \mathsf{ans}((\mathsf{refq}(\rho_j), \Sigma), D)$, for each $j \in [n]$ implies that $\bar{t} \in \mathsf{ans}((\mathsf{refq}(\rho), \Sigma), D)$ as needed.

Assume now that $i > 0$, and assume that $\bar{t} \in q_\rho(\mathsf{chase}^i(D, \Sigma \cup \mathcal{R}))$. Then, there exists a homomorphism h from $\mathsf{body}(q_\rho)$ to $\mathsf{chase}^i(D, \Sigma \cup \mathcal{R})$ with $h(\bar{x}) = \bar{t}$. We conclude that $R_j(h(\bar{x}_j)) \in \mathsf{chase}^i(D, \Sigma \cup \mathcal{R})$, for each $j \in [n]$. The latter means that there must be rules $\rho_1, \ldots, \rho_n$, with R_j the head predicate of ρ_j, and such that there exists a homomorphism h_j from $\mathsf{body}(\rho_j)$ to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$, with $\bar{t}_j = h_j(\bar{x}_j) = h(\bar{x}_j)$, for each $j \in [n]$. We observe again that $\bar{t}_j$ can only contain constants, for each $j \in [n]$, and thus, for each $j \in [n]$, if ρ_j falls in **Case 1**, we immediately conclude that $\bar{t}_j \in \mathsf{ans}((\mathsf{refq}(\rho_j), \Sigma), D)$,

while if ρ_j falls in **Case 2**, since h_j maps the body of ρ_j to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$, by inductive hypothesis, we also conclude that $\bar{t}_j \in \mathsf{ans}((\mathsf{refq}(\rho_j), \Sigma), D)$.

Thus, by definition of optimal decomposition, and of $\mathsf{refq}(\rho)$, we conclude that $\bar{t} \in \mathsf{ans}((\mathsf{refq}(\rho), \Sigma), D)$, and the claim follows. □

With the above lemma in place, we can now prove the soundness of the algorithm WardedRewrite.

Theorem 4. *Consider an OMQ $\mathcal{O} = (q, \Sigma)$, with Σ not necessarily warded. If the algorithm* WardedRewrite *terminates with input $\mathcal{O}$, producing the Datalog query $\mathcal{D}_\mathcal{O}$, then for every database D, it holds that:*

$$\mathcal{D}_\mathcal{O}(D) \subseteq \mathsf{ans}(\mathcal{O}, D).$$

Proof. Consider a database D, and a tuple $\bar{t} \in \mathsf{dom}(D)^k$, with k the arity of q. We prove that $\bar{t} \in \mathcal{D}_\mathcal{O}(D)$ implies $\bar{t} \in \mathsf{ans}(\mathcal{O}, D)$. Assume $\mathcal{D}_\mathcal{O} = (Q, \mathcal{R})$, where Q is the head predicate of q, and assume $\bar{t} \in \mathcal{D}_\mathcal{O}(D)$. Then, there must be a rule $\rho \in \mathcal{R}$ such that $\bar{t} \in \mathsf{ans}((\mathsf{freshcq}(\rho), \mathcal{R}), D)$. Thus, by monotonicity of certain answers, $\bar{t} \in \mathsf{ans}((\mathsf{freshcq}(\rho), \Sigma \cup \mathcal{R}), D)$. By Lemma 3, we conclude that $\bar{t} \in \mathsf{ans}((\mathsf{refq}(\rho), \Sigma), D)$. Moreover, since the head predicate of ρ is the head predicate of q, $\mathsf{refq}(\rho)$ coincides with q, up to variable renaming, and thus $\bar{t} \in \mathsf{ans}((q, \Sigma), D) = \mathsf{ans}(\mathcal{O}, D)$ as needed. □

8.3 Completeness of WardedRewrite

In this last section, we prove that the Datalog query $\mathcal{D}_\mathcal{O}$ produced by WardedRewrite, with input an OMQ $\mathcal{O} = (q, \Sigma)$, when evaluated over a database D, provides all certain answers of $\mathcal{O}$ over D. Towards this goal, we first establish the following auxiliary result.

Lemma 4. *Consider an OMQ $\mathcal{O} = (q, \Sigma)$, with $\mathsf{headPred}(q) = Q$, and such that* WardedRewrite *terminates with input $\mathcal{O}$, producing the Datalog query $\mathcal{D}_\mathcal{O} = (Q, \mathcal{R})$. Then, for every database D, and every predicate P occurring in the head of some rule of $\mathcal{R}$, the following holds:*

$$P(\bar{t}) \in \mathsf{chase}(D, \Sigma \cup \mathcal{R}) \text{ implies } P(\bar{t}) \in \mathsf{chase}(D, \mathcal{R}).$$

Proof. To prove the claim, we assume w.l.o.g. that Σ contains only TGDs having only one atom in the head; the set $\mathcal{R}$ already satisfies this property by construction. It

is well-known that for an ontology Σ', a TGD $\sigma \in \Sigma'$ with multiple atoms in the head can be replaced in Σ' with a linear number of TGDs with a single head atom, obtaining an ontology Σ'' such that $P(\bar{t}) \in \mathsf{chase}(D, \Sigma')$ iff $P(\bar{t}) \in \mathsf{chase}(D, \Sigma'')$, for every database D, and every predicate $P \in \mathsf{sch}(\Sigma')$. We prove the claim by showing that $P(\bar{t}) \in \mathsf{chase}^i(D, \Sigma \cup \mathcal{R})$ implies $P(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$, for each $i > 0$; the case $i = 0$ is trivial, since no atoms can occur in D with predicate P. The proof is by induction on i.

Assume $i = 1$. Then, there exists a rule $\sigma \in \mathcal{R}$, and a homomorphism h from $\mathsf{body}(\sigma)$ to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R}) = D$ with $h(\mathsf{head}(\sigma)) = P(\bar{t})$. Since $D \subseteq \mathsf{chase}(D, \mathcal{R})$, and $\sigma \in \mathcal{R}$, $P(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$ as needed.

Assume instead $i > 1$. Then, there exists a rule $\sigma \in \mathcal{R}$ such that σ is of the form $P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n) \rightarrow P(\bar{x})$, and a homomorphism h from $\mathsf{body}(\sigma)$ to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$ with $h(\bar{x}) = \bar{t}$. Note that, since $\sigma \in \mathcal{R}$, the predicates $P_1, \ldots, P_n$ either all belong to $\mathsf{sch}(\Sigma)$, or none of them does. We complete our proof by distinguishing these two cases.

Case 1. Assume that $P_j \notin \mathsf{sch}(\Sigma)$, for each $j \in [n]$. In this case, each P_j occurs in the head of some rule $\sigma_j \in \mathcal{R}$, and $P_j(h(\bar{x}_j)) \in \mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$. By inductive hypothesis, $P_j(h(\bar{x}_j)) \in \mathsf{chase}(D, \mathcal{R})$, for each $j \in [n]$, and thus h is also a homomorphism from $\mathsf{body}(\sigma)$ to $\mathsf{chase}(D, \mathcal{R})$ with $h(\bar{x}) = \bar{t}$, and the claim follows.

Case 2. Assume that $P_j \in \mathsf{sch}(\Sigma)$, for each $j \in [n]$. Since $P_j(h(\bar{x}_j)) \in \mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$, there must exist n TGDs $\sigma_1, \ldots, \sigma_n$, and n substitutions $h_1, \ldots, h_n$, such that for each $j \in [n]$, h_j maps each variable occurring in σ_j to a constant or null, $h_j(\mathsf{body}(\sigma_j)) \subseteq \mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$, and $h_j(\mathsf{head}(\sigma_j)) = P_j(h(\bar{x}_j))$ (recall that we assumed Σ contains only single-head TGDs). Now, assume w.l.o.g. that $P_1(h(\bar{x}_1)), \ldots, P_k(h(\bar{x}_k))$ are all the atoms in $h(\mathsf{body}(\sigma))$ occurring in $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R}) \setminus \mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$. We prove that $P(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$ by another induction on k.

Assume $k = 1$. Hence, $P_1(h(\bar{x}_1))$ is the only atom in $h(\mathsf{body}(\sigma))$ that occurs in $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R}) \setminus \mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$. By the existence of the homomorphism h_1 with the properties discussed above, by definition of chunk unifier, and from the fact that $P_1(h(\bar{x}_1))$ is the only atom in $h(\mathsf{body}(\sigma))$ occurring in $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R}) \setminus \mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$, there must be a σ_1-resolvent q' of $\mathsf{cq}(\sigma)$ via some MGCU M, and a homomorphism h' from $\mathsf{body}(q')$ to $\mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$ with $h'(\mathsf{head}(q')) = P(\bar{t})$. Moreover, since $\sigma \in \mathcal{R}$, by construction of WardedRewrite, q' must occur (modulo variable renaming) in the set $\mathcal{Q}_{\text{current}}$ at some iteration of WardedRewrite with input $\mathcal{O}$. We distinguish two cases w.r.t. q'. If q' does not admit an existential-join decomposition, then the fact that q' occurs (modulo variable renaming) in the set $\mathcal{Q}_{\text{current}}$ at some iteration of

WardedRewrite, must imply that $\mathsf{rule}(q')$ occurs in $\mathcal{R}$, modulo variable renaming. Thus, since $h'(\mathsf{body}(q')) \subseteq \mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$ and $h'(\mathsf{head}(q')) = P(\bar{t})$, we conclude that $P(\bar{t}) \in \mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$. Hence, by inductive hypothesis on i, we conclude that $P(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$ as needed. If instead q' admits an existential join decomposition, since q' occurs in $\mathcal{Q}_{\text{current}}$ (modulo variable renaming) at some iteration of WardedRewrite with input $\mathcal{O}$, $\mathcal{R}$ must also contain the modified version of the reconciliation rule of $\mathcal{S}$ and q', where $\mathcal{S}$ is an optimal decomposition of q' w.r.t. Σ, together with all the root rules of each query in $\mathcal{S}$. More formally, there must be a rule ρ in $\mathcal{R}$ of the form

$$P'_1(\bar{y}_1), \ldots, P'_m(\bar{y}_m) \rightarrow \mathsf{head}(q'),$$

where for each predicate P'_ℓ, with $\ell \in [m]$, a rule ρ_ℓ with P'_ℓ in its head occurs in $\mathcal{R}$, and $\mathsf{cq}(\rho_1), \ldots, \mathsf{cq}(\rho_m)$ form an existential-join decomposition of q' (modulo head predicate renaming). By definition of existential-join decomposition, and by the existence of the homomorphism h' from $\mathsf{body}(q')$ to $\mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$, there must be a homomorphism h'' from $\mathsf{body}(\rho)$ to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$ with $h''(\mathsf{head}(\rho)) = P(\bar{t})$. Since each predicate $P'_1, \ldots, P'_m$ in the body of ρ occurs in the head of a rule in $\mathcal{R}$, and since $h''(\mathsf{body}(\rho)) \subseteq \mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$, the inductive hypothesis on i implies that $h''(\mathsf{body}(\rho)) \subseteq \mathsf{chase}(D, \mathcal{R})$. Hence, h'' is also a homomorphism from $\mathsf{body}(\rho)$ to $\mathsf{chase}(D, \mathcal{R})$ with $h''(\mathsf{head}(\rho)) = P(\bar{t})$, and since $\rho \in \mathcal{R}$, we conclude that $P(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$.

Assume instead that $k > 1$. Consider again the atom $P(h_1(\bar{x}_1)) \in \mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R}) \setminus \mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$. As for the base case with $k = 1$, the existence of h_1 implies that there exists a σ_1-resolvent q' of $\mathsf{cq}(\sigma)$ via some MGCU M. Moreover, q' must be such that there is a homomorphism h' from $\mathsf{body}(q')$ to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$ with $h'(\mathsf{head}(q')) = P(\bar{t})$, and $h'(\mathsf{body}(q'))$ contains at most $k-1$ atoms from $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R}) \setminus \mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$.

Moreover, since $\sigma \in \mathcal{R}$, by construction of WardedRewrite, q' must occur (modulo variable renaming) in the set $\mathcal{Q}_{\text{current}}$ at some iteration of WardedRewrite with input $\mathcal{O}$. We distinguish two cases w.r.t. q'. If q' does not admit an existential join decomposition, then the fact that q' occurs (modulo variable renaming) in the set $\mathcal{Q}_{\text{current}}$ at some iteration of WardedRewrite implies that $\mathsf{rule}(q')$ occurs in $\mathcal{R}$, modulo variable renaming. However, we have that $h'(\mathsf{body}(q'))$ contains at most $k-1$ atoms from $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R}) \setminus \mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$. The latter, together with the fact that $h'(\mathsf{head}(q')) = P(\bar{t})$, and by induction on k, implies that $P(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$ as needed. If instead q' admits an existential join decomposition, then the fact that q' occurs in $\mathcal{Q}_{\text{current}}$ (modulo variable

renaming) at some iteration of WardedRewrite with input $\mathcal{O}$ implies that $\mathcal{R}$ must also contain the modified version of the reconciliation rule of $\mathcal{S}$ and q', where $\mathcal{S}$ is an optimal decomposition of q' w.r.t. Σ, together with all the root rules of each query in $\mathcal{S}$. More formally, there must be a rule ρ in $\mathcal{R}$ of the form

$$P'_1(\bar{y}_1), \ldots, P'_m(\bar{y}_m) \rightarrow \mathsf{head}(q'),$$

where $\mathsf{refq}(P'_1), \ldots, \mathsf{refq}(P'_m)$ all occur in $\mathcal{R}$, and form an existential-join decomposition of q' (modulo head predicate renaming).

By definition of existential-join decomposition, and by the existence of the homomorphism h' from $\mathsf{body}(q')$ to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$, there must be a homomorphism h'' from $\mathsf{body}(\rho)$ to $\mathsf{chase}^{i}(D, \Sigma \cup \mathcal{R})$ with $h''(\mathsf{head}(\rho)) = P(\bar{t})$. We now observe that the atoms $P'_1(h''(\bar{y}_1)), \ldots, P'_m(h''(\bar{y}_m))$ all belong to $\mathsf{chase}(D, \mathcal{R})$. Indeed, for each $\ell \in [m]$, if $P'_\ell(h''(\bar{y}_\ell)) \in \mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$, $P'_\ell(h''(\bar{y}_\ell)) \in \mathsf{chase}(D, \mathcal{R})$ by the inductive hypothesis on i. If instead $P'_\ell(h''(\bar{y}_\ell)) \in \mathsf{chase}^{i}(D, \Sigma \cup \mathcal{R})$, due to the existence of the homomorphism h', then the body of $\mathsf{refq}(P'_\ell)$ can be mapped homomorphically to $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R})$, with some homomorphism μ, such that $\mu(\mathsf{head}(\mathsf{refq}(P'_\ell))) = P'_\ell(h''(\bar{y}_\ell))$, and $\mu(\mathsf{body}(\mathsf{refq}(P'_\ell))) \subseteq h'(\mathsf{body}(q'))$. Since $h'(\mathsf{body}(q'))$ contains no more than $k-1$ atoms from $\mathsf{chase}^{i-1}(D, \Sigma \cup \mathcal{R}) \setminus \mathsf{chase}^{i-2}(D, \Sigma \cup \mathcal{R})$, so does $\mu(\mathsf{body}(\mathsf{refq}(P'_\ell)))$. Hence, by inductive hypothesis on k, we conclude that $P'_\ell(h''(\bar{y}_\ell)) \in \mathsf{chase}(D, \mathcal{R})$.

Hence, h'' is a homomorphism from $\mathsf{body}(\rho)$ to $\mathsf{chase}(D, \mathcal{R})$, with $h''(\mathsf{head}(q')) = P(\bar{t})$, and thus $P(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$ as needed. □

With the above lemma in place, we can now prove the main claim of this section.

Theorem 5. *Consider an OMQ $\mathcal{O} = (q, \Sigma)$, with Σ not necessarily warded. If the algorithm* WardedRewrite *terminates with input $\mathcal{O}$, producing the Datalog query $\mathcal{D}_\mathcal{O}$, then for every database D, it holds that:*

$$\mathsf{ans}(\mathcal{O}, D) \subseteq \mathcal{D}_\mathcal{O}(D).$$

Proof. Assume that $\mathsf{headPred}(q) = Q$, and that $\mathcal{D}_\mathcal{O} = (Q, \mathcal{R})$. We prove the claim by showing that $Q(\bar{t}) \in \mathsf{chase}(D, \Sigma \cup \{\mathsf{rule}(q)\})$ implies $Q(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$. Assume $Q(\bar{t}) \in \mathsf{chase}(D, \Sigma \cup \{\mathsf{rule}(q)\})$. We distinguish two cases.

Case 1. The rule $\mathsf{rule}(q)$ occurs in $\mathcal{R}$, modulo variable renaming. Thus, by monotonicity of the chase over TGDs, $\mathsf{chase}(D, \Sigma \cup \{\mathsf{rule}(q)\}) \subseteq \mathsf{chase}(D, \Sigma \cup \mathcal{R})$. By Lemma 4, we conclude that $Q(\bar{t}) \in \mathsf{chase}(D, \Sigma \cup \{\mathsf{rule}(q)\})$ implies $Q(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$ as needed.

Case 2. The rule $\mathsf{rule}(q)$ does not occur in $\mathcal{R}$, even up to variable renaming. If this is the case, then, by definition of WardedRewrite, it must mean that q admits an existential-join decomposition w.r.t. Σ. Assume $\mathcal{S} = \{q_1, \ldots, q_n\}$ is the optimal decomposition of q w.r.t. Σ. Then, $\mathcal{R}$ contains the reconciliation rule ρ of $\mathcal{S}$ and q, as well as the rules $\mathsf{rule}(q_i)$, for $i \in [n]$, up to variable renaming. Since $Q(\bar{t}) \in \mathsf{chase}(D, \Sigma \cup \{\mathsf{rule}(q)\})$, then, there exists a homomorphism h from $\mathsf{body}(q)$ to $\mathsf{chase}(D, \Sigma)$ with $h(\mathsf{head}(q)) = Q(\bar{t})$. By definition of optimal decomposition, the existence of h implies there for each $i \in [n]$, there must be a homomorphism h_i from $\mathsf{body}(q_i)$ to $\mathsf{chase}(D, \Sigma)$ with $h_i(\mathsf{head}(q_i))) = P_i(\bar{t}_i) \in \mathsf{chase}(D, \Sigma \cup \mathcal{R})$, such that another homomorphism h' from $\mathsf{body}(\rho)$ to $\{P_1(\bar{t}_1), \ldots, P_n(\bar{t}_n)\} \subseteq \mathsf{chase}(D, \Sigma \cup \mathcal{R})$ exists with $h'(\mathsf{head}(\rho)) = Q(\bar{t}) \in \mathsf{chase}(D, \Sigma \cup \mathcal{R})$. By Lemma 4, we conclude that $Q(\bar{t}) \in \mathsf{chase}(D, \mathcal{R})$, as needed. □

Our main result on the correctness of WardedRewrite, i.e., Theorem 2, simply follows from Theorems 3, 4, and 5.

Chapter 9

Implementation Details

In this chapter we discuss how we concretely implemented the algorithm WardedRewrite. We point out that the general structure of the algorithm remains the same, and we simply discuss how certain parts of the algorithm are optimized. In particular, we focus on the implementation of the most expensive aspects of the algorithm: *1)* the procedure decomposeOptimal; *2)* the construction of an MGCU M of a given CQ q with a given TGD σ as well as the construction of the σ-resolvent of q via M; *3)* reducing the size of the set $\mathcal{R}$ of rules of the final rewriting during the execution of the algorithm. We observe that item *3)* does not actually appear in the algorithm, as it is not crucial for its correctness.

The rest of the algorithm, such as constructing the reconciliation rule of a decomposition $\mathcal{S}$ and a CQ q, or computing the canonical version of a CQ, is trivial to implement.

9.1 Computing Optimal Decompositions

We start by discussing how we implement the procedure decomposeOptimal. That is, given a CQ $q(\bar{x})$ and an ontology Σ, compute an optimal decomposition $\mathcal{S}$ of q w.r.t. Σ. We recall that in order to compute an optimal decomposition, we first need to compute the set $\mathsf{aff}(\Sigma)$ of all affected positions of Σ. Since the procedure decomposeOptimal is executed at each iteration of the algorithm WardedRewrite, computing the set $\mathsf{aff}(\Sigma)$ each time decomposeOptimal is called would be very inefficient. Thus, our implementation of WardedRewrite computes the set $\mathsf{aff}(\Sigma)$ once, at the beginning, and makes it globally available to the rest of the code. Then, the procedure decomposeOptimal is implemented by accessing the globally available set $\mathsf{aff}(\Sigma)$.

The set $\mathsf{aff}(\Sigma)$ is computed by simply following the definition of affected position in

a bottom-up fashion: first a set S is initialized as the empty set $\emptyset$. Then, in an initial pass, each TGD σ of Σ is scanned, and each head position of σ containing an existential variable is added to S. Then, in a second pass, each TGD σ of Σ is scanned, and all head positions of σ containing a variable that occurs in $\mathsf{body}(\sigma)$ only at positions of S are collected in a temporary set S'. If after the second pass, S' is not empty, the positions of S' are added to S, and the second pass is executed again; otherwise, the procedure stops, and S contains the set of all affected positions $\mathsf{aff}(\Sigma)$.

Assume now a CQ $q(\bar{x})$, and an ontology Σ are given. We discuss how we implement the construction of an optimal decomposition of q w.r.t. Σ, exploiting the already computed set $\mathsf{aff}(\Sigma)$. The implementation is reported in Algorithm 2. Assuming q is of the form $Q(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$, the idea is to initially assume that q can be optimally decomposed into n queries, each having a single atom from $\mathsf{body}(q)$ as their body; such bodies are stored in the set $\mathcal{C}$ (line 1). Then, the set $\mathcal{V}$ of all "problematic" variables is computed (line 2), i.e., all the variables that cannot be mentioned by two different queries of the decomposition (cf. Definition 5). Then, for each variable $x \in \mathcal{V}$, the set $\mathcal{C}$ is "fixed" by merging together the CQ bodies of $\mathcal{C}$ that all mention the variable x; the set $\mathcal{C}$ is updated by removing all CQ bodies that have been merged, and by adding the new merged body (lines 3-6). After all variables of $\mathcal{V}$ are considered, no two distinct sets in $\mathcal{C}$ mention a variable of $\mathcal{V}$, and each set in $\mathcal{C}$ corresponds to the body of a CQ of the final decomposition. The rest of the procedure (lines 9-12) is in charge of constructing the actual CQs corresponding to each set in $\mathcal{C}$, by computing the correct head atom, according to Definition 5. The set $\mathcal{S}$ will contain the final decomposition, which is returned at the end of the procedure. It is not difficult to verify that the returned set is always an optimal decomposition of q w.r.t. Σ.

Remark. We point out that an implementation of the procedure decomposeOptimal already exists in the literature, and has been employed in the context of parallelizing rewriting algorithms for UCQ-rewritable OMQs, e.g., see [41]. However, in this case, an optimal decomposition needed to be constructed only once throughout the execution of the whole UCQ-rewriting algorithm, and thus the efficiency of the implementation was not a concern. In our case, however, an optimal decomposition must be constructed quite often: once for each query processed by the algorithm WardedRewrite. Hence, we required a much more efficient implementation that, from preliminary experiments, has shown to be orders of magnitude faster than the one from [41].

Algorithm 2: decomposeOptimal

Input: A CQ q of the form $Q(\bar{x}) \leftarrow R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$, and an ontology Σ
Output: An optimal decomposition of q w.r.t. Σ

$\mathcal{C} := \{\{R_1(\bar{x}_1)\}, \ldots, \{R_n(\bar{x}_n)\}\}$;
$\mathcal{V} = \{x \in \mathsf{var}(q) \setminus \bar{x} \mid x \text{ occurs in } \mathsf{body}(q) \text{ only at positions of } \mathsf{aff}(\Sigma)\}$;
// Compute the body of each CQ of the optimal decomposition
foreach $x \in \mathcal{V}$ **do**
    $\mathcal{C}_x := \{C \in \mathcal{C} \mid x \in \mathsf{var}(C)\}$;
    $C_{\text{merged}} := \bigcup_{C \in \mathcal{C}_x} C$;
    $\mathcal{C} := (\mathcal{C} \setminus \mathcal{C}_x) \cup \{C_{\text{merged}}\}$;
// Finalize the decomposition
$\mathcal{S} := \emptyset$;
$i := 1$;
foreach $C \in \mathcal{C}$ **do**
    $\bar{y} = \mathsf{var}(C) \cap (\bar{x} \cup \mathsf{shared}(C, \mathcal{C}))$;
    $\mathcal{S} := \mathcal{S} \cup \{Q_i(\bar{y}) \leftarrow C\}$;
    $i := i + 1$;
return $\mathcal{S}$;

9.2 Constructing MGCUs

We now discuss how we construct, given a CQ q and TGD σ, all MGCUs of q with σ. As already discussed in previous sections, the notion of (most general) chunk unifier (and equivalent notions thereof) is well-established in the literature, and has been extensively used and implemented for a plethora of scenarios, mostly related to the design and implementation of rewriting algorithms for UCQ-rewritable OMQs, e.g., see [69, 52, 41]. Hence, different efficient implementations already exist for the construction of MGCUs in the literature, and we decided to employ the implementation of [41]. The reason is mostly of practical nature, as the implementation is part of a larger library[1] of convenient algorithms and utilities for dealing with the syntax of ontologies that we exploit for the rest of the implementation of WardedRewrite, such as parsing ontologies from files, computing the canonical form of a CQ, computing reconciliation rules, and other similar simple tasks.

[1] https://bitbucket.org/giorsi/nyaya/

9.3 Reducing the Size of the Final Rewriting

The ultimate goal of constructing a Datalog rewriting $\mathcal{D}_\mathcal{O}$ for an OMQ $\mathcal{O} = (q, \Sigma)$ is to provide a mean to perform query answering. That is, given a database D, we can retrieve the certain answers $\mathsf{ans}(\mathcal{O}, D)$ by simply evaluating the query $\mathcal{D}_\mathcal{O}$ over D, i.e., computing the set $\mathcal{D}_\mathcal{O}(D)$. The latter can be achieved by exploiting well-established and well-optimized Datalog engines from the literature. However, for this approach to be effective, the resulting Datalog query $\mathcal{D}_\mathcal{O}$ should contain as few rules as possible, in order to not hinder the performance of query answering. In this regard, techniques such as the one introduced in [63], which aims at minimizing Datalog programs by eliminating redundant rules and atoms while preserving equivalence, can be employed to reduce the size of the final rewriting and improve efficiency. Currently, WardedRewrite simply collects all produced rules of the Datalog rewriting in the set $\mathcal{R}$, without further optimization. In this section, we discuss how to improve WardedRewrite so to keep the set $\mathcal{R}$ small.

We recall that given two Datalog queries $\mathcal{D}_1$ and $\mathcal{D}_2$, $\mathcal{D}_1$ is *contained* in $\mathcal{D}_2$ if for every database D, $\mathcal{D}_1(D) \subseteq \mathcal{D}_2(D)$. Moreover, $\mathcal{D}_1$ and $\mathcal{D}_2$ are *equivalent* if $\mathcal{D}_1$ is contained in $\mathcal{D}_2$, and $\mathcal{D}_2$ is contained in $\mathcal{D}_1$. Clearly, when $\mathcal{D}_1$ and $\mathcal{D}_2$ are equivalent, they return the exact same set of answers, for every database.

Thus, ideally, we would like to extend WardedRewrite so that whenever a new rule is added to the set $\mathcal{R}$ (lines 14 and 20 in Algorithm 1), then rules from $\mathcal{R}$ are removed so as to obtain the smallest subset $\mathcal{R}'$ of $\mathcal{R}$ such that $(\mathsf{headPred}(q), \mathcal{R}')$ and $(\mathsf{headPred}(q), \mathcal{R})$ are equivalent. However, this requires checking whether two Datalog queries are equivalent, which is known to be undecidable, e.g., see [64]. Hence, we rely on a simpler, yet effective, way to reduce the size of the set $\mathcal{R}$ which we describe next.

Rather than trying to reduce the size of $\mathcal{R}$ as a whole, we first consider all the 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 is performed by checking for equivalence between UCQs, which is decidable (e.g., see [1]). Clearly, replacing in $\mathcal{R}$ the set of rules $\mathcal{R}_{Q_i}$ with the set of rules $\mathcal{R}'_{Q_i}$, for each $i \in [n]$, obtaining the set of rules $\mathcal{R}'$ leads to an equivalent Datalog query, i.e., $(\mathsf{headPred}(q), \mathcal{R}')$ and $(\mathsf{headPred}(q), \mathcal{R})$ are equivalent. For the equivalence check between UCQs we again rely on the implementation provided by the library developed in [41].

Chapter 10

Experimental Evaluation

The goal of this chapter 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 question will allow us to provide insights on the practical applicability of our implementation and its limits, as well as providing conclusions on the expected behavior 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 feasible is for the different Datalog-based reasoning engines from the literature to evaluate the query $\mathcal{D}_\mathcal{O}$ over D?*

In the rest, Section 10.1 discusses the tools that we used to generate our benchmarks, while Section 10.2 describes how our benchmark is generated using the tools discussed in the previous section, and presents our experimental setup. Then, in Sections 10.3 and 10.4 we discuss the experimental evaluations regarding Question 1 and Question 2, respectively.

10.1 Experimental Infrastructure

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 require a way to generate a large pool of synthetic database-OMQ pairs $(D, \mathcal{O})$, where $\mathcal{O} = (q, \Sigma)$ and $\Sigma \in \mathsf{WARDED}$, having different properties that affect the behaviour of our rewriting algorithm and the reasoning engines; we call a database-OMQ pair a *scenario*. To the best of our knowledge, the only existing tool from the literature that allows to generate such scenarios, and specifically scenarios $(D, \mathcal{O})$ where the ontology of $\mathcal{O}$ is warded is the iWarded benchmark generator, or simply iWarded, from [6]. Such a generator allows to generate scenarios $(D, \mathcal{O})$, with $\mathcal{O} = (q, \Sigma)$ and $\Sigma \in \mathsf{WARDED}$, by considering a wide range of parameters. In order to explain how iWarded works and the meaning of such parameters, we first need to recall some auxiliary notions.

For an ontology Σ, the *predicate graph of* Σ, denoted $\mathsf{pg}(\Sigma)$, is the directed graph (V, E) where the set of nodes V coincides with the set of predicates occurring in Σ, and E contains an edge (R, S) iff there exists a TGD $\sigma \in \Sigma$ such that R occurs in the body of σ, and S occurs in the head of σ. We say that a predicate occurring in Σ is *recursive* iff it appears in a cycle of $\mathsf{pg}(\Sigma)$. We are now ready to recall how the iWarded benchmark generator works.

Overview of iWarded. When generating a scenario $(D, \mathcal{O})$ with $\mathcal{O} = (q, \Sigma)$, the bulk of the work of iWarded is in the construction of the ontology Σ; the database D and the query q are generated in a simple way, which we are going to discuss later. iWarded controls the structure of Σ by acting on the form of all the simple non-overlapping paths of its predicate graph, each corresponding to a sequence of TGDs $\sigma_1, \ldots, \sigma_n$, where the atom produced during the construction of $\mathsf{chase}(D, \Sigma)$ by the TGD σ_i contributes to trigger the TGD σ_{i+1}, for each $i \in [n-1]$. Such sequences are generated according to different parameters that affect different aspects of the sequences, e.g., whether the sequences induce a cycle in the dependency graph, and the length of this cycle.

Due to the large number of parameters supported by iWarded, and in order to keep our experimental evaluation manageable, we will focus only on few, key parameters that mostly affect either the size of the database D, or the sequences of TGDs discussed above in a way that they have a higher impact on the efficiency of our rewriting algorithm WardedRewrite. All other parameters were conceived to mainly affect the efficiency of specialized algorithms designed to compute the certain answers of OMQs with warded

ontologies *directly*, i.e., without relying on a Datalog rewriting (e.g., see the "structural scenarios" from [6]) and thus are less relevant for our analysis. For such parameters, we let iWARDED adapt their value so to satisfy the desiderata w.r.t. the parameters we consider.

We point out that, regardless of the chosen parameters, iWARDED always constructs TGDs with at most 2 atoms in their body and one atom in the head; this is done w.l.o.g. since an arbitrary ontology can always be rewritten to an ontology having these properties, and with only a linear increase in the size of the ontology.

In the following, we discuss the main parameters we consider and then explain how we use iWARDED to generate scenarios according to those parameters; we categorize the parameters on whether they affect the generated database D, the CQ q, or the ontology Σ.

Ontology Parameters. Regarding the parameters supported by iWARDED that affect the shape of the generated ontology Σ, and in particular the shape of the sequences of its TGDs, we focus on two crucial parameters.

The *number of left-right-join recursions*, denoted `num_lrj`, specifies the number of TGDs that iWARDED will add to Σ that mention two recursive predicates in their body. Such a parameter essentially affects how many TGDs in Σ participate to multiple recursive sequences of TGDs. Intuitively, such TGDs are more challenging than other TGDs of Σ, since in order to trigger such TGDs, one first needs to follow two recursive sequences of TGDs in order to produce the needed body atoms.

The *average recursion length*, denoted `avg_recursion`, specifies the average length of a cycle in the predicate graph of Σ. In particular, for each recursive predicate R of Σ, iWARDED will construct a sequence of n TGDs $\sigma_1, \ldots, \sigma_n$ in Σ, where n is chosen using a Gaussian distribution with `avg_recursion` as the mean, and variance 1.0, and such that the sequence induces a path in the dependency graph of Σ starting from R and ending in R. In other words, the parameter `avg_recursion` affects "how deep" one has to follow the TGDs of Σ in order to recursively produce atoms with predicate R.

Database Parameters. As for the database-related parameters, we focus on the *number of facts per predicate*, denoted `db_facts`: iWARDED will add `db_facts` randomly generated facts to the database D for each predicate of D; the number of predicates of D is chosen automatically by iWARDED so to guarantee the desiderata w.r.t. the ontology-related parameters discussed above. Clearly, the parameter `db_facts` is not relevant towards answering our first experimental question, since it has no effect on

the behavior of the rewriting algorithm WardedRewrite, which constructs a *database-independent* Datalog rewriting of a given OMQ. On the other hand, such a parameter, and thus the generated database D, will have an impact when we will focus on our second experimental question regarding query answering using existing Datalog-based reasoning engines.

Query Parameters. There are no parameters that 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, and that there are no parts of Σ that do not affect the answers of q.

Although the CQs constructed by iWarded are of a limited form, we point out that these kinds of queries are very common in practice, especially in the context of the so-called fact entailment problem, i.e., a special version of the certain query answering problem where the goal, given a predicate R, a database D, and an ontology Σ, is to find all facts $R(\bar{t}) \in \mathsf{chase}(D, \Sigma)$. Clearly, for an OMQ $\mathcal{O} = (q, \Sigma)$ having q of the form above, the certain answers of $\mathcal{O}$ over a database D coincide with all tuples of constants $\bar{t}$ such that $R(\bar{t}) \in \mathsf{chase}(D, \Sigma)$. Indeed, different works have focused on the fact entailment problem, both from a theoretical, and an experimental point of view, and for different ontology languages, e.g., see [19, 17]. In fact, among the reasoning engines we are going to consider in our experimental evaluation, Vadalog exclusively supports fact entailment as its main reasoning task. Hence, for practical reasons, and in order to be able to consider a wide variety of reasoning engines from the literature, in our experimental evaluation we will focus on CQs as generated by iWarded.

10.2 Test Scenarios and Experimental Setup

In this section we discuss how we generated our main benchmark using iWarded and provide additional details on our experimental setup.

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 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 stress 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; all other parameters supported by iWarded are left to their default as discussed in the previous section.

Experimental Setup. Regarding our experimental setup, for all our experiments we used an Amazon Elastic Compute Cloud (EC2) instance type c5.2xlarge with an Intel(R) Xeon(TM) Platinum 8000-series @3.6 GHz, 16 GB of RAM, running Amazon Linux 2 LTS Candidate.[1] Each scenario $(D, \mathcal{O})$ generated by iWarded is stored on disk as simple text files, where the database is encoded as a set of CSV files, for compatibility with the different Datalog-based engines we consider. For the experimental evaluation related to Question 1, we implemented the algorithm WardedRewrite in Java 22, and used openJDK 22 for its execution. Regarding Question 2, each Datalog-based reasoning engine we considered has been executed in its default configuration.

10.3 Evaluating the Rewriting Algorithm

In this section, we address our first experimental question, related to the behavior of our implementation of the WardedRewrite algorithm. 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 end-to-end running time of the algorithm, i.e., the total time required by the algorithm to both parse the OMQ and produce the Datalog rewriting $\mathcal{D}_\mathcal{O}$ of $\mathcal{O}$. We point out that the time of parsing the OMQ never required more than 250 ms, and thus, as expected, most of the time is spent actually constructing the Datalog rewriting. We present the results of the above experimental evaluation in Figure 10.1, Figure 10.2, Figure 10.3 and Figure 10.4, where Figure 10.1 and Figure 10.2 focus on the impact that `num_lrj` has on the efficiency of our implementation, while Figure 10.3 and Figure 10.4 focus on the relationship with the parameter `avg_recursion`.

Impact of the Number of Left-Right-Join Recursions. We start by discussing how the parameter `num_lrj` affects the running time of our implementation, i.e., Figure 10.1 and Figure 10.2. In particular, Figure 10.1 shows how the end-to-end run-

[1] `https://aws.amazon.com/ec2/instance-types/c5/`

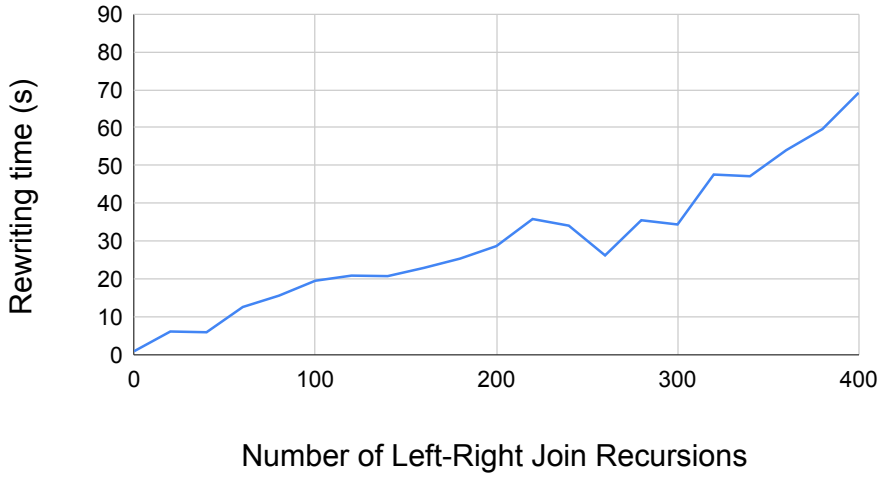

Figure 10.1: Average rewriting time versus number of left-right join recursions (global averages).

ning time of our implementation of WardedRewrite evolves as the `num_lrj` parameter increases; each point on the plot corresponding to a value $i \in \{0, 20, 40, \ldots, 400\}$ of `num_lrj` is computed as the average of the running times collected in all scenarios of the from $S[i, j]$, with $j \in \{0, 20, 40, \ldots, 400\}$, i.e., across all values $j \in \{0, 20, 40, \ldots, 400\}$ for the parameter `avg_recursion`. For a more fine-grained analysis, we also partitioned our scenarios in 4 groups, denoted $AR_{[0-100]}$, $AR_{[100-200]}$, $AR_{[200-300]}$, $AR_{[300-400]}$, according to the value of `avg_recursion`, i.e., for each $k \in \{[0, 100], [100, 200], [200, 300], [300, 400]\}$, AR_k contains all scenarios of the form $S[i, j]$ where j belongs to the interval k. Figure 10.2 shows the end-to-end running time of our implementation of WardedRewrite w.r.t. the `num_lrj` parameter, where each line corresponds to one of the 4 groups AR_k, and each point on each line that corresponds to a value $i \in \{0, 20, 40, \ldots, 400\}$ of `num_lrj` is computed as the average running time across all scenarios of the form $S[i, j] \in AR_k$.

As we can see from Figure 10.1, the number of left-right-join recursions indeed has an impact on the running time of our implementation of WardedRewrite, with the higher values of `num_lrj` leading to an average running time of just over a minute. The trend can be explained by the fact that every time a TGD of the input OMQ is used to compute an MGCU w.r.t. a CQ being processed by WardedRewrite, this leads to a new CQ containing at least two atoms, each over a recursive predicate, e.g., say R and S. The latter means that both such atoms will need to be unfolded in subsequent iterations of the algorithm, and that subsequent unfoldings of both atoms are likely to lead to

Figure 10.2: Average rewriting time versus number of left-right join recursions, grouped by `avg_recursion` groups.

a query having again two atoms over the same predicates R and S, which is likely to trigger again further unfoldings. 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. The latter can be explained from the fact that, assuming a fixed average recursion length, increasing the number of TGDs with a left-right-join recursion means essentially increasing the number of "disjoint" TGD sequences in the underlying ontology, i.e., sequences of TGDs leading to disconnected cycles in the underlying predicate graph. Hence, the running time of WardedRewrite linearly depends on the number of such sequences, since they do not interact with each other. This is confirmed by Figure 10.2, where we plot the running time as `num_lrj` increases, by considering different intervals for the average recursion length `avg_recursion`. 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 lower average recursion length interval [0-100], and 33 seconds for [100-200].

Impact of the Average Recursion Length. We now focus on the parameter `avg_recursion`. The results of running our implementation of WardedRewrite are shown in Figure 10.3 and Figure 10.4. In particular, Figure 10.3 and Figure 10.4 are obtained similarly to Figure 10.1 and Figure 10.2, with the difference that we now vary the parameter `avg_recursion`, and compute the average running time across all (resp., 4 groups of) values of the parameter `num_lrj`. We observe from Figure 10.1 that also the

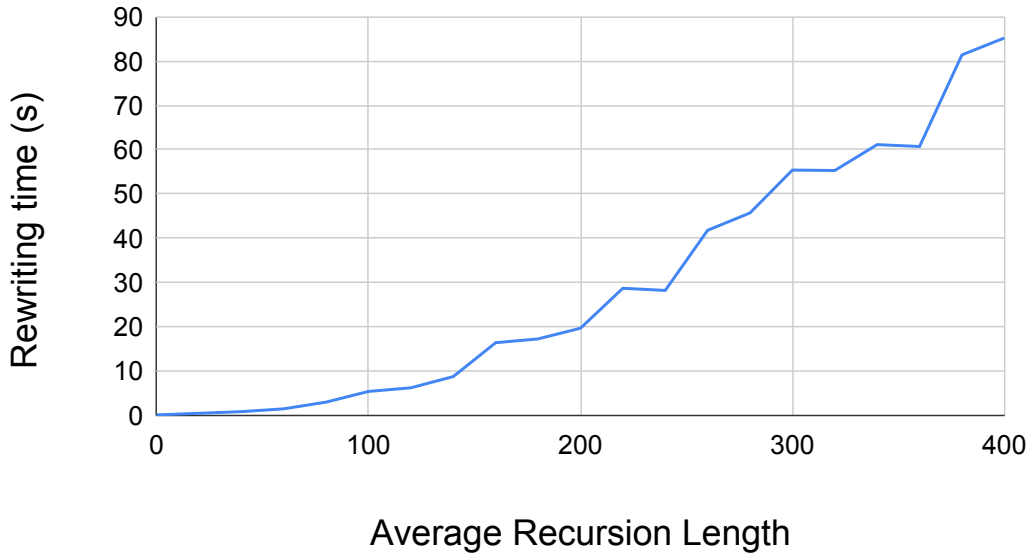

Figure 10.3: Average rewriting time versus average recursion length (global averages).

average recursion length affects the running time of our implementation, on average. However, differently from the case of the number of left-right-join recursions, here we see a steeper increase in the running time on average, as the parameter `avg_recursion` increases, with the highest average running time at almost 1.5 minutes, corresponding to `avg_recursion` = 400. Hence, the average recursion length has a higher impact on the efficiency of our implementation; we can explain this behavior as follows. Assuming we fix a certain value k for the parameter `num_lrj`, the higher the parameter `avg_recursion`, the higher the average length of a cycle in the predicate graph of the ontology. The latter implies that, when some query q is being processed by the algorithm by means of a resolution step, this is likely to produce a new CQ that will require a large number of further resolution steps, and this number is proportional to `avg_recursion`. Moreover, depending on the value of k, in each of these resolution steps, there is a certain chance that a new CQ q' with two body atoms, both having a recursive predicate, is produced. Each time this happens, at the next iteration either q' will be decomposed into at least two queries, each containing one of the two atoms, for which later all MGCUs need to be computed in a resolution step, or q' (or one of the subqueries of its decomposition) contains the two atoms in its body, and will go through a new resolution step, which will lead to the construction of all MGCUs w.r.t. each TGD of the ontology. In both cases, the number of queries produced by the algorithm that originated directly or indirectly from q' is at least *twice* w.r.t. to the

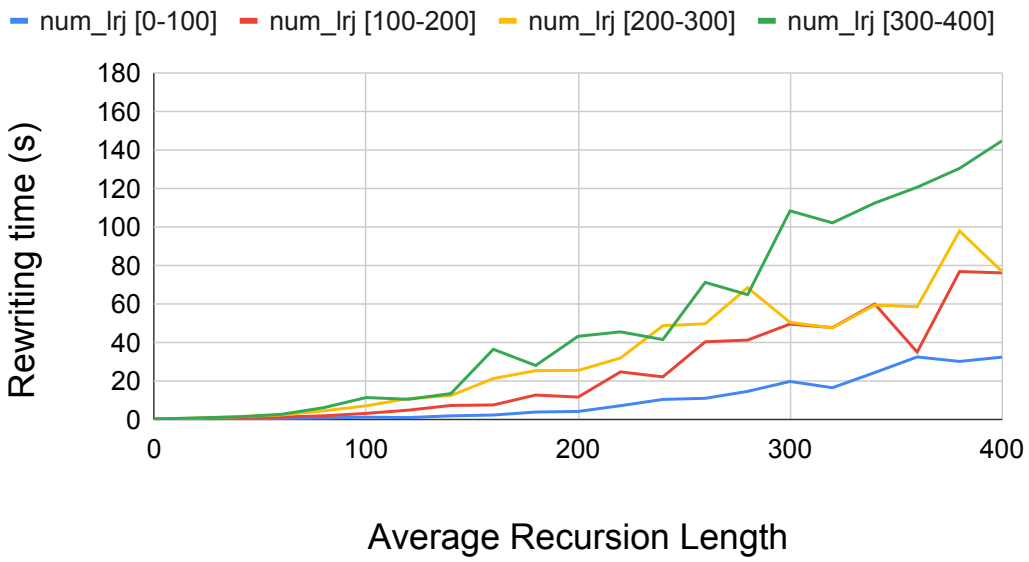

Figure 10.4: Average rewriting time versus average recursion length, grouped by `num_lrj` groups.

case whether q' did not contain two recursive atoms; the latter then causes the steep increase observed in the plot.

We can confirm this behavior with the more detailed plot in Figure 10.4, where we group the average running time w.r.t. 4 different intervals of the parameter `num_lrj`, similarly to what we have done in Figure 10.2 when analyzing the impact of `num_lrj`.

We conclude this section by observing that in terms of absolute running times, the most demanding scenario is $S[400, 400]$, as expected, requiring almost 4 minutes for building the rewriting. Considering that $S[400, 400]$ is a rather extreme scenario that would hardly appear in practice, this is quite encouraging. In comparison, for more reasonable scenarios, such as the ones in $\{S[i, j] \mid i, j \in \{0, 20, 40, \ldots, 140\}\}$, the rewriting time is 5 seconds in the worst case.

Validation with Other Benchmarks. To place our analysis above in perspective, we also executed our implementation of WardedRewrite on a set of 8 OMQs from the literature that use warded ontologies. In particular, we considered the set of 8 scenarios, i.e., database-OMQ pairs, called *structural scenarios*, that have been used to stress test the Datalog-based reasoning engine Vadalog [6]. The scenarios are named as synthX, with $\mathsf{X} \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}\}$, each having a database with $10k$ facts per predicate, and an OMQ with a warded ontology with different characteristics, meant to stress test different aspects of a Datalog-based reasoning engine; we are not aware

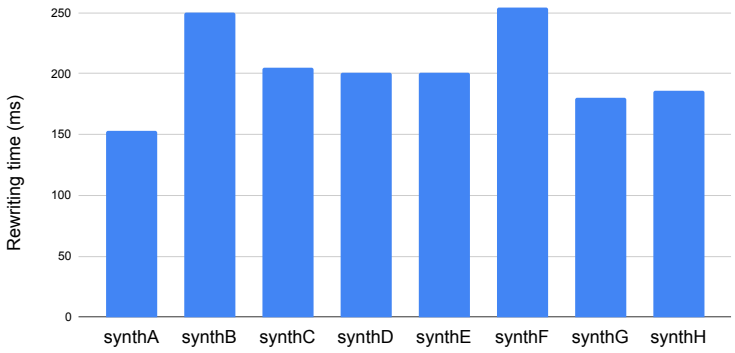

Figure 10.5: Rewriting time over structural scenarios.

of other existing benchmarks collecting OMQs whose ontologies are warded.[2] We executed WardedRewrite over the OMQ of each structural scenario and the running times are reported in Figure 10.5; the time is measured in milliseconds. As we can see, for the structural scenarios, the Datalog rewriting can be constructed very efficiently, with the worst case requiring about 250 ms.

Remark: We point out that warded is a relatively new TGD-based language, and thus very few real-world OMQs which use warded ontologies exist, all of which are, to the best of our knowledge, not publicly available. The latter is due to the fact that they have been developed for internal use by private companies. This further suggests that an open source alternative for reasoning over Warded ontologies, such as the one proposed in this thesis, would further foster the use and publication of OMQs based on warded ontologies.

10.4 Impact of the Rewriting on Reasoning

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

Datalog-based Reasoning Engines. We start by briefly discussing the different engines that we consider from the literature.

Vadalog. This is a reasoning engine that, given an OMQ (q, Σ) with $\Sigma \in$ WARDED and q of the form $Q(\bar{x}) \leftarrow R(\bar{x})$, and a database D, is able to compute the certain

[2]Other benchmarks mostly contain either OMQs with ontologies falling in much simpler fragments, e.g., UCQ-rewritable ontologies, for which more specialized rewriting algorithms can be employed, or OMQs falling in other fragments that are incomparable to WARDED, such as guarded TGDs.

answers of $\mathcal{O}$ over D, i.e., the set $\mathsf{ans}(\mathcal{O}, D)$ [17]. Hence, Vadalog supports warded ontologies, but focuses solely on the task of fact entailment, i.e., computing all facts $R(\bar{t}) \in \mathsf{chase}(D, \Sigma)$.[3] Moreover, although Vadalog focuses on warded ontologies, we point out that it does not employ any Datalog rewriting technique as the one we have developed in the thesis. Rather, to compute the set of all facts $R(\bar{t}) \in \mathsf{chase}(D, \Sigma)$, for a given database D, ontology $\Sigma \in \mathsf{WARDED}$, and predicate R, it first constructs the finite, isomorphically-closed portion, here denoted $\mathsf{ichase}(D, \Sigma)$, of the (possibly infinite) instance $\mathsf{chase}(D, \Sigma)$, and then collects all facts $R(\bar{t}) \in \mathsf{ichase}(D, \Sigma)$. It is shown in [16] that $\Sigma \in \mathsf{WARDED}$ implies that for every fact α, $\alpha \in \mathsf{chase}(D, \Sigma)$ iff $\alpha \in \mathsf{ichase}(D, \Sigma)$. The latter implies the correctness of the fact entailment procedure employed by Vadalog for warded ontologies. Note that this technique does not support computing certain answers over OMQs with warded ontologies having an arbitrary CQ. Obviously, since Vadalog supports fact entailment over warded ontologies, it is implicitly able to evaluate standard Datalog queries as well, since every ontology without existential variables (i.e., a Datalog program) is trivially warded.

VLog. The reasoning engine VLog, developed in [67], computes the certain answers of a given OMQ $\mathcal{O} = (q, \Sigma)$ over a database D by exploiting Theorem 1. That is, it first constructs the instance $\mathsf{chase}(D, \Sigma)$, it then computes all the answer tuples in $q(\mathsf{chase}(D, \Sigma))$. Clearly, since $\mathsf{chase}(D, \Sigma)$ can be infinite in general, the only inputs VLog supports are OMQs (q, Σ) and databases D such that $\mathsf{chase}(D, \Sigma)$ is finite; the latter implies that VLog also supports evaluating Datalog queries, since $\mathsf{chase}(D, \Sigma)$ is always finite, when Σ contains only TGDs without existential variables, regardless of the database D. The key technological aspect of VLog is the use of a column-based internal representation of the atoms produced during the construction of the instance $\mathsf{chase}(D, \Sigma)$. Such an encoding generally reduces the amount of memory used and makes the construction of atoms in $\mathsf{chase}(D, \Sigma)$ that are obtained via simple copy or permutation-based TGDs, such as $R(x, y, z) \rightarrow S(y, z, z)$, very fast.

DLV. DLV is a Datalog-based reasoning engine that has its roots in the logic programming field; an overview of the system can be found in [2]. In particular, DLV supports computing the so-called cautious answers of a given logic program P over a given database D. Although DLV does not natively support OMQs, we can exploit a well-known translation from an OMQ $\mathcal{O}$ to a logic program P, such that for every database D, $\mathsf{ans}(\mathcal{O}, D)$ coincides with the cautious answers of P over the database D. When given as input a logic program P obtained as the translation of an OMQ

[3] More specifically, Vadalog supports a slightly more general notion of fact entailment, where nulls can also appear as part of the output tuples. However, this goes beyond the scope of our work.

$\mathcal{O} = (q, \Sigma)$, and a database D, DLV will essentially construct a structure that coincides with the instance $\mathsf{chase}(D, \Sigma)$, and thus employs the same approach as the one of VLog. However, the optimization techniques it implements differ from the ones of VLog. Clearly, DLV supports evaluating standard Datalog queries as well.

DLV-E. DLV-E is an extension of DLV that natively supports computing the certain answers of a given OMQ over a database D, and has been developed in [55]. The ontology language that DLV-E supports is called shy. It is well-known that shy ontologies and warded ontologies are incomparable, i.e., there are shy ontologies that are not warded, and vice versa (e.g., see [12]). Certain answers over OMQs with shy ontologies can always be constructed by relying on a variation of the chase instance, called the *parsimonious chase with resumption*; DLV-E employs this variant for certain answering. However, it is implicit in [55] that when the input OMQ $\mathcal{O} = (q, \Sigma)$ and database D are such that $\mathsf{chase}(D, \Sigma)$ is finite, the parsimonious chase with resumption is finite, and can still be used to compute the certain answers $\mathsf{ans}(\mathcal{O}, D)$. Hence, DLV-E implicitly supports OMQs for which the chase instance is finite, and Datalog queries as well.

InteGraal. This is the recent reimplementation of the Datalog-based reasoning engine Graal, e.g. see [10]. Although the main focus of InteGraal is on OMQs with UCQ-rewritable ontologies, and thus performs certain answering using a UCQ rewriting algorithm in such cases, it also provides a chase-based reasoning module which exploits once again the construction of the chase instance to compute the certain answers, whenever the chase instance is finite. As a consequence, InteGraal also supports evaluating Datalog queries. In our experimental analysis, we employ the chase-based module of InteGraal.

Measuring Engine Efficiency. We remark that our goal is not a comparison of the above engines, as this would go beyond the scope of our work. Rather, we want to assess, for each reasoning engine we consider, whether the Datalog rewriting produced by our implementation of WardedRewrite can be evaluated efficiently. Moreover, this analysis should be performed by taking into account the capabilities and the limits of the engine itself. Clearly, in order to perform such an analysis, when taking each engine in isolation, measuring running times in absolute terms would not be very helpful. In order to provide useful insights that take into account the capabilities and the limits of each individual engine, we propose the following analysis.

For each engine $E \in \{\text{Vadalog}, \text{VLog}, \text{DLV}, \text{DLV-E}, \text{InteGraal}\}$, we consider each test scenario $S[i, j] = (D, \mathcal{O})$, with $i, j \in \{0, 20, 40, \ldots, 400\}$, and then measure the running times t_{certain} and t_{rew} required by E to compute (i) the *certain answers of* $\mathcal{O}$ *over* D,

i.e., the set $\mathsf{ans}(\mathcal{O}, D)$, and (ii) the set of answers $\mathcal{D}_{\mathcal{O}}(D)$ of the Datalog rewriting $\mathcal{D}_{\mathcal{O}}$ constructed by our implementation of WardedRewrite with input $\mathcal{O}$, respectively. Then, we report the *efficiency ratio* of E w.r.t. $S[i, j]$, i.e., the fraction

$$\mathcal{R}_{E,S[i,j]} = \frac{t_{\text{rew}}}{t_{\text{certain}}}.$$

As it is customary in evaluating reasoning over OMQs rewritings (e.g., see [19, 10]), we do not include in t_{rew} the time required to construct the Datalog rewriting, as this is a *database-independent* task that can be generally performed offline. The idea of the efficiency ratio $\mathcal{R}_{E,S[i,j]}$, for some engine E and scenario $S[i, j]$, is to allow to understand how efficient is E in evaluating our Datalog rewriting, using as reference the time required by E to compute the same set of answers by directly evaluating the original OMQ $\mathcal{O}$ over the database D, assuming $S[i, j] = (D, \mathcal{O})$. An efficiency ratio smaller than 1 implies that evaluating the Datalog rewriting is even faster than evaluating the original OMQ, and thus it is "relatively" efficient for E to evaluate the Datalog rewriting. On the other hand, an efficiency ratio larger than 1 indicates that, relative to how E would deal with the original OMQ itself, evaluating the Datalog rewriting is not that efficient. Finally, an efficiency ratio of exactly 1 means that evaluating the Datalog rewriting is as efficient as evaluating the original OMQ. Hence, for each engine E, the distribution of the efficiency ratios of E w.r.t. our pool of test scenarios will allow us to understand how effective is for E the evaluation of the Datalog rewriting produced by our implementation w.r.t. the evaluation of the original OMQ directly. This is precisely the kind of analysis we are going to perform in the rest of this section; but first we need to provide some clarifications.

A Common Ontology Language. All the Datalog-based reasoning engines we consider in our work support the evaluation of standard Datalog queries. However, their ability in computing the *certain answers* of a given OMQ $\mathcal{O}$ over a database D varies considerably. This is due to different factors, but the most crucial one being that, as discussed in chapter 1 and chapter 4, the problem $\mathsf{CQAns}[\mathsf{TGD}]$ is undecidable, and thus each engine can only support OMQs whose ontology falls in some decidable fragment. Among all the engines we are going to consider, only Vadalog supports warded TGDs, and as discussed above, its support is limited to fact entailment, i.e., OMQs (q, Σ) with q of the form $Q(\bar{x}) \leftarrow R(\bar{x})$. All the other engines we consider either support other ontology fragments that are incomparable to warded ontologies, and thus cannot, in general, compute the certain answers of an OMQ $\mathcal{O} = (q, \Sigma)$ over a database D, when $\Sigma \in \mathsf{WARDED}$, or rely on the finiteness of the chase instance. Hence, in order to per-

Figure 10.6: Efficiency ratio over test scenarios $S[i, j]$, with $i, j \in \{0, 20, 40, \ldots, 200\}$.

form our analysis using the efficiency ratio, we need to consider ontologies Σ falling in a fragment that is the intersection of the ontology fragments supported by each engine we consider. It turns out that all the test scenarios $S[i, j]$ with $i, j \in \{0, 20, 40, \ldots, 400\}$, that we generated with iWarded mention a OMQ (q, Σ) where Σ falls in a fragment supported by all the Datalog-based reasoning engines we mentioned above. In particular, all such ontologies Σ are such that, for every database D, $\mathsf{chase}(D, \Sigma)$ is finite. Since all engines we consider support this family of OMQs, it follows that every Datalog-based reasoning engine we consider is able to take as input a scenario $S[i, j] = (D, \mathcal{O})$, with $i, j \in \{0, 20, 40, \ldots, 400\}$, and effectively compute the set $\mathsf{ans}(\mathcal{O}, D)$. Hence, we can carry out our experimental analysis using the efficiency ratio, and the rest of this section is devoted to do exactly this.

Answering Question 2. We are now ready to perform the experimental analysis needed to answer our second experimental question. For each Datalog-based reasoning engine $E \in \{\text{Vadalog}, \text{VLog}, \text{DLV}, \text{DLV-E}, \text{InteGraal}\}$, and each pair of integers $i, j \in \{0, 20, 40, \ldots, 200\}$, we executed E with input the scenario $S[i, j] = (D, \mathcal{O})$, i.e., we asked E to compute the certain answers $\mathsf{ans}(\mathcal{O}, D)$ of $\mathcal{O}$ over 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 10.3, over the database D, and collected the time t_{rew} required by the engine. Finally, we computed the corresponding efficiency ratio $\mathcal{R}_{E,S[i,j]} = \dfrac{t_{\text{rew}}}{t_{\text{certain}}}$. Note that we only considered the test scenarios with `num_lrj` and `avg_recursion` in the range $\{0, 20, 40, \ldots, 200\}$, since the scenarios with higher values are quite demanding, with

most of the engines running out of memory. Finally, each time an engine was executed, it was given a timeout of 5 minutes.

The efficiency ratios $\mathcal{R}_{E,S[i,j]}$, for each $E \in \{\text{Vadalog}, \text{VLog}, \text{DLV}, \text{DLV-E}, \text{InteGraal}\}$, and each pair $i, j \in \{0, 20, 40, \ldots, 200\}$, are reported in Figure 10.6, expressed as percentages. For example, a value of 50% corresponds to an efficiency ratio equal to 0.5, meaning engine E evaluates the Datalog rewriting in half the time required by the same engine to evaluate the original OMQ. Figure 10.6 also groups the efficiency ratios by engine, where for each engine E, we report all the efficiency ratios $\mathcal{R}_{E,S[i,j]}$, for $i, j \in \{0, 20, 40, \ldots, 200\}$ in a violin plot. In each violin plot corresponding to an engine E, the bottom and the top borders of the inner box denote the first and third quartile, i.e., the efficiency ratio under which 25% and 75% of all the efficiency ratios $\mathcal{R}_{E,S[i,j]}$ with $i, j \in \{0, 20, 40, \ldots, 200\}$ occur, respectively. The line inside the box denotes the median efficiency ratio. Moreover, the area surrounding the box denotes the distribution (i.e., the kernel density estimation) of all the efficiency ratios $\mathcal{R}_{E,S[i,j]}$, with $i, j \in \{0, 20, 40, \ldots, 200\}$: the wider a certain portion of the area, the more test scenarios fall in that portion.

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 efficiently 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 constructed chase instance considerably. Surprisingly, although DLV-E supports OMQs natively, while DLV does not, DLV-E benefits the most from the use of the rewriting against the original OMQ. This might be due to the fact that DLV-E employs a more complex version of the chase, i.e., the parsimonious chase, which is more affected by the introduction of nulls.

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 the column-based representation employed, and thus it does not usually gain much in computing the certain answers via the Datalog rewriting, which, on the other hand, tends to contain more rules than the original ontology.

We conclude with the Vadalog engine, which appears to be in a middle ground between the two extreme cases discussed before. Indeed, we have that for roughly 80% of the test scenarios we considered, Vadalog efficiently evaluates the Datalog rewriting w.r.t. evaluating the original OMQ, with most of the efficiency ratios below 50%. This is somehow surprising, considering that Vadalog is specialized on fact entailment over warded ontologies. On the other hand, for the remaining 20% of the test scenarios we considered, the Datalog rewriting performs poorly, compared to the original OMQ, with extremely 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 construction of the chase instance, due to focusing only on atoms of the isomorphim-closed part of the chase. On the other hand, our rewriting algorithm WardedRewrite cannot take advantage of this property when constructing the Datalog rewriting, as it has been designed to work with OMQs over warded ontologies with *arbitrary* CQs. Nonetheless, we can conclude that the Datalog rewriting produced by our implementation of WardedRewrite can be efficiently evaluated by the Vadalog engine, in most cases.

Validation and Scalability Analysis. We conclude this section by employing once again the structural scenarios from [6] in order to validate our analysis performed above. Moreover, since such structural scenarios have been designed to stress test Datalog-based reasoning engines, we also use these scenarios to understand how the efficiency in evaluating the Datalog rewriting produced by WardedRewrite scales w.r.t. the size of the database. We recall that for each $\mathsf{X} \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}\}$, the scenario synth$\mathsf{X}$ = $(D, \mathcal{O})$ is such that D contains $10k$ facts of the form $P(i, \ldots, i)$, with $i \in [10k]$, for each each predicate P occurring in D. In order to assess the scalability of evaluating the Datalog rewriting produced by WardedRewrite w.r.t. the size of the database, we extend each scenario synth$\mathsf{X}$ into 5 scenarios, denoted synth$\mathsf{X}[10k]$, synth$\mathsf{X}[30k]$, synth$\mathsf{X}[50k]$, synth$\mathsf{X}[70k]$, and synth$\mathsf{X}[90k]$, where synth$\mathsf{X}[s] = (D_s, \mathcal{O})$ is the scenario where the database D_s contains s facts of the form $P(i, \ldots, i)$, with $i \in [s]$, for each predicate P occurring in D; clearly, the scenario synth$\mathsf{X}[10k]$ coincides with the scenario synth$\mathsf{X}$.

For each Datalog-based reasoning engine $E \in \{\text{Vadalog}, \text{VLog}, \text{DLV}, \text{DLV-E}, \text{InteGraal}\}$, each $\mathsf{X} \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}\}$, and for each $s \in \{10k, 30k, 50k, 70k, 90k\}$, assuming

synth$\mathsf{X}[s] = (D_s, \mathcal{O})$, we let E compute the certain answers of $\mathcal{O}$ over D_s[4], as well as compute the answers of the Datalog rewriting $\mathcal{D}_\mathcal{O}$, produced by WardedRewrite, over D_s. We collected the runtime of the above two tasks, and computed the corresponding efficiency ratio $\mathcal{R}_{E,\text{synth}\mathsf{X}[s]}$; in both cases, we set a timeout of 5 minutes. All the efficiency ratios are reported in Table 10.1, where a value of "O.M.E." (resp., "R.M.E.", "B.M.E.") in a cell corresponding to a certain scenario synth$\mathsf{X}[s]$ and a Datalog-based reasoning engine E means that E ran out of memory when evaluating the original OMQ (resp., the Datalog rewriting, or both).

We can observe that regarding DLV, DLV-E, and InteGraal, the results of Table 10.1 are coherent with our conclusions made on the test scenarios, for the same engines. Indeed, when the execution does not run out of memory, almost all the efficiency ratios are below 100%, while in case of running out of memory, this happens either for the OMQ evaluation only, or for both the OMQ and the rewriting, but never for the rewriting alone. Interestingly, regarding the scalability of evaluating the Datalog rewriting, we observe that for DLV, DLV-E, and InteGraal, the efficiency ratio improves as the database size increases, and thus evaluating the Datalog rewriting is more and more effective as the database becomes larger. This is further confirmation of the fact that the main reason for such low efficiency ratios is the difference in size between the chase instance constructed using the original ontology, which tends to contain much more atoms as the database increases, and the one constructed using the TGDs of the Datalog rewriting, which can only mention constants of the database.

Moving to VLog, we can see that for half of the scenarios, VLog evaluates the original OMQ faster than evaluating the Datalog rewriting produced by our implementation of WardedRewrite, which is in line with our analysis on the test scenarios. The increase in the database size has a much milder effect on the efficiency of evaluating the Datalog rewriting, confirming that VLog is less affected by the size of the chase instance being built, and thus it is likely to not benefit from the use of our Datalog rewriting, even for large databases.

Finally, Vadalog again exhibits a less regular behavior, where in the cases in which the efficiency ratio is below 100%, this is usually quite low. On the other hand, for efficiency ratios above 100%, we obtain quite varied results, with values even up to 200%. Moreover, there are even a few cases where the evaluation of the Datalog rewriting runs out of memory, while the one of the original OMQ did not. This is the only engine where the latter happens, and further confirms that in some circumstances, Vadalog is

[4]We verified that for all structural scenarios, their ontology guarantees the finiteness of the chase instance, regardless of the given database, and thus we can safely compute the certain answers.

able to exploit the isomorphic-closed part of the chase instance to its advantage. In fact, as the database size increases, we do not observe any consistent increase (or decrease) of the efficiency ratio, as the efficiency of evaluating the original OMQ highly depends on the internal structure (i.e., the isomorphic-closed part) of the chase instance being constructed by Vadalog.

Table 10.1: Efficiency ratio over the scenarios synth$\mathsf{X}[s]$, with $\mathsf{X} \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}\}$, and $s \in \{10k, 30k, 50k, 70k, 90k\}$. A value of "O.M.E." (resp., "R.M.E.", "B.M.E.") in a cell corresponding to a certain scenario synth$\mathsf{X}[s]$ and a Datalog-based reasoning engine E means that E exceeded the system memory, when evaluating the original OMQ (resp., the Datalog rewriting, or both).

Scenario	**Vadalog**	**VLog**	**DLV**	**DLV-E**	**InteGraal**
synthA[10k]	14.29%	88.28%	30.59%	25.14%	47.87%
synthA[30k]	38.46%	119.08%	31.33%	19.88%	43.54%
synthA[50k]	38.10%	109.41%	35.47%	19.77%	40.32%
synthA[70k]	37.50%	85.02%	31.93%	19.57%	29.21%
synthA[90k]	45.16%	97.87%	28.83%	19.59%	O.M.E.
synthB[10k]	5.88%	115.20%	86.35%	54.46%	76.53%
synthB[30k]	200%	142.17%	85.71%	53.82%	77.90%
synthB[50k]	R.M.E.	135.24%	84.32%	54.52%	70.75%
synthB[70k]	B.M.E.	123.77%	83.22%	53.75%	O.M.E.
synthB[90k]	B.M.E.	134.21%	83.40%	53.69%	B.M.E.
synthC[10k]	5.56%	92.11%	46.44%	34.05%	54.51%
synthC[30k]	41.03%	113.30%	45.47%	32.33%	48.85%
synthC[50k]	52.46%	83.03%	44.18%	32.55%	45.37%
synthC[70k]	57.14%	86.21%	42.74%	32.18%	O.M.E.
synthC[90k]	O.M.E.	84.44%	43.44%	32.15%	B.M.E.
synthD[10k]	175%	94.42%	78.32%	62.63%	86.90%
synthD[30k]	133.33%	120.31%	77.90%	59.81%	82.57%
synthD[50k]	136.84%	112.38%	75.99%	60.50%	78.41%
synthD[70k]	140%	89.33%	75.18%	61.28%	68.93%
synthD[90k]	106.67%	104.59%	75.95%	61.87%	62.91%
synthE[10k]	20%	91.46%	23.87%	17.60%	36.44%
synthE[30k]	11.76%	118.32%	21.32%	15.03%	23.60%
synthE[50k]	4.17%	95.51%	19.99%	15.11%	21.55%
synthE[70k]	15.15%	85.60%	18.45%	14.67%	O.M.E.
synthE[90k]	13.33%	83.08%	18.57%	14.74%	O.M.E.
synthF[10k]	14%	90.46%	35.82%	27.41%	42.25%
synthF[30k]	8.70%	120.28%	32.99%	24.01%	29.91%
synthF[50k]	10.53%	99.05%	31.53%	23.71%	28.13%
synthF[70k]	2.04%	85.60%	29.78%	23.60%	27.21%
synthF[90k]	1.56%	84.61%	30.45%	23.26%	27%
synthG[10k]	100%	101.46%	73.28%	41.86%	67.49%
synthG[30k]	152.63%	129.27%	81.90%	38.35%	62.80%
synthG[50k]	151.61%	107.88%	69.41%	39.74%	59.80%
synthG[70k]	153.49%	100.91%	67.81%	38.09%	O.M.E.
synthG[90k]	129.63%	99.51%	65.20%	38.71%	O.M.E.
synthH[10k]	110.53%	117.36%	109.66%	62.02%	88.24%
synthH[30k]	142.86%	149.70%	107.13%	62.11%	78.58%
synthH[50k]	159.26%	129%	106.46%	61.83%	80.68%
synthH[70k]	R.M.E.	126.69%	105.63%	62.07%	O.M.E.
synthH[90k]	1.22%	130.71%	106.90%	61.74%	B.M.E.

Chapter 11

Conclusions

In this chapter we highlight the most significant findings of the study, its applicability, its limitations, and the future lines of research.

Main Findings. Regarding our findings, our work is the first to provide a Datalog-rewriting algorithm, dubbed WardedRewrite, for ontology-mediated queries over warded ontologies which is amenable to practical implementations. We have shown that the algorithm is able, given an ontology-mediated query $\mathcal{O}$ over a warded ontology, to construct a Datalog rewriting $\mathcal{D}_{\mathcal{O}}$ of $\mathcal{O}$, i.e., $\mathcal{D}_{\mathcal{O}}$ is 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. Moreover, we provided a concrete implementation of the algorithm, and an experimental evaluation, geared towards answering key questions related to the applicability of the algorithm in practice. Our analysis revealed that our implementation is quite efficient in producing Datalog rewritings which, in turn, can be effectively exploited for query answering purposes by means of off-the-shelf Datalog-based engines. Our experimental analysis has revealed that the technique of Datalog rewritability is an effective approach for query answering in the context of warded ontologies, as we have shown that both the construction of the Datalog rewriting, as well its evaluation via existing Datalog-based engines are quite promising, performance-wise.

Applicability. Regarding the applicability of our approach in practice, our implementation of the Datalog rewriting algorithm WardedRewrite, proposed in this thesis, has shown to perform reasonably well in the stress test scenarios which better resemble real-world settings. Moreover, most of the existing engines are able, within the limits of their capabilities, to efficiently compute certain answers via the Datalog rewriting. Thus, we believe that ontology-mediated query answering with warded ontologies can indeed be enabled by exploiting off-the-shelf tools, which, in our opinion, represents a

promising alternative to ad-hoc and specialized implementations, which might not have the same level of maturity of implementations of more standard reasoning techniques.

Limitations. Regarding limitations of our implementation, we found that the relevant bottlenecks are related to the recursion depth of the TGDs of the input ontology. Thus, we believe further improvements of the implementation should be towards mitigating the dependency on this parameter.

Future Work. Regarding possible venues for future work, we would like to verify whether the use of parallelism can improve the efficiency of our implementation of WardedRewrite. This would exploit the fact that all intermediate queries produced by the algorithm can be processed independently from each other, thanks to the optimal decomposition employed by the algorithm. This should also help improve performance for extremely demanding settings, with deep recursive chains of TGDs. Some preliminary results on this parallelization effort were published in [68].

Another interesting direction for future work is the incremental maintenance of the produced Datalog rewriting, in settings where OMQs change frequently, e.g., during the initial definition of the ontology. This would allow ontology designers to iteratively test the validity of the ontology by being able to perform reasoning without major delays due to constructing the rewriting from scratch, each time.

Bibliography

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

[2] 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.

[3] Shqiponja Ahmetaj. Rewriting approaches for ontology-mediated query answering. *KI - Künstliche Intelligenz*, 34, 06 2020.

[4] Shqiponja Ahmetaj, Magdalena Ortiz, and Mantas Simkus. Rewriting guarded existential rules into small datalog programs. In *ICDT*, volume 98, pages 4:1–4:24, 2018.

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

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

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

[8] 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.

[9] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.

[10] 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.

[11] 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.

[12] Teodoro Baldazzi, Luigi Bellomarini, Marco Favorito, and Emanuel Sallinger. On the relationship between shy and warded datalog+/-. In *KR*, 2022.

[13] Teodoro Baldazzi, Luigi Bellomarini, Marco Favorito, and Emanuel Sallinger. Ontological reasoning over shy and warded datalog+/- for streaming-based architectures. In *PADL*, volume 14512, pages 169–185, 2024.

[14] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.

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

[16] 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.

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

[18] Davide Benedetto, Marco Calautti, Hebatalla Hammad, Emanuel Sallinger, and Adriano Vlad-Starrabba. A datalog rewriting algorithm for warded ontologies. (in press). In *In the 34th International Joint Conference on Artificial Intelligence*, 2025.

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

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

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

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

[23] Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Checking termination of logic programs with function symbols through linear constraints. In *RuleML*, volume 8620, pages 97–111. Springer, 2014.

[24] Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Logic program termination analysis using atom sizes. In Qiang Yang and Michael J. Wooldridge, editors, *IJCAI*, pages 2833–2839, 2015.

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

[26] Marco Calautti, Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Checking termination of bottom-up evaluation of logic programs with function symbols. *Theory Pract. Log. Program.*, 15(6):854–889, 2015.

[27] Marco Calautti, Sergio Greco, and Irina Trubitsyna. Detecting decidable classes of finitely ground logic programs with function symbols. In *PPDB*, pages 239–250, 2013.

[28] Marco Calautti, Mostafa Milani, and Andreas Pieris. Semi-oblivious chase termination for linear existential rules: An experimental study. *VLDB*, 16(11):2858–2870, 2023.

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

[30] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: query answering under expressive relational constraints. *J. Artif. Int. Res.*, 48(1):115–174, October 2013.

[31] Andrea Cali, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics*, January 2012.

[32] Andrea Calì, Georg Gottlob, and Andreas Pieris. Query answering under non-guarded rules in datalog+/-. In *Proceedings of the RR 2010*, page 1–17, 2010.

[33] Andrea Cali', Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence*, 193:87–128, 2012.

[34] 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.

[35] 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.

[36] B. Cuenca Grau, I. Horrocks, M. Krötzsch, C. Kupke, D. Magka, B. Motik, and Z. Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *Journal of Artificial Intelligence Research*, 47:741–808, August 2013.

[37] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[38] Alin Deutsch, Alan Nash, and Jeff Remmel. The chase revisited. In *PODS*, page 149–158, 2008.

[39] 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.

[40] GEORG GOTTLOB, MARCO MANNA, and CINZIA MARTE. Dyadic existential rules. *Theory and Practice of Logic Programming*, 24(2):227–249, 2024.

[41] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Query rewriting and optimization for ontological databases. *ACM Transactions on Database Systems*, 2014.

[42] Georg Gottlob, Giorgio Orsi, Andreas Pieris, and Mantas Simkus. Datalog and its extensions for semantic web databases. *Reasoning Web*, 2012.

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

[44] Georg Gottlob, Sebastian Rudolph, and Mantas Simkus. Expressiveness of guarded existential rule languages. In *PODS*, pages 27–38. ACM, 2014.

[45] Georg Gottlob, Sebastian Rudolph, and Mantas Simkus. Expressiveness of guarded existential rule languages. In *PODS*, page 27–38, 2014.

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

[47] Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *J. Artif. Int. Res.*, 47(1):741–808, May 2013.

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

[49] Philipp Hanisch and Markus Krötzsch. Chase termination beyond polynomial time. *Proceedings ACM Manag. Data*, 2(2):93, 2024.

[50] André Hernich and Nicole Schweikardt. Cwa-solutions for data exchange settings with target dependencies. In *PODS*, pages 113–122, 2007.

[51] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In *CAV*, volume 9780, pages 422–430, 2016.

[52] 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.

[53] Markus Krötzsch and Sebastian Rudolph. Extending decidable existential rules by joining acyclicity and guardedness. In *IJCAI*, pages 963–968, 01 2011.

[54] Michel Leclère, Marie-Laure, Michaël Thomazo, and Federico Ulliana. A single approach to decide chase termination on linear existential rules. In *ICDT*, pages 18:1–18:19, 2019.

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

[56] 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.

[57] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, dec 1979.

[58] Bruno Marnette. Generalized schema-mappings: from termination to tractability. In *PODS*, pages 13–22, 2009.

[59] Michael Meier, Michael Schmidt, and Georg Lausen. On chase termination beyond stratification. *PVLDB*, 2(1):970–981, 2009.

[60] Hector Perez-Urbina, Boris Motik, and Ian Horrocks. A comparison of query rewriting techniques for dl-lite. In *DL*, 2009.

[61] 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.

[62] Riccardo Rosati. On conjunctive query answering in $\mathcal{EL}$. In *DL*, 2007.

[63] Yehoshua Sagiv. Chapter 17 - optimizing datalog programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, 1988.

[64] Oded Shmueli. Equivalence of datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, 1993.

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

[66] Giorgio Stefanoni, Boris Motik, and Ian Horrocks. Small datalog query rewritings for $\mathcal{EL}$. *CEUR Workshop Proceedings*, 846, 01 2012.

[67] 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.

[68] Heba M. Wagih and Marco Calautti. Enhancing ontological query-rewriting via parallelization. In *SEBD*, volume 3478, pages 401–409, 2023.

[69] Zhe Wang, Peng Xiao, Kewen Wang, Zhiqiang Zhuang, and Hai Wan. Efficient datalog rewriting for query answering in tgd ontologies. *IEEE Transactions on Knowledge and Data Engineering*, 35(3):2515–2528, 2023.

[70] Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalayci, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. The virtual knowledge graph system ontop. In *ISWC*, pages 259–277, 2020.